Managing Complexity in Software:
Part 1 – N-Dimensional Architecture with Scaling [Software Architecture]

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

Abstract - Software architectures are becoming increasingly sophisticated. 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 systems. But, this flexibility comes at a price and the architectures for these systems can be very complex. N-Dimensional Architecture with Scaling is an attempt to solve this problem of the increased complexity of our systems while still providing a means for architects to use more powerful design-techniques. The main principle of N-Dimensional Architecture with Scaling is that architectures should be organized using more than one characteristic of the system. And, architects organize the system using these characteristics all at the same time by creating N-Dimensional Architectural diagrams. In this paper (which is part one of this series of papers on managing complexity), we present the main concepts of N-Dimensional Architecture with Scaling. In part two, we present User-Oriented Architecture and the details of N-Dimensional Architecture with Scaling. The combined result of using these two techniques in a system is that the system is easier for developers to learn and take less effort for them to change.

Index Terms – software architecture, separation of concerns, software design, multi-dimensional architecture

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. When a new developer begins to work with these larger systems, they typically find them more difficult to learn and therefore 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. More and more, architects are using powerful techniques like design patterns and architectural patterns in their systems. One of the main reasons why they use a technique like design patterns is to create systems that developers can, in the future, 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. For instance, this system could be originally implemented to run on Windows using Win32. Later on, it could be more easily ported to run on UNIX using X-Windows. But, using a large number of these patterns in a system can dramatically increase its 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 series of papers are solutions to this problem of the increasing complexity of our system architectures.

Solution:

The solution proposed in this series of papers on managing complexity in software 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 [1], [2], [3]. Both these concepts have the same basic principle of organizing systems using more than

[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 [6], 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 of design patterns.

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. In this paper (part one), we will describe the main concepts of N-Dimensional Architecture (with Scaling). In part two, we will delve more into the details.

2. Scaling – Scaling (or Architectural Scaling) is a secondary concept in N-Dimensional Architecture with Scaling. 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, breaking them up again, and creating 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 proposed here of scaling downwards, with in a system itself has been less researched. Scaling is introduced in part one and more fully explained in part two.

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 system as a whole, to the architecture. This concept will be discussed in part two.

The reason N-Dimensional Architecture with Scaling and User-Oriented Architecture are presented together, in a series of two papers, is because these two techniques naturally complement one another. To properly apply User-Oriented Architecture, it naturally should be used with N-Dimensional Architecture with Scaling. In fact, the ideas developed in one technique were used in the development of the other (and vice-versa). Used together, these two techiques reduce the complexity of a system.

B. Introductory Information on N-Dimensional Architecture with Scaling

There are many aspects of a system that an architect considers when he designs the system's architecture. When creating a system that uses N-Dimensional Architecture with Scaling, he is addressing two main considerations (as well as many others). The first main consideration is the actual code. The architect addresses this consideration in the design of the system's structure. Specifically, the architect creates descriptions of the classes, modules, and subsystems. These classes, modules, and subsystems are the containers for the code and define where the different pieces of code should be placed. Therefore, when an architect designs an architecture, he is creating (amongst other things) an organizational system. This organizational system defines where the developers should place the code they write.

The second main consideration of the architect is the developers. Developers are the people who actually use the architecture to determine where to place the code. A good architecture will not only meet the first consideration by providing a good structure for the code, but also meet this second consideration by designing an architecture that is easy for the developers to use. They should be able learn the architecture quickly, and be able to perform their tasks efficiently.

The primary goal of N-Dimensional Architecture with Scaling is to satisfy these two considerations. This architecture is a template for system architectures that meet the needs of both the code and the developers. This is done by presenting the developer with simpler, clearer interfaces into the system.

The first section of this paper presents the main concepts of N-Dimensional Architecture with Scaling using a running example of an online piano store. The second section will discuss in more detail the motivation for using the architecture.

II. MAIN CONCEPTS OF N-DIMENSIONAL ARCHITECTURE WITH SCALING

A. What is N-Dimensional Architecture?

The main concept of N-Dimensional Architecture is systems should be organized using more than one characteristic of the system.

Many architectures are organized using some form of three-tier architectural-pattern. They use architectural patterns like “Model, View, Control” (MVC); “Presentation, Business-Logic, Model” (PBM) or “Boundary, Control, Entity” (BCE) [4], [5], [5]. These architectures organize the objects of the system into three main groups with the goal of “separation of concerns.”[3] They are organizing the objects based mainly on their behavioral characteristics: for example, is this particular object a user-interface object mainly dealing with text layout or user input through GUI components? Or, does this object hold data gathered from the database?

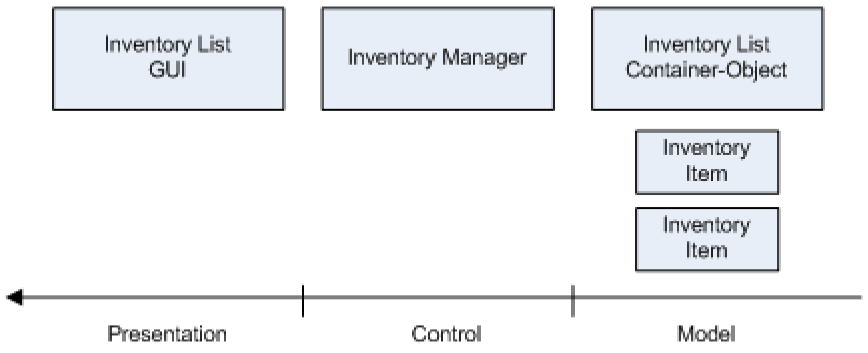

Figure 1: Example of a Three-Tier Architecture for an Inventory System.

In an N-Dimensional Architecture, objects are organized using more than one characteristic of the system. For example, objects can be organized by three characteristics at the same time such as abstraction, behavior, and structure. Using these three characteristics would result in a three-dimensional architecture.

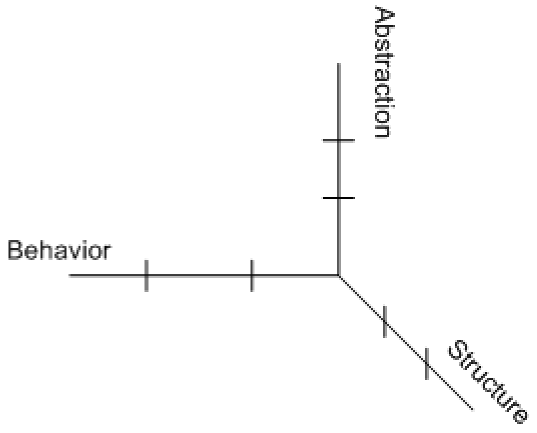

Fig. 2. Three-Dimensional Architecture with Organizing Characteristics

In the figure, each organizing characteristic is assigned to a dimension on the graph. The objects of your system are placed on the graph based on how the object's characteristics measure against the different values of the axes[2]. To explain this more thoroughly, an example of an application for a piano store will now be presented.

One-Dimensional Architecture

[2] It should be noted at this point that although N-Dimensional Architecture has many similarities to the work on Multi-Dimensional Separation of Concerns[1][2][3], the two techniques are actually very different and will be discussed later on in the section on "Related Work."

The purpose of the piano store application is to insert and display a piano store's inventory and sales data to its users: the piano store's employees. The sales and inventory information is stored in a database and is written in an object-oriented language like Java or C++. The main flow of how a user request is handled is as follows: The user opens the inventory page for the application. This request is initially handled by the GUI objects. These objects then use other objects that handle database access and manage the overall data request. These two services are provided by the Data Tool and Inventory Manager classes. A typical three-tier architecture for this application might look like this:

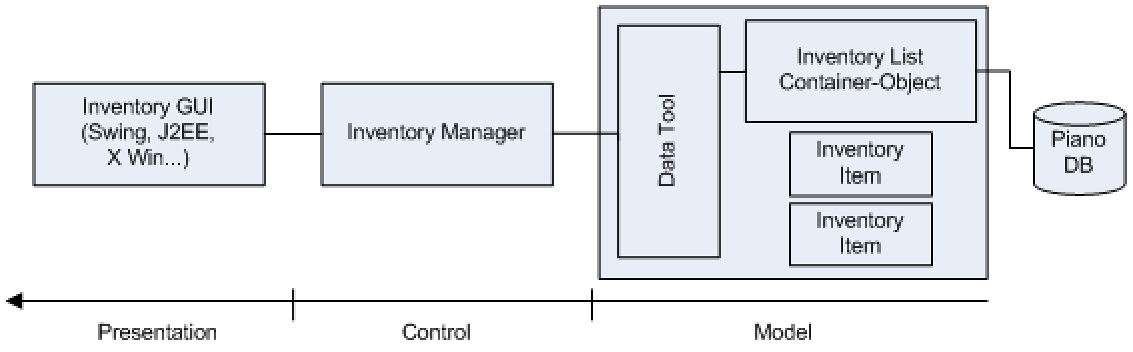

Fig. 3. One-Dimensional Architecture for the Piano Store Example

The objects for this application are organized into three groups, all presentation related objects are placed on the presentation tier, all business / control object are placed in the control tier, and all data related objects are placed in the model tier.

Two-Dimensional Architecture

Now, we will redo this architecture as an N-Dimensional architecture. First, we will create a two-dimensional version and then a three-dimensional one. For the two-dimensional version, the two organizing principles are abstraction and behavior.

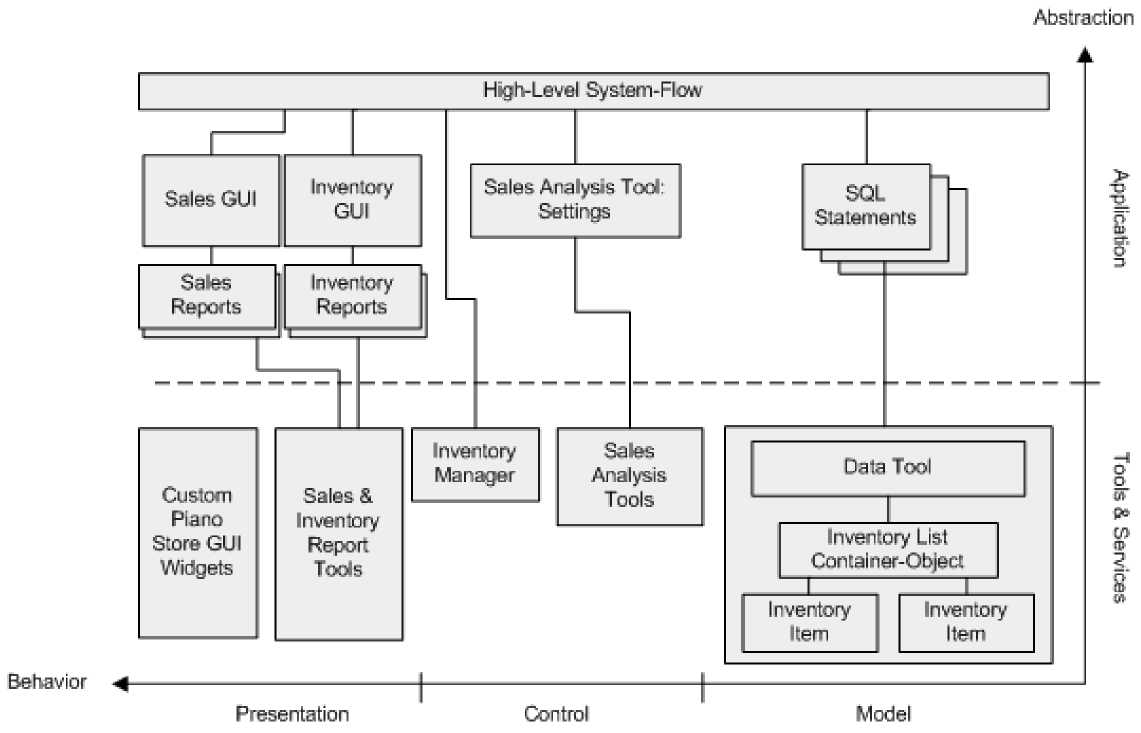

Fig. 4. Two-Dimensional Architecture for Piano Store Example

Here, the previous architecture has been divided into two main levels of abstraction: application code, and tools and services. Many more levels could be added, but, in general, 2 to 4 levels have been found to be all that are needed in a single system[3].

The way the abstraction axis should be interpreted is the more low-level (or more tool-oriented) an object is, the lower its abstraction value. It should be placed further down the axis. The more high-level (or application specific) an object is, the higher it should be placed on the axis. Similar ideas have been proposed for organizing systems by abstraction in other architectures[4]. Note that the term "tools" in N-Dimensional Architecture include more than 3rd party tools. This includes custom tools created just for the system under development.

We have mentioned the term abstraction. In software architecture, abstraction can have multiple meanings. For our discussion, this term need to be better defined, so we will now define the meaning of abstraction in N-Dimensional Architecture with Scaling.

[3] In N-Dimensional Architecture with Scaling, if the complexity of a system is becoming too great, instead of creating a new abstraction level (or abstraction layer), the architect "scales" the architecture (we will return to this later).
[4] Organizing systems by abstraction is a fairly established practice that began at least as far back as the first procedural languages. A recent of example is found in [12]. Although, it is not the exactly the same as what is presented here. The organization here makes more distinction between the different types of tools and API's. Here, the focus is more on organizing the objects based on abstraction.

There are two types of abstraction, **behavioral abstraction** and **structural abstraction**. We will discuss these two different types of abstraction:

Behavioral abstraction is organizing the parts of a system based on what each part does. Depending on how close the part is to the user, it has a higher behavioral abstraction value:

User Interface → Business Logic → Data

User Interface components are higher in behavioral abstraction than business logic components which are higher in behavioral abstraction than data components.

In **structural abstraction**, the different parts of a system are organized based on whether they are built on top of one another:

Application-Specific → Tools → Structural Code / Framework

Application-Specific code is higher in structural abstraction than the tools they use, which are higher in abstraction than the structural code for the system (such as the framework code).

In N-Dimensional Architecture with Scaling, when we refer to abstraction, we are referring to *structural abstraction*.

One last note on abstraction in N-Dimensional Architecture is that not only should the classes be moved up or down the abstraction axis based on the class's abstraction level, but the code belong to the classes should be moved up or down the axis as well. If a section of code should belong to a lower-level object, then the code should be moved out of its current object and into a lower one (or into a new object in a lower-level). This moving of the code itself has important results and will be further discussed in the "Control / Execution Architectural Pattern," "Super-Architecture / Sub-Architecture" and "Scaling" sections in this series of papers.

Three-Dimensional Architecture

The third dimension we will be using is a structure axis. This axis organizes the objects and the code by their structural properties. We will divide the axis into three structural types: Application (or Behavior), Design Features, and Structure. We will now return to our piano store example, and focus on just the DB Tool:

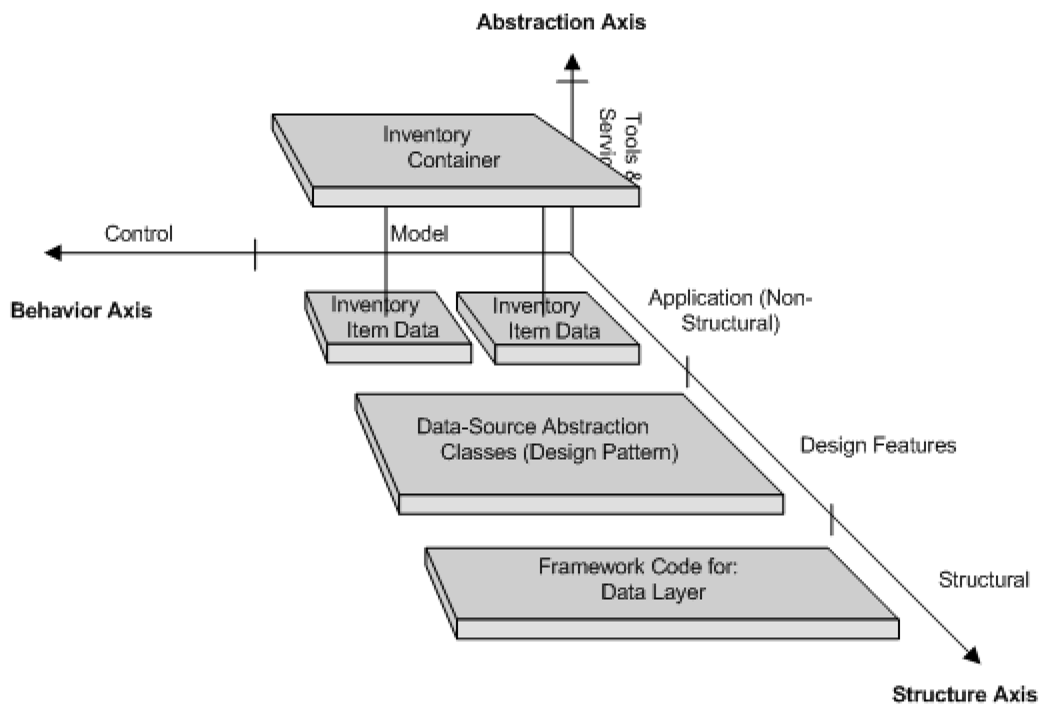

Fig. 5. Three-Dimensional Architecture for Piano Store Example: Structure Axis for the Data Tool

The first structural type, application, is for objects / code that have a close relationship to the problem domain, or code that is directly related to providing the outward behavior of the system. For instance in the DB tool, these are the classes that contain the code for the algorithm that determines how the data from the database should be assembled together into objects. The code for this algorithm is very closely related to providing the outward behavior of DB tool. But, on the other hand, the application structural-type does not include objects like those devoted to providing design-related features. For instance, those involved in the Factory pattern for the logging tool to provide an abstraction for creating instances of the tool.

The second type, design-level features, is for code and objects that provide design-level functionality. A good example is the classes and interfaces used to abstract the data sources. For our example, in our piano store application, we are using a relational DB for our data source. But, let's say there is a future requirement that XML data from a Web Service from piano vendors may also be used. We need to allow for the relatively easy addition of new types of data sources in the future. We can implement this using some form of design abstraction, possibly using the "Composition" design-pattern with product families [6]. The ability to add new data types is not a code-level feature, meaning this behavior is not one that can be used when we run the system. It is a design-level feature that the developers can use mainly by modifying the classes of the system. This feature is mostly implemented by the choice of classes and relationships, with possibly some smaller pieces of code.

The third type, structural, is for code and objects that are involved in purely structural behavior. These classes are usually apart of a framework, defining the various layers of the architecture, or containers for other parts of the system.

The general purpose behind this separation by structural type is that these three types of code tend to be mixed together. For example, a developer who is interested in adding a new application enhancement like a customer account page is not interested in modifying the structural code. He should not have to change it to accomplish his development task.

B. What are some of the main principles behind N-Dimensional Architecture?

When an architect designs a system that has an N-Dimensional Architecture, one goal he or she should keep in mind is the creation of a system architecture that follows the **Control / Execution Architectural Pattern**. A typical example of the pattern from the world of computer can be found in the design of a CPU. In CPU's, there are two major parts: the control and the execution. The execution piece has the parts of the chip that do things like mathematical operations (add, subtract, divide, multiply), branching, and memory access. The execution piece is the part of the CPU that actually does the work. The control piece has higher level logic that oversees the execution piece. It contains the logic for decision-making, determining when and what the execution piece is to perform. When an architect creates an N-Dimensional Architecture with Scaling, the resulting architecture should follow this same Control / Execution pattern[5]. This pattern will be discussed in more detail through out the rest of the paper.

C. Scaling

To complete this overview of N-Dimensional Architecture with Scaling, we will discuss one last major principle. We will explain the technique of Scaling. Once an architect has created an N-Dimensional architecture, he "scales" the architecture by reapplying the same concepts of N-Dimensional Architecture to a smaller part of the system. How an architect would scale a system is as follows:

> The architect creates an N-Dimensional Architecture for the system. The result is an architecture with two major layers, an application layer and a tools layer. For each layer, we can now reapply the same concepts we used for the design of the overall system. We think of each layer like it is its own independent system. Then, we can create an N-Dimensional Architecture for each layer. This means we draw a separate graph with the three axes, and put the elements of the layer on the graph.

This may be more easily understood by using a diagram:

[5] The Control / Execution architectural pattern has similarities to the Controller design pattern, but it also has some differences, one of which is the Control / Execution pattern is an architectural pattern. This will be further discussed in the "Control / Execution Architectural Pattern" section of this paper.

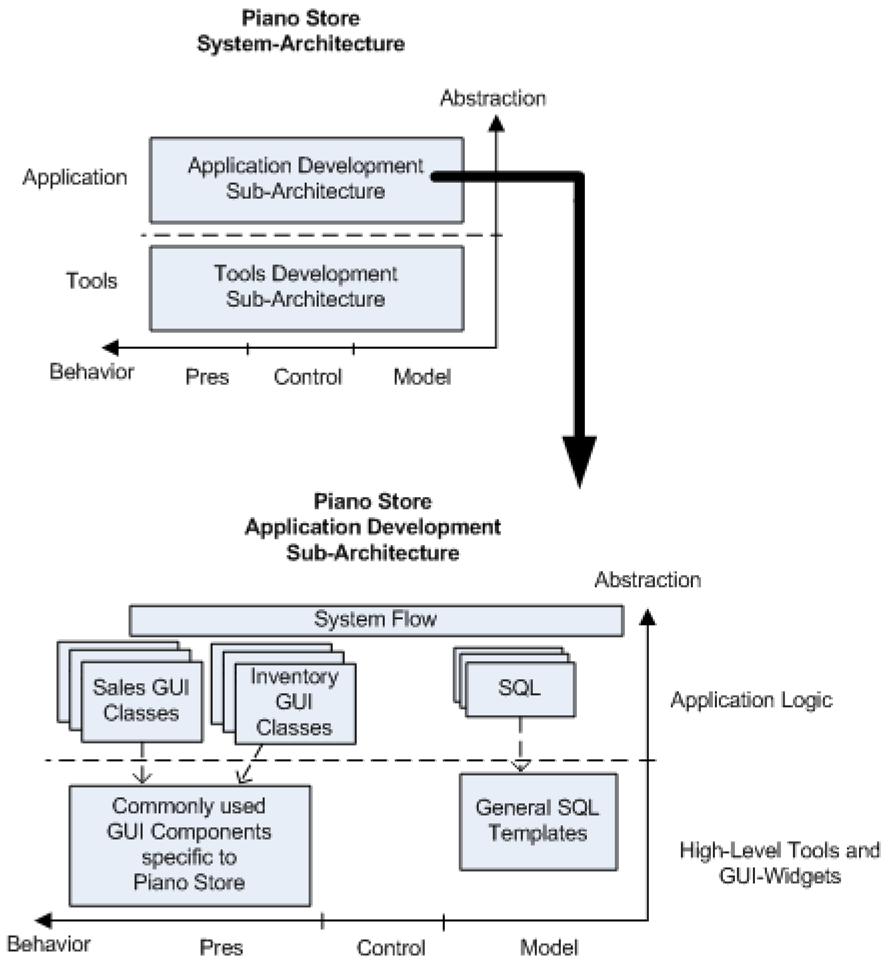

Fig. 6. An Example of Scaling for the Piano Store Example

Here, we have taken the 2-dimensional architecture that we created for the piano store (see Figure 4) and "scaled" one of its layers. We have taken the higher-level layer that has the application code and created a separate mini-architecture for this layer.

The reason we scale an architecture is because software is by nature hierarchical. And, to manage the complexity of large software systems, we need to create a hierarchy of architectures to best organize these systems. Scaling will be discussed in more detail in part two of this series of papers.

III. MOTIVATION FOR N-DIMENSIONAL ARCHITECTURE WITH SCALING: Comparing N-Dimensional Architecture with Scaling versus 3-Tier Architecture

To understand how N-Dimensional Architecture with Scaling better manages complexity than today's architectures, we will need to make a comparison. We will briefly compare an architecture from the real world (specifically, 3-Tier) with a version that uses N-Dimensional Architecture with Scaling. First, we will discuss what a typical, real world architecture looks like. Then, we will apply this real world architecture to our piano store example. Then, lastly, we will re-do this system as an N-Dimensional Architecture and analyze the new architecture's benefits.

Please note: In this section, we only present a brief comparison between the two architectures. If you are interested in seeing the full comparison[6], please refer to our supplemental material [7].

A. What are the basic elements of a typical architecture from the real world?

Many of today's architectures make use of these two techniques**:**

1. Some form of 3-Tier Architectural Pattern (such as MVC, PBM…)
2. Design Patterns

Of course, many other techniques are used besides these two. And, many other architectural patterns are used besides 3-Tier. But, a large proportion of new systems use both of these techniques to some degree.

In this paper, we will try to remain as platform independent as possible. But, to present an example that will be useful to our discussion, we need an architecture that is frequently used in the real world. So, for our example architecture, we will use the architecture and the design patterns of the Java 2 Enterprise Edition (J2EE) from the Sun Java Center. We will use these two books [8], [9], from the Sun Java Center, as our source for the information on J2EE architecture and design patterns. The authors of these books present the information clearly and understandably. These books can be found on the Sun Java Center website.

[6] In this section of the paper, we present a the sample 3-Tier architecture and an N-Dimensional version. In the supplemental material, these two architectures are more fully explained and analyzed. We explain in more depth how a typical, three-tier, J2EE architecture works and also provide a great deal more supporting information. We also describe in greater detail how a typical development task that might be performed on both these systems.

The Piano Store Example

We will now apply the J2EE architecture to the piano store example. This is best seen with an architectural diagram.

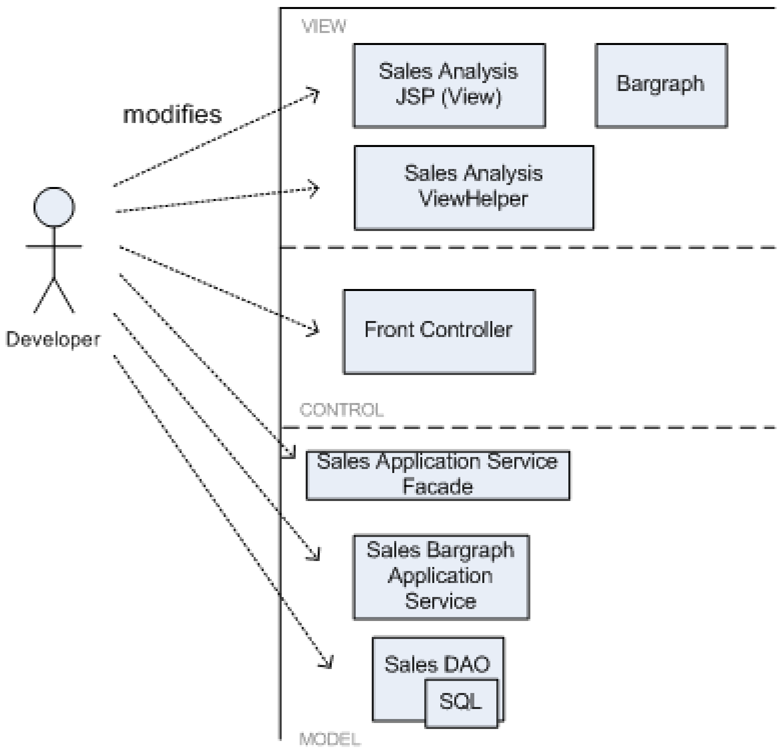
VIEW
Sales Analysis JSP (View)
Bargraph
modifies
Sales Analysis ViewHelper
Developer
Front Controller
CONTROL
Sales Application Service Facade
Sales Bargraph Application Service
Sales DAO
SQL
MODEL

Fig. 7. The Piano Store Example as a J2EE Architecture

B. Analysis of the 3-Tier, J2EE Architecture:

Benefits of J2EE Architecture and 3-Tier Architecture

The benefits of a three-tier architecture like the J2EE architecture have been discussed a great deal in numerous other publications. So, we will only briefly mention a few of the benefits that relates to our discussion. One of the greatest strengths of this architecture is the way the objects are organized into three distinct groups. This decreases the complexity of a large, GUI system tremendously, as the different types of classes and code are much less tangled together. This allows a developer new to the system to more easily learn its architecture. This is because the system is more easily comprehensible from a high-level. When this developer needs to make a change to the code, a better organized system is easier for a developer to navigate. It will take them less time and effort to find the sections of code they need to modify.

Another major benefit is it reduces the amount of coupling and dependencies within the system. A simple change in the GUI is less likely to require a change in the business logic and data objects. This is because of the two strong boundaries that have been set up between the tiers.

And, ideally, there should be weak dependencies between these classes that have a relationship between the tiers. This decreases the chances of a change rippling through the system.

Problems of J2EE Architecture and 3-Tier Architecture

One of the main principles of 3-Tier architectures is that presentation objects change more than control objects, which change more than business objects. *This is a general belief of three-tier architectures and is not a correct assumption.* In many applications, the business objects change the most, in others, it is the presentation objects, and in others, it is the control objects that change the most. The degree of change of an object tends to depend more on the object's relationship to the user and business needs than to its type of behavior. For instance, in an application that does mainly batch processing, the business logic will change more than the presentation. An example of this might be an application to process the automotive insurance-claims for an insurance company. The interface for this application simply controls the starting, stopping, and monitoring of the processing of the insurance claims. For this application, during development, its interface will probably change little compared to the classes in the other tiers.

The bulk of the application is in its business rules. For example, an insurance application such as this would have a rule for determining which automotive insurance-claims should be reviewed by the company's claims analysts. More specifically, on a claim where there is suspicious activity during an accident, this claim would be sent to an analyst for review. And, the application might have another business rule that complements this one for claims which have no problems. For these claims with no problems, payment on damages can be immediately sent out to the customer. For example, on a small claim from a long-term customer who has never missed a payment and has no previous accident history, this person's claim would be immediately paid out. In applications like these that are mainly rules, the developers will probably spend the most time changing the business logic.

Moreover, not only do the lower tiers sometimes change more than the upper tiers, many developer tasks require modifications in two or three tiers[7]. For instance, in our piano store example, if the developer needs to modify one of the bar-graphs that shows the daily sales figures, he would need to make major modifications to classes in all three tiers. These changes can be seen in Figure 7.

As we can see from Figure 7, the developer must modify a large number of classes to make this change. And, in order for the developer to know which classes to modify and which classes he can leave untouched, he will need to have a high-level understanding of even a larger number of classes. He will be navigating his way through a great number of classes, and will be required to have an understanding of a large portion of the system. This is another problem of three tier architectures. Developers often need to "manage" a large amount of "complexity" in order to make a relatively routine change.

[7] To better understand why, please refer to the supplemental material [7].

C. What is the Solution?

Now, we will re-do the J2EE architecture of the piano store as an N-Dimensional Architecture:

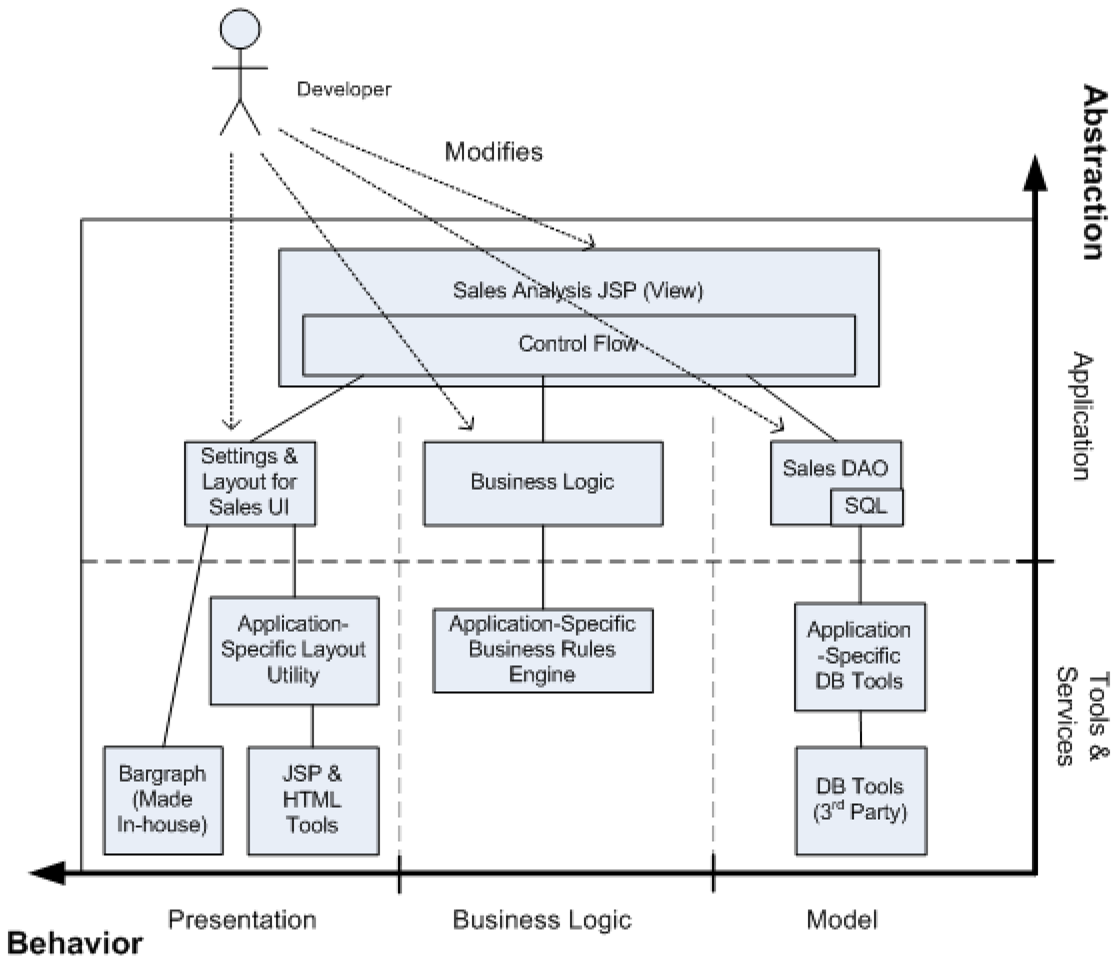

Fig. 8. The 2-Dimensional Version of the Piano Store Application

Here, we have created a two dimensional version of the Piano Store application. Notice how the architecture is now organized by both abstraction and behavior (three-tier).

How does this N-Dimensional Architecture with Scaling solve many of the problems of three-tier architectures?

N-Dimensional Architecture with Scaling solves many of the problems of three-tier architectures because developers will take less time and effort to a make change that requires modifications in presentation, business logic, and model classes. And, it better manages the complexity of the architecture by better organizing the different parts of the system.

What we notice when we look at the N-dimensional version of the system is that classes that deal with the outside behavior of the application are placed higher up. Developers that make application specific-changes (as opposed to changes in the system's structure or its tools) will have easier access to the code they need. More specifically, they will have better access to code like the high-level control-flow of the application, the business rules, the GUI layout and settings code, and the SQL. These types of code have been grouped closer together. They now have a strong boundary between their code, and the tools they use. More specifically, a strong boundary now exists between their abstraction layer and the abstraction layer of the tools. There is still a boundary that exists between Presentation, Business Logic, and Data, but it is weaker than what is found in three-tier architecture. The goal is to organize the system primarily by abstraction while preserving the three-tier characteristics of the system. As can be seen in the diagram, we are still able to follow the behavioral characteristics (such as Presentation or Business Logic) through the system.

Now, let us say the developer is performing the same change we discussed before in the three-tier example, the change of making modifications to the bargraph that displays daily sales figures. If he were to do make this change on the two-dimensional version, he would need to modify the following classes as seen in Figure 8.

If we compare the modifications in Figure 8 with those of the 3-Tier J2EE architecture in Figure 7, we observe that in N-Dimensional version, the developer is working with a smaller portion of the system. He needs to make changes in a select number of classes all of which are in the same layer of the system. In this two-dimensional version, many of the classes have been moved down to lower layers. Developers who work mainly with the application-specific code are better shielded from the parts of the system that they do not need to interact with. They are working with mainly the classes that are directly related to their development tasks.

D. How does N-Dimensional Architecture with Scaling reduce code-tangling?

Through the use of N-Dimensional Architecture, we reduce abstraction code-tangling, behavioral code-tangling, and structural code tangling at the same time. Architectures that use N-Dimensional Architecture are separating the more application-specific code that changes more, from the code that should be built upon and that changes less. Typically, this separation is best done as application-specific code vs. tools / services. At the same time, this architecture is still separating the different types of behavioral code from one another[8].

[8] Please note: In this section, we only present a brief explanation on how N-Dimensional with Scaling reduces code-tangling. If you are interested in seeing the explanation, please refer to our supplemental material [7].

IV. PRINCIPLES OF N-DIMENSIONAL ARCHITECTURE WITH SCALING

A. Control-Execution Architectural Pattern (Revisited)

The Control-Execution Architectural Pattern is a fundamental pattern found to some degree in most major systems (software and non-software). To discuss this pattern, we will now return to the example mentioned in the earlier part of in this paper on the design of a CPU. Typically, a CPU is split into two major pieces, the Execution piece and the Control piece. The Execution piece is responsible for performing the actual work of the CPU. It performs the arithmetic operations like add, subtract, multiply and divide and also handles branching. The Control piece provides the higher-level behavior of overseeing the execution piece. It determines what should happen in the execution piece and when. Control is the brain while execution is the body[9].

When we create a system that is uses N-Dimensional Architecture with Scaling, the result is that the system follows the Control-Execution pattern:

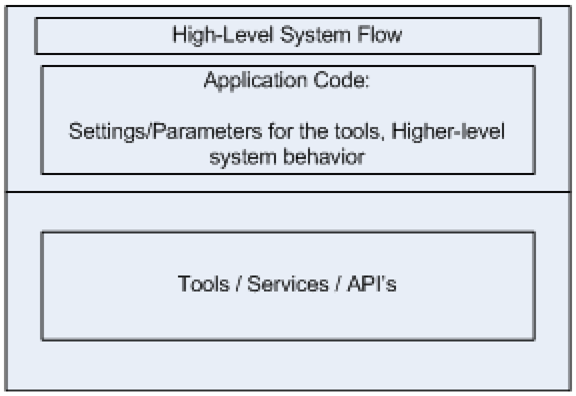
High-Level System Flow
Application Code:
Settings/Parameters for the tools, Higher-level system behavior
Tools / Services / API's

Figure 9: General Organization of an Architecture that was designed using the Control / Execution Pattern

At the highest level is the "flow code." This code is the highest-level method calls for the system. Typically, this code contains the system initialization sequence, the high-level sequence of method calls for the application running state, and the system destruction sequence. This flow code is a high level map of what the system does, and can greatly ease the learning curve for new developers. Developers can use the flow code to easily identify the objects and components that

[9] The Control-Execution Pattern has been used in many of engineering for years, at least as far back as the first century B.C. The great Greek inventor, Heron, used a control unit for his "Automatic Theater." The Automatic Theater was a small theater that showed little figures that were automated using mechanical means. Little ships and people would move across that stage and sound effects would play through offstage devices, telling a story. The sequence of events that took place was controlled by a device that had a rotating drum and strings with weights. The strings were set at different positions on the drum. These different positions determined when the stage pieces would perform actions – an effective, centralized control-unit.

are created, the configurations of these objects, and the sequence of events that occur when the system is running.

Below this is the application-behavior code, or the application logic. This part of the system should only contain code that is specifically related to the outward behavior. This is the code that the application developer (mentioned in section III, D, Figure 13) will spend most of his time making his changes and enhancements to. The flow code plus the application code is considered the Control portion of the pattern. This higher-level code defines what is going to happen and when. The tool code is considered the Execution portion of the pattern. It takes care of doing most of the work for the system[10].

There is a space in the Control-Execution pattern to handle objects that are not quite tools and are not quite application code. This space is called the Components Layer for "white-box" components. Classes that have a mixture of application code and tool code are placed here. The number objects in this layer should be minimized as much as possible and, often, should not necessary at all. But, sometimes there is a need. One situation is when a certain feature will not be changed frequently. It would not be an efficient use of the developer's effort to break this component into application and tool parts. In this case, it is acceptable to create a white-box component. These components are usually placed between the application and tool layers in N-Dimensional diagrams.

One of the primary goals of using the Control-Execution pattern is to minimize the number of classes you need to touch in order to carry out a change. The breaking of the system into the two main layers, application and tools, should result in a reduction in the number of classes needed by developers to make their changes. Another related goal is to create a "single point of development" for each type of developer. Developers should only need to access a small portion of the system to carry out their developer tasks. For instance, by using the Control-Execution pattern, the architect should be creating a "centralized control" for the system, where most of task performed by the application developer should take place.

The Controller pattern [6] is highly related to the Control-Execution pattern. The general mechanics of how these two work are generally the same. They both try to centralize the control of the system into a one or two classes, providing the flow for a portion of the system. The main difference is the Control-Execution pattern is more of a general design philosophy with guidelines for organizing an entire system (or a large piece of a system). And, in addition, for this pattern, the control has been clearly separated from the parts of the system that does the work. A strong boundary is created between the two. On the other hand, the Controller pattern tends to work on a smaller scale. It is a design pattern and does not provide guidelines for organizing the entire system. It operates more at the class level where the Control-Execution pattern operates at the architectural level.

[10] Note that the flow code can combine with the application code. The code for many of the object settings are placed in the flow code. This can occur if a system is very sequential in nature and has a full set of tools and services. These systems would then be the flow code combined with application code (a sequence of object and tool creation with their settings, then method calls for the running state), and then below this, the tool code.

B. Combining principles from Structured Programming with Object-Oriented Programming

When a system is created that applies the concepts of N-Dimensional Architecture with Scaling, one of the results is the architecture is organized using principles from both Structured Programming / Design and Object-Oriented Design. One of the key principles of Structured Programming is the creation of architectures that are organized by abstraction. In N-Dimensional Architecture with Scaling, this is done by organizing the system using the abstraction axis. The result of using this axis on a system is that this system has a top-down structure. The benefit of this organization is the system can more easily be understood at a higher level. Large systems are difficult to understand as a whole because of the great number of classes, components, and relationship. Having them organized from the top down allows a developer to see the high-level processing of the system very quickly by viewing the Flow code. And, typically, he will only need to navigate through one or two classes away from this flow code to get to the major components and subsystems of the system.

In Object-Oriented Programming, two of its key principles are the use of objects to encapsulate the details and the use of architectural patterns. In N-Dimensional Architecture with Scaling, architects organize their systems with an architectural pattern like MVC or some other three-tier architecture using the behavior axis. As we mentioned previously, there are numerous benefits to creating three-tier architectures. The clean organization of objects into three types has the benefit of decreasing the dependencies within the system and easing the understanding of the architecture by developers. In N-Dimensional Architecture with Scaling, these benefits are still there. Developers are still creating the three groups by using the behavioral axis.

Therefore, N-Dimensional Architecture with Scaling combines principles from both Structured Programming / Design and Object-Oriented Design, gaining benefits from both.

V. CONCLUSION

In this paper, we presented N-Dimensional Architecture with Scaling. This architecture can be broken into two separate techniques: N-Dimensional Architecture and Scaling.

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 handled, and therefore, developers will find these system easier for them to understand. 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 software systems.

Architectural Scaling is the repeated process of breaking an architecture into pieces and creating mini-architectures for these pieces. This technique is useful because system architectures have different types of users, each of which work with only a piece of the system. 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 (this technique will be discussed further in part two of this series of papers).

The combined effect of using these two concepts on a system is the system should be better organized, and 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.

VI. FURTHER RESEARCH

In this paper (part one), we discussed the main concepts of N-Dimensional Architecture with Scaling. In part two, we will present some new concepts related to N-Dimensional Architecture with Scaling and delve into the details. More specifically, we will introduce User-Oriented Architecture and Tool-Oriented Development, and discuss Scaling in further detail.

In addition, one concept that was not mentioned in this series of two papers but is used in N-Dimensional Architecture with Scaling is that of "Sub-Applications". This technique is used to encapsulate the elements of the application’s core behavior in to its own separate part the system. The basic ideas is to create a version of the system that uses little or no higher functions or tools (no graphical interface, no object communications like CORBA, or no code to handle input devices like sensors). This concept will be further explained in forthcoming papers.

VII. RELATED WORK

A. Work Related to the Main Ideas of N-Dimensional Architecture with Scaling

There are a number of research efforts that are related to N-Dimensional Architecture with Scaling. The two with the most similarities are: Multi-Dimensional Separation of Concerns (MSDOC) [3] and Aspect Oriented Programming (AOP) [10]. The first similarity amongst all three is all these solutions are used to help solve the problem of code tangling.

The main similarity between Multi-Dimensional Separation of Concerns and N-Dimensional Architecture with Scaling is that both try to solve the problem of code tangling by viewing the system using more than one characteristic of the system. And, both view the system using all these characteristics *simultaneously*. Both even have multi-dimensional graphs (but use them in different ways). The main differences are:

1. MSDOC is a non-invasive approach. The power and simplicity of MSDOC is that developers are able to view any slice of the system they are interested in. These slices in themselves do not directly effect the organization of the system. They can be considered a specific view of the system.

   On the other hand, N-Dimensional Architecture has both non-invasive and invasive aspects. The non-invasive aspect is the N-Dimensional diagram. This diagram does not need to have any direct effect on the system and can simply be a view of the system. The invasive aspect is the idea that the developers should use this diagram to directly organize the system using all N dimensions all at the same time. And, more over, the developers should divide up the system and apply a similar N-Dimensional organizing scheme to these parts as well (scaling).

2. The focus of N-Dimensional Architecture is to create a view of the whole system in one N-Dimensional diagram (although, developers will more often only use a particular view of the diagram, such as a two dimensional version instead of the full 3 or 4 dimensional version). MSDOC, on the other hand, is a technique to separate the architecture into many different slices. Its focus is on creating a cross-section of the architecture that is of interest to a developer.

Another related work to N-Dimensional Architecture with Scaling is Aspect-Oriented Programming. Like MSDOC, Aspect Oriented Programming is also a cross cutting technique. The focus is to pull individual concerns out of the system into separate code bases. As mentioned in the previous paragraphs, N-Dimensional Architecture with Scaling does not focus on cross-cutting the system. Therefore, these two are only related at a high level.

Another powerful concept that is related to N-Dimensional Architecture is Model Driven Architecture (MDA) [11]. This concept has fewer similarities than MSDOC, but has one important one. In MDA, the system is also organized by its abstraction characteristics. A system is split into its higher-level analysis-objects (which are platform independent) and lower level-design objects (which are platform dependent). But, this form of abstraction is of a different type than the type used in N-Dimensional Architecture. In MDA, objects and code are organized by their design-element's abstraction characteristics. In N-Dimensional Architecture with Scaling, they are organized by both their design and code abstraction characteristics.

B. Work Related to the Secondary Ideas of N-Dimensional Architecture with Scaling

Now, a list of some of the concepts that are 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, so only one or two major examples will be mentioned:

- Controller Pattern [6], Inversion of Control (IOC), and Dependency Injection Pattern (DIP) [12] are related to the Control / Execution Pattern.
- The Unified Software Development Process' package organization-scheme [13] is related to organizing systems by Abstraction.

VIII. REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

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

[7] JOH, S., "Managing Complexity in Software: Part 1 – N-Dimensional Architecture with Scaling – Supplemental Material", ****ADD URL TO MATERIAL**** 2005.

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

[9] SINGH, I., STEARNS, B., JOHNSON, M., *Designing Enterprise Applications with the J2EE Platform, Second Edition,* Addison-Wesley, 2002, Ch. 11 and pp. 360-361.

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

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

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

[13] JACOBSON, I., BOOCH, G., RUMBAUGH, J., *The Unified Software Development Process*, Addison-Wesley, 1998, pp 234-240.