Peter Joh
Unconventional Thinking
peter@unconventionalthinking.com
7/27/2014
doc v0.03

NDA Expanded:
More Important Concepts of NDA, Universal-Data Definition and an Introduction to the Description Architectural-Pattern

Overview:

This article provides more info into the concepts of NDA that couldn't be discussed in the article, "An Overview of NDA." Specifically, it provides more depth into the main ideas of an NDA architecture (for example, changeability + learnability = usability), all explained in the context of a new, software-design technique called the Description Architectural-Pattern. The Description Architectural-Pattern and NDA most often work together and by explaining one, you often describe aspects of the other.

With the Description Architectural-Pattern, developers and architects use it in their systems to create components that are built around centralized descriptions. These descriptions are often simply code-versions of the behaviors captured during requirements. For example, often, the tables defining business behaviors (like discounts given to customer based on their type of customer account) can easily be mapped to some form of object representation or, better yet, matrices. The developer then creates worker code that uses these descriptions as input to do some type of work.

Note that the Description Architectural-Pattern is really an existing technique that has been used intuitively by many developers for years, but is now properly labeled, integrated into NDA, and much more fully developed.

Also, note that this is preliminary version of this document! The authors wanted to make sure these ideas were available online, but still need to refine both this document and the ideas presented. In addition, very little cross-referencing has been done for existing work on the subject! (Apologies to authors whose work this article builds on) This will also be done in the future.

Pre-Requisites:

To read this article, is necessary to have a basic understanding of Hierarchy's matrices (see the "Main/Learn" section of the projecthierarchy.org website).

Also, it is not necessary, but very helpful if you've already read the article, "An Overview of NDA." You can still gain many useful ideas in software engineering if you haven't read this article, but it's highly encouraged that you do so before hand.

And, for those who haven't read this previous article, NDA stands for "N-Dimensional Architecture." It is a new type of architecture that allows devs to organize their system by multiple characteristics at the same time (for instance, by both MVC *and* abstraction). And, like we mentioned, it's not too important that you know NDA to read this article, but we will be mentioning it from time to time.

Also, we'll often refer to the Description Architectural-Pattern by its acronym, DAP.

So, to get started, we'll begin by discussing a common problem in our applications.

Solving the Separation of a Data-Element's Definition Across an Application using Universal Data-Definition

In large, business applications, data elements (like customers info) are often used in many places throughout the system and must be defined in multiple places. For instance, for a user-registration page of a website, there is typically a last-name field. Think of the places this data element must be defined for just this simple page:

1. The HTML form:

   <input name='lastName' max='25'>

2. The client-side Javascript validation code:

```
var lastName = form.lastName;
checkNameString(lastName, 25);  // Of course, there are often libraries/frameworks that
                                // do validation, but they still need the value and
                                // settings passed to them.
```

3. The server-side code for the JSP/JSF registration page:

```
public class userRegPage {
    String lastName;
    String firstName;
       .
       .
       .
}
```

4. The DB model objects:

```
public class userDAO{
    String lastName;
    String firstName;
       .
       .
       .
}
```

5. In the DB itself:

```
Table User
  lastName   VARCHAR;
  firstName   VARCHAR;
       .
       .
       .
```

For a developer to modify an existing data-element (for instance, if he needs to change the Last Name field's max character length from 18 to 25), he may need to move to five different parts of the system to make a relatively simple change. Not only is this inefficient, it's error prone as the dev is likely to miss a necessary modification in one of the locations.

We can solve this problem of having a data element defined in multiple places using the concepts of NDA. Specifically, we'll apply the technique of grouping all the developer's most commonly used settings together along with **Automated Architecture** (refer to the article, "An Overview of NDA" for more info on this concept).

The basic idea is: For each data-element, collect all its data-type definition, placing all this together. This can be placed into arrays of objects, or more ideally, into a matrix (again, see the "Main/Learn" section on the project Hierarchy website for more info on matrices).

This matrix becomes the **universal data-definition** (or universal scheme) for the entire application, in all its different usages. So, for our user-registration page example, the matrix for this info would look like this:

```
package com.williespetstore;

MATRIX WebForm.Registration USES (Web.Form, Database) {

   `First Name` {
      FORM.REQUIRED: { +:IsRequired };
      FORM.CONTROL.TEXTBOX { 25 };
      HELP.TEXT: { "Please enter in your first name" };
      //  Field names: Table, ColumnName, Type
      DB.COLUMN: { +:Customer, +:First_Name, :String };
   }
   `Last Name` {
      FORM.REQUIRED: { +:IsRequired };
      FORM.INPUT.TEXTBOX { 25 };
      HELP.TEXT: { "Please enter in your last name" };
      DB.COLUMN: { +:Customer, +:Last_Name, :String };
   }
   `Gender` {
      FORM.REQUIRED: { +:IsRequired };
      FORM.CONTROL.RADIOBOX { "GenderRadioBox" } {
         RADIO.ITEM { "Male", "Male",:Checked };
         RADIO.ITEM { "Female", "Female" };
      }
      HELP.TEXT: { "Please select your gender" };
      DB.COLUMN: { :Customer, +:Gender, :"int" };
   }
   `Address 1` {
      FORM.REQUIRED: { +:IsRequired };
      FORM.CONTROL.TEXTBOX { 60 };
      HELP.TEXT: { "Please enter in your home address" };
      DB.COLUMN: { +:Customer, +:Address1, :String };
   }
      .
      .
      .
}
```

** Notice that in this matrix, we're using ITEM descriptors as the container for each field's information (if you're unfamiliar with the ITEM descriptor, see "Chapter 5 – Matrices"). ITEM descriptor are good for grouping related information together.*

Each field in the registration form has four types of information:

- Is this a required field?
- What type of form field it is.
- Help text
- The database column information that corresponds to each data-element

We can now use this matrix to generate a multi-page web form using a servlet or JSP. And also, we can use it to generate the insert query's SQL to store this information in the database. We can create this code in .mjava files (or .mjsp files if you're using JSP's. And for those that have not learned about Hierarchy, .mjava and .mjsp's are special files for the Hierarchy metacompiler that simply have extended Java syntax for working with matrix objects). Here's an .mjsp that uses this matrix to auto-generate the web form:

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

<%@page import="MATRIX com.williespetstore::WebForm.Registration"%>

<form>
   <%
   for (DESCRIPTOR<WebForm.Registration->ITEM> formField_Item:
         WebForm.Registration->ITEM{*}) {

      if (formField_Item->FORM.CONTROL:>ControlType == :TextBox) {
         %>
         <input type='text' name='<%=formField_Item.label%>'
            size='<%=formField_Item->FORM.CONTROL:>size'%>"/>
         <%

      } else if (formField_Item->FORM.CONTROL:>ControlType == :RadioBox) {
         %>
         <div>
         <%

         for (DESCRIPTOR<WebForm.Registration->ITEM->FORM.CONTROL->RADIOBOX->
               RADIO.ITEM radio_Desc :
               formField_Item->FORM.CONTROL->RADIOBOX->RADIO.ITEM{*}) {
            %>
            <input type='radio' name='<%=radio_Desc:>name>'
               value='radio_Desc:>value'%>"/>
            <%
         }
         %>
         </div>
         <%

      } else if (…) {
        //  Add other else-if statements to generate the rest of the
        //  form elements from the matrix.
              .
              .
              .

      }

   }
   %>
</form>
```

This code generates the user-registration form by looping over all the data elements in the WebForm matrix. The benefits of using matrices in this way (as a source for the "universal data definition" for your application), is now, all the definition information for each data-item isn't scattered across the application, where some of it is in the database layer in Hibernate objects, some in JSP's, and others in the Javascript. Instead, it's all been collected into one location, the matrix. And you can use this information to more easily process this form in a more automated fashion. It is an example of **Automated Architecture,** because the elements of the form aren't hard coded, but instead, generated using a description of what the form should look like. In fact, in the future, we'll do most of the automation for you, as we plan to release libraries for both form generation and databases queries. And because of this automation and the fact that you've collected the settings together, the form should be much

easier to maintain. For instance, if we need to add a new field, you don't have to search through the system for all the different places we need to add information for this field, you simply need to add it to one place, the matrix.

One important note about the matrix: notice that the above Universal Data-Definition matrix was extended to mix *settings and content information* along with the data-definition info. This may seem like a mistake, as most of us are taught in architectures like MVC to "separate the concerns," but this was actually on purpose. In our experience using NDA architecture, we have come to realize that at the high-level, strict separation of concerns *(often robotically done using MVC)* does not always lead to the best architectures. When introduced, MVC was a revolutionary architecture and its usage to *some degree* in our business applications will almost always be beneficial. But, for developers who work with the system, its maintainability and ease of understanding the architecture is often not ideal as it may separate elements that might best be kept together (MVC is often used as a one-size-fits all type of architecture, where in many cases the blind application of its principles isn't ideal for the situation).

What our experience has found is that the mid-level (manager type objects) and lower level (libraries and services) objects often benefit by a stricter separation of concerns, but the high-level, application-specific content and settings is often better kept together, with the content organized more logically by how the developers naturally would group this information! This *may* mean a MVC style grouping, but often it doesn't, and instead, results in an organization scheme designed around *how the developer sees and works with the system*. One common scheme is to keep conceptually related settings together: for instance, the matrix above is organizing its info around the data elements used in the system, and is actually grouping data and view settings together!

This may seem like sacrilege, but having more natural groupings of our settings leads to easier understanding of the system and greater efficiency in making changes. On top of this, as many of us agree, most applications *don't* under go the drastic future enhancements we developers often plan for (like allowing for the adding of new views), so one of the main reasons necessitating the separation is never used. Also, since matrices are very light-weight objects, if, in the future, we need to refactor them and pull settings out, it is relatively easy to do so.

But, on the flip side, in many situations, it *is* actually not a good idea to have all the settings and content in one matrix! In many situations, for instance where the system is large, mixing info like this could lead to future problems. In these situations, it is advisable to split the settings into its separate matrices based on expected future development. For instance, if this code actually *will* have multiple views (for example, it will support both HTML and PDF versions of the pages), then splitting off the html settings into its own matrix may make sense.

The main point is, how your organize this highest layer of settings of your systems should not be done using a blanket, MVC approach. Each design requires its own thought about the correct scheme and balance to use.

Now, even though we have just criticized the blanket use of MVC, generalizing and future-proof your settings matrices is still often desirable. We'll talk about this next.

Generalizing your Settings Matrices using More Generic Names and Structure

The above WebForm matrix may (and often is) ideal for situations where an application will be used in a limited context. But consider the situation where this matrix is used in a large, company wide-system, where it's accessed by multiple applications (think of the ecosystem of applications that run both the website and internal corporate side of Amazon.com. A customer's info will be used in possible dozens of systems). The matrix has been designed too specifically to be used well in this broader context.

For example, our current WebForm description of data may fit how the user info is used in the website, but not how the info would be used in, say, the Android phone app. To see this problem more explicitly, let's say we tried to reuse our existing, WebForm matrix in another system, like the customer-billing batch-processor. This system goes through each customer's outstanding-purchases and sends them an invoice.

Let's say a developer working on the customer-billing system needs to have this system know what the max size of the LastName field is. So, in this system, he then adds code to access the WebForm matrix's last-name field:

```
CustomerBillingProcessor {

  checkForOutstandingBalances(CustomerList customerList) {

     for(customer : customerList) {
        if (customer.lastName >
           WebForm.Registration->`Last Name`->FORM.INPUT.TEXTBOX:>MaxSize) {
           throw new BillingError("Last name too long");
        }
     }


  }

}
```

But, if you think this over, this is pretty hacker-like and prone to many problems, because the naming of the elements of this matrix implies that it is used in a very *specific* context, in the web application. For instance, let's now say the front-end dev needs to make a change to the form's max-size because of some strange caching limitation. The field is stored much better with the smaller field size, so he wants to set it from 25 to 24. And, since the naming used in the matrix is completely geared towards the website's registration form, the front-end web-developer naturally assumes that this matrix is only used by the website's code. So, he decides to change the MaxSize of the last-name field directly in the matrix! The problem is now, our customer-billing code, which uses this same value, has been inadvertently changed, causing errors in how it processes the last names.

A better solution would be to use a more general naming-scheme and structure for our customer info, one that isn't so tied to the registration form:

```
package com.williespetstore;

MATRIX UserData.Definition USES (DataDefinition, Database) {

   `First Name` {
      //  Field names: IsRequired, DataType, MaxSize
      DATA.DESC: { +:IsRequired, :"String", 25 };
      HELP.TEXT: { "Please enter in your first name" };
      //  Field names: Table, ColumnName, Type
      DB.COLUMN: { +:Customer, +:First_Name, :String };
   }
   `Last Name` {
      //  Field names: IsRequired, DataType, MaxSize
      DATA.DESC: { +:IsRequired, :"String", 25 };
      HELP.TEXT: { "Please enter in your last name" };
      DB.COLUMN: { +:Customer, +:Last_Name, :String };
   }
   `Gender` {
      //  Field names: IsRequired, DataType, MaxSize
      DATA.DESC: { +:IsRequired, :"Symbol", null };
      DATA.VALUEOPTIONS {
         //  Field names: Value, OptionalObjectValue
         VALUE: { +:Male, 0 };
         VALUE: { +:Female, 1 };
      }
      HELP.TEXT: { "Please select your gender" };
      DB.COLUMN: { :Customer, +:Gender, :"int" };
   }
      .
      .
      .
}
```

the names used are now not specific to the web form, so confusion like in the previous situation would be less likely to occur. As most of us devs would agree, the names you use for your variables and classes are important. They communicate what they are and how they can be used. As we've heard time and again, naming is the first way we devs document our systems, and this more general naming of descriptors and fields allows for this matrix to be reused in much more broader contexts.

So, when deciding whether to use general vs. specific naming, both extremes and everything in between should be considered. Many factors go into this, and the most important is the *current* use of matrix in multiple situations as well as the anticipated *future* use.

Back in the overview of this document, we mentioned that we'd explain a new style of design/programming called the **Description Architectural-Pattern** (and again, this pattern is actual something that programmers have done for years, it is just being explicitly defined especially in regards to NDA, as NDA and DAP naturally complement each other). The running website example we have been using is an example of DAP and leads us nicely into talking about it in this next section.

What is the Description Architectural-Pattern?

The Description Architectural-Pattern (DAP) is a software-design technique where parts of a system are designed around **descriptions**. These descriptions capture some aspect of the system and are fed as inputs to worker objects that do some form of work. It's typically done in three steps. Let's just briefly describe this without too much detail:

1. **Find some aspect of you system that can be *described* in a substantial way.**
2. **Translate this description into a code representation** – Complex behaviors/structures are often captured as tables, hierarchical diagrams, data-flow diagrams… Turn each into a group of constants, a matrix, a group of objects….
3. **Create worker code that uses this description as input to perform some type of work or compute some type of result.**

Admittedly, it's a pretty simple concept, but is often not used in situations that it probably should be. So now, to describe this in more detail, we'll go through these three steps with the WebForm application in mind (which, we mentioned, is an example of DAP):

1. **Find some aspect of you system that can be *described* in a substantial way** - This means find parts of your system that typically has some form of **related complexity** that has a lot of details. So, for the WebForm example, the data-elements that is used in the system is a glob of related info that has a lot of details: each data-element has a name, type, size limitations, sometimes a set of values…

2. **Translate this description into a code representation** – Complex behaviors/structures are often described as tables, hierarchical diagrams, data-flow diagrams… So, in our WebForm example, we translated the description of the data-elements into a hierarchical data object, more specifically, into a matrix. And, internally, the main way this matrix was organized was by data-element name (feel free to refer back to the WebForm matrix to see this for yourself).

3. **Create worker code that uses this description as input to perform some type of work or compute some type of result** – In the WebForm example, the .mjsp file used the WebForm matrix as input to generate the HTML form. The .mjsp is a worker that uses the matrix as description of what it should create.

So again, what is the Description Architectural-Pattern? It is organizing parts of a system around objects/data-structures that describe some aspect of the system, and then, performing work using these descriptions as input. To understand more fully when this is more useful, let's take a look at another common-usage of DAP, in your business logic.

Using the Description Architectural-Pattern with your Business Logic

A natural fit for DAP (used along-side NDA) in a system is to use it for parts of your business logic. And, by business logic, we mean the code in the system the analysis the current situation or state of the system, and then from this analysis, computes a result or performs some action that should be taken. And, considering business logic is already a type of description (it's a description of how the system behaves given some current state, broken down into a set of rules), it makes sense that it often is well suited for DAP.

As an example of business logic, let's look at an online, costume website. It sells costumes and accessories for parties / Halloween… The specific aspect of this business that we'll look at is the discounts given to customers during the different Holidays.

The amount of discount given is based on three things: The holiday, the type of account the customer has (Guest, Registered, or Premium), and the type of product. The discount is given as a percentage and is best represented as a table:

Customer Type

Product Class	Guest	Registered	Premium
Halloween			
Off-the-Shelf Costume	5%	7%	15%
Custom Costume	10%	12%	20%
Accessories	5%	7%	20%
Makeup	7%	10%	17%
Easter			
Off-the-Shelf Costume	2.5%	3.5%	7.5%
Custom Costume	5%	6%	10%
Accessories	5%	7%	10%
Makeup	3.5%	5%	8.5%

And, this table is very naturally represented as a matrix:

```
package com.willieshauntedhouse;


MATRIX Production.Descriptions USES (Products) {

   PRODUCT.CLASS +`Off-The-Shelf Costume` {
      CUSTOMER.TYPE +`Guest`: {
         //  Field names:   Discount
         SALE +`Halloween`: { .05 };
         SALE +`Easter`: { .025 };
      }
      CUSTOMER.TYPE +`Registered`: {
         //  Field names:   Discount
         SALE +`Halloween`: { .07 };
         SALE +`Easter`: { .035 };
      }
      CUSTOMER.TYPE +`Premium`: {
         //  Field names:   Discount
         SALE +`Halloween`: { .15 };
         SALE +`Easter`: { .075 };
      }
   }
```

```
   PRODUCT.CLASS +`Custom Costume` {
      CUSTOMER.TYPE +`Guest`: {
         //  Field names:   Discount
         SALE +`Halloween`: { .10 };
         SALE +`Easter`: { .05 };
      }
      CUSTOMER.TYPE +`Registered`: {
         //  Field names:   Discount
         SALE +`Halloween`: { .12 };
         SALE +`Easter`: { .06 };
      }
      CUSTOMER.TYPE +`Premium`: {
         //  Field names:   Discount
         SALE +`Halloween`: { .20 };
         SALE +`Easter`: { .10 };
      }
   }
      .
      .
      .
}
```

Here, as you can see, we've translated our requirements for our discounts into a code representation as a matrix. Now, according to the three steps of DAP, let's give this to some worker code. In the customer's shopping cart, we'll apply this discount to the items by simply grabbing the correct discount from the matrix:

```
Customer customer = currentCustomerObj;
int totalDiscount = 0;

for (Product product: cart) {

   totalDiscount += product.getCost() *
      Product.Descriptions->PRODUCT.CLASS[product.getClass()]->
         CUSTOMER.TYPE[customer.getCustomerType()]->SALE:>Discount;

}
```

The main design-principle being shown here is to separate your logic from the code that does the actual work, one of the benefits being better understandability. This may seem like an obvious idea, but in practice, we devs often mix the business logic freely with our worker code (because we're being lazy about design or we don't fully understand the large, complex architectures they are working on and what belongs where…). This leads to systems that are hard for other devs to learn as they must jump from component to component to understand even a simple behavior.

And, another major benefit of this separation of logic and worker code it is more maintainable, better growing as the system grows. For example, for our discounts, maybe later on we find that discounts over 15% need manager approval and should be processed using a separate shopping-cart process, or that each discount/holiday combination should have its own, custom message associated with it ("Spooktacular savings!"). Having the logic separated out like this makes it easier to change and enhance because you're able to better se the systems as a whole and not have to search through different parts.

But, on the other hand we should also mention, commonly, it *is* preferable to keep business logic with the component it is related to. For instance, for a shipping component, it may have a set of standard shipping-rates that are used through out the system. The common set of rules and rates on shipping should be kept with the shipping component, not with the main application-logic. The point is that each case should be considered

individually, and the guidelines are: first, that business logic is often a natural fit for DAP, and second, that *the structure of the system should not be the determining factor of where a piece of business logic resides!*

Next, let's look at another situation where DAP often fits, in our high-level control flow logic.

Control Flow of a System:
Using NDA and the Description Architectural-Pattern for Control Flow

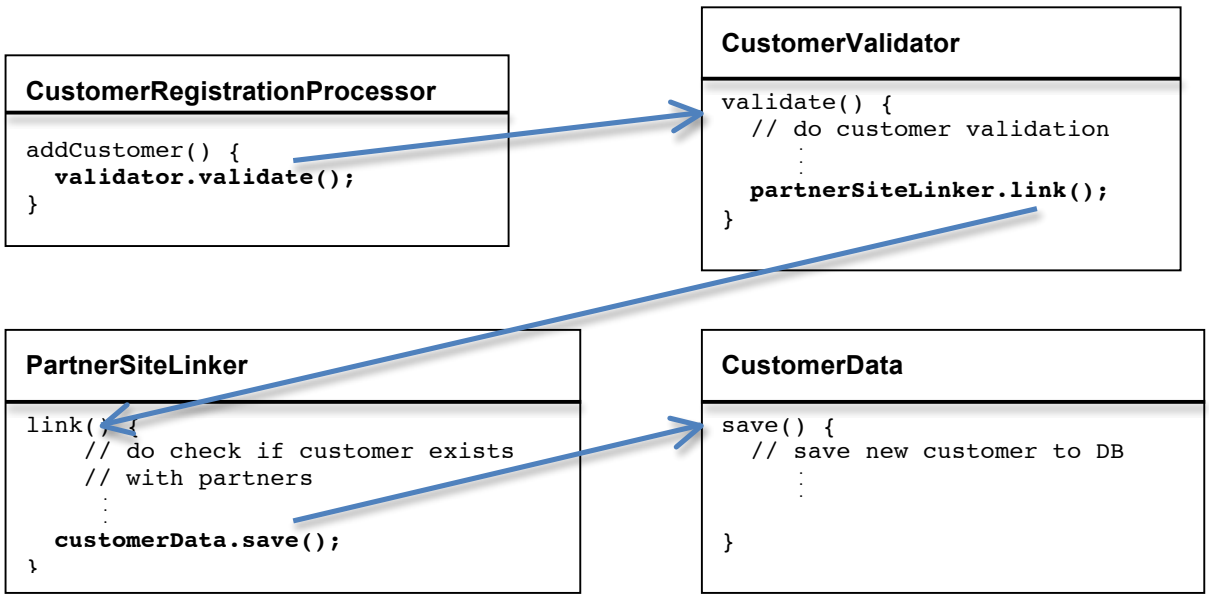
The high-level control-flow of a system is also a form of a description and is a good candidate for NDA and DAP. By high-level control-flow, we mean the sequence of important, high-level methods that are called on the major components of a system. For instance, back to our WebForm example, consider this components control flow:

- **initialization()** methods are first called on the different components in our system first starts up. Then, when our webform is hit:
- A **security.check()** is performed on the users session. Then, we do:
- A **cache.check(pageID)** to see if our page has already been generated. Then, since the page is not available, we generate the form using a servlet:
- **userRegistrationServlet.request()**. Lastly, we do any:
- **page.finish()** tasks that need to be performed, like session-state maintenance.

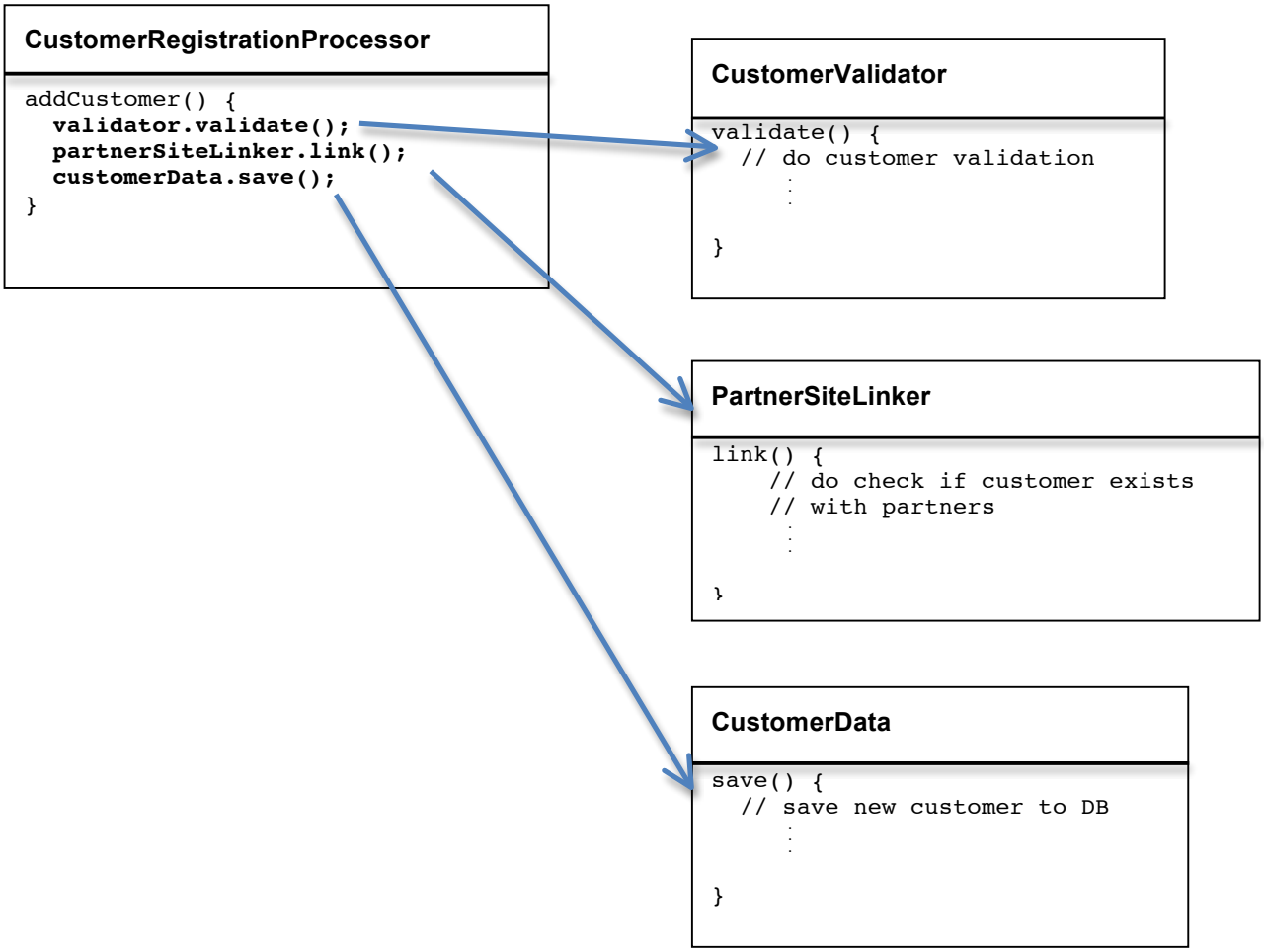
As you can see, the control flow through a system is also a description, the main idea being just like business-logic, it should be separated out from the lower-level parts of the system and kept together when possible (note, this is actually *not* in accordance with what Object-Oriented Programming suggests, but more on this later). To see this in another example, we can again look at our WebForm example, but look at a smaller part, the process of processing a web form submission.

When our web form processes a web, it performs the following steps:
1. **Validation** - Validate the customers info, such as address, phone number, username…
2. **Partner Site Link** – Check our partner websites if this customer already exists and link the accounts if they do.
3. **Add the new customer to database**.

For this process, a typical sequence of method calls on the objects of the system might look like this:



This type of method call structure is often called "**chaining**" or ("**stair structure**") as the method calls are chained together, one after another (note, this example is obviously *not* a good situation to use chaining, but it is often useful in other situations which will be discussed). On the other end of method-call structure is the "**forked**[1]" shape, where there is a central method that is the "main" method and calls the others. In fact, let's redo this example in a forked structure:



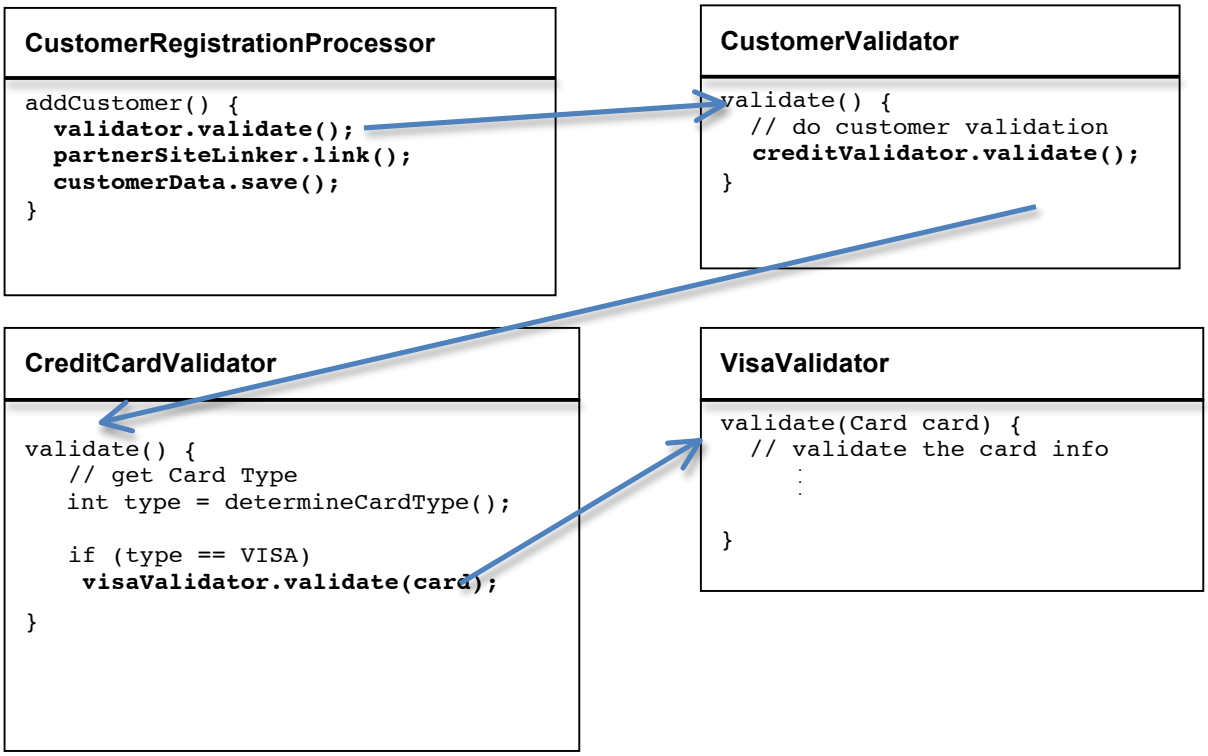
These two extremes for the structure of method calls is useful in different situations. In NDA and DAP, *for the high-level control flow* of a system or mechanism, we prescribe the forked structure. The reason is the forked structure is easier to understand. This centralized method-calling structure becomes a high-level description of what the steps are taken to perform a certain task. It is an automatic form of documentation and often closely resembles the related use case for the task captured during the requirements phase.

And because these high-level control-flow objects are descriptions, they fit well into NDA and DAP. In NDA, these control-flow objects should be pushed up into the highest-level layer of the architecture, existing in the Control Layer (and, it's important to note, when we say the control-flow objects should be pushed to the *highest-level layer* of the architecture, this layer depends on the "scale" that the objects exist in – for more info on scaling, see our article called "N-Dimensional Architecture with Scaling – part 2;" pg 7, "Scaling, Revisited"). And, in DAP, these control-flow objects are considered the *description* part of the DAP architectural-pattern as they *describe* the high-level flow of the system.

[1] From the book, "Object-Oriented Software Engineering," by Ivar Jacobson, pg.225

But, in Object-Oriented Programming, it is often suggested that chained/staired calls are more preferable to forked. The reason is because this spreads out the responsibility of what does what and when[2] (and in addition, in our opinion, chained/staired calls tends to make objects "smarter" and more independent which is often desirable). In NDA, even though we prescribed the usage of forked calls for the high-level control-flow, both forked and chained calls are used. And overall, whether a part of a system has forked or stair structure really only depends on how you look at your system in terms of scale. The general idea is if you look at the system a specific-level of scale, often, a small number of high-level objects make calls into many objects just below it in abstraction using the *forked* structure. The high-level object is in the role of a manger / control object and the lower level objects are in the worker role[3].

But, in many systems that *seem* forked, if you look at the method-call structure *across* all the different levels of scale/abstraction, you see that it's actually very strongly chained. For instance, if we take another look at the above forked-example, we can further expand the calls on the **CustomerValidation** class' **validate**() method. We can now see that this method actually has a chain of calls made up of many objects beneath it in abstraction:



As you can see, the customerValidator.validate() method isn't an isolated call, and has many levels of chained calls of its own. Most forked call-structures are actually chained across different levels of abstraction[4].

To sum up, the general principles on the forked vs. chained structure of method calls in NDA/DAP (and in all OO programming) is:

[2] Again, from the book, "Object-Oriented Software Engineering," by Ivar Jacobson, pg.225
[3] Note, this doesn't mean this high-level object actually is a manager/control object. It's just that in its *role in the relationship* to the lower-level objects, it's a manager/control object
[4] or across **component/object boundaries of responsibility**

1. In general, the method call structure *at the same-level of scale/abstraction* should *consciously* be put in a more forked structure.
2. But, in actuality, the structure of the calls tend to be more of a *chained/stair* call structure when you see the method call sequence across the levels of scale/abstraction

Note that for this first guideline, we said "the method call structure at the same-level of scale/abstraction should *consciously* be put in a more forked structure" – Unlike the stair structure, which happens naturally as we design, often, we devs must make an effort to identify a sequence of chained method calls that would better be organized as a forked structure, placing the calls into some type of control/manager object.

Now, not all types or parts of a system are good candidates for NDA and/or DAP. And, just like any design technique such as Design Patterns (or design choice such as which technologies to use), NDA and DAP should be used carefully, only when appropriate to the situation. This next section will discuss what situations NDA and DAP are good for.

What Parts of your System are Good Candidates for NDA and DAP?

When applying NDA and/or DAP to a system architecture, the two main-characteristics we look for in a part or aspect of system are:

1. **Changeability** – How often does a certain section or aspect of a system need to be changed by devs? For instance, in NDA, we often identify the settings and content of the system as probably needing to be changed by devs a great deal in their future maintenance-tasks. So, we pull these out and move them to the highest layer, **automating** these tasks as much as possible.

2. **Understandability –** What parts of the system, if centralized, best aid the devs in understanding the complex mess into which our large systems often evolve? A good example of this is centralizing the high-level control-flow into manager/control objects, and moving these objects to the higher layers of our architecture. These control-flow objects won't be changed much (so their "changeability" is low), but they become a source of documentation for the main steps performed by the system (as their aid in "understandability" is high), and also become a map for the main components used by the system.

And, together, systems that are organized using these two characteristics increase the overall system's **Architectural Usability**[5]. By architectural usability, we mean how easy is it for developers to learn the system and also how easy is to for them to perform their development tasks (such as fixing bugs, adding new business logic and features…). Another, existing term for this from software engineering might be maintainability, and the two are very similar, but Architectural Usability is more of a superset of the two. Architectural Usability actual takes the concepts of Usability from interface design and applies them to system architecture.

To wrap up this discussion on the NDA and DAP, let's briefly answer the question, why are their two techniques?

Why the two separate techniques?

NDA and DAP seem to have a lot in common, so much so that they seem like they should be merged into one. Why have they been kept separate? There are a couple reasons, the first is because their focuses are different even if their results are often similar. To be more specific, NDA focuses on the overall structure of a system (into layers often organized by abstraction and sections often by behavior), while DAP is a smaller technique, almost at the level of a design pattern (which is why it is called "DAP," the Description *Architectural-Pattern*). DAP defines how you should think of individual objects (as description and workers) while NDA describes the system as a whole.

[5] Again, refer back to our article, "N-Dimensional Architecture with Scaling – part 2," pg. 3 on "User-Oriented Architecture"

Lastly, because NDA and DAP present different but overlapping *design-philosophies,* they can give developers and architects two ways of viewing the same part of their system, hopefully with the goal of creating a better thought-out system as a whole. Aspects of both should be considered when designing any part of a system.