

DRAFT – 9/30/2004  
Comments welcome

## **An Analysis of Modularity in Aspect Oriented Design**

Cristina Videira Lopes  
Sushil Krishna Bajracharya

Department of Informatics  
University of California, Irvine

{lopes, sbajrach}@ics.uci.edu

# An Analysis Of Modularity In Aspect Oriented Design

Cristina Videira Lopes and Sushil Krishna Bajracharya  
Department of Informatics  
Donald Bren School of Information and Computer Sciences  
University of California, Irvine  
{lopes, sbajrach}@ics.uci.edu

## ABSTRACT

We present an analysis of modularity in aspect oriented design using the theory of modular design developed by Baldwin and Clark [8]. We use the three major elements of that theory, namely: i) Design Structure Matrix (DSM), an analysis and modeling tool; ii) Modular Operators, units of variations for design evolution; and iii) Net Options Value (NOV), a quantitative approach to evaluate design. We study the design evolution of a Web Services application where we observe the effects of applying aspect oriented modularization.

Based on our analysis we get to the following three main conclusions. First, on the structural part, it is possible to apply the DSM to aspect oriented modularizations in a straightforward manner, i.e. without modifications to DSMs basic model. This shows that aspects can, in fact, be treated as modules of design. Second, the evolution of a design into including aspect modules uses the modular operators proposed by Baldwin and Clark, with a variant of the Inversion operator. This variant captures taking redundant, scattered information hidden in modules and moving it down or keeping it at the same level in the design hierarchy. Third, when calculating and comparing NOV of different designs, we obtained higher NOV for the design with aspects than for the design without aspects. This shows that, under this theory of modularity, aspect oriented modularizations add value to the design.

## Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques—*Modules and interfaces, Object-oriented design methods*; D.2.8 [Software Engineering]: Metrics—*Product metrics*; D.2.11 [Software Engineering]: Software Architectures—*Information hiding*; D.3.3 [Programming languages]: Language Constructs and Features—*Modules, packages*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X .

## General Terms

Design, Languages, Management, Measurement

## Keywords

Aspect Oriented Programming and Design, Modularity, Design Space Matrix, Net Options Value

## 1. INTRODUCTION

Design Structure Matrix (DSM), also known as Design Space Matrix or Dependency Structure Matrix, is an analysis and design tool used in engineering disciplines. The applicability of DSM in computer design is illustrated in [8] and, particularly, in software design in [23]. There are two fundamental components in [8]: (1) A general theory of modularity in design with six modular operators as sources of design variation, and DSM as a tool to model modularity in complex systems; and (2) NOV (Net Options Value) as a mathematical model to quantify the value of a modular design. The six modular operators are: (i) Splitting, (ii) Substitution, (iii) Augmenting (Augmentation), (iv) Exclusion, (v) Inversion, and (vi) Port(ing).

[23] extends the DSM structure by introducing *Environment Parameters*, and applies this extended model to the design of KWIC (Key Words in Context), the program originally presented by Parnas in [17]. Information hiding is achieved by defining appropriate interfaces as *Design Rules*, which facilitate future changes in the design by reducing inter-modular dependencies.

This paper contributes to these earlier works in two ways: it takes a more realistic, modern day example compared to KWIC, and it looks into a new form of modular construct, *Aspect* [13], additionally to the conventional constructs for creating independent modules with representations for data structure, interface and algorithm. We demonstrate how DSM fits as a modeling tool and NOV as a quantitative model for evaluating design alternatives.

To illustrate this, we take an existing third party application, identify the design parameters within it and make changes to those parameters to obtain a design for a new application we desire. We describe each change in terms of one or more of the six modular operators and calculate NOV for each design. The system we describe was developed using Java, so all possible design modifications due to object oriented techniques are applicable to the system. We present the modular operations we performed while introducing aspects as a special case of Baldwin and Clark's *inversion* operator. We also present an estimate for the

NOV of aspect oriented modularization. We conclude by discussing the results of our analysis, the limitations of our work, and further work we intend to pursue in this field.

## 2. EXAMPLE APPLICATION

We take an example of a web application that uses web services to meet most of its functional requirements. The application, *WineryLocator*, uses web services to locate wineries in California. A user can give a point of interest in California as a combination of Street Address, City and Zip code. The address need not be exactly accurate. Once this information is given, the user is either presented with a list of matching locations to his/her criteria or is forwarded to another page if the given address uniquely maps to a valid location in California.

Once the application gets a valid starting point, the user then can select preferences for the wineries. Based on the preferences and the starting point the application generates a route for a tour consisting of all the wineries that match the criteria. The result is a set of stops in the route and a navigable map. From the result the user can also get driving directions.

### 2.1 Functional Decomposition

With the functionality described above we need the following types of services:

**Finding Accurate Locations (List)** A service that takes an incomplete description of a location and returns exact/accurate locations that match the description.

**Getting List of Wineries** A service that returns a list of all the wineries around the vicinity of the starting point the user is interested in. The user must be able to filter (her)his selection according to the different criteria (s)he wanted regarding the wineries to be visited.

**Getting Wineries Tour** Once an accurate starting point is obtained we need to get a set of wineries around that starting location. This further breaks down as:

- Getting all the wineries stops and wineries information that form a tour
- Getting a map for the tour that constitutes the wineries
- Navigating the map that highlights the tour with appropriate marks and support basic operations like Panning and Zooming.

**Driving Directions** Given a route made up of locations, we need a set of driving directions to visit all the destinations in the tour.

We use an existing application for MapPoint webservices<sup>1</sup> called *StoreLocator*<sup>2</sup>, developed by SpatialPoint<sup>3</sup>, as our starting design so that we can make changes in it to get *WineryLocator*. *StoreLocator* is similar in many ways to *WineryLocator*. Given a starting point of interest, *StoreLocator* displays several matching locations. Once the user

picks the starting location it generates a navigable map and a list of all coffee stores close to that starting location within a radius specified by the user. The user then can click on each store to get driving directions from the start location.

Hence, as far as the functionalities are concerned we need to make only two changes in *StoreLocator* to get *WineryLocator*: (i) replace the coffee store search with winery search, and (ii) present the user with a tour including the start location and all the wineries, unlike a list of directions from the start location to a selected store in *StoreLocator*.

In order to locate points of interest, such as coffee stores or wineries, MapPoint allows their service users to either use an already available *datasource* or upload new geographic data as a custom *datasource*. To bring out more opportunities for design changes, we substitute this functionality from MapPoint by our own webservice *WineryFind*, that provides a list of wineries around a vicinity of an exact start location. *WineryFind* also allows the users to set their search criteria by giving different preferences related to wines and wineries.

Table 1 shows the mapping of core application functionalities to the available webservices. The implementation was done in Java, using Apache AXIS [5] as well as the SOAP [24] toolkit to access the webservices.

### 2.2 Subsidiary Functions

Besides the main functionalities that *WineryLocator* offers to its end users, we consider two subsidiary functions that the application needs to provide but that are not so visible to the users: (1) *Authentication*: Before using any of the MapPoint services the application needs to provide a valid credential (username and password) to it. MapPoint uses the HttpDigest authentication mechanism for this. (2) *Logging*: We also introduce a logging feature in the system as a non-functional (subsidiary) requirement to trace all the calls made to webservices. Such feature is useful in many scenarios that require maintaining statistics about the access to the webservices within the application. This feature can simply be implemented by tracing every call to a webservice in the system.

## 3. REPRESENTING DESIGN STRUCTURES WITH DSM

Figure 1 depicts the design of *StoreLocator* in a DSM. Before presenting the design evolution from *StoreLocator* to *WineryLocator* in DSMs, we first describe some fundamental design concepts presented by Baldwin and Clark in [8], focusing primarily in software.

### 3.1 Elements Of Modular Design In Software

In this paper, our interpretation of the terms like modularity, architecture and hierarchy, remain the same and as generic as that presented in [8]. Almost all of the constituents of design that make up Baldwin and Clark's theory can be seen in our designs for *StoreLocator* and *WineryLocator*. We briefly summarize the definitions of the core elements from [8], as they are seen in our examples. We show all the definitions and vocabulary we borrow from [8] in *italics* below.

1. *Design*: *Design* is defined as *an abstract description of the functionality and structure of an artifact*. Representations such as software architectures [19, 22] design models in UML or source code fit this definition.

<sup>1</sup>www.mappoint.com

<sup>2</sup>available online at demo.mappoint.net

<sup>3</sup>www.spatialpoint.com

Task	Services	Providers	Method Signatures **
Finding set of exact locations	FindService	MapPoint	FindResults findAddress(FindAddressSpecification)
Getting wineries matching criteria	WineryFind	local service we developed	Destination[] getLocationsByScore(WinerySearchOption)
Generating route from the tour given set of destinations	RouteService	MapPoint	Route calculateSimpleRoute(ArrayOfLatLong, String /*dataSourceName*/, SegmentPreference)
Getting a map representing a route/tour. Also, navigating the map	RenderService	MapPoint	ArrayOfMapImage getMap(MapSpecification)
Getting driving directions	RouteService	MapPoint	can be obtained from a Route object

\*\* showing only the most relevant methods in format - `return.type web.service.function_name (input.parameter.type)`. The types shown in the list represent the classes in Java that maps to the types defined in the MapPoint object model. These classes were auto generated by the tool WSDL2Java that is a part of the Apache AXIS toolkit [5].

**Table 1: Mapping tasks to services.**

2. *Hierarchies*: The notion of hierarchy concurs with the one defined by Parnas in [18]. A module A is dependent on module B if A needs to know about B to achieve its function, i.e if B is visible to A.
3. *Medium* for expressing design: A designer expresses the basic structure and configuration of design elements with a *medium* (s)he chooses to work with. Examples are Architecture Description Languages (ADLs) for software architecture [16], UML for object oriented modeling and Java for program design (code). Media are among the highest parameters in the design hierarchy.
4. *Design parameters - the elements of design*: Parameters are the attributes of the artifact that govern the variation in design. Choosing new values for parameters give new design options. We use Java as the medium to express our design options, so the basic structural constructs like classes, objects, attributes, methods and packages all could be seen as design parameters. In our example, we remain at the granularity of classes and interfaces.<sup>4</sup>
5. *Module*: Structural elements that are strongly connected are grouped together as a module. Modules adhere to these three fundamental characteristics [8]:
  - (a) *Modularity increases the range of manageable complexity.*
  - (b) *Modularity allows different parts of a design to be worked on concurrently.*
  - (c) *Modularity accomodates uncertainty.*
7. *Abstraction*: *Abstraction hides the complexity of the element.* As a measure to reduce the complexity of design parameters, we represent complex modules (made up of further sub-modules) as a single parameter, as long as the details inside need not be revealed. An example of this in our models is treating a webservice as a single parameter.
8. *Interface and Design Rules*: Making changes in modules that have highly interdependent structures often requires endless tweaking as the designer tends to get lost in the cyclic side effect one module has on others. To avoid such cycles, decisions common to modules, that are unlikely to change are factored out as design rules. These design rules constitute the interfaces that designers use to connect modules with each other.
9. *Architecture*: provides a *framework that allows for both independence of structure and integration of function.* In our designs, we consider frameworks for enabling enterprise computing capabilities, such as J2EE [7], and APIs (Application Programming Interfaces), such as Java Servlet [6] (also a part of J2EE), as architectures.

## 3.2 Categories Of Design Parameters

We categorize the modules in the DSMs based of our ability to change them:

**External Parameters:** Parameters that we cannot modify and take for granted from some external providers. These parameters might be replaceable with similar parameters providing same functionality. External services, imported libraries and frameworks fall under this category. External parameters usually bring their own set of design rules into the application.

Extending DSMs with *environment parameters* was a major enhancement made to DSMs in [23]. We take *external Parameters* to be a particular category of *environment parameter* as they have similar characteristics.

**Design Rules:** Parameters we use as interface between modules and that are less likely to be changed are design

<sup>4</sup>Design elements, design parameters and modules are all interchangeable terms that can be used to describe individual elements that constitutes a design. We follow this convention throughout this paper.

rules. Design rules can either be imported from external parameters or designed specifically for the application.

**Application (functional) Modules:** Functional units in the system that perform application specific task(s) are categorized as *application modules*.

**Subsidiary Modules:** We further classify modules that contribute to subsidiary or secondary functionalities as *Subsidiary Modules*.<sup>5</sup>

**Application Controller:** These are mostly connector modules, as they use the design rules as interface to access the functionalities provided by the application modules, gluing them up in an application, and serving the end users. We also put configuration modules such as *deployment descriptors* in this category as they contribute in assembling modules even though they do not directly serve the end users of the system.

We believe most of the modules in modern day applications fall into one of the above categories. Furthermore, most of the development task in today's applications lies within mapping application specific requirements to the imported functionalities from external modules.

### 3.3 Conventions For DSMs

Figure 1 shows the DSM for StoreLocator. The DSMs have been constructed as normally is done [8, 23, 1, 21]. We arrange all the design parameters in a row-column form, with marks in those cells where we need to show the interdependencies between the parameters. We have adopted the simplest form of showing interdependencies, by putting an 'X' mark in the relevant cell.

The DSMs and the design parameters roughly match the application structure but there is not an exact one to one mapping from the elements in a DSM to the syntactic constructs in the program. The parameters we have shown are semantic, rather than syntactic, objects that occur to a designer's mind. However, all of the parameters, excluding the external ones, can be mapped to any one of these: Java classes or interfaces, aspects written in AspectJ [2], or, XML deployment descriptor files. In short, a DSM presents an abstract view of module dependencies in an application.

*Clustering* and *partitioning* are two standard DSM operations to get a modular or a *protomodular* [8] structure from an otherwise unmodularized DSM. Since the elements in our DSM are taken from a readymade application, StoreLocator, they already have a basic modular structure, and these operations do not have a very significant role in our process. The only explicit clustering we do in the DSMs is to categorize the parameters into one of the parameter categories we have listed in section 3.2.

The following list describes the graphical and visual clues we include in our DSMs:

1. The leftmost (first) column in DSM is used to label the clusters the parameters have been categorized based on the classification presented in section 3.2
2. The second column lists the name of all the design parameters

<sup>5</sup>If a module performs both application specific tasks as well as subsidiary tasks, we treat it as an *application module*.

		1	2	3	4	5	6	7	8	9	10	11
< EP >	< service > MapPoint	1	*									
	< API > Apache AXIS	2		*								
	< API > Servlet	3			*							
	HttpSessionBindingListener	4			X	*						
< DR >	MapPoint Design Rules	5	X	X		*						
	StoreLocator	6				X	*					
< AM >	HttpSessionStoreLocator	7		X	X		X	*				X
	< jsp > locate	8		X	X		X	*	X			
< AC >	< jsp > display	9		X	X		X	X	X	*	X	
	< jsp > directions	10		X	X		X	X	X		*	
	< DD > web.xml	11	X	X	X							*

Figure 1: DSM for StoreLocator.

3. The third column assigns numbers to all these parameters for easy reference. Rest of the columns constitute the matrix showing dependencies. The top most row resembles the parameters by the numbers as assigned in the third column.
4. In the matrix area (fourth column onwards), thick solid borders in the cells set the boundary for modules, dark dashed lines set boundaries for the interaction areas between different categories of modules, (for example between design rules and external parameters) and light dotted lines are markers for individual cells.
5. Shaded group of cell(s), enclosed within a dark border, represents a group of parameters (or a single parameter) that we treat as individual modules for NOV analysis.<sup>6</sup>
6. We use a descriptive text inside a pair of opening and closing angular bracket (e.g. < DR >) for two purposes; (i) in first column to abbreviate the category name and (ii) in second column as a stereotype to mark the special connotation some design parameters bear. Table 2 lists all such stereotypes we have used.

### 3.4 Design Hierarchy Diagrams

Figure 2 depicts the *design hierarchy diagram* (or simply, hierarchy diagram) of StoreLocator. Hierarchy diagrams and DSMs model the same structure and information about dependencies among design elements. A hierarchy diagram is a dependency graph of all the parameters in a design. A parameter in a hierarchy diagram has two set of connections: connections from above, to those parameters it depends on, and connections from below, to those parameters that depend on it.

To avoid line cluttering in the hierarchy diagrams we have omitted few dependencies from figure 2.<sup>7</sup> Furthermore, we grouped some related parameters into big boxes, and show their common dependency with other parameters using the box that encloses them. We label, or stereotype, special elements in hierarchy diagrams (as in the DSMs). These labels are listed in table 2.

Hierarchy diagrams are useful in visualizing the effect of different modular operators on existing designs, but they get cluttered with a handful of parameters and dependencies. Thus, we discuss all our designs for StoreLocator and WineryLocator using DSMs. We also omit some parameters

<sup>6</sup>NOV analysis is presented in section 6.

<sup>7</sup>The DSM for StoreLocator in figure 1 show these dependencies that are excluded from figure 2.

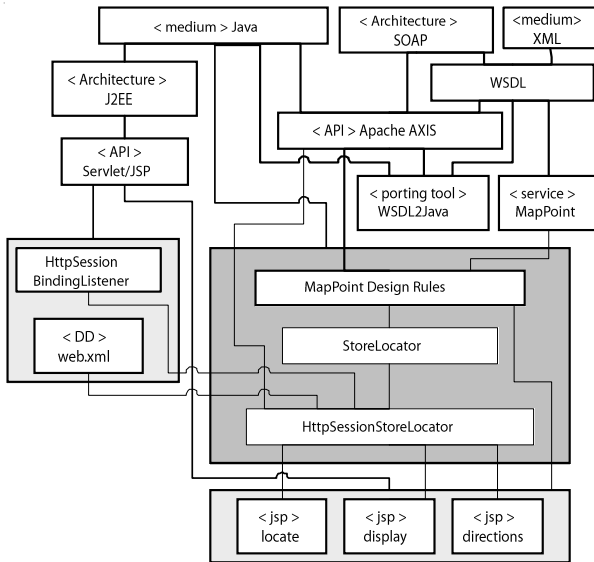


Figure 2: Hierarchy Diagram for StoreLocator.

that are at the higher *level* (e.g. *< medium > XML* in figure 1), from the DSMs, as they are not significant for our analysis.

## 4. TRACING DESIGN EVOLUTION WITH DSMs

### 4.1 Design Goals For WineryLocator

The starting design of WineryLocator, StoreLocator, is sufficient for the purpose of our analysis and meets the goals we had set, namely:

1. Identifying separate functional units as application modules so that we can plug-in our own webservice providing winery information between the several functionalities offered by the MapPoint webservices.
2. Decoupling the application controller from MapPoint's design rules.
3. Defining a set of simple, yet sufficient, design rules for our application that allow us to have different implementation of application controller modules; for example, to switch from web based to a GUI application based on Java Swing.
4. Being able to replace each of the application modules with an alternative implementation with the least possible side effects to the other modules.

With the required background and conventions we now discuss the different design variants.

### 4.2 Identifying Basic Design Elements

DSM in figure 1 is the starting point for our design exploration. We created these diagrams after understanding the code structure of StoreLocator and the design rules of MapPoint.

The list below enumerates all the design parameters and their role in the initial design of StoreLocator shown in figures 1 and 2.

Labels	Meaning	used in
Medium	as defined in section 3.1	HD
Architecture	as defined in section 3.1	HD
API	Application Programming Interface	HD, DSM
porting tool	translation tool used to convert artifacts produced in one medium to another	HD
service	Remote Webservice	HD, DSM
DD	Deployment Descriptor	HD, DSM
JSP	Java Server Pages	HD, DSM
Aspect	A modular unit representing a crosscutting concern	HD, DSM
Design Rules	Java Server Pages	HD, DSM
EP	External Parameters	DSM
DR	Design Rules	DSM
AM	Application Modules	DSM
AC	Application Controller	DSM
SM	Subsidiary Modules	DSM

Table 2: Labels used in DSMs and Hierarchy Diagrams (HD).

1. *MapPoint Design Rules*: These constitute the classes and methods as defined in the MapPoint object model [3], that are used to access and interact with its services. The porting tool *WSDL2Java*, part of the *Apache AXIS* toolkit [5] for webservices, generates all these required classes in Java from the description of the web services expressed in XML as a *WSDL* (Webservices Description Language) [25] file.
2. *StoreLocator*<sup>8</sup>: This is an application module implemented as a Java class. It handles the mapping of the application tasks to the services available from MapPoint by providing methods that take user inputs as parameters and call the appropriate service methods to list starting locations. It also provides access to the list of stores, maps, map navigation functions and driving directions. This StoreLocator module uses MapPoint's classes as parameters in its helper methods.
3. *HttpSessionStoreLocator*: Since a valid credential comprising of a user name and a password needs to be provided to MapPoint before using any of its services, the module StoreLocator is extended as a class *HttpSessionStoreLocator* that adds the authentication capability. There are no design parameters in the MapPoint object model that reflects this authentication mechanism because MapPoint relies on *HttpDigest* authentication. This authentication mechanism is a part of the the XML based communication protocol that webservices use. The *AXIS* toolkit [5], that implements such protocol, injects parameters that support such protocol specific tasks in the *MapPoint design rules* during the process of generating them. Consequently

<sup>8</sup>The name of this module is same as the application. Whenever this distinction is not clear from the context, we explicitly specify whether we are referring to the application or this module

StoreLocator ( <i>Old</i> )	WineryLocator ( <i>New</i> )	Changes
Figure 1 (DSM)	Figure 3 (DSM)	StoreLocator application modified as WineryLocator application
StoreLocator ( <i>F.1, P.6</i> )	AddressLocator ( <i>F.3, P.8</i> ), WineryFinder ( <i>F.3, P.10</i> ), RouteMapHandler ( <i>F.3, P.11</i> )	The composite functionality of StoreLocator module has been split into ( <i>F.3, P.8</i> ) that locates an accurate starting address, ( <i>F.3, P.10</i> ) that generates list of wineries (this enabled the substitution of 'store search' with 'winery search') and ( <i>F.3, P.11</i> ) that generates maps and routes.
HttpSessionStoreLocator ( <i>F.1, P.7</i> )	AuthAddressLocator ( <i>F.3, P.9</i> ), thRouteMapHandler ( <i>F.3, P.12</i> )	Splitting of ( <i>F.1, P.6</i> ), led to the splitting of ( <i>F.1, P.7</i> ) into ( <i>F.3, P.9</i> ) and ( <i>F.3, P.12</i> ). This split was necessary to carry on the authentication feature ( <i>F.1, P.7</i> ) provided to ( <i>F.1, P.6</i> ) into the newly created modules ( <i>F.3, P.9</i> ) and ( <i>F.3, P.12</i> ).
locate ( <i>F.1, P.8</i> )	startWineryFind ( <i>F.3, P.13</i> )	( <i>F.1, P.8</i> ) substituted by ( <i>F.3, P.13</i> ), both provide an equivalent functionality
--	searchWinery ( <i>F.3, P.14</i> )	( <i>F.3, P.14</i> ) is a new module that help users to specify criteria for refining winery search, this functionality was absent in StoreLocator. (Addition of this module can be taken as augmentation)
display ( <i>F.1, P.9</i> )	tour ( <i>F.3, P.15</i> )	( <i>F.3, P.15</i> ) presents the user with list of wineries and a navigable map that constitutes a tour.
directions ( <i>F.1, P.10</i> )	directions ( <i>F.3, P.16</i> )	( <i>F.3, P.16</i> ) presents the user with a detailed driving directions for a tour of all the wineries, whereas, ( <i>F.1, P.10</i> ) presents the directions from a start location to a destination.

**Table 3: Changes made in StoreLocator for the first version of WineryLocator.**

*HttpSessionStoreLocator* depends on the Apache AXIS API to submit the authentication credentials to Map-Point as the parameters related to authentication come along with Apache AXIS. This is a subtle dependency as the otherwise unnecessary detail has to be known to understand the full working of this authentication mechanism.

4. *HttpSessionBindingListener*: This is an interface defined in the Java Servlet API [6]. *HttpSessionStoreLocator* implements this interface and provides methods that are called by the servlet container whenever an object of *HttpSessionStoreLocator* is brought into a session. In this way the servlet container can provide the values for the 'username' and 'password', configured in the *deployment descriptor* of the application, to *HttpSessionStoreLocator*.
5. *web.xml*: is the *deployment descriptor* of the application and stores configuration information like user name/password values and URLs for accessing the web services. These values are passed into *HttpSessionStoreLocator* through the methods it implements from *HttpSessionBindingListener*.
6. *Application controller* modules (JSPs, Deployment Descriptor): *locate* takes the information on starting location, presents the matching list and picks a starting address. It links to *display* for rest of the functionalities. *display* presents the user with the matching store locations and also a navigable map with the stores highlighted. *display* takes the information on a particular store to be visited and links to *directions* that displays the driving directions from the start address to the store selected in *display*.

Most of the external parameters are omitted in the DSMs as the application modules do not directly depend on them. Since the changes we make are concentrated within the application this omission does not affect the comprehensibility of design evolution.

Hierarchy diagram in figure 2 shows all the external parameters in the StoreLocator. The dependencies among these external parameters show how *porting* [8] works at higher level, enabling the interoperability of externally implemented services with a custom application and how external design rules can be imported in applications.<sup>9</sup>

### 4.3 First Version Of WineryLocator After Performing Splitting and Substitution on StoreLocator

Figure 3 shows the DSM for the first version of WineryLocator we obtained from StoreLocator. A new set of design rules, *WineryFind Design Rules* (*F.3, P.7*)<sup>10</sup>, have been imported into the application for using the services provided by the *WineryFind* webservice (*F.3, P.2*). The *StoreLocator* module (*F.1, P.6*) along with *HttpSessionStoreLocator* (*F.1, P.7*) have been split resulting in 5 parameters (*F.3, P.8* through 12). The application controller modules have been substituted and augmented with a new module *searchWinery* (*F.3, P.14*). The design changes are listed in table 3 describing all the splitting, substitution and augmentation made from figure 1 to figure 3.

### 4.4 Introducing Subsidiary Functionality With Augmentation

We get to the design in figure 4 by adding a logging feature to the first version of WineryLocator shown in figure 3. We introduce a new module *WebServicesLogger* (*F.4, P.8*) that is responsible for logging the access of web services and maintaining any pertaining statistics.

(*F.4, P.9*), (*F.4, P.11*) and (*F.4, P.12*) are the three modules that access the webservices. All the calls to the webservices within these modules need to be traced and linked to *WebServicesLogger* (*F.4, P.8*) to maintain the log.

<sup>9</sup>Most of these external parameters remain unchanged in all of the design variants we discuss.

<sup>10</sup>For brevity we will be referring to Figures by 'F' and Parameters by 'P', with this convention we can refer to any  $n_{th}$  parameter in a Figure  $m$  as  $F.m, P.n$ .

## 4.5 Setting Application Specific Design Rules For WineryLocator

The design of WineryLocator in figure 4 is functionally complete. It fulfills all the functional requirements that we had set for WineryLocator in section 2. But it lacks the design goals we listed in section 4.1. We introduce a new set of design rules for WineryLocator to decouple the application controller modules from MapPoint’s design rules. This allows us to move MapPoint and WineryFind Design rules to the *External Parameters* category. These new design rules are specific to WineryLocator and independent of the MapPoint and WineryFind design rules.

Five new parameters (*F.5, P.8* through *12*) are introduced as application design rules for WineryLocator. The application controller modules (*F.5, P.19* through *22*), use these application design rules as interfaces to the application modules (*F.5, P.14* through *18*). Table 4 lists the role of the newly introduced design rules through the tasks they model.

Design Rules	Id	Models
startAddress:Address	( <i>F.5, P.8</i> )	starting location users provide and select
matches:Address[]	( <i>F.5, P.9</i> )	collection of address matches for the starting location
WinerySearchOption	( <i>F.5, P.10</i> )	preferences for winery search
Tour	( <i>F.5, P.11</i> )	tour representation for all wineries visit including a representation for map
MapOperation	( <i>F.5, P.12</i> )	standard map operations users perform

Table 4: Application specific design rules for WineryLocator.

## 4.6 Applying Aspect Oriented Modularization

We use the two forms of modularization that Aspects<sup>11</sup> provide to reduce the dependencies among the modules (application and subsidiary) in the design for WineryLocator in figure 5. We perform aspect oriented modularization for two of WineryLocator’s features:

1. *Logging*: Using the *pointcut-advise* mechanism [15] we remove the dependencies that modules (*F.5, P.14*), (*F.5, P.16*) and (*F.5, P.17*) have on module (*F.5, P.13*). We add a *Logging* aspect (*F.6, P.22*), that captures the calls to the webservices directly from the design rules for MapPoint (*F.6, P.6*) and WineryFind (*F.6, P.7*). The logging aspect, module (*F.6, P.22*), hooks these calls with the module WebServicesLogger (*F.6, P.21*).
2. *Authentication*: We use *introductions*, also known as *open-class* mechanism [15], to inject the authentication specific functionality into the application modules (*F.6, P.13*) and (*F.6, P.8.15*). This adds another aspect, *Authentication* (*F.6, P.23*), in the final design.

With this modification we have achieved all the design goals we had for WineryLocator.

<sup>11</sup>Although we conceive these modularizations with AspectJ in mind any aspect oriented framework providing *pointcut-advise* and *introductions* can be used for this purpose.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
External Parameters	< service > MapPoint	1	*																
	< service > WineryFind	2		*															
	< API > Apache AXIS	3			*														
	< API > Servlet	4				*													
	HttpSessionBindingListener	5					X	*											
<DP>	MapPoint Design Rules	6	X	X			*												
	WineryFind Design Rules	7		X	X			*											
Application Modules	AddressLocator	8						X	*										
	AuthAddressLocator	9			X	X			X	*									X
	WineryFinder	10						X			*								
	RouteMapHandler	11							X			*							
	AuthRouteMapHandler	12			X	X							X	*					X
Application Controller	< jsp > startWineryFind	13			X	X			X						*	X			
	< jsp > searchWinery	14			X	X	X	X			X				X	*	X		
	< jsp > tour	15			X	X					X				X		*	X	
	< jsp > directions	16			X	X							X		X			*	X
	< DD > web.xml	17	X	X	X	X									X				*

Figure 3: DSM for WineryLocator application after splitting and substituting the modules in StoreLocator.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
External Parameters	< service > MapPoint	1	*																
	< service > WineryFind	2		*															
	< API > Apache AXIS	3			*														
	< API > Servlet	4				*													
	HttpSessionBindingListener	5					X	*											
<DP>	MapPoint Design Rules	6	X	X			*												
	WineryFind Design Rules	7		X	X			*											
<SM>	WebServicesLogger	8							*										
	AddressLocator	9						X		*									
	AuthAddressLocator	10			X	X			X		*								X
	WineryFinder	11						X	X			*							
	RouteMapHandler	12						X	X				*						
Application Controller	< jsp > startWineryFind	13			X	X				X					*	X			
	< jsp > searchWinery	14			X	X	X	X			X				X	*	X		
	< jsp > tour	15			X	X					X				X		*	X	
	< jsp > directions	16			X	X							X		X			*	X
	< DD > web.xml	18	X	X	X	X									X				*

Figure 4: DSM for WineryLocator with logging feature after augmentation.

## 5. EFFECT OF ASPECTS ON DEPENDENCIES

The hierarchy diagrams in figures 7 and 8 show the effects that aspects have on module dependencies. In both these figures the direction of the dark arrow indicates the design change after introducing aspects. Figure 7 models the design change that was made to perform aspect oriented modularization to logging and figure 8 models the same for authentication.

Specifying interfaces and providing implementations are common practices in software design that concur with the notion of design rules and hidden modules. But the consequence of this separation technique is that the *client modules* need to be linked to the *server modules* (those that implement the interfaces) in some way. (To prevent any confusion due to terminologies we refer to these *server modules* as *providers* and *client modules* simply as *clients*.)

Different strategies exist to resolve these dependency issues between *providers* and *clients*. Fowler discusses these in some depth in [9] and [10], where he introduces *dependency injection* as one of the solutions. Dependency injection relies on *Assembler* modules that are responsible to connect the client modules with implementation, thus clients become virtually independent on implementation, they just see the interface. Assembler modules lie lowest in the hierarchy.

Aspect oriented mechanisms eliminate the dependencies *clients* have on *providers* by introducing aspects as new modular structures. Aspects depend on (or see) these *clients*



and *providers*, and are responsible to provide connections between them. Aspect oriented modularization with introductions is similar to the dependency injection technique. The effect that the Pointcut-Advise mechanism has on dependencies is similar to that of the *inversion* modular operator.

		4	6	1	2	3	5	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
External Parameters	< service > MapPoint	4	*																					
	< service > WineryFind	5	*																					
	< API > Apache AXIS	1		*																				
	< API > Servlet	2		*																				
	HttpSessionBindingListener	3			*																			
	MapPoint Design Rules	5	X	X	X	*																		
	WineryFind Design Rules	7	X	X	X	*																		
Design Rules	startAddress : Address	8						*																
	matches : Address []	9						*																
	WinerySearchOption	10						*																
	Tour	11						*																
	MapOperation	12						*																
Application Modules	WebServicesLogger	13							*															
	AddressLocator	14				X		X	X				X	*										
	AuthAddressLocator	15		X	X								X	*									X	
	WineryFinder	16		X	X			X	X	X			X	*										
	RouteMapHandler	17		X	X			X	X	X			X	*										
	AuthRouteMapHandler	18		X	X			X	X	X			X	*										X
Application Controller	< jsp > startWineryFind	19		X	X			X	X	X			X	*		X				X				
	< jsp > searchWinery	20		X	X			X	X	X			X	*		X				X				
	< jsp > tour	21		X	X			X	X	X			X	*		X				X				
	< jsp > directions	22		X	X			X	X	X			X	*		X				X				
	< DD > web.xml	23	X	X	X	X								*										*

Figure 5: DSM for WineryLocator after introducing application specific design rules.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
External Parameters	< service > MapPoint	1	*																					
	< service > WineryFind	2	*																					
	< API > Apache AXIS	3		*																				
	< API > Servlet	4		*																				
	HttpSessionBindingListener	5			*																			
	MapPoint Design Rules	6	X	X	X	*																		
	WineryFind Design Rules	7	X	X	X	*																		
Design Rules	startAddress : Address	8						*																
	matches : Address []	9						*																
	WinerySearchOption	10						*																
	Tour	11						*																
	MapOperation	12						*																
Application Modules	AddressLocator	13			X			X	X				X	*										
	WineryFinder	14			X			X	X				X	*										
	RouteMapHandler	15			X			X	X				X	*										
	< jsp > startWineryFind	16		X	X			X	X	X			X	*		X								
	< jsp > searchWinery	17		X	X			X	X	X			X	*		X								
	< jsp > tour	18		X	X			X	X	X			X	*		X								
Application Controller	< jsp > directions	19		X	X			X	X	X			X	*		X								
	< DD > web.xml	20	X	X	X	X								*										*
	WebServicesLogger	21																					X	*
	< Aspect > Logging	22					X	X															X	*
Subsidiary Modules	< Aspect > Authentication	23		X	X								X	X								X		*

Figure 6: DSM for WineryLocator after aspect oriented modularization.

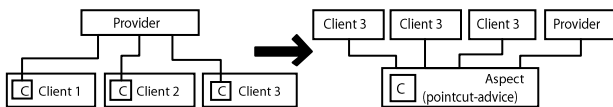


Figure 7: Effect of aspects with pointcut-advise on dependencies. 'C' represents common points in clients accessing the providers. (Direction of the arrow shows the change after applying aspect oriented modularization.)

### 5.1 Aspect Oriented Modularization As A Variant Of Inversion

Looking into the structural changes aspect oriented modularization brings we treat it as a variant of Baldwin and

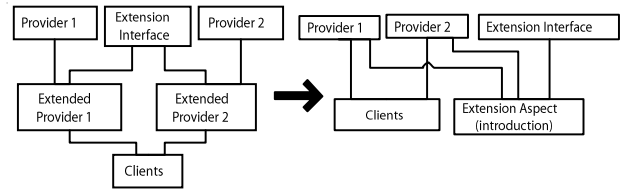


Figure 8: Effect of aspects with introduction on dependencies.

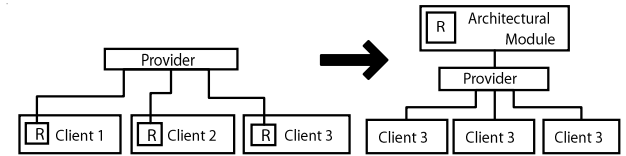


Figure 9: The effect of inversion. 'R' represents redundant parameters/code in clients that is moved to Architectural module after inversion.

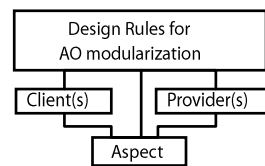


Figure 10: Design rules for aspect oriented (AO) modularization.

Clark's *inversion* operator. Inversion has two major effects, (i) it captures common elements hidden inside the modules, and (ii) puts them above the existing modules as architectural modules, thus changing the *levels* of the modules and dependency relationships between them. A simple example of the structural change inversion brings is shown in figure 9.

Comparing figure 9 (inversion) with figures 7 and 8 we can see that both inversion and aspect oriented modularization involve capturing common parameters and moving those common parameters to a single module. This changes the levels of the parameters and makes aspect oriented modularization similar to inversion. However, aspect oriented modularization introduces modules (aspects) that depend on existing modules, whereas inversion introduces modules on which existing modules are dependent. This makes aspect oriented modularization a variant of inversion.

### 5.2 Design Rules For Aspects

Figures 7 and 8 are just one of several variations of pointcut-advise and introduction mechanisms. Particularly, in these figures we do not see what the visible design rules for aspects are. In both cases aspects depend on *clients* or/and *providers*.

In figure 7 a small box labeled as C denotes the common points in *clients* accessing the *providers*. C is moved into the aspect after aspect oriented modularization and it represents two things: (i) interfaces that a *provider* provides, and (ii) points in *clients* that access such interfaces. A typical way to design aspects following this process (as in Aspect.J) is to capture these points as *joinpoints*, (for example the method names a *provider* provides and the method names of *clients* that access the *provider*), that need to be *advised*.

Such joinpoints constitute C, and, in a way, become design rules for the aspect. Defining design rules for aspects implies making such joinpoints explicit. Just as architectural modules emerge after sustaining a considerable design evolution, an aspect oriented design would also result in well defined design rules for aspect oriented modularization, as in the structure shown in figure 10.

## 6. QUANTITATIVE ANALYSIS WITH NET OPTIONS VALUE

In this section we present the quantitative analysis of the various design options for WineryLocator starting from StoreLocator. Our analysis is based on a generic expression for evaluating the option to redesign a module as developed in [8], represented mathematically as shown below.

$$V = S_0 + NOV_1 + NOV_2 + \dots + NOV_n \quad (1)$$

$$NOV_i = \max_{k_i} \{ \sigma_i n_i^{1/2} Q(k_i) - C_i(n_i)k_i - Z_i \} \quad (2)$$

$$Z_i = \sum_{j-sees-i} c_n n_j \quad (3)$$

Given below is a brief explanation of mathematical model for NOV as given in [8].

- $V$  denotes the value of a system.
- $S_0$  is the value of the system with no modular structure, that can be normalized to 0.
- $NOV_i$  is the NOV for  $i^{th}$  module, taken as the maximum return value possible out of  $k$  design experiments on the  $i^{th}$  module.
- $(\sigma_i n_i^{1/2} Q(k_i))$  represents the expected benefit to be gained from the  $i^{th}$  module. This value is assumed to be the expected value of a random variable with a normal distribution having a variance of  $\sigma_i^2 n_i$ .
  - $\sigma_i$  is the *Technical Potential* of the module.
  - $Q(k)$  is the expected value of the best  $k$  independent trials from a standard normal distribution for all positive values in the distribution.
- $(C_i(n_i))$  represents the cost of running a design experiment on the  $i^{th}$  module. Mathematically the cost of an experiment is a function of the module's complexity. Thus,
  - $(C_i(n_i)k_i)$  is the cost of running  $k$  experiments on the  $i^{th}$  module.
  - If  $N$  represents the total complexity of a system, then  $n_i$ , the complexity of the  $i^{th}$  module would be given as,  $m_i/N$ , where  $m_i$  is the  $i^{th}$  module's contribution to  $N$ .
- $(Z_i)$  is the visibility cost, the cost to replace the  $i^{th}$  module.
  - Mathematically,  $Z_i = \sum_{j-sees-i} c_j n_j$ . It sums up the cost to redesign each  $j^{th}$  module containing  $n_j$  parameters that depends on (*sees*) the  $i^{th}$  module.
  - $c_j$  is the redesign cost per parameter for the  $j^{th}$  module.

## 6.1 NOV For Aspect Oriented Modularization

Based on the analogy between inversion and aspect oriented modularization presented in section 5.1, table 5 presents a model for the NOV of aspect-oriented modularization, comparing it with NOV for inversion.

$NOV_{inv}$	=	Option Value of architectural module
-		Cost of designing architectural module
-		Option value lost in hidden modules' experiments
+		Cost savings in hidden modules' experiments
-		Costs of visibility
$NOV_{asp}$	=	Option Value of Aspect Module (Aspects and design rules for aspects)
-		Cost of designing aspects and design rules for aspects
-		Option value lost in scattered code's experiments in hidden modules
+		Cost savings in scattered code's experiments in hidden modules
-		Costs of visibility of modules on design rules for aspects

**Table 5: NOVs for inversion ( $NOV_{inv}$ ) and aspect oriented modularization ( $NOV_{asp}$ ).**

Baldwin and Clark have defined NOV expressions for all the six modular operators. In evaluating the design options, we have not used these individual expressions. We believe further work is needed for these individual NOV expressions to be directly used in evaluating various forms of fine grained design changes made in software. Instead, we use the generic expression for NOV (discussed earlier in this section) that is applicable to any modular design.

## 6.2 Assumptions For NOV Analysis

The main objective behind our NOV analysis is to compare the difference between the values of the different designs, rather than to assess the individual worth of the design in terms of a market value. We believe our assumptions give us consistent values for comparing the different designs.

We omit *external parameters* as modules for NOV analysis because they are not subjected to further experimentation. We treat all parameters under *design rules* as a single module. All other design parameters are treated as individual modules. Our assumptions for rest of the parameters are given below.

### 6.2.1 Redesign Cost Per Parameter ( $c_i$ )

We assume the redesign cost of a single module to be 1 (following [8]).

### 6.2.2 Technical Potential ( $\sigma$ ) Of A Module

A fundamental relation between the technical potential  $\sigma$ , and cost  $c$  for (re)designing individual modules comes from the *break-even* assumption of one experiment on an unmodularized system [8]. This assumption says that in an unmodularized system (or a system with only one module),  $\sigma N^{1/2} Q(1) - cN = 0$ . With this relation we can assume the maximum value for  $\sigma = 2.5$ , as we have assumed  $c_i$ , redesign cost of a single module to be 1 and  $Q(1) = 0.4$ .

Based on the observations made for  $\sigma$  in [23] we can say that  $\sigma$  should be higher for those modules that add more value to the system. [23] estimates the value for  $\sigma$  as the

Design Parameter	Tasks	#
ExternalParameters	-	0
DesignRules (StoreLocator)	Provide structures that model start location, address matches, directions and map	4
DesignRules (WineryLocator)	Provide structures that model start location, address matches, winery search option, tour and map	5
StoreLocator	locate addresses, list stores, provide maps, map navigation, provide directions	5
HttpSessionStoreLocator	authentication	1
locate	list location, specify starting address	2
display	list store, map, map navigation	3
directions	list directions	1
web.xml	application configuration	1
AddressLocator	list locations	1
AuthAddressLocator	authentication	1
WineryFinder	list wineries, provide options for wineries selection	2
RouteMapHandler	generate maps, provide navigation, list directions	3
AuthRouteMapHandler	authentication	1
startWineryFind	list location, specify starting address	2
searchWinery	set winery search options	1
tour	list wineries, map, map navigation	3
directions	present directions	1
WebServicesLogger	implement logging	1
Logging (Aspect)	provide logging	1
Authentication (aspect)	authentication	1

**Table 6: Task list used to calculate the complexity ( $n_i$ ) of individual modules. (# denotes number of tasks.)**

system technical potential scaled by the fraction of the *Environment Parameters* relevant to the module. Since the external parameters we have listed represent a category of environment parameters, we assume  $\sigma$  of a module to be dependent on the number of external parameters it depends on. Secondly, we also assume  $\sigma$  to be dependent on the module’s relevance to the end users of the system. A module that an end user directly interacts with or, benefits from, is likely to add more value than a module that is hidden from or irrelevant to the users. With these assumptions we are simply elaborating the observations made in [23] for estimating  $\sigma$ .

With these assumptions, we define the technical potential to be a function given as

$$\sigma_i = f(e_i, p_i) = (e_i + 1) \times p_i$$

where  $e_i$  is the number of external parameters that the  $i^{th}$  module depends on.<sup>12</sup>

$p_i$  indicates the  $i^{th}$  module’s relevance to end users of the system. Users directly interact with the *application controller* modules, thus we assign them a  $p_i$  value of 2. *Func-*

<sup>12</sup>We add 1 to  $e_i$  as we do not want to assign  $\sigma$  a value of zero just because a module does not depend on the external parameters.

*tional modules, subsidiary modules* and *design rules* are less visible to the users so we assign them a  $p_i$  of 1. We assume the value of  $p_i$  to be 0 for *web.xml* since this parameter is merely a configuration file and does not contribute much to the system’s functionality and the end users.

We scale the value of  $\sigma$  for all the modules, with 2.5 as the maximum value.

### 6.2.3 Module Complexity ( $n_i$ )

We assume  $N$ , the complexity of the whole design, to be the total tasks performed by all the modules in the system. We calculate the complexity of a module by dividing the total number of tasks it performs by  $N$ . Table 6 lists the number of tasks for each module that we used to calculate the complexity of individual modules.

Design	Fig	ID	NOV	$I_c$ %	$I_s$ %	$I_{w1}$ %
StoreLocator	1	s	0.72	NA	NA	NA
WineryLocator	3	w1	1.38	91.41	91.42	NA
WineryLocator with Logging	4	w2	1.41	2.18	95.6	2.18
WineryLocator with design rules for application	5	w3	1.59	12.59	120.2	15.05
WineryLocator with Aspects	6	w4	1.76	24.55	143.6	27.28

**Table 7: NOVs for different design options. ( $I_c$  = Cumulative increase in value,  $I_s$  = Net increase in value with respect to ‘s’,  $I_{w1}$  = Net increase in value with respect to ‘w1’.)**

## 6.3 Observations

The result of the NOV analysis is shown in Table 7. The NOV increased with each of our subsequent designs, but the highest increment was observed after aspect oriented modularization.

A closer look at the NOVs of all the modules for different experiments gives more insight on the effect aspects have on the value of the overall design. Figure 11 compares the option values of the modules before and after introducing aspects in the design.

We can see that after aspect oriented modularization the NOV curves for existing *application modules* (such as *RouteMapHandler*) go down. The increase in overall NOV for *WineryLocator* after aspect oriented modularization is due to the NOV of newly introduced aspects, *Logging* and *Authentication*.

An important observation to make is the NOV curve for the module *WebServicesLogger* that goes up after aspect oriented modularization. This increase is significantly higher than the NOV changes for other modules, but does not yet contribute to the total NOV of the system, because the curve stays below 0 in both of the graphs. A slightly higher value of  $\sigma$  for *WebServicesLogger*<sup>13</sup> can make its curve go above 0, resulting in a higher NOV for the application.

We can conclude from these results that aspect oriented modularization makes augmentation more profitable even if the added modules have comparatively low technical poten-

<sup>13</sup>Our assumptions led to a lower technical potential of 1 for *WebServicesLogger*.

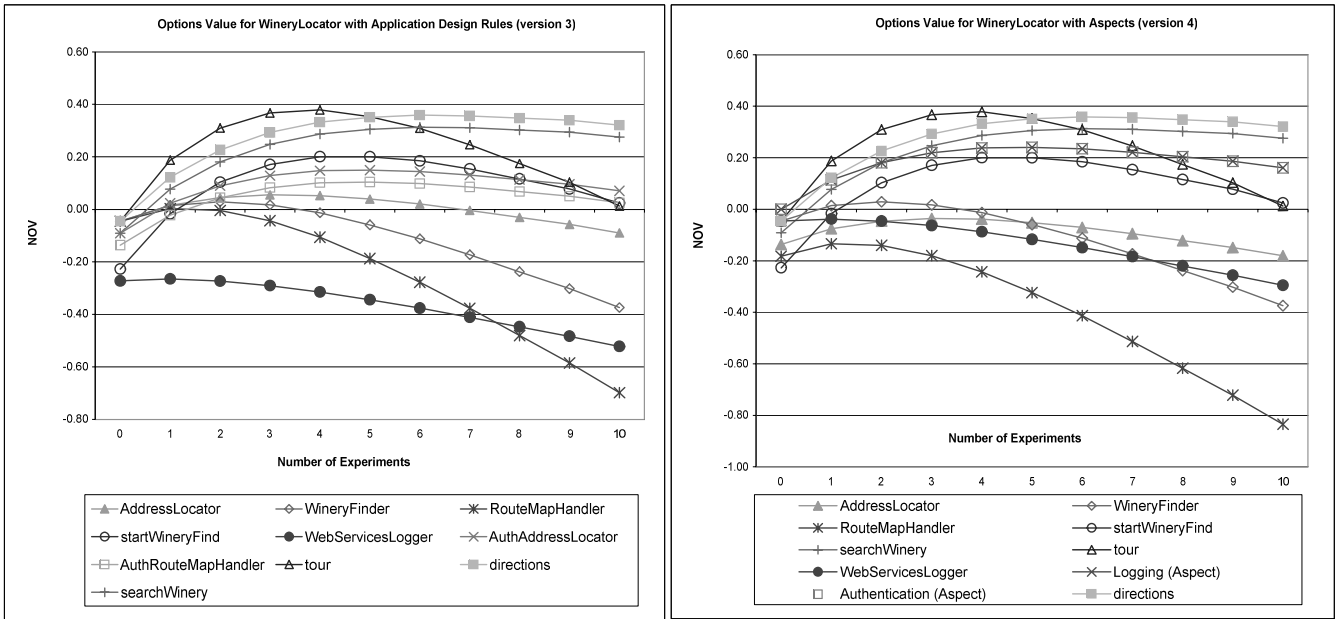


Figure 11: Charts showing the effect of aspect oriented modularization on option values.

tial. Without aspects, the newly added modules will need to have higher technical potential to achieve the same increase in the overall value of design. In short, aspects add value to an existing design.

## 7. LIMITATIONS AND FURTHER WORK

The assumptions we made in section 6.2 lack formal or empirical verification; they are based on assumptions made in [23], as well as on our own reasoning. Standard techniques to estimate parameters for NOV and richer models for NOV for software are issues of further research in this field.

Our analysis of NOV for the various designs is based on the general expression to calculate NOV of a modular design rather than the NOV expressions for the individual operators. We followed this approach because we still need to have a precise understanding about how a NOV for a design change resulting in a different dependency structure, other than the modular operators model, should be expressed in terms of these individual NOV for the operators. For example, we cannot exactly pick which operator models the design change we presented in section 4.5 (from figure 4 to 5). We can treat it as a refinement of *inversion*, that previously had created the *design rules* in figure 3 and 4, but accurate evaluation cannot be done without a precise NOV expression that models this change in the design. Such problems arise as we move into finer granularity of design changes and thus need to investigate the feasibility of modeling and evaluating finer design changes with Baldwin and Clark's theory (for example, various form of small and big refactorings [11]). We need to understand how far (or deep) we can go with these basic operators, and in particular with NOV, in considering fine grained design changes in software.

In this paper we have only considered two aspect oriented techniques, pointcut-advise and introductions. We intend to investigate the structure of dependencies and modularity in several other models for representing aspects, using techniques and theories presented in related works, such as

[14], [20], [4], [15] and [12]. We believe such research efforts will contribute in discovering standard techniques to define design rules for aspects and help us understand the implications these design rules have on the overall value of an aspect oriented design.

We observed that it is tedious and error prone to work with DSMs and NOV analysis without proper tool support. We believe DSM can be implemented as an interactive and direct manipulation tool for software design. We intend to look into the issues of scalability and possibilities for alternative representations of design in DSM. We also plan to incorporate advanced features of DSMs such as numerical DSMs with dependencies classified according to their strength and modeling software with various types of DSMs using component-based and team-based DSMs [1]. With these, we intend to investigate whether DSM can be further extended as a tool for real software practitioners and designers.

## 8. CONCLUSIONS

Our work contributes to the earlier work by Sullivan et.al [23]. We have provided a realistic and modern example, and presented an intuition to estimate values for parameters such as technical potential. Our main contribution, however is our analysis of aspect oriented modularization and its effect on the value of the overall design. We have demonstrated that: (i) DSMs are capable of modeling dependencies in design including those with aspects, (ii) Design changes can be expressed in terms of the modular operators and (iii) We can perform NOV analysis to compare design alternatives. Finally, we have observed that introducing aspects increase the value of an already modularized design.

## 9. REFERENCES

- [1] Tutorials and resources on DSM, DSM web site <http://www.dsmweb.org>.

- [2] AspectJ project web site. <http://www.aspectj.org>.
- [3] MapPoint Object Model, available from MSDN online <http://msdn.microsoft.com/>.
- [4] Concern Manipulation Environment (CME), project web site <http://www.eclipse.org/cme/>.
- [5] The Apache Foundation. Apache AXIS. <http://ws.apache.org/axis/>.
- [6] Sun Microsystems. Java Servlet Specification. <http://java.sun.com/products/servlet/>.
- [7] Sun Microsystems. J2EE, Java 2 Enterprise Edition Specification. <http://java.sun.com/j2ee/>.
- [8] C. Y. Baldwin and K. B. Clark. *Design Rules vol I, The Power of Modularity*. MIT Press, 2000.
- [9] M. Fowler. Inversion of control containers and the dependency injection pattern. <http://www.martinfowler.com/articles/injection.html>.
- [10] M. Fowler. Module assembly. *IEEE Software*, 21(2), March 2004.
- [11] M. Fowler, K. Beck, J. Brant, O. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999.
- [12] W. H. Harrison and H. L. Ossher. Member-group relationships among objects. Technical Report IBM Technical Report RC22048, April 2002.
- [13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th European Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.
- [14] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.
- [15] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP 2003—Object-Oriented Programming 17th European Conference*, pages 2–28. Springer-Verlag, July 2003.
- [16] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.
- [17] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.
- [18] D. L. Parnas. On a "Buzzword": Hierarchical structure. In *Software pioneers: contributions to software engineering*, pages 429–440. Springer-Verlag New York, Inc., 2002.
- [19] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.
- [20] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 406–416, New York, May 19–25 2002. ACM Press.
- [21] D. Sharman and A. Yassine. Characterizing complex product architectures. *Systems Engineering Journal*, 7(1), 2004.
- [22] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., 1996.
- [23] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108. ACM Press, 2001.
- [24] W3C. SOAP (Simple Object Access Protocol) version 1.2 specification. <http://www.w3.org/TR/soap12>.
- [25] W3C. Web services description language (WSDL). <http://www.w3.org/TR/wsdl>.