

# Assessing Design Quality From a Software Architectural Perspective

Jeromy Carrière, Rick Kazman  
Software Engineering Institute, Carnegie Mellon University  
Pittsburgh, PA 15213  
{sjc, kazman}@sei.cmu.edu

**Abstract:** In this paper, we take the position that good object oriented designs accrue from attention to both the design of objects and classes and to the architectural framework which defines how instances of those classes interact. We argue that an architecture should be assessed for conceptual integrity, and describe tool support for making such an assessment.

## 1 What Makes a Good Object Oriented Design?

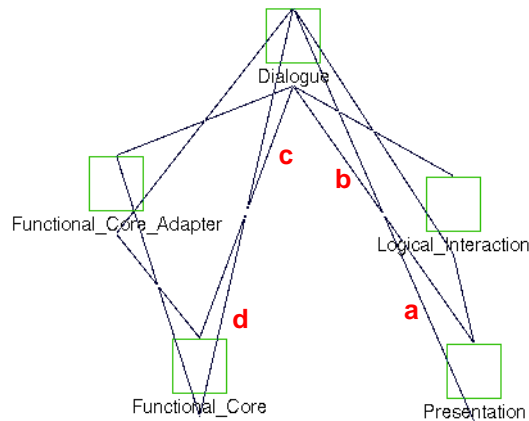
At the Software Engineering Institute, we are regularly asked to assess the design of large software intensive systems. We have gathered and documented a considerable amount of expertise aimed at answering the question of what makes a good design, object oriented or otherwise ([1], [2]). We have also produced and widely applied a method for investigating the “goodness” of designs [4]. So, what has this wealth of experience taught us?

In our experience, design goodness flows from the ability to do architectural reasoning. This appears to merely replace one question with another, for what factors contribute to the ability to do architectural reasoning? We will discuss two of these factors (a more complete discussion can be found in [2]):

- the existence of an architecture, *on top of* any object/class design
- the internal regularity (sometimes called pattern-based simplicity or *conceptual integrity* [3]) of the architectural design

For example, Figure 1 shows the software architecture of VANISH, a medium-sized (about 50,000 lines of code) system for prototyping visualizations, written in C++. VANISH provides a programming environment for creating visualizations, and a means of easily integrating new domains to be visualized and new interaction toolkits.

VANISH’s software architecture follows the Arch metamodel [7], which divides an interactive application into 5 layers: Functional Core (the system’s core functionality or purpose), Functional Core Adapter (which mediates between the dialogue and functional core by providing a unified, generic view of the functional core to the dialogue), Dialogue (a programmable mediator between domain specific and presentation specific functions), Logical Interaction (a virtual interaction toolkit layer), and Presentation (which contains the toolkits that implement the physical interaction between the user and the computer).



**Figure 1: The Software Architecture of VANISH**

We claim that VANISH's design is “good” because we can reason about it at the architectural level. At this level, we do not see individual objects or classes. In particular, we can reason about whether the high-level clusters of components—the layers—exhibit conceptual integrity, and whether the links between the layers are semantically consistent. Given that we can do this reasoning, it then follows that we can identify architectural *violations*: areas of the architecture that do not follow the rules of the architecture, and hence to not exhibit conceptual integrity.

At these areas we, as designers, can choose to fix the violation, document the violation as an allowable exception, or change the architectural description.

## 2 How is a Good Design Achieved?

Having established the characteristics of a good design, we present an infrastructure (or *work-bench*) that provides an environment for assessing a system's design based on its implementation. There are four steps involved in such an assessment:

1. extraction of a *source model* from the system's implementation
2. identification of a set of *design rules* specifying a view of the system's architectural design
3. visualization of the architectural design
4. evaluation of the architectural design

Source model extraction is a necessary first step for an assessment of this type; unfortunately, it is also a step that frequently requires a great deal of effort. Parsing is the most well-known method for extracting static (as opposed to dynamic or run-time) elements from source code. However, implementing a parser is difficult and time consuming. We have explored the use of an alternative for static extraction: lightweight lexical extraction using regular expression-based patterns [6]. This technique does not purport to be a fully-accurate one: there are a number of particular types of

source elements that it is not effective at extracting. One such type of element is calls through virtual functions.

The next step in the assessment process is creation of a set of design rules describing a view of the system's architectural design. These rules are intended to concisely describe how elements of the source model (such as functions, classes and files) are related to architectural components. We have found that queries over an SQL database storing the source model are an effective representation of the these types of relationships.

Visualization of the system's architecture as it evolves from the source model provides useful guidance in determining the value of the design rules. Evaluating how a given rule transforms the model is important: each rule should attempt to maintain semantic consistency within the model. Rigi provides an appropriate environment for the iterative process of developing design rules and visualizing a system's architecture.

Finally, once a system's architecture has been constructed from design rules, the architecture itself can be evaluated. The overall model can be evaluated for consistency and conceptual integrity. This is a type of *conformance testing*—how well does the system match the architectural model that has been developed for it? When the system exhibits deviations from the architectural model, these deviations can be identified as *acceptable* (the architectural description need not be altered to accomodate the deviation), as *exceptions* (the architectural description should be altered to incorporate the deviation) or as *opportunities for improvement* (the implementation should be modified to remove the deviation).

Referring to Figure 1, we can assess how well the implementation of VANISH conforms to its architectural design. The Arch-Slinky metamodel is strictly layered: a component should only have “connections” to its immediate neighbors. We can immediately observe in the figure that this constraint is violated in four places (labelled **a**, **b**, **c** and **d**). These violations are easily classified as described above by examining the nature of the links between the layers:

- **a** represents the instantiation by the Dialogue of classes that are part of the Presentation. This is an *acceptable* deviation imposed by the nature of the programming language.
- **b** represents the calls by the Dialogue to the constructors of the classes whose instantiation makes up **a**. For the same reasons as **a**, **b** is an *acceptable* deviation.
- **c** is similar in nature to **b**: it represents the calls by the Dialogue to constructors of classes in the Functional Core. For the same reasons as **a** and **b**, **c** is an *acceptable* deviation.
- **d** represents calls by the Functional Core to member functions of classes in the Dialogue. These calls bypass the intent of the Functional Core Adapter, exposing the Functional Core to the details of the Dialogue. This is an *opportunity for improvement*. (Note: because the Dialogue does not maintain instances of classes in the Functional Core, there is no analogue of **a** between these two layers).

### 3 References

- [1] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, A. Zaremski, "Recommended Best Industrial Practice for Software Architecture Evaluation", Software Engineering Institute Technical Report, CMU/SEI-96-TR-025, 1996.
- [2] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1997 (in press).
- [3] F. Brooks, *The Mythical Man-Month—Essays on Software Engineering*, Addison-Wesley, 1975.
- [4] R. Kazman, G. Abowd, L. Bass, P. Clements, "Scenario-Based Analysis of Software Architecture", *IEEE Software*, Nov. 1996, pp. 47-55.
- [5] R. Kazman, J. Carrière, "An Adaptable Software Architecture for Rapidly Creating Information Visualizations", *Proceedings of Graphics Interface '96*, (Toronto, ON), May 1996, pp. 17-27.
- [6] G. Murphy, D. Notkin, "Lightweight Lexical Source Model Extraction", *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 3, July 1996, p. 262-292.
- [7] UIMS Tool Developers Workshop, "A Metamodel for the Runtime Architecture of an Interactive System", *SIGCHI Bulletin*, 24(1), January 1992, pp32-37.