Toward a Catalogue of Architectural Bad Smells

Joshua Garcia, Daniel Popescu, George Edwards and Nenad Medvidovic

University of Southern California, Los Angeles, CA, USA
{joshuaga,dpopescu,gedwards,neno}@usc.edu

Abstract. An *architectural bad smell* is a commonly (although not always intentionally) used set of architectural design decisions that negatively impacts system lifecycle properties, such as understandability, testability, extensibility, and reusability. In our previous short paper, we introduced the notion of architectural bad smells and outlined a few common smells. In this paper, we significantly expand upon that work. In particular, we describe in detail four representative architectural smells that emerged from reverse-engineering and re-engineering two large industrial systems and from our search through case studies in research literature. For each of the four architectural smells, we provide illustrative examples and demonstrate the smell's impact on system lifecycle properties. Our experiences indicate the need to identify and catalog architectural smells so that software architects can discover and eliminate them from system designs.

1 Introduction

As the cost of developing software increases, so does the incentive to evolve and adapt existing systems to meet new requirements, rather than building entirely new systems. Today, it is not uncommon for a software application family to be maintained and upgraded over a span of five years, ten years, or longer. However, in order to successfully modify a legacy application to support new functionality, run on new platforms, or integrate with new systems, evolution must be carefully managed and executed. Frequently, it is necessary to *refactor* [1], or restructure the design of a system, so that new requirements can be supported in an efficient and reliable manner.

The most commonly used way to determine how to refactor is to identify code *bad smells* [2] [1]. Code smells are implementation structures that negatively affect system *lifecycle properties*, such as understandability, testability, extensibility, and reusability; that is, code smells ultimately result in maintainability problems. Common examples of code smells include very long parameter lists and duplicated code (i.e., clones). Code smells are defined in terms of *implementation-level* constructs, such as methods, classes, parameters, and statements. Consequently, refactoring methods to correct code smells also operate at the implementation level (e.g., moving a method from one class to another, adding a new class, or altering the class inheritance hierarchy).

While detection and correction of code smells is one way to improve system maintainability, some maintainability issues originate from poor use of *software* *architecture-level* abstractions — components, connectors, styles, and so on — rather than implementation constructs. In our previous work [3], we introduced the notion of *architectural bad smells* and identified four representative smells. *Architectural bad smells* are combinations of architectural constructs that induce reductions in system maintainability. Architectural smells are analogous to code smells because they both represent common "solutions" that are not necessarily faulty or errant, but still negatively impact software quality. In this paper, we expand upon the four smells identified in our previous work by describing them in detail and illustrating their occurence in case studies from research literature and our own architectural recovery [4] [5] and industrial maintenance efforts.

The remainder of this paper is organized as follows. Section 2 explains the characteristics and significance of architectural smells. Section 3 summarizes research efforts in related topics. Section 4 introduces two long-term software maintenance efforts on industrial systems and case studies from research literature that we use to illustrate our four representative architectural smells. Section 5 describes our four architectural smells in detail, and illustrates the impact of each smell through concrete examples drawn from the systems mentioned in Section 4. Finally, Section 6 provides closing discussion and insights.

2 Definition

In this section, we define what constitutes an architectural smell and discuss the important properties of architectural smells.

We define a software system's *architecture* as "the set of principal design decisions governing a system" [6]. The system stakeholders determine which aspects are deemed to be "principal." In practice, this usually includes (but is not limited to) how the system is organized into subsystems and components, how functionality is allocated to components, and how components interact with each other and their execution environment. The term *architectural smell* was originally defined in [7] as a commonly used architectural decision that negatively impacts system quality. Architectural smells may be caused by applying a design solution in an inappropriate context, mixing combinations of design abstractions that have undesirable emergent behaviors, or applying design abstractions at the wrong level of granularity. Architectural smells most directly affect lifecycle properties, such as understandability, testability, extensibility, and reusability, but they may have harmful side effects on other quality properties like performance and reliability. Architectural smells are remedied by altering the internal structure of the system and the behaviors of internal system elements without changing the external behavior of the system. We extend, in three ways, the definition of architectural smell found in [7].

Our first extension to the definition is our explicit capture of architectural smells as design *instances* that are independent from the engineering *processes* that created the design. That is, human organizations and processes are orthogonal to the definition and impact of a specific architectural smell. In practical terms, this means that the detection and correction of architectural smells is not dependent on an understanding of the history of a software system. For example, an independent analyst should be able to audit a documented architecture and indicate possible smells without knowing about the development organization, management, or processes.

For our second extension to the definition, we do not differentiate between architectural smells that are part of an *intended* design (e.g., a set of UML specifications for a system that has not yet been built) as opposed to an *implemented* design (e.g., the implicit architecture of an executing system) because architectural smells can appear in both designs.

For our last extension, we attempt to facilitate the detection of architectural smells through specific, concrete definitions captured in terms of standard architectural building blocks — components, connectors, interfaces, and configurations. Increasingly, software engineers reason about their systems in terms of these concepts [8, 6], so in order to be readily applicable and maximally effective, our architectural smell definitions similarly utilize these abstractions (see Section 5). The definition in [7] does not utilize explicit architectural interfaces or first-class connectors in their smells.

In many contexts, a design that exhibits a smell will be justified by other concerns. Architectural smells always involve a trade-off between different properties, and the system architects must determine whether action to correct the smell will result in a net benefit. Furthermore, refactoring to reduce or eliminate an architectural smell may involve risk and almost always requires investment of developer effort.

3 Related Work

In this section, we provide an overview of four topics that are directly related to architectural smells: code smells, architectural antipatterns, architectural mismatches, and defects.

The term *code smells* was introduced by Beck and Fowler [2] for code structures that intuitively appear as bad solutions and indicate possibilities for code improvements. For most code smells, refactoring solutions that result in higher quality software are known. Although bad smells were originally based on subjective intuitions of bad code practice, recent work has developed ways to detect code smells based on metrics [9] and has investigated the impact of bad smells using historical information [10]. As noted in Section 1, code smells only apply to implementation issues (e.g., a class with too many or too few methods), and do not guide software architects towards higher-level design improvements.

Closely related to code smells are *antipatterns* [11]. An antipattern describes a recurring situation that has a negative impact on a software project. Antipatterns include wide-ranging concerns related to project management, architecture, and development, and generally indicate organizational and process difficulties (e.g., design-by-committee) rather than design problems. The general definition of antipatterns allows both code and architectural smells to be classified as antipatterns. However, antipatterns that specifically pertain to architectural issues typically capture the causes and characteristics of poor design from a system-wide viewpoint (e.g., stove-piped systems). Architectural smells, on the other hand, focus on design problems that are independent from process and organizational concerns, and concretely address the internal structure and behavior of systems.

Another concept similar to architectural smells is *architectural mismatch* [12]. Architectural mismatch is the set of conflicting assumptions architectural elements may make about the system in which they are used. In turn, these conflicting assumptions may prevent the integration of an architectural element into a system. Work conducted in [13] and [14] has resulted in a set of conceptual features used to define architectural designs in order to detect architectural mismatch. While instructive to our work, architectural mismatch research has focused heavily on the functional properties of a system without considering the effects on lifecycle properties.

Finally, *defects* are similar to architectural smells. A defect is a manifestation of an error in a system [15]. An error is a mental mistake made by a designer or developer [15]. In other words, a defect is an error that is manifested in either a requirements, design, or implemented system that is undesired or unintended [16]. Defects are never desirable in a software system, while smells may be desirable if a designer or developer prefers the reduction in certain lifecycle properties for a gain in other properties, such as performance.

4 Systems Under Discussion

Our experience with two long-term software projects brought us to the realization that some commonly-used design structures adversely affect system maintainability. In this section, we introduce these projects by summarizing their context and objectives. Later in the paper, we utilize specific examples from these projects to illustrate the impact of architectural bad smells.

Maintenance of large-scale software systems includes both architectural *recovery* and *refactoring* activities. Architectural recovery is necessary when a system's conceptual architecture is unknown or undocumented. Architectural refactoring is required when a system's architecture is determined to be unsatisfactory and must be altered. We discovered architectural bad smells during both an architectural recovery effort (summarized in Section 4.1) and an architectural refactoring effort (summarized in Section 4.2). To substantiate our observations, we found further examples of architectural bad smells that appear in recovery and refactoring efforts published in the research literature.

4.1 Grid Architecture Recovery

An extensive study of grid system [17] implementations contributed to our collection and insights of architectural smells. Grid technologies allow heterogeneous organizations to solve complex problems using shared computing resources. Four years ago, we conducted a pilot study [18] in which we extracted and studied the architecture of five widely-used grid technologies and compared their architectures to the published grid reference architecture [17]. We subsequently completed a more comprehensive grid architecture recovery project and recently published a report [5] on the architectures of eighteen grid technologies, including a new reference architecture for the grid. The examined grid systems were developed in C, C++, or Java and contained up to 2.2 million SLOC. Many of these systems included similar design elements that have a negative effect on quality properties.

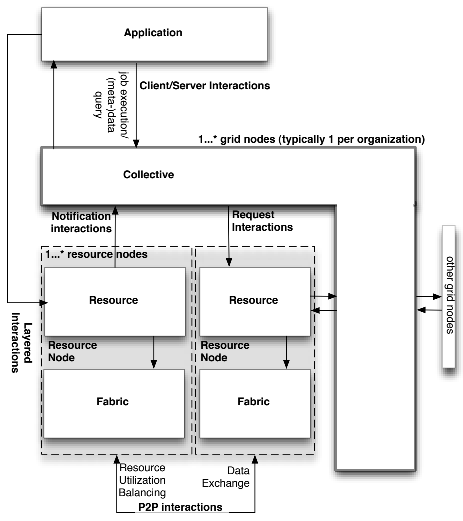


Fig. 1. Structural View of the Grid Reference Architecture

Figure 1 shows the identified reference architecture for the grid. A grid system is composed of four subsystems: *Application*, *Collective*, *Resource*, and *Fabric*. Each subsystem is usually instantiated multiple times. An Application can be any client that needs grid services and is able to use an API that interfaces with Collective or Resource components. The components in the Collective subsystem are used to orchestrate and distribute data and grid jobs to the various available resources in a manner consistent with the security and trust policies specified by the institutions within a grid system (i.e., the virtual organization). The Resource subsystem contains components that perform individual operations required by a grid system by leveraging available lower-level Fabric components. Fabric components offer access capabilities to computational and data resources on an individual node (e.g., access to file-system operations). Each subsystem uses different interaction mechanisms to communicate with other subsystems, as noted in Figure 1. The interaction mechanisms are described in [5].

4.2 MIDAS Architecture Refactoring

In collaboration with an industrial partner, for the last three years we have been developing a lightweight middleware platform, called MIDAS, for distributed sensor applications [19] [20]. Over ten software engineers in three geographically distributed locations contributed to MIDAS in multiple development cycles to address changing and growing requirements. In its current version, MIDAS implements many high-level services (e.g., transparent fault-tolerance through component replication) that were not anticipated at the commencement of the project. Additionally, MIDAS was ported to a new operating system (Linux) and programming language (C++), and capabilities tailored for a new domain (mobile robotics) were added. As a consequence, the MIDAS architecture was forced to evolve in unanticipated ways, and the system's complexity grew substantially. In its current version, the MIDAS middleware platform consists of approximately 100 KSLOC in C++ and Java. The iterative development of MIDAS eventually caused several architectural elements to lose conceptual coherence (e.g., by providing multiple services). As a consequence, we recently spent three person-months refactoring the system to achieve better modularity, understandability, and adaptability. While performing the refactoring, we again encountered architectural structures that negatively affected system lifecycle properties.

Figure 2 shows a layered view of the MIDAS middleware platform. The bottom of the MIDAS architecture is a virtual machine layer that allows the middleware to be deployed on heterogeneous OS and hardware platforms efficiently. The host abstraction facilities provided by the virtual machine are leveraged by the middleware's architectural constructs at the layer above. These architectural constructs enable a software organization to directly map its system's architecture to the system's implementation. Finally, these constructs are used to implement advanced distributed services such as fault-tolerance and resource discovery.

4.3 Studies from Research Literature

Given the above experiences, we examined the work in architectural recovery and refactoring published in research literature [4] [21] [22] [23], which helped us to understand architectural design challenges and common bad smells. In this paper, we refer to examples from a case study that extracted and analyzed the architecture of Linux [4]. In this study, Bowman et al. created a conceptual architecture of the Linux kernel based on available documentation and then extracted the architectural dependencies within the kernel source code (800 KSLOC). They concluded that the kernel contained a number of design problems, such as unnecessary and unintended dependencies.

5 Architectural Smells

This section describes four architectural smells in detail. We define each architectural smell in terms of participating architectural elements — components, connectors, interfaces, and configurations. Components are computational elements

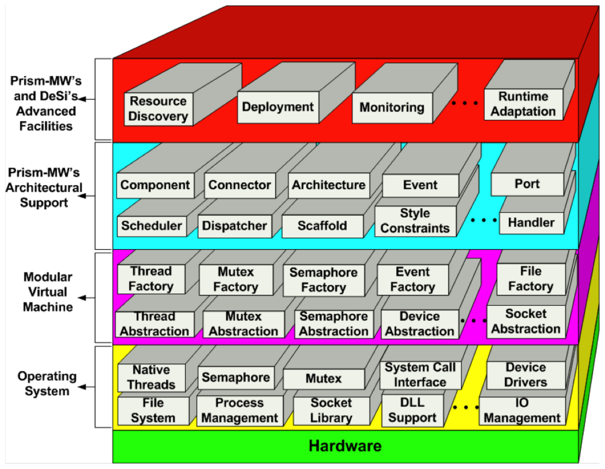


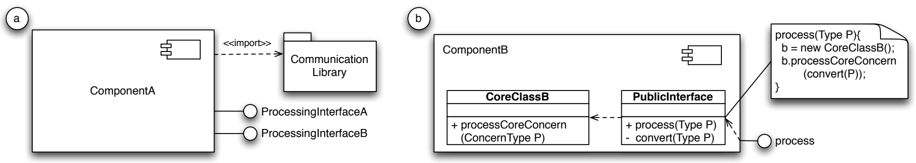
Fig. 2. System Stack Layers in MIDAS

that implement application functionality in a software system [24]. Connectors provide application-independent interaction facilities, such as transfer of data and control [25]. Interfaces are the interaction points between components and connectors. Finally, configurations represent the set of associations and relationships between components and/or connectors. We provide a generic schematic view of each smell captured in one or more UML diagrams. Architects can use diagrams such as these to inspect their own designs for architectural smells.

5.1 Connector Envy

Description. Components with *Connector Envy* encompass extensive interaction-related functionality that should be delegated to a connector. Connectors provide the following types of interaction services: communication, coordination, conversion, and facilitation [25]. Communication concerns the transfer of data (e.g., messages, computational results, etc.) between architectural elements. Coordination concerns the transfer of control (e.g., the passing of thread execution) between architectural elements. Conversion is concerned with the translation of differing interaction services between architectural elements (e.g., conversion of data formats, types, protocols, etc). Facilitation describes the mediation, optimization, and streamlining of interaction (e.g., load balancing, monitoring, and fault tolerance). Components that extensively utilize functionality from one or more of these four categories suffer from the Connector Envy smell.

Figure 3a shows a schematic view of one Connector Envy smell, where *ComponentA* implements communication and facilitation services. *ComponentA* imports a communication library, which implies that it manages the low-level networking facilities used to implement remote communication. The naming, delivery and routing services handled by remote communication are a type of facilitation service.



```
process(Type P){
  b = new CoreClassB();
  b.processCoreConcern
        (convert(P));
}
```

Fig. 3. The top diagram depicts Connector Envy involving communication and facilitation services. The bottom diagram shows Connector Envy involving a conversion service.

Figure 3b depicts another Connector Envy smell, where *ComponentB* performs a conversion as part of its processing. The interface of *ComponentB* called *process* is implemented by the *PublicInterface* class of *ComponentB*. *PublicInterface* implements its process method by calling a conversion method that transforms a parameter of type *Type* into a *ConcernType*.

Quality Impact and Trade-offs. Coupling connector capabilities with component functionality reduces reusability, understandability, and testability. Reusability is reduced by the creation of dependencies between interaction services and application-specific services, which make it difficult to reuse either type of service without including the other. The overall understandability of the component decreases because disparate concerns are commingled. Lastly, testability is affected by Connector Envy because application functionality and interaction functionality cannot be separately tested. If a test fails, either the application logic or the interaction mechanism could be the source of the error.

As an example, consider a *MapDisplay* component that draws a map of the route followed by a robot through its environment. The component expects position data to arrive as Cartesian coordinates and converts that data to a screen coordinate system that uses only positive x and y values. The *MapDisplay* suffers from Connector Envy because it performs conversion of data formats between the robot controller and the user interface. If the *MapDisplay* is used in a new, simulated robot whose controller represents the world in screen coordinates, the conversion mechanism becomes superfluous, yet the *MapDisplay* cannot be reused intact without it. Errors in the displayed location of the robot could arise from incorrect data conversion or some other part of the *MapDisplay*, yet the encapsulation of the adapter within the *MapDisplay* makes it difficult to test and verify in isolation.

The Connector Envy smell may be acceptable when performance is of higher priority than maintainability. More specifically, explicitly separating the interaction mechanism from the application-specific code creates an extra level of indirection. In some cases, it may also require the creation of additional threads or processes. Highly resource-constrained applications that use simple interaction mechanisms without rich semantics may benefit from retaining this smell. However, making such a trade-off simply for efficiency reasons, without considering the maintainability implications of the smell, can have a disastrous cumulative effect as multiple incompatible connector types are placed within multiple components that are used in the same system.

Example from Industrial Systems. The Gfarm Filesystem Daemon (*gfsd*) from a grid technology called Grid Datafarm [26] is a concrete example of a component with Connector Envy that follows the form described in Figure 3. The *gfsd* is a Resource component and runs on a Resource node as depicted in Figure 1. The *gfsd* imports a library that is used to build the lightweight remote procedure call (RPC) mechanism within the *gfsd*. This built-in RPC mechanism provides no interfaces to other components and, thus, is used solely by the *gfsd*. While the general schematic in Figure 3 shows only an instance of communication and facilitation, this instance of the smell also introduces coordination services by implementing a procedure call mechanism. The interfaces of the *gfsd* provide remote file operations, file replication, user authentication and node resource status monitoring. These interfaces and the *gfsd*'s RPC mechanism enable the notification, request, and P2P interactions shown in Figure 1 that occur across Resource nodes in Grid Datafarm.

Reusability, modifiability, and understandability are adversely affected by the Connector Envy smell in the *gfsd*. The reusability effects of Connector Envy can be seen in a situation where a new Resource component, called Gfarm workflow system daemon *(gwsd)*, that provides workflow-based services is added to Grid Datafarm. The RPC mechanism within the *gfsd* is built without interfaces that can be made available to other components, hence the RPC mechanism cannot be used with the *gwsd*. Understandability is reduced by the unnecessary dependencies between the *gfsd*'s application-specific functionality (e.g., file replication, local file operations, etc.) and RPC mechanism. The combination of application-specific functionality and interaction mechanisms throughout the functions of the *gfsd* enlarge the component in terms of function size, number of functions, and shared variables. Both modifiability and understandability are adversely affected by having the overwhelming majority of the *gfsd*'s functions involve the use or construction of Grid Datafarm's RPC mechanism.

It is possible that since grid technologies need to be efficient, the creators of Grid Datafarm may have intentionally built a *gfsd* with Connector Envy in order to avoid the performance effects of the indirection required for a fully separated connector. Another fact to consider is that Grid Datafarm has been in use for at least seven years and has undergone a significant number of updates that have expanded the *gfsd*'s functionality. This has likely resulted in further commingling of connector-functionality with application-specific functionality.

5.2 Scattered Parasitic Functionality

Description. *Scattered Parasitic Functionality* describes a system where multiple components are responsible for realizing the same high-level concern and, additionally, some of those components are responsible for orthogonal concerns. This smell violates the principle of separation of concerns in two ways. First, this smell scatters a single concern across multiple components. Secondly, at least one

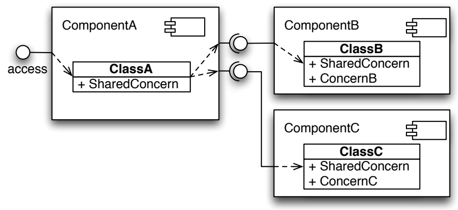


Fig. 4. The Scattered Parasitic Functionality occurring across three components.

component addresses multiple orthogonal concerns. In other words, the scattered concern infects a component with another orthogonal concern, akin to a parasite. Combining all components involved creates a large component that encompasses orthogonal concerns. Scattered Parasitic Functionality may be caused by cross-cutting concerns that are not addressed properly. Note that, while similar on the surface, this architectural smell differs from the *shotgun surgery* code smell [2] because the code smell is agnostic to orthogonal concerns.

Figure 4 depicts three components that are each responsible for the same high-level concern called *SharedConcern*, while *ComponentB* and *ComponentC* are responsible for orthogonal concerns. The three components in Figure 4 cannot be combined without creating a component that deals with more than one clearly-defined concern. *ComponentB* and *ComponentC* violate the principle of separation of concerns since they are both responsible for multiple orthogonal concerns.

Quality Impact and Trade-offs. The Scattered Parasitic Functionality smell adversely affects modifiability, understandability, testability, and reusability. Using the concrete illustration from Figure 4, modifiability, testability, and understandability of the system are reduced because when *SharedConcern* needs to be changed, there are three possible places where *SharedConcern* can be updated and tested. Another facet reducing understandability is that both *ComponentB* and *ComponentC* also deal with orthogonal concerns. Designers cannot reuse the implementation of *SharedConcern* depicted in Figure 4 without using all three components in the figure.

One situation where scattered functionality is acceptable is when the *SharedConcern* needs to be provided by multiple off-the-shelf (OTS) components whose internals are not available for modification.

Example from Industrial Systems. Bowman et al.'s study [4] illustrates an occurrence of Scattered Parasitic Functionality in the widely used Linux operating system. The case study reveals that Linux's status reporting of execution processes is actually implemented throughout the kernel, even though Linux's conceptual architecture indicates that status reporting should be implemented in the PROC file system component. Consequently, the status reporting functionality is scattered across components in the system. This instance of the smell resulted in two unintended dependencies on the PROC file system, namely, the

Network Interface and Process Scheduler components became dependent on the PROC file system.

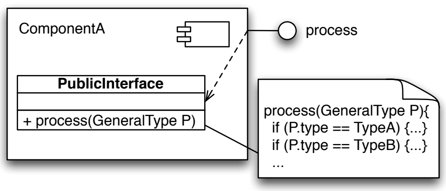
The PROC file system example suffers from the same diminished lifecycle properties as the notional system described in the schematic in Figure 4. Modifiability and testability are reduced because updates to status reporting functionality result in multiple places throughout the kernel that can be tested or changed. Furthermore, understandability is decreased by the additional associations created by Scattered Parasitic Functionality among components.

The developers of Linux may have implemented the operating system in this manner since status reporting of different components may be assigned to each one of those components. Although it may at first glance make sense to distribute such functionality across components, more maintainable solutions exist, such as implementing a monitoring connector to exchange status reporting data or creating an aspect [27] for status reporting.

5.3 Ambiguous Interfaces

Description. *Ambiguous Interfaces* are interfaces that offer only a single, general entry-point into a component. This smell appears especially in event-based publish-subscribe systems, where interactions are not explicitly modeled and multiple components exchange event messages via a shared event bus. In this class of systems, Ambiguous Interfaces undermine static dependency analysis for determining execution flows among the components. They also appear in systems where components use general types such as strings or integers to perform dynamic dispatch. Unlike other constructs that reduce static analyzability, such as function pointers and polymorphism, Ambiguous Interfaces are not programming language constructs; rather, Ambiguous Interfaces reduce static analyzability at the architectural level and can occur independently of the implementation-level constructs that realize them.

Two criteria define the Ambiguous Interface smell depicted in Figure 5. First, an Ambiguous Interface offers only one public service or method, although its component offers and processes multiple services. The component accepts all invocation requests through this single entry-point and internally dispatches to other services or methods. Second, since the interface only offers one entry-point, the accepted type is consequently overly general. Therefore, a component



```
process(GeneralType P){
  if (P.type == TypeA) {...}
  if (P.type == TypeB) {...}
  ...
```

Fig. 5. An Ambiguous Interface is implemented using a single public method with a generic type as a parameter.

implementing this interface claims to handle more types of parameters than it will actually process by accepting the parameter P of generic type *GeneralType*. The decision whether the component filters or accepts an incoming event is part of the component implementation and usually hidden to other elements in the system.

Quality Impact and Trade-offs. Ambiguous Interfaces reduce a system's analyzability and understandability because an Ambiguous Interface does not reveal which services a component is offering. A user of this component has to inspect the component's implementation before using its services. Additionally, in an event-based system, Ambiguous Interfaces cause a static analysis to over-generalize potential dependencies. They indicate that all subscribers attached to an event bus are dependent on all publishers attached to that same bus. Therefore, the system seems to be more widely coupled than what is actually manifested at run-time. Even though systems utilizing the event-based style typically have Ambiguous Interfaces, components utilizing direct invocation may also suffer from Ambiguous Interfaces. Although dependencies between these components are statically recoverable, the particular service being invoked by the calling component may not be if the called component contains a single interface that is an entry point to multiple services.

The following example helps to illustrate the negative effect of the wide coupling. Consider an event-based system containing n components, where all components are connected to a shared event bus. Each component can publish events and subscribes to all events. A change to one publisher service of a component could impact $(n-1)$ components, since all components appear to be subscribed to the event, even if they immediately discard this event. A more precise interface would increase understandability by narrowing the number of possible subscribers to the publishing service. Continuing with the above example, if each component would list its detailed subscriptions, a maintenance engineer could see which m components $(m \leq n)$ would be affected by changing the specific publisher service. Therefore, the engineer would only have to inspect the change effect on m components instead of $n - 1$. Often times, components exchange events in long interactions sequences; in these cases, the Ambiguous Interface smell forces an architect to repeatedly determine component dependencies for each step in the interaction sequence.

Example from Industrial Systems. A significant number of event-based middleware systems suffer from the form of Ambiguous Interface smell depicted in Figure 5. An example of a widely used system that follows this design is the Java Messaging Service (JMS) [28]. Consumers in JMS receive generic Message objects through a single *receive* method. The message objects are typically cast to specific message types before any one of them is to be processed. Another event-based system that acts in this manner is the *Information Bus* [29]. In this system, publishers mark the events they send with subjects and consumers can subscribe to a particular subject. Consumers may subscribe to events using a partially specified subject or through wild-cards, which encourage programmers to subscribe to more events then they actually process.

The event-based mechanism used by MIDAS conforms to the diagram in Figure 5. In the manner described above, MIDAS is able to easily achieve dynamic adaptation. Through the use of DLLs, MIDAS can add, remove, and replace components during run-time, even in a highly resource-constrained sensor network system. As mentioned in Section 4.2, we have recently spent three person-months refactoring the system to achieve better modularity, understandability, and adaptability. During the refactoring, determining dependencies and causality of events in the system was difficult due to the issues of over-generalized potential dependencies described above. An extensive amount of recovery needed to be done to determine which dependencies occur in what context.

5.4 Extraneous Adjacent Connector

Description. The *Extraneous Adjacent Connector* smell occurs when two connectors of different types are used to link a pair of components. Eight types of connectors have been identified and classified in the literature [25]. In this paper, we focus primarily on the impact of combining two particular types of connectors, procedure call and event connectors, but this smell applies to other connector types as well. Figure 6 shows a schematic view of two components that communicate using both a procedure call connector and an event-based connector.

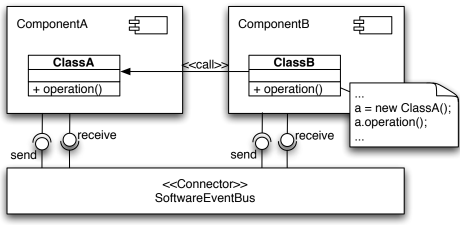


Fig. 6. The connector SoftwareEventBus is accompanied by a direct method invocation between two components.

In an event-based communication model, components transmit messages, called events, to other components asynchronously and possibly anonymously. In Figure 6, *ComponentA* and *ComponentB* communicate by sending events to the *SoftwareEventBus*, which dispatches the event to the recipient. Procedure calls transfer data and control through the direct invocation of a service interface provided by a component. As shown in Figure 6, an object of type *ClassB* in *ComponentB* communicates with *ComponentA* using a direct method call.

Quality Impact and Trade-offs. An architect's choice of connector types may affect particular lifecycle properties. For example, procedure calls have a positive affect on understandability, since direct method invocations make the transfer of control explicit and, as a result, control dependencies become easily traceable. On the other hand, event connectors increase reusability and adaptability because senders and receivers of events are usually unaware of each other and, therefore, can more easily be replaced or updated. However, having two architectural elements that communicate over different connector types in parallel carries the danger that the beneficial effects of each individual connector may cancel each other out.

While method calls increase understandability, using an additional event-based connector reduces this benefit because it is unclear whether and under what circumstances additional communication occurs between *ComponentA* and *ComponentB*. For example, it is not evident whether *ComponentA* functionality needs to invoke services in *ComponentB*. Furthermore, while an event connector can enforce an ordered delivery of events (e.g., using a FIFO policy), the procedure call might bypass this ordering. Consequently, understandability is affected, because a software maintenance engineer has to consider the (often unforeseen and even unforeseeable) side effects the connector types may have on one another.

On the other hand, the direct method invocation potentially cancels the positive impact of the event connector on adaptability and reusability. In cases where only an event connector is used, components can be replaced during system runtime or redeployed onto different hosts. In the scenario in Figure 6, *ComponentA*'s implementation cannot be replaced, moved or updated during runtime without invalidating the direct reference *ComponentB* has on *ClassA*.

This smell may be acceptable in certain cases. For example, standalone desktop applications often use both connector types to handle user input via a GUI. In these cases, event connectors are not used for adaptability benefits, but to enable asynchronous handling of GUI events from the user.

Example from Industrial Systems. In the MIDAS system, shown in Figure 2, the primary method of communication is through event-based connectors provided by the underlying architectural framework. All high-level services of MIDAS, such as resource discovery and fault-tolerance were also implemented using event-based communication. While refactoring as described in Section 4.2, we observed an instance of the Extraneous Adjacent Connector smell. We identified that the Service Discovery Engine, which contains resource discovery logic, was directly accessing the Service Registry component using procedure calls. During the refactoring an additional event-based connector for routing had to be placed between these two components, because the Fault Tolerance Engine, which contains the fault tolerance logic, also needed access to the Service Registry. However, the existing procedure call connector increased the coupling between those two components and prevented dynamic adaptation of both components.

This smell was accidentally introduced in MIDAS to solve another challenge encountered during the implementation. In the original design, the Service Discovery Engine was broadcasting its events to all attached connectors. One of these connectors enabled the Service Discovery Engine to access peers over a UDP/IP network. This instance of the Extraneous Adjacent Connector smell was introduced so that the Service Discovery Engine could directly access the

Service Registry, avoiding unnecessary network traffic. However, as discussed, the introduced smell instance caused the adaptability of the system to decrease.

6 Conclusion

Code smells have helped developers identify when and where source code needs to be refactored [2]. Analogously, architectural smells tell architects when and where to refactor their architectures. Architectural smells manifest themselves as violations of traditional software engineering principles, such as isolation of change and separation of concerns, but they go beyond these general principles by providing specific repeatable forms that have the potential to be automatically detected. The notion of architectural smells can be applied to large, complex systems by revealing opportunities for smaller, local changes within the architecture that cumulatively add up to improved system quality. Therefore, architects can use the concept (and emerging catalogue) of smells to analyze the most relevant parts of an architecture without needing to deal with the intractability of analyzing the system as a whole.

Future work on architectural smells includes a categorization of architectural smells, architectural smell detection and correction processes, and tool support to aid in those processes. A categorization of architectural smells would include an extensive list of smells and an analysis of the impact, origins, and ways to correct the smells. Architectural smells may be captured in an architectural description language, which would allow conceptual architectures to be analyzed for smells before they are implemented. Correction of smells would include the inception of a set of architectural refactoring operations and the provision of tools to help recommend particular operations for detected smells. In attempting to repair architectures of widely-used systems, the authors of [23] identified a set of operations that can be used as a starting point for determining a complete set of architectural refactoring operations. By trying to correct some of the architectural smells we found in both our own and others' experiences, such as [4] [21] [22] [23], we hope to identify other architectural refactoring operations and determine which operations are relevant to particular smells.

References

1. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE TSE (Jan 2004)
2. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (1999)
3. Joshua Garcia, Daniel Popescu, G.E., Medvidovic, N.: Identifying Architectural Bad Smells. In: CSMR 2009. (2009)
4. Bowman, I., et al.: Linux as a case study: its extracted software architecture. In: Proc. of the 21st ICSE. (1999)
5. Mattmann, C.A., et al.: The anatomy and physiology of the grid revisited. Technical Report USC-CSSE-2008-820, Univ. of Southern California (2008)
6. Taylor, R., et al.: Software Architecture: Foundations, Theory, and Practice. John Wiley & Sons (2008)

7. Lippert, M., Roock, S.: Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. Wiley (2006)
8. Shaw, M., Garlan, D.: Software architecture: perspectives on an emerging discipline. Prentice-Hall, Inc. Upper Saddle River, NJ, USA (1996)
9. Marinescu, R.: Detection strategies: metrics-based rules for detecting design flaws. In: Proc. of the 20th IEEE ICSM. (2004)
10. Lozano, A., et al.: Assessing the impact of bad smells using historical information. 9th IWPSE (2007)
11. Brown, W., et al.: AntiPatterns - Refactoring Software, Architectures, and Projects in Crisis. Wiley, New York (1998)
12. Garlan, D., et al.: Architectural mismatch or why it's hard to build systems out of existing parts. In: Proc. of the 17th ICSE. (1995)
13. Gacek, C.: Detecting Architectural Mismatches During Systems Composition. PhD thesis, Univ. of Southern California (1998)
14. Abd-Allah, A.: Composing heterogeneous software architectures. PhD thesis, University of Southern California (1996)
15. Roshandel, R.: Calculating architectural reliability via modeling and analysis. In: Proc. of the 26th ICSE. (2004)
16. Leveson, N.G.: Safeware: System Safety and Computers. Addison-Wesley (1995)
17. Foster, I., et al.: The anatomy of the grid: Enabling scalable virtual organizations. IJHPCA **15**(3) (2001)
18. Mattmann, C., et al.: Unlocking the Grid. In: Proc. of the 8th CBSE. (2005)
19. Malek, S., et al.: Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In: Proc. of the 29th ICSE. (2007)
20. Seo, C., et al.: Exploring the role of software architecture in dynamic and fault tolerant pervasive systems. SEPCASE (2007)
21. Godfrey, M.W., Lee, E.H.S.: Secrets from the monster: Extracting mozilla's software architecture. In: Proc. of the Second CoSET. (2000)
22. Gröne, B., et al.: Architecture recovery of apache 1.3 – a case study. In: Proc. of SERP 2002. (2002)
23. Tran, J., et al.: Architectural repair of open source software. 8th IWPC (2000)
24. Shaw, M., et al.: Abstractions for software architecture and tools to support them. IEEE TSE (1995)
25. Mehta, N.R., et al.: Towards a taxonomy of software connectors. In: Proc. of the 22nd ICSE. (2000)
26. Tatebe, O., et al.: Grid datafarm architecture for petascale data intensive computing. In: Proc. of the 2nd IEEE/ACM CCGrid. (2002)
27. Kiczales, G., Hilsdale, E.: Aspect-Oriented Programming. Springer (2003)
28. Haase, K.: Java message service tutorial (2002)
29. Oki, B., et al.: The Information Bus: an architecture for extensible distributed systems. In: Proc. of the 14th ACM SOSP. (1994)