DRAFT

Managing Complexity in Software: Part 2 – User-Oriented Architecture and N-Dimensional Architecture with Scaling [Software Architecture]

Peter Joh, *University of Michigan, BS CS*
November 2005

DRAFT

Abstract **-** Software architectures are becoming increasingly sophisticated. One of the reasons for this increased sophistication is so the developers of these systems will be able to make major changes with less time and effort. Examples of this sophistication are the use of helpful techniques like design patterns and architectural patterns, or even simply the creation of large object-oriented designs. But, this flexibility comes at a price and the architectures for these systems can be very complex. Developers may often find them difficult to learn and may also find them difficult to work with. In part one of this series of papers, the core concepts of N-Dimensional with Scaling were explained. In this paper (part two), we discuss User-Orienteded Architecture and delve further into the details of N-Dimensional Architecture with Scaling. The main principle of User-Oriented Architecture is: similar to how a system should be oriented around its users, the architecture of a system should also be oriented around a user, the developer. Specifically, this is done by applying techniques like Use-Cases and User Interfaces which are normally applied only to the system to its architecture. The combined result of using User-Oriented Architecture and N-Dimensional Architecture with Scaling on a system is this developers should find learning and modifying these systems easier and less time consuming.

Index Terms – *software architecture, scenarios, use-cases, software design, scaling*

I. Introduction

A. Problem and Proposed Solution

Problem:

Software Architectures are becoming increasingly sophisticated. We can observe this increase in many ways. Over the past decade, one trend we can see in our systems is the growing number of classes. Developers are building larger, more feature rich systems to meet the rising expectations of users. Developers typically find these larger systems more difficult to learn and thus they need more initial time and effort in training before they feel comfortable making changes. Another way we see this increased sophistication is in the increase in the intricacy of the designs. Architects are using powerful techniques like design patterns and architectural patterns. One of the main reasons why they use a technique like design patterns is to create systems that developers can make major

changes to with less effort and fewer problems. For example, for a system with a graphic user-interface, use of the Composition design pattern with product families allows developers to write the system for different platforms that use different windowing API's. This system could be run either on Windows using Win32 or UNIX using X-Windows. But, using a large number of these patterns in a system can dramatically increase the complexity. Developers can find these systems more difficult to change on a daily basis. Simple changes may require modifying a large number of classes[1]. To summarize the problem, the ideas in this paper are solutions to the problem of the complexity of our system architectures.

Solution:

The solution proposed in this paper has three main concepts:

1. N-Dimensional Architecture - N-Dimensional Architecture is a technique for organizing a system based on a few of the main characteristics of a system all at the same time. The result is a system architecture that is easier for developers to learn, and a system that is easier for them to change on a daily basis. The previous work that is of most relevance to N-Dimensional Architecture is Multi-Dimensional Separation of Concerns [10, 11, 12]. Both these concepts have the same basic principle of organizing systems using more than one characteristic of the system simultaneously, but the specifics on how this is done is very different. N-Dimensional Architecture is a technique that views the system as a whole; where as Multi-Dimensional Separation of Concerns is a cross-cutting technique that views cross sections of the system. The main concepts of N-Dimensional Architecture with Scaling were discussed in part 1 of this series of papers. In this paer, we delve further inot the details of N-Dimensional Architecture, describing concepts such as Scaling, Super / Sub-Architecture, and Tool-Oriented Architecture.

2. Scaling – Scaling is a secondary concept in N-Dimensional Architecture with Scaling. Scaling (or Architectural Scaling) is when an Architect creates an architecture for a system, then breaks the architecture into pieces and creates separate architectures for these pieces as well. Then, he or she can do the same for these pieces and break them up again, and create separate architectures for these even smaller pieces. And, this process can be carried on and on. Scaling is an established technique when architects create systems that work with in larger systems (scaling upwards, often called "system of systems"). But, the technique

[1] The use of Design Patterns does not always increase the complexity and size of a system. In one of the seminal works on design patterns [1], the authors advocate that a careful use of design patterns should decrease the number of classes in an architecture, creating systems that are simpler. But, in many real world systems, the use of design patterns often results in systems that are more complex and larger. This may be due to overuse.

proposed here of scaling downwards, with in a system itself has been less researched.

3. User-Oriented Architecture – User-Oriented Architecture is applying development techniques such as Use-Cases (or Scenarios), user interfaces, and Usability that are typically applied to the application as a whole, to the architecture. Within the past twenty years, developers have formally recognized that development of an application should be driven by needs of the user (Use-Cases [5] and the study of Human Computer Interface). But, the system architectures for these applications have their own users, the developers and the architects. The fundamental principle of User-Oriented Architecture is that the creation of the architecture must also be driven by the needs of its users. The previous work that is of the most relevance to User-Oriented Architecture is Scenario-Based Architectural Engineering [7]. The scope of User-Oriented Architecture entirely encompasses Scenario-Based Architectural Engineering. But, in addition to the architectural requirements and scenarios (similar to use-cases) proposed by Scenario-Based Architectural Engineering, in User-Oriented Architecture, architectural user-interfaces, architectural usability, and the concept of "architectures oriented towards architectural users" are also proposed.

The reason N-Dimensional Architecture with Scaling and User-Oriented Architecture are presented together in a series of papers (instead of two separate papers) is because these two techniques naturally complement each other. The ideas used to develop one technique were used in the development of the other (and vice-versa). If these two techniques are used together, they significantly reduce the complexity of a system.

A fourth concept that is presented is Tool-Oriented Development. Tool-Oriented Development has many of the same concepts of Component-Based Development [2]. But, the concepts of Tool-Oriented Development are built on top of the concepts of N-Dimensional Architecture with Scaling. Because of this, Tool-Oriented Development has some slight variations from Component-Based Development.

II. User-Oriented Architecture

A. User-Oriented Architecture

Before User-Oriented Architecture can be explained, we must first revisit a concept that was presented in part one, that a software-architecture has users which are the developers and architects. The developer needs to accomplish his development tasks, such as adding a new page to the GUI, and he uses the system's architecture to accomplish his tasks. The architecture provides the developer with a description of the structure of the system, allowing him to identify where he should place specific sections of code. It also provides architectural (or design-level) features. An example might be using a design abstraction

over the windowing API. This would allow a system to more easily port to different operating systems.

Since the architecture is something that has users, and these users work with the architecture to accomplish tasks. Therefore, a system-architecture is a product. It is a product similar to how a pencil or a text book is a product; it provides some service to a user. A pencil provides a service to a user by providing features that allow him to write letters and drawings on paper. A text book is a product because it provides information to a reader. The architecture is a product because it provides services that help developers accomplish their development tasks.

But, the users of the architecture must not be confused with the actual users of the system. These are two separate types of users.

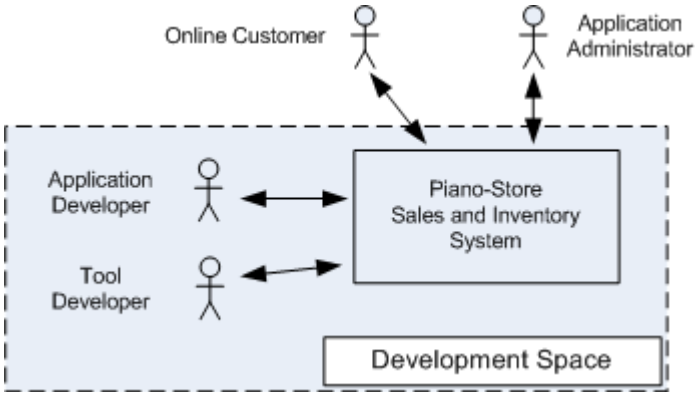

Figure 16: Diagram of the System Users and the Architectural Users.

This concept of viewing the system architecture as a product with users is the main concept behind User-Oriented Architecture. One of the main goals of the design of the system architecture is then to meet the needs of its users, the developers.

From this concept, we can then reapply some of the main concepts of systems development that are used on the application itself to the application's architecture. The architecture has user needs, those of the developers. These needs can be captured in architectural requirements (this concept has previously been proposed in [7]). They are typically captured in the form of an architectural requirements document and a set of architectural use-cases. The architectural use-cases describe how the architectural users will accomplish their various development tasks. To create the architectural requirements, the standard techniques of requirements gathering for systems can be adopted. Besides the creation of the architectural requirements document and uses cases, another important concept that is also used from system development is that of user interfaces.

In an application, user interfaces are provided to the user to allow him to complete his tasks. For the application's architecture, architectural interfaces are provided to the developers to allow them to accomplish their development tasks. These architectural interfaces take the form of the classes, subsystems, and other internal parts the system that the developer must interact with to accomplish his development tasks.

User Oriented Architecture can be used (along with Architectural Requirements and Architectural Use Cases) to create architectures that are oriented towards the developer. This is done in a fashion that is similar to how Use-Case Driven development can be used to create applications that are end-user oriented. This type is discussed further in the next section on Architectural Interfaces and, Super-Architecture and Sub-Architecture.

B. Architectural Interfaces and, Super-Architecture and Sub-Architecture

Architects and developers create Architectural Interfaces during the system design and implementation. These architectural interfaces are used together with the system's Super-Architecture and Sub-Architectures to provide the developers a way to work on the system.

As previously mentioned, each architectural interface is designed around a particular type of architectural user. All related architectural interfaces of the same abstraction level are grouped together to form a sub-architecture. This sub-architecture is a subset of the overall architecture. It should be design encapsulated from the rest of the architecture and contain only the architectural elements that are required by the users of the sub-architecture to complete their tasks.

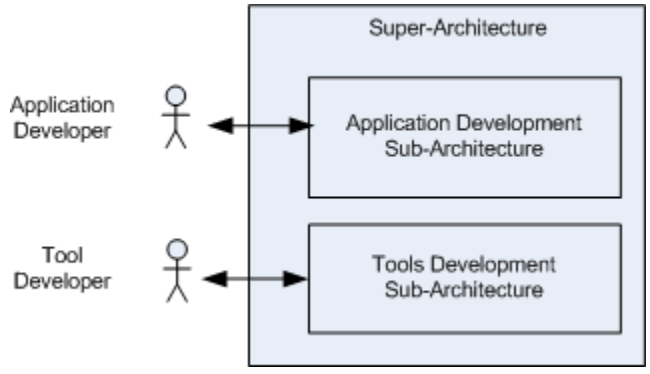

Figure 17: Super and Sub-Architecture

The Application-Development Architectural-User (or Application Developer) only needs to know his sub-architecture to complete his tasks. This sub-architecture should mostly contain classes and code that are related to his development tasks. For example, in our

piano store, let's say the application developer is working on the sales statistics page of the application. This pages lists information on the amount of sales for each month. The developer is adding a new pie chart to this page that shows the percentage of employee sales vs. online sales. The goal here is the developer should only need to navigate and change code that is relevant to this task. So, for the piano store, he would need to do something like this:

1. Create an SQL query to retrieve the employee and online sales results from the database.
2. Add the settings code for configuring the pie chart (size of chart, number of pieces of the pie, colors, titles…).
3. Add code to connect the query results and the pie chart together.
4. Add structural code to place the pie chart in the appropriate place in the sales analysis page.
5. Create the text and graphics code for the pie chart.

If we look over this list, there is no mention of making modifications to any of the structural code or the code for the tools. Ideally, for the application developer, his sub-architecture should only contain the code directly related to his tasks and all the other similar application-level tasks he performs.

Application Developer Sub-Architecture should contain:	**It should contain as little as possible:**
SQL	
GUI Code	Structural Code
Customization / Settings Code for the Tools	Tool Code
Controller-Code specific to the sub-architecture	General Controller-Code for rest of system

Figure 18: The Application Developer Sub-Architecture for the Piano Store Example

One item to note is that a sub-architecture has been defined as the grouping together of many architectural interfaces. This means that the sub-architecture itself is an architectural interface, and the architect should thinking of as an interface. It should be simple and easy to use. The idea here is similar to the idea of creating good GUI interface for applications. A good GUI should be clear and intuitive, while providing a high-level of functionality to the user. It should shield the user from the complexity of what goes on behind the scenes. To some degree, and good architecture should do the same for the developers. This is called "Architectural Usability."

The super-architecture is what the sub-architectures fit in to. The main purpose of the super-architecture is to provide a structure for the sub-architectures. The secondary purpose of the super-architecture is to provide architectural features and services for the sub-architectures to use.

One important item to note is that two sub-architectures can overlap. They may have common classes and code that developers of both sub-architectures need to access. But, the number of shared components should be kept to a minimum, and in many architectures, may not be necessary at all. The reason for this is because ownership and coordination of access to these components can become very confusing, so care should be taken when using them.

III. Details of N-Dimensional Architecture with Scaling and User-Oriented Architecture

A. Scaling, Revisited

Each sub-architecture should be fairly independent. They can be thought of as their own mini-architectures for their own mini-systems. These mini-architectures should provide the developers who work on them with the tools and the structural code that will help them complete their tasks efficiently. And, this mini-architecture should have a low learning-curve. In other words, a sub-architecture should have the same benefits as the N-Dimensional Architecture that was created for the whole system. The way this is done is by reapplying the concepts of N-Dimensional Architecture to each of the sub-architectures.

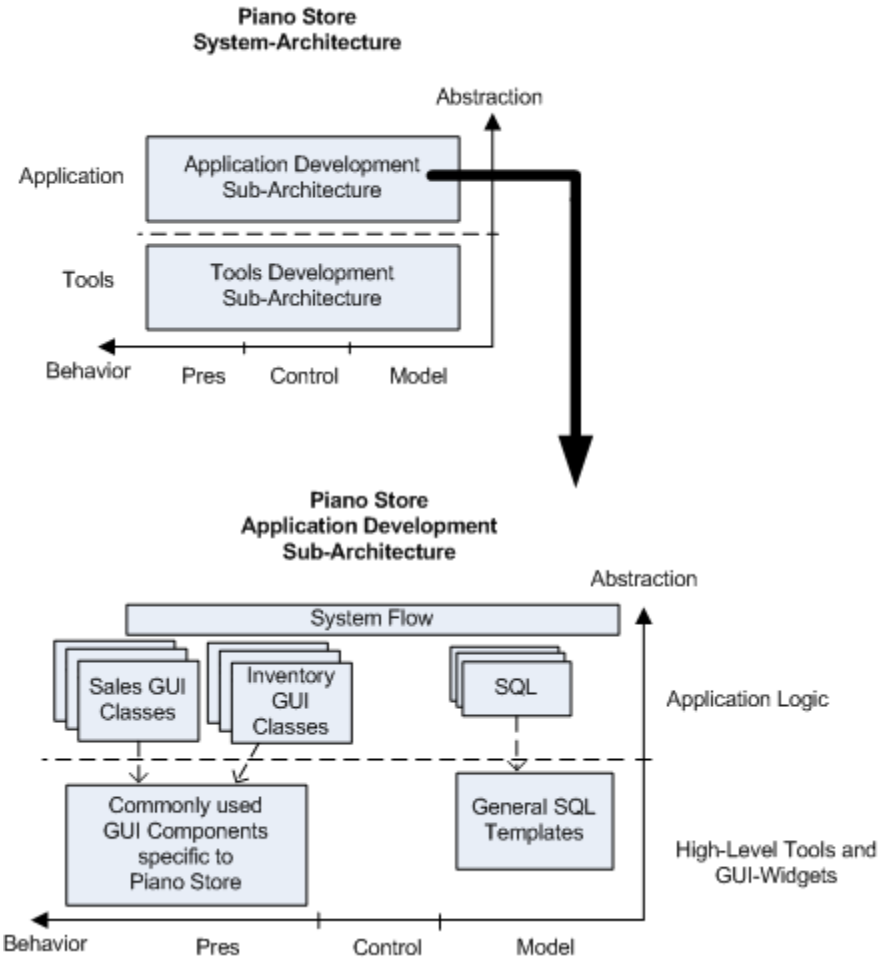

Figure 19: Illustration of Scaling – An N-Dimensional Architecture and one of its Sub-Architectures

From the diagram, we can see that we have taken just the sub architecture, and made it N-Dimensional as well. In this instance, we are only using a two-dimensional architecture. We have split it into two abstraction layers, and also organized the classes behaviorally into presentation, control, and model groupings.

We can take this even further. Components in the system should be thought of as their own mini-architectures. The classes of the components should be organized using an N-Dimensional architecture. And, we can (and should) apply this to the individual classes themselves and apply N-Dimensional concepts to the code inside the classes. The code can be organized along two axis, breaking it up into higher-level code (typically for the general flow of the code and behavioral or application code), and lower-level code (helper functions and service code).

As has been mentioned previously, this reapplication of N-Dimensional Architecture to smaller pieces of the architecture is called "scaling." The reason for this name is because we have taken a basic technique and applied it to a system. Then, we have taken the parts of the system and applied the same technique just to these parts. And then, we have broken up these parts even further, and reapplied the technique, and so on and so on[2].

It is also important to note that scaling can also go up as well. If the system being developed is a part of a larger, super-system. This super system can be organized as an N-Dimensional Architecture.

B. Tool-Oriented Development and N-Dimensional Architecture with Scaling

Before we discuss Tool-Oriented Development, a few terms must be defined. A "tool" in N-Dimensional Architecture with Scaling is some code that provides a set of functionality and is re-useable. They tend to provide clearly defined interfaces for the developers to use, generally have some documentation on its use, and often provide supporting code and tools to help developers write code to use the tool. Good examples are the numerous free Java tools that are now available, like Struts, Hibernate, Xalan, Xerces, MSXML, and JDBC. These tools are independent pieces of code that a developer can plug into their own systems and gain functionality with a relatively low learning curve. Most tools are "black box".

A component can be thought of as a piece of the system that provides some service to the rest of the system. They are neither application code, nor a full-fledged tool. Components can often be white box or black box, and can be designed for reuse, but do not have to be.

In N-Dimensional Architecture, for code that has a high degree of use by other parts of the system, it is encouraged to create a tool as opposed to a component. For a piece of code to be considered a tool, its features should be relatively flexible, allowing a decent degree of customization. It should also be highly self-contained and encapsulated, and designed with the users of the code (the developers) in mind, so that it is easy to learn and work with. It should supply supporting code or tools for the developers to help them solve any problems they might run into. Note, similar ideas have proposed in [5] with Jacobson's concept of Service Packs as well as in Component-Based Development.

This does not mean that a tool can not have some white box classes that the developers can modify. But, if there are white box classes, they need to provide a simple interface for developers to learn and use. These white box classes should still shield the developers as much as possible from rest of the internals of the tool. White-box tools are some what common in N-Dimensional Architecture. They are used in situations where the amount of

[2] This could be called "recursion," but this term is not quite accurate. Recursion is applying the same technique to different parts of a whole, and is not defined as reapplying the technique to smaller and smaller parts. Scaling is a more accurate term and is based on the definition used in Chaos Theory from mathematics. This can be visually seen in how a fractal works.

productivity gained from creating a black box tool would not be worth the time spent developing it. In addition, in N-Dimensional Architectures, we try to reduce the number of components in the system. They should mainly be used in situations where even the time spent developing a white-box tool would not be worth the amount of productivity gained.

C. Organizing Methods:

How do we transition a design to the actual code of a system?

For each axis, an organizing method is created. This organizing method defines a rule where each object in the system architecture maps to a set of folders in the source tree. Together, all the organizing methods should define a specific folder (or a limited number of folders) in the source tree where the object can be placed. For example, N-Dimensional Architecture, the abstraction axis is the main organizing method. The method should map the objects and code into different directories and major packages. For example:

Figure 20: Source Tree for Piano Store Organized by Abstraction

The organizing method of the behavior axis arranges the files and folders inside the two major packages. In each package, all the presentation code and classes are grouped together, all the control code and classes are grouped together, and all the data code and classes are grouped together.

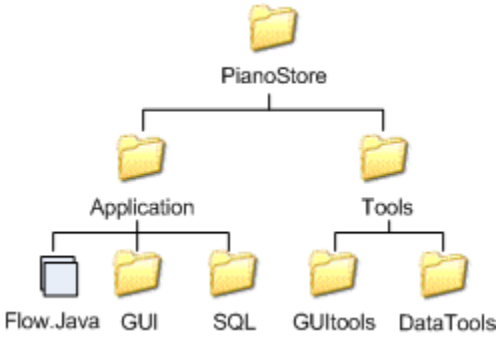

Figure 21: Source Tree for Piano Store Organized by Abstraction and Behavior

Typically, the structural organizing-method has more than one guideline for how it maps objects to code. The first guideline is there should be a high-level package for the framework of the system. The files and folders in this package are organized just like the other two high-level packages, into the three types of behavioral characteristics. The second guideline is inside the application and tool packages, a structural folder will be created. These folders will contain any structural code that is not appropriate for the higher-level framework package. But, at times, it will also be necessary to place some structural classes along with the application or tool code itself. The details for when these three guidelines require further research.

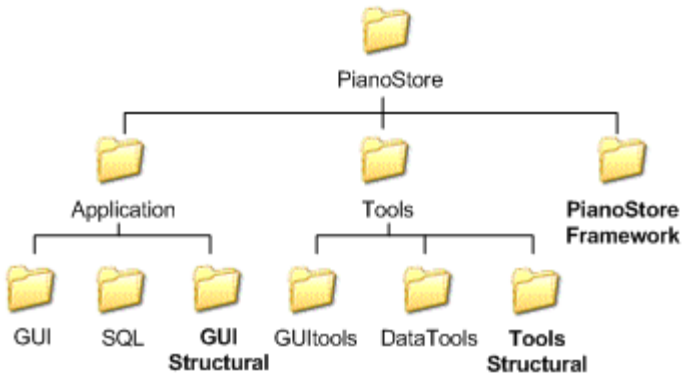

Figure 22: Source Tree for Piano Store Organized by Abstraction, Behavioral and Structural Characteristics

The result of the all mappings of the design objects should be a source tree that is organized based on the N-Dimensional Architecture that was created. The system architecture with the organizing methods is now a map to the source tree, the developers can use this map to locate their objects. If the organizing methods are properly defined, the source tree should group most of the higher-level application code together, the tool code together, and the structural classes together. And, the code in each of these three types is organized by behavior allowing the developer to easily follow the behavioral characteristics through the system (Presentation, Control, Model).

Another possibility that requires further research is the use of N-dimensional file systems (or N-dimensional source-code management systems). An N-dimensional file system would allow a developer to view his files along multiple dimensions. For example in a three-dimensional file system, the developer would see his files organized along three axes. The developer should also be able to pick a specific value for a dimension, and limit his viewing of the files to just that value. This allows the developer to view slices of the system (similar to how slices of the architecture are shown in Figure 5).

Note that this feature would allow the developer to view files in more than three dimensions. But, this may not be advisable and requires more research. Using file systems of more than three dimensions may become very confusing, especially for large systems, and in many cases, a two-dimensional file system will often work best. It might also be beneficial if these two-dimensional files systems provided a light amount of graphical-modeling features (this might look similar to the diagram in Figure 4).

D. "General" N-Dimensional Architecture with Scaling

The architecture that was presented, N-Dimensional Architecture with Scaling, was actually a specific version of N-Dimensional Architecture with Scaling. To be more specific, it was a Three-Dimensional architecture using Abstraction, Behavior and Structure as its axes. But, different architectures can be created then the one presented. One very useful alternative is to replace the "Abstraction" axis with "Degree of Change."

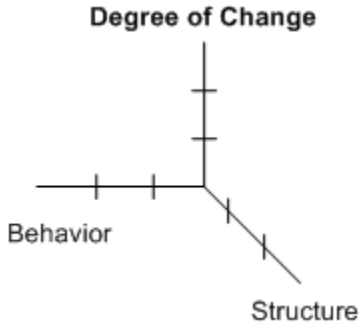

Figure 23: Three-Dimensional Architecture with a "Degree of Change" Axis

Here, classes or objects whose code are predicted to change the most are placed further up the axis, while those that are predicted to change the least are placed lower down.

In fact, any number of combinations of architectural characteristics can be used for an N-Dimensional Architecture. This concept, of being able to create any type of N-Dimensional Architecture is called "**General** N-Dimensional Architecture with Scaling."

General N-Dimensional Architecture with Scaling defines that a system-architecture should be organized using more than one characteristic of the system. Each characteristic an architect chooses becomes an axis in the N-Dimensional Architecture. Each characteristic is often orthogonal to one another in its semantic meaning.

It is important to note that although it is common to have the axes orthogonal to one another; they do not have to be. For example, we can do something like this:

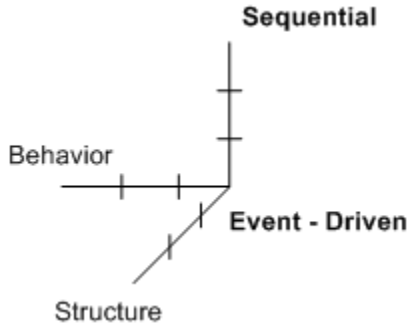

Figure 24: Three-Dimensional Architecture with a "Sequential / Event-Driven" Axis

Here, we are organizing the objects based on whether they participate in functionality that is more sequential in nature versus more event-driven. An example of objects that participate in more sequential behavior are those that provide the functionality for the purchase of a piano (1. access an account, 2. access the inventory, 3. create an order…). An example of something more event driven might be an object that provides a directory like service to large company (for instance, a phone-book web-service that is used by all the other applications in the company). The Sequential - Event-Driven axis can be described as a specific type of behavioral characteristic and is not completely orthogonal to the behavior axis.

We can also use more than three dimensions. We could create an architecture that uses four dimensions: abstraction / sequential-event / behavior / structure. And, we can even use dimensions that are more specific in how they characterize the system, such as threading, distribution, or performance. To diagram architectures with more than three dimensions, one technique is to only display cross sections of the architecture. This is done by first selecting values on one or more axes of the architecture, then diagramming the other axes (See Figure 5).

One specific axis that has interesting properties is that of "time". One way to use a time axis is to think of it as application time, and break the axis into different stages of the application's life cycle: main initialization & GUI creation; main processing; alternate processing; application destruction & cleanup. This can help to organize the objects, especially for those systems that are highly sequential, with many different stages in the application life cycle.

Another technique that should be used is that each axis of the N-dimensional system architecture should be ranked in terms of importance during a certain phase of the development process. For example, for the Three-Dimensional architecture that was presented earlier, in the analysis phase, behavior is typically the predominant axis.

During design, in some cases abstraction is the predominant axis, and in others, behavior is. During implementation, it is usually abstraction that is predominant.

During the implementation phase of development, the system architecture is used as a guide to decide what the actual code of the system should be. In N-Dimensional Architecture, we use “organizing methods“ to map the design elements to a location in the project’s source tree. These organizing methods are used to arrange the source tree to reflect the system architecture.

IV. Conclusion

In this series of two papers, we presented four main concepts that can be used by developers to solve the problem of managing the complexity of software systems. These four concepts were: N-Dimensional Architecture, Scaling, User-Oriented Architecture, and Tool-Oriented Development.

N-Dimensional Architecture is the technique of viewing software systems using many characteristics simultaneously. Software systems are very complex, and often, an architect needs a way to view and organize a system by considering more than just one of its characteristics. Behavior alone is often not enough, nor is just focusing on abstraction or just structural characteristics. Viewing a system in N dimensions results in systems whose complexity is better managed. This is because the system is organized in multiple ways at the same time. Architects and developers should find they can comprehend more of the system with less effort, in a way that is more intuitive to how they naturally think about the system.

Architectural Scaling is the repeated process of breaking an architecture into pieces and creating mini-architectures for these pieces. This technique is useful for many reasons. One of the primary reasons is because architectures have different types of users. Each type of user works with only a piece of the architecture. Through the process of scaling, we can break the architecture up and create mini-architectures organized around each type of developer. The result is each type of developer should need to work with a smaller portion of the system and should find their development tasks easier and less cumbersome to perform.

The concept of User-Oriented Architecture builds on Scaling. Its main principle is architectures are products and should be oriented around its various types of users. More over, each sub-architecture should be oriented around a specific user (or set of users). An architect does this by creating Architectural Interfaces which are created using a technique like Architectural Use Cases. The result is a sub-architecture that is highly usable while still providing all the necessary classes and objects to the developers.

The main idea of Tool-Oriented Development is most architectures can be simplified if as much of the system is moved into tools as possible. A tool has a very strong boundary, so

even more so than a component. And, it has a simpler interface, geared towards reuse. These strong boundaries and simple interfaces result in a strong separation between tool code and application code. This should decrease the number of dependencies inside a system, making the system simpler to understand and maintain.

The combined effect of using these four concepts on a system is its complexity should be better managed. Developers will require less time and effort to learn these systems, and be able to make daily changes and enhancements with more efficiency.

Related Work

Now, a list of some of the papers and books that related to the secondary ideas of the N-Dimensional Architecture will be presented. For these secondary ideas, there are many examples that are very similar each of these ideas, so only one or two major examples will be mentioned:

- Scenario-Based Architectural Engineering [7] and Role-Based Development are related to User-Oriented Development.
- Associative Databases are related to N-Dimensional File Systems with Modeling.
- Component-Oriented Development [2] and Service Packs (Jacobson) [5] are related to Tool-Oriented Development / Development.

[1] ALUR, D., CRUPI, J., MALKS, D., *Core J2EE Patterns: Best Practices and Design Strategies,* Sun Microsystems Press, 2003.

[2] GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J., *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1994.

[3] HEINEMAN, G., COUNCILL, W., Component Based Software Engineering: Putting the Pieces Together, Addison-Wesley, 2001.

[4] BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., STAL, M., SOMMERLAD, P., STAL, M., *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns,* John Wiley & Sons, 1996.

[5] FOWLER, M., "Inversion of Control Containers and the Dependency Injection pattern," *http://www.martinfowler.com/articles/injection.html*, Jan. 3, 2004.

[6] JACOBSON, I., *Object-Oriented Software Engineering: A Use Case Driven Approach,* Addison-Wesley, 1992.

[7] JACOBSON, I., BOOCH, G., RUMBAUGH, J., *The Unified Software Development Process,* Addison-Wesley, 1998.

[8] KAZMAN, R., CARRIERE, S.J., AND WOODS, S.G., "Toward a Discipline of Scenario-Based Architectural Engineering," *Annals of Software Engineering* 9, pp. 5-33, 2000.

[9] KLEPPE, A., WARMER, J., BAST, W., *MDA Explained: The Model Driven Architecture: Practice and Promise,* Addison-Wesley Professional, 2003.

[10] KICZALES, G., "Aspect-Oriented Programming," ECOOP '97: European Conference on object-oriented Programming, 1997, Invited presentation.

[11] OSSHER, H., AND TARR, P., "Multi-Dimensional Separation of Concerns and the Hyperspace Approach," *Proceedings of the Symposium on Software Architectures and Component technology: The State of the Art in Software Development,*. Kluwer 2001.

[12] OSSHER, H., AND TARR, P., "Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software," *Communications of the ACM*, Vol. 44, No.10, 2001. 43-50.

[13] SINGH, I., STEARNS, B., JOHNSON, M., *Designing Enterprise Applications with the J2EE Platform, Second Edition,* Addison-Wesley, 2002.

[14] TARR, P., OSSHER, H., AND HARRISON, W., "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proceedings of the 21st International Conference on Software Engineering,* May 1999.