Identifying Architectural Bad Smells

Joshua Garcia, Daniel Popescu, George Edwards and Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{joshuaga,dpopescu,gedwards,neno}@usc.edu

Abstract

Certain design fragments in software architectures can have a negative impact on system maintainability. In this paper, we introduce the concept of architectural "bad smells," which are frequently recurring software designs that can have non-obvious and significant detrimental effects on system lifecycle properties. We define architectural smells and differentiate them from related concepts, such as architectural antipatterns and code smells. We also describe four representative architectural smells we encountered in the context of reverse-engineering eighteen grid technologies and refactoring one large industrial system.

1 Introduction

In order to successfully modify a legacy application to support new functionality, run on new platforms, or integrate with new systems, software evolution must be carefully managed and executed. Frequently, it is necessary to *refactor* [6] a system, so that new requirements can be supported in an efficient and reliable manner.

The most common way to determine how to refactor is to identify code *bad smells* [2] [6]. 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 long parameter lists and duplicated code. 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).

While detection and correction of code smells improves system maintainability, many maintainability issues originate from poor use of *software architecture-level* abstractions — components, connectors, styles, and so on — rather than implementation constructs. In the course of recovering the architecture of eighteen grid technologies [4] and maintaining an industrial middleware platform, called MIDAS [3], we noticed frequently recurring design fragments that have non-obvious and significant detrimental impacts on maintainability. We term these *architectural bad smells* because they are analogous to code smells: they represent common "solutions" that are not necessarily faulty or errant, but still negatively impact software quality. In the same way that identification of code smells and methods for addressing them have proven quite useful to software engineers [2], we posit that clearly codifying architectural smells will allow engineers to avoid and correct common design pitfalls.

The research contribution of this paper is two-fold. In Section 2, we introduce the notion of architectural smells, define the characteristics of architectural smells, and discuss how they differ from architectural antipatterns. In Section 3, we describe four architectural smells selected from a larger set that we discovered during our architectural recovery and refactoring projects.

2 Definition and Related Work

In this section, we define what constitutes an architectural smell and explain how smells are different from architectural antipatterns.

We define a software system's *architecture* as "the set of principal design decisions governing a system" [7]. 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. An *architectural smell* is a commonly (although not always intentionally) used architectural decision that negatively impacts system quality. Architectural smells may be caused by applying a design solution in an inappropriate context, mixing design fragments that have

undesirable emergent behaviors, or applying design abstractions at the wrong level of granularity.

Architectural smells directly affect lifecycle properties, but they may have harmful side effects on other quality properties such as performance and reliability. Therefore, architectural smells involve a trade-off between different properties, and system architects must determine whether action to correct the smell will result in a net benefit. Architectural smells are remedied by altering the internal structure of the system and the behaviors of internal system elements without changing the external behavior.

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). Furthermore, we do not consider the non-conformance of an implemented architecture to an intended architecture, by itself, to be an architectural smell.

Related to architectural smells are *architectural antipatterns* [1]. 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 processes difficulties (e.g., design-by-committee) rather than design problems. Antipatterns that pertain to architectural issues only capture the characteristics of poor design from a system-wide viewpoint (e.g., stove-piped system). 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.

3 Architectural Smells

This section describes four architectural smells in detail. 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. 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.

3.1 Connector Envy

Description. Components with *Connector Envy* encompass extensive interaction-related functionality that should be delegated to a connector. Connectors provide communication, coordination, conversion, and facilitation [5]. 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 or fault tolerance). Components that extensively utilize functionality from any of these four categories suffer from the Connector Envy smell.

Figure 1a shows a schematic view of a Connector Envy smell in which *ComponentA* implements communication and facilitation services. *ComponentA* imports a communication library because it manages the low-level networking facilities used to implement remote communication. Figure 1b depicts another Connector Envy smell in which *ComponentB* performs a conversion during processing. The *process* interface of *ComponentB* is implemented by the *PublicInterface* class. *PublicInterface* implements the *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 dependencies between interaction services and application-specific services, which make it difficult to reuse either without including the other. The overall understandability of the component decreases because disparate concerns are commingled. Testability is affected by Con-

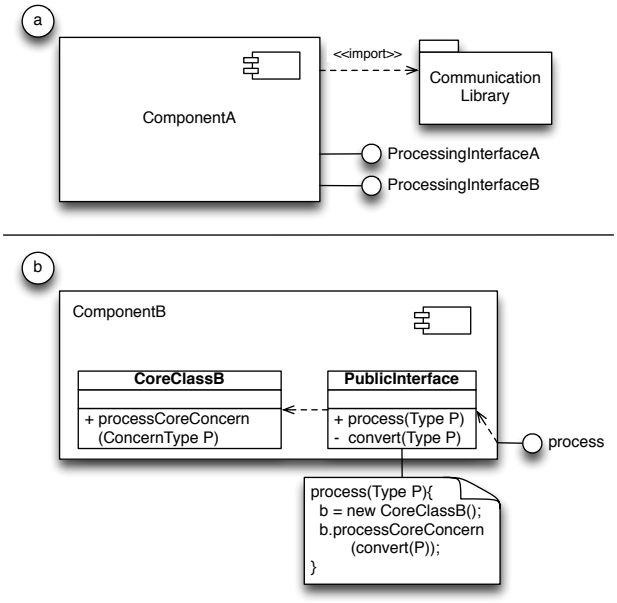

Figure 1. The top diagram depicts Connector Envy involving communication and facilitation. The bottom diagram shows Connector Envy involving conversion.

nector Envy, as any given test failure could be the result of a fault in either the interaction functionality or the application logic. As a result, software developers have to investigate two possible sources for the error instead of just one.

In cases where efficiency is of greater concern than maintainability, the Connector Envy smell may be acceptable. More specifically, explicitly separating the interaction mechanism from the application-specific code creates an extra level of indirection.

3.2 Scattered Functionality

Description. *Scattered 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 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 Functionality may be caused by cross-cutting concerns that are not addressed properly.

Figure 2 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 2 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 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 2 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.

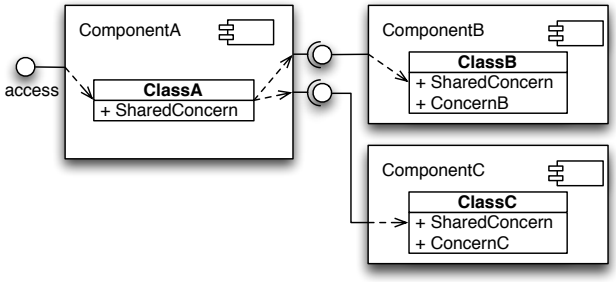

Figure 2. Scattered Functionality occurring across three components

3.3 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. They also appear in systems where components use general types such as strings or integers to perform dynamic dispatch.

Two criteria define the Ambiguous Interface smell depicted in Figure 3. 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 implementing this interface claims to handle more types of parameters than it will actually process. The decision whether the component filters or accepts an incoming event is part of the component implementation and hidden to other elements in the system.

Quality Impact and Trade-offs. Ambiguous Interfaces reduce a system's analyzability and understandability because a user of this component has to inspect the compo-

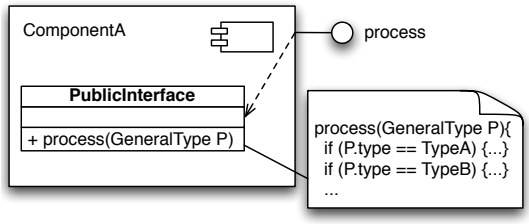

Figure 3. An Ambiguous Interface is implemented using a single public method with a generic type as a parameter.

nent's implementation before knowing about its offered services. Therefore, in an event-based system, Ambiguous Interfaces cause a static analysis of all component interfaces 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. Consequently, the system seems to be more widely coupled than what is actually manifested at run-time.

3.4 Extraneous Connector

Description. The *Extraneous Connector* smell occurs when two connectors of different types [5] are used to link a pair of components. In this paper, we focus primarily on the impact of combining two particular types of connectors, procedure call and event connectors as depicted in Figure 4, but this smell applies to other connector types as well.

In an event-based communication model, components exchange events asynchronously and possibly anonymously. In Figure 4, *ComponentA* and *ComponentB* communicate by sending events to the *SoftwareEventBus*, which dispatches each event to the recipient. Synchronous procedure calls transfer data and control through a service interface provided by a component. As shown in Figure 4, an object of type *ClassB* in *ComponentB* communicates with *ComponentA* using a synchronous method call.

Quality Impact and Trade-offs. 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 more 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. While method calls increase understandability, using an additional event-based connector between the same components reduces this benefit because it is unclear whether and under what circumstances additional communication occurs. In the example scenario, it is not evident whether *ComponentA* is invoking 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. 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 4, *ComponentA*'s implementation cannot be replaced, moved or updated during runtime without invalidating the direct reference *ComponentB* has on *ClassA*.

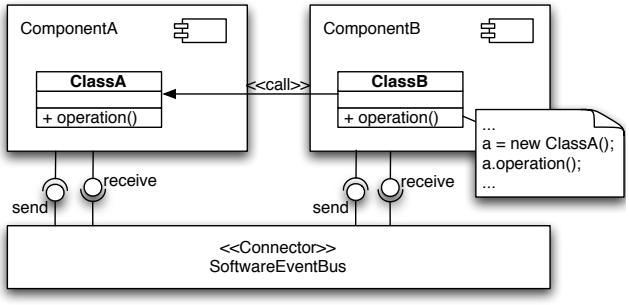

Figure 4. An event- and a procedure-call-connector between two components

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 concurrent input from the user.

4 Conclusion

Architectural smells constitute a class of potential problems caused by the presence of design fragments that detract from lifecycle properties. The schematic diagrams like those shown in Section 3 can be used by architects to detect architectural smells. Architectural smells can be detected in both the conceptual architecture of a software system and the recovered architecture of an implemented system. Once an architectural smell is detected, an architect can assess the impact of the smell on relevant qualities by conducting an analysis such as that outlined in Section 3.

References

[1] W. Brown, R. Malveau, H. M. III, T. Mowbray, J. Wiley, and I. Sons. *AntiPatterns - Refactoring Software, Architectures, and Projects in Crisis*. Wiley, New York, 1998.

[2] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[3] S. Malek, C. Seo, S. Ravula, B. Petrus, and N. Medvidovic. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In *Proc. of the 29th Int. Conf. on Software Engineering*, 2007.

[4] C. A. Mattmann, J. Garcia, I. Krka, D. Popescu, and N. Medvidovic. The anatomy and physiology of the grid revisited. Technical Report USC-CSSE-2008-820, Univ. of Southern California, 2008.

[5] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proc. of the 22nd Int. Conf. on Software Engineering*, 2000.

[6] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Trans. on Software Engineering*, Jan 2004.

[7] R. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2008.