A Comparison of Software Architectural Styles Using Aspects

Maziar Boddoohi, Devon M. Simmonds
Department of Computer Science
University of North Carolina Wilmington
/mb4596, simmonsd}@uncw.edu

Abstract

To meet the demand and stay competitive, many systems were traditionally pieced together without much consideration given to their quality, modifiability, scalability, security, or maintainability. This has since littered the computing landscape with brittle applications with high maintenance and complexity. Over time, these maligned systems have propagated in size and merged or integrated to become too complex and fragile to amend or administer. To combat this misalignment, the Software Architecture (SA) discipline promises to answer and realign IT with its origins of productivity and proficiency.

In this paper we will (1) introduce SA, (2) decompose SA into its unique styles and quality attributes, (3) present a case study to gauge each style, (4) and finally assess and compare SA methodologies. In this paper we will compare alternative architectures and measure their effectiveness in order to identify and compute factors that affect SA by utilizing Aspect Oriented Programming (AOP).

1. Introduction

Over the last few decades, informational technology (IT) has rapidly emerged as the primary subject that chief executive officers have resourced to address rising costs and promote productivity (1). The introduction and integration of the World Wide Web alongside advanced computing frameworks has elevated IT into an essential commodity (2). Information Technology is no longer a luxurious extension to the logistics of large corporations, or simply an analytical tool for data. Rather IT has become a fundamental component in businesses of all sizes, a focal point for business expansion and a competitive advantage leveraged to trim costs, reach a broader audience, and forge alliances (2). It is no wonder then that this technology has prompted and promoted exponential growth and influence around the globe (3) resulting in greater reliance on and more significance being given to company’s computational resources (4).

As with any emerging technology however, IT suffers from its immaturity (5). To meet market demands and stay competitive, many software systems were pieced together without ample consideration given to overall software qualities such as modifiability, scalability, security, or maintainability (6) (7). As a result, this rapid proliferation has led to a computing landscape littered with brittle applications having high associated maintenance costs (8) and unnecessary accidental complexity (9). As it turns out, the same IT innovation that aided companies has in many cases become an expensive and liable tumor. Over time, these maligned systems have propagated in size and merged or integrated with other systems to become too complex and fragile to amend or administer (10).

To combat this misalignment, the discipline of Software Architecture (SA) was forged (11) (12). From its conception, SA boasts the capacity to realign IT with its quality goals including proficiency and efficiency (13). To do this, SA proposes to decompose the software development process by assigning functional requirements and partitioning interests in order to minimize complexity and predict software quality before it is deployed to the stakeholder (14). Accurately designed systems have proven to not only dramatically reduce overall cost and customer satisfaction, but also ensure such important quality goals as modifiability, security, testability, re-usability, comprehensibility and scalability.

As Shaw and Garlan outlines (5), software architecture can resolve wide organizational or global structures, functional assignments, physical distribution, scaling and performance. SA not only identifies commonalities among alternative paradigms, but also recognizes that a detailed documentation and understanding of the system enables engineers to both sustain and scale complex systems.

Yet, to fully meet its potential, SA must be proven to be cost effective before the foundation of the software has been laid. SA must enable the stakeholder to weigh the potential long term financial burden. Today, most quality variables are unused in the context of conceptual modeling of architecture. They can only be evaluated after the code has been developed and deployed.

This paper describes the evaluation of the overall performance and modifiability of applications developed using the shared data and abstract data type architectural styles. The evaluation is done using an oblivious, non-invasive aspect-oriented (15) approach. The goal of the research is to empirically demonstrate, with numerical values, how architectural decisions made in the dawning hours of a software system can directly affect its long
term quality. Section 2 provides an overview of software architectural styles and section important architectural quality attributes. The research method is presented in section 3 along with an overview of the target application. Sections 4, 5 and 6 present the research results, discussion and lessons learned, and conclusion and future work respectively.

2. Architectural Styles

To fully understand software architecture we must place ourselves in the architect’s role. The first step in approaching SA is to outline the software requirements. Once the requirements of the system are verbalized and agreed upon, the architect must prioritize the requirements and identify the major software components or subsystems. The components are then organized according an architectural style that encompasses the overall structure of the system. An Architectural style (16) refers to “a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done” (17). Architectural components refer to computational artifacts, architectural connectors refer to the mechanisms of interactions between components (18) and architectural constraints are rules that govern the interactions between the connectors and components. We next describe the data abstraction, pipe and filter, implicit invocation and data-centric and client-server architectural styles (16).

2.1 Object Oriented or Data Abstraction

The Object-Oriented architectural style (see Figure 1) features abstract data types or objects that represent data and interact through functional or procedural invocations. While objects are responsible for preserving the integrity of their representation, they also hide information from other objects. Many examples of OO systems exist in the software landscape.

Reusability and modifiability are the strengths of this style, as it facilitates the modification of functionality while reusing other stable objects. As a side effect, if one object interacts with another, it must know the identity of that other object. Therefore if the first object changes, the method that invokes that object, will be agnostic to those changes. Also, if a child object inherits functionality from a parent object and the parent object changes, then the functional integrity of the child object may be adversely affected.

2.2. Pipe and Filter or Data Flow

This architectural style (see Figure 2) comprises of components that represent set of inputs and outputs, where the components read from a stream of data as its input and processes the data to deliver a stream of data for its output. Since each output is a result set of another component’s input, these independent entities are called filters. The communication or connection between the filters is termed pipes, since they serve as delivery mechanisms. An example of this architectural style is a message based system in which clients send requests to the queue, where the message is stored until an application removes it and processes it (16).

This style’s strengths lie in its ease of understanding and ability to reuse and maintain filters. Also, this style promotes loose coupling, in that no direct binding connects the filters, therefore each filter is oblivious of the others’ states or conditions. Yet, the weakness of pipe-and-filter styles lies in their inability to process threaded instructions, and its restrictions on the development of interactive applications.

2.3. Implicit Invocation or Event-Driven

In the Implicit invocation or event-driven architectural style (see Figure 3) an architectural component has the ability to publish or announce one or more events. Components that have an interest in a published event may register that interest by associating a procedure with the event of interest. When an event is announced, the broadcasting system invokes all of the procedures that have been registered for the event itself [Dong, Chen and Jeng, Garland and Shaw]

An advantage of event-based invocation is that it encourages reuse across the system. Also, different component such as agents, objects, processes, and servers, can be introduced in a system simply by registering them,
therefore making it easily extendable and modifiable. A drawback to this approach is that event-based systems become quite unpredictable and hard to control due to their autonomous nature.

2.4. Data-Centric or Shared Data

A shared repository or data-flow centric architecture (see Figure 4) consists of a central data structure and a collection of independent components which operate on the central data structure. This architectural style typically consists of components running in parallel and communicating through a data channel.

![Figure 4: Data-Centric Architectural Style](image)

The drawback to this style is the heavy reliance on the health of the shared data structure – if the data container is affected so is the rest of the system and its agents. Yet there are major advantages to simply having to change/modify one source rather than the segregates and agents.

2.5. Client-Server

The Client-Server style (see Figure 5) has been commonly used across distributed software applications (19). The server is designed to provide services to multiple instances of client. The clients can then only communicate with the server, and not each other. The communication is typically a request initiated by the client, in response to which the server typically performs some computation and delivers the values back to the client.

The greatest advantage of this style lays in its promotion of or separation of concerns into different tiers or layers. This enables the system to be portioned onto different independent machines or platforms, each operating autonomously, until an action is required. Also this style enables systems to communicate synchronously through request-reply methodology, ensuring the delivery/receipt of actions.

Once the software architect identifies potential architectural styles, he/she can begin to outline and classify each alternative using appropriate quality metrics after which a final selection may be made.

2.6. Quality Attributes

The benefits of one architectural style over another may be determined by comparing quality attributes of the architectures. Each quality attribute defines characteristics that the software must attain in order to meet developmental and execution specifications. In general, it is the aggregate of the software’s quality attributes that determines the extent to which the goals of the stakeholders are met by the software (19). The IEEE [ISO9126] has defined several important quality attributes (20) (see Table 1). From this list we report on our investigation into performance and maintainability of software architectures.

<table>
<thead>
<tr>
<th>Maintainability</th>
<th>A long-term requirement that can typically be combined with readability, understandability, modifiability, and comprehensibility.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>A measure that is evident immediately to the stakeholder and can be described by resolving such issues as availability, load-balancing, and usability.</td>
</tr>
<tr>
<td>Testability</td>
<td>The amount of effort it takes to verify the developed system against the functional requirements of the stakeholder.</td>
</tr>
<tr>
<td>Portability</td>
<td>Is a long-term goal that enables software migration and system upgrades.</td>
</tr>
<tr>
<td>Functionality</td>
<td>The set of attributes that bear on the existence of a set of functions and their specified properties.</td>
</tr>
<tr>
<td>Reusability</td>
<td>The ability to reuse components or the system itself in a new or modified stage.</td>
</tr>
</tbody>
</table>

| Table 1: IEEE Software Quality Attributes |

3. Method

Our research is motivated by several research questions including: (1) Which architectural style provides or promotes the best realization of each quality attribute in an implemented system?, and (2) Given a specific architectural specification, can we classify potential code realizations and rank then in terms of how they reflect the quality attributes represented in the architecture?

In this paper we present our attempts to answer the first question for the data-centric and object-oriented architectural designs by comparing their final implementations. By evaluating the final product, we may gain perspectives on the development of the system that can aid the development of future architectural projects.

To realize our goal, we use Java implementations of our target application [ref] for the Shared Data and Object-Oriented architectural styles. The code for each program is then compiled and tested using input of differing sizes, against a fixed size file with a large number of lines of text in the English language.

The performance and modifiability quality attributes are then measured by obliviously interjecting code that effects each attribute into the applications using AspectJ. AspectJ (21) is an Aspect-oriented programming language that facilitates the identification, separation and representation of crosscutting software concerns. AspectJ is a great fit for evaluating architectural styles
because it is a simple and practical aspect-oriented extension to Java. Implementing each quality variable using AspectJ code atop of the working architectural style enables noninvasive evolution using the same controls for each system.

Use of AspectJ will also enable the reusability of the test cases themselves so that other systems and requirements, providing numerical values can be compared and contrasted.

### 3.1. The Target Application

We utilize the Key Work in Context (KWIC) index system first introduced by David L. Parnas in his classic 1972 paper “On the Criteria to Be Used in Decomposing Systems into Modules” (22). The KWIC system, “accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be “circularly shifted” by repeatedly removing the first word and appending it at the end of the line.” The KWIC index system is a convenient search mechanism for information in a long list of lines. It is now widely used for a web search engine and the permuted index for the Unix Man pages (16). The system output a listing of all circular shifts of all lines in alphabetical order.

We used KWIC Java source code for the Shared Data and Object-Oriented architectural designs written at Graz University of Technology, Austria (23).

Some sample content and input file sizes are presented in Table 2.

<table>
<thead>
<tr>
<th>A Input</th>
<th>B Input</th>
<th>C Input</th>
<th>D Input</th>
<th>E Input</th>
<th>F Input</th>
<th>G Input</th>
<th>X Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 kb</td>
<td>33 kb</td>
<td>66 kb</td>
<td>132 kb</td>
<td>394 kb</td>
<td>788 kb</td>
<td>1707 kb</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: File Sizes Used To Test Architectures

This case study provides a suitable test container for experimenting and studying the effects of architectural styles. Below we will describe how the implementation has been designed in both Object-Oriented and Shared Data designs.

### 4. Results

Before any modifications or extensions are applied to the base-code, metrics were collected from the baseline source. Each approach’s total time of execution was recorded using Aspects against each inputted file size. The Table 3 describes the time recorded (in seconds):

<table>
<thead>
<tr>
<th>Object-Oriented</th>
<th>1.12</th>
<th>4.34</th>
<th>1.80</th>
<th>1.77</th>
<th>5.66</th>
<th>9.96</th>
<th>21.76</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Data</td>
<td>1.14</td>
<td>2.26</td>
<td>0.45</td>
<td>0.92</td>
<td>4.68</td>
<td>14.13</td>
<td>82.39</td>
</tr>
</tbody>
</table>

Table 3: Performance of Base Code

As the results show, while Object Oriented approach grows in proportion to the size of the input file, the Shared-Data approach grows at a faster rate, but performs better when the file size is relatively small. In order to identify why one approach performs better than the other, we must decompose the style into its components. To do so, we will define some software metrics to compare the underlying code against (24). Some of these rudimentary metrics are described in Table 4.

<table>
<thead>
<tr>
<th>Number of Attributes (NOF)</th>
<th>Total number of attributes in the selected scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Classes (NOC)</td>
<td>number of classes in the selected scope</td>
</tr>
<tr>
<td>Number of Methods (NOM)</td>
<td>Total number of methods defined in the selected scope</td>
</tr>
<tr>
<td>Number of Parameters (PAR)</td>
<td>Total number of parameters in the selected scope</td>
</tr>
<tr>
<td>Number of Static Methods (NSM)</td>
<td>Total number of static methods in the selected scope</td>
</tr>
</tbody>
</table>

Table 4: Metrics Used to Normalize Base Code

The results of applying these metrics to the two architectures are shown in Table 5. As we can see, although the OO style contains more parameters, methods, and classes it’s able to sustain a linear relationship with the size of input. But, while the Shared-Data seems to be implemented in a simpler fashion (less parameters and classes); it lacks the abstractions needed to sustain the approaches when the number of instructions and processes increase.

<table>
<thead>
<tr>
<th>Object Oriented</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Attributes</td>
<td>4</td>
</tr>
<tr>
<td>Number of Classes</td>
<td>6</td>
</tr>
<tr>
<td>Number of Methods</td>
<td>38</td>
</tr>
<tr>
<td>Number of Parameters</td>
<td>61</td>
</tr>
<tr>
<td>Number of Static Methods</td>
<td>1</td>
</tr>
<tr>
<td>Grand Total</td>
<td>110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shared Data</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Attributes</td>
<td>6</td>
</tr>
<tr>
<td>Number of Classes</td>
<td>1</td>
</tr>
<tr>
<td>Number of Methods</td>
<td>11</td>
</tr>
<tr>
<td>Number of Parameters</td>
<td>5</td>
</tr>
<tr>
<td>Number of Static Methods</td>
<td>1</td>
</tr>
<tr>
<td>Grand Total</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 5: Comparative Results

Yet, while the above metrics give a more accurate depiction of the source code, they lack the complexity needed in achieving true comparative analysis. Therefore, we enlisted other multifaceted comparison metrics. These metrics and the results of applying them to the two architectures are shown in Table 6 and Figure 7.
Depth of Inheritance Tree (DIT) is the maximum length from a node to the root (base class) where lower level subclasses inherit a number of methods making behavior harder to predict.

Lack of Cohesion of Methods (LCOM*) is calculated with the Henderson-Sellers method. A low value indicates a cohesive class and a value close to 1 indicates a lack of cohesion and suggests the class might better be split into a number of (sub)classes.

McCabe Cyclomatic Complexity counts the number of flows through a piece of code. Each time a branch occurs this metric is incremented by one.

Method Lines of Code (MLOC) is the total number of lines of code inside method bodies, excluding blank lines and comments.

Nested Block Depth is the depth of nested blocks of code.

Weighted methods per Class (WMC) is the sum of the McCabe Cyclomatic Complexity for all methods in a class.

Object-Oriented Shared Data

<table>
<thead>
<tr>
<th>Metric</th>
<th>Object-Oriented</th>
<th>Shared Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth of Inheritance Tree</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Lack of Cohesion of Methods</td>
<td>0.125</td>
<td>0.833</td>
</tr>
<tr>
<td>McCabe Cyclomatic Complexity</td>
<td>64</td>
<td>48</td>
</tr>
<tr>
<td>Method Lines of Code</td>
<td>197</td>
<td>172</td>
</tr>
<tr>
<td>Nested Block Depth</td>
<td>48</td>
<td>26</td>
</tr>
<tr>
<td>Weighted methods per Class</td>
<td>71</td>
<td>49</td>
</tr>
</tbody>
</table>

Table 6: Comparison Using Additional metrics

Again, as we can see OO approach proves to be the more complex implementation. It has a higher Weighted and cyclomatic complexity, with more method lines of code and nested depth. The Shared-Data implementation’s lack of cohesion is evident in the LCOM measurement. With the cohesive mark near 1, the methodology shows that methods within a class are less related to one another.

Now that we understand the behavior of the base-line code, what if we were to extend or modify the code? How easily can we append a class or functionality? How easily can we modify an existing functionality?

In order to further gauge the effectiveness and modifiability of the architectural styles, a Dictionary class was appended to each style. The purpose of the Dictionary class is to verify the file’s text against a known (indexed) list of English words. If a word is not contained in the dictionary, then the word is truncated from being alphabetized and excluded from the result set altogether.

During the implementation of the dictionary, the prior metrics that pointed to the lack of cohesion and elegance to the underlying code became evident. While the OO style could be easily extended due to its cohesion and inheritance, the shared data implementation was very difficult to decode and test. Furthermore, since two arrays of data (one for the actual data and one for the end-of-line index) had to be wrapped in for loops every time any data needed to be extracted or reordered, the implementation was prone to more errors and null pointers. As a result the time to weave the dictionary was extensively more than the object oriented implementation.

<table>
<thead>
<tr>
<th>Style</th>
<th>Time to test &amp; develop</th>
</tr>
</thead>
<tbody>
<tr>
<td>OO</td>
<td>4.73 hours</td>
</tr>
<tr>
<td>Shared Data</td>
<td>43.5 hours</td>
</tr>
</tbody>
</table>

Table 7: Total hours of development

As Figure 8 shows, although the metrics for complexity and lines of code remained nearly the same proportionally, the impact of cohesion as a result of the Dictionary class rose significantly. OO style’s cohesion rose from .125 to 1.292 while the Shared-Data raised from .833 to 1.589, and as a result so the latency of the application to process.

Yet, while the OO style’s increase in processing time was relatively proportional to the size of the input file, after the extension of the Dictionary, the Shared-Data’s processing time curved toward infinity at a sharper rate.
5. Discussion and Lessons Learned

As we can see, while a simpler and less modular approach to software architecture may resolve the problem at hand, but the lack of cohesion causes significant challenges when the application is faced with large inputs or processes. Furthermore, while design and implementation of code cultivated in a simple data container may be effortless initially, the extension or modification of that code by another party can take a significantly higher amount of time accomplish. Also, as a result the code could no longer be reused in other applications or processes.

If the source is amended or extended within the same architectural boundaries, then each extension could then add another layer of inconsistency and in time default existing code for a rewrite.

6. Conclusion and Future Work

7. Bibliography


