Peter Joh
Chief Architect
Project Hierarchy
peter@projecthierarchy.org
June 12th, 2014
Doc V0.02

An Overview of N-Dimensional Architecture

Collecting All the Content and Settings for an Application into a Matrix – Or what we'd like to call "N-Dimensional Architecture"

This is one of our personal favorite usages of matrices, and was one of the two inspirations for creating them. The basic idea is that we can pull all the high-level settings out of a system, and put them in their own layer, in to matrices. This is very similar to what happens when you use Spring and Dependency Injection to collect all the settings for your objects and put them into xml files, but takes these ideas much further, as you'll soon see (by the way, Spring and Dependency Injection were actually not inspirations for this architecture, it was just a convergence of similar ideas).

But, before we discuss NDA, let's discuss some of the existing architectural-concepts that NDA builds upon. If you have a decent understanding of MVC / 3-tier, this discussion should be fairly easy to follow.

For most business applications, we tend to use MVC or 3-tier style architectures. When we use this type of architecture, we are favoring the *behavioral* characteristics of a system, with less emphasis on its *abstraction* characteristics. This last statement may sound kind of vague, so let's break this down further.

When we use an MVC type architecture, the major questions we ask ourselves when designing a system are: *Is this particular object a model object? Or is it a view object?* We are placing the objects into architectural buckets each of which has a different **behavioral** characteristic of the system: *If this object deals with the user interface, then it's most likely a view object, so put it with the other view objects for you system. If the object deals with accessing the database, it's probably a model object, put it with the other model objects.*

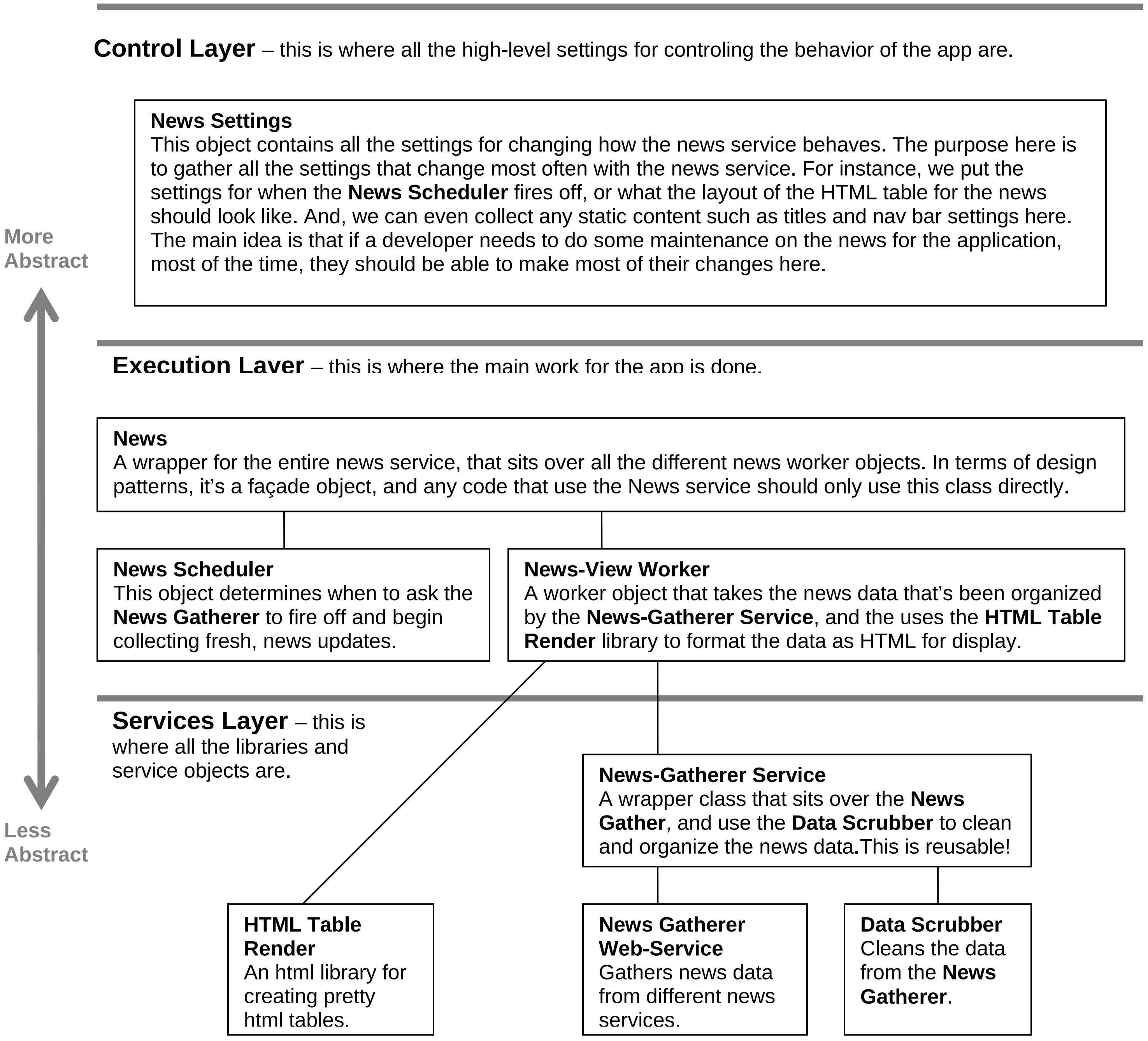
This is an extremely useful way of organizing a system, and the benefits of MVC to the discipline of software design cannot be overstated. But, we must remember that *abstraction* is also a very important way that we organize our systems. In fact, if we think about a computer, abstraction may be the most important way: The lowest layer of a computer is the hardware, with its different collection of chips and silicon to perform computations, graphics, and I/O. Then, above this, the hardware runs a mini operating-system, which is some form of BIOS that does the initial booting up of the different elements of the hardware. The BIOS also provides low-level API's that the main operating system can use to access the hardware. And, above this layer is the operating system itself, which provides many different services for starting and running applications. And, finally, above this are the applications, which use the API's of the operating system.

Most of us intuitively organize our own systems using abstraction as well, but we often don't make clear distinctions between the different layers. Typically, when we create a system, we ask ourselves questions like: *Is this object a low-level service that belongs in its own library? Or, does it belong at a level just above this, because it's a wrapper class that joins a bunch of different services together? Or, is it at an even higher level, using all the different service-objects to do the main work for the system?* When one organizes systems using these characteristics, one is organizing the system *by abstraction*.

But, we developers shouldn't limit ourselves to using *just* abstraction or *just* behavior (MVC) in our system architectures. We should consciously be organizing our systems by both at the same time! We call this type of architecture where you are organizing a system using multiple characteristics *simultaneously* an **N-Dimensional Architecture (NDA)**.

To see a NDA in action, let’s design architecture for a sample, news-feed application. The way this news-feed application works is: based on a schedule, it periodically starts up and collects articles from different sites on the web. It then cleans the ads and pulls out the content, and then displays it to the user in an html table.

First, we’ll start designing this system by organizing the different objects *by abstraction* (by the way, the following diagram may look a little daunting, but have no fear. We find it’s easiest to just start by reading the description of the objects at the bottom layer and working your way up. It’s actually not too bad):



Control Layer – this is where all the high-level settings for controling the behavior of the app are.
News Settings
This object contains all the settings for changing how the news service behaves. The purpose here is to gather all the settings that change most often with the news service. For instance, we put the settings for when the **News Scheduler** fires off, or what the layout of the HTML table for the news should look like. And, we can even collect any static content such as titles and nav bar settings here. The main idea is that if a developer needs to do some maintenance on the news for the application, most of the time, they should be able to make most of their changes here.
More Abstract
Execution Layer – this is where the main work for the app is done.
News
A wrapper for the entire news service, that sits over all the different news worker objects. In terms of design patterns, it’s a façade object, and any code that use the News service should only use this class directly.
News Scheduler
This object determines when to ask the **News Gatherer** to fire off and begin collecting fresh, news updates.
News-View Worker
A worker object that takes the news data that’s been organized by the **News-Gatherer Service**, and the uses the **HTML Table Render** library to format the data as HTML for display.
Services Layer – this is where all the libraries and service objects are.
News-Gatherer Service
A wrapper class that sits over the **News Gather**, and use the **Data Scrubber** to clean and organize the news data.This is reusable!
Less Abstract
HTML Table Render
An html library for creating pretty html tables.
News Gatherer Web-Service
Gathers news data from different news services.
Data Scrubber
Cleans the data from the **News Gatherer**.

**Typically easiest to read this diagram from the bottom up.*

Here, we've organized the system into three main layers. Let's describe each one (we'll start from the bottom layer as this lowest layer is typically the most familiar to most devs):

1. **Services Layer** – Basically, this layer contains our reusable libraries and web-services. This code is more *general purpose* and reusable through out the entire system (and possibly other systems as well). Examples of code in the Services Layer for our sample news-feed application would be: An in-house API for connect to the different media-site web-services, or a custom HTML-library for transforming the news articles grabbed from the different sites, converting them into a standard data-format.

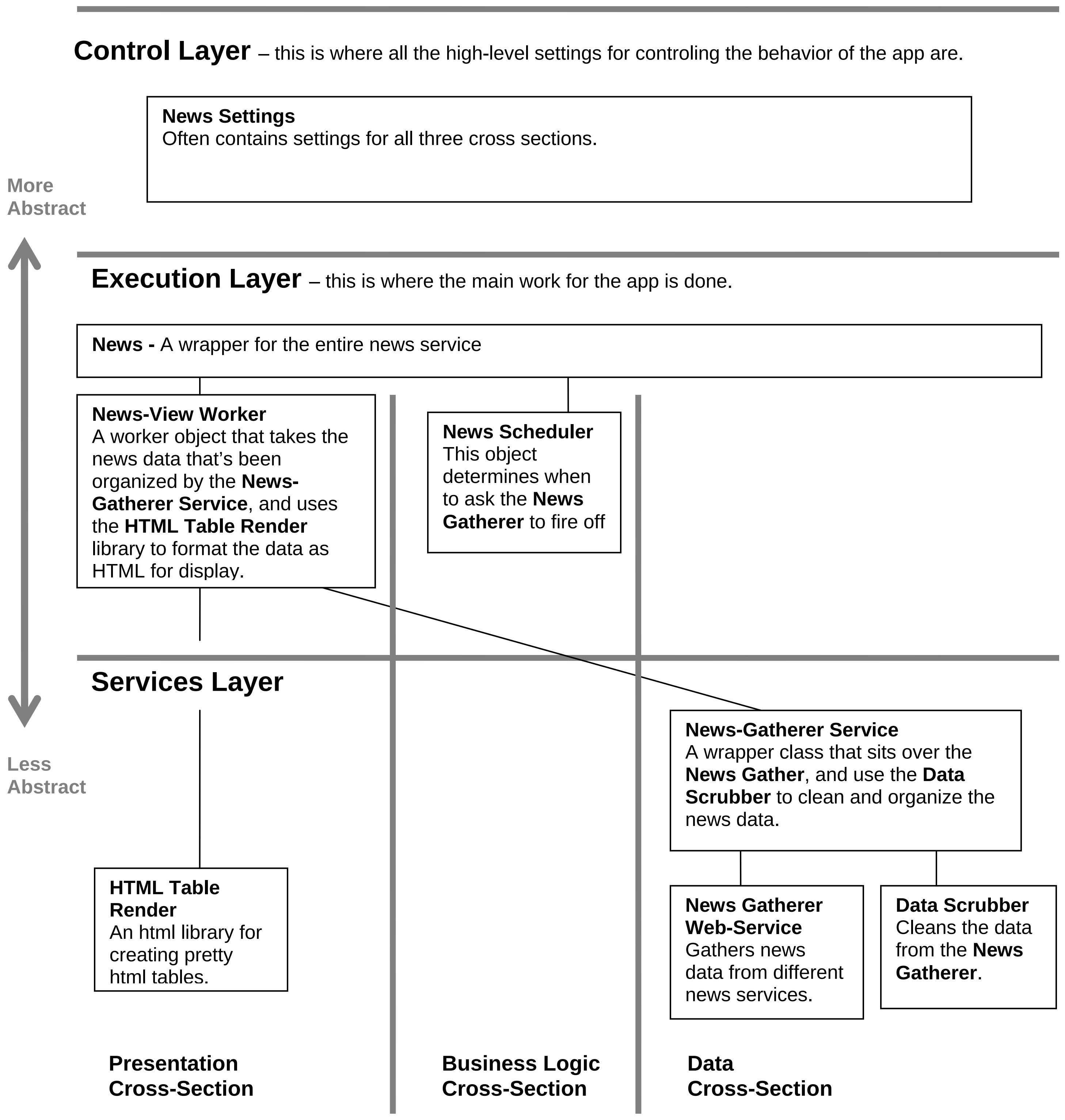
2. **Execution Layer** – This is where the main work for the application is done. The Execution Layer *uses* the tools provided by the Services Layer to perform the features of the system. For our news-feed application, a couple examples would be the `NewsScheduler` component and the `NewsViewWorker`. These components handle the most important work for the application, and this layer is thought of as the "body" of our application.

   But, there is another, even more important way to think about the Execution Layer. It is where the work *specific* to the system is performed. In the Services Layer, we placed our more *general*-purpose code there, like home-grown networking libraries or a custom data-scrubber API. But *here*, in the Execution Layer, we place the code that does the work that is *specific* just to this system.

3. **Control Layer** – In this highest-level layer, we pull out all the **settings** for how system behaves and all the **content** that is displayed to a user, grabbing this from throughout the entire application (*whatever* layer it is in). We put all these settings and content info together, placing them in high-level data-objects. For instance, for our news-feed application, some of the settings data we would grab would be: the settings for how to connect to each type of news website (web-service vs. JSON) and what data scrubber to use (XML vs. HTML documents), and the actual polling schedule for when the application should check the different websites/webservices for new articles.

   We'll talk more about the Control Layer later on.

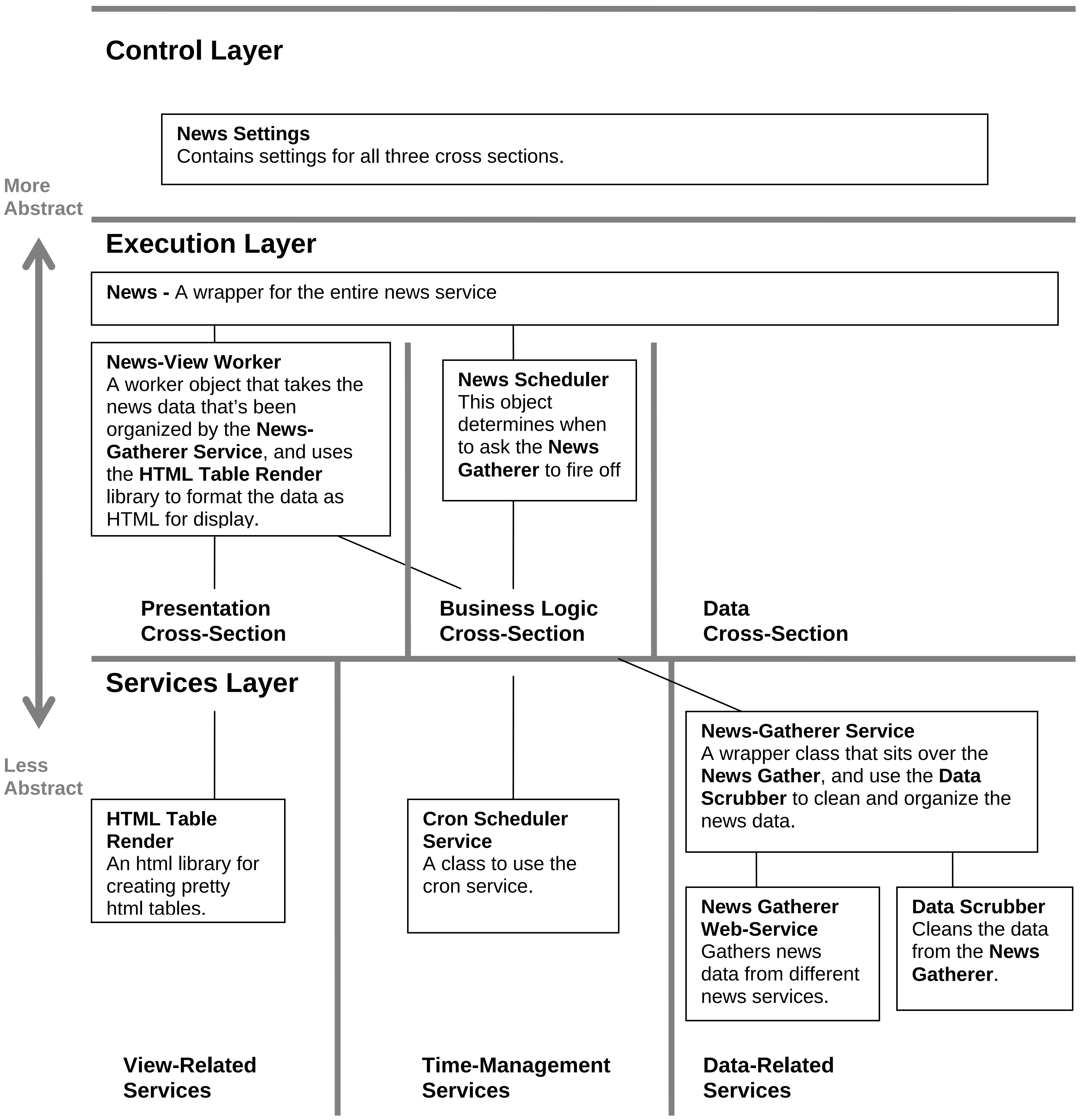
Organizing a system by abstraction is a very natural way of ordering a system. It's something we do intuitively, but should also be consciously done every time a developer designs a system. But, note, we are not replacing or throwing out MVC. In fact, the above architecture is *not* an NDA architecture yet. For that, we need to consider *both abstraction and behavior at the same time*. Here's the above system reorganized by both abstraction and behavior:


Control Layer – this is where all the high-level settings for controling the behavior of the app are.
News Settings
Often contains settings for all three cross sections.
More Abstract
Execution Layer – this is where the main work for the app is done.
News - A wrapper for the entire news service
News-View Worker
A worker object that takes the news data that's been organized by the News-Gatherer Service, and uses the HTML Table Render library to format the data as HTML for display.
News Scheduler
This object determines when to ask the News Gatherer to fire off
Services Layer
Less Abstract
News-Gatherer Service
A wrapper class that sits over the News Gather, and use the Data Scrubber to clean and organize the news data.
HTML Table Render
An html library for creating pretty html tables.
News Gatherer Web-Service
Gathers news data from different news services.
Data Scrubber
Cleans the data from the News Gatherer.
Presentation Cross-Section
Business Logic Cross-Section
Data Cross-Section

Here, we've created another axis to order our system by. The vertical axis is still abstraction, but the new, horizontal axis orders the system *by behavior*, using an MVC style architecture. Notice that we are understanding the system two ways at the same time, by both abstraction and by behavior.

As you can see in this diagram, objects don't need to live in just one cross-section. Some objects span multiple or all cross sections. A good example is the **News** object that wraps all the other worker objects, or the **News Setting** object that contains settings for many different behaviors of the system.

And, we don't have to have the same horizontal axis for each layer. Each layer can have its own. In fact, it might make more sense to organize the system this way:



Control Layer
News Settings
Contains settings for all three cross sections.
More Abstract
Execution Layer
News - A wrapper for the entire news service
News-View Worker
A worker object that takes the news data that's been organized by the News-Gatherer Service, and uses the HTML Table Render library to format the data as HTML for display.
News Scheduler
This object determines when to ask the News Gatherer to fire off
Presentation Cross-Section
Business Logic Cross-Section
Data Cross-Section
Services Layer
Less Abstract
HTML Table Render
An html library for creating pretty html tables.
Cron Scheduler Service
A class to use the cron service.
News-Gatherer Service
A wrapper class that sits over the News Gather, and use the Data Scrubber to clean and organize the news data.
News Gatherer Web-Service
Gathers news data from different news services.
Data Scrubber
Cleans the data from the News Gatherer.
View-Related Services
Time-Management Services
Data-Related Services

As we mentioned, one of the most important layers is the upper-most one, the **Control Layer**. To talk further about it, the idea is to put all the high-level info of our system *that changes the most* here. This way, devs will be able to look in one spot to do a majority of their changes, instead of having to search through the multiple parts of the system. An example of an object in the Control Layer for our news-feed application would be a data-source settings object. This object would be used to collect the different configurations for how to access each different news-website that the `NewsGathererService` connects to. Some settings that it might contain would be: the URL for the site and what format the articles were in (they could be XML generated from a news webservice, not just HTML). Now, anytime we had to connect to a new web site, a new data-source settings object could be quickly created for it.

But, moreover, for this new data-source, we would not only define the connection info, but all the other info relevant info for our system to work with this new website. For instance, the developer would probably need to create some type of data-scrubber to transform the news articles from the website's format to our own, internal data-format. Often in most systems, a dev would have to move to an entirely different part of the system to configure another component. But with an NDA architecture, we place the data-scrubber's config *with* its connection configuration. *And then,* on top of this, we also place all the view-type settings for this website here too, such as the settings for the html layout when a news article is displayed to our user. Unlike MVC architectures, we are grouping model, control and view settings together:

```
public class NewsDataSourceSettings {

   public NewsDataSourceSettings(String URL, DataSourceType dataSourceType,
     DataScrubberType dataScrubberType, String dataScrubberTemplateFilename,
     DocumentDisplayLayout documentDisplayLayout,
     String documentBorderColor){

     //  (Constructor code left out for brevity, but it's just assignment to
     //   the various members of the object: ex. this.URL = URL; )
   }

   //  CONNECTION SETTINGS:
   public String URL;
   public DataSourceType dataSourceType; // (see enum definition below)

   //  DATA-SCRUBBER SETTINGS
   public DataScrubberType dataScrubberType;  // (see enum definition below)
   public String dataScrubberTemplateFilename; // a template used by the
                                               // datascrubber to pull the content

    //  HTML DISPLAY SETTINGS
    public DocumentDisplayLayout documentDisplayLayout; // (see enum below)
    public String documentBorderColor;
}



//  This enum defines the different types of online news-feeds our
//  application can connect to
public enum DataSourceType {
    HTML, WEBSERVICE, JSON
}
```

```
    //  This enum contains the names of the different types datascrubber
    //  objects that are available to us.
    public enum DataScrubberType {
        //  A custom library that was internally created that can be feed a
        //  template to understand the layout of HTML websites (what is the
        //  content area, nav bar, title bar…)
        TemplateBasedHtmlWebsiteScrubber,

        //  This next scrubber library uses a neural net to learn the pattern
        //  of the html layout to figure out the different areas.
        NeuralNetHtmlWebsiteScrubber,

        //  This data scrubber cleans xml docs from webservices
        WebserviceXmlScrubber
    }

    //  This enum defines the different types layouts an article can be
    //  displayed in to our user
    public enum DocumentDisplayLayout {
        OneColumnFixedWidth, OneColumnVariableWidth,
        TwoColumnFixedWidth, TwoColumnVariableWidth
    }
```

Now, to actually define some settings objects, a dev would new instances of the objects, populating it with settings data:

```
    //  This is are the connections to the nytimes website.
    NewsDataSourceSettings appSettings = new NewsDataSourceSettings(
       "http://www.nytimes.com", DataSourceType.HTML,
       DataScrubberType.TemplateBasedHtmlWebsiteScrubber, "nytimesTemplate.txt",
       DocumentDisplayLayout.OneColumnVariableWidth, "#FFFFFF");
```

What this boils down to is that when a dev has to add a new website to our system, he will often have to work with one or two objects to make a majority of his changes. These settings object become the primary source for his development task. And, the *overall* end-result is that the Control Layer becomes a *control panel* for the entire system, making it easy and efficient for devs to modify the application. If the Execution Layer is the body, the Control Layer is the brain, telling the body what to do.

…This might seem sacrilegious to most of us who have grown up on MVC to place model, view and controller settings together, but we have built many architectures using NDA, and it seems like the settings is the part of the system where religiously splitting things up by MVC is the least helpful. What we have found is that this control panel for the system acts basically as a user interface to the system architecture, where the users of this architecture are the developers. And, just like designing any other type of user interface, architects shouldn't organize them by the internal structure of the system, but by the *natural way a developer works* with the system.

As an analogy, if we were to create a product page for a website interface like Amazon.com, we wouldn't split up the different parts of the page across multiple sections of the website. Meaning that if a user is researching say the book, "How to Base Jump for Dummies," we wouldn't make the user, first, do a search for the product description of the book in the *Product* section of the site, and then, when the user wants to read the reviews have him redo his search in the *User-Review* section of the site, and lastly, when he looks up the price have him move again this time to the *Product-Pricing* section, redoing his search in this last section of the site. We would try to group the information he needs together on a single page that is well laid-out and easy to understand.

To talk more generally about the overall principles of designing *any* user interface (like for designing GUI's), it's probably safe to say, most *graphical* user-interfaces should *not* be organized by the internal structure of the system, but by principles of **Usability**. Usability is the study of "the ease of use and learnability of a human-made object." And, one of the main principles of Usability is *to design the interface around the tasks performed by the user*.

So, for the "*development* interface" to our news-feed system (meaning our settings objects[1]), these objects should also display their information to the user based on the ease-of-use of the *tasks performed by the devs* – more specifically, the tasks of maintaining and enhancing the system. And we just did this, as you just saw in our news-feed application. In our `NewsDataSourceSettings` object, instead of placing our connection settings for a news website in the config objects, then our data scrubber settings for the articles in a separate datascrubberConfig objects, and lastly our html display settings in another, separate viewSettings objects, we created one settings-object that contained all three! This object is now the primary source for a dev to perform this maintenance tasks for each data-source!

Also, it's important to note that the idea is *not* to be rigid about putting everything for a dev task together, this can sometimes create settings objects that are huge and unwieldy. The idea is to group the settings as *logically and intuitively for the devs as possible!* Sometimes that means have multiple settings objects (very typical), sometimes that means having just one large one.

How does Hierarchy fit into NDA?

For the implementation of the settings object in our news-feed application example, we used regular Java objects. But, actually, Hierarchy's matrices are a better fit for holding this type information (this was one of the main inspirations for why Hierarchy matrices were created). They hold large sets of different type of data very easily and are easy to access from inside your Java programs.

To see this in action, let's redo the news feed setting objects as a matrix:

```
MATRIX NewsDataSourceSettings USES (NewsDataSourceSchema){

   CONNECTION.SETTINGS: { “http://www.nytimes.com”, :ConnectionType.Html };

   DATASCRUBBER.SETTINGS: { :TemplateBasedHtmlWebsiteScrubber,
      “config\nytimes\templates\nytimesTemplate.txt” };

  HTMLDISPLAY.SETTINGS: { :OneColumnVariableWidth, :BorderColor.Blue };
}
```

[1] The development interface also includes the front-most facing class definitions of the libraries most commonly used by the devs!

Pretty simple. And, here's the `NewsDataSourceSchema` schema defintion:

```
SCHEMA NewsDataSourceSchema{

   DESCRIPTOR +:%CONNECTION.SETTINGS {
      FIELD.NAMES: { +:%URL, +:%ConnectionType };
      FIELD.TYPES: { :"String", :Symbol };
   }
   DESCRIPTOR +:%DATASCRUBBER.SETTINGS {
      FIELD.NAMES: { +:%DataScrubberType, +:%DataScrubberTemplate.FilePath};
      FIELD.TYPES: { :Symbol, :"String" };
   }
   DESCRIPTOR +:%HTMLDISPLAY.SETTINGS {
      FIELD.NAMES: { +:%DocumentDisplayLayout, +:%DocumentBorderColor };
      FIELD.TYPES: { :Symbol, :Symbol };
   }
}
```

As you can see, matrices are an extremely efficient way to collect all the settings of your system together. But, before we move on, let's take a look at a real world example of an NDA architecture. We'll take a look at a simplified version of the matrix used in the unconventionalthinking.com website. First! Open up the Unconventional Thinking website in your browser (http://www.unconventionalthinking.com). Compare the information in the website as you read through the matrices in the following example. As we just mentioned, the website was created with heavy usage of matrices and should give you a look at using N-Dimensional Architecture in the real world:

```
MATRIX Unconventional.Content USES (com.unconventional.matrix.j2ee::Web.Content)  {

    `Home` {
        PAGE.INFO: { "home", 0, -1, "Home",
          "We build good software. We develop software products to help improve our world. We truly enjoy
             developing software that enhances your quality of life.",
                              "","", false };
        NEWS {

            NEWS.STORY: {"Hierarchy beta release",
                "The Hierarchy beta is ready for you to try!",
                "June 6<sup>th</sup>, 2011",
                 "After years of development, the beta version of the Hierarchy Meta-Compiler for Java is ready
                    for developers to try out!."
            };

            NEWS.STORY: {"N-Dimensional Architecture",
                "Research and Development on N-Dimensional Architecture Yields New Meta-Compiler for Java",
                "December 1<sup>st</sup>, 2009",
                 "We have been researching theories on system architecture for over four years and are nearly
                    ready to show the results of our countless days working late into the night...”
            };
        }
    }

   `Products`  {
        PAGE.INFO: { "products", 1, -1, "Products",
          "We truly enjoy creating good software. Many applications are slow, crash or have poorly designed
             features. Bad applications make users sad. Building software that is powerful and easy to use, and
             reliable and responsive takes a lot of hard work, but we feel the greatest satisfaction in seeing
             our users happy. ",
           "","", false};

        NEWS {
            NEWS.STORY: {
                "Matrix", "Hierarchy for Matrix-Programming ", null,
                "We are excited to finally let users try what we've been working on for the past 4 years: " +
                "the Hierarchy Beta is ready for Java developers to try out!..."
            };
        }

    }
```

```
`Contact`  {
     PAGE.INFO: { "contactus", 5, -1, "Contact Us",
          "Please feel free to contact us",
          "","", false};


     // { +:Person_ID, +:Name, +:Title, +:Role, +:Email, +:Description }
     PEOPLE {
         PERSON.INFO: {"info", "Information", null, null, "info@unconventionalthinking.net",
                 "If you like to speak with someone at Unconventional Thinking..."
         };
         PERSON.INFO: {"sales", "Sales", null, null, "sales@unconventionalthinking.net",
                 "If you like to speak with someone from Sales, please feel free to send an email here."
         };
         PERSON.INFO: {"careers",
                 "Careers",
                 null, null,
                 "careers@unconventionalthinking.net",
                 "If you interested in a career here at Unconventional Thinking..."
         };
         PERSON.INFO: {"support",
                 "Support", null, null, "support@unconventionalthinking.net",
                 "If you have a support question on one of our products..."
         };
         PERSON.INFO: {"location",
                 "Location", null, null, null,
                 "Unconventional Thinking is located in beautiful Ann Arbor, Michigan..."
         };
     }
 }
```

Unconventional$__$Content.matrix

In this matrix, we've collected all the content for the site. Again, if you haven't done so already, take a look at the real, unconventionalthinking.com website and compare the generated html with this matrix. You should see where different matrix elements line up with the generated content.

This next file is the jsp file that uses this matrix. Jsp files that have embedded matrix code use the **.mjsp file extension!** So, this file is called index.**mjsp**.

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@page import="MATRIX com.unconventional.web::Unconventional.Content"%>


<%

   //  This next method call looks at the request objects parameters and determines which page we're on. Then,
   //  it grabs the corresponding descriptor for the page from the Unconventional.Content matrix
   DESCRIPTOR<Unconventional.Content->ITEM> curr_PageItemDesc = determineCurrPageItemDescriptor(request);
%>

<table cellpadding=0 cellspacing=0 border=0 bordercolor=black width=532>
<tr>
    <td ><div style="width:500px">

        <font style='font-size:15px;font-weight:bold'>
              <img src="images/subtitle_<%=currPageInfo_Desc:>Name_NoSpaces%>.gif"
                  alt="<%=currPageInfo_Desc:>Title%>"><br>
        </font>
        <div style="margin-top:17px">
            <img src="images/blurb_<%=currPageInfo_Desc:>Name_NoSpaces%>.gif"
                alt="<%=escapeHTML(currPageInfo_Desc:>BlurbText)%>"
                style="margin-top:3px" border="0">
        </div>

        <% if (curr_PageItemDesc.label == `Home`) { %>
            <div style="margin-top:47px; ">
                <img src="images/subsubtitle_news.gif" alt="News"><br>
                <div style="margin-top:25px">

                    <%
                     for (DESCRIPTOR<Unconventional.Content->ITEM->NEWS->NEWS.STORY> newsItemDesc :
                           curr_PageItemDesc->NEWS->NEWS.STORY{*}) {
                    %>
                       <%=newsItemDesc:>Title%><br>
                       <%=newsItemDesc:>DateText%><br>
                       <%=newsItemDesc:>Story_Blurb%><br>
                    <% } %>

                </div>
            </div>
```

```
<% } else if (curr_PageItemDesc.label == `Products`) { %>
    <div style="margin-top:62px">
        <div style="margin-top:12px">

            <%
             for (DESCRIPTOR<com.unconventional.web::Unconventional.Content->ITEM->NEWS->NEWS.STORY>
                newsItemDesc : curr_PageItemDesc->NEWS->NEWS.STORY{*}) {
            %>
                <%=newsItemDesc:>Title%><br>
                <%=newsItemDesc:>Story_Blurb%><br>
            <% } %>

        </div>

    </div>



<% } else if (curr_PageItemDesc.label == `Contact`) { %>
    <div style="margin-top:34px; width:340px">
        <table cellpadding=0 cellspacing=0 border=0 bordercolor=black>


            <%
             for (DESCRIPTOR<Unconventional.Content->ITEM->PEOPLE->PERSON.INFO> personInfoDesc :
                  curr_PageItemDesc->PEOPLE->PERSON.INFO{*}) {
            %>
                <tr>
                   <td valign=top>
                       <img src="images/contact_title_<%=personInfoDesc:>Person_ID%>.gif"
                           alt="<%=personInfoDesc:>Name%>"><br>
                               <a class="main-body-email" href="mailto:<%=personInfoDesc:>Email%>">
                                 <%=personInfoDesc:>Email%></a>
                           </div>
                       <%}%>
                   </td>
                   <td width=100% valign=top></td>
                </tr>
                <tr>
                   <td colspan="2">
                       <div>
                           <%=personInfoDesc:>Description%>
                       </div>
                   </td>
                </tr>
```

```
                    <% } %>

                </table>
           </div>

        <% } %>



    </div>
    </td>
</tr>
</table>
```

index.mjsp

Here, in the unconventionalthinking.com website, we're pulling out all the content and settings, and placing them into the `Unconventional.Content` matrix. Then, we use this matrix to generate all the information for the site in the `index.mjsp` page. This is a typical usage of N-Dimensional architecture to separate elements from your system that change the most with those that change the least.

And, for those that are interested, on the next page is a simplified version of the schema used by this matrix.

```
package com.unconventional.matrix.j2ee;

SCHEMA Web.Content {

      DESCRIPTOR +:%PAGE.INFO {

            FIELD.NAMES: { +:%Name_NoSpaces, +:%Page_ID, +:%Page_childID, +:%Title, +:%BlurbText, +:%LNav_Href,
                           +:%LNav_Image, +:%LNav_IsActive, +:%ContentAreaFormat };

            FIELD.DESC: { "Name of the Page (no spaces)", "The page id (int)",
                          "The child page id (optional)(int)", "Page Title",
                          "Short, descriptive text about the page", "The left nav Href",
                          "The left nav image name", "The left nav, is active field",
                          "The formatting of the content area. Symbol values are: :Normal and :Wide" };

            FIELD.TYPES: { :"String", :"int", :"int", :"String", :"String", :"String", :"String",
                           :"boolean", :Symbol };
            FIELD.DEFAULTS:   { null, null, -1, null, null, null, null, true, +:Normal };
      }

      DESCRIPTOR +:%NEWS {
        DESCRIPTOR +:%NEWS.STORY {
            FIELD.NAMES: { +:%NewsItem_ID, +:%Title, +:%DateText, +:%Story_Blurb, +:%Story_Text };
            FIELD.DESC: { "Id of news item", "News item description", "Date", "The short blurb about the story",
                          "The text of the story." };
            FIELD.TYPES: { :"String", :"String", :"String", :"String", :"String"};
            FIELD.DEFAULTS:   { null, null, null, null, null };
        }
      }

      DESCRIPTOR +:%PEOPLE {
        DESCRIPTOR +:%PERSON.INFO {
            FIELD.NAMES: { +:%Person_ID, +:%Name, +:%Title, +:%Role, +:%Email, +:%Description };
            FIELD.TYPES: { :"String", :"String", :"String", :"String", :"String", :"String"};
            FIELD.DEFAULTS: { null, null, null, null, null, null };
        }
      }

}
```

Web$__$Content.schema

As we keep mentioning, putting all the content and settings for the entire website together makes it easy to maintain the site. And to see this in action for this site, the most common change we (the maintainers of the unconventionalthinking.com & projecthierarcy.org websites) find we have to perform is modifying or adding new content. It is a very simple task for us to do so with matrices: simply find the correct location to put or modify the content and add it in. We find for most of our maintenance tasks on these sites, we have to make almost all our changes in the matrices, with the occasional need to modify one or two other files.

In addition, we also find that we've had to write much less code for the same subsystems as compared to traditional MVC style systems. We think this may have to do with the fact that a lot of the organizational structure of MVC puts a lot of small amounts of information in lower level classes when they really belong at a higher level.

Now, to wrap things up, let's go into more detail about the benefits of N-Dimensional Architecture.

Benefits of N-Dimensional Architecture

Developer-Task Automation

As we looked at the code for the Unconventional Thinking website, we see that the creation of many of the elements in the website have been *automated.* What we mean by automation is, for example, how the news event info that's displayed on the home page for the site is dynamically generated from the matrix data (as opposed to being hard coded). So that now, if we need to add another news story to the home page, we simply need to add a new NEWS.STORY descriptor to the matrix, and the code for the site will then dynamically generate this new news story.

So, to define this more generally: **Developer-Task Automation** is taking a development task that a programmer typically would have to perform through multiple changes directly to the code and providing some type of interface (like a matrix) so that these changes can now be done in one spot, without actually having to write any (or little) code, just by changing some of the settings. And, a system architecture that automates developer tasks is called an **Automated Architecture**.

There are a lot of benefits to automation. Maintaining an automated system-architecture can be often much easier. If a developer needs to add or modify a new element to an automated component, you don't need to change any of the code, just the data that's used by the component. But, the cons of using automation are: often, developers need to write a good amount of additional code as compared to hard coding the content in a static component. But, with matrices, automated code is much more efficent to write.

It's worth mentioning that in other, non-NDA systems, automation is actually sometimes done, but normally using a DB or XML file. So, if we were to do the news feed with these techniques, we'd create a database table for the news info, and create a matching News model-object in a hibernate layer. And then, create application code that would use this model object to query the DB for the news info and format the results in a news feed. That's quite a bit of work. But, with matrices, automation is much simpler with much less code. You don't need a database, a database connection, or any model objects. The info for the dynamically generated content is simple stored in a matrix, which is a native data structure in our system.

Creating automated components when pratical is often a good idea, but with Hierarchy and matrices, automation becomes much more easy to do. In our systems, we have found we create automated components three to four times more than we normally do. Especially for websites, because of their content-driven nature, practically the entire site can be designed around them (the unconventionalthinking.com website is nearly entirely an automated system-architecture).

High-Performance Automation

And, compared with a database, automated components in Hierarchy have a couple of orders of magnitude better performance. The reason is because Hierarchy stores matrices in RAM, while a database access requires a trip over the network to retrieve data. Accessing a matrix is comparable to accessing an array or a hash, they're extremely light weight.

In a real world usage, again the Unconventional Thinking website, we found the generation of dynamic elements was so fast, we didn't need to add any caching of generated HTML. This makes using Hierarchy for automation ideal on many different levels, on ease of use and on performance.

Developer-Oriented Architecture

Using N-Dimensional Architecture tends to make systems **Developer-Oriented**. Developer-Oriented architecture is based on the Use Cases (*user tasks*) found in User-Oriented Software Development. It's basically the same idea. We previously mentioned this a few pages ago, but to build on what we said: In Use-Cases, the designers of an application try to view the system through the eyes of the user, and create tasks or "Use-Cases" for common actions the user would perform with the system. Developer-Oriented Architecture is the same thing. Developers are now users of the system, and a software architect tries to determine the maintenance and enhancement tasks that a future developer will need to perform. The architect is trying to see the architecture through the eyes of future developers.

N-Dimensional Architecture (NDA) is a Developer-Oriented Architecture. There are two main reasons why this is true:

1. NDA architectures put the elements that change the most at the top of the system and collect them together. So now, developers don't need to search across the entire system to make changes. In NDA, we are designing the system around the needs of the developers, and not just to meet the needs of end users.

2. NDA architectures tend to have **Development Interfaces**. Because we are ordering the system by abstraction, components and layers tend to get wrapped in classes that act as main interfaces into the component (or layer). These development interfaces are used by developers to work with the components, and so developers do not directly access any of the classes behind the development interface. An example of a development interface can be seen a few pages ago in the sample N-Dimensional Architectural diagram. In the Execution layer, there is a high-level class called "News." This News class wraps the entire news component. This is the main class that developers will use to work with it, and they won't directly access any of the objects behind this class. This is really just encapsulation, but on a larger scale.

   And, as we previously saw, the high-level settings objects are another type of developer interface. The developer uses these to change the behavior of the system.

   Also, since we think of this development interface as an actual interface for developers to work with the system, one of the goals of an architect is to create these development interfaces with a high degree of ease of use or what we call **architectural usability**. We want this interface to be as easy for developers to use as possible.

   One last small note. Notice that this News wrapper class is following the façade design-pattern. The Façade design-pattern is used a great deal in Developer-Oriented Architectures.

This was a summary explanation of some pretty detailed concepts, but we hope you get a feel for why N-Dimensional Architecture can be beneficial to your systems.