Advanced Software Analysis and Design

Architectural Foundations for Scalable, Distributed Software Systems

James William Dunn
jwdunn2@asu.edu

I. Introduction

The following report addresses the architectural foundations for designing scalable, distributed software systems. There are five phases described herein, with each phase corresponding to a project within the course. The first phase focuses on the initial design process of identifying actors and use cases, performing object identification, establishing relationships, and building use-case and class diagrams. The second phase builds on object-oriented design principles and aims to clearly delineate the roles each class performs by decoupling modules and increasing efficiency through the utilization of well-known design patterns. The third phase develops quality attribute scenarios and further documents the architecture with hierarchical diagrams. The fourth phase highlights the model-view-controller architectural design pattern and considers a native mobile application with remote web service calls. Finally, the fifth phase concentrates on designing and implementing scalable, distributed services. Collectively, these phases can be employed to solve complex, large-scale, real-world problems.

II. Solution

In phase one, the analysis of a project brief or requirements description reveals several important characteristics including actors and how they interact with a software system. These can be refined further into use-case descriptions to clearly explain the basic flow of information and activity. During this process, an extraction of nouns and verbs helps to identify objects and functionalities. Additionally, categorization of objects aids in identifying boundary, entity, and control classes. Class-Responsibility-Collaborator (CRC) cards assist in describing how the classes behave and interconnect. Unified Modeling Language (UML) documents use-cases, classes, and their relationships in the form of diagrams [1]. With a tool such as Astah UML, there exists an option to export the class structure as skeleton code for further development.

Phase two analyzes the system from an object-oriented design standpoint to ensure modularity by separating the concerns of each component. This leads to further analysis to determine the degree of coupling or independence between components and classification of the cohesiveness within each module. Furthermore, employment of design patterns such as factory, object pool, or singleton increases the efficiency of the design. Finally, it is important to clearly define the interface for each module in order to accommodate evolutionary modifications with minimal side effects, increased maintainability, and reduced complexity.

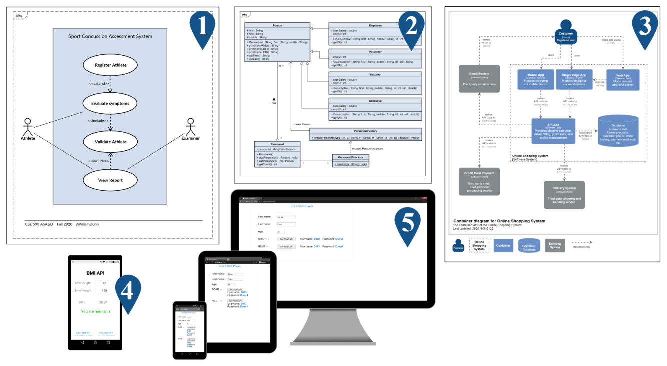

Figure 1: Artifacts produced: 1-use case diagram, 2-class diagram, 3-container diagram, 4-Android app, and 5-a responsive website consumes distributed services.

With phase three, the analysis focuses on defining design quality factors such as performance, confidentiality, and recoverability. Evaluation of these factors requires the creation of quality attribute scenarios which identify stimulus and response measures. Scenarios contain response predicates such that they can be *falsifiable* during testing conditions. For example, if a measure of performance exceeds *n* seconds, then it is possible to conclude that the time response measure has failed the test. This is far more effective than "the system must be fast."

This phase also includes architectural documentation. The C4 model hierarchical diagrams represent system views from different perspectives [2]. For example, a high-level contextual view is useful for a non-technical audience, while the next level of detail indicates the general shape of the software architecture and targets more technical individuals. Additional levels of detail drill even further to map a model into a physical infrastructure.

Phase four explores the model-view-controller architectural design pattern, mobile applications, and remote service calls. The model embodies the main application logic, business logic, and common data. A view presents information to a user. A controller processes input from the user and makes update requests to the model and revisions to a view. The model remains independent of any controller or view and is usually responsible for making remote calls for data. Input validation prevents erroneous parameters. At this stage, it also makes sense to perform user experience testing to gain valuable and unexpected feedback which can inform an additional iteration through the design process.

Finally, phase five turns to the design of distributed services using web communication protocols such as Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) [3]. For backwards compatibility, Windows Communication Foundation (WCF) is useful to implement a service endpoint with SOAP, along with its corresponding Web Services Description Language (WSDL) [4]. The service reference must be included in the client application. Modern approaches utilize ASP.NET Core (without WCF) to develop a web application programming interface (API) with REST. In this case, there is no need for the service reference in the client and Web Application Description Language (WADL) is optional. Instead, standard HTTP verbs identify the activity and method calls.

A website application developed using ASP.NET consumes the API services. The SOAP client procedures operate with a proxy object such that the remote methods and attributes of that object can be invoked as if it were residing in local memory. Communication with the remote service is automatic and unobtrusive. The REST client procedures may require JSON de-serialization if the data structure is non-primitive. When developing a website application, responsive web design provides an improved user experience across a plurality of devices with minimal effort of defining a few break-points in a Cascading Style Sheet (CSS). Finally, early input validation, such as checking for reasonable values, reduces or prevents erroneous parameter passing.

These phases constitute the architectural foundations that can be applied to many complex problems. Additional scalability considerations may include microservices architecture and highly-available workflow patterns [5].

III. Results

For phase one, I identified actors and use-cases from a requirements document. I then created a use-case diagram using Astah UML. Next, I identified objects using the noun technique and potential methods by highlighting verbs. I also created several instances of CRC cards to determine class responsibilities and collaborations. With this more concrete information, I constructed a class diagram with UML notation using the Astah tool and subsequently exported Java skeleton code. Having established the architectural framework, I was able to swiftly develop the application in conformance with requirements.

Next, for phase two, I reviewed existing code for any object-oriented violations, and coupling issues. My analysis revealed violations in abstraction, inheritance, modularity, and information hiding. For instance, several classes failed to encapsulate their member data elements. As a result, the calling class was able to manipulate internal values in an uncontrolled fashion. To fix this, I made the data members private and defined additional methods to retrieve and update the data. Further, I identified an instance of *control coupling* where one module controlled the logic of another; I repaired this issue by installing additional methods and removing the control logic. Without changing any intended functionality, the refactored code conforms to object-oriented principles and as a result is more modifiable and maintainable.

In phase three, I developed a scenario for each of three quality factors: time behavior (performance), confidentiality, and recoverability. In each case, I identified the stimulus, the software environment, and the response. For instance, to test performance, I suggested the development of an automated tool that can inject a transaction representing a standard order at a regular interval (say each hour). The response time would be recorded in a log. If the time ever exceeds the prescribed quality factor measure of 30 seconds, then the scenario is *falsified*. I also developed three levels of hierarchical C4 diagrams using the *draw.io* tool to document the context, containers, and deployment of a system (see Fig. 1). I utilized a Pantone family of colors for a consistent visual design.

During phase four, I focused on creating a mobile application, starting from a blank project in Android Studio. I defined the model component first as a subclass of *Application*. This module contains the logic such as determining a category based on a data value range. The model also includes methods to perform RESTful web API calls to retrieve and parse JavaScript Object Notation (JSON) data response packets (see Fig. 2). Next, I defined three views and their corresponding controllers. The first view-controller pair is for a briefly-displayed launch screen. Control transitions to the primary application view to collect input data and present two options. When the user touches one button, a request is made to the remote API. If the user touches the other button, control segues to a secondary view to present a web page within a *WebView* component. Beyond the requirements, I also constrained inputs for precautionary error checking. Further, I included event handling for screen orientation updates.

```java
public void requestBMI(String h, String w, uiCallback u) {
  height = h;
  weight = w;
  String uri = "http://webstrar99.fulton.asu.edu/page3/Ser
  String url = String.format(uri, h, w);
  JsonObjectRequest jobReq = new JsonObjectRequest(Request
    response -> {
      try {
        bmi = Float.parseFloat(response.getString("bmi"));
        risk = response.getString("risk");
        link = response.getJSONArray("more").getString(0);
        state = State.RESULT;
        u.notifyController();
      } catch (JSONException e) {
        e.printStackTrace();
      }
    },
    error -> Log.e("Volley", error.toString()));
  queue.add(jobReq);
}
```

Figure 2: Java code to invoke a web API from a mobile application.

Within phase five, I installed the optional WCF platform in Microsoft Visual Studio 2019 and proceeded to create two new C# service applications. I found it helpful to set the services project property to *Always Start When Debugging*. By default, a new WCF service uses SOAP as its messaging protocol. To convert the protocol to REST, I found two modifications are necessary: 1) add *WebInvoke* attributes to the interface definition file; and 2) modify the binding to "webHttpBinding" in the Web.config file and add the following endpoint behavior to the <*behaviors*> tag:

```
<endpointBehaviors>
  <behavior>
    <webHttp defaultOutgoingResponseFormat="Json" />
  </behavior>
</endpointBehaviors>
```

Next, to consume the services, I created an ASP.NET Web Application (using the .NET Framework) and added references to the services. I found it necessary to select *Update Service Reference* after each fresh rebuild of the application. I also discovered subsequently that it was unnecessary to include a reference to the REST service because it is loosely bound. To make the web application more robust, I included preemptive input value checking to reduce the necessity for parameter checking at the service endpoint. I wanted to make the website interface responsive to mobile, tablet, laptop, and desktop browsers. This requires setting the browser viewport, making CSS additions including defining several media breakpoints, and scaling fonts appropriately. I tested this across a set of devices and browsers ranging from mobile to desktop. Finally, to submit a minimal set of source code, I removed non-essential files and compressed the results into a 58K package.

IV. Lessons Learned

Companies worldwide depend on software architecture to work, behind the scenes, without fail. The highlight of this project was the perspective I gained in thinking through the larger picture and identifying the pertinent structures and their inter-relationships. Although I have extensive experience in software architecture and microservice development and optimization, this course provided a solid foundational review of object-oriented analysis and design fundamentals. The assignments in this course afforded me the opportunity to gain a more comprehensive ability to analyze the modularity of both the code I design and develop and that which I inspect and refactor.

The documentation of software architecture is invaluable in communicating significant design and development goals with stakeholders of a project. The focus on creating UML diagrams and C4 hierarchical visualizations boosted my understanding of these tools. The diagrammatic notation helps to map, record, and convey complex models and their components and interactions.

Today's needs change as new challenges arise. Thinking beyond the initial brief, the software architect is responsible for guiding the project toward a flexible future. Any system is only as good as the framework within which it resides. I found the study of modularity, coupling, and cohesion particularly applicable to my development practice. As I build components for systems, my increased awareness of how these elements interplay with each other allows me to better structure the overall project at hand. The architecture needs to be clearly defined for a cohesive product or service. If not, the end result may fall short of expectations and ultimately, as we become more dependent on software, human life could be in jeopardy.

Real-time space-craft and self-driving automobiles require quality factors operating at the highest degree, with emphasis on functional correctness, performance, and reliability. The method of identifying the stimulus, artifact, and response helped me to understand that a quality, which is typically considered ethereal or unquantifiable, can indeed be quantified, tested, and measured. The experience of developing quality attribute scenarios is a key takeaway from the project. I will apply this to my ongoing work with autonomous systems.

Within the context of a single-page application on a website, I have experimented with the MVC architecture in the past. This project, however, provided me with hands-on experience in applying the pattern to a *native* mobile application. Several years ago, I developed a game application using JavaScript, jQuery, and oCanvas and then deployed it to the mobile environment through PhoneGap Build. The present project has helped me to rapidly gain relevant experience and learn the tools to refactor that game into a native version.

I particularly enjoyed building a web service to perform computations remotely. This method has had and will continue to have an enormous impact on technology adoption and scalability. The project motivated me to investigate additional web service build options without WCF. For example, in Visual Studio, I was able to create a REST service project using ASP.NET Core. Then, I constructed a simple consumer page using only JavaScript which uses the Fetch API to call the service. I will use this lightweight approach in future web service developments.

Lastly, I found the project experience to be assistive in my current volunteer work, planning a scalable urban emergency environment to assist victims following a natural disaster. Utilizing the analytical tools and concepts gained, I can better help determine, guide, and document the software architecture for the management systems that control the complex operational fabric of a mid-scale city.

V. References

[1] Unified Modeling Language v2.5.1
https://www.omg.org/spec/UML/2.5.1/PDF
[2] The C4 model for visualising software architecture (Context, Containers, Components, and Code)
https://c4model.com
[3] Fielding, R. "Architectural Styles and the Design of Network-based Software Architectures", 2000,
https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
[4] Web Services Description Language (WSDL) Version 2.0
https://www.w3.org/TR/wsdl20-primer
[5] Microsoft TechNet, "Architectural Design: A Scalable, Highly Available Business Object Architecture", 2000.
https://motivara.com/Archives/WinDNA.pdf