Published in the Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)

Reengineering User Interfaces of E-Commerce Applications Using Business Processes

Qi Zhang, Rongchao Chen and Ying Zou
Department of Electrical and Computer Engineering
Queen's University, Kingston, Ontario, Canada
{3qz, 4rcc} @qlink.queensu.ca, ying.zou@queensu.ca

Abstract

E-commerce applications are designed to streamline the business processes for an organization. Graphical user interfaces allow business users to perform daily business activities by interacting with e-commerce applications through menu-driven user interface components, such as toolbars and dialog windows. However, business users are often overwhelmed by the enormous functionality available. Users struggle in deciding where to start and where to go next in order to accomplish tasks required by business processes. In this paper, we utilize the knowledge embedded in business processes to reengineer the user interfaces of existing e-commerce applications that implement business processes. We aim to improve the usability of user interfaces by providing contextual information and guiding users to fulfill business processes step by step. We evaluate our proposed approach by reengineering the user interface of an existing e-commerce application.

1. Introduction

A business process is a sequence of tasks which need to be carried out to achieve business objectives for an organization. For example, a book purchasing process may consist of several tasks, such as selecting a book from the catalog, using a credit card for the payment, and printing out a receipt. A workflow provides specifications for describing business tasks (e.g., selecting a book), roles (e.g., customers and sales representative), and data (e.g., a book order request) involved in a business process. E-commerce applications support and execute workflows for various business scenarios, such as call center applications, customer relation management, and on-line retail stores. E-commerce applications are designed to streamline the business processes for an organization. Graphical user interfaces allow business users to perform daily business activities by interacting with e-commerce applications through menu-driven user interface components, such as toolbars and dialog windows. To fulfill the growing requirements from the business world, e-commerce applications have gradually evolved to provide sophisticated functional features in user interfaces. However, such rich features are often not obvious for users to navigate in the user interfaces (UIs). As a result, business users, especially novice business users may struggle in deciding where to start or where to go next after a result is received from a particular component in the user interface. Business users are required to take continual training to use e-commerce applications, since the user interfaces of these e-commerce applications are frequently updated to reflect the continuous evolution of the underlying business processes. In addition, the user interfaces of e-commerce applications are designed from an IT personnel perspective, rather than the perspective of business users [19]. The context of the user interface presented to the business users may not directly follow their natural work rhythms. These problems increase operational costs and decrease work efficiency. Therefore, a more systematic approach is required to design business applications to ensure business users can carry out their work more efficiently and effectively.

In this paper, we propose an approach for reengineering user interfaces of existing e-commerce applications using workflows. We leverage the contextual knowledge in business workflows to improve the usability of user interfaces of e-commerce applications. We automatically generate navigational sequences which indicate the progress of a workflow and associate tasks with the UI components which implement these tasks. Our aim is to automatically guide users, interacting with an e-commerce application, by prompting the next UI components. Furthermore, we adapt the contextual information embedded in workflows to dynamically show only the UI components relevant to the currently progressing workflow, and hide irrelevant UI components.

The rest of this paper is organized as follows: Section 2 describes a sample scenario for applying our approach. Section 3 presents the architecture of our business process driven user interface. Section 4 discusses the techniques to establish the correspondences between tasks and UI components. Section 5 presents the methods used to generate navigation sequences for the user interfaces. Section 6 discusses our case studies which evaluate the reengineered user interface. Section 7 presents related work. Section 8 concludes the paper and presents future work.

2. Sample Scenarios

In this section, we illustrate several usability problems in the design of user interfaces through an example e-commerce application. In general, the usability is a software quality attribute which measures the extent to which an application can be used by users to achieve effectiveness, efficiency and satisfaction in a specified context of use [3],[7]. Furthermore, the usability can be decomposed into five attributes: (1) *learnability*, which measures the ease of learning the application functionality; (2) *efficiency*, which measures the ease of use and the level of productivity attainable by the user; (3) *memorability*, which measures the ease of remembering the application's functionality; (4) *low error rate*, which measures how the application support users in making less errors; and (5) *user's satisfaction*, which measures how the users enjoy the application.

Figure 1 depicts a screenshot of a call center sales application, where Customer Service Representatives (CSRs) can create and manage customer orders by receiving phone calls from customers. The user interface of the application is divided into five UI components: A *Customer viewer* displays the stores and the current customers registered in each store; A *Customer editor* is used to edit customer's records; A *Related products viewer* provides similar or related products that a customer may want to purchase; A *Reminder viewer* records any reminders for the CSR; and a *Promotions viewer* presents promotions for a product. As shown in Figure 1, the *Reminder viewer* is placed on top of the *Promotions viewer*.

Such an application supports approximately 30 business processes. A UI component, such as a menu on the top, an editor and a viewer, is designed to fulfill one or more business tasks. A typical *Purchase order* process for the call center sales application is shown in Figure 2. To fulfill this process, a CSR is required to navigate through different UI components. He/she first looks for the menu bar that can initiate the *Customer*

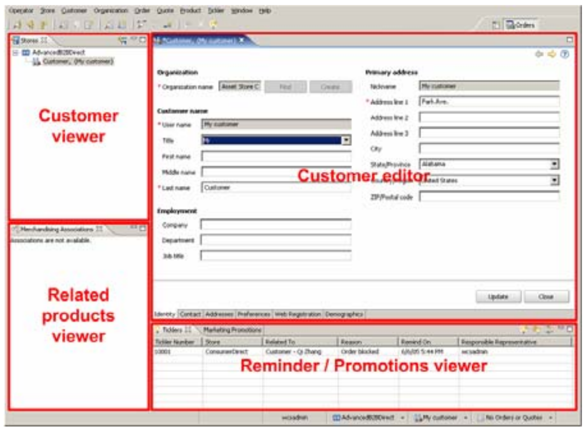

Figure 1: User Interface of a Call Center Application

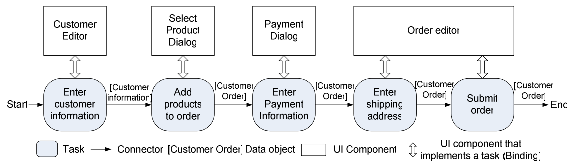

Figure 2: Binding Tasks to User Interface Components for a Purchase Order Process

editor. A CSR, especially a novice one, needs to be familiar with the functionality of a large number of buttons in the menu bar, and must decide which menu is used to perform their current task. After entering the *Customer information* in the *Customer editor*, the CSR needs to check an appropriate UI component in the user interface in order to trigger *Select Product Dialog* for adding products. At this point, no help is available for a CSR to locate UI components needed to complete the process. In short, a CSR should memorize the functionality of each UI component to perform his/her tasks. The learnability of this user interface design is low. Moreover, during a phone call from one customer, a CSR often handles multiple processes which involve opening numerous windows and dialogs. In this case, a CSR has to keep track of the correspondence between a workflow and their UI components. The CSR's work efficiency may decrease and his/her error rate may increase.

The *Related products viewer* and *Promotions viewer* have data displayed only when a product is added to an order in the *Add product to order* task. However, when no data is available for display, these viewers remain open as shown in Figure 1. These unused viewers waste the limited screen space. Moreover, a CSR is often unaware of the availability of data in these viewers, especially when the CSR places the *Promotions viewer* and the *Reminder viewer* underneath the other UI components to save screen space. The user interface cannot capture the CSR's

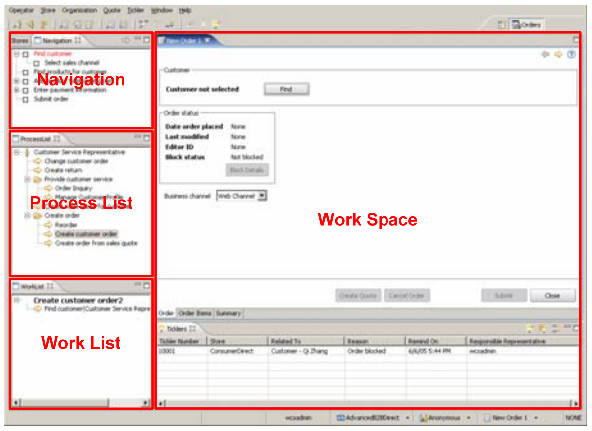

Figure 3: Prototype Process Driven User Interface for a Call Center Application

work context, and does not adjust to help business users to work more effectively.

In our research, we help the CSR in the above scenarios. We integrate the workflows to the exiting user interface and provide navigational and contextual information for the CSR to move through the user interface. Figure 3 illustrates the reengineered user interfaces of the call center application in Figure 3. In the reengineered user interface, we provide the following features:

Support for Evolving Business Processes: To facilitate the consistency of the underlying business process, we generate the appropriate content of the user interface based on a particular workflow (i.e., the specification of a business process). The framework for generating user interfaces is presented in [20], [22]. We automatically generate the following components that are essential for the fulfillment of tasks and coordination.

- **Work Space:** provides a screen space that loads the interfaces from the backend applications and allows users to conduct their daily work. As illustrated in Figure 3, the user interface presented in *Work Space* is directly taken from the UI components of the existing call-center application. The bindings between UI applications and workflows are specified in a configuration file.
- **Process List:** gathers a set of workflows that a CSR participates in. A user can select a process name from the process list. A workflow instance, a run-time instance of a workflow specification, is spawned after the selection.
- **Work List:** A work list displays a set of workflow instances waiting for the user to complete. The user can select a workflow instance from the *Work List* to work on. The *Work Space* will launch the UI components corresponding to the current progressing workflow instance.
- **Navigation:** describes the simplified views of a workflow structure. It is used to indicate the progress of a user, and highlight the task which is currently in progress.

Context-Awareness Support: Context awareness is the ability of a user interface to sense and analyze context from various sources. The user interface then takes different actions based on the current context. From the perspective of an e-commerce application, the availability of UI components is determined by different contexts, such as, business processes involved, resources available to execute a task, and user's information. To improve the efficient use of screen space, UI components are adjusted dynamically based on the underlying business processes to permit the user to focus on the current tasks through relevant menus This allows the user interface to provide proactive assistance which improves the end users' work efficiency [17]. For example, when data is available to display in the *Promotions viewer*, the user interface automatically opens the viewer. Furthermore, when the user switches from one workflow instance to another in the user interface, the reengineered user interface can automatically display the relevant information for the latter workflow instance and hide the information relevant to the former workflow instance.

Navigational Transitions for Fulfilling Business Processes: To avoid memorizing the functionality of UI components, we guide users to access the appropriate user interface components. We utilize the bindings between tasks and applications to divide a workflow into segments. Each segment contains a sequence of tasks that are performed by the same user interface component. We then provide step-by-step task instructions in the user interface to improve the learnability of the user interface [4]. For example as depicted in Figure 2, *Enter Shipping Address* task and *Submit Order* task are presented in the order editor. In this case, once the CSR completes *Enter Shipping Address* task in the first page of the order editor, the *Work Space* component automatically transitions to the second page which allows the user to continue with the order submission.

3. An Approach for Developing Business Process Driven User Interface

The high-level architecture for our business process driven user interface is illustrated in Figure 4. We analyze workflows and source code in order to locate

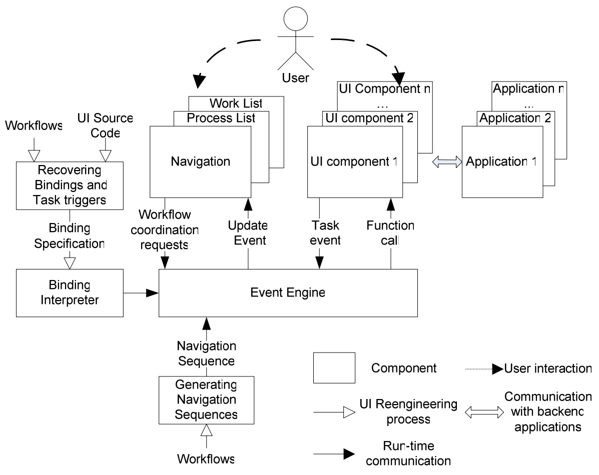

Figure 4: Architecture of the Proposed Business Process Driven User Interface

the code block in existing applications that implement the tasks specified in workflows. The associations between tasks and code blocks are recorded in a binding specification. Moreover, the *navigation sequences* are automatically generated by analyzing workflows. We develop an event mechanism to coordinate *Navigation*, *Process List*, *Work List* and UI components of applications. The core of the event mechanism lies in the *Event Engine*. The *Event Engine* queries the binding interpreter to acquire the binding information at run-time, and uses the binding information to provide navigation transitions and context awareness in the user interface. The *Event Engine* launches applications by issuing *functional calls* to UI components that implement a task. It listens to events (i.e., *task events*) triggered from UI components to determine the executing task in a workflow instance. The *Event Engine* notifies the *Process list*, the *Work List* and the *Navigation* the status of the progressing workflow instances via update events. Moreover, the *Event Engine* is responsible for handling the requests from the *Process List*, the *Work List* and the *Navigation*. Examples of the requests are coordinating workflow instance, starting a new workflow instance and switching between workflow instances.

In a typical scenario, a business user starts a workflow instance by selecting the workflow in the *Process List*. The *Event Engine* queries the binding interpreter to determine the first set of tasks to be performed, and automatically launches the UI component for these tasks. When a user starts working on a task, task events from UI components are received by the *Event Engine* to indicate the active task in a workflow instance. Once a task is complete, the *Event Engine* closes the corresponding UI component and

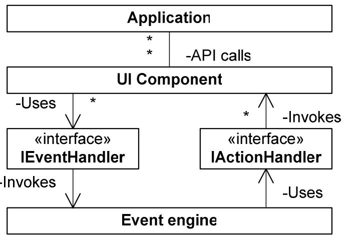

Figure 5: Synchronizing UI Components of Existing Applications with the *Event Engine*

```
public interface IEventHandler {
  // Process a task event
  public void handleTaskEvent(TaskEvent event);
}
public interface IActionHandler {
  // Show an application UI to the user
  public Object OpenUIComp(String AppName,
              TaskContext context);
  // Switch to a previously opened UI component
  public void SwitchToUIComp(Object UICompInstance);
  // Close an application UI
  public void CloseUIComp(Object UICompInstance);
}
```

Figure 6: Definitions of the *IActionHander* and the *IEventHandler* Interface

determines the next UI components to open. In addition, the *Work List* and the *Navigation* update their contents as work progresses.

3.1 Integration of Existing Applications

To minimize the coupling between the business process driven user interface and the UI components from existing applications, we define two interfaces, the *IEventHandler* and *IActionHandler*, to deal with the communication between the *Event Engine* and UI components as shown in Figure 5. The *IEventHandler* is used by the UI components to send task events to the *Event Engine*. The *IActionHandler* is used by the *Event Engine* to display one UI component in the *Work Space* and unload another from the *Work Space* through a set of operations defined in the *IActionHandler*: An *OpenUIComp* operation launches a UI component in the *Work Space* (i.e., a screen space to load the UI components of applications) for a business user to conduct the next task. The *OpenUIComp* operation returns an instance of the UI component (*UICompInstance* in Figure 6), which can be used by subsequent operations. A *CloseUIComp* operation terminates a UI component in the *Work Space* when a task is completed; a *SwitchToUIComp* operation automatically brings a previously-opened UI component to the top of the *Work Space* and highlights the UI component. Moreover, these operations allow the *Event Engine* to dynamically show or hide UI components in the context of progressing of workflow instances.

4. Recovering Bindings between Tasks and Existing Applications

Currently, most of e-commerce applications adapt MVC (Model-View-Controller) architecture [2]. In this architecture, the *model* manages the data and behaviors of the application. The *views* describe the content and layout of the user interface. The controllers are used to gather data from the users, invoke backend services and present the results to the users. This architecture separates user interfaces (i.e., views) from the business logics (i.e., model). In our research, we consider that workflows capture high-level abstraction of execution flows of an application. A task represents the lowest-level of details in a workflow and its functionality is implemented using one or more UI components in an e-commerce application. In this case, the granularity of user interface components matches well with the high level abstraction of a workflow. Therefore, we recover the correspondences between existing UI components and the tasks specified in workflows. Moreover, we insert task event statements in the identified code blocks in order to trigger task events during the execution of a UI component. The detailed steps are described in the following subsections.

4.1 Recovering Bindings

A task in a workflow is described in terms of name, input data, output data and internal attributes (e.g., manual task or automatic task). Since the structure of the user interface code is highly dependent on the user interface platform, we utilize the workflow structures and literal string description of tasks to identify the matching implementations in the source code. By examining the design and implementation of user interfaces of e-commerce applications, we identify four generic matching rules as described below.

(1) **Match the task name with the names of UI components**. This is the case when one or many user interface components are dedicated to perform the task. For example, a *Find Order* task is implemented in the *Find Order Dialog* component in the user interface code. This dialog is used by the business user to find orders. Therefore, the *Find Order* task is bound to the *Find Order Dialog*.

(2) **Match the task name with the name of widgets in UI components**. To improve the efficiency of a UI component, developers tend to group a few tasks in one UI component, and deliver the maximum information in a single UI component page. Therefore, a task can be performed by one or more widgets (e.g., selection options and buttons) in a UI component.

(3) **Match the task name with the names of the function calls to backend services of UI components**. For instance, a *Create Order* task may be bound to an *Order Editor* window, if the *Order Editor* window invokes the function, *CreateOrder* as a backend service.

(4) **Match input and output data of the task with the data required or produced by a UI component**. The user interface of an e-commerce application provides more than just UI components to assist user in performing tasks. Often, supporting UI components, such as image viewers, and product promotion reminders are integrated into the user interface. These supporting UI components do not directly correspond to tasks in workflows. Typically, such supporting UI components are useful in specific contexts. For example, when a customer is purchasing a product, a promotion viewer can prompt discount information for the CSR once the product is selected. In other times, the content of the promotion viewer is unavailable. To improve the efficient use of screen space, we strive to display the supporting UI components based on the availability of the content. We analyze the data dependencies between the supporting UI components and the UI components that implement a task. In particular, the data input/output of a task is served as a starting point to locate the supporting UI components. An input data or output data is associated a supporting UI component that has the data dependencies with the UI component that implements the task. As illustrated in Figure 7, when a data is generated from one UI component and passed into the supporting UI component, we launch the supporting UI component.

A tool can parse the user interface of the application, and identify candidate UI components to match with tasks. The matching could be done using naming conventions and other heuristics. Developers can then verify the matchings. We define a binding specification that captures the task implementation in UI components. We use the *Purchase Order* process shown in Figure 2 as an example to describe the format of our binding specification. As shown in Figure 8, a *ProcessBinding* element specifies the name of a workflow. For each task in a workflow, a *TaskBinding*

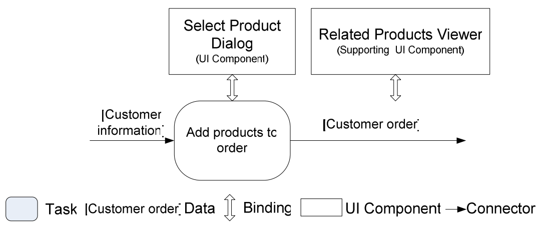

Figure 7: Supporting UI Component bound to Data

```
<Configuration>
<ProcessBinding ProcessName="Purchase Order">
 <TaskBinding  TaskName="Enter customer information"
            Optional="false">
     <UIComponent
             Name="UI.Editor.CustomerEditor" />
 </TaskBinding>
 <TaskBinding TaskName="Add products to order"
           Optional="false">
    <UIComponent Name=" UI.Dialog.SelectProductDialog"
       /UIComponent>
 </TaskBinding>
 <TaskBinding TaskName="Enter payment information"
            Optional="false">
    <UIComponent Name=" UI.Dialog.PaymentDialog" />
 </TaskBinding>
 <TaskBinding TaskName="Enter shipping address"
             Optional="false">
  <UIComponent Name="UI.Editor.OrderEditor" />
 </TaskBinding>
 <TaskBinding TaskName="Submit order" Optional="false">
    <UIComponent Name="UI.Editor.OrderEditor" />
  </TaskBinding>
 </ProcessBinding>
</Configuration>
```

Figure 8: Example Binding Specification for the Purchase Order Process

element is constructed to specify the associated UI Components, known as, *UIComponent* tag. The *Name* of the *UIComponent* refers to the class name that implements the UI Component. The template of such a configuration is automatically generated from the source code and workflows.

4.2 Inserting Task Event Triggers

Once we identify the binding between a task and UI components, we determine the location in the source code where a task event can be triggered when a UI component is executed. A task event is used to indicate the current active task in a workflow instance. Every task is linked with a unique event which is generated directly from the unique task name specified in workflows. Furthermore, each event has three types, including "start", "end" and "cancel". A start task event is triggered when a UI component is initiated to execute a task. Similarly, when a UI component is closed, an end task event is triggered. The cancel task event is triggered when a UI component is cancelled.

We adapt three heuristics rule to identify the location to insert task event triggers:

(1) **The start task event trigger is inserted where a user issues a menu action, such as pressing a button, selecting a menu item and choosing an option in a combo box in a UI component.** The position to place a start task event trigger is dependent on the granularity of a UI component that implements the task. We consider the following two cases. In the case that a task is implemented by an entire UI component, the start task event trigger is located at the point where the UI component is about to prompt for display. For example, the *Find Customer* task starts when a *Find Customer Dialog* is popped. In the case that a task is carried out by a widget, the start task event is located at the point where the widget is performed. For instance, the *Submit order* task is launched using a "Submit" button in an *Order Editor Window*.

(2) **The end task event trigger and cancel task event trigger are usually located where a data as an output of a task has been derived.** A task may produce output data when a task is completed. In this case, we examine the location of the output data from a task and place either end task event trigger or cancel task event trigger based on the availability of the data. For example, if a *Find Customer* task is performed in a *Find Customer Dialog*, and the task ends when customer information is retrieved in the *Find Customer Dialog*. Otherwise, the *Find Customer* task is cancelled if no customer information is returned.

(3) **No triggers for the tasks that require no user interaction**: Users always prefer to click fewer buttons to improve their work efficiency. In this case, the developers may choose to provide default values or settings for UI widgets, such as combo boxes and text boxes. In this context, a business user only needs to verify if the default value is appropriate, and may or may not need to perform any actions in the user interface. We call this type of tasks *optional* tasks. The remaining tasks, which require user's actions, are called *mandatory* tasks. Typically, *optional* tasks can be detected by determining the default value of widgets when a UI component is initialized. At runtime, the event engine determines the completion of an optional task by receiving the start task event trigger of the task subsequent to an optional task.

Once the locations of *task event triggers* are identified, a wizard tool can be developed to guide developers to automatically insert statements that trig-

```
public class TaskEvent {
  // Event name
  public String Name;
  // Event type, such as Start, End, etc.
  public String Type;
  // Workflow related data for each task
  TaskContext context;
}
```

Figure 9: Task Event definition

```
//In UI component source code, at trigger point
TaskEvent event = new TaskEvent(
    "Enter customer information", "Start", null);
EventHandler.getInstance().handleTaskEvent(event);
```

Figure 10: Triggering a Task Event in the Source Code of a UI component

ger task events. A task event has three attributes, including name, type, and context, as shown in Figure 9. The *TaskName* is the name of a task to which the event belongs. The *Type* determines the type of the task event trigger. The *TaskContext* holds a set of data related to a workflow instance which can be passed to the receivers of the event. Examples of data are the parameters to the *Event Engine*, and workflow instance identifiers. Figure 10 gives an example to demonstrate how a task event is sent to the *Event Engine*. A start task event trigger for *Enter Customer Information* task is sent to an *EventHandler*, which is a reference variable to the *Event Engine*.

5. Developing Navigation Sequences

To improve the performance of a user interacting with the user interface, we generate a navigation sequence that guides a user to work on a sequence of UI components in order to accomplish a workflow. Moreover, the reengineered user interface also provides context-aware assistances to handle multiple workflow instances.

5.1 Generating Navigation Sequences from Workflows

To illustrate task steps that a user needs to perform in a workflow, we interpret the structure of the workflow and display it in the *Navigation* viewer. However, a workflow normally contains detailed information related to tasks. A business user will be overwhelmed by the large number of tasks and will get distracted by multiple execution paths because of the control flows (e.g., decisions and loops) specified in the workflow. To reduce the complexity of the navigation sequence, we present only *mandatory* tasks as an initial navigation sequence, and display an optional task when its related *mandatory* task is comp-

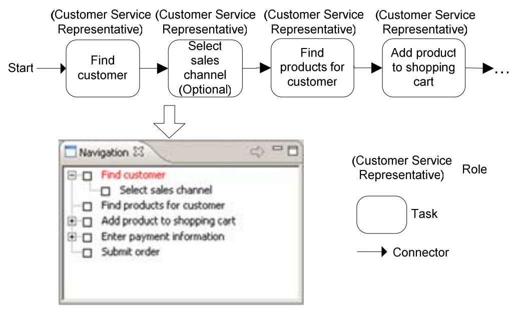
(Customer Service Representative)
(Customer Service Representative)
(Customer Service Representative)
(Customer Service Representative)
Start
Find customer
Select sales channel (Optional)
Find products for customer
Add product to shopping cart
…
Navigation
Find customer
Select sales channel
Find products for customer
Add product to shopping cart
Enter payment information
Submit order
(Customer Service Representative)
Role
Task
Connector

Figure 11: Handling Optional Tasks in the Navigation Viewer

leted. For the example business process shown in Figure 11, the *Select Payment Method* task is an *optional* task since it performed in a combo box with a default value specified. Therefore, we display *Select Payment Method* at a sub level of *Create Customer Order* task. When *Create Customer Order* task is complete, the user can either perform the *Select Payment Method* task to make a new selection other than the default value, or proceed to the subsequent task (i.e., *Enter Shipping Information* task). This allows business users to focus on *mandatory* tasks while gives them chances to conduct *optional* tasks as needed.

To reduce the complexity of displaying possible navigation paths, we include only one navigation path at a time. For a decision in a workflow that produces more than one path, *the Navigation* viewer dynamic determines the possible navigation path from the user's selection and updates the viewer to attach the new navigation sequence. The parallelism in workflows simply implies the independence between the business operation paths. The tasks in different parallel path can be performed in any order. In this case, we treat tasks in parallel as sequential tasks. These simplification techniques give the business users a better understanding of the overall workflows.

5.2 Providing Context-Aware Guidance

A context is composed of a workflow instance, an active task, the input/output data of the task, and the related UI components. As a business user progresses through the user interface, contextual information are gradually collected from the task events received from the UI components, the navigation sequence, and the binding specification. The navigation sequence indicates the active tasks to proceed. The task event indicates the status of the current active task. By examining the binding specification, the *Event Engine* can dynamically determine which UI components need to be displayed as well as the supporting UI component for presenting the data required or produced by a task.

To guide business users to "walk through" the navigation sequence, the user interface automatically displays a UI component when a task needs to be performed. Once a UI component is no longer used for subsequent tasks, the UI component is automatically closed. However, in some occasions when a task ends, a business user may want to verify the result displayed in a UI component before proceeding to the next UI component. Automatically opening a UI component may be intrusive to the user's perception of the user interface. Therefore, it is desirable to grant users some control over the navigation, instead of forcing business users to adapt to the behavior of automatically opening and closing of UI components by the *Event Engine*.

We have implemented a "next" button to allow user to control the pace of his/her own navigation. After pressed by the user, the "next" button can automatically close the current UI component and open the next UI component which implements the subsequent tasks. By default, the "next" button is disabled. The "next" button is enabled when a task in the current UI component generates data that requires the user's verification. In the cases that the current complete task leads to multiple tasks performed in parallel, the "next" button is enabled and users can bring up the UI component which implements one of the parallel tasks. Since the navigation view always highlights the current tasks to be performed, the business user can easily navigate to the other concurrent tasks by selecting them in the *Navigation* viewer.

6. Case Studies

A case study has been performed to verify the effectiveness of our approach for generating business process driven user interface. We have reengineered a commercial call center software to provide contextual and navigational support for this software.

6.1 Setup

The call center software has 100K lines of code, and is implemented on Eclipse Rich Client Platform (RCP) [5] which includes its workbench, views, editors and menu items. This software provides full functionality for a sales representative to interact with customers and sell products over the phone conversation. However, from our observation, this software has limitations in terms of usability of the user interface design and consistency with its underlying business processes. Especially, the customer service representatives of this call center software have limited computer experience. In particular, they do not stay long in their job and frequently change their jobs. Therefore, most of customer service representatives are novice users. In this case, the usability of the user interface becomes critical issue in using this software. Moreover, the workflows that describe the activities in a call center are modeled independently from the software development.

Table 1: Sizes of the Major Components

Component	RCP (LOC)
Workflow Parsers for IBM WBI Workbench	1658
Role Model Generator	540
RCP UI Model Generator	1212
Navigation Sequence Generator	132
Source Code Generator	1020
Generated Code for Event Engine	1523
Generated Code for Process list	173
Generated Code for Work List	395
Generated Code for Navigation	321

6.2 Implementation

The workflows provided by a business solution architect for our case studies are modeled using IBM WebSphere Business Integration (WBI) Modeller in Eclipse environment. In IBM WBI Modeller, the workflows are stored using the schema of XMI and EMF (Eclipse Metamodel Framework). Moreover, workflows described in various specification languages can be imported into IBM WBI Modeller. We developed a parser to analyze business processes stored in the schema of XMI and EMF and extract information required by the business process user interface.

In the reengineered user interface, we inherit the same user interface presentation style as the original call center software using Eclipse RCP. We automatically generate our prototype business process driven user interface with the major components including 1) *Workflow parsers* which parses workflows modeled by IBM WBI Modeller, 2) *Role model generator* which extracts the tasks relevant only to one role (i.e., CSR) from workflows, 3) RCP UI *model generator* which transforms the information in role models to contents needed for the presentation and layout in the RCP user interface, 4) *Navigation configuration generator* which produces the navigation sequence of a possible processing order for each workflow, 5) *Code generator* which generates source code for the *Event Engine*, the *Process List*, the *Work List*, and the *Navigation* in RCP platforms. Moreover, we also generate bindings between tasks and UI components. Our prototype user interface is integrated with the call center application as an Eclipse plug-in. The screenshot of the prototype system is illustrated in

Figure 3. The sizes of the major components of the generated user interface are illustrated in Table 1.

Within the original call center sales application, as shown in Figure 2, we have identified bindings using the heuristic rules discussed in Section 4.1. The heuristic rules use the naming convention and are applied in the level of UI components. We have observed that the task names in workflows and names for the UI components are intended to be easy to understand for the users. Therefore, naming differences in tasks and their corresponding UI components are low. In other cases, the call center software is developed without reference to workflows. Some of tasks specified in workflows are not implemented in the software. In this case, we remove the not implemented tasks from the workflows in order to generate a precise navigation sequence. To integrate with the existing application, we simply insert task event triggers to the UI components that are bound to workflow tasks with no modification to their existing code. We have added approximately 30 task events in total. These task events are handled by the *Event Engine* within our plug-in project and the corresponding actions are performed from the UI components that are bound to the task event through the operations defined in the two interfaces, including *IEventHandler* and *IactionHandler*.

6.3 Evaluation

The reengineered user interface is evaluated by the user interface design experts and the developers of the call center software. We have received positive feedback from them. For example, the *Navigation* lists only *mandatory* tasks and can dynamically adapt to a user's selection. *Navigation* reduces the complexity of presenting actual workflows to the user and gives users an overview of the steps to perform. The "Next" button replaces multiple mouse clicks for searching for the subsequent UI components with one simple click. This reduces the business users' think time and improves work efficiency. The context-awareness is considered to be desirable, especially for supporting UI components that are mixed with other UI components or which require users to manually initiate. As business processes evolve over time, the underlying implementation, such as the user interface of e-commerce systems, also need to be changed. In our approach, the content of the business process driven user interface is automatically regenerated from the updated business processes.

We also evaluate the limitations of our approach. First, a workflow often specifies only one possible way to carry out business activities. As a result, it often imposes a strict order on how business users should carry out their activities. However, developers tend to offer many alternative ways to perform tasks. For example, a CSR can add multiple products in *Add Product Editor*. However, the corresponding *Add Product task* is non repeatable as specified in the workflow. Experienced users may not like to follow a strict process to conduct their work. Nevertheless, this improves the productivity of novice users and increases the accuracy for both novice and experienced user especially when business processes evolve. Furthermore, the UI is updated automatically when the workflow is updated. This in turn reduces the time needed to train users for workflow changes.

Another limitation of our approach is that one UI component may be used to perform multiple workflows. When a business user is working in one UI component, there is a chance that he/she may perform tasks specified in other workflow instances, resulting in deviation from the current workflow instance.

Our current implementation is that the process of identifying bindings and inserting task event triggers is done in a semi-automatic fashion. We use the search functionality in the Eclipse IDE, since the task names and UI components tend to follow a similar naming convention. In future work, we plan to leverage our workflow recovery techniques [21] to automate this process.

7. Related Work

Many research on user interface reengineering focuses on migrating user interfaces, either from text-based platforms to GUI based platform [16][18], or from one GUI platform to another [14]. The authors of [1] points out that reengineering existing user interfaces improves the performance and user satisfaction, while shortening training and reducing error rates. Typically, a user interface reengineering process starts by reverse engineering the user interface components using static [11] and dynamic analysis [14]. In our context, we used a heuristic-based approach similar to [11] to identify binding and task event triggers. [8] presents methods to establish dependencies by matching task names, input and output between workflows and the workflows that extracted from source code. In our work we examine the association between tasks and UI components.

Driven by the need to support workflows in web-based systems, web application development approaches, such as OOHDM [6], UWA [10] and WebML [1], have been extended to support process models and incorporate workflows in the process of designing web applications [9],. These approaches are applied to design navigational structure among web pages. However, such approaches have not been applied to GUI based applications. In [13] the authors utilize the bindings between activities and user interface components to develop a user-guidance environment for software toolsets. However, the developers need to manually specify the binding and processes which fit the behaviors of the toolsets. Our approach automatically generates navigation sequences directly from workflows. In addition, we are able to handle multiple workflow instances (i.e., user scenarios) and can provide context-aware assistance.

8. Conclusion

In this paper, we present an approach to reengineer the user interface of existing e-commerce applications, in order to improve the usability of an existing user interface. We provide navigational guidance to facilitate business users accomplishing business activities in user interfaces. To permit business users to focus on their current activities, our approach provides context-aware assistance by displaying related UI components in the context of the current user's activities in the user interface. In the future, we plan to apply our framework to more e-commerce applications, to conduct more usability evaluations and to receive feedback from novice business users. We also plan to study the effectiveness of our approach in the context of business process evolution and software evolution.

Acknowledgement

This research is sponsored by IBM Canada, Centers for Advanced Studies (CAS), and National Sciences and Engineering Research Council (NSERC).

We would like to thank Jen Hawkins, Bhadri Madapusi, Ross McKegney, and Tack Tong at the IBM Canada Toronto Laboratory for their valuable feedback on this work.

References

[1] M. Brambilla, S. Ceri, S. Comai, P. Fraternali, and I. Manolescu, "Specification and Design of Workflow-Driven Hypertexts". *Journal of Web Engineering*, 2003.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stahl, A System of Patterns, Wiley, New York, 1998

[3] M. Costabile, "Usability in the Software Life Cycle", *Handbook of Software Engineering and Knowledge Eng.*, Volume 1, World Scientific Publishing, 2001.

[4] J. Tidwell. *Designing Interfaces*. O'Reilly Media, 2005.

[5] Eclipse Rich Client Platform. www.eclipse.org/rcp

[6] N. Guell, D. Schwabe and P. Vilain, "Modeling Interactions and Navigation in Web Applications", *In Proceedings of the World Wide Web and Conceptual Modeling Conference*, Springer, 2000.

[7] ISO 9241: Ergonomics Requirements for Office Work with Visual Display Terminal (VDT) – parts 1-17, 1997

[8] I. Ivkovic and K. Kontogiannis, "Using Formal Concept Analysis to Establish Model Dependencies", *In Proc. of the Intern. Conf. on Information Technology*, 2005.

[9] N. Koch, A. Kraus, C. Cachero, and S. Melia, "Modeling Web Business Processes with OO-H and UWE". *In Proceedings of the International. Worskhop on Web-oriented Software Technology*, July 2003.

[10] G. Kappel, B. Proll, W. Retschitzegger, W. Schwinger, and T. Hofer. Modeling ubiquitous web applications - a comparison of approaches. *In Proceedings of the International Conference on Information Integration and Web-based Applications and Services*, 2001.

[11] E. Merlo, J. Girard, K. Kontogiannis, P. Panangaden, R. De Mori, "Reverse engineering of user interfaces", *Working Conference on Reverse Engineering*, 1993.

[12] C. Plaisant, A. Rose and B. Sheideman, "Low Effort, High Payoff User Interface Reengineering", *IEEE Software*, July 1997.

[13] T. Sliski, M. Billmers, L. Clarke, and L. Osterweil, "An Architecture for Flexible, Evolvable Process-Driven User Guidance Environments". *In Proceedings of ESEC/FSE 2001*, Vienna Austria, September 2001.

[14] E. Stroulia, M. El-Ramly, P. Iglinski, P. Sorenson: "User Interface Reverse Engineering in Support of Interface Migration to the Web", *Automated Software Engineering Journal*, 10(3) 271–301, 2003.

[15] E. Stroulia, M. El-Ramly and P. Sorenson, "From Legacy to Web through Interaction Modeling", *In Proc. of International Conf. on Software Maintenance*, 2002.

[16] K. Tucker and K. Stirewalt,. "Model based user-interface reengineering" *in Proceedings of the 6th Working Conference on Reverse Engineering*, 1999.

[17] "User Interface Architecture", Technical Report, IBM Corporation, available at http://www-06.ibm.com/ibm-/easy/eou_ext.nsf/publish/1392/$File/IBM_UIA.pdf

[18] A. Vanniamparampil, B. Shneiderman, C. Plaisant & A. Rose: "User interface reengineering: A diagnostic approach", Technical Report CS-TR-767 University of Maryland, Department of Computer Science, 1995.

[19] M. Vering, G. Norris, P. Barth, J. R. Hurley, B. Mackay, D. J. Duray, *The E-Business Workplace*. John Wiley & Sons, Inc., June 2001.

[20] Q. Zhang, Y. Zou, T. Tong, R. MacKegney and J. Hawkins, "Automated Workplace Design and Reconfiguration for Evolving Business Processes", *In Proc. of Centre for Advanced Studies Conference*, Nov. 2005.

[21] Y. Zou, M. Hung, "An Approach for Extracting Workflows from E-Commerce Applications", *In Proc. of the 14th IEEE International Conference on Program Comprehension*, June, 2006.

[22] Y. Zou and Q. Zhang, "A Framework for Automatic Generation of Evolvable E-Commerce Workplaces Using Business Processes", *in the Proc. of 28th International Conference on Software Engineering*, May 2006.