Pragmatic Enterprise Web User Interface Development and Reuse: An Industrial Case Study

Jingang ZHOU*, Deyang ZHANG, Dazhe ZHAO

State Key Laboratory of Software Architecture (Neusoft Corporation), Shenyang, 110179, China

Abstract
User Interface (UI) development is a time-consuming task for enterprise application (EA) development. Most of UIs are programmed manually from scratch or with some opportunistic reuse methods (e.g., copy & paste) in the industry, which lead to little productivity improvement. In this paper, we report an industrial case study on a successful UIs reuse program conducted in Neusoft™. In such a program, the product family methodology is applied to UIs development. The key idea is that we use XML-based models in UIs development with components and refinement reuse in a complete model-driven development process. Our approach is developed and being used in the industrial environment with the nature of practicability yet systematicness. Empirical study performed on real EAs confirms that the approach improves UIs development productivity dramatically and a high return-on-investment can be achieved.

Keywords: user interface; component-based development; systematic reuse; enterprise application

1. Introduction
User interface (UI) development is a very laborious and time-consuming activity. A survey research conducted in the early 1990s showed that 50% of application time was devoted to implementation of UIs [1]. This figure is still considered relevant today and even more so with the increased demands of richly graphical web UIs [2], and it is the case of modern enterprise applications (EAs) that delivered through the Web with more and more business services and operations are incorporated with the development of IT [3]. EA vendors are usually struggling in meeting the tough criterion—time to market (with quality), since speed is a critical factor for a company to keep competitive, especially in the context of market globalization [4]. Therefore, any process and technology improvement methodologies or tools can be meaningful in this field.

*This work is supported by the National Electronic Information Industry Development Fund (NO. [2012]407).
*Corresponding author.
Email address: zhou-jg@neusoft.com (Jingang Zhou)
To improve this situation, many approaches from the academy were proposed. However, most of them (if not all) are suffering from a lack of adoption in the industry due to problems like [5, 6, 7, 8]: a) difficulty caused by the formality of methods and ambiguity of terminology, b) incomplete or very high level design without development support, c) difficulty in approaches adaptation for specific environment, and d) assumptions mismatch between academic authors and industrial developers in background and interests.

From the industry view, many companies have established technology (and process) investment and are reluctant to change. Therefore some forms of pragmatic reuse [9] and opportunistic reuse [10], such as copy-and-paste, code scavenging [11], are the “solution” for them to improve productivity. Currently, enterprise web UI development can be categorized into four groups by reuse maturity a) development from scratch with basic widgets, b) opportunistic reuse and adapt some basic existing UIs, c) controlled reuse via templates, and d) automated generation via a domain model with a default UI schema. All of these approaches have weakness or limitations, e.g., too fine-grained widgets, personal knowledge about existing “assets”, small portion of UIs coverage, and reengineering cost. All these approaches are being practiced day by day within our ten years of experience and generally being combined pragmatically. Though they work in some level, it is still costly to reuse those designed not for reuse because of repeated effort on trial-and-error [9, 12], which offsets the benefits of reuse and commonly leads to obscure return on investment (ROI), and consequently damages the confidence of continuous investment on reuse. Therefore the situation remains and the improvement (if has) is imperceptible.

In this paper, we report our case study on a UI reuse program recently conducted in Neusoft™, a big enterprise application vendor in China. We focus on reuse since reuse is the basic discipline for not only software, but also other engineering discipline. Such a program applies the software product lines methodology [13] and bridges the gap between domain engineering (DE) and application engineering (AE) with the idea of “development as reuse” which makes it to be adopted in an incremental way, thus reducing the initial cost. We use model-driven development paradigm in both DE and AE, and take the model artifacts as the central reuse assets, which allow us to leverage many reuse mechanisms and avoid the limitations of directly programming in the final code. The case study performed on four real EAs shows a very positive productivity enhancement, and we find that the structural and XML-based UI model artifacts can be a very efficient way to UI development (with tools support).

The remainder of the paper is organized as follows. Section 2 presents the background of the case study and an overview of the UI development and reuse process. Section 3 details the key practices adopted for the reuse program. Section 4 and 5 present the evaluation and discussion, respectively. Related work is in Section 6 and we conclude with future work in Section 7.

2. Background & Process Overview

2.1. Background
Neusoft™ has developed and delivered a wide range of IT solutions and services for various industries such as telecom, energy, e-government, finance, manufacturing, healthcare, education and more with a platform-based software development (PBSD) paradigm plus component-based development (CBD) under JEE technology framework [14]. However, due to the vast diversity requirements and multidisciplinary nature [15] of UIs, UI development is still immature as opposed to the business logic part in which CBD is commonly used, e.g., [16].

In Neusoft™, a typical UI development scenario for each new application begins with checking out an existing application from a configuration management system, then manually adapting the specific part of the UIs with very few tools support. Along this way, problems occur like both the bad and the valued parts are inherited, modifications are hard to identify in the mist of Web page tags, etc. Above all, whether a UI can be reused or not greatly depends on the developers’ awareness and knowledge on it, which limits the reuse level and scope, and makes the potential of reuse imperceptible and hard to measure. On the other hand, the pressure of application delivery is increasing under the current competitive business environment. So, the company initiates a reuse program to leverage the potential developed assets efficiently and to lower the development cost.

Rooted in the industrial environment, the approach sets up the following principles to assure its adoption:

- **Utility.** The approach should have a quick ROI or incremental nature to avoid big initial investment.
- **Generality.** The approach should have a size-free character and can be scaled to large applications.
- **Practicality.** The approach should focus on the main reuse issues of UIs and have low cognitive burden on developers.
- **Systematicness.** The approach should conform to some established software engineering principles and be supported by systematic reuse.

### 2.2. Process Overview

The goal of the reuse program is to establish a company level reuse discipline for UIs, and make it measurable and improvable continuously, which leads to the expansion of PBSD and CBD from business services to UIs. Fig. 1 illustrates this paradigm for UIs.

The process has two essential stages, i.e., domain engineering (DE) and application engineering (AE) in terms of software product line engineering, and four main activities. In the DE stage, the infrastructure for systematic UI reuse is prepared, while in the AE stage, the UI assets are reused to produce particular applications. All the UI assets are in a model format rather than final executable UI code (like JSP or html). Such a model based approach allows us to reuse them flexibly and improve productivity with the aids of tools.

The main concept behind this process is “development as reuse”, by which we encapsulate the UIs in well structured models both in DE and AE, i.e., working artifacts in both stages are models not final code, and produce the final UIs (to be deployed) via model transformation. The key activities and techniques used will be illustrated next.
3. Approach & Practices

Our approach takes a PBSD paradigm which provides a technical platform for UIs to be relied on and constructed according to some architecture pattern constraints the platform sets on them.

3.1. Domain Architecture Extraction

Domain architecture extraction (DAE) is the first step to implement an architecture based software constraint assurance paradigm. The main purpose of DAE is to build a uniform architecture platform at the company level to realize maximal reuse.

In our approach, the domain architecture for UIs is set as:
- JSP/Servlet: the most general JEE front end technology used in the company.
- JavaScript (Dojo): an Ajax style JavaScript framework used to support rich internet applications.
- Struts framework: a popular server end requests forward and processing framework.

3.2. UI Meta-Modeling

UI development is tedious and error-prone, the main difficulties lie in:
- Tags for different roles are mixed with confusing format in the code resulting in the code not easy to understand and maintain.
- A lot of manual typing without the aid of powerful tools leads to many accidental errors often found in testing or running time.
- No efficient reuse mechanism except the basic UI controls and the copy/paste reuse approach.
To address these problems, we provide a model-based development method in which all working artifacts of UIs are models. Such a model-based approach allows us:

- Record UI artifacts in well structured models with a uniform set of tags.
- Develop tools efficiently to aid models development with separation of concerns, model checking for model consistency, and code generation to enforce code patterns and reduce accidental typing errors.
- Leverage several reuse mechanisms based on the model format to facilitate UIs reuse.

The core of the model-based approach is a meta-model which guides how a UI model should be correctly constructed. Fig. 2 shows a simplified version of the meta-model in our approach.

The central concept of the model is view. A view is the abstraction of a web page which includes:

- Client end page presentation via controls and a theme of the view. The latter specifies the view’s style.
- Interactions between the client end (operation) and the server end (action). They can be divided into two categories, an entry and the rest of processors. The former performs the initial interaction when a view is loaded, while the processors for the rest of the time. The interactions are bound with control events. And
- Data to be processed and presented, which include schema definition (dataObject) and runtime properties (rowSet). Data are bound with controls.

The view model contains all information we need to instantiate a UI page and is encoded in a XML document. Via the XML-based model description, we cannot only transform them into other formats by using some XML processing tools like XPath or XSLT, but also allows us leverage more efficient reuse mechanisms.
3.3. UI Componentization

UI componentization is the process of extracting or designing reusable UI models for later reuse. Based on the view model, we call a UI component a ViewComponent (VC), which is an independent UI part both in the logical and physical perspective. A VC has its own presentation (visual part) and interaction (via operations and events) semantics. Both are encapsulated in an autonomic entity (i.e., the component itself). Such encapsulation and autonomy form the basis of CBD and facilitate components assembly.

The composition of VCs takes both the presentation and collaboration among them into consideration. Presentation composition is natural via the parent (composed) component’s layout manager; while the collaboration is achieved by using event publish & subscribe (P/S) mechanism according to the P/S information specified in the <interEvents> part of the VCs. Fig. 3 omitting detail JavaScript code for P/S events demonstrates a scenario where two VCs are composed in a Staff_Retrieval view. VC1 represents the search criteria and VC2 represents the search results.

The P/S mechanism allows two or more VCs to be connected dynamically and loosely. The VCs are not aware of the existence of each other, and only care about the events. The composed component (assembly) in our approach also conforms to the VC model, i.e., the assembly has its...
own visual part (combination of children’s presentation) and event interactions by embracing the children’s P/S interfaces*. Therefore, our VC model is hierarchical, which not only allows us accumulate any granularity components, but also makes a uniformity of components composition and eases learning and adoption.

Another technique to facilitate reuse is parameterization which is used in the view or VC models to make them configurable. E.g., the VC model fragments above set data as parameters to make them reusable in different data model scenarios while keeping the same look and feel. This is very helpful in enterprise applications since most UIs are used to collect data and process, present them with similar behaviors.

3.4. Model Driven Development

MDD is the central methodology for both DE and AE. By using UI models, we apply some fundamental principles of SE to reuse, i.e., refinement and divide-and-conquer (via CBD). CBD approach is illustrated above. We show the refinement approach to models, the general MDD process, and the generated UI artifacts architecture in this section.

3.4.1. Refinement reuse

Refinement allows us to develop software incrementally [18] and is the main task of application developers to customize specific UIs.

We bring the refinement concept into object-oriented reuse in which a base view (or VC) can be refined by a child view (or VC) in a way similar to class inheritance. This simplification lowers the design complexity of base views and their refinements.

Fig. 4 illustrates the concept of refinement reuse, in which Fig. 4(a) represents a basic security checking for login and Fig. 4(b) demonstrates the final result generated by adding a verification code to password anti-crack for the login of Fig. 4(a) with the loginEx refinement (key elements and modified parts are emphasized with underline).

<super/> means retaining original elements of parent view under <tablelayout> (with id=“layout”). The script for check is also refined with an extra check for verification code. To reduce the complexity of the computation of composition in the final UI generation phase, we use override policy (default) for script refinement, namely, the script within CDATA includes checks for not only verification code, but also name and psw. (generated automatically via view creation tool) inherited from the parent view, which. But the refinement policy also works by inserting the keyword super. The refinement reuse will take an effect in the UI generation phase.

3.4.2 Process and tools support

Tools are an indispensable part of a development process. To support the web UI development and reuse, we have developed a UI designer environment covering all the UI development aspects for our approach discussed above. Fig. 5 illustrates the general process and main modeling tools.

* We use a simple evaluation propagation policy for the P/S interfaces of the assembly which delegates the P/S interfaces of its children properly.
The first step is data modeling to prepare business data to be processed by the data modeling tools with an inner tool to help modelers to extract database schemas. After which, the UI presentation and interaction modeling have begun and proceeded alternately through the UI modeling step. We use a manual rather than an automated way to UI modeling because the default UI presentation (generation) schema set by the automated approaches cannot accommodate the complex UI requirements and always needs laborious adaptation in real practices as well as they not taking assets reuse into consideration. The transformation tool transforms the UI models into deployable UI code in a template-based code generation approach with XSLT after validating and checking the model completeness and consistency. Specifically, we use closure technique in JavaScript language to qualify the variables and identifies of VCs with namespaces to avoid naming conflicts among different VCs when they being composed in the generated artifacts.

Other tools contained in the tool environment include a fast entry viewer for all the modeling tools, some wizards for UI reuse and creation, a workspace viewer for UI assets check-in,
checkout, and navigation, etc.

3.4.3 Generated UI artifacts architecture

The generated artifacts architecture for a view is depicted in Fig. 6 with a separation of concerns principle since a view has many responsibilities. The presentation is encapsulated in the view.jsp with the UI controls mapping to the Dojo UI controls. The view JavaScript object (declared in view.js) is responsible to manage the lifecycle of client side interaction objects (defined in entry.js and processor.js), while the server side interaction objects are mapping to Struts actions (Entry.java and Processor.java).

![Fig.6 Architecture for the Generated UI Artifacts](image)

4. Evaluation

4.1. Case Study

Our approach is being validated in a business line (BL1) of the company. This BL has rich domain knowledge and system development experience and is competitive in that field. Also, the BL has rich application assets and use PBSI paradigm with a DE team and several AE teams.

We first train the technical staff of BL1 for our approach, e.g., how to use the tools, how a UI model created efficiently and more reusable. In the end of the training, all the staff of BL1 can use our approach for real development. To validate the productivity gained with the new development approach, we collect production data on the domain UI assets extraction and four real applications launched successively by BL1. To reduce bias on process control, we do not allow new staff to join BL1, but allow them quit.

4.2. ROI Evaluation

Our ROI analysis covers the UI assets and four derived applications with BL1. The return means the saved effort in the UI development with our approach, and the investment denotes the cost of
domain engineering and reuse capability establishment. Both return and investment are qualified through the unit Person-Month (PM). The sub-items of return and investment are listed below with denoting symbols.

- **$R_i$:** saved effort on ith EA UIs development.
- **$R_\Sigma$:** accumulated $R_i$ for all EAs.
- **$I_{org}$:** total training cost for the BL1 for the approach including both of the BL (denoted by $I_{org-bl}$) and the approach inventor (denoted by $I_{out}$).
- **$I_{assets}$:** domain engineering cost for BL1.
- **$I_{tools}$:** tool environment development cost.

The investment data of these sub-items have been recorded in a project management system of the company and shown in Table 1 with some data on UI assets for later calculation for return. From table 1, we see that the total investment is 55.5 PM whilst the cost of BL ($I_{bl}$) is 50 PM (sum of $I_{org-bl}$ and $I_{assets}$).

<table>
<thead>
<tr>
<th>Tools</th>
<th>Org</th>
<th>Data on UI Assets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>I_{org-bl}</strong></td>
</tr>
<tr>
<td>23</td>
<td>1.5</td>
<td>9</td>
</tr>
</tbody>
</table>

The return on ith application is calculated through the following equation:

$$R_i = C_{before} - C_{after} \quad (1)$$

Where $C_{before}$ means development cost before our approach adoption, i.e., traditional approach, $C_{after}$ means development cost after our approach adoption. The main reason for us to use Eq. (1) to calculate $R_i$ is that the business services development of BL is component-based SPL and very stable, so that the saved cost can be attributed to UIs development. Though $C_{before}$ is an estimate, many similar applications developed before can serve the references in UI complexity and size. Table 2 illustrates the collected data on the four applications. The reuse related data are collected by a statistical tool we developed to scan and extract the reuse information from all the application views.

<table>
<thead>
<tr>
<th>#App.</th>
<th>$C_{before}$</th>
<th>$C_{after}$</th>
<th>$R_i$</th>
<th>#derived views (from #base views / with #total times of VCs)</th>
<th>#totally new created views (with #total times of VCs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>77</td>
<td>49</td>
<td><strong>28</strong></td>
<td>387 (108/77)</td>
<td>153 (48)</td>
</tr>
<tr>
<td>2</td>
<td>75</td>
<td>56</td>
<td><strong>19</strong></td>
<td>207 (99/105)</td>
<td>147 (78)</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>21</td>
<td><strong>12</strong></td>
<td>184 (65/53)</td>
<td>87 (63)</td>
</tr>
<tr>
<td>4</td>
<td>58</td>
<td>39</td>
<td><strong>19</strong></td>
<td>195 (79/82)</td>
<td>143 (47)</td>
</tr>
</tbody>
</table>

From table 2, we see the $R_\Sigma$ of the four applications is 78 PM, so we have

$$ROI = \frac{(R_\Sigma - I_{bl})}{I_{bl}} = \frac{(78-31)}{31} \approx 1.516$$

for BL1. Note, this is the ROI only for four applications. Each year, BL1 will release two versions of the core UI product line which will cost extra 15% of $I_{assets}$ ($I_m$, the cost on training can be neglect), whilst at least 20 applications ($\Sigma_r$) will be derived, this will yield an annual ROI of 58.09 ($\frac{R_\Sigma \times 4 \times 20}{I_{assets} \times 0.15 \times 2} - I_{assets} \times 0.15 \times 2$) for the following years with the simple assumption that the average UI complexity and size of all these applications is equal to those of
4.3. Findings and Results Analysis

We have gained the following findings:

The approach improves UIs development efficiency at least by 63.9%. If we simply assume that the ratio \((r)\) of UIs development cost \((C_{UI})\) to total application cost \((C_{before})\) for all the four applications are 0.65, 0.60, 0.55, 0.50, respectively, we can get Table 3 for \(C_{UI}\) and improved efficiency \((P)\) which is calculated through the equation:

\[
P = \frac{R_i}{C_{UI} - R_i}
\]

<table>
<thead>
<tr>
<th>#App.</th>
<th>(r (0.65))</th>
<th>(r (0.60))</th>
<th>(r (0.55))</th>
<th>(r (0.50))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50.05 (127.0%)</td>
<td>46.2 (153.8%)</td>
<td>42.35 (195.1%)</td>
<td>38.5 (266.7%)</td>
</tr>
<tr>
<td>2</td>
<td>48.75 (63.9%)</td>
<td>45 (73.1%)</td>
<td>41.25 (85.4%)</td>
<td>37.5 (102.7%)</td>
</tr>
<tr>
<td>3</td>
<td>21.45 (127.0%)</td>
<td>19.8 (153.8%)</td>
<td>18.15 (195.1%)</td>
<td>16.5 (266.7%)</td>
</tr>
<tr>
<td>4</td>
<td>37.7 (101.6%)</td>
<td>34.8 (120.3%)</td>
<td>31.9 (147.3%)</td>
<td>29 (190.0%)</td>
</tr>
</tbody>
</table>

The resuability in UI assets needs to be improved. From Table 2, we can see that the ratios of number of derived views to the number of total views in each application (except app.1 from which the UI base is referenced) are in the interval \((57\%, 68\%)\). The numbers of distinct view and VC reused for all the four applications are 122 and 46, respectively, computed via our statistical tool. These figures reveal, on one hand, considerable diversity exists in UIs so that developers may choose anew over reuse when they consider that the benefits of reuse are not evident; on the other hand, the UI base may need to slim to pay more attention to “core” UIs, which will reduces maintenance and evolution cost and improve reusability. Anyway, such data provide a benchmark for further improvements.

The approach improves productivity even without taking reuse into consideration. This is not obvious from table 2, but an inference we make, since the ratio of newly created part for each application is high, so that we have confidence to infer that the development by our approach without reuse also facilitates productivity. The positive feedback from BL1 also confirms this inference. This may probably lie in tools support (with code generation) which leads to faster UI design with less code typing, less probability of bugs introduced, as well as related testing. But we have not collected valid data yet to prove it. A thorough case study may need to measure how much development without reuse contributes to the productivity improved.

4.4. Threats to Validity

The estimation of \(C_{before}\). In real industrial case, we cannot develop two same applications with different approaches. Even so, many issues still exist like the order problem for the same staff, the different skill background for different teams, etc. To reduce the bias, we use the guides provided
in [33]: assess the complexity of the applications prior to and after their completion for correction, use an average estimation assessed by several experts, etc. Also, many similar applications exist for reference. So, we believe the assessment of error is very small.

*The new approach has no influence on the business logic part.* Our approach focusses on UI development and does not make any assumptions on the development of business logic. In the case study, BL1 still use their traditional CBD and PBSD for business logic parts and told us the new approach does not influence them on business logic parts development during our interview with them. However, the psychological impact, e.g., the pleasure caused by the new approach because of fast UI development may still has subtle influences, e.g., less pressure, fewer bugs introduced.

*The accuracy of some hypothetical variables.* In the calculation of ROI and productivity, we introduced variables for some assumptions, such as $I_m$, $\sum_v$, and $r$. For $I_m$, we take a more conservative figure (15%) than the general industrial figure (10%) [34] according to the actual performance of BL1. And we also set the values for $\sum_v$ and $r$ as a real case and reasonable assumptions, respectively.

5. Discussion

**Systematic Software Reuse & Adoption.** Systematic software reuse is advocated by the academic with preplanned reuse and also proved in producer-driven product markets. But this is not the norm, and most of the software (especially the EA) vendors are undertaking some pragmatic reuse approaches with an incremental nature to improve the reuse program (if have). Our approach can be used in the extractive and reactive way for SPLs adoption [20] in a valued-based approach [17]. The model-based design and development makes the transition (from AE to DE) smooth via AE first to accumulate “assets” since these “assets” have a same shape with truly assets except for abstraction, configurability, and reusability. So an organization can decide to adopt a systematic reuse approach later when resources (time, “assets”, staff, etc.) are available.

**OO and Structured Design.** OO is a reuse paradigm in language level for class reuse. Our approach takes an OO approach for UI reuse that UI refinement is a kind of class inheritance and the relationship between a view and its belonging VCs is analogous to the aggregation relationship of a class and its enclosing classes. Both of these are achieved via UIs composition due to the structural nature of the XML-based UI model in which the elements of <processors>, <widgets>, <scripts>, etc. are analogous to the field and method declarations in Java and each has its own composition policy (e.g., overriding or appending).

**Future Web UI development.** With Web (application) technology development and service pattern evolution, mashup becomes popular in web application design and UI integration emerges as an important issue after integration in data and application level [21, 22]. Our VC model can serve as an implementation level model for UI integration due to the autonomy nature of VCs and the P/S messaging mechanism which is the de facto of UI integration [21]. Under such a context, we are designing a runtime framework for VCs be directly mapped to runtime entities for online UIs reuse and aggregation.
6. Related Work

Gaedeke et al. [23] propose WebComposition approach via WCML, a language based on XML, in which Web entities are modeled as components with a set of properties specifying the component state and a set of operations specifying the component behavior. All these components are composed and transformed to target Web pages via WCML parser. WebComposition is based on a prototype-instance OO model rather than a class-oriented OO model we use. Gomes de Sousa et al. [24] propose XICL, a markup language which combines XML, HTML and ECMAScript (or JavaScript). A UI component includes properties, script, structure, and events and can be reused via <import>. Though the authors claimed that a component can be extended, no details were proposed. The XICL studio used to write XICL files, compose and interpret them is very preliminary in writing XICL with manually typing. Fiala et al. [25] propose a three-level architecture for UI components, namely, media, content unit, and document components with more and more coarser granularity. All components are described via XML and composition occurs at document component level. The document components only focus on presentation composition rather than events as opposed to VCs. The component adaption method used is similar to template we used with variability declaration. Tanaka et al. [26] propose an approach to clipping and reusing fragments of Web pages in order to compose new applications targeting HTML elements (specifically HTML forms with no attention to JavaScript). Also, the applications created in this way are to be executed within their tool –HTMLviewPad.

Schramm et al. [27] propose a model-based generation approach for enterprise UIs via a domain model (with predefined mapping rules between domain model to UI widgets presentation) which allows the layout to be manually adjusted and targets for an Eclipse RCP based platform. But this approach does not consider UI reuse. Costa et al. [28] propose another Web UIs generation approach aligning with MDA concepts. However, this approach puts much effort on UI modeling rather than development and reuse CBD. Cadavid et al [29] propose a mapping and navigation rule set between data schema and web pages to facilitate code generation via domain model. Behrens et al [30] propose a full software engineering lifecycle process for Web UI development from requirement analysis to implementation with extended UML profiles and a UI is composed via some UI objects (a UI component model used in [30]). Maras et al [12] proposes a widget extraction approach to extract all related artifacts scattering in UI pages to lower UI widget reuse cost and facilitate widget reuse. Because of the well-structured format of widget description of our model, such an approach can be achieved easier.

Pettersson et al [31] demonstrate the power of XVCL in Web pages (also other kinds of artifacts) design and implementation for a Portal product line. XVCL use a frame-based meta-programming paradigm to greatly remove clone in artifacts. However, a prerequisite for XVCL being used is a stable domain with considerable commonality in products, which is not the case for ordinary EAs. We design a frame refinement technique [32] to alleviate such burdens to allow Frame technique be used in a small portion of EAs easily.

7. Conclusion and Future Work
Improving software productivity and quality is an essential issue of SE. In this paper, we proposed an industrial case study for enterprise Web UI development under a pragmatic and systematic reuse program. Such an approach leverages OO reuse and CBD in a domain specific way with full tool support. The low cognitive complexity in concept and pragmatic operations support makes it popular among developers. The case study also confirmed that our approach can improve UI productivity greatly. We believe our approach has a good potential for providing a foundation to pragmatic and systematic enterprise UI development and even software industrialization. Note, the UI development approach illustrated in this paper is a main part of a more complete MDD for web applications [19].

There are three folds for future work:

- Frame-based [31] model assets componentization adoption to realize model-based meta-programming and improve productivity further.
- More heterogeneous UI component technologies (e.g., applets, flex, mashups) introduction to make the approach fit for more application scenarios.
- Tools augmentation to ease DE tasks on legacy applications, e.g., to support VCs extraction semi-automatically in a heuristic way.

References