Isolation and Dependency Resolution of Presentation, Processing and Persistence

Mehrab Monjur
Marquette University

ISOLATION AND DEPENDENCY RESOLUTION OF PRESENTATION, PROCESSING AND PERSISTENCE

by
Mehrab Monjur

A Thesis submitted to the Faculty of the Graduate School,
Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Master of Science

Milwaukee, Wisconsin

Dec 2009

ABSTRACT
ISOLATION AND DEPENDENCY RESOLUTION OF PRESENTATION, PROCESSING AND PERSISTENCE

Mehrab Monjur

Marquette University, 2010

For business application development it is important to isolate programming efforts of the concerns: Presentation, Processing and Persistence. Development of each of these concerns has an independent thinking process and requires somewhat different programming languages and development tools. In order to isolate the concerns, we provide passages between the concerns and control the flow of execution by following essentially three rules: 1. Presentation and Processing are coroutines, 2. Processing is finished before Presentation can begin to show output, and 3. Persistence is a subsystem of Processing. We explain how these rules come to existence, and what the implications are in the thesis. By following the rules, a Turing complete Presentation capable of pulling web resource from the server interactively cannot lead a programmer to write code tangling Presentation and Processing. We analyze our design and develop systems in Web 1.0 and Web 2.0 settings. For the latter setting, the system has multiple business component/service dependencies where some of the components run in the browser and some in the server. We show that such distributed component dependency can be resolved while keeping the isolation in place.

ACKNOWLEDGMENTS

Mehrab Monjur

I am dedicating this thesis to my mother Hamida Begum and my father Monjurul Hoque. Their prayers and love give me strength. I am thankful to my wife, Jannatul Ferdous, for her love and support, and to my two years old daughter, Mahdiya, for making me feel like a father.

During my stay at Marquette University, two people have changed my life, Dr. Douglas Harris and Dr. Sheikh Iqbal Ahamed. I am thankful to Dr. Harris for making me feel special. His art of learning and style of writing have great influence on me. I am thankful to him for helping me do research on ‘separation of concerns’ and specifically, for pointing out the PreProPer system. ‘Separation of concerns’ is a topic worth spending the rest of any mortal life. I hope I will. Dr. Iqbal showed me the art to find a problem and the importance of finding the problem and he taught me how to write in a logical sequence.

I would like to thank Dr. Praveen Madiraju for helping me out in my thesis.

I would like to thank Edsger Dijkstra for talking about ‘separation of concerns’, Melvin Conway for defining the term ‘coroutine’, and John P. Morrison for Flow Based Programming.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS AND ELABORATIONS

Term	**Elaboration**
AOP	Aspect Oriented Programming
BPEL	Business Process Execution Language
CDL	Choreography Description Language
CFG	Context Free Grammar
FBP	Flow Based Programming
GWT	Google Web Toolkit
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MDV	Model Driven Visualization
MVC	Model View Controller
OOP	Object Oriented Programming
RPC	Remote Procedure Call
SOA	Service Oriented Architecture
WS	Web Service

I. INTRODUCTION

``Separation of concerns'' [Dijkstra(1979)] is a fundamental practice in computing to solve a problem. The idea is to divide a problem into almost independent modules [Parnas(1979)] and to conquer them in such a way so that they are strung together in the form of a `string of pearls' [Dijkstra(1979)]. Such a system is reusable, adaptable to evolution, and provides clarity of understanding. While separating concerns we take into consideration issues like abstraction, encapsulation, and loose coupling, we often forget that some of the concerns need to be separated because they require isolated development endeavor by people of different skill sets. In building software for business data processing the core such concerns are Presentation, Processing and Persistence. Apart from the fact that these concerns have independent thinking process, they are performed by people of varied background who use somewhat different programming languages and development tools. Thus the need is to isolate development task “in order to reliably produce separation in practice” [Harris (2009)] which will bring much needed productivity from the business standpoint.

We look at “separation of concerns” for business application development in the world wide web setting. Business service providers acknowledge the increasing trend of ubiquitous data dissemination and they are developing web applications to make data available and accessible readily to their users who may use heterogeneous platforms. Generally, web applications are server side resources which upon HTTP request by a browser can be made available in response. However today one the one hand browsers have come a long way and we see the coming of browser based operating system; on the other hand, servers can be thought in terms of clouds where web resources can be made

available to multiple edges of the globe instantly. Most importantly, the web applications are no longer "typical Web 1.0 page-at-a-time applications", as Bruce Johnson pointed out [Johnson(2009)], rather they are rich and interactive. This is something that we come to know as Web 2.0. In such interrelation isolating development tasks poses a new set of problems.

The new problem stems mainly from the fact that currently entire business application can be run in the browser. Using only JavaScript and HTML one can write an application and run that in the browser. Specially with the facility of Google Web Toolkit [GWT(2009)] where a programmer can write Java code and then cross-compile the code to JavaScript, such application can be developed easily. Earlier Processing took place in the server and whatever Presentation it generated got rendered in the browser unaltered. Presentation was passive, stateless and were generated one at a time, that is, any new request created a new response and a page refresh. With interactive Presentation it can request new resources and can alter the existing resource itself. Thus, “what we have got here is a failure to” prevent code tangling Presentation and Processing.

Another problem, which is although similar, is worth pointing out separately, and it is component dependency. In previous web application setting all component dependency were resolved by the Processing in the server whereas now browsers can run a component that another component in the server may be dependent on. Thus, if we look at the problem scenario closely, we see that the task of isolating Presentation and Processing has impact depending on where they are run whether in the server or the client. This is because the environment changes their capability.

Earlier T. Parr [Parr(2004)] separated model (Processing and Persistence) from view (Presentation) and used the term ‘enforce’ to achieve ‘strict’ separation. He showed that ‘strict’ separation can be realized following essentially two rules: 1. the template (HTML presentation layout) must not follow an unrestricted grammar and 2. view cannot pull data from model. His work is a central part to provide problem background of this thesis. However, the work becomes limited in two settings: 1. the interactive Presentation which is capable of pulling data by an AJAX call and 2. the prevalent use of Turing complete JavaScript to manipulate the HTML DOM.

We already have in place a software system Pre-Pro-Per [Harris(2009)] which conceives projects as divided into three tiers for Presentation (Pre), Processing (Pro), and Persistence (Per) (thus Pre-Pro-Per) where the communications between the tiers take place via two passages called PrePro and ProPer. Programming within each tier can be done in any way appropriate for the problem in relative independence from the other tiers, and the user does no programming within the passages, but merely configures them using standard configuration means like XML. A large part of the thesis is about understanding Pre-Pro-Per from a practitioner’s perspective. By way of example and theoretical argument, we extract the rules of isolation that made the system to work in the first place. We follow the acronyms used by Pre-Pro-Per for Presentation, Processing and Persistence in the rest of this thesis.

A. Contributions

Our contributions are mainly two folds:

1. We layout the rules of isolation of Presentation, Processing and Persistence.

2. For components capable of running both in the browser and in the server, we identify and classify components depending on their execution environment. We identify for each environment what Pre or Pro can do. We also resolve their dependency while keeping isolation.

Apart from these contributions, while building the system and analyzing the requirements for isolation we find the importance of 'coordination languages', like FBP [Morrison and Morrison.(1994)], Linda[Gelernter(1985)] that communicates asynchronously with modules, for 'separation of concerns'. We identify the importance of coroutines for 'separable programs' [Conway(1963)]. For the new Pre-Pro-Per extension we use Google Web Toolkit's development facility. This gives us opportunity to look closely at the statement: "whether JavaScript was a good language in which to write business-critical applications" [Johnson(2009)].

B. Outline

We divide our background into two problem and domain. The problem background is related in Chapter II. In Chapter III, we see the domain background which talks about the environment where we want to resolve the problem. The problem has implications that go beyond the domain and the domain has concerns that are more than the addressed problem.

Chapter IV gives a brief overview of the PreProPer system [Harris(2009)].

The topic of Isolation is addressed in Chapter V and VII. The first one is for 'programming in the small' and the second one is for 'programming in the large' [DeRemer and Kron(1975)]. Some of the problems of isolation and therefore, the

implementation is different for each domain, ‘small’ and ‘large’. Chapter VI describes the system developed for ‘small’ and Chapter VIII describes the system developed for ‘large’. Finally, Chapter IX describes future directions for the research and development and a brief conclusion of what we have addressed and resolved.

II. PROBLEM BACKGROUND

We divided the background study into two: problem and domain. The domain is the environment or the setting where we look at the problems. This chapter talks about the problems that are addressed in the thesis. The first problem is about how we isolate the development of Presentation, Processing and Persistence. The second problem is how do we resolve component/service dependency and where do we position dependency whether in the browser or the server and in that respect how do we isolate Presentation and Processing where both are capable of processing.

A. View and Model Separation Requirements

T. Parr [Parr.(2004)] made a formal study of template engines and layed out rules for 'strict' separation of model and view. He defined restricted and unrestricted templates and showed that unrestricted templates cannot be used in order to separate model logic from view. According to the definition unrestricted templates are comprised of output literals and action expressions. If the expressions are 'unrestricted computationally', it "may embody a Turing machine" and thus can modify the model. Then he uttered the real concern: "If a template can modify the model, it is part of the program". Unrestricted templates are equivalent to context free grammar capable of generating context-free languages like XML. The capabilities are limited to referencing attributes and templates. T. Parr [Parr.(2004)] concluded that adding context-sensitivity to the template that is how non terminals can be expanded depending on certain context, is an added benefit but does not violate separation. He came up with 5 rules of separation. For our system to work

both in the 'small' and in the 'large', we need to follow these rules. For ease of following the rules in this thesis we are re-itemizing those:

1. View cannot modify Model.
2. View cannot compute on dependent data values.
3. View cannot compare dependent data.
4. View cannot make data type assumptions.
5. Model data cannot have display information.

Apart from the last rule we follow the other 4 rules. In section VI E we discuss these rules elaborately.

While the separation rules are important for us, there are several other issues that we need to look into which go beyond the issues that are addressed.

Problem 1: T.Parr does not divide model into Processing and Persistence. Does the rules for separation give different perspective?

Problem 2: T. Parr [Parr.(2004)] does not mention how to incorporate JavaScript in the template which are essentially Turing complete. JavaScript, XSD are all Turing complete and use of them are prevalent in current browser. It is no wonder that some processing task can be handled in the view if it uses JavaScript.

Problem 3: In [Parr.(2004)] template engine generates view which is ultimately rendered in the browser. A JavaScript overlaying such view can easily manipulate HTML DOM. The power to manipulate HTML DOM is not a problem as it does not entangle model. However, current Web 2.0 with RSS and ATOM technology and with the growing use of Ajax, Presentation can call services in the Processing. In such a situation does not Pro become a subroutine of Pre?

1. Related Works

Model Driven Engineering:

There are important takeaways from Object Management Group's (OMG) Model Driven Architecture [OMG(2001)]. The main objective of MDA is, "separating the business part from the platform part of systems in order to independently control their evolution" [OMG(2001)]. The hierarchical abstraction of problem domain and platforms are achieved by having metadata between the two and by applying metadata transformation. Thus they make an important point, "Interoperability in heterogeneous environments is ultimately achieved via shared metadata" [OMG(2001)]. More general and something which is not standardized by OMG is Model Driven Engineering [Kent.(2006)]. It is a paradigm shift from object oriented programming (OOP) where "Everything is an object" to "Everything is a Model".

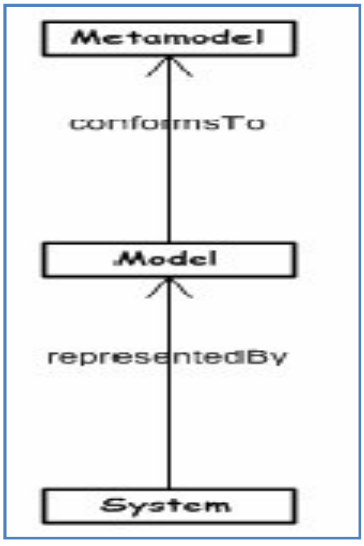

Figure II-1 Basic notion of a model in MDE [Bézivin(2005)].

In [Bézivin(2005)] we see the difference between the notion of object and model, and understand the basic principles to identify a model. A model is a representation of a system and it conforms to the languages defined by its met model.

For the concerns Presentation, Processing and Persistence, let us discuss if object oriented programming has all the right answers. If we take for example Processing, we see that using OOP we can define it and can put relationships with Presentation and Processing. However, it has a view which is specific to business rules and logic and it can be represented by certain languages or tools that the organization platform can support. Thus each of these concerns can be viewed as a Model according to MDE. This is where the importance of the work of rule based engines come. Because business rules can be written by the people who knows it and then it can be run by the platform that the organization provides [Leannah(2006)].

Model driven visualization (MDV) [Bull et al.(2006)] has some of the answer to what we are doing. It uses the MDE principle to move focus of realizing visualization tool from implementation details to customization. Instead of presentation and processing it talks about domain and view correspondingly. Domain can generate meta-models that can be transformed to other meta-models which is realizable by viewers. A View is a concrete implementation by a viewer.

WebJinn [Kojarski and Lorenz(2003)] addresses the issue of inter and intra cross cutting that takes place in view and model implementations. The work is focused toward JSP implementation. It has notes on where tangling can take place: “functionality”, “presentation”, “control” and “structure”. The inter cross cut is because of tangling of

structure and the inter cross-cutting is because of the tangling of other three concerns. It provides a framework that is applicable in a JSP page.

MVC and Frameworks:

Any talk of Templating frameworks will always start by mentioning model view controller [Krasner and Pope(1988)]. For use view is Presentation, model is Processing and Persistence, and the controller is part of Processing. However, we will see later that the passage that we have between Presentation and Processing acts like a controller. It has an input controller which decides which Processing service to use and an output controller which decides which template to use for a Presentation.

MVC based frameworks like struts, spring provide development ease by providing guidelines and interfaces that need to be followed and implemented for view, model and control code to be separated. However, tangling of model and view cannot be avoided because view can reference model attributes/beans and can execute actions/expressions. These may lead some model code be dependent on view. Other such violations include model data type assumption by view. T. Parr [Parr 2004] showed 5 such common view model entanglement criteria and he showed that the template engines such as “Tapestry, WebMacro, Velocity, PTG, UniT, Tea, WebObjects, FreeMarker, ColdFusion, Template Toolkit, and Mason” violate more than one criteria.

B. Dependency Resolution

Components hide their implementation details behind their interfaces. An interface gets exposed and other components read or write in it. This is interaction of components. Sometimes a component generates an event that another component

receives. These interactions among components create dependency. According to Service Oriented Architecture, we can think of these components as services. To us service is more granular than a component, in the sense that component may be a composition of services. However, we will use service and component interchangeably.

When we think of building a business application of multiple components where some of the components may even cross the trusted boundaries of the organization, the issue of coordinating and resolving dependency is important. For us in such situation it is not just about resolving component dependency but to position them in such a way so that isolation is preserved.

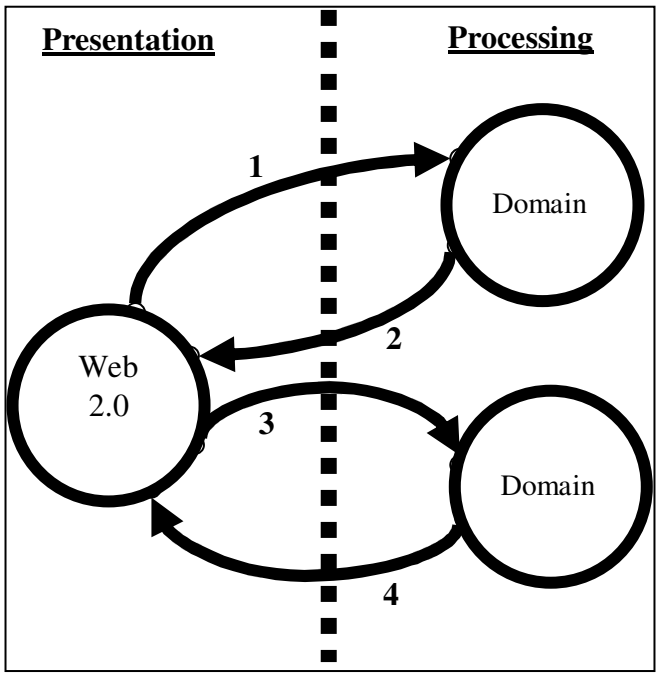

Figure II-2 Presentation mediates the dependency

We want both Presentation and Processing Turing complete, this follows that any component that can be done in Processing can also be done in Presentation. Thus a

component in Processing can be dependent for its input on a component in Presentation and vice versa. Let us look at two problem scenario and later see what the solution may look like.

Figure II-2 shows three components where one of them is in Pre and the other two are in Pro. Both components in Pro domain are dependent on the component in the web. The opposite is also true because the web component requires output from both the other components in Pro (as shown by 2 and 4 edges in Figure II-2). It is to note that both the components in Pro is dependent on each other indirectly. The component in Pre is mediating their dependency. If all Pre needed was the output from the components, the need to do some processing is violating separation and also hampering performance because of the extra communication. Finally it is making Pro a subsystem of Pre.

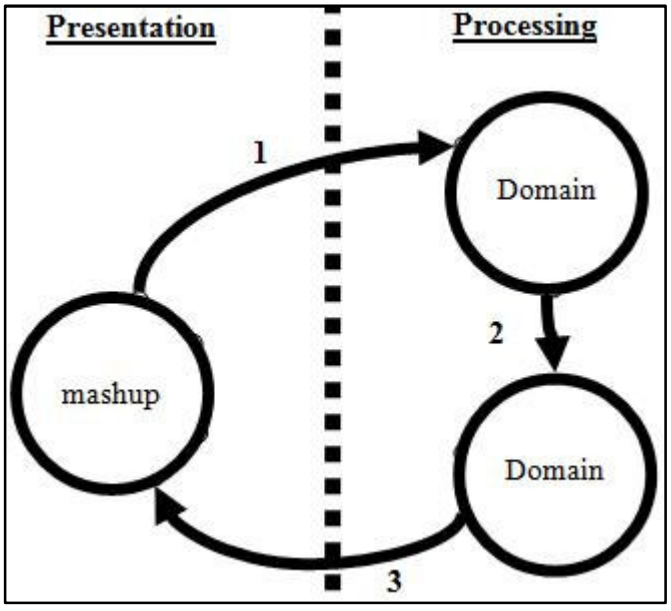

Figure II-3 Processing alone handles the dependency.

In Figure II-3, the component in Pre gets the output from the second domain. It does not resolve dependency and do not need to do processing. The first component resolves the dependency which was earlier handled by Pre in Figure II-2. In that figure the web component needed outputs from both the component which is now facilitated by the second component. That is it gets the output from first one and gives it to Pro with its own output. In a Servlet, CGI, or JSP setting this is normally the case because the Pre gets to be generated when all Pro stuff is finished. Thus Pre need not be Turing complete. However, when Pre has the opportunity to make AJAX calls and it has a language like JavaScript it can create situations like the one shown in Figure II-2. This is the problem of component dependency. The problem is where to position the dependency to resolve them. Is there any other way to look at this dependency?

1. Related Works

[Sharma(2009)] presents a linked list based approach to represent component dependency and provides metrics to identify how complex a system is in relation to component dependency.

In this thesis, we look at how components capable of running in browser and server interact, which is distributed. [Gomaa(2009)] looks at different reusable patterns for distributed component interconnections. In a distributed system components interact based on synchronous, asynchronous or brokered communication. For us in order to make Pro a coroutine of Pre we need to have the connections between the two asynchronous.

In Service Oriented Architecture there are two terms called choreography and orchestration [Peltz(2003)] which are worth pointing out to describe our system in the

'large'. Orchestration is about executing multiple processes. All the input and output specification needs to be known priory to position them in a laguage in a coordinated way for orchestration. Business Process Execution Language, BPEL in short, is an orchestration language [BPEL(2007)]. It is also possible to write BPEL using Java [Juric(2005)]. In choreography every component defines how it will interact with other component and then we find the overall contract information of the processes. This means every party has a say to the ultimate coordination. The choreography language or specification follow a grammar but it is not meant for execution. Web Services Choreography Description Language (WS-CDL) is an choreography language [Kavantzas et al.(2005)].

C. Coroutine and Subroutine

We are now going to talk about a term which will give us a solution space for the problems as outlined earlier. We look at the term **coroutine** [Conway(1963)], first mentioned by Conway in 1963. We will just quote some of his writings so that we may understand what a **coroutine** is.

"*…each module may be made into a **coroutine**, that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, **eaeh acting as if it were the master program when in fact there is no master program**. …*

The coroutine notion can greatly simplify the conception of a program when its modules do not communicate with each other synchronously."

He writes all these in a section called “Coroutines and Separable Program.” Once we understand that, we see the problems with Figure II-2 and II-3.

III. DOMAIN BACKGROUND

Our implementation focus is on Web 1.0 and the interactive web, which we come to know today as Web 2.0. There is a lot of hype in terms of whether it is really a new trend in the computing world or it is something that is already in place. However, we describe what the community and we think about it and try to delineate its scope and the future implications.

A. Web 1.0 to Web 2.0 Transition

HTML is the hyper text language that has been serving our browser interface. The browsers know how to parse it and how each tag needs to be rendered. With cascading style sheet the interface 'look and feel' is easy to modify. With JavaScript it became 'reactive' [Raman(2009)]. And with AJAX [Murray(2005)] the web has become interactive.

There is a growing trend of shifting processing from server to the client. Part of the answer to the difference between Web 1.0 and Web 2.0 lies in where the processing gets done. Other part lies in how the communication takes place. The capability of the browser itself and its JavaScript support determines the processing power.

Although it is difficult to define Web 2.0 which may have other issues involved, for the thesis Web 2.0 means processing by JavaScript and communication by Ajax calls. Another important thing about Web 2.0 is that there will be different webparts involved in a single page which has multiple domain dependency. Although 'Same Origin Policy' dictates that a web page instance can only access data from one site, a server can deploy a site that can serve as an endpoint to mitigate and serve multiple domains.

B. Mashups

Web 2.0 has given birth to a new form of web application called mashups. It is the new form of earlier portals where each web part in a web application represented a different source. Mashups can interact to multiple resources and aggregate the data to produce new effect. They are different from portals because of their aggregation capability. [Programmable Web (2009)] lists more than 5000 Mashups.

Most of the time a mashup web interface gives the user the capability to choose from various sites, feeds and ATOMs. To let this happen, the presentation is made more user friendly by extensive use of JavaScript and Ajax.

Discrete parts of the web get integrated in a Mashup in order to serve the user's need. It gives the opportunity to make a product which has multiple domain dependency. This makes a site a single-point-of-focus of multiple domains. To resolve dependency and to transform data from all of the sources, mashups need to do a lot of processing in the browser. Thus the problems outlined in section II B need to be resolved for mashups.

[Riabov et. al(2008)] describes a mashup called 'wishful search' where users can choose and search for different feeds from different sites and the mashup can compose them together according to the user's choice of aggregation. It gives the user the chance to define a flow. The flow shows how the output from one service/feed should go to the input of another feed. In it, Presentation defines the flow and help the user draw the flow diagram, and Processing executes the flow in the server to produce the desired output.

Damia [Simmen et. al(2008)] is a system to help users create Mashups to combine data from sources like desktop applications, resources from another web sites and other

feeds using Ajax. All information is consumed and converted to a standard XML format and a flow based algorithm combines their effect.

Marmite [Wong and Hong (2009)] is a mashup where data is processed using a series of operators that are placed like unix pipes. Marmite has a rich web interface to select operators and to draw the data flow connections. This is actually end user programming.

C. Java to JavaScript

Browser based applications have come a long way from ``typical Web 1.0 page-at-a-time applications", as Bruce Johnson pointed out [Johnson(2009)], to the rich interactive web applications, the Web 2.0. Thanks to JavaScript this interactivity is possible. To write JavaScript with more ease we find GWT [GWT(2009)] which has a cross-compiler that converts java code to JavaScript. Similar to GWT is [XML11 Puder.(2007)] which can compile object code (from .NET or Java) to javascript.

1. Google Web Toolkit

The vision of GWT is to enable development ease for business applications that will run in the browser. It has great relation to the motivation of Web 2.0. [Johnson(2009)] is a good summary of the motivation of GWT, the reason for choosing Java as the language, and pointed out the challenges involved to build the toolkit. Bruce says, *"Instead of being implemented as a sequence of individual HTML "pages" rendered by the server, Wave might be described as a client/server application in which the client is a browser executing a JavaScript application, while the server is 'the cloud.*"

We use GWT in our project. The reason for involving GWT in this project is that it does a wonderful job of letting the code be written in a language such as Java with all the code development tools available, and then compiled and sent to the browser as JavaScript which will run in that particular setting. This means that Pro can maintain responsibility for proper code, and be tested for following business rules, and scripts then can be shipped out to the browser just to run.

GWT has rich set of web application resources known as widgets. It is easy to define layouts and draw on canvases. GWT has made GUI development similart to any desktop application user interface design. Smart GWT [SmartGWT(2009)] provides wide variety of rich interactive widgets like data grids, windows, stacks, portlets etc. Data Binding to these grid is possible using Java Script Object Notation, XML and using the plain old collections, termed as DataRecord.

Ajax development in GWT is about Remote Procedure Calls(RPC). As we have shown in figure II-2 it is really possible to use processing as a subroutine of presentation using Ajax.

Because of GWT it is possible to develop complex applications. Google Wave [Wave(2009)] is such an example. However, we see it as a difficulty as Client side business rule processing means that the end-user not only knows what to expect from the business but also knows how the business is run. And in the client side an attacker can inject their own code (using JavaScript) to compromise the legal representation of the organization, that is Persistence.

IV. THE PREPROPER SYSTEM

The thesis came to existence by our work with PreProPer System developed by Dr. Douglas Harris who is the architect of the system. We worked as a practitioner of the system, figured out what it is capable of and why it is that it works so fine. Thus to understand the rest of the thesis it is essential to give a details of how we interpret the system.

A. The Roles

We need to look back and think of how a system gets produced, and who has the responsibility. There is a simple map to be considered, and the acronym MAP (Manager, Architect and Programmer) actually reminds us who is involved.

Manager: This is the role of responsibility for choices relating to the ultimate operation of the entire system, with focus primarily upon the results in relation to the operation of the organization which sponsors the system. In this role the details of what is done inside are much less important than what happens from the outside perspective.

Architect: This is the role of accepting the needs and goals for the system as determined by the manager, and determining how the data flows and the processing is done, in order that the goals set by the manager are accomplished. The architect follows the direction of the manager, and depends upon the activities of various programmers. In our scenario it is the manager that separates the concerns, at least those of Pre, Pro and Per. The architect determines tasks as related to these concerns, and attempts to keep each task related to only one concern. Then the individual tasks are assigned to programmers (or teams of them), and in fact the teams might end up being composed of people with

very different sets of skills. Of course the system has to have substantial flows of communication between the programs that accomplish these separated tasks, and the passages are the channels through which that communication flows.

Programmers: The programmers for the different tiers may have very different backgrounds: they automatically deal with different concerns. We divide them into three types.

Webbie: The concerns here are the appearance of things, from the output side, and interpreting the user wishes, from the input side.

Middie: The concerns here are from the perspective of the organization, learning what the ”business rules” are and determining how to properly implement them.

Techie: The concerns here are the proper and reliable storage of data, so that it may be quickly stored and retrieved, perhaps simultaneously by multiple users. It will be the concern of the Middie that certain things are done together as a single transaction, for example, but it will be the concern of the techie to assure that the technical details of performing a transaction are formally done. In other words the Middie worries about who participates in a transaction, and the techie worries that what occurs actually is a transaction.

B. The Architecture

Napoleon Bonaparte said, “A good sketch is better than a long speech”. This is similar to the well known proverb, “A picture is worth a thousand words”. And this is what D. Harris [Harris(2009)] did to show the architecture of PreProPer.

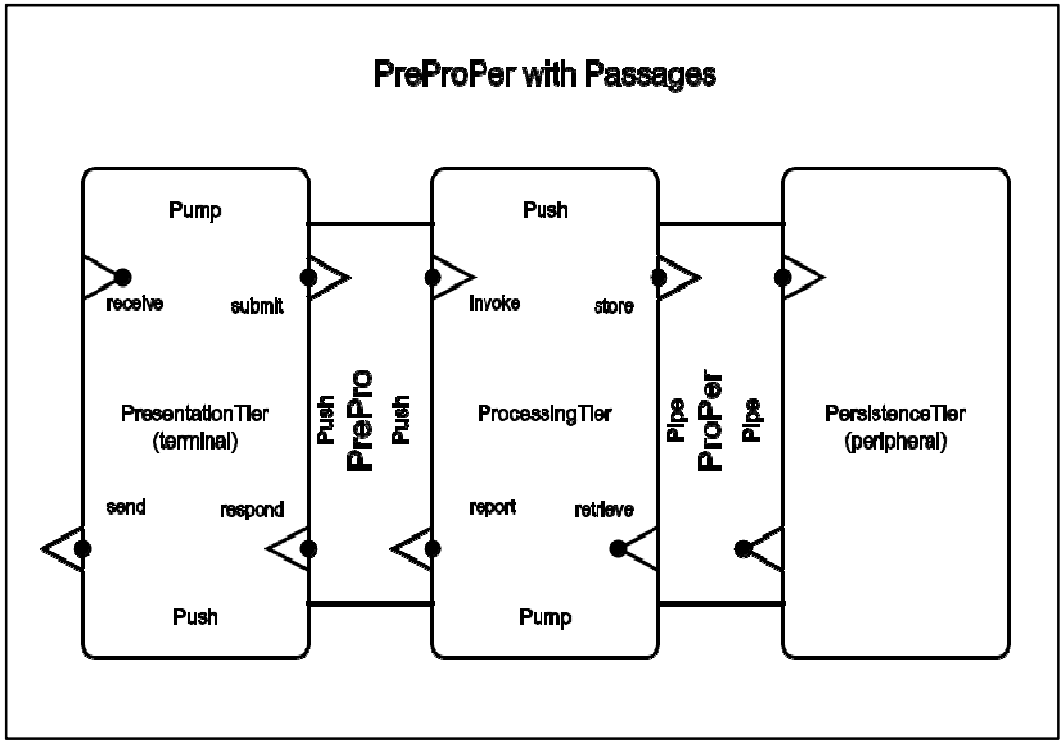

Figure IV-1 PreProPer with Passage as shown in [D.Harris(2009)].

It is obvious from the diagram that in between each tier, there are passages; we call them PrePro, one that sits between Pre and Pro and ProPer, sits between Pro and Per. The flow of data to and from an object is because of an action and it is important to identify the initiator and receptor of that action. We will talk in this section about the action shown in figure IV-1: receive, submit, invoke, store, retrieve, report, respond and send. To better understand the flow, we suggest a reading of PushPullPipePump by Douglas Harris [Harris(2005)].

Pre is listening or waiting to **receive** query from the client/customer. Upon reception of the query it will create **RequestMap** of the request parameter and push the map to the glue tier, PrePro by calling **PrePro.submit(what,how)**; the `what' argument tells about the data that has arrived and the `how' argument specifies a class of the Processing Tier that should process it. PrePro has configuration for processing. In the

configuration, prepro.cfg.xml, for each processing entry we have a name, a processing class, a service class, and a service method. Thus, in the implementation `how' argument points to a name of a processing entry.

Table IV.1 Configuration of Processing in PrePro

```
<prepro>
<processing>
 <name>processingName</name>
 <class>net.sks.preproper.processing.StraightThroughProcessor</class>
 <serviceClass>net.sks.preproper.processing.Service</serviceClass>
 <serviceMethod>provide</serviceMethod>
</processing>
</prepro>
```

The processing class is part of the PrePro tier; it uses proxy to call the method of the service class. Before and after **invoke** the processing class does tasks which concerns PrePro. One of the important task is to store reply path in PrePro and remove the path from **RequestMap**.

After **invoke** Pro starts business data processing. Of course, at first some controller of Pro decides which rules to run depending on the RequestMap}. While running the business rules, Pro uses **store** and **retrieve** to talk to its subsystem Per through the glue tier, ProPer. Pro stores some sort of Bean object to Per and wants to **retrieve** that way. Now there can be many Per both in number and in type. Per can be a flat file system or any big shot DBMS. Here ProPer comes to make life easy for Pro. All the manager has to do is to set the configurations straight for each type of Per and then the middies will get appropriate **SessionFactory** from ProPer and can think only about the Beans that it wants to **store** or **retrieve**. In the configuration for persistence there are entries like a name, a class and a resource file.

Table IV.2 Configuration for persistence in ProPer

```
<proper>
  <persistence>
    <name>sports</name>
<class>net.sks.preproper.proper.FlatFilePersistenceConfiguration</class
>
    <resource>someDir</resource>
  </persistence>
  <persistence>
    <name>users</name>
    <class>net.sks.preproper.proper.HibernateConfiguration</class>
    <resource>/hibernate-team.cfg.xml</resource>
  </persistence>
</proper>
```

During processing Pro fills up different Beans and from which different **ResponseMap** maps are created. When Pro becomes ready to deliver it calls **report(how, what)** method of PrePro. Indeed, `what' is the **ResponseMap** from which the reply will be created and which is the actual data that needs to be presented and `how' argument specifies the template name which knows how to format the reply. These templates are created by the Webbies and how to get to these templates is configured in PrePro by the manager.

Table IV.3 Configuration for presentation in PrePro

```
<prepro>
  <presentation>
    <name>complexPresentation</name>
    <class>net.sks.preproper.prepro.HTMLStreamTemplate</class>
    <resource>test/complexpage.html</resource>
  </presentation>
</prepro>
```

C. The Passage PrePro

PrePro between Pre and Pro acts as a passage to for the data. When the data is flowing from Pro to Per, the name best suited then might be ProPre. The PrePro interface has only submit **(String processingName, Object processingObject)** to pass data from Pre to Pro, in which **processingObject** is a request map to a processing class specified in the configuration entry of **prepro.cfg.xml**, which in turn passes the map to the service class by calling its service method. The passages has **only report(String processingName, Object processingObject)** for passing data from Pro to Pre, which gives the response map processingObject to a Presentation Template specified in the **processingName** configuration entry. Finally, the method **digestConfiguration()** inside PrePro uses **prepro.cfg.xml** to add presentation (see Table IV.3) and processing (see Table IV.1) entries.

Table IV.4 Interface PrePro

```
package net.sks.preproper.prepro;

public interface PrePro {
	public void submit(String processingName, Object
processingObject);

	public void report(String presentationName, Object
presentationObject);

	public void digestConfiguration();
}
```

We made the implementation class of PrePro a singleton so that one configuration file could serve the entire problem, and it really belongs to the PrePro as a container. In case of Web Applications, deployed from a Web Application Archive, they will expect to find ThePrePro already available.

Figure IV-2 shows some important properties and methods of ThePrePro class.

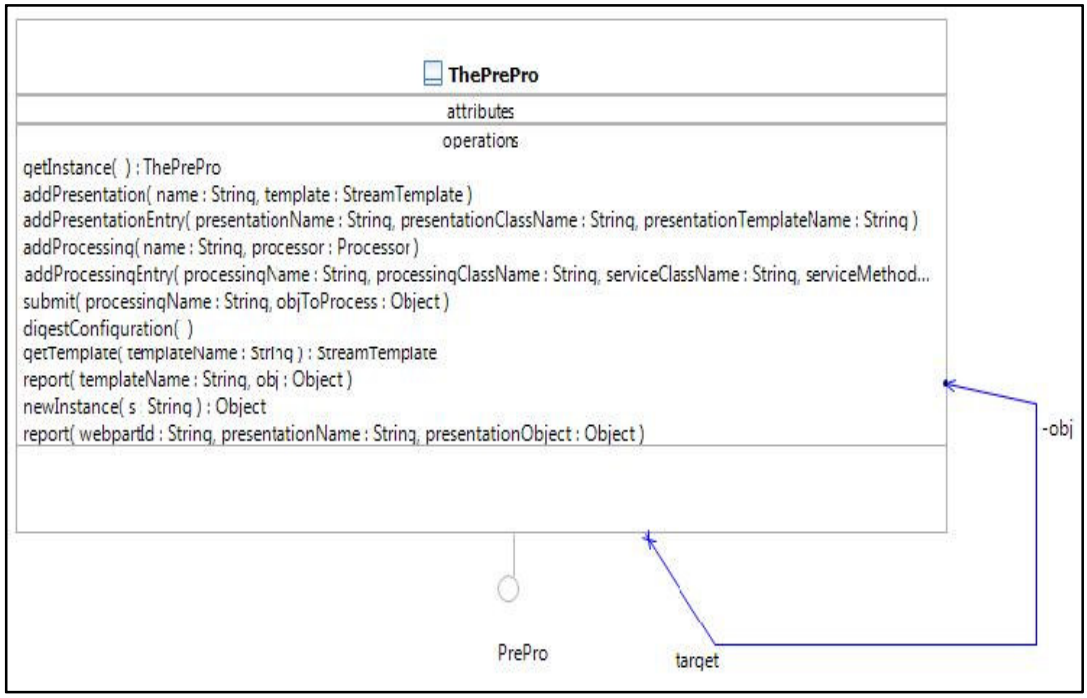

Figure IV-2 The class diagram of the passage PrePro.

D. The Passage ProPer

While processing Middie may require some data from Per and after processing needs to persist the data in Per.

The data stored to or retrieved from Per can be some directory of a file system or can be a database management system (DBMS). Middie should not have to worry about whether Per used a directory or a DBMS, and ProPer abstracts this away. Some of its important methods are **digestConfiguration**, **getSessionFactory** and **openSession**. A singleton implementation of the ProPer interface, TheProPer, is initially loaded by the container. It loads the configuration by using a ThePreProConfig object. It uses a very simple parser (it could use any XML parser) to parse proper.cfg.xml (see Table IV.2) and

to load an instance of the Configuration interface. The object loaded in our example can be of **FlatFilePersistenceConfiguration** or **HibernateConfiguration** type.

Middie gives name (as in Table IV.2 **<name>users</name>**) to use getSessionFactory(). Once he gets the factory he can open a session and start storing or retrieving from Per. Middie does not know how the architect configured it. Thus he is oblivious to what Techie does.

V. ISOLATION IN THE 'SMALL'

Before going into 'isolation' let us see what we mean by 'small'. It is how we define 'small' that makes it small. In the most general setting it is about a browser making HTTP request GET or POST and a Servlet in a web server container making response. The client POSTs an HTML form and the Servlet generates an HTML output that gets rendered in some standard browser. This is what is referred to as 'one-at-a-time page generation' [Johnson(2009)]. Thus communication steps are 'small' and we do not care how big the processing may be or how much output it may generate for Pre to show.

At the end of each section, we write a short summary of the described section. These summaries will comprise the 'rules of isolation'.

A. Understanding End to End Flow

PreProPer achieves isolation by defining a strict flow. This flow is shown in figure IV-1. The flow shows how the components communicate [D.Harris (2005)]. A component can read from or write to another component.

Without further ado, below is the strict flow that PreProPer defines:

receive-submit-invoke-[store/retrieve]*-report-respond-send

What this means is that **submit** takes place after **receive**, **invoke** after **submit** and so on. By **[store/retrieve]*** we mean, **store** or **retrieve** may take place multiple times and any of them taking place is optional. More specifically, for example, 'submit' is a method defined in the interface of the passage PrePro and Pre having an instance of the passage can call the method. The best way and according to the way of [D.Harris (2005)], Pre can write something to PrePro by calling **submit** method. Similarly, 'invoke' is called by

PrePro to get Pro started. According to our implementation, although there may not be any ‘invoke’ method in Pro, the idea is that PrePro calls a method defined in Pro. There are similar function that Pro defines and writes in the configuration which is available to PrePro and thus, PrePro can write to Pro by ‘invoke’.

Processing gets started by an ‘invoke’. It can do multiple transaction with Persistence by ‘store’ and ‘retreive’. Processing makes the processed data available to the passage by a ‘report’. By the time Processing makes a report there remains no data which is *dependent*. We explain this dependency later on when we talk about isolation in the ‘large’. Indeed, there can be multiple ‘report’s. However, multiple ‘reports’ cannot lead a programmer to report dependent data as there is no feedback mechanism available for Pro to process the dependent data later on. By the time another ‘invoke’ comes Pro will have to start over again as the same piece of code will be executed next time. Because of such flow PreProPer achieve what [Parr(2004)] pointed: View cannot compute on dependent data values. What this mean is that Presentation gets only independent data values. For performance benefit a good programmer may report data as soon as it becomes independent. Having said this, for Servlets and CGIs there will always be one report because HTTP response dies with the first request by the browser.

Once Pro makes a ‘report’, the passage uses a template and the template engine maps the data to the template and pushes on to Pre by a ‘respond’. Figure V-1 shows the ‘report’ and ‘respond’ by two ‘writes’. Pro writes to the interface of PrePro by ‘report’ and PrePro writes to Pre by ‘respond’. This is a ‘push’ [Harris(2005)]. What happens in a ‘push’ is shown in Figure V-2.

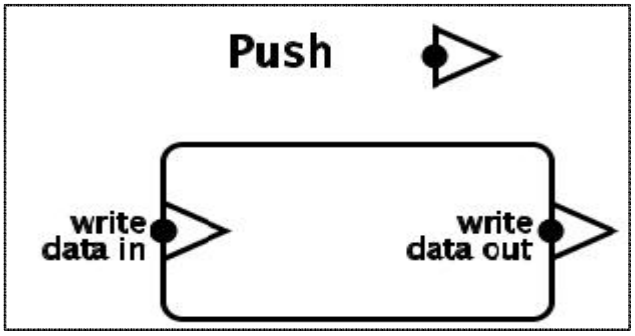

Figure V-1 A Push.[Harris(2005)]

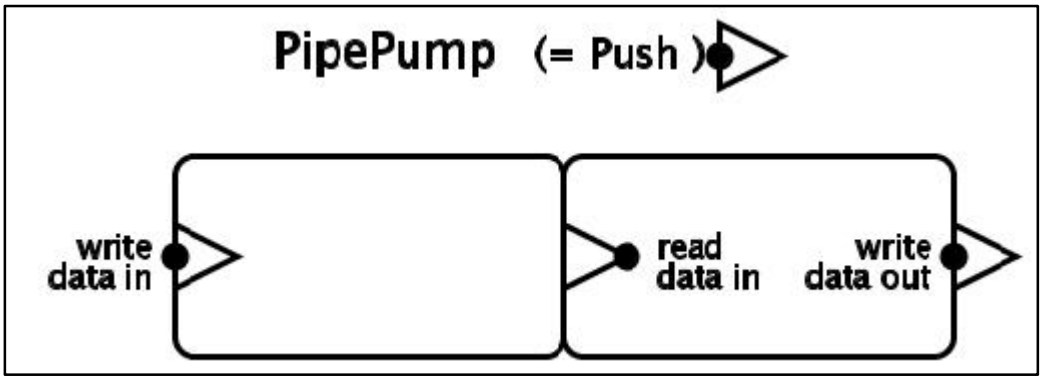

Figure V-2 A Push = PipePump.[Harris(2005)]

One way to implement a 'push' is by the consecutive use of a 'pipe' and a 'pump' [Harris(2005)]. A 'pipe' can be thought of as a buffer and once something is written to it, it stays there until a read from another component occurs. In our case, Pro writes map to the passage and the passage also contains the parsed templates. A pump is active, it reads from another components and writes to another. Here, the template engine is the 'pump' which reads the map and the parsed template and writes to Pre.

Summary:

- **The flow prevents Pro from giving dependent data.**
- **Pre cannot pull rather Pro pushes.**

- **An explicit flow can achieve the push.**

B. Writing Template and Template Engine

Webbies write templates that follow a grammar and an output device can parse and render. To show in a browser, the template is written in HTML. The templates are written in such a way so that it can be tested in isolation by Webbies.

According to the need of the organization Webbies make a bunch of templates. In order to parse these templates and incorporate data from Pro, the Architect writes template engines that run in the passage. Template engines may vary depending on the output device that presents the data and also on the type of data Pro provides.

Template engines need to have two common functionalities:

1. **Parse:** Any implementation is required to parse a language which follows a context-free regular grammar. For example, if XHTML is the language, the template engine need to be able to recognize it. This follows that the parser knows the terminal and non terminal vocabularies and the production rules. We developed HTMLStreamTemplate which parses in a top-down fashion. The terminals in this case are HTML tag literals.
2. **Map:** The template engine considers the data from processing as non-terminals for the template grammar and the method maps them according to the rules. As [Parr (2004)] pointed out it is often suitable to generate template languages by having the grammar sensitive to context.

   For HTML, we need to map both attribute and content. For **mapContent** we used HTMLMapper that generates HTML from the reported ‘map’ by Pro.

While generating, it takes into account contexts such as HTML tag type and the data type. For example, for <table> the map is interpreted so that it can contain <td>, <tr>. Again, depending on the type of data, for example, one/two dimensional array or Collection etc, the output HTML varies. This is context sensitivity.

We have seen so far how and what PrePro template engine may output. Pre can now do processing over the language. There are two possible ways for Webbie to do this. One way is to embed JavaScript in the templates so that it can reactively modify HTML DOM once the presentation is run in the browser. Another possibility is that the output HTML by PrePro 'respond' can be transformed using a style sheet such as XSL or by JavaScripts. Thus, Pre can do processing with whatever PrePro outputs. This way Pre will not need to assume any data type which is specific to Pro rather Pre can work with plain old HTML.

Summary:

- **PrePro provides a language which follows a Context-free grammar.**
- **A language in Pre can process only what PrePro provides.**
- **Pre makes no 'type' assumption of Pro data.**

C. Making Concerns 'Oblivious'

We find the importance of making concerns 'oblivious' to one another in [Filman and Friedman(2000)]. We formally look at the requirements to make Pre and Pro oblivious to one another. It is easy to understand what we want to achieve by making the concerns oblivious if we look at the antonyms for the word. Thus we really do not want

Pre be aware of how Pro does things, and we do not want Pre to be sensitive to what Pro provides.

Table V.1 shows a sample template. There are three ids **title**, **heading** and **body**. It is important for Pre to be totally oblivious to what Pro might fill in. The ids in the template cannot in anyway suggest Pro to fill something. Thus ids cannot suggest an attribute reference to Pro. Another important thing is by looking at mapped data Webbie should not be able to decide where to put the data rather Middie needs to decide where to put.

Table V.1 A sample presentation template.

```
<html>
<head>
<title id="title">TITLE</title>
</head>
<body>
<h1 id="heading">HEAD</h1>
<div>
<p id="body">BODY</p>
</div>
```

Let us look at the solution space more formally and following that we see how its variants could have entangled Presentation and Processing.

Let g be a function such that given a data map D, a template name N, and the template id I where data should go, the function generates the presentation P which is verifiable by a context-free grammar. Both the domain and co-domain are a finite sequence of symbols or strings over some alphabet $\sum$.

$$\boldsymbol{g}: \boldsymbol{D} \times \boldsymbol{N} \times \boldsymbol{I} \rightarrow \boldsymbol{P}$$

What this means is that Pro decides what the data will be, what sort of template the business might need, and in which position of the template the data needs to be

presented. Given these parameters the passage generates the presentation. Given *P,* Pre can later do whatever transformation or modification it may want to do.

If we change the execution place of *g* from the passage PrePro to other places and if we change the domain or co domain parameters, we will come up with some combinations which will make our problem spaces.

First, if we consider running *g* in Pre that would mean Webbie having template N, list of template ids *I,* it gets data from Pro and generates the Presentation. What this means is Webbie evaluates the data and decides the position *I* in the template *N* and finally generates the view *P*. Instead of Webbie being oblivious to what Middie does, it makes him conscious of the business data.

Secondly, instead of PrePro running *g* let us assume it runs $\acute{g}$ function such that given a data it transforms that to an Output *O* such that it is verifiable by a context-free grammar.

$$\acute{\boldsymbol{g}}: \boldsymbol{D} \rightarrow \boldsymbol{O}$$

Now Pre runs a function $\acute{p}$ such that given the output O by $\acute{g}$, a template *N*, and list of template ids *I* it generates presentation *P*.

$$\acute{p}: \boldsymbol{O} \times \boldsymbol{N} \times \boldsymbol{I} \rightarrow \boldsymbol{P}$$

What this means is that Webbie has the capability to decide where to put the output in the presentation. This is Webbie doing processing by using some language constructs like *if-else*.

Third possibility is that Pre uses *O* given by $\acute{\boldsymbol{g}}$ and creates a presentation *P* using $\ddot{\boldsymbol{p}}$. Both the domain and range are a finite sequence of symbols or strings over some alphabet $\sum$.

$$\ddot{p}: O \rightarrow P$$

Let us try to summarize and then discuss the possibilities to create a presentation from Pro data. The possibilities we have looked so far depends on who makes the choice and who executes the function.

Table V.2 The actors for choosing paramerters and the actors for executing functions.

Options	Choice	Execution	Function
1	Pro	PrePro	$g: D \times N \times I \rightarrow P$
2	Pre	Pre	$g: D \times N \times I \rightarrow P$
3	Pro	PrePro	$\acute{g}: D \rightarrow O$
3.1	Pre	Pre	$\acute{p}: O \times N \times I \rightarrow P$
3.2	Pre	Pre	$\ddot{p}: O \rightarrow P$

According to the Table V.2, implementation of PreProPer uses option 1. Middie chooses the template which will be good for the organization and it also decides where data should go by choosing the ids.

Option 2 means Webbie is choosing what data to use and how to present. This is the situation where Webbie is tangling Pro code. In option 3, PrePro generates an output from Pro data and depending on that output Pre can choose to use a predefined template and ids to generate presentation (Option 3.1) or Pre can make a presentation depending on that output (Option 3.2). The latter case suggest that Webbie will make presentation looking at the output. Option 3.1 suggests that Webbie uses templates and ids depending

on the output. It is not a problem for Webbie to use logical expression to manipulate however when Webbie need to think where to place output in the template, it is at this point they are doing the job of Middie. Thus, other than option 1, other options make Pre aware of what Pro does.

Summary:

- **Pre need to be oblivious to Pro.**
- **By following Option 1 (of Table V.2) Pre can be oblivious to Pro.**

D. 'The Relationship of the Concerns'

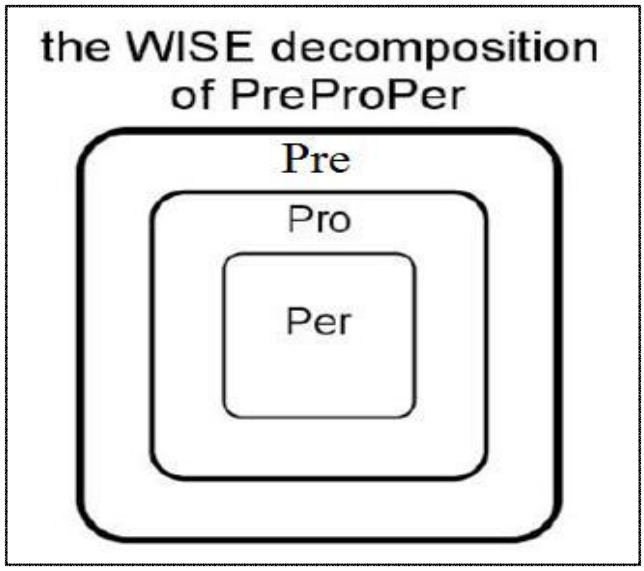

Figure V-3 The WISE Decomposition of PreProPer[Harris(2005)]

D.Harris in [Harris(2005)] looked into the issue of 'the relationship of the concerns.' We will be using his findings to deal with the problems of this thesis. He showed that Per is the legal representation of the organization or 'the state of the Enterprise.' Pro is actually the organization and it runs the business. Per is a subsystem of Pro and Per need to be safe from the outside which is Pre. In need Pro saves the

enterprise and sometimes the operational state in Per. Pre is the front end of the organization that deals with the customers.

The organization needs to know from Pre what goes in to Per and what goes out of Per to Pre and the way of knowing is the use of Pro.

Let us look at an example which may suggest that Pre needs to accesss Per directly. The example is about a service provider who gives users storage to save their personal data. From user's perspective s/he does not want Pro to know what s/he stores in Per. So s/he encrypts the data and wants direct access. May be Pro has an encryption algorithm that the user uses keeping the private key to itself. Even if he does not use the algorithm of Pro, the organization knows which storage it wants to keep and therefore Pro needs to interfere. May there is a policy agreement between the user and the provider that needs to be enabled by Pro. May there policy states that Pro can share part of the data with other providers keeping the anonymity of the user. In every sense, Pro needs to intrude in both direction, getting input and storing or retrieving and giving output.

'The best practice' for an organization is to let Pro access Per even if the Pro has a very thin tier.

Summary:

- **Pre cannot use Per explicitly but Pro can.**
- **Per is a subroutine of Pre.**

E. Coroutine and Subroutine

A coroutine can have multiple subroutines where each coroutine is made up of an input subroutine and an output subroutine [Conway(1963)]. Two modules that

communicate via coroutine always have a shared space which can be a network pipe or a storage system.

If we have coroutine then there is no ‘master program’. We find the need to make Pre and Pro work as coroutines. Pre may expect what Pro will do but we do not want Pre to dictate what Pro will provide and when it will provide. In Figure II-2 Pre is using Pro as a subroutine. Pre dictates by choosing the component to use and also makes data type assumptions as Pre will most likely to have a placeholder to store what Pro returns and then interpret accordingly. While Pre is about look and feel it has nothing to do with the what the data means.

For servlets, this sort of subroutine call to Pro is not possible. Thus only Figure II-3 is possible. This figure can be thought of in both ways either as a subroutine call or a coroutine call.

However, if we have a passage that Pre and Pro will have to explictly cross and that transforms the way data is interpreted by the concerns, we really have a coroutine which is synchronous.

Summary:

- **Pre and Pro need to act as coroutine.**

F. The Rules of Isolation

After each sections, from 5.1 to 5.5 we write one to two line summary. From the summaries we get following three rules of isolation.

1. Processing is finished before Presentation can begin to show output: If we consider end to end flow (section B), we find that the flow prevents Pro from giving dependent data. Pre cannot pull data and Pro pushes only independent data.

2. Presentation and Processing are coroutines: Pre cannot make Pro a subroutine (section E). Subroutine call is deterministic and if Pro is a subroutine of Pre, Pre determines what needs to be done by Pro.

However, we point out (section C) that Pro rules. Pro will decide what Pre will have but Pre cannot expect what Pro will provide. This does not mean that Pro can use Pre any number of times. Actually it cannot (it can with multiple reports that we discussed in section A). Our flow leaves no feedback mechanism for Pro to come back after making output to Pre.

3. Persistence is a subsystem of Processing: We conclud (from Section E) that Pre cannot access Per. Only Pro can access Per. This is the best practice and it works in both direction: input and output. Pro and Per relationship suggest that Per is a subsystem of Pro. Pro will be using Per as subroutines.

VI. THE SYSTEM FOR WEB 1.0

We look at the structure of PreProPer system when Servlet and Hibernate [Hibernate(2009)] technology is involved. First we develop the passage PrePro suitable for handling HTTP request and response. The passage is deployed in a Servlet container (for example, Tomcat). For the ProPer passage we use Hibernate and show how store and retrieve take place. We provide an illustrative example in section C.

We closely look at the Templating capability in section D and finally, we make comparative study in section E.

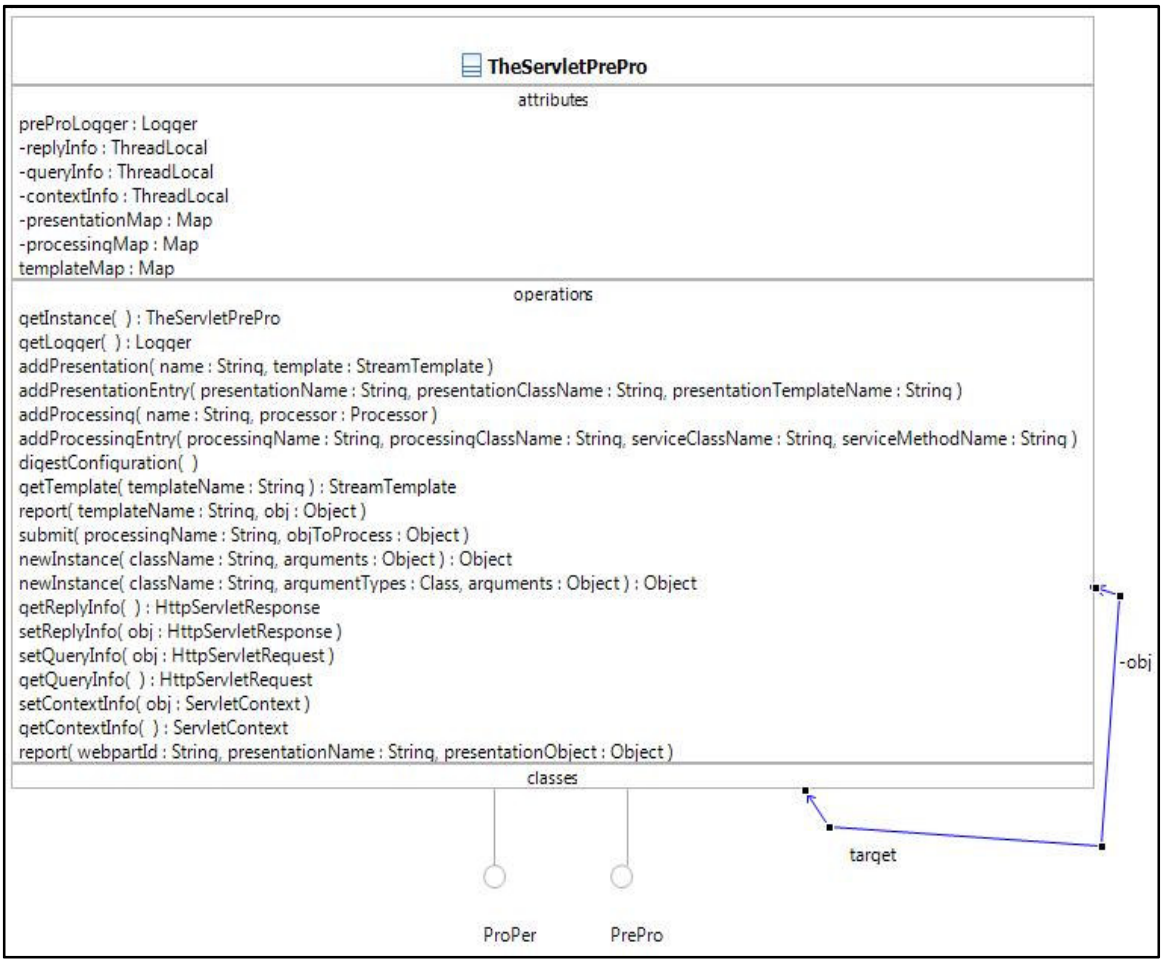

Figure VI-1 The class diagram for TheServletPrePro.

A. The Servlet PrePro

We name the passage for handling a Servlet **TheServletPrePro**. The code and the example illustrated are available in [Harris and Monjur(2009)]. We look at the implementation details and also what Webbie, Middie and Techie have to do in Pre, Pro and Per. We also describe how the flow and its directions are achieved within the passages and the tiers, and show the role of the Manager and Architect.

1. PrePro

PrePro between Pre and Pro acts as a passage to for the data. When the data is flowing from Pro to Per, the name best suited then might be ProPre. The PrePro interface has only submit(String processingName, Object processingObject) to pass data from Pre to Pro, in which processingObject is a request map to a processing class specified in the configuration entry of prepro.cfg.xml, which in turn passes the map to the service class by calling its service method. The passages has only report(String processingName, Object processingObject) for passing data from Pro to Pre, which gives the response map processingObject to a Presentation Template specified in the processingName configuration entry. Finally, the method digestConfiguration() inside PrePro uses prepro.cfg.xml to add presentation (see Table IV.3) and processing (see Table IV.1) entries.

We made the implementation class of PrePro a singleton so that one configuration file could serve all requests, and it really belongs to the PrePro as a container. In case of

Web Applications, deployed from a Web Application Archive, they will expect to find ThePrePro already available.

2. From Presentation to Processing: submit and invoke

From the implementation standpoint, when Pre calls **ThePrePro.submit(name, map)**, it is basically handing off the request map to the named processing entry (Table IV.1). This entry has a processing class apart from the actual ServiceClass/ ServiceMethod of Pro. In short, this processing class uses **Java Proxy** to call the appropriate method of the service class. The service class and the classes that depends on it comprise Processing tier.

Every processing class implements **PreProProcessor** which extends the **Processor** interface and has a method **setOutputProcessor(Processor p)**. **Processor** has only one method **process(Object obj)**. All **PreProProcessor** objects invoke the **process** method of a **Processor**.

When the Manager adds a processing entry from configuration (see Table VI.1) each **PreProProcessor** object is given the Proxy instance of a **Processor**.

Now you know and we know that while we call **Processing.process**, we actually want to call ServiceClass/ServiceMethod, perhaps with a modified argument. This is where **MappedInvocationHandler** comes in (see Table VI.2).

Table VI.1 Code for adding Processing Entry.

```
public void addProcessingEntry(String processingName,
            String processingClassName, String serviceClassName,
            String serviceMethodName) {
        /*
         *
         */
```

```
            Method mRequested;
            try {
                  mRequested =
                              Processor.class.getDeclaredMethod("process",
                              new Class[] { Object.class });
            } catch (NoSuchMethodException ex) {
                  throw new PreProcException(ex);
            }
            /*
             *
             */
            Method mActual;
            Object serviceObject = newInstance(serviceClassName);
            try {
                  mActual = serviceObject.getClass().getDeclaredMethod(
            (String) (serviceMethodName), new Class[]{Object.class });
            } catch (NoSuchMethodException ex) {
                  throw new PreProcException(ex);
            }
            /*
             *
             */
            Map methodMap = new HashMap();
            methodMap.put(mRequested, mActual);
            /*
             *
             */
            MappedInvocationHandler handler = new
                        MappedInvocationHandler(serviceObject, methodMap);
            /*
             *
             */
            Processor preproProcessor = null;
            Class[] whichInterfaces = new Class[] { Processor.class };
            ClassLoader serviceObjectLoader = serviceObject.getClass()
                        .getClassLoader();
            preproProcessor = (Processor) Proxy.newProxyInstance(
                        serviceObjectLoader, whichInterfaces, handler);
            /*
             *
             */
            PreProProcessor processingObject = (PreProProcessor)
                         newInstance(processingClassName);
            processingObject.setOutputProcessor(preproProcessor);
            addProcessing(processingName, processingObject);
}
```

Table VI.2 MappedInvocationHandler class.

```
public class MappedInvocationHandler implements InvocationHandler {
      private Object actualObject;

      private Map methodMap;
```

```
	public MappedInvocationHandler(Object actualObject, Map
methodMap) {
		this.actualObject = actualObject;
		this.methodMap = methodMap;
	}

	public Object invoke(Object proxy, Method mRequested, Object[]
argsSupplied)
			throws Throwable {
		Method mActual = mRequested;
		try {
			if (methodMap.containsKey(mRequested)) {
				mActual = (Method) methodMap.get((mRequested));
			}
			Object result =
				mActual.invoke(actualObject, argsSupplied);
			return result;
		} catch (InvocationTargetException ex) {
			throw ex.getTargetException();
		} catch (Exception ex) {
			throw ex;
		}

	}
}
```

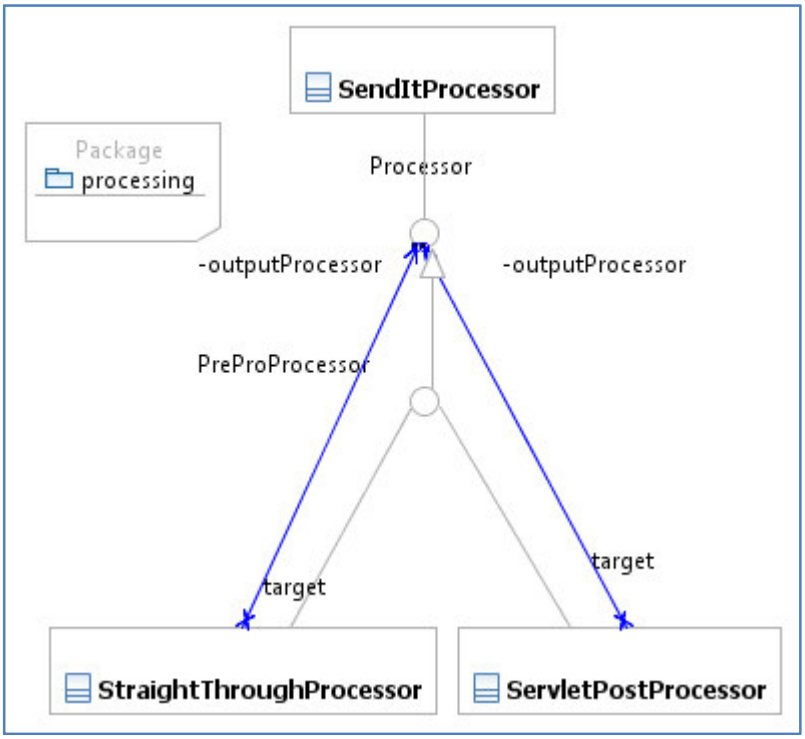

Figure VI-2 Class diagram of Processor and PreProProcessor classes.

In figure VI-2 we see some classes of PreProProcessor: **StraightThrough-Processor** and **SevletPostProcessor**. **SendItProcessor** is a an exception as it is not a **PreProProcessor** but it implements **Processor**. This means that it is not invoking any ServiceClass/ServiceMethod. Such Processor is used when PrePro responds directly to a request without going to Pro. This is sometimes needed, for example, to send an exception.

The Architect and Manager will make decisions as to what sort of processing class to use for a problem and the functionality they should have.

In order to hold the flow strict, Pro does not know where to reply and thus in the **RequestMap** we did not provide **replyInfo**. Providing the reply path may tempt Middie to perform the Pre task and report directly to the customer. We set reply and context information in ThreadLocals so that each thread can have an independent copy of these variables.

Here are some of the basic tasks that each **PreProProcessor** has to do: Provide only the request parameter as 'what' object to Pro, set the **replyInfo** and **contextInfo** in thread local variable**,** clear ThreadLocal variables when they are no longer needed. It is important to clear because If the Servlet container uses a thread pool, the threads can cause memory leak or even compromise the integrity at a later time.

PrePro can also point out if there is any integration error with Pre and Pro. In the following list, we see **StraightThroughProcessor** report back to Pre in case of exception to process Pro.

Table VI.3 StraightThroughProcessor class

```
import java.util.Map;
import net.sks.preproper.prepro.ThePrePro;
```

```
public class StraightThroughProcessor implements PreProProcessor {
	/*
	 * implements Processor so configuration will have a way to refer
to these
	 */

	private Processor outputProcessor;

	public void setOutputProcessor(Processor outputProcessor) {
		this.outputProcessor = outputProcessor;
	}

	public void process(Object obj) {
		try {
			outputProcessor.process(obj);
		} catch(Exception ex) {
			ThePrePro.getInstance().report("proerro", ex);
		}
	}
}
```

3. From Processing to Presentation: report and respond

This is the time when Middie has finished running the business rules, stored and retrieved the 'State of the Enterprise' and now wants to **report** by using a string (representing the name of a template) and a map (see Table VI.4).

The string refers to a presentation entry in **prepro.cfg.xml** file and in that entry apart from the name we have a class name (which is a **StreamTemplate** class that knows how to parse templates and feed attributes and contents from a map), a resource name (which refers to the location of the template or many be parameter in the container that has the actual location) and an optional style name (which refers to an XSL file).

Table VI.4 Pro is putting values in the Map and calling 'report'.

```
		ThePrePro prepro = TheServletPrePro.getInstance();
		prepro.report("userPresentation", map);
		//Not shown how the map comes to existence
```

Templates can be written in markup languages like XML, XHTML and HTML. Currently, we use plain HTML files as one variety of template. We use an **id** attribute of an element whose content or attribute we may wish to replace or add. The only constraint is that an IdAttribute with a particular value must be unique within the document. According to W3C in XHTML an **id** attribute assigns a name to an element and this name must be unique in the document.

However, because of this uniqueness **id** is also widely used as a style sheet selector. For our current version of templating it will refer to an element whose attribute values can be replaced and into which element content can be filled. Like XMLC we can use **<span ID=``idName''>**. We can also use **<div ID=``idName''>** because according to the HTML standard these elements go inside any tag. Another interesting option is to put `id's inside comments, i.e. **<!- id=``idName$''-->**.

Table VI.5 A sample template.

```
<html>
<head>
<title id=" pageTitle ">TITLE </ title>
</head>
<body>
<table>
      <tr>
            <td id=" pageHeader " align=" center">Page Header</td>
      </tr>
      </ table >
      <table id=" listoflists ">
            <tr>
                  <td>Only one cell</td>
            </tr>
      </ table >
</body>
</html>
```

The PrePro parses a template using a **StreamTemplate** which is an interface for the template engine. It declares two methods: **parse** and **mapStreams**. We have one implementation of it, **HTMLStreamTemplate** that is what we call `dumb'. It does nothing but parses HTML template in a way so that it can accomodate any content or

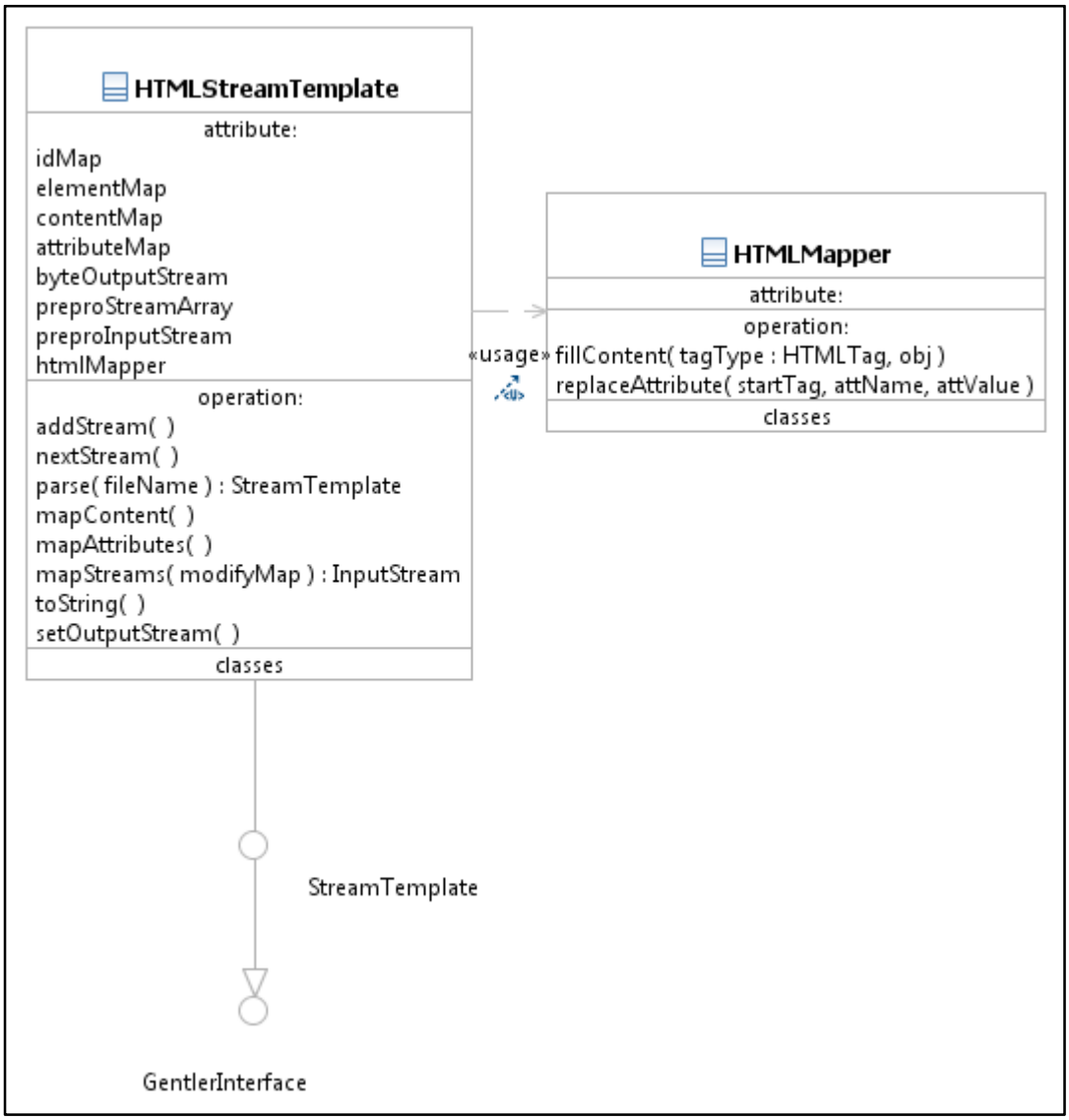

Figure VI-3 HTMLStreamTemplate, a StreamTemplate implementation.

attribute pushed by Middie and can place the data in Template positions as outlined by Webbie. It really is a compiler that can parse a context-free language and map data according to a context-sensitive language.

The **parse** method acts like a finite state machine (FSM). It takes a character at a time. Depending on the current state and the character encountered, it moves to a new state or remain in the same state. For example, if it is in **S_FILL_START** state (which means that it has already encountered a start-tag (<) and has started filling in the value) and encounters stop-tag (>), then it has finished reading a **tagName** and moves on to the state **S_FILL_CONTENT**.

When we are in the state **S_FILL_VALUE** where we are filling in an attribute value, if we encounter a single or a double quote, then we know that we got the attribute value and we need to move on to the **S_FIND_NAME** state (where the parser is looking for another attribute name). Here we are insisting that attribute values be surrounded by quotes. Now, if it is the value of an id attribute, we use the **attValue** in two HashMaps: **idMap** and **elementMap**.

Table VI.6 shows the code block to parse a template and Table VI.7 shows the output from the parser.

Table VI.6 HTMLSteamTemplate is parsing a template.

```
HTMLStreamTemplate t = new HTMLStreamTemplate();
t.parse("testTemplate.html");
```

Table VI.7 A parsed template output.

```
<html>
<head>
<t i t l e id=”pa g eTi t l e ”> 1= TITLE 2= </ t i t l e>
</head>
```

```
<body>
<table>
<tr>
<td id=”pageHeader ” al ign=”c ent e r ”>
3= Page Header 4= </td>
</ tr>
</table>
<table id=” l i s t o f l i s t s ”>
5= <tr> <td> Only one c e l l </td> </ tr> 6=
</table> </body> </html> Entry Set [
l i s t o f l i s t s =4, pageHeader=2,
pa g eTi t l e=0]
```

Let us look at the structure of a **ResponseMap**. It extends a **BaseMap** which is itself a HashMap and contains two public maps (also HashMaps): **attributeMap** and **contentMap**. Each **ResponseMap** fills these two maps of **BaseMap**. These maps are basically (name, value)/(key, value) pairs. The names are idAttributes of the template and they come into existence by joint consultation between Webbie and Middie in the presence of the Manager. Webbie does not care what the value will be but does care for its presentation and knows how to interpret the data. Middie does not care how the value will be presented but knows how to populate the map with appropriate data made available after the business rules/actions for the problem are already executed.

The data for the map can be attributes or contents. Depending on the name, which is the idAttribute in the template, these attributes or contents will be placed in the corresponding HTML element of the template.

We look at the **mapStreams** method of **HTMLStreamTemplate**. which places the **ResponseMap** in the template. It uses two subroutines: **mapContent()** and **mapAttribute()**. Depending on the tag/element type and on the data type, content will be placed accordingly with the help of **HTMLMapper** (see Figure VI.3). For example, for a table tag with map data of type **String**, **Collection**, **Object[][]**, and **Object[]**

HTMLMapper knows how to represent those values appropriately in HTML. Thus, it is not necessary for template tags to be of HTML types but we need a mapper which knows how to interpret data to be filled in the new tag into HTML.

Table VI.8 UserRoleMap, a sample Map class.

```
import java.util.Map;
import net.sks.preproper.prepro.BaseMap;

public class UserRoleMap extends BaseMap {
  public UserRoleMap(String[] attributes, String[] contents){
        this.put("--ContentMap--", contentMap(contents));
        this.put("--AttributeMap--", attributeMap(attributes));
  }

  public Map contentMap(String[] contents){
        contentMap.put("username", contents[0]);
        return contentMap;
  }

  public Map attributeMap(String[] attributes){
        attributeMap.put("username", attributes[0]);
        return attributeMap;
  }
}
```

Let us look at **prepro.report("presentationName", map)**. Table VI.9 shows the basic functionality. However, in an actual **report** implementation there are several possibilities and we can handle each one individually:

- **report(``'', obj)}**
- **report(``name'', null)}**
- **report(``name'', obj)}**
- **report(``'', null)}**

Table VI.9 A Map is mapped into a template.

```
StreamTemplate t = getTemplate(templateName);
 if (null!= t) {
        InputStream i = t.mapStreams((Map) obj);
 }
```

B. Hibernate and ProPer

The obliviousness of Middie about the subsystem Per comes because of the passage ProPer, configured by the Architect who designs ProPer for whatever Per systems are desired. While the Architect looks at all technical design issues of the system, the Manager decides what systems are appropriate for the organization and he comes to relate to Middie what the organization/business wants them to perform and for that what Techie must know to persist the 'State of the Enterprise'. The Manager may sit in a meeting between Middie and Techie to see them relate names in Pro to data in Per.

These issues need to be resolved during system design so that Middie knows for example that by working with PoJos (Plain old Java objects) according to the Business Rules all will be fine and Techie needs only to to properly persist the data associated with those names. Listing 12 shows a typical example of Middie using the name of a configuration entry, "users", to perform a transaction.

Table VI.10 A sample example of a store and a retreive (find).

```
SessionFactory f = null;
Session s = null;
TheProPer theProPer = TheProPer.getInstance();
theProPer.initConfiguration();
f = theProPer.getSessionFactory("users");
if (f == null) {
      throw new ProPerException("null SessionFactory");
}
s = f.openSession();
Member m = new Member("Mehrab Monjur");
s.store(m);
List<Member> ml = s.find("from Member as m where m.name=
\"Mehrab Monjur\"");
System.out.print(ml.get(0).getName());
s.close();
f.close();
```

In Table IV.2, we looked at the **proper.cfg.xml** file which had two persistence <name> entries “sports” and “users”. The former name causes TheProPer, a singleton, to load **FlatFilePersistenceConfiguration** and the later one uses a **Hibernate Configuration**. For the example shown in Table VI.10 **thePrePro.getSessionFactory** gives an instance of **HibernateSessionFactory** and with f.openSession we get **HibernateSession.** Important to note that Middie does not have to know that. It only knows that using ‘users’ name of the configuration will help him/her achieve the objective. Each **Session** has store and retreive functionality that Middie cares about.

C. Illustrative Example

We will try to capture the entire flow of PreProPer through a simple scenario: From the browser the user makes a GET request for a page that has a input box to get username, then see that the form is filled POSTed to the server.

The server sees if the username is currently existent and if so might determine the role the user can assume. The server sends back a confirmation page. The confirmation page could have a different color depending on the user’s role.

For our illustration we decided to have two templates for HTTP GET and POST. First the servlet loads PrePro and ProPer configuration through init(). Table VI.11 shows the configuration for PrePro. It is important to note here that physical locations for templates and stylesheets and similar items are in the web.xml as parameters.

UserRoleServlet receives a GET request from the browser and executes thePrePro.submit(“get”,params). Here there is no parameter. Although there could have been a configuration with entry (URI name, processing entry), currently the servlet needs

to know which processing entry to use; in this case, it is ‘get’. PrePro hands this request to ServletGetProcessor which mainly stores the HttpServletResponse and ServletContext in threadlocal variables, replyInfo and contextInfo and hands it over to UserNameReporting/main. Here there are two options for Pro, either it can select a template from prepro.cfg.xml or it can retrieve a page from Per. In the first case, it calls **thePrePro.report(“userForm”, null)** where “userForm” is a presentation entry and there is no map to provide. In the second case, it calls **thePrePro.report(“”, in)** where in is the InputStream’ for the stored page.

Table VI.11 The configuration file for PrePro in the illustrative example.

```
<prepro>
  <presentation>
    <name>userForm</name>
    <class>net.sks.preproper.prepro.HTMLStreamTemplate</class>
    <resource>userform</resource>
    <style>xslPath</style>
  </presentation>
  <presentation>
    <name>userPresentation</name>
    <class>net.sks.preproper.prepro.HTMLStreamTemplate</class>
    <resource>usertemplate</resource>
    <style>xslPath</style>
  </presentation>
  <processing>
    <name>send</name>
    <class>net.sks.preproper.processing.ServletGetProcessor</class>

<serviceClass>net.sks.preproper.gwt.server.processing.UserNameProcessin
g</serviceClass>
    <serviceMethod>main</serviceMethod>
  </processing>
  <processing>
    <name>straight</name>
    <class>net.sks.preproper.processing.ServletPostProcessor</class>

<serviceClass>net.sks.preproper.gwt.server.processing.UserNameProcessin
g</serviceClass>
    <serviceMethod>main</serviceMethod>
</prepro>
```

The user enters a name and submits. The servlet receives it, stores the request parameters in a List and calls **thePrePro.submit("post", params)**. According to "post" (in Table VI.12) **ServletPostProcessor** handles the processing in PrePro and after process we enter UserNameProcessing/main. Here Middie retrieves the role data for the given user, and uses theProPer.getSessionFactory("users") to get the SessionFactory. In Table IV.2, for the users entry it is talking about a Hibernate configuration file. Thus ProPer will use a HibernateSession to load the Role for Pro.

Table VI.12 ServletPostProcessor class.

```
public class ServletPostProcessor implements PreProProcessor {
      private Processor outputProcessor;

      /**
       * This processor takes a servlet request as input, retrieves the
parameter
       * map, and writes to the specified processor.
       */

      public void process(Object obj) {
            Map params = new HashMap();
            if (!(obj instanceof List)) {
                  throw new PreProException("Expected a list, got "
                              + (obj == null ? "nothing" :
obj.getClass().getName()));
            }
            List inData = (List) obj;

            if (!(inData.get(0) instanceof HttpServletRequest)) {
                  throw new PreProException("Expected a servlet
request, got "
                              + (obj == null ? "nothing" :
obj.getClass().getName()));
            } else {
                  HttpServletRequest req = (HttpServletRequest)
inData.get(0);
                  params = req.getParameterMap();
                  TheServletPrePro.getInstance().setQueryInfo(req);
            }

            if (!(inData.get(1) instanceof HttpServletResponse)) {
                  throw new PreProException("Expected a servlet
response, got "
                              + (obj == null ? "nothing" :
```

```
obj.getClass().getName()));
            } else {
                  HttpServletResponse resp = (HttpServletResponse)
inData.get(1);
                  TheServletPrePro.getInstance().setReplyInfo(resp);

            outputProcessor.process(params);
      }

      public void setOutputProcessor(Processor outputProcessor) {
            this.outputProcessor = outputProcessor;
      }
}
```

Thus Pro gets the role and sets up arguments for UserRoleMap and reports to PrePro (Table VI.8 shows what goes on at this point). Middie wanted the report to use **userPresentation** which has a template, usertemplate and a stylesheet, xslPath (see Table VI.13 for the configuration). Thus PrePro maps the **UserRoleMap** to usertemplate using **HTMLStreamTemplate** and transforms the input stream according to the style sheet.

Table VI.13 Pro fills in the map and reports.

```
            Class[] argsClass = new Class[] { String[].class,
String[].class };

            String[] roleAttribute = new String[] { "title=emphasis" };

            String[] username = new String[] { "Mehrab Monjur" };
            Object[] arguments = new Object[] { roleAttribute, username
};
            TheServletPrePro prepro = TheServletPrePro.getInstance();
            String mapClassName =
"net.sks.preproper.processing.map.UserRoleMap";// args[3];
            Map map = (Map)
prepro.newInstance(mapClassName,argsClass,arguments);
            prepro.report("userPresentation", map);
```

Before all began in the presence of Manager, Middie informed Webbie that users can have roles, ‘very-important’, ‘important’ and ‘less-important’. Middie also wanted Webbie to emphasize roles in the presentation accordingly. Webbie understood the

importance in the meeting. Later on, Webbie selected for each role CSS class (in the example, the color) and made the style sheet accordingly.

Table VI.14 The template for the example.

```
<html >
<head >
<link
rel=" stylesheet "
type ="text /css"
href =" style.css"
>
</link >
</head >
<body >
<p
id=" username "
class=" value"
title=" default "
>
</p>
</body >
</html >
```

Table VI.15 The XSL file used for the example.

```
<xsl:stylesheet version ="1.0"
xmlns:xsl =" http:// www.w3.org /1999/ XSL/
Transform "> <xsl:output method="html " indent ="yes" /> <!-- the
identity template -- >
<xsl:template match="@*| node ()">
<xsl:copy >
<xsl:apply-templates select="@*| node ()"/> </ xsl:copy >
</ xsl:template > <xsl:template match="p">
<xsl:if test =" @title = 'very-important '">

<p id="{@id}" style="color:red"><xsl:apply-templates /></p>

</ xsl:if >

<xsl:if test =" @title = 'important '"> <p id="{@id}" style=" color:blue "><xsl:apply-
templates /></p> </ xsl:if >
```

```
<xsl:if test =" @title = 'less-important '"><p id="{@id}" style=" color:green
"><xsl:apply-templates /></p>
</ xsl:if >
</ xsl:template >
</xsl:stylesheet >
```

D. Templating Capability

For templating we use **HTMLStreamTemplate**. For an id attribute, it is not necessary to fill HTML element from attributeMap/contentMap. If there is no value given in the map but the Webbie gave some default attribute and/or content, the element will remain that way. On the othe hand, if a map entry consists of a name that does not correspond to an id, then that entry will not be included in the final presentation. Middie can fill in what and where if Webbie so desires. An attribute that is not expected by Webbie will not be filled in by **HTMLStreamTemplate**. For example, below is a template element which is expecting attribute class and title (provides extra information about an element).

Table VI.16 Elements with different attributes

```
<p id=" idName " class =" className " title ="titleName "></p>
```

Now apart from the content, Middie can fill value for class and title but not any additional attribute. However, as Middie is not concerned with presentation and s/he is not likely to fill in a value for class. Filling such a value requires understanding of the included CSS file, which is created by Webbie. The Architect can design an HTMLStreamTemplate such that it does not allow some attributes like, class, style, onclick to be placed by Middie. This is really strict and may apply when unruly or unduly

knowledgeable Middies are around! All we want to say is that Middie should not meddle in a task that Webbie should do. Because of such notions **HTMLStreamTemplate** does not allow Middie to add attributes. First look at the following example, where Webbie is expecting that rest of the presentation will be filled by Middie.

Table VI.17 A template with id in its body.

```
<html ><body id=" bodyId " /></ html >
```

Below is something that middie might want to insert in 'bodyId' as content:

Table VI.18 Presentation of aggregage attribute.

```
<img src="/ images / camel .gif" align ="middle " alt=" Camel "/>
```

There are two possibilities for Middie to map this content in the template. One obvious solution would be to stick the entire string for the id bodyId in the map. Another possibility is to provide an aggregate Object with an overloaded toString() method.

Table VI.19 Using overloaded toString() for Map.

```
public class ImageAndLabel {
      private String imgFileName, labelString, altImgMsg;
//All the getters and setters go here.
      public String toString() {
            StringBuffer b = new StringBuffer();
            b.append("<img src=\"");
            b.append(imgFileName);
            b.append("\" alt=\"");
            b.append(altImgMsg);
            b.append("\">");
            b.append("<br>");
            b.append(labelString);
            return b.toString();
      }
}
```

Here we see that Middie is giving value as well as presentation literals. This is because Webbie has given up and he is asking Middie to do the presentation himself. But the situation would have been different if Webbie took the task to present and if the template was like this:

Table VI.20 Attribute difference in template.

```
<img id=" imgId " src=" defaultSrc " align ="middle " alt=" defaultAlt "/>
```

In this case, Middie would need to provide attributeMap for only 'defaultSrc' and 'defaultAlt'.

Earlier in section II A we have seen the 5 separation rules that T.Parr mentions in [Parr(2004)]. These rules serve as the 'entanglement index'. Among the 5 rules there is one which needs mentioning:"data from the model must not contain display or layout information." He shows that including his own **StringTemplate** [Parr(2005)] other templates like "Tapestry, WebMacro, Velocity, PTG, UniT, Tea, WebObjects, FreeMarker, ColdFusion, Template Toolkit, and Mason"[Parr(2005)] all have entanglement index 4. The rule that is quoted is the tricky one and it seems that **HtmlStreamTemplate** is breaking it. It will be violated only if Webbie wants Middie to violate it. Another good question to ask at this point: is the rule valid all the time? Of course it is not. For example, if Middie is trying to make an article that shows HTML samples, he will need to push some HTML code in Pre.

E. Comparative Study of Templating Capabilities

We look at the evolution of template engines and compare with some of the well knowns. Let us start with a plain Servlet. There we see entanglement of HTML in the

java code and later, in JSPs we see Java in the HTML file (although in reality JSP is a servlet). While the former gives Middie the upper hand, the latter gives Webbie the same. Indeed both of them are unrestricted and both Middie and Webbie can forget who they really are.

Table VI.21 Unrestricted Templates using presentation literals and action expression.

```
<title >[% request . getParameter (" title")%]
</ title > // JSP

Or

<title ><tiles. getAsString name =" title"></title >
// Spring hiding action expressions
```

In figure VI.22, we give examples of unrestricted templates with unbounded actions wrapped around 'dumb' output HTML elements. For Spring although the action expression is just to get model data from the controller, because it is a framework and presentation is written in a JSP file, one can easily modify part of the model in that JSP page.

HTMLSteamTemplate is very similar to Enhydra **XMLC** [XMLC(2004)]. An id attribute or span tag is used to specify the content or attribute position in the template. XMLC converts the template to a java or a class file. Each id in the template is converted to getId or setId. Model can use these getters and setters to push business data. Both XMLC and our HTMLStreamTemplate has an entanglement index 1. However, they have differences, mainly based on their ease of usage. For example, for XMLC template engine does not know how to map data according to their tag literals rather model needs to define how it wants to map. In HTMLSteamTemplate we added some of these tag

sensitive mapping functionality in the template engine so that the job of Middie can be alleviated.

Table VI.22 An XMLC Template before and after parsing

```
//Before parsing

<title ID=" title"></title>

<SPAN ID=" id"></ SPAN>

//After parsing

---

   /**
    * Get the element with id title.
    * @see org.w3c.dom.html.HTMLTitleElement
    */
   public org.w3c.dom.html.HTMLTitleElement getElementTitle() {
      return $elementtitle;
   }
……….
```

Where do we stand? HTMLStreamTemplate is unrestricted. We allow Turing complete languages like XSLT and JavaScript to transform and to play with our template outputs. We can embed or overlay a Turing complete language over the unrestricted language provided by the template engine because we have a strict flow of execution. The flow pushes data in Pre and the pushed data is independent. Also the Turing complete language need not make any type assumption of Pro data.

Let us look once more how Pre and Pro (thus Webbie and Middie work) work step by step. The following review is just a short summary of what we have described in section V A, B, C and D.

First, ideally, Middie only pushes map filled with model data when it has persisted the 'state of the enterprise'. Secondly, once the data is pushed and template is

output there is no feedback mechanism available to come back to persist in Per. Only way to get again in Pro is by submit and process, and if Middie does not understand or accept such input, then the changed data dies there let alone going to be persisted.

It seems really safe so far, but Webbie can certainly show anything to the world: perhaps the Manager wants Webbie to present that way. Also, Per being responsible for the ‘legal representation of the organization’ can be held accountable and not Pre.

Finally, and most importantly, such unruly XSLT should not have been configured by the Manager into PrePro in the first place.

How about Middie colluding with Webbie? Let us not dig any further.

VII. ISOLATION IN THE 'LARGE'

We look at what dependency resolution and positioning means. To isolate such system in the 'large' we see the importance of Flow Based Programming for processing whether we do it in Pre or Pro or both. We look at the use of GWT here. We look at the design and implementation details of TheGWTPrePro so that it can have GWT/Presentation running JavaScript and making Ajax calls in one side and Pro doing Flow Based Programming in other side.

A. Resolving Dependency

Pre and Pro both are Turing complete. Both can do processing. Only difference is Pro does processing of data that it reads from Per or data that it gets from PrePro (something that may come unaltered from Pre), and Pre does processing of what it gets as output from PrePro (an output from template engine) and what it receives from the user input. It is clear that processing of Pre will take place in the browser but processing of Pro may take place in the browser or in the server. Both Pro and Pre will have multiple components which are independent and to solve a particular problem that requires multiple domain knowledge the components will have to coordinate their execution. We intend to analyze dependency of such components so that we can determine which parts are suitable to be executed in the Browser and which parts need to remain in the server. This also suggests that while we talk of component dependency, we need to consider that part of the components are distributed. We are expecting to figure out what sort of processing we can do in the browser.

Component dependency can be represented in terms of a directed graph. We call such a graph Directed Dependency Graph.

A **Directed Dependency Graph (DDG)** *D* is a ordered pair *D=(V,A)* where *V* is a set of vertices such that each element represents a component and *A* is a set of arcs. An arch *a=(x,y)* where *a*ϵ*A* is directed from *x* to *y*. This means component *y* is dependent for input from *x*. As components are thought of running as coroutines it is not necessary that a component that receives an input from another component, has to come back and return something to the calling component. However, they can return something at times. These are the two situations that are shown in the Figure VII-1. We do not show any ordering of calling, because for the purpose of what will follow next it is not necessary.

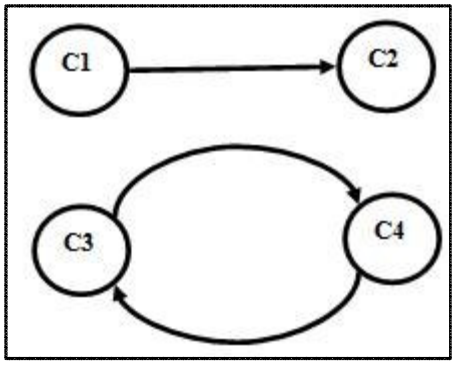

Figure VII-1 Directed Dependency Graph

We classify components into two:

1: Storage Processing (S): These components write to Per by **store** and read from Pro by **retrieve.** We call such a component S.

2: Pure Processing (P): These components do not write or read from Per. We call such a component P.

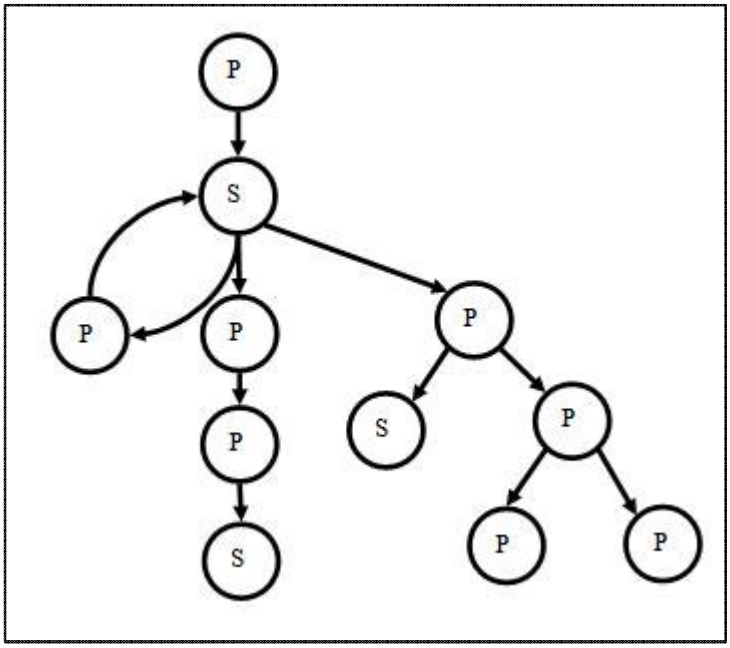

Figure VII-2 Directed Dependency Graph of S and P components

Figure VII-2 shows a complex DDG consisting of S and P components. Some S are dependent on P and some P are dependent on S. In one case both are dependent on one another.

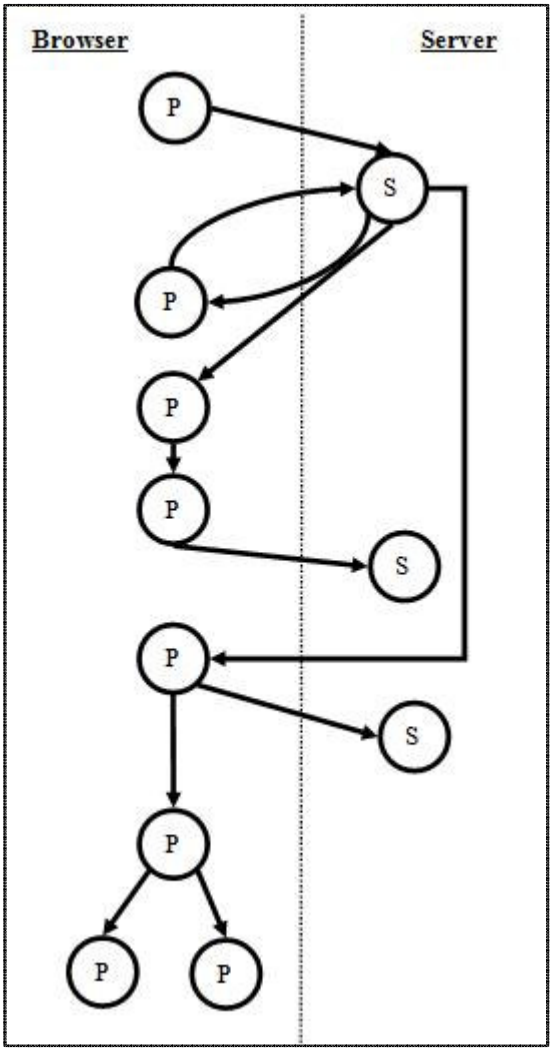

Figure VII-3 P and S are running in the browser and the server respectively.

We are now in a position to look at which components need to run in the browser and which ones need to reside in the server. One thing for sure, Pro resides in the server. This is the practical scenario, although client side storage is a possibility. Let us assume that it is efficient to put S components in the server. This is almost always true unless the network topology is such that communication to Pro is less costly from client than from server. Now deploying P components in the browser and S components in the server we achieve a diagram (Figure VII-3).

Let us define two types of **P**:

1. **Bind-P**: We use two criteria to define the term: (i) if S depends on P, then P is a **Bind-P**. (ii) If a **Bind-P** depends on **P,** then P is a **Bind-P**.
2. **Free-P**: A **P** which is not a **Bind-P** is a **Free-P**.

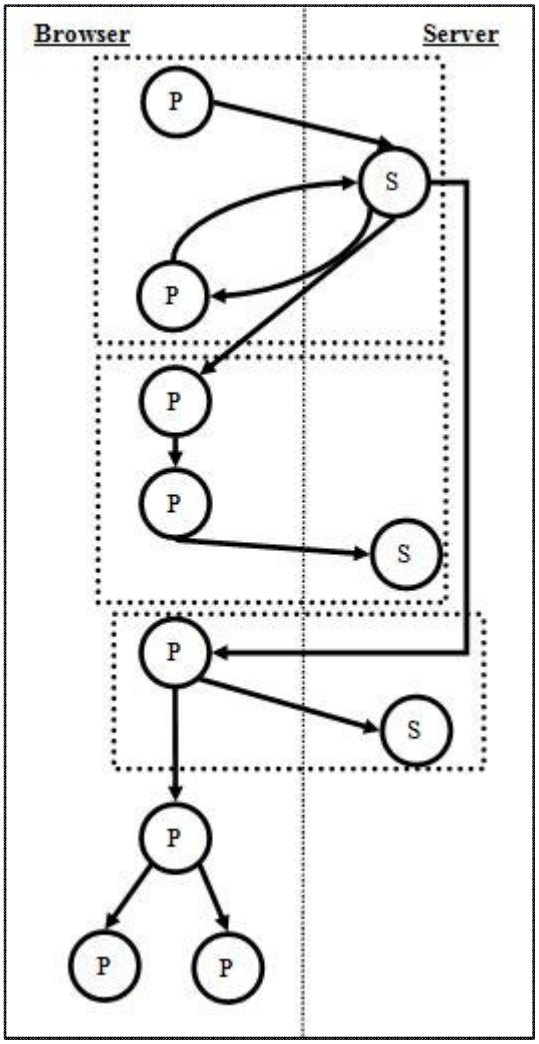

Figure VII-4 Grouping Bind-P and their dependent S in a box.

Let us group Bind-Ps and the S that depends on it together. Figure VII-4 shows such grouping.

We say that the Ps in the box as in Figure VII-4 needs to run in the server. This is because **Bind-Ps** have dependent S which has not finished transaction. According to **rule one** they cannot start showing any output until processing is complete.

Thus the conclusion is that Bind-P and S component need to run in the same place. Most of the time they will be in the server. Apart from thinking S in terms of storage, if we even think of applications where clients need to have a single server which receives requests, processes and responds to one or multiple clients, S and thus, Bind-Ps will reside in the server.

B. Coroutine and Subroutine Revisited

In Figure II-2 and Figure II-3 we show possible ways to position component dependency. In a Servlet setting figure II-2 is not possible because Pre cannot request back once it receives the response. For communication using AJAX it is possible. In such a situation Pro becomes a subroutine of Pre. The controller then resides in Pre to resolve component dependencies in Pro.

For a Servlet Figure II-3 can mean both a subroutine or a coroutine. The problem with the figure is that although the first component in Pro becomes independent early, its output is not available until the second component finishes. This hampers performance. What we mean is, while the relationship between Pre and Pro is asynchronous it is making the relation synchronous. Indeed for a Servlet setting, the subroutine in Figure II-3 can be thought of as a coroutine. When PrePro is in between the two it is actually a coroutine and for web 1.0 we follow that figure.

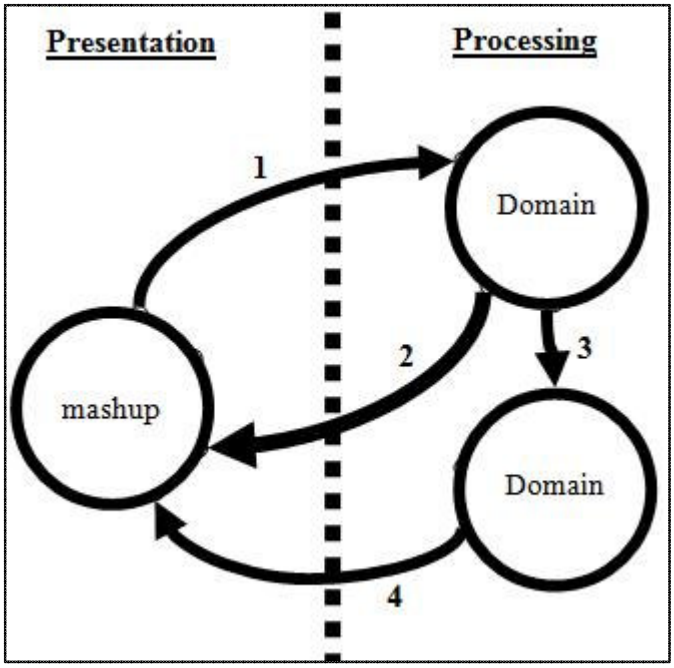

Figure VII-5 Processing is a Coroutine of Presentation

The solution is shown in Figure VII-5. Here Pro and Pre are running as coroutines. Once Pre gives an input, whenever Processing thinks of providing any output, it does. Pre cannot dictate the timing and what will get output. The relationship of Pre and Pro is asynchronous.

Widely used programming languages like Java, C etc are deterministic, and sequential. There the one who calls the subroutine dictates the timing of events and positioning of the returned result. For our problem at hand we do not want Pre to dictate what Pro will do. Thus we look at some other form of language, specifically, Flow Based Programming [Morrison(1994)] to resolve the dependency.

C. The Possible Pre-PrePro-Pro Structures

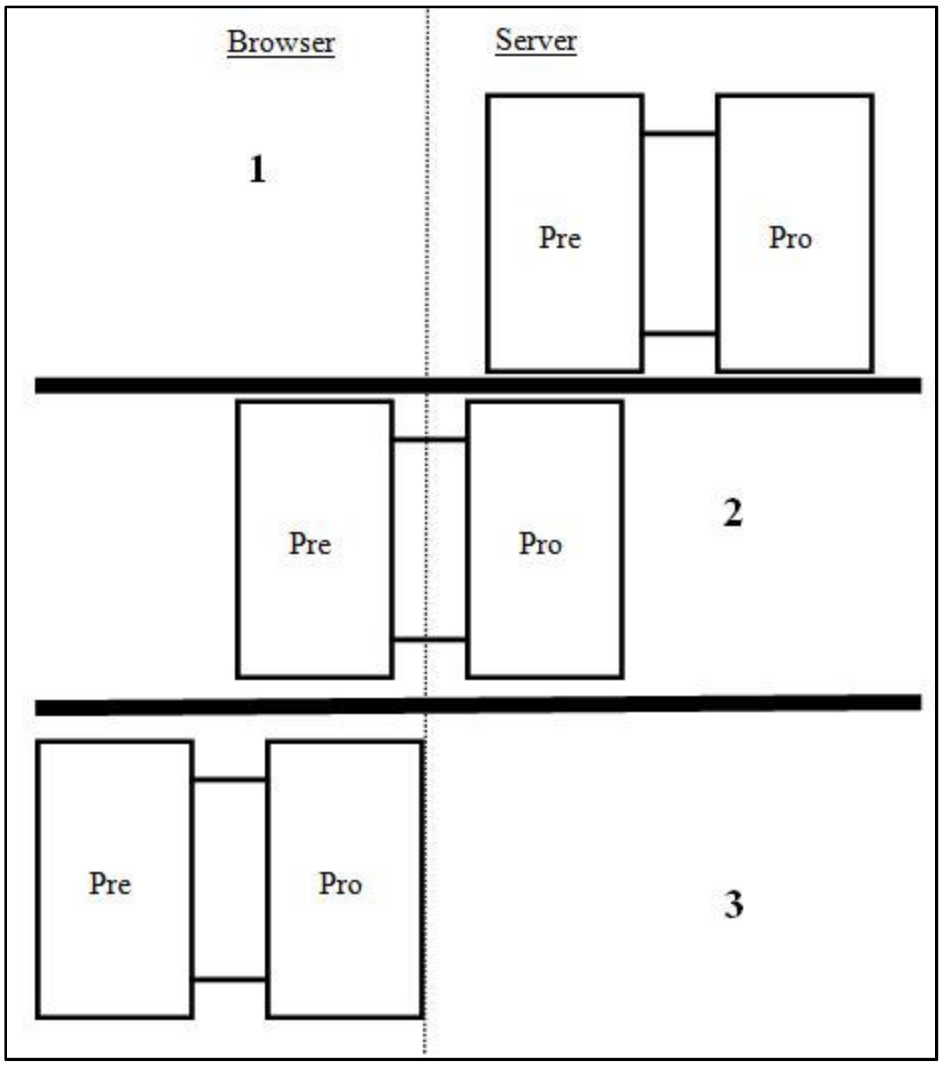

Figure VII-6 Running whole or part of Pre and Pro in the browser. 1, 2 and 3 marks the three possibilities.

Pre can do processing of what PrePro outputs. PrePro outputs a language which is suitable for Webbie to manipulate. Pre remains oblivious until PrePro 'responds' with an output. Now Pre does not have to work with dependent data and it does not need to make type assumptions of Pro data. We discussed in chapter V that either a Turing complete language (JavaScript) can remain embedded in a template or the PrePro output can be

transformed/overlaid (XSL/JavaScript) using that language. Pre becomes reactive to these languages as soon as it starts running in the browser.

Figure VII-6 shows ways to run Pre, PrePro and Pro. On occasions 1, Pre is passive that is whatever PrePro responds is shown in the browser. In other two occasions browser can change Pre capability.

From our previous discussion in this chapter, we see that Free-Ps can run in the browser. Occasion 3 shows that.

Depending on where Pre or Pro runs, PrePro can be run in the server, in the browser or partially in both places.

D. Flow Based Programming

The answer to making Processing modules coroutines of Presentation lies in Flow Based Programming [Morrison and Morrison.(1994)]. He actually compared coroutine and subroutine as "Think of 'cooperative' vs. 'subordinate'". Without further ado, let us see what FBP is and what it can do for us.

Flow Based Programming (FBP) is a useful programming methodology for business data processing. Instead of viewing an application as a single program running many subroutines, FBP sees an application as running multiple processes where each one communicate by means of sending or receiving structured data packets. In FBP every processes are coroutines of one another. FBP is a major paradigm shift from von Neumann machine. There is no list of sequential processes, although at times to solve some problems processes may have to work sequentially. This is similar to the way

worldly systems work where many systems are running at the same time and in case of need they share with one another.

One of the key design methodology of FBP comes from the idea to make business data as primary point of focus and see how they get ``*transformed, combined and separated, to produce the desired outputs and modify stored data according to business requirements."*[Morrison and Morrison.(1994)]. This is opposite to the traditional programming where process is the primary and data is secondary. In business applications the idea is to process, store and report/present data. Data seldom gets lost or deleted. If we think of, for example, of custermer information, does it ever get deleted? The old ones are archieved just to keep a history of the organization. It does not mean that organizations keep everything, but whenever there is a deletion, it is done explicitly.

This storage situation is stated in [Morrison and Morrison.(1994)] like the followin:

"…most of today's computers have a uniform array of pigeon-holes for storage, and this storage behaves very differently from the way storage systems behave in real life. In real life, paper put into a drawer remains there until deliberately removed. It also takes up space, so that the drawer will eventually fill up, preventing more paper from being added. Compare this with the computer concept of storage - you can reach into a storage slot any number of times and get the same data each time (without being told that you have done it already), or you can put a piece of data in on top of a previous one, and the earlier one just disappears."

Another important difference between conventional programming and FBP comes from the issue of timing of events. What this means is that when we call a subroutine in a

conventional programming, the caller needs to know when the other party will return otherwise things will not be right. In [Morrison and Morrison.(1994)] they illustrate the problem of enforcing synchronization in a factory environment: ``*In our factory image, on the other hand, we don't really care if one machine runs before or after another, as long as processes are applied to a given work item in the right order. For instance, a bottle must be filled before it is capped, but this does not mean that all the bottles must be filled before any of them can be capped.*''

Each process in FBP are referred to as components. Some of the terms used in FBP are network, process, information packet, initial information packet, port, connection, subnet etc.

For a given problem, FBP is about making a **network** of connected reusable **components**. All processes or components in FBP are thought of as working concurrently. They pass between them Information **Packets (IPs)**. Processes communicate by making connections and the point where a process and a connection meet is called a **port**. Whenever a component has dependency to another, it waits to **receive** an IP or a collection of them. Whenever a component is ready to deliver it **send**s to its output **port**. Sometimes to initialize a component it may use **Initial Information packet (IIP)**.

We will look at an example FBP network later in the next section.

E. Developing A System

We demonstrate a system which cover ‘orchestration’ or ‘coordination’ of multiple IT systems in an organization. Let us give the name of the organization A. A is

in the business of selling products to customers and they have employees. A has four IT systems: 1. Profile Management System (PMS): keeps track of all the employees information and their designation. 2. Customer Management System (CMS): keeps record of the customers, the amount they purchased, and the employees that did the transaction. 3. Payroll System (): Keeps track of employees salaries, computes them and reports them. 4. Commission Management System (CoMS): knows about the rules that govern commission calculation which is based on the role of the employee and reports them.

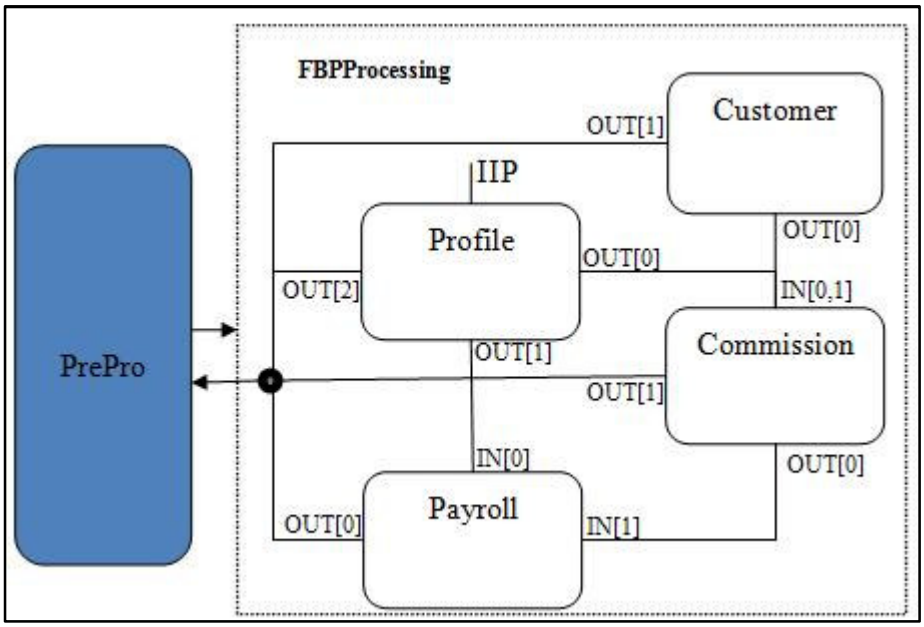

Figure VII-7 The System is executed using Flow Based Programming.

Figure VII-7 shows the FBP network for the stated system. **Customer** can update data independent of any other modules. **Commission** depends on **Profile** and **Customer**. Payroll depends on **Profile** and **Commission.** Thus Profile and Customer will report data first to PrePro, then **Commission** and finally, **Payroll**.

VIII. THE SYSTEM FOR WEB 2.0

In the previous chapter we looked at the issue of positioning Pre and Pro in the server and in the browser, and for distributed components how we resolve dependency. We also talked about developing a system which has a web interface with multiple web parts and which is dependent on multiple components in the server. In this section we look at how do we develop such system using Google Web Toolkit [GWT(2009)] and FBP [Morrison and Morrison.(1994)]. We keep "The Rules of Isolation" as pointed out in chapter 5. Thus, we build the system where Pre is developed using GWT, Pro using FBP and PrePro an event based passage which we named TheGWTPrePro.

A. TheGWTPrePro

TheGWTPrePro is the PrePro passage. To make the PrePro services available to the client, to make the client understand the Event and Listener interfaces, we provide with our package these interfaces and an abstract class which is serves as a listener adapter.

Mainly, TheGWTPrePro's submit method needs to be exposed. That method is exposed to the client GWT application by **GWTPreProService**. Apart from the passage we also provide abstract listener adapter with our package so that any event by PrePro can be received by a GWT Pre with their concrete adapters. We also provide with the package the client side interfaces for PrePro services to be called.

We look at how a GWT written Pre submits the 'how' and 'what' arguments so that it can cross the PrePro passage by an **invoke** to Pro. Every Pre written in GWT that

talks to PrePro will use the same GWT RPC. This means that the service's client side interface, which we named **PreProService**, will always be the same.

First we write the Pre side interface (Table 8.1) and then we make its aysnchronous interface (Table 8.2) which is based on the synchronous interface. GWT requires that the asynchronous interface have the same name with a suffix 'Async' with it. Because of the serialization limitation of GWT RPC we use String parameters to talk to PrePro.

Table VIII.1 PreProService Interface.

```
package net.sks.preproper.gwt.client;

import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;

@RemoteServiceRelativePath("preproservice")
public interface PreProService extends RemoteService{
      public void submit(String processingName, String objToProcess);
}
```

Table VIII.2 PreProServiceAsync Interface.

```
package net.sks.preproper.gwt.client;

import com.google.gwt.user.client.rpc.AsyncCallback;

public interface PreProServiceAsync {
      void submit(String process, String map, AsyncCallback<Void>
callback);
}
```

We use **GWT.create()** to create an instance of the interface. We can cast to the aynchronous interface safely because the proxy implements the asynchronous version. The callback implements **onSuccess** and **onFailure** of GWT's AsyncCallback<T>. Normally, as submit has no return the implemented methods are not used. Once we have the remote service proxy Pre can execute **preproService.submit** at anytime from

anywhere in Pre whenever it sees fit to provide input for processing. However, Pre does not know how the input will be processed in Pro.

Table VIII.3 The code snippet to create a remote service proxy and use it.

```
/**
 * Creating a remote service proxy to talk to the server-side
 * PrePro service.
 **/

      private final PreProServiceAsync preproService = GWT
                  .create(PreProService.class);
/*
 * From anywhere, anytime and on multiple
 * occasions Pre can submit a map to
 * PrePro by the following line
 */
      preproService.submit("fbp", "requestMap",
                                    new VoidAsyncCallback());
/*
 * Here is how the VoidAsyncCallback looks like
 */
      private abstract class DefaultAsyncCallback<T> implements
AsyncCallback<T> {
            public void onFailure(Throwable e) {
                  GWT.log("Error on processing conversation!", e);
            }
      }

      private class VoidAsyncCallback<T> extends
DefaultAsyncCallback<T> {
            public void onSuccess(T aResult) {
            }
      }
```

Now that we made a stub to call it let us see how the server-side implementation lookd like. **GWTPreProService** implements serverside PreProService interface and extends **RemoteEventServiceServlet**. Most of the time GWT's server side implementation uses a **RemoteServiceServlet**. However, in order for a server to generate events we implement GWTEventService library's [GWTEventService(2009)] **RemoteEventServiceServlet.**

Figure VIII-1 GWTPreProService

The service servlet essentially calls theGWTPrePro, the implementation of the singleton passage PrePro. Before calling it lets **theGWTPrePro** to store a reference to **RemoteEventServiceServlet** so that later events can be reported by the passage. We store the servlet reference in a ThreadLocal variable to make it thread safe.

Table VIII.4 Code snippet to show 'submit' call to TheGWTPrePro.

```
public void submit(String processingName, String objToProcess) {
      TheGWTPrePro thePrePro = TheGWTPrePro.getInstance();
      thePrePro.setService(this);
      thePrePro.setDomain(PREPRO_DOMAIN);
      thePrePro.digestConfiguration();
      thePrePro.submit(processingName, objToProcess);
}
```

Before looking at how events get generated let us see how client receives that. The listener is implemented completely in the client side. We provide them as a package of PreProPer. **PreProListener** interface has only one method **respond** (see Table 8.5)

and it extends **RemoteEventListener** [GWTEventService(2009)] which has a method called **apply**.

Table VIII.5 PreProListener Interface.

```
public interface PreProListener extends RemoteEventListener {
      public void respond(String id, String reportedMap);
}
```

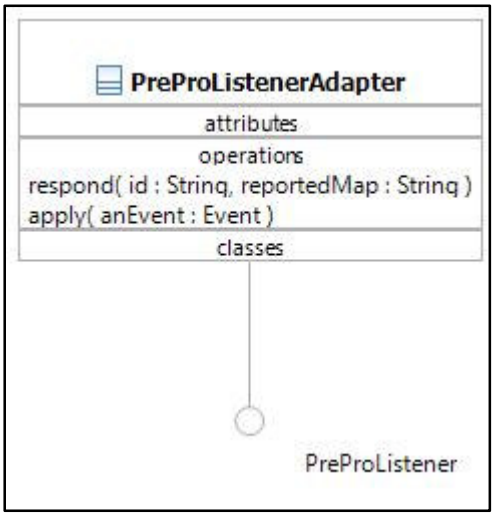

Figure VIII-2 PreProListener and the adapter implementation.

We provide with our package an abstract implementation of the **PreProListener.** The abstract adapter only implements **apply** method and it leaves the **respond** to be implemented by GWT Pre (see Table 8.6).

Table VIII.6 An abstract implementation of the listerner adapter.

```
public abstract class PreProListenerAdapter implements PreProListener {

      public abstract void respond(String id, String reportedMap);

      public void apply(Event anEvent) {
            if (anEvent instanceof PreProRespondEvent) {
                  respond(((PreProRespondEvent) anEvent).getId(),
                              ((PreProRespondEvent)
anEvent).getReportedMap());
            }
      }
```

```
}
```

To receive events Pre just have to listen to events generated by the passage. Table 8.7 shows the code snippet to add a listerner to a **remoteEventService**.

Table VIII.7 A code snippet showing how to receive events.

```
final RemoteEventServiceFactory theRemoteEventHandlerFactory =
                                    RemoteEventServiceFactory.getInstance();
remoteEventService =
                    theRemoteEventHandlerFactory.getRemoteEventService();
remoteEventService.addListener(PREPRO_DOMAIN,
                    new DefaultPreProListenerAdapter());
```

Now lets look at the event class and how it gets generated in theGWTPrePro.

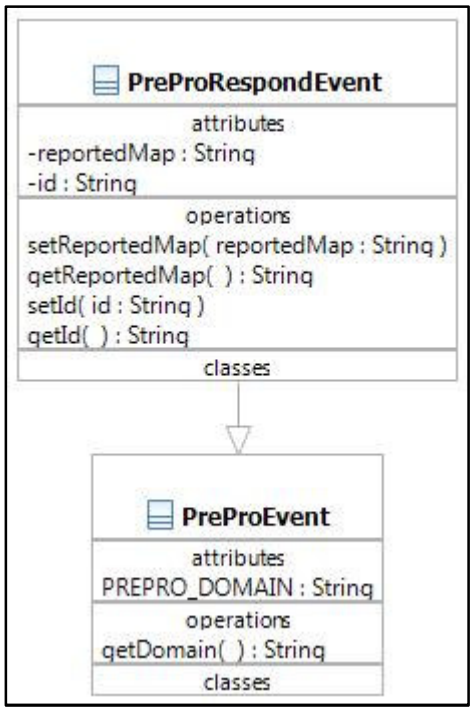

Figure VIII-3 A class diagram of PreProRespondEvent and PreProEvent.

After finishing processing Pro calls **report** to TheGWTPrePro. As Pro can finish processing at different times **report** will generate events as soon as it gets an output from

Pro. In web 2.0 setting Pre may use multiple template in a single page. Each template can will fill a web part where the web part can be filled by multiple domain. In current implementation one web part is filled by one domain. Thus apart from Pro giving the name of a template it can also specify the web part that will fit the template along with the map.

Table VIII.8 TheGWTPrePro report methods.

```
	public void report(String templateName, Object obj) throws
PreProException {
		report("", templateName, obj);
	}

	public void report(String webpartId, String templateName, Object
obj)
			throws PreProException {
		/*
		 * If there is no map provided then the template will be
sent unaltered
		 * to the browser.
		 */
		if (obj == null) {
			String t = (String) templateMap.get(templateName);
			service.get().addEvent(domain.get(),
					new PreProRespondEvent(webpartId, t));
		}
		/*
		 * If there is a map we need to fill the template according
to the map
		 * value.
		 */
		if (obj instanceof Map) {
			StreamTemplate t = getTemplate(templateName);
			if (null != t) {

				String s = t.mapStrings((Map)obj);
				l.config("send: ThePrePro.send() saw
InputStream " + s);
				service.get().addEvent(domain.get(),
						new PreProRespondEvent(webpartId,
s));
			}
		}
	}
```

B. Developing The System

We have the event based PrePro passage developed. For the company **A** we build an application that shows the customer information as it gets updated, the commission of a particular employee for a particular designation depending on the number of customers served and we see how if affects the employee's overall payroll. Before describing how we develop it let us see how the interface might look like (Figure VIII-4).

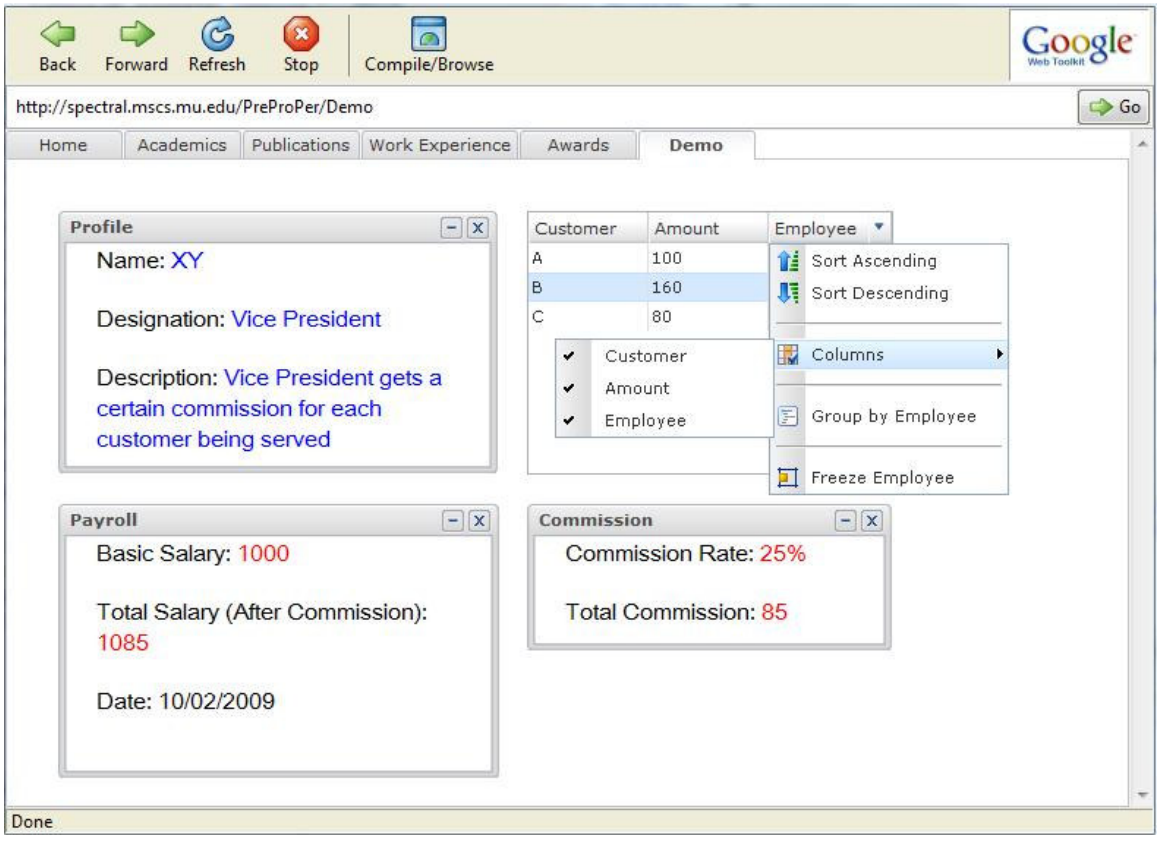

Figure VIII-4 Profile, Commission and Payroll information of employee XY. The system shows the customers that are served. The system is updated anytime a relevant change takes place in the business domain.

Below are the steps needed to develop the system:

1. Webbie uses GWT to write the main template. This template has multiple web parts defined in a layout. Each web part can be a grid, a window, etc. Using GWT we can define events and functionalities which is independent of any data for the web parts.

2. Webbie writes some HTML templates and the Manager or the Architect configures them in GWTPrePro passage. Each of these templates will go to one of the GWT web parts.

3. Webbie subscribes each to the listener to receive events from GWTPrePro.

4. Middie writes components for the system using JavaFBP [JavaFBP(2001)]. Middie or the Architect defines the network for FBP. A component sends data packets to or receives data packets from another component. Whenever the component is finished sending its data packets, it 'reports' to GWTPrePro.

5. Middie 'reports' by giving the name of the web part where its component output will go, the template to use to fill the web part, and the data.

6. GWTPrePro fills the data in the web part template and generates events. Pre receives the event and adds the HTML output to the web part.

Apart from step 4 all other steps require development which are all covered in section VIII.A (steps 1, 3 and 6) and section VI.A and VI.C (steps 2 and 5). Let us look at how step 4 is developed.

TheGWTPrePro is provided with our developed package. Pre uses the interfaces that are exposed. Once any web part makes an input request that keeps the HTTP connection open so that anytime Pro has something new it can update any part of Pre

webparts. To have each domain independent and to coordinate such domains we use FBP.

Let us look at the FBP developed code for the creating different components and connecting them in an FBP network. We make four component for the system: **Profile**, **Commission**, **Customer** and **Payroll**. We look at how to implement that in Java [JavaFBP(2001)]. The **Commission** [] component takes input from Profile and Customer and outputs to **Payroll** and **TheGWTPrePro**. Thus it has two input ports and two output ports. Similarly we define input and output ports for other components.

Table VIII.9 A component Commission skeleton.

```
@InPorts( { @InPort(value = "INP", description = "Packets to from
Profile", type = String.class),
@InPort(value = "INC", description = "Packets from Customer", type =
String.class) })
@OutPorts({@OutPort(value = "OUTP", optional = true, description =
"Packets output to Payroll", type = String.class),
      @OutPort(value = "OUT", optional = true, description = "Output
port, if connected", type = String.class)})
public class Commission extends Component {
      @Override
      protected void execute() throws Exception {


      }

      @Override
      protected void openPorts() {
      }

}
```

The next step is to add the ports together of different components in a way to develop an FBP network. We follow the connections as shown in Figure VII-7. Table VIII-10 shows the way we add components in java to build a network.

Table VIII.10 A code snippet to define the network for the System.

```
import com.jpmorrsn.fbp.components.Commission;
```

```
import com.jpmorrsn.fbp.components.Customer;
import com.jpmorrsn.fbp.components.Payroll;
import com.jpmorrsn.fbp.components.Profile;
import com.jpmorrsn.fbp.engine.Network;

public class SystemANetwork extends Network {

	@Override
	protected void define() throws Exception {
		component("Profile",Profile.class);
		component("Customer", Customer.class);
		component("Commission", Commission.class);
		component("Payroll", Payroll.class);

		initialize("Name", component("Profile"), port("NAME"));
		connect("Customer.OUTC", "Commission.INC");
		connect("Profile.OUTC", "Commission.INP");
		connect("Profile.OUTP", "Payroll.INP");
		connect("Commission.OUTP", "Payroll.INC");
	}
}
```

IX. FUTURE DIRECTIONS AND CONCLUSION

We start looking at some of the possibilities that leap out from our work in the following section and finally in conclusion we discuss what we have done and what its implications are.

A. Future Directions

Pre and Pro can run in the browser or the server. While resolving dependency we defined **Free-P** and **Bind-P. Bind-P**s which can reside in either Pre or Pro, run in the browser. Research and implementation of browser side Pre, Pro and PrePro is indeed a future possibility. One problems caused us not to implement it for this thesis: 1. PrePro is configuration driven and there is no reflection mechanism available in current GWT.

There is another research issue. Most of the systems are likely to have both components Loose-Ps and Unloose-Ps. In such a situation there will likely to be two PrePro passages. How will they communicate. The situation is more understandable if we look back to the figure VII-6. We developed the first two scenario and are proposing the thrid scenario. However in case all three communicate how will that happen. We believe FBP has the answer to it. It requires further analysis.

Processing Pre from the input side is something that we did not look into. One of the common use of such input processing is mostly about input validation. Business specific validation are likely to be done in Pro or we can do that in a Pro which runs in the browser. However, in our implementation Input will first need to go to Pro which may run in the server or the browser.

Although we point out the need to have a template which can reference other template (nested template), we do not have such implementation. Obviously developing in that direction will give a more robust template engine.

B. Conclusion

In the thesis, we look at the problem of "separation of concerns" of Presentation, Processing and Persistence for business application development. Our endeavor was to find the "rules of isolation" both when all components run in the server and when components run distributed in a browser and a server. We agree with the 'separation rules' that T.Parr [Parr(2004)] mentions. However, we do not agree that Pre needs to follow restricted grammar. We can have Pre unrestricted because of our end to end flow and because Per is explicitly isolated by Pro. Pre can only compute on what PrePro provides. This need to make Pre unrestricted is important for a Web 2.0 setting.

The thesis is motivated by the work of PreProPer [Harris(2009)]. In the thesis, we start looking at the system as a practitioner's perspective and we end up finding the principles of isolation.

Important work of the thesis is to think of isolation when the capability of Pre and Pro changes because of the environment where they run. We address the problem of resolving and positioning distributed component dependency by classifying such components and by developing a system to show the use in practice.

We relate the relationship of the concerns, the roles that are involved with the concerns and we see how it affects development of a system. We find important revelations in the thesis from terms such as 'obliviousness', 'coroutine', 'flow', and

‘coordination’. Finally, although the implementation is focused toward the world wide web, the isolation rules are fundamental to any business data processing.

BIBLIOGRAPHY

Bull, Ian R. Storey, Margaret-Anne. Favre, Jean M. and Litoiu, Marin. (2006). *An architecture to support model driven software visualization.* In ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension, pages 100–106, Washington, DC, USA, IEEE Computer Society. ISBN 0-7695-2601-2. doi: http://dx.doi.org/10.1109/ICPC.2006.11.

Kent, Stuart. (2002). *Model driven engineering.* In 3rd international Conference on Integrated Formal Methods, pages 286–298.

DeRemer, Frank. and Kron, Hans. (1975). *Programming-in-the large versus programming-in-the-small.* In Proceedings of the international conference on Reliable software, pages 114–121, New York, NY, USA, 1975. ACM. doi: http://0-doi.acm.org.libus.csd.mu.edu/10.1145/800027.808431.

Dijkstra, Edsger W. (1979). Programming considered as a human activity. Classics in software engineering, pages 1–9, 1979.

Gill, Nasib S. and Balkishan (2008). Dependency and interaction oriented complexity metrics of componentbased systems. SIGSOFT Softw. Eng. Notes, 33(2):1–5,

2008. ISSN 0163-5948. doi: http://0 doi.acm.org.libus.csd.mu.edu/10.1145/1350802.1350810 .

Gomaa, Hassan. Menascé, Daniel A. and Shin, Michael E. (2001). *Reusable component interconnection patterns for distributed software architectures.* In SSR '01: Proceedings of the 2001 symposium on Software reusability, pages 69–77, New York, NY, USA, 2001. ACM. ISBN 1-58113-358-8. doi: http://0-doi.acm.org.libus.csd.mu.edu/10.1145/375212.375252.

GWTEventService, (2009). *Remote event listening for gwt.* http://code.google.com/p/gwteventservice/.

Krasner, G. and Pope, S. (1988). *A description of the model-view-controller user interface paradigm in the smalltalk-80 system.* Journal of Object Oriented Programming, 1(3):26–49.

W. B. 2.0. (2007). *Web services business process execution language version 2.0.* http://docs. oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

Harris, Douglas. (2005). *Communicating components*. http://spectral.mscs.mu.edu/PatternsOfProtocols/Communications/CommunicatingComponents.html.

Harris, Douglas. and Monjur, Mehrab. (2009). *PreProPer code.* http://spectral.mscs.mu.edu/PreProPer/code/.

Hibernate (2009). *Hibernate.* http://www.hibernate.org, 2009.

Johnson, Bruce. (2009). *Reveling in constraints.* Commun. ACM, 52(9):44–48. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1562164.1562181.

Juric, Matjaz. (2005). *BPEL and Java.* http://www.theserverside.com/tt/articles/article.tss?l=BPELJava.

Kavantzas, Nickolas. Burdett, David. Ritzinger, Gregory. Fletcher, Tony., Lafon, Yves. and Barreto, Charlton. (2005). *Web services choreography description language version* 1.0. http://www.w3.org/TR/ws-cdl-10/.

Kojarski, Sergei. and Lorenz, David H. (2003). *Domain driven web development with webjinn.* In OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 53–65, New York, NY, USA, 2003. ACM. ISBN 1-58113-751-6. doi: http://0-doi.acm.org.libus.csd.mu.edu/10.1145/949344.949351.

Lesko, J. (2009). *Mvc frameworks for gwt: A brief survey.* http://supplychaintechnology.wordpress.com/2009/04/07/gwt_mvc_survey/.

Morrison, J. (2001). *Asynchronous component-based programming.* http://jpaulmorrison.com/fbp/2001paper.htm.

Morrison, John P. (1994). *Flow-based programming: a new approach to application development.* Van Nostrand Reinhold Princeton, NJ, July.

Murray, Greg. (2005). *Asynchronous javascript technology and xml (ajax) with the java platform.* http://java.sun.com/developer/technicalArticles/J2EE/AJAX/, June 2005.

Parnas, David L. (1979). *On the criteria to be used in decomposing systems into modules.* Pages 139–150.

Parr, Terrence. (2005). *String template documentation.* http://www.antlr.org/wiki/display/ST/StringTemplate+Documentation.

Parr, Terrence. (2004). *Enforcing strict model-view separation in template engines.* In WWW '04: Proceedings of the 13th international conference on World Wide Web, pages 224–233, New York, NY, USA, 2004. ACM. ISBN 1-58113-844-X. doi: http://doi.acm.org/10.1145/988672.988703.

Pautasso, Cesare. and Wilde, Erik. (2009). *Why is the web loosely coupled?: a multi-faceted metric for service design.* In WWW '09: Proceedings of the 18th

international conference on World wide web, pages 911–920, New York, NY, USA. ACM. ISBN 978-1-60558-487-4. doi: http://doi.acm.org/10.1145/1526709.1526832.

Puder, Arno. (2007). *A cross-language framework for developing ajax applications.* In PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java, pages 105–112, New York, NY, USA. ACM. ISBN 978-1-59593-672-1. doi: http://doi.acm.org/10.1145/1294325.1294340.

Raman, T. V. (2009). *Toward 2w, beyond web 2.0. Commun.* ACM, 52(2):52–59, 2009. ISSN 0001-0782. doi: http://0-doi.acm.org.libus.csd.mu.edu/10.1145/1461928.1461945.

Riabov, A. V. Boillet, E. Feblowitz, M. D. Liu, Z. and Ranganathan, A. (2008). *Wishful search: interactive composition of data mashups.* In WWW '08: Proceeding of the 17th international conference on World Wide Web, pages 775–784, New York, NY, USA. ACM. ISBN 978-1-60558-085-2. doi: http://0-doi.acm.org.libus.csd.mu.edu/10.1145/1367497.1367602.

Sharma, Arun. Grover, P. S. and Kumar, Rajesh. (2009). *Dependency analysis for component-based software systems.* SIGSOFT Softw. Eng. Notes, 34(4):1–6, 2009. ISSN 0163-5948. doi: http://0-doi.acm.org.libus.csd.mu.edu/10.1145/1543405.1543424.

Simmen, David E. Altinel, Mehmet. Markl, Volket. Padmanabhan, Sriram. and Singh, Ashutosh. (2008). *Damia: data mashups for intranet applications.* In SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 1171–1182, New York, NY, USA. ACM. ISBN 978-1-60558-102-6. doi: http://doi.acm.org/10.1145/1376616.1376734.

G. W. Toolkit (2009). *Google web toolkit.* http://code.google.com/webtoolkit/, 2009.

W3C (2001). *Semantic web.* http://www.w3.org/2001/sw/.

Wang, Guiling. Yang, Shaohua. and Han, Yanbo. (2009). *Mashroom: end-user mashup programming using nested tables.* In WWW '09: Proceedings of the 18th international conference on World wide web, pages 861–870, New York, NY, USA. ACM. ISBN 978-1-60558-487-4. doi: http://0-doi.acm.org.libus.csd.mu.edu/10.1145/1526709.1526825.

Programmable Web (2009). *Programmable web.* http://www.programmableweb.com/.

Wong, Jeffrey. and Hong, Jason I. (2007). *Making mashups with marmite: towards end-user programming for the web.* In CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems, pages 1435–1444, New York, NY, USA.

ACM. ISBN 978-1-59593-593-9. doi:
http://doi.acm.org/10.1145/1240624.1240842.

Object Management Group (2001). *Model Driven Architecture (MDA)*, document ormsc/2001-07-01,Architecture Board ORMSC, July.

Bézivin, Jean. (2005). *On the unification power of models.* Software and System Modeling, 4(2):171–188.

Leannah, William (2006). *Exploring business application agility through rules-based processing*, 2006. http://spectral.mscs.mu.edu/PreProPer/LeannahThesis.pdf.

Conway, Melvin E. (1963), *Design of a separable transition-diagram compiler*, Communications of the ACM, Vol. 6, No. 7, July.

Peltz, Chris. (2003). *Web Services Orchestration and Choreography,* Computer, vol. 36, no. 10, pp. 46-52, Oct., doi:10.1109/MC.2003.1236471

Gelernter, David. (1985). *Generative communication in Linda.* ACM Transactions on Programming Languages and Systems (TOPLAS), 7(1):80–112.

Morrison, John P. (2001). *Syntax of JavaFBP (Java Implementation of FBP) and Component API,* http://www.jpaulmorrison.com/fbp/jsyntax.htm.

Filman, Robert E. and Friedman, Daniel P. (2000). *Aspect-Oriented Programming is Quantification and Obliviousness*, RIACS.

Google (2009). *Google Wave,* http://wave.google.com/help/wave/closed.html.

SmartGWT (2009). *Smart GWT – GWT APT's for Smart Client,* http://code.google.com/p/smartgwt/.

XMLC (2004). *Enhydra XMLC – Documentation*, http://xmlc.ow2.org/doc/index.html