Recovering Business Processes from Business Applications

Ying Zou[1], Jin Guo[2], King Chun Foo[1], Maokeng Hung[3]

Dept. of Electrical and Computer Engineering[1]
Queen's University
Kingston, ON, K7L 3N6, Canada
{ying.zou, 3kcdf}@queensu.ca,

School of Computing[2]
Queen's University
Kingston, ON, K7L 3N6, Canada
guojin@cs.queensu.ca

ASUS Tek Computer Inc.[3]
Yonghe City, Taipei County 234
Taiwan (R.O.C.)
alex1_hung@asus.com.tw

ABSTRACT

A business process, such as the process followed when ordering a book, describes the order of executing tasks (e.g., check inventory, verify credit card, and ship book). Business applications implement the business processes for the daily operations of an organization. Organizations must continuously modify their business applications to accommodate changes to business processes. However, business applications are often designed and developed without referring to the documented definitions of business processes. Modifying business applications is a time-consuming and error-prone task. To correctly perform this task, developers require an in-depth understanding of multi-tiered applications and the definitions of the business processes which they implement. In this paper, we present an approach which automatically recovers business process definitions from multi-tiered business applications. Given the starting UI screen of a particular business process, the approach recovers the process definition by tracing the flow of control throughout the different tiers of the business application. We demonstrate the effectiveness of our approach through a case study using 15 business applications from three large open-source projects. Our case study demonstrates that our approach can recover business process definitions from the implementation with high precision and recall.

Keywords: Program Comprehension, Business Process Recovery, Business Application, Static Analysis

1. Introduction

Organizations use business applications to automate their daily business processes. Creating the content of an organization's website and making an on-line purchase are two example processes. A business process is composed of a set of interrelated tasks which are joined together by control flow constructs. The control flow constructs specify the order (e.g., sequential, parallel, or alternative) of the execution of tasks. For example, the business process for purchasing a product on-line consists of the following sequence of tasks such as "Select product", "Add to the shopping cart", and "Validate buyer's credit card".

To accommodate rapid changes to business requirements, organizations must modify their business processes and the implementation of these processes (i.e., the supporting business applications). All too often, there are no explicit links between the tasks specified in process definitions, and the implementation of these tasks in the business applications. The lack of links between business requirements and implementation leads to difficulties in identifying the appropriate code segments which must be modified in response to changing business requirements. A change, such as adding a new task to an existing business process, or merging two tasks into one can be labor-intensive and requires an in-depth understanding of business applications and their corresponding business processes. Business applications, which are significant investments for most organizations, risk being abandoned, if such systems do not adapt to rapidly changing business requirements [30].

Systematic methods and tools are needed for the development and evolution of business applications. Most current research concentrates on managing software evolution from aspects related to the code itself, such as program understanding, change impact analysis, and co-evolution of software design and the code [59]. A few approaches propose to automatically identify business tasks from source code [21][25][38][41][44][47]. However, the control flow constructs (e.g., sequence, alternative and iteration) that connect business tasks are often neglected in existing approaches. Therefore, the recovered business processes from the implementation are incomplete. In previous work, we applied static analysis techniques on the Java source code to automatically recover implemented process definitions [26][60]. Our prior work was limited to the process definitions implemented within a single source code file. The functionality of a large business application is distributed across several files or components. Simply analyzing the source code of a particular component is not sufficient.

In this paper, we present an approach which recovers business process definitions from their three-tier implementation. The three-tiers (User Interface (UI), business logic, and databases) react to users' requests and generate results according to UI selections, business rules and status data stored in databases. Starting from the initial screen for an application implementing a particular business process, we use static analysis to recover the navigation flow of the UI. We analyze the navigation flows and the interactions between UI and business logic tiers to identify business tasks and their dependencies on data and controls. We represent the recovered tasks and the data/control dependencies as a business process. We also examine the source code of the different back-end components which reside in the business logic tier and which implement the functionality in the UI screens. We represent the process recovered from a back-end component as a sub-process which describes the reusable sub sets of tasks in a business process. Finally we use the glue code (i.e., controller scripts in the business logic tier) to integrate the sub-processes recovered from the business logic tier with processes recovered from the UI navigation flows to form a complete business process.

The recovered processes are visualized in a Business Process Explorer (BPE) tool which we developed. The BPE tool links process entities (i.e., tasks and control entities) with their corresponding software entities (i.e., code blocks and control statements). Our approach and its associated BPE tool ensure the consistency of business processes and their implementation by maintaining an up-to-date view of the relation between process entities and software entities. The BPE tool integrates into the Eclipse software development environment to support developers when evolving a business application. Moreover, the BPE tool integrates into the IBM WebSphere Business Modeler (WBM) [56], a leading commercial business process modeling tool. A business analyst can visualize, analyze and enhance the recovered processes by leveraging the capability of a familiar platform (i.e., WBM). This paper extends earlier work published in the International Conference on Program Comprehension (ICPC) 2007 [26]. We enhance the earlier publication in ICPC 2007 in the following aspects:

1) We recognize the design patterns used in the UI screens to recover tasks and control flows from the navigation flows.
2) We use the data flow analysis to identify tasks and control flows from the business logic tier.
3) We apply our approach on another two large open source projects and validate the benefits of our approaches through a detailed case study.
4) A prototype tool is designed and developed to show links between the recovered processes and the corresponding source code.

The rest of the paper is organized as follows. Section 2 presents a motivating example to cover the needed background for the paper and motivate our work. Section 3 gives an overview of our approach. Section 4 discusses the steps for recovering business process definitions from UI screens. Section 5 presents the techniques used to recover sub-process definitions from back-end components. Section 6 presents a case study to demonstrate the effectiveness of our approach. Section 7 reviews the related work. Finally, Section 8 concludes the paper.

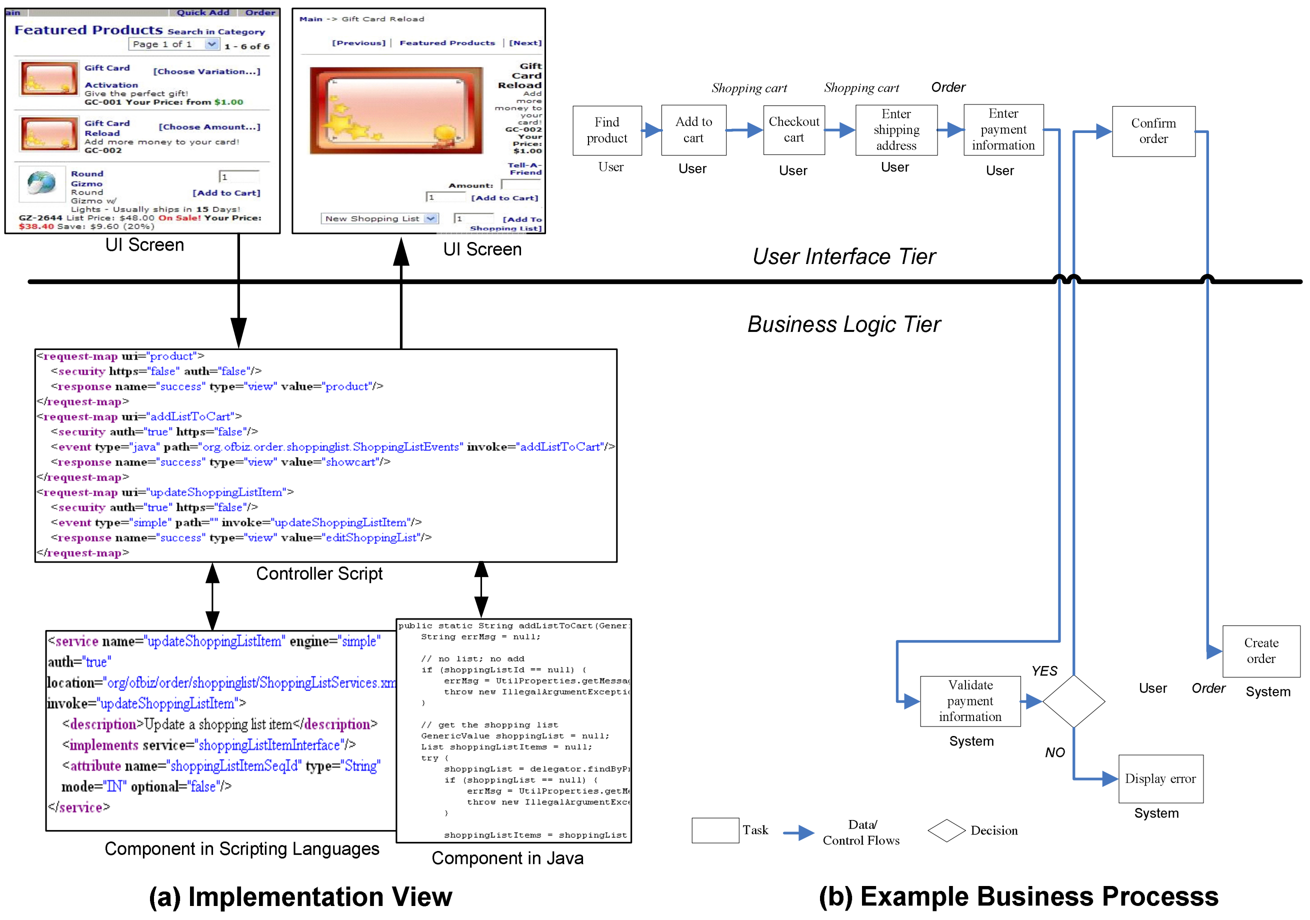
2. A Motivating Example

Figure 1(a) shows an example implementation of a three-tier business application. In the UI tier, a user interacts with a business application by filling and submitting requests using a web browser. In the business logic tier, a controller captures user requests and dispatches them to the appropriate back-end components for processing. A controller also determines the flow of execution by either invoking another back-end component to execute a task or requesting additional user input by loading another UI screen. While processing a user request, a back-end component may communicate with the database tier to retrieve or update data from the database.

To motivate our work, we use as a running example a simplified business process for the "Product purchase" process from the OFBIZ open source project [51]. The simplified process definition for the product purchase process is shown in Figure 1(b). The process consists of several tasks, such as "Find product", "Add to cart", and "Checkout cart". Each task in the process can be implemented using different technologies and programming languages. A process can be composed of human and automatic tasks. As shown in Figure 1(b), human tasks, such as "Find product", and "Add to cart" are manually accomplished by users and require interaction with the web-based screens in the UI tier. Automatic tasks, such as

"Validate payment information" and "Display errors", are automatically invoked by a controller and performed by back-end components in the business logic tier.

An up-to-date view of the definition of a business process, such as the one shown in Figure 1(b), rarely exists in practice. The process definitions are usually not updated after the initial implementation of a business application. The lack of up-to-date documentation hinders the understanding of business analysts who may want to modify a business process by adding an "Inventory check" task to prevent users from purchasing an out-of-stock product. Moreover, a developer working on the application requires an up-to-date view of the definition of a business process and the links between process entities and software entities. Using this knowledge, a developer can easily determine the appropriate parts of the code to change when asked to add the "Inventory check" task or to modify the "Payment information" task to permit other forms of payment (e.g., to permit Paypal payments instead of only credit cards payments). However, the structure of



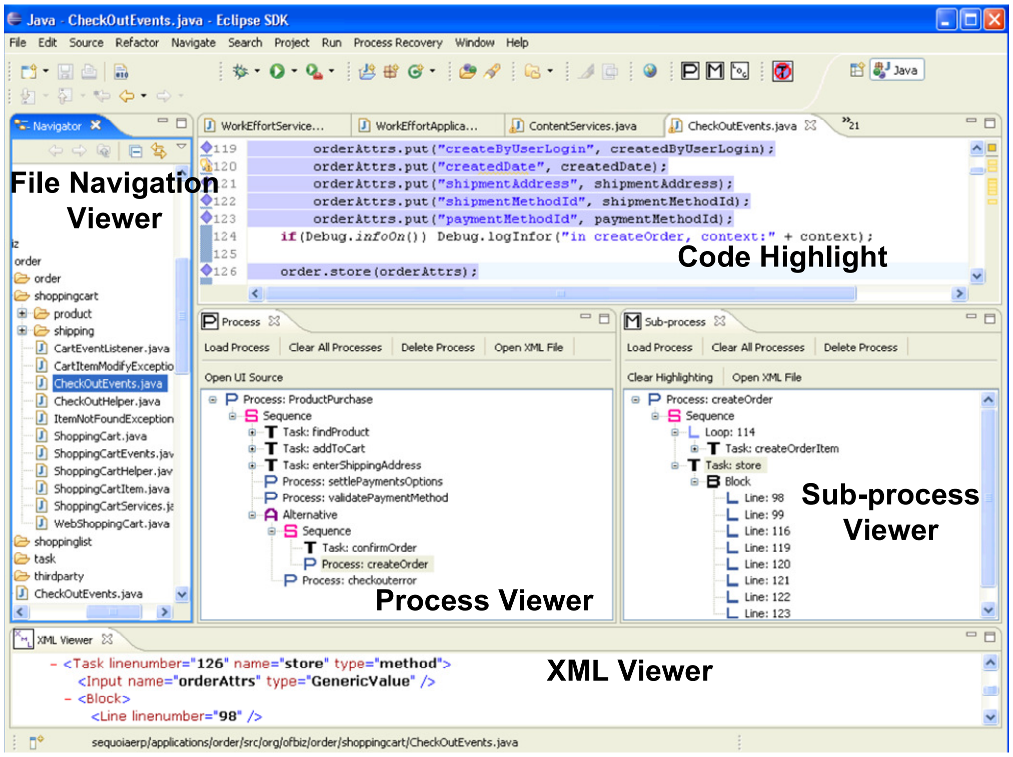
```
<request-map uri="product">
   <security https="false" auth="false"/>
   <response name="success" type="view" value="product"/>
</request-map>
<request-map uri="addListToCart">
   <security auth="true" https="false"/>
   <event type="java" path="org.ofbiz.order.shoppinglist.ShoppingListEvents" invoke="addListToCart"/>
   <response name="success" type="view" value="showcart"/>
</request-map>
<request-map uri="updateShoppingListItem">
   <security auth="true" https="false"/>
   <event type="simple" path="" invoke="updateShoppingListItem"/>
   <response name="success" type="view" value="editShoppingList"/>
</request-map>
```

```
<service name="updateShoppingListItem" engine="simple"
auth="true"
location="org/ofbiz/order/shoppinglist/ShoppingListServices.xm
invoke="updateShoppingListItem">
   <description>Update a shopping list item</description>
   <implements service="shoppingListItemInterface"/>
   <attribute name="shoppingListItemSeqId" type="String"
   mode="IN" optional="false"/>
</service>
```

```
public static String addListToCart(Gener
    String errMsg = null;

    // no list; no add
    if (shoppingListId == null) {
        errMsg = UtilProperties.getMessa
        throw new IllegalArgumentExcepti
    }

    // get the shopping list
    GenericValue shoppingList = null;
    List shoppingListItems = null;
    try {
        shoppingList = delegator.findByP
        if (shoppingList == null) {
            errMsg = UtilProperties.getM
            throw new IllegalArgumentExc
        }

        shoppingListItems = shoppingList
```

Figure 1: An Example Implementation of a Business Application

multi-tiered applications is complex. Without the support of automatic tools, it is a time-consuming process for a developer to understand the code and locate the portion of the code in order to make changes. Looking at Figure 1(a), we observe that each screen contains multiple buttons and links to other screens, and may deliver more than one task. The navigation flow between screens is specified in controller scripts. As shown in the Figure 1(a), once a user selects a product in the screen, a request is sent to the controller. The controller dispatches the in-coming request to an appropriate back-end component which can provide the response to the request. As specified in the controller scripts, the target component for handling the request is defined using a <response> element in the controller script. A response can be mapped to another screen (if the type attribute of the response is "view"), Java components (if the type attribute is "Java")



```
        orderAttrs.put("createByUserLogin", createdByUserLogin);
        orderAttrs.put("createdDate", createdDate);
        orderAttrs.put("shipmentAddress", shipmentAddress);
        orderAttrs.put("shipmentMethodId", shipmentMethodId);
        orderAttrs.put("paymentMethodId", paymentMethodId);
    if(Debug.infoOn()) Debug.logInfor("in createOrder, context:" + context);

    order.store(orderAttrs);
```

```
order
  order
  shoppingcart
    product
    shipping
    CartEventListener.java
    CartItemModifyExceptio
    CheckOutEvents.java
    CheckOutHelper.java
    ItemNotFoundException
    ShoppingCart.java
    ShoppingCartEvents.jav
    ShoppingCartHelper.jav
    ShoppingCartItem.java
    ShoppingCartServices.j
    WebShoppingCart.java
  shoppinglist
  task
  thirdparty
  CheckOutEvents.java
```

```
Process: ProductPurchase
  Sequence
    Task: findProduct
    Task: addToCart
    Task: enterShippingAddress
    Process: settlePaymentsOptions
    Process: validatePaymentMethod
    Alternative
      Sequence
        Task: confirmOrder
        Process: createOrder
      Process: checkouterror
```

```
Process: createOrder
  Sequence
    Loop: 114
      Task: createOrderItem
    Task: store
      Block
        Line: 98
        Line: 99
        Line: 116
        Line: 119
        Line: 120
        Line: 121
        Line: 122
        Line: 123
```

```
- <Task linenumber="126" name="store" type="method">
    <Input name="orderAttrs" type="GenericValue" />
  - <Block>
      <Line linenumber="98" />
```

```
sequoiaerp/applications/order/src/org/ofbiz/order/shoppingcart/CheckOutEvents.java
```

Figure 2: An Annotated Screenshot of the Business Process Explorer

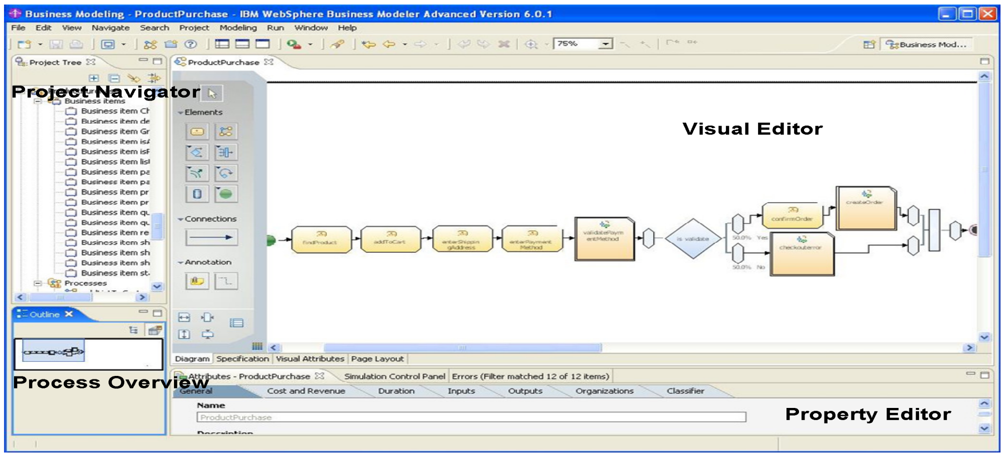


Figure 3: Visualizing Recovered Processes in IBM WBM

or scripting languages (if the type attribute is "simple"). The value of the response element specifies the location of the target component for handling the request.

To support the evolution of business applications, we have designed and developed an approach to automatically recover business processes from the UI and business logic tiers of a business application and to establish the links between the recovered processes and their implementation. If we solely study the UI code, we can only understand the human tasks, which require input from users, and miss the automatic tasks which are directly performed by back-end components. Therefore, we must trace the navigation flow between screens and back-end components. We must examine the controller scripts, which integrate various back-end components, to understand the execution order of the back-end components and the execution constraints. Moreover, we need to analyze the source code of the back-end components which provide the functionality required by these screens. The recovered processes are visualized in a Business Process Explorer (BPE) tool.

An annotated screenshot of the BPE tool inside Eclipse is shown in Figure 2. The code of the business application is imported into the BPE tool and listed in the Eclipse file navigation viewer. To allow developers to quickly identify the code corresponding to a business task, the BPE tool tracks the lines of code relating to each recovered task and control construct. As shown in the sub-process viewer of Figure 2, the tasks for "Create order" are listed. The block icon in the tree structure presents the lines of the code that implement the task "Store". When a user double clicks on the task node in the sub-process viewer, the lines of code corresponding to the clicked task are automatically loaded and highlighted in the Eclipse text editor.

To ease the adoption of the recovered processes by practitioners, we integrate the PBE tool into the IBM WebSphere Business Modeler (WBM). Figure 3 shows the recovered on-line purchase process after it is imported into WBM. A developer can use the modeling tool to view and understand the overall functionality of the business application. A business analyst can navigate through the recovered processes and assign more meaningful names to the recovered tasks, since our approach automatically assigns tasks generated names. A business analyst can leverage WBM to analyze the recovered processes. For example, WBM provides a simulation tool which allows the business analyst to simulate the execution of a process to locate possible design problems. Consequently, the process can be optimized to improve the performance of the organization which owns the business applications.

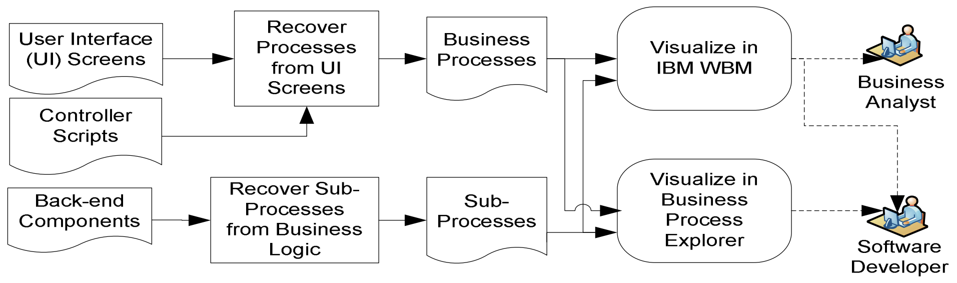

User Interface (UI) Screens
Controller Scripts
Back-end Components
Recover Processes from UI Screens
Recover Sub-Processes from Business Logic
Business Processes
Sub-Processes
Visualize in IBM WBM
Visualize in Business Process Explorer
Business Analyst
Software Developer

Figure 4: Steps for Recovering Business Processes from Business Applications

3. An Approach for Recovering Business Processes

It is challenging to recover the complete definition of a business process from its implementation due to the intricate control and data dependencies among the intermixed human and automatic tasks. Figure 4 gives an overview of our approach for recovering process definitions. To identify possible execution flows among UI and business logic tiers of business applications, we analyze the execution paths of controller scripts to understand the structure of a business application. By analyzing the controller scripts, we locate the starting point for each business process and retrieve the screens and the back-end components which are invoked from these screens. We analyze the screens and controller scripts to recover business processes. To recover sub-processes implemented in back-end components, we parse the source code of the back-end components in the business logic tier to extract the automatic tasks and their dependencies. Database accesses are captured in the business logic tier. The recovered business processes are viewed using the BPE tool in Eclipse and IBM WBM. In the following sections, we detail the steps followed by our approach for recovering the complete processes from the UI screens (Section 4) and sub-processes for complex automatic tasks (Section 5). But first we discuss the content of process definitions which our approach recovers.

3.1 Representing Recovered Business Processes

Business process definitions are specified using various specification languages, such as BPEL4WS (Business Process Execution Language for Web Services)[9], XPDL (XML Process Definition Language)[58], or BPMN (Business Process Modeling Notation Specification)[10] . Essentially, a process definition consists of four elements.

- **Tasks** describe the lowest level of details needed to achieve a business activity, for example, "Check credit limit", or "Calculate final price". A task can be an automatic task or a human task. Users interact with human tasks by providing data. Human tasks in turn invoke automatic tasks which accept inputs, perform functionality, and return outputs (to other tasks, users or databases). Automatic tasks are implemented in various programming languages, such as Java.

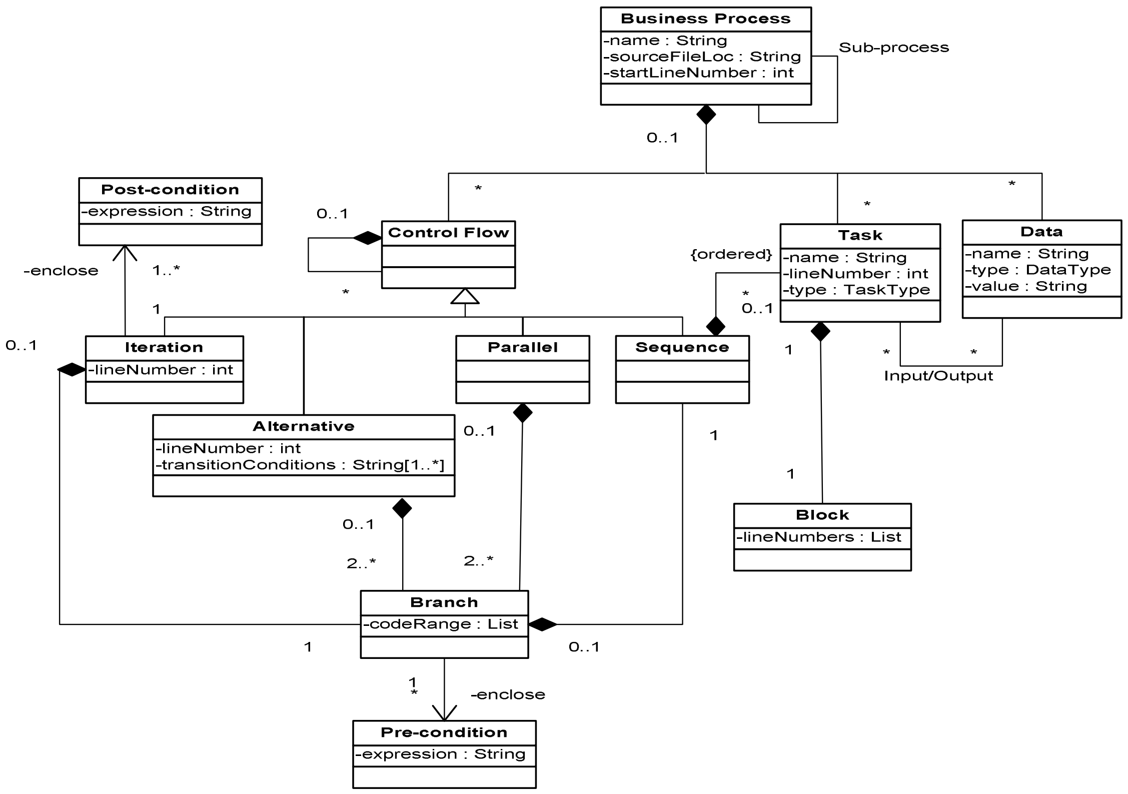


Figure 5: Schema for Representing the Recovered Business Processes

- **Sub-processes** are a grouping of interrelated simpler tasks. For example, the “Get payment address” is composed of several simpler tasks, such as “Get the receiver’s name”, “Get billing address”, and “Get postal code”. A sub-process can be further divided into more sub-processes and tasks.
- **Data flows** describe the input and output data for a task or a sub-process. For example, tax information and product price are the inputs for the “Calculate final price” task.
- **Control flows:** define routing constraints for executing tasks. We list below examples of routing constraints:

  1) *Sequence* – one or more tasks (or sub-processes) and control flow constructs connected in a sequential order. Their sequence of execution is unconditional. For instance, the “Payment” task must be made before we can move to the “Shipping goods” task.

  2) *Alternative* – describes a single thread of control which selects an execution path among two or more alternatives. For instance, a customer can pick a payment method during the check-out process. A *Decision–* relation, which describes two alternative paths, is a special case of the alternative relation.

  4) *Iteration* – are used for repeating the execution of one or more tasks. Iteration has a termination condition. For instance, a customer can keep on putting items in a shopping cart until the customer decides to check-out.

  5) *Parallel* – allows two or more threads of control to proceed autonomously and independently until all threads of control are merged once they are completed. For instance, both the customer and the inventory department must be informed when an ordered item is out of stock.

  6) *Pre-* and *post- conditions* represent entry and exit conditions for a particular sequence of tasks.

We define a schema for representing the recovered processes, as shown in Figure 5. We keep the relevant software entities which correspond to each process entity in a recovered process. For example, for a process entity, we record the source file and the line numbers in the file. Each task includes a block entity, denoting the code portions which contribute to the task.

Control flow constructs, such as iterations and alternatives, contain one or more branches with pre-conditions for the execution of each branch. A branch encloses a sequence of tasks. If the pre-conditions for an iterative branch are evaluated to be true, a sequence of tasks in the branch is executed. The post-conditions are evaluated to decide whether to stop the repeated execution of branches within an iteration control construct. A branch contains the code ranges corresponding to the starting and ending lines of the branch in the implementation. The branches in parallel control constructs have no pre- and post-conditions for their execution order, since they share multiple threads of controls. The XML representation for a process is shown in the annotated screenshot in Figure 2.

A business application typically implements a large number of tasks. To reduce the complexity of understanding the recovered business processes, we represent the recovered processes in terms of two abstraction levels: business processes and sub-processes:

(1) **A business process** gives a complete view of a business process recovered from the UI screens and interactions between the UI and business logic tiers. A business process includes human tasks, which require the interaction with users, simple automatic tasks, executed by a back-end component, or sub-processes. For example, “Find product” and “Enter payment information” are human tasks, while “Create order” is a sub-process that is implemented as a Java method in a back-end component

(2) **A sub-process** details the major steps for a complex task implemented by a back-end component or a UI screen. For example as illustrated in the sub-process viewer of the BPE tool in Figure 2, the sub-process for “Create order” would show the need to create an order for each selected item and the need to update the inventory. However, the sub-process would not show any tasks related to converting the user input or error checking (i.e., utility or data conversion steps) since they are not business relevant tasks. When a UI screen contains more than one task, we represent the tasks recovered from the UI screen as a sub-process.

Separating complex processes into two levels provides a clearer view of the overall structure of a business application while hiding processing details. We represent a recovered business process using a unified schema as shown in Figure 5.

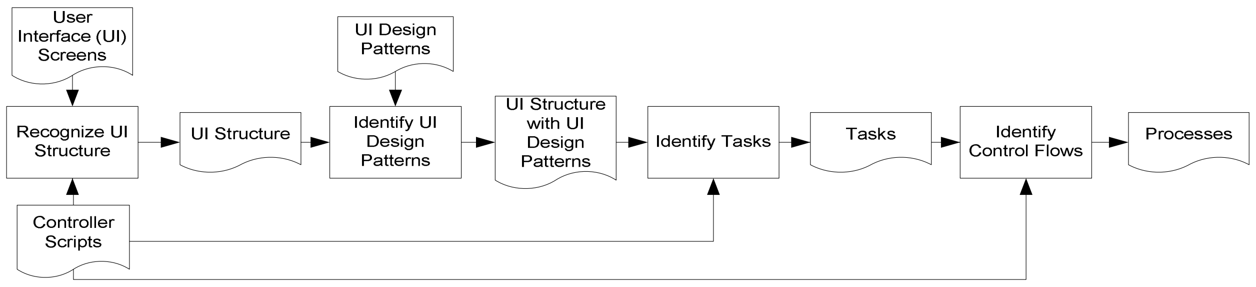


Figure 6: Overall Steps for Recovering Business Processes from UI Screens

4. Recovering Processes from UI Screens

Figure 6 illustrates the overall steps for recovering a business process from the UI screens. The screens can be written using HTML, XML, JSP, or FTL (Freemaker Template Language)[22]. We developed a parser for each language to recognize the UI structures. A UI design pattern describes a functionality delivered through a group of UI widgets [24]. UI design patterns are re-used in interactive UI designs [37][52][57]. Each screen implements several UI design patterns in the business applications we analyzed. To deal with the complexity of UI designs, we recognize UI design patterns to group UI widgets into more abstract tasks. We also capture the control flows between tasks. As a result, we produce a complete process.

In the following sub-sections, we detail our approach. In Section 4.1, we discuss the structure of a screen and the features used for recognizing UI design patterns. In Section 4.2, we present our approach for recovering tasks from the UI code. Section 4.3 explains our approach for identifying control flows.

4.1 Identifying UI Design Patterns

To automate the analysis of screens, we developed a schema for describing the structures of a screen. The schema is shown in Figure 7. Each screen is composed of various UI widgets, such as forms, menus and triggers. A form is specialized into

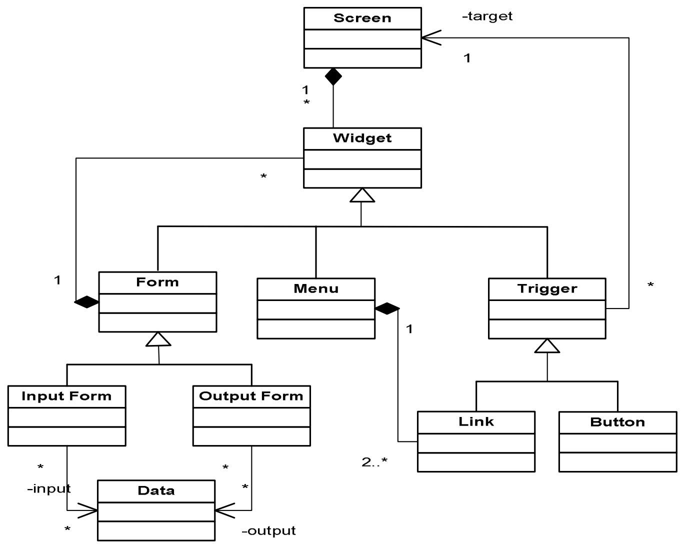
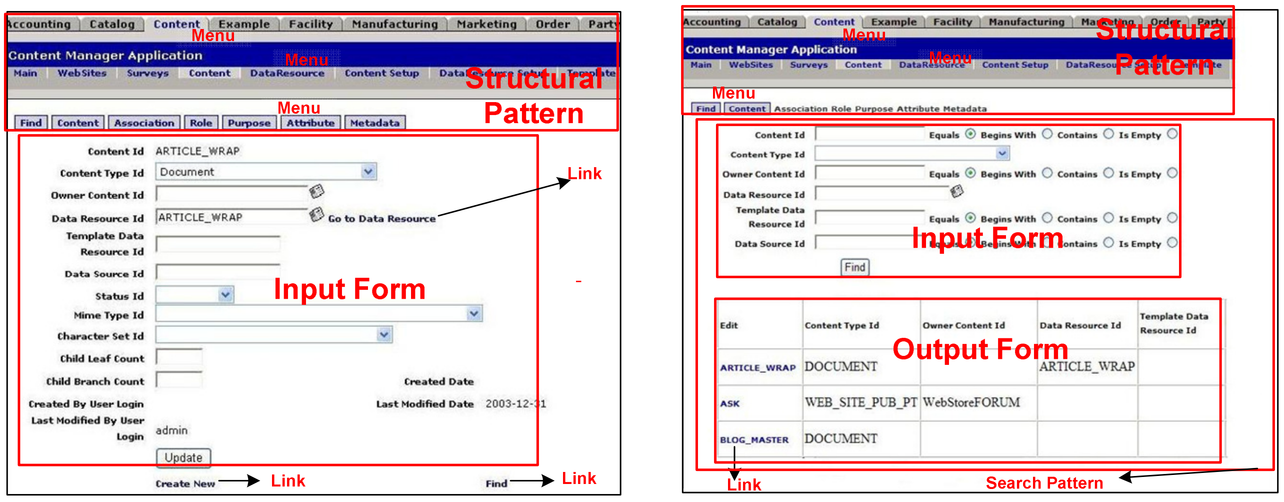


Figure 7: The Schema for the Structure of a UI screen



Content Id	ARTICLE_WRAP
Content Type Id	Document
Owner Content Id	
Data Resource Id	ARTICLE_WRAP Go to Data Resource
Template Data Resource Id	
Data Source Id	
Status Id	
Mime Type Id	
Character Set Id	
Child Leaf Count	
Child Branch Count	
Created Date	
Created By User Login	
Last Modified Date	2003-12-31
Last Modified By User Login	admin

(a) Edit Content Screen

Content Id		Equals [✓ Begins With] Contains Is Empty
Content Type Id		
Owner Content Id		Equals [✓ Begins With] Contains Is Empty
Data Resource Id		
Template Data Resource Id		Equals [✓ Begins With] Contains Is Empty
Data Source Id		Equals [✓ Begins With] Contains Is Empty

Edit	Content Type Id	Owner Content Id	Data Resource Id	Template Data Resource Id
ARTICLE_WRAP	DOCUMENT		ARTICLE_WRAP	
ASK	WEB_SITE_PUB_PT	WebStoreFORUM		
BLOG_MASTER	DOCUMENT			

(b) Find Content Screen

Figure 8: Example of an Annotated Screen of SequoiaERP

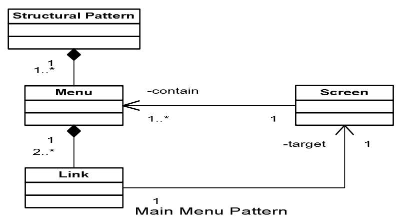
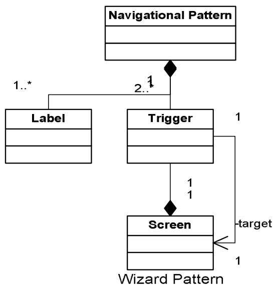
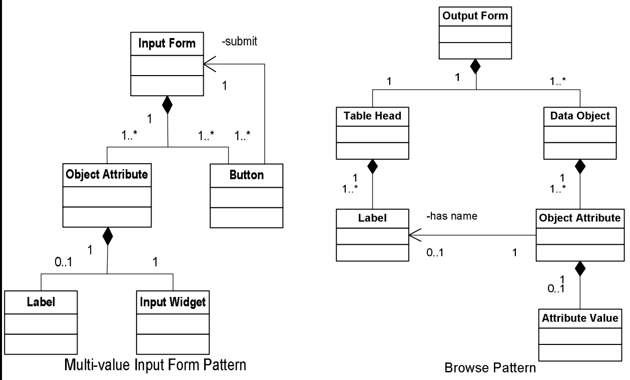
two types: input forms and output forms. An input form allows a user to provide information required by back-end components. An output form displays the results from a back-end component. Buttons and links are triggers. A link connects to a new screen, and a button invokes a back-end component. For example, in the screen shown in Figure 8(a), "Update" button and three links (e.g., "Go to Data Resource") are presented. Essentially, form and menu are high-level widgets that can encapsulate other widgets. Triggers are the widgets at the lowest level of granularity without nested structures.

The complexity of the UI stems from the complex links and hierarchical structure of UI widgets. The complexity of the UI structure causes difficulty in identifying the tasks and the control flows (e.g., sequential and alternative orders) among these tasks in the consecutive screens. Looking at Figure 8(a), an input form contains multiple rows. Each row encapsulates widgets: text fields, links, buttons and selection lists. Many possible tasks with different granularity can be identified from the widgets. For example, the entire input form could be considered as a task for editing the information of a specific content. Alternatively, each text field can be considered as a task for editing one attribute of the specific content. Similarly, the "Update" button could be considered as a task that submits the new content to the database. However, a business process is intended to capture business operations with coarser granularity, rather than the detailed implementation steps. A recovered process definition with an excessive number of fine grained and detailed non-business relevant tasks would be difficult to understand and would be of limited value to practitioners. UI design patterns are used to capture a single business task delivered by a group of UI widgets.

We classify UI design patterns into three categories: structural patterns, navigational patterns, and behavioral patterns. In Table 1, we summarize the features used to detect each category of UI design patterns. We list example UI design patterns for each category in Table 1. We analyze the UI code to identify UI design patterns applied in each screen. We describe our approach below.

Structural patterns: organize the overall structure of a UI screen by clustering different functionalities into widget groups. The features of structural patterns are presented in Table 1. Essentially, the menu provides multiple links, each of which connects to a screen with a functional group (e.g., agreement, accounting, and catalog). An example structural pattern is the main-menu pattern [52][57]. This pattern is used to display a main-menu which appears at the top of each screen and permits quick switching among different functionality groups. Figure 8(b) is an example screen taken from the open source Sequoia ERP [50] in the Apache OFBiz project. The top-level menu widget contains two hierarchically structured menus for a user to select a screen to work on.

Table 1. Features of UI Design Patterns

Category	Pattern instances	Description	Features for Example Patterns
Structural patterns	main menu, double tab navigation, fly-out menu, icon menu[20][52][55]	A set of ordered links grouped into a menu. The menu can be structured in a hierarchy. Each link points to a new screen. The target screen contains the menu and possible sub-menus.	 Structural Pattern 1..* 1 Menu -contain 1..* 1 Screen 1 2..* -target 1 Link 1 Main Menu Pattern
Navigational patterns	wizard pattern[52][55]	A set of triggers are grouped to present the screens in a sequential order. A trigger leads to a subsequent screen, which contains links or buttons labeled with “next” or “continue” to guide a user to navigate through screens.	 Navigational Pattern 1..* 2..1 Label Trigger 1 1 1 Screen -target 1 Wizard Pattern
Behavioral patterns	multi-value input form pattern, browse pattern, search pattern[42]	A set of widgets in the same screen are grouped together to deliver a single functional unit, such as search, browse or input.	 Input Form -submit 1 1 1..* 1..* 1..* Object Attribute Button 1 0..1 1 Label Input Widget Multi-value Input Form Pattern Output Form 1 1 1..* Table Head Data Object 1 1..* 1 1..* Label -has name Object Attribute 0..1 1 0..1 Attribute Value Browse Pattern

To detect structural patterns, we follow the hierarchy relations in the UI widgets and identify the identical link groups appearing in the same level of hierarchy. We examine two features of link groups: each link group must appear in all connected screens; all links in a group are arranged following the same order at a hierarchy level of UI menus. Such a link group is regarded as a menu. By traversing hierarchical structure of the UI menus, we identify the link group for each level of hierarchy. The traversal stops when the link group directly presents UI widgets for performing a collection of business processes. For example, as shown in Figure 8(a), the content management application contains a set of links (e.g., Find, Content and Role). Each link in this level initiates a collection of UI widgets for conducting the first task (e.g., Find existing content) in a business process.

Navigational patterns: describe the navigational structure of the screens. A navigational pattern guides users through screens, as they progress through a process. A business process can be implemented within one screen or can be accomplished across multiple screens. An example navigational pattern is a wizard pattern [52][57] which provides step-by-step instructions to help users complete a complex process. A navigational pattern indicates the progress of the work and uses "previous" or "next" links to move back for fixing problems or to move forward once the current step is completed. We consider the tasks fulfilled by a navigational pattern as a business process.

Similar to the features of structural patterns, a navigational pattern includes a set of text labels and triggers. The features of navigational patterns are shown in Table 1. The text labels display a sequence of steps. In contrast to the links in the structural patterns, the text labels are not clickable since the labels are not associated with screens. A subsequent screen is connected by the "previous" or "next" triggers on each screen. The matching process starts by locating the target screen linked from the "next" trigger and checks if the target screen contains the same set of labels and triggers. The matching process stops when the "next" trigger is not associated with a new screen.

Behavioral patterns: characterize functional units delivered in a screen. To name a few, Table 1 shows the behavioral patterns recognized in the business applications we analyzed: (1) a multi-value input form pattern [42] (i.e., an input form), which encapsulates multiple input widgets; (2) a browse pattern [42] (i.e., an output form), which shows the output from back-end components; (3) a search pattern [42] which allows a user to specify search criteria, and to review the results of a search. A search pattern is abstracted to include an input form for entering search criteria and an output form for displaying the result. The screen as shown in Figure 8(b) exemplifies the aforementioned patterns.

To recognize behavioral patterns in one screen, we first map a form into a candidate behavioral pattern. We expand the candidate behavioral patterns by examining the data dependencies between the form and other widgets (e.g., triggers). A behavioral pattern is identified once we establish data flow relations between the UI widgets and a form. We analyze the data dependencies between the widgets and the forms in a controller script. We group a widget with an input form, if the data passed to the widget is taken from the form; we group a widget with an output form if the data retrieved from the widget is displayed in the output form. For example, the screen, shown in Figure 8(a) has an input form which contains a set of the widgets which gather input from the user. The "Update" button in the screen submits the data gathered from the form to the back-end components. A data flow relation is identified between the form and the button. As shown in the screen in Figure 8(a), we group the form and the submit button into one behavioral pattern (i.e., the multi-value input form pattern).

To ensure that a single unit of functionality is carried in a behavioral pattern, we further analyze the data dependencies among the identified behavioral patterns. If there is data shared among two identified pattern instances, we group one or more instance into one instance at a higher-level abstraction. An input form and an output form often contain common data fields since the output form may display results triggered by the input form. Both forms are grouped as an instance of the search pattern. For example, looking at the screen shown in Figure 8(b), the dependences are identified from the text fields in the input form and the columns in the output form. These two forms are an instance of a search pattern. The result of the searching is listed in the output form.

4.2 Identifying Tasks

Given the starting screen of a process, we determine the controller script which describes the connected screens and the invoked back-end components from that screen. We use navigational patterns and triggers in each screen to trace the progress of a business process across multiple screen files and their corresponding controller scripts. This tracing process continues until we re-visit the starting screen or there are no new triggers to visit. We use the title of the initial screen to name the recovered business process.

To identify tasks, we apply the following three steps in each UI screen:

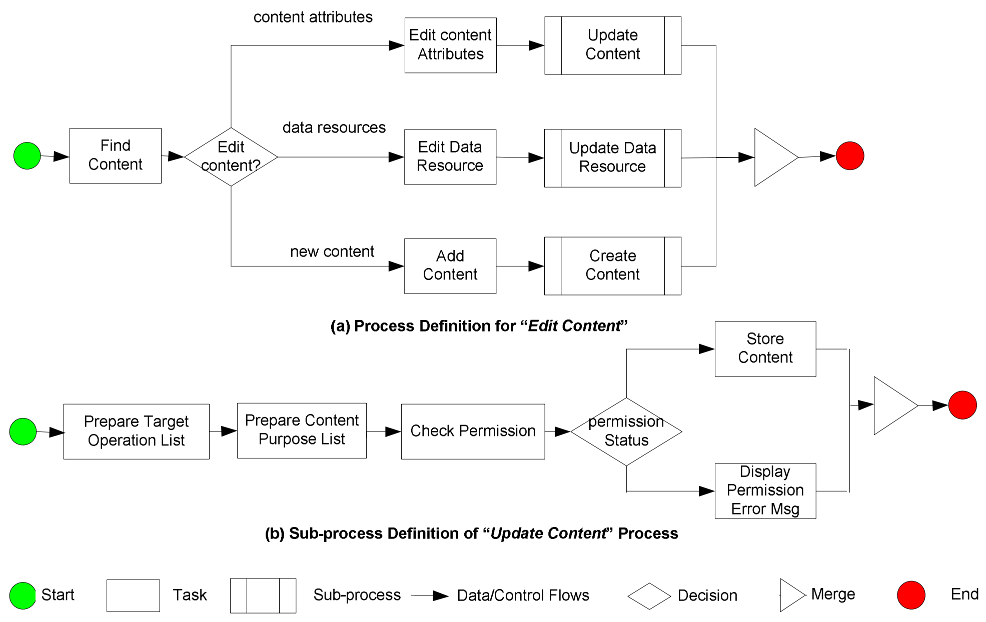
1) identify tasks by recognizing the behavioral patterns (e.g., input or output form) in each screen. Each behavioral pattern is identified as a task.

2) name the tasks using the name of the form. When a task is recovered from a behavioral pattern which is composed of several forms, the name of a recovered task is prefixed using the name of the behavioral pattern (e.g., input, browse or search).

3) identify a group of tasks as a reusable sub-process. A behavioral pattern may contain buttons that invoke back-end components. The detailed steps conducted in the back-end components are resolved when recovering the corresponding sub-processes. The button is, therefore, recovered as a sub-process which corresponds to a back-end component. One screen in the UI can be reused to implement different processes. When more than one task is recovered from the reusable screen, we capture the tasks recovered from the screen into a sub-process. Such a sub-process is reused as a single unit in other recovered processes when the same screen is used.

For example as shown in the "Edit Content" screen of Figure 8(a), the input form is recovered as "Edit Content Attributes" task. The update button invokes a back-end component that takes all the data entered in the input form. The portion of the process recovered from the screen is depicted in Figure 9(a). The "Update Content" sub-process encapsulates the detailed steps carried in the back-end component as shown in Figure 9(b).

When no behavioral patterns are identified in a screen, we render the links which connect to new screens or buttons which invoke back-end components as tasks.



(a) Process Definition for "*Edit Content*"

(b) Sub-process Definition of "*Update Content*" Process

Figure 9: Example Process Recovered from UI Screens vs. Sub-Process Recovered from a Back-End Component

4.3 Identifying Control Flows

In addition to recovering the tasks or sub-processes, we must identify the interaction and coordination between these tasks or sub-processes. The tasks (or sub-processes) are coordinated by control flow constructs such as sequence, loop, and alternative. We currently cannot identify parallel control constructs. We determine each of the control flow constructs by considering the invocation orders and data dependencies among the identified task (or sub-processes).

Sequence: To reduce the number of clicks in a screen and avoid frequently transferring small amount of information over the network, developers group a sequence of tasks into one screen. Once all the tasks in the screen are completed, a new screen is generated for performing the next step. Therefore, the sequential order of the tasks recovered within one screen is derived from the ordering of the behavioral patterns within one screen. Moreover, the data dependencies among the tasks (or sub-processes) convey the sequential order.

To locate the sequential order of the tasks conducted among multiple screens, we recognize if navigational patterns are used to move between screens. The order of the links presented in a navigational pattern indicates the order of executing tasks (or/and sub-processes) recovered from each screen. When no navigational patterns are detected in the screens, the order of screen transitions determines the sequential order of the tasks recovered among multiple screens.

Alternative: Multiple triggers can be selected in one screen. Once one of the triggers is selected, a new screen is generated to replace the previous screen without completing other widgets in the screen. The triggers are independent without data flowing among them. Such triggers in a screen present alternative relations. For example in the screen shown in Figure 8(a), the four triggers (i.e., selecting one of the three links or clicking on the "Update" button) in this screen are in an alternative relation. To locate alternative relations, we examine the data dependencies of the triggers in the same screen. The tasks (or sub-processes) recovered from the independent triggers are in alternative relations. We ignore the triggers that link to the first screen and restart the entire process.

Iteration: If a screen other than the starting screen for a process is re-visited, a user is required to repeat the tasks that were completed until the repetition is terminated by the user. The sequence of tasks being repeated is encapsulated in an iteration control flow construct in the recovered process.

5. Recovering Sub-Processes from Business Logic Tier

Figure 10 illustrates the overall steps for recovering sub-processes from back-end components. To give a complete record of the possible execution paths of an application, we analyze the execution of methods from the entries of the methods invoked from the UI tier. The recovery process visits every statement in the method body in sequence, and recursively follows the call path of the method using the abstract syntax tree (AST) of a back-end component to identify tasks and control flows. As shown in Figure 10, we recover sub-processes from business logic tier in three steps: 1) identify business data (discussed in Section 5.1); 2) locate business tasks (discussed in Section 5.2); and 3) concatenate business tasks using control flows (discussed in Section 5.3).

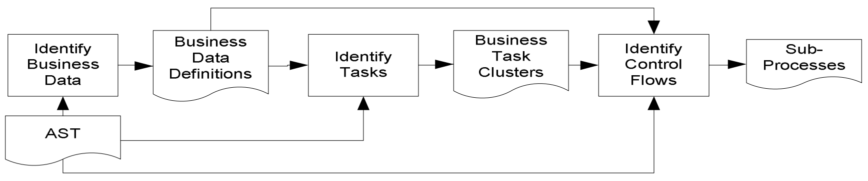
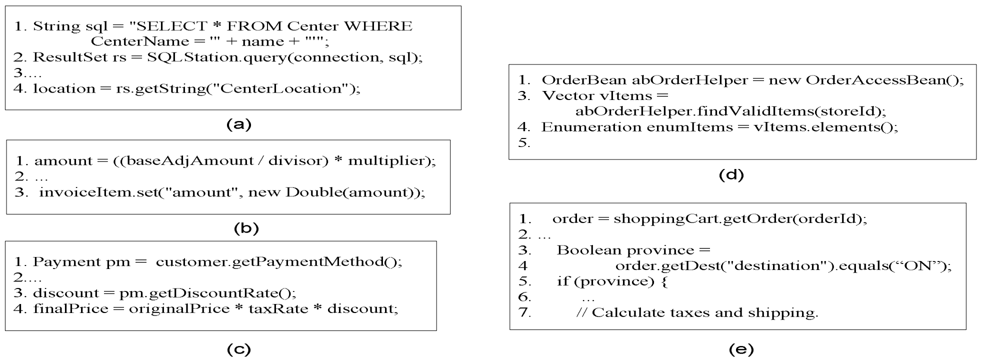


Figure 10: Overall Steps for Recovering Sub-Processes from Business Logic Tier



```
1. String sql = "SELECT * FROM Center WHERE
           CenterName = '" + name + "'";
2. ResultSet rs = SQLStation.query(connection, sql);
3....
4. location = rs.getString("CenterLocation");
```

(a)

```
1. amount = ((baseAdjAmount / divisor) * multiplier);
2. ...
3.  invoiceItem.set("amount", new Double(amount));
```

(b)

```
1. Payment pm =  customer.getPaymentMethod();
2....
3. discount = pm.getDiscountRate();
4. finalPrice = originalPrice * taxRate * discount;
```

(c)

```
1.  OrderBean abOrderHelper = new OrderAccessBean();
3.  Vector vItems =
          abOrderHelper.findValidItems(storeId);
4.  Enumeration enumItems = vItems.elements();
5.
```

(d)

```
1.    order = shoppingCart.getOrder(orderId);
2. ...
3.     Boolean province =
4                  order.getDest("destination").equals("ON");
5.     if (province) {
6.           ...
7.         // Calculate taxes and shipping.
```

(e)

Figure 11: Examples for Business Relevant Code

5.1 *Identifying Business Data*

For a business application, the business data described in a business process supports all the functions related to business operations. For example, in an e-commerce website, the business application contains a catalog of products for sale (i.e., catalog data), and the data associated with processing orders (i.e., tax and shipping address). The business data is populated and stored in the databases. A task takes business data as inputs and generates business data as output. The code blocks which manipulate business data are considered as business tasks [8].

From an implementation viewpoint, business data, such as catalog, product, and customer account, are defined as data beans which are well defined in the code using class definitions. A data bean is automatically mapped to a table in a database and acts as a wrapper for the database accesses. The values of the business data are collected from the users through UI interactions, manipulated in the back-end components and stored in data beans. A set of methods provided in the data beans allow the developers to access databases and populate the values of data beans into the database without knowing SQL statements. In general cases, SQL statements are embedded in the code for direct database accesses.

Business data are detected by identifying from two types of variables within a method: 1) direct business data (i.e., variables directly passed to databases); and 2) indirect business data (i.e., variables defined using the direct business data). To obtain direct business data, we analyze two basic operations for accessing a database: fetch operations and update operations. A fetch operation retrieves data from a database using SQL query statements. An update operation stores data into a database using SQL creation, insertion or modification statements. We scan the code for SQL statements to locate parameters passed to update operations or results returned from fetch operations. The parameters or the results are business data. For example as shown in Figure 11(a), *rs* is the data retrieved from database using a SQL query statement, and therefore is business data. In the case of the data bean implementation, fetch operations are mapped to the getter methods of data beans without changing the state of data bean objects. Update operations are mapped to setter methods that change the status of the data bean objects by modifying the attributes of the data bean objects. For example as shown in Figure 11(b), the *invoiceItem* object is a data bean that corresponds to one table in a database. The *invoiceItem* object uses the *set* method to issue a database update operation with the *amount* variable as an input. The variable, *amount*, as a parameter of the update operation is identified as business data.

To obtain indirect business data, we apply def-use analyses [35] that examine the uses of the direct business data in the assignment statements which define variables. More specifically, the business data is used as one of the variables in the right hand side of an assignment statement; a local variable is defined in the left hand side of the assignment statement. For example shown in Figure 11(c), *customer* is a business data appearing on the right side of the assignment statements. *pm*, appearing on the left side of the assignment statement, is defined by *customer*. Such local variables obtain values from the business data and are used in the same manner as business data for fulfilling a business task.

To infer indirect business data as the result of fetch operations, we analyze the assignment statements following the direction of the execution once a business data is defined by the return of fetch operations. We collect a local variable as business data when the business data are used in the assignment statements to define the local variable. The analysis stops when the business data are re-defined. We continue to analyze the uses of indirect business data in an assignment statement in order to infer other local variable as business data. For example as shown in Figure 11(c), there are three indirect business data from *customer* object (i.e., business data). Once *pm* is assigned from the *customer* object, we further analyze the uses of the *pm* variable in the assignment statements, and obtain its derivations (e.g., the *discount* variable is initialized using *pm* variable). Therefore, the variables, *pm*, *discount* and *finalPrice* are identified as business data. For example in Figure 11(a), *location* is a business data derived from business data *rs* obtained by SQL fetch operation.

Update operations commit updated information to the database. We locate business data as the variables passed as parameters to the update operations. Furthermore, we identify the definition of the business data following the reverse direction of the execution from the update operation. For example in Figure 11(b), we identify that the assignment statement that defines the *amount* variable in line 1 (i.e., a definition of the business data). In some cases, a variable can be assigned multiple times, but only the final assignment determines the value of the variable that is stored as a parameter to the update operation. Therefore, the analysis stops till the last assignment of a variable (i.e., the first assignment statement before the update operations). Once the definition of business data is identified, we apply the def-use analysis following the same procedure as inferring local variables as indirect business data using fetch operations.

5.2 Identifying Business Tasks

To determine whether a code block contains any business logic, we traverse through each statement in the method invoked by the UI tier. To locate a task in the code, we start with identifying a seed statement that indicates business relevant functionality. We detect business relevance of a seed statement by examining the uses of business data in the seed statement [7]. We expand the seed statement by including the related statements on which the seed statement has data dependencies. The related statements declare and use variables appearing in the seed statement. Finally, the seed statement and related statements are marked as a task. We summarize the following three heuristics used to recognize a seed statement with business relevant functionality and produce code blocks corresponding to tasks.

1) A statement, which conducts mathematic computations with the appearance of business data on the right hand side of an assignment statement, is a seed statement. The business data can be used as operands or parameters passing to the functions which return values to the computation. In the example shown in Figure 11(c), line 4 uses a business data, *discount*, in a mathematic computation. Such statement becomes a seed for locating a business task. A seed statement is dependent on all variables used on the right hand side of the assignment statement for the computation. When the left hand side variable that contains the computation result is passed as a parameter to database update operations, the seed statement is also dependent on the left hand side variable. To include a complete set of dependent statements to the seed, we use def-use analysis to locate the statements that define the variables (e.g., *originalPrice*, *tax* and *discount* in Figure 11(c)) used in the seed statement and the statements that use the resulting variable (e.g., *finalPrices* in Figure 11(c)) of the seed statement. Finally, we name the identified task using the name of resulting variable with "cal" as a prefix since this task is involved in a mathematical calculation. As the example in Figure 11(c), the task is named as *cal_finalPrice* which is derived from (i.e., *finalPrice*).
2) A user-defined method invocation, which takes business data as input or produce business data as a return, is a seed statement. The business relevance of such user-defined methods is evident from the manipulation of business data, passed from the input parameters. The business relevance is also supported if the return variable is business data which is passed back to the database update operations. A task is formed by including the related statements which define the variables passed as parameters or which use the return variable. We name the recovered task using the name of the method.
3) Business tasks are conducted in the database using stored procedures. Due to the lack of the access to the source code of databases, we are not able to recover the business tasks performed in the databases. However, the fetch and update operations on the data bean object provide the indicators on the business tasks embedded in the database. Except the trivial methods (i.e., setters and getters), a data bean performs a certain set of business rules, such as data item validation, approval, and removal. This type of methods is a good candidate for a task implementation. Using this rule, we collect non-trivial methods in data beans. These non-trivial methods are seeds for forming task implementation. A task implementation includes the non-trivial method and its related statements. We name the recovered task using the name of the non-trivial methods of the data beans. As illustrated in Figure 11(d), *abOrderHelper* is a data bean object of the *OrderBean* type. The method *findValidItems()* finds items from the database, and also employs business rules to validate the returned items. *findValidItems()* is a seed statement.

There exist many implementation specific features: utility functions, library calls, error handling code, and trivial methods (such as getters and setters) that operate on business data, but are not relevant to achieving business objectives. Specifically, utility function, such as database transactions and I/O classes, perform a set of internal basic operations and are invoked by business tasks. Such utility functions have no impact on the high-level description of the functionality for a process and are filtered.

5.3 Identifying Control Flow Constructs

To link the tasks, we follow the execution paths of the seeds. The order of the recovered tasks is determined according to the order of the seed statements for each recovered task. When a seed, *A*, appears before a seed, *B*, (as shown in Figure 12), we consider that task *A*, which is formed from the seed, *A*, is placed before task *B*, which is formatted from the seed, *B*.

A task can be executed under certain conditions. The conditions are evaluated in a control statement: alternative statement or iterative statement. The different types of alternatives (e.g., ? operator, if-else, if-else-if-else and switch statements) are converted to the alterative control construct in process definitions. The three variations of iterations statements (i.e., for-loop, do-while loop and while loop) are mapped to iterative control construct in process definitions. To determine if a conditional expression in an alterative or an iterative construct is business relevant, we evaluate the uses of the business data in the conditional expressions. If one or more business data appear in the conditional expression, the control statement is business relevant. A sub-process is formed by linking the recovered tasks using the recovered control flow constructs. In the code example listed in Figure 11(d), the variable, *province*, is used in a conditional expression. By analyzing the data dependencies of the *province* variable, the *province* variable is derived from the business data, *order* (as discussed in Section 5.1). The control construct at line 5 is considered as a business-relevant control construct.

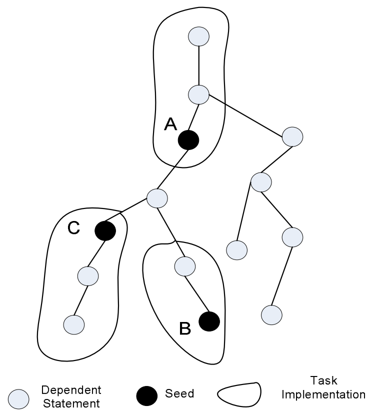


Figure 12: Expanding Seeds to Business Tasks

Similar to recovering business tasks from the source code, not all conditional statements using business data are relevant to business objectives and should not be part of the control flow constructs in a recovered process. A conditional statement may check the initialization of business data in order to avoid errors. Such conditional statements are specific to programming languages and are not meaningful in business domains; thus such control flow constructs should not be shown in the recovered process definition. Moreover, we also filter the recovered control constructs without tasks recovered in the bodies.

6. Case Study

We conducted a case study to measure the effectiveness of our approach in identifying all business tasks from the code of business applications. We used 15 applications from 3 large open source projects in our case study. In the following subsections, we discuss how we measure the effectiveness of our approach. We present the process we followed in our case study. We briefly introduce the studied applications. We present the results of our case study and discuss the limitations of our approach.

6.1 Measuring the Effectiveness of Our Approach

To evaluate the effectiveness of our recovery approach, we use two standard information retrieval metrics, precision and recall. Precision measures the ability of our approach to identify and exclude non-business relevant tasks from the recovered processes. Recall measures the ability of our approach to identify and include all business relevant tasks in the recovered processes. *Eq (1)* measures the precision of our approach for a business process. *Eq (2)* measures the recall of our approach for a business process. Misidentified tasks refer to the tasks which our approach identified incorrectly as business relevant tasks. Missed tasks refer to tasks which our approach did not identify. Each application contains several business processes so we report the precision and recall of all the processes recovered from each application in our case study. We seek an approach with high precision to ensure that a practitioner's time is not wasted examining irrelevant code or tasks, and with high recall to ensure that a practitioner does not miss any business relevant code or tasks.

$$precision = \frac{\text{\# of identified tasks - \# of misidentified tasks}}{\text{\# of identified tasks}} \qquad (1)$$

$$recall = \frac{\text{\# of identified tasks - \# of misidentified tasks}}{\text{\# of identified tasks - \# of misidentified tasks + \# of missed tasks}} \qquad (2)$$

6.2 Case Study Process

To measure precision and recall, we need access to fully documented business processes. However, in practice such documented processes do not exist and if the documentation exists it is usually incomplete. In our case study, we asked a graduate student to manually inspect the source files and recovered business processes to verify the correctness (i.e., precision) and completeness (i.e., recall) of the recovered business processes. Due to the large size of the applications studied, it is infeasible to manually recover business processes without the aid of tools. In the case study, the inspector viewed and verified the recovered processes using our prototype tool (BPE). The BPE tool is used to view various source files (e.g., controller scripts, screen files, and Java code) associated with each recovered task. An annotated screenshot of the BPE tool is shown in Figure 2. The inspector has experience developing business applications and did not participate in the development of our approach or our prototype tool.

The inspector studied the on-line documentation for the subject applications, and navigated through the different screens associated with each business process to better understand the overall structure of each process. The inspector used this acquired knowledge to determine the correctness of the recovered human tasks. The inspector also examined the source code of the applications to understand the structure of automatic tasks.

To measure the precision of our approach for a process recovered from UI screens, the inspector executed the application and observed the screens in order to examine if the recovered human tasks are business relevant or if they are misidentified tasks. For a sub-process process recovered from back-end components, the inspector uses the BPE tool to click through each recovered task and examines the corresponding source code in order to determine if a task is misidentified.

To measure the recall of our approach, the inspector needs to identify tasks which are missing in the recovered processes. To determine if there are any missing human tasks, the inspector runs through the screens of every process and compares the performed tasks with the tasks in the recovered processes. To determine if there are any missing automatic tasks, the inspector uses the BPE tool to highlight all code, which our approach considers to be business relevant. The inspector then manually inspects the code that is not highlighted to determine if our approach missed a task.

Table 2: Characteristics of the Studied Applications

Projects	**Application**	**User Interface Tier**		**Business Logic Tier**		
		Language	**Number of Parsed Files**	**Language**	**Number of Classes**	**Line of Code**
SequoiaERP (0.8.0)	Accounting	XML, FTL, JSP	54	Java, XML minilang	19	8,240
	Catalog	XML, FTL, BSH	136	Java, XML minilang	1	234
	Content	XML, FTL, BSH	204	Java, XML minilang	41	10,178
	Ecommerce	BSH, FTL	165	Java, XML minilang	1	145
	Facility	XML, FTL, BSH	87	XML minilang	N/A	N/A
	Manufacturing	XML, BSH	77	Java, XML minilang	12	3,622
	Marketing	JSP, XML	15	Java, XML minilang	1	231
	Order	XML, BSH, FTL	127	Java, XML minilang	29	13,677
	Party	XML, FTL, BSH	46	Java, XML minilang	8	2,330
	Workeffort	XML, BSH	50	Java, XML minilang	5	781
Opentap (1.0.0)	Crmsfa	XML, FTL, BSH	255	Java, XML minilang	37	7,535
	Warehouse	XML, FTL, BSH	99	Java, XML minilang	11	1,595
	Purchasing	XML, FTL, BSH	85	Java, XML minilang	8	1,085
	Financials	XML, FTL, BSH	108	Java, XML minilang	22	5,611
SMaCs	N/A	HTML, JSP, JavaScript	183	Java	73	5,600

6.3 Studied Applications

We used 15 business applications in our case study. The applications come from three open source projects: OFBIZ SequoiaERP [50], OFBIZ Opentaps [36], and SMaCS[43]. Table 2 lists the characteristics of the studied applications.

SequoiaERP and Opentaps

Both SequoiaERP and Opentaps are available as open source projects and as commercial projects by Open Source Strategies Inc. The open source projects provide the major functionality of the commercial projects and are developed by the same team of developers. We use the open source projects in our case study.

SequoiaERP is an earlier version of Opentaps. Opentaps and SequoiaERP offer a suite of independent applications for ERP (Enterprise Resource Planning), CRM (Customer Relationship Management) and E-Commerce. SequoiaERP contains 10 applications [36]. Opentaps, as a new version of SequoiaERP, has more applications in addition to the 10 applications in SequoiaERP. The newer versions of the 10 common applications have a better structured UI with less links among screens. To demonstrate the strength of our recovery techniques in handling complex UIs, we decided to use the older versions of the applications from SequoiaERP. We also use 4 additional applications from Opentaps. Similar to SequoiaERP, each of the applications in Opentaps is independent from others, and can be used separately. Detailed information about each application in Opentaps and SequoiaERP is available online at [36].

Both projects are built on the Apache Open for Business (OFBIZ) framework. The OFBIZ framework offers a common infrastructure for invoking back-end components and representing the UIs for business applications. The framework defines an XML scripting language, called minilang, which allows developers to easily retrieve and update business data in databases. The screens and controller scripts are written in XML. The controller scripts configure the screens and specify the input and output of widgets displayed in screens. The controller scripts describe the parameters used in the interfaces of the invoked back-end components. The back-end components are implemented using Java or minilang. Database accesses are performed by methods defined in a generic Java class called the *GenericDelegato*r. Moreover, the *GenericDelegator* has methods that dispatch the requests from a UI screen to an appropriate back-end component.

Each application in SquoiaERP and Opentaps is developed by different developers. Although each application must follow the infrastructure defined in the OFBIZ framework, OFBIZ still allows a great degree of flexibility for developers to exhibit their own coding styles. For example, developers can implement a task using different code blocks spread out across several files, or they can choose to implement a task as a single method within a file**.**

SMaCS

To examine the generality of our approach, we selected SMaCS, another business application which does not depend on the OFBIZ framework. The application helps manage casual staff in an organization. The application has business processes for electronic bookkeeping of rosters and timesheets, and for generating pay schedules. In contrast to the OFBIZ applications, where the data beans are defined using data model files, SMaCS defines data beans using Java code. The methods in the data beans capture the SQL statements needed to access and update a particular business data stored in a database.

Table 3: Evaluating Recovered Processes

Application	UI and Controller				Business Logic Tier				Precision	Recall
	# Proc	# Human Tasks	# Missed Tasks	#Misidentified Tasks	# Sub-Proc	# Automatic Tasks	# Missed Tasks	# Misidentified Tasks		
SequoiaERP										
Accounting	29	63	0	0	53	402	0	1	0.998	1
Catalog	68	247	0	0	0	0	0	0	1	1
Content	45	114	0	0	56	235	3	0	1	0.991
Ecommerce	34	93	0	0	1	1	0	0	1	1
Facility	32	151	0	0	0	0	0	0	1	1
Manufacturing	25	85	0	0	36	227	5	4	0.987	0.984
Marketing	8	16	0	0	0	0	0	0	1	1
Order	35	87	0	0	13	60	1	5	0.966	0.993
Party	37	134	0	0	29	175	1	4	0.987	0.997
Workeffort	13	38	0	0	0	0	0	0	1	1
Opentaps										
Warehouse	28	107	0	0	16	69	1	1	0.994	0.994
Purchasing	34	110	1	0	11	91	0	1	0.995	0.995
Crmsfa	64	271	0	0	52	282	0	3	0.995	1.000
Financials	65	127	0	0	27	96	2	0	1.000	0.991
SMaCS										
SMaCS	79	161	0	0	140	153	0	0	1	1
Proc: Business Processes recovered from UI screens Sub-Proc: Sub-Processes Recovered from Back-End Component										

6.4 Analysis of the Results

Table 3 summarizes the results of our case study. Our approach performs well when recovering business processes from UI screens and sub-processes from back-end components. Our approach has precision and recall values which are above 98%. Our approach misidentified tasks primarily due to methods that take business data as parameters, but are used for non-business relevant operations such as data type conversions. On the other hand, our approach missed tasks due to the following reasons:

1) We identify utility methods using coding and naming conventions. For example, we use the naming convention for utility classes and our knowledge of the utility packages to remove utility tasks from our recovered processes. When user-defined methods use similar naming conventions as the utility methods, our approach misidentifies business relevant tasks as utility tasks. The misidentified tasks are not shown in the recovered processes.
2) We rely on database operations to determine the business relevance of tasks. Tasks which process user input and which have no database access are missed during the recovery because such intermediate tasks have no dependencies on data from a database.
3) We use static analysis techniques to identify tasks based on their manipulation of business data. However in some applications, the data is propagated at run-time. For example, in some application the following screen is determined at run-time based on user input. In the future, we plan to combine dynamic analysis with static analysis to capture the tasks invoked at run-time.

In the rest of this sub-section, we discuss in detail the results of our case study relative to each studied application. We also give examples for the misidentified and missed tasks.

Table 4: Commonly Used Utility Classes and Methods in SequoiaERP

Class	**Method**
UtilDateTime	formatInterval getInterval nowTimestamp
UtilFormatOut	formatPriceNumber formatCurrency
UtilMisc	toMap toList

Analysis of Results for SequoiaERP

The “Catalog”, “Facility” and “Workeffort” applications in SequoiaERP did not have a Java back-end implementation. Instead, most of their functionalities were provided by OFBIZ minilang components. Minilang components primarily create, modify and store business data into databases. We consider all minilang back-end components as business relevant tasks since they manipulate business data.

Table 3 summarizes the tasks recovered from the UI and business logic tiers. For each SequoiaERP application, we show the number of recovered processes (Proc) along with the number of human tasks recovered from the screens. We also list the number of recovered sub -processes (Sub-Proc) along with the number of automatic tasks recovered from the business logic tier. For both types of recovered processes, we count the number of missed tasks and misidentified tasks.

Utility Methods: We identify the utility methods by noting that most of utility methods are static methods which are invoked without initiating an object (e.g., *UtilDateTime.formatInterval()*). By using this heuristic, we can filter most of the utility code from the recovered processes. Some of the commonly used utility classes in SequoiaERP are summarized in Table 4.

Missed Tasks: Our heuristics ignore getter methods so we do not parse the body of getter methods. This heuristic works well for simple getter methods, such as *content.getString(contentId).* Although the *content* object is business data, the method *getString()* simply extracts a field from the *content* object and does not contribute to a business operation. Our approach successfully ignores simple getter methods. However, this heuristic does not work well for complex getter methods. The heuristic resulted in missing a few tasks. *getBillingAccountBalance()* and *getPaymentsAddress()* are two examples of complex getter methods which our approach missed. By checking the code inside these methods, the code inspector found that the method body involves retrieving information (e.g., billing and account information) from a database, and performing business relevant tasks. Therefore, such getter methods should be considered as business tasks.

In other occasions, the missed tasks were due to that our approach failing to recognize tasks from methods that did not have business data as parameters. For example, *calPaymentTotal(List payments)* calculates the total payments. The parameters and the return value are not derived from or written to the databases accesses. Instead, the payment information is retrieved from the screens and the value is returned to the screens. Such methods should be treated as business tasks.

Misidentified Tasks: Misidentified tasks are caused by incorrectly identifying methods with business data parameters as business relevant methods. The *findByAnd()*, and *findByOr()* defined in the *GenericDelegator* class, a generic class which handles database accesses, are examples of misidentified tasks. These methods provide no additional business logic other than looking up data entities from the database using supplied conditions. These methods should be treated as utility methods and filtered from the recovered business processes.

The missed and misidentified tasks lead to a slight decrease in precision and recall for our approach. Our approach achieves an average precision of 99.4% and an average recall of 99.7% for SequoiaERP.

Analysis of Results for Opentaps

Table 3 summarizes the tasks recovered from the four Opentaps applications. Table 5 shows some of the commonly used utility classes. Similar to SequrioaERP, the missed tasks in OpenTaps are due to the complex getter methods in user-defined classes and the methods without business data parameters. Different from SequroiaERP, one missed task in Opentaps is due to the fact that although multiple buttons in a screen may link to the same back-end component, the actual back-end components to be executed is determined at run-time based on the clicked UI button. Our static analysis cannot capture information generated at run-time. The misidentified tasks found in Opentaps are due to incorrectly identifying utility methods for handling database accesses only as business tasks. The missed and misidentified tasks lead to a slight decrease in precision and recall. Our approach achieves an average precision of 99.6% and an average recall of 99.4%.

Table 5: Commonly Used Utility Classes and Methods in Opentaps

Class	Method
UtilActivity	removeAllAssociationsForWorkEffort activityIsInactive
UtilDateTime	nowTimestamp
UtilFinancial	toMap toList recurseGetGlAccountTypeIds determineUomConversionFactor

Table 6: Commonly Used Utility Classes and Methods in SMaCS

Class	Method
Utility	teardown
JSPUtils	encodeURL decimalFormat

Table 7: Human Tasks Recovered using UI Design Patterns

Application	Human Task	
	Precision	Percentage
SequoiaERP		
Accounting	100%	37/63=58.73%
Content	100%	73/114=64.04%
Manufacturing	100%	58/85=68.24%
Marketing	100%	12/16=75%
Party	100%	88/134=65.67%
Facility	100%	67/151=44.37%
Catalog	100%	170/247=68.83%
Workeffort	100%	24/38=63.16%
Order	100%	53/87=60.92%
Ecommerce	100%	85/93=91.40%
Opentaps		
Warehouse	100%	45/107=42.1%
Purchasing	100%	63/110=57.3%
Crmsfa	100%	162/271=59.78%
Financials	100%	82/127=64.6%
SMaCS		
SMaCS	100%	56/161=34.8%

Analysis of Results for SMaCS

Table 6 summarizes the common utility classes and methods used throughout SMaCS. We can filter all the utility code from the recovered business processes. Our approach achieves 100% precision and recall rates for the SMaCS project.

Analysis of the Use of UI Design Patterns for Identifying Human Tasks

Our approach uses UI design patterns to abstract a large number of UI widgets to a limited number of human tasks. We sought to closely examine the precision of our approach in abstracting UI widgets into higher level tasks. Table 7 breaks down our precision calculations for the overall approach and shows the precision of the recovery for processes. The table also indicates the percentage of human tasks that are identified using the UI design patterns in comparison with the total number of tasks recovered from the UI screens. For example, in the accounting application our approach achieves a 100% precision in identifying human tasks. Out of the 63 identified tasks recovered from the UI screens, 37 tasks are identified using our UI design patterns. The precision of our approach for identifying human tasks is 100% for all the studied applications. Our results show that the UI design patterns result in simpler processes with business relevant tasks.

6.5 Limitations and Threats to Validity

We now discuss the limitations of our approach and different types of threats which may affect the validity of the results of our case study.

External Validity: tackles the issues related to the generalization of the results. Among the 15 studied applications, 14 applications are based on the OFBIZ framework. While Opentaps has undergone several updates from the initial SequoiaERP version, both projects share many design and coding aspects. To address this potential bias, we chose to analyze, SMaCS, a non-OFBIZ project. All the studied applications use data beans for accessing databases. Moreover, the business applications are developed using clearly defined multi-tiered architecture. Although heuristic, such as UI design patterns and the use of business data, are abstracted from the business applications studied, we should in the future study the benefits of our approach using more projects to determine of our approach works well in other applications and domains.

In our case study, a graduate student, acted as a code inspector and evaluated the recovered processes. Our code inspector has experience in developing business applications and studied the on-line documentation of the applications. It is hard to find a professional business analyst to perform such time-consuming task. Business analysts usually have limited technical background. An analyst with limited programming knowledge would not be able to provide feedback on the effectiveness of our static analysis approach. In the future we plan to conduct a smaller evaluation using a business analyst. The analyst can provide more feedback on the meaningfulness of a small number of recovered processes and would help enhance our results so they are more acceptable in practice. The BPE tool we developed is used to help the code inspector to verify the results due to the large code base of the applications studied. The result may be biased toward the capability of the BPE tool. In the future, we should give the code inspector unlimited time to evaluate our results without the help of the BPE tool. We also want to investigate the feasibility of using dynamic analysis techniques to automate the verification processes.

In the case study, the code inspector found the recovered business processes are very useful to understand the overall functionality of the business applications by simply going through the tasks in the recovered processes. In particular, our approach can link the recovered tasks with the corresponding code blocks with high precision and recall. It allows a developer to easily locate the code needed to be modified to accommodate the changing business processes. In the future, we plan to invite more developers to use our tool to evaluate the usefulness of the recovered business processes for the software maintenance tasks.

Internal Validity: is concerned to the issues related to the design of our case study. The manual inspection introduces bias since a single code inspector could make mistakes. We should have recruited additional code inspectors and evaluated the agreement between their analyses. Unfortunately, we were not able to recruit more code inspectors with sufficient knowledge about business processes and who can spend considerable time to manually inspect our results. Our use of static analysis has limitations as observed in the case study; we will explore the use of dynamic analysis in a future study.

7. Related Work

Identifying Business Logics

Business logic identification is a technique to identify business logic (i.e., decisions and knowledge) in the source code of a business application. Prior research requires extensive human intervention to recover business logics from COBOL applications. For example, Sneed [47] uses code restructuring and use cases to identify business logic from COBOL programs. Earls *et al.* [21] suggest that error-handling code describes business logic violations. In contrast, our work focuses on automating the recovery of business tasks and control flow constructs.

Other researchers have used access to persistent data as criteria for identifying objects. For example, Van Deursen *et al.* [14][15] use persistent data written to or read from files to group COBOL functions into objects. De Lucia *et al.* [12][19] use persistent data accesses to create objects from procedural RPG programs. Sneed and Nyary [46] group persistent data and dependent functions into objects. Similar to these approaches, we use persistent data to identify business tasks. Instead of identifying objects by combining methods and persistent data, we mark the code fragments that operate on persistent data as a business task, then we apply static analysis to propagate the uses of the persistent data and increase the size of an identified task.

Recovering Business Processes and Design Documents

Wil van der Aalst *et al.* [1][2][3] use dynamic analysis to monitor the events generated from workflow management systems in order to recover business processes. In comparison, we use static analysis of the source code of the UI and business logic tiers of business applications which can be developed without using the workflow management systems. In the future, we plan to leverage tools developed by Wil van der Aalst *et al.* to verify the results of our static analysis.

The dynamic and static analyses are primarily used in recovering design documents (e.g., UML sequence diagrams). Examples of using dynamic analysis are presented [8][27][53]. Jiang *et al.* [27] construct usage scenarios to re-document APIs. To understand the collaborations among classes, Richner and Ducasse [40] designed and developed a tool to identify the interactions of classes. Briand *et al.* [8] reverse engineered UML sequence diagrams from distributed systems. A static analysis technique to recover design documents is presented by Tonella and Potrich [53] for reverse engineering interaction diagrams from C++ code. Sneed [48] provides an approach which bridges the gap between programs and specifications. This approach leverages static analysis and tools to re-document programs, and conducts testing against the new specifications. Sneed and Jandrasics [45] propose an approach to translate source code to specifications using an intermediate design schema and entity-relationship model. Using dynamic and static analysis, Lucca *et al.* [16] propose a method that identifies use cases from web applications. These approaches assume that one method invocation delivers a single functionality. In our work, business tasks can be extracted from method invocations as well as the code blocks.

Linking Code and Documents

Marcus and Maletic [31] use latent semantic indexing to establish links between the code and documents (such as bug reports). Different information retrieval techniques are used to compute semantic distances between source code artifacts (such as variable types and names) and documents. Antoniol and Gueheneuc [5] propose using information retrieval techniques to establish traceability links between code and documents. Chen and Rajlich [11] represent a manual search process to map domain knowledge to source code using abstract system dependence graph. The aforementioned research requires the availability of documentation or good domain knowledge. In contrast, our approach works well for business applications with no documentation as demonstrated in our case study.

Analyzing User Interfaces

Much of the research on user interface reengineering focuses on migrating user interfaces, either from text-based platforms to GUI based platform [33][34][54], or from one GUI platform to another [49]. Moore and Moshkina recognize UI code patterns written in C using static analysis. Ricca and Tonella [39] identify architectural features of web sites by using directed graphs to represent navigation flows. Stroulia *et al.* [49] analyze the traces of system-user interactions to replace the legacy UI with web-based UI. Memon *et al.* [32] reverse engineered GUI and generated testing cases. Antoniol *et al.* [4] recover the design of a static website by abstracting the navigational structure of a website. The recovered design is represented using the relationship management data model and the ER+ model proposed with the Relationship Management Methodology. To understand the behavior of web applications, the proposed approaches [16][17][18] analyze the static and dynamic content of web applications to comprehend the dynamic interactions among web components. Our work focuses on identifying UI design patterns in web-based UIs by analyzing the structure of the UI widgets.

Recovering Design Patterns

Design patterns are the solutions to non-trivial design problems that commonly appear in the software development process. In Object Oriented (OO) source code, design patterns are described by a set of classes and the relations among classes [6]. The well known design patterns applied in OO source code are presented by Gamma et al. [23.]. The techniques for recovering design patterns ease program understanding and designs [29][28][6]. Antoniol et al. [6] propose a multi-tag reduction strategy that recovers design patterns from OO source code using software metrics and structural properties. Krämer et al. [29] develop a system, called Pat, to recover design pattern instances through searching the design pattern repository and software design information. Keller et al. [28] represent the source code in UML and use queries to recognize design patterns from source code. Costagliola et al. [13] recover the class diagrams represented using Scalable Vector Graphics (SVG) from OO source code and visualize the recovered design patterns. The proposed approaches are applied on OO source code [23]. Different from the existing approaches, we recover UI design patterns from UI code in order to capture the functionalities delivered by UI widgets. The UI code is developed in various non-OO programming languages, such as XML, HTML and JSP.

8. Conclusions

Organizations are continuously modifying their business processes and the implementation of these processes (i.e., the supporting business applications) to accommodate user requests and market pressures. To improve the agility of business applications, practitioners need tools and techniques to support the co-evolution of the business requirements and the supporting applications. In this paper, we present an approach to automatically recover business processes from multi-tiered business applications, and to establish links between the process entities and their software entities. We demonstrate the effectiveness of our approach by applying it on 15 open source business applications from the Apache OFBIZ and SMaCS projects. The *precision* and *recall* in the studied applications are above 96%. We developed the PBE tool which integrates into Eclipse, a leading software development environment, and into IBM WBM, a leading commercial business process modeling tool. Using the PBE tool, developers and business analysts can better understand business processes and their implementations in familiar powerful toolsets such as Eclipse and WBM.

Our approach applies to the business applications with three tier applications. The UI design patterns are used in the Web-based user interfaces of the applications we studied. In the future, we plan to evaluate the proposed approaches in the business applications without clear separation among tiers. We want to extend the proposed approaches to recover tasks from non Web based UIs where the UI design patterns may not be applied.

References

1. van der Aalst WMP, de Beer HT, van Dongen BF. Process Mining and Verification of Properties: An Approach Based on Temporal Logic. *International Conference on Cooperative Information Systems,* 2005*;* Springer Verlag; 130-140.
2. van der Aalst WMP, Herbst J. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 2003; 47(2): 237–267.
3. van der Aalst WMP, Reijers HA, Weijters AJMM, van Dongen BF, de Medeiros AKA, Song M, Verbeek HMW. *Business Process Mining: An Industrial Application*. Information Systems; 32(1): 713-732.
4. Antoniol G, Canfora G, Casazza G, De Lucia A. Web Site Reengineering using RMM. *Proceedings of International Workshop on Web Site Evolution*, 2000; 9-16.
5. Antoniol G, Gueheneuc Y. Feature Identification: A Novel Approach and a Case Study, *Proceedings of IEEE International Conference on Software Maintenance*, 2005; IEEE Computer Society; 357-366.
6. Antoniol G, Fiutem R, Cristoforetti L. Design Pattern Recovery in Object-Oriented Software. *Proceedings of the 6th International Workshop on Program Comprehension*, 1998; IEEE Computer Society; 153-160.
7. Boehm B. Value-Based Software Engineering. ACM SIGSOFT *Software Engineering Notes*, 2003; 28(2).
8. Briand LC, Labiche Y, Leduc J. Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Transactions on Software Engineering*, 2006; 32(9): 642-663.
9. Business Process Execution Language for Web Services, http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf.
10. Business Process Modeling Notation Specification, http://www.bpmn.org/Documents/OMG%20Final%20Adopted%20BPMN%201-0%20Spec%2006-02-01.pdf.

11. Chen K, Rajlich V. Case Study of Feature Location Using Dependence Graph. *Proceedings of the 8th International Workshop on Program Comprehension*. 2000; IEEE Computer Society; 241 – 247.
12. Cimitile A, De Luica A, Di Lucca GA, Fasolino AR. Identifying Objects in Legacy Systems using Design Metrics. *Journal of Systems and Software, 1999*; 44(3):199–211. DOI:10.1016/S0164-1212(98)10057-2.
13. Costagliola G, De Lucia A, Deufemia V, Gravino C, Risi M. Design Pattern Recovery by Visual Language Parsing. *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, 2005; IEEE Computer Society; 102-111.
14. van Deursen A, Kuipers T. Identifying Objects Using Cluster and Concept Analysis. *Proceedings of the 21st International Conference on Software Engineering*. ACM, 1999; 246-255.
15. van Deursen A, Kuipers T. Rapid System Understanding: Two COBOL Case Studies. *Proceedings of 6th IEEE International Workshop on Program Comprehension*. 1998; 90-97.
16. Di Lucca GA, Di Penta M, Antoniol G, Casazza G, An Approach for Reverse Engineering of Web-based Applications. *Proceedings of Working Conference on Reverse Engineering, 2001*; IEEE Computer Society; 231-240.
17. Di Lucca GA, Fasolino AR, De Carlini U, Pace F, Tramontana P. WARE: A Tool for the Reverse Engineering of Web Applications. *Proceedings of European Conference on Software Maintenance and Reengineering*, 2002; IEEE Computer Society; 241-250.
18. Di Lucca G, Fasolino A, De Carlini U, Tramontana P. Abstracting Business Level UML Diagrams from Web Applications. *Proceedings of IEEE International Workshop on Web Site Evolution*; IEEE Computer Society; 12 – 19.
19. De Lucia A, Di Lucca GA, Fasolino AR, Guerra P, Petruzzelli S, Migrating Legacy Systems towards Object-Oriented PJlatforms. *Proceedings of the International Conference on Software Maintenance*, 1997; 122–129.
20. Designing Interfaces, http://designinginterfaces.com.
21. Earls AB, Embury SM, Turner NH. A Method for the Manual Extraction of Business logics from Legacy Source Code. *BT Technology Journal,* 2002; Springer; 20(4):127-145.
22. FreeMarker Template Language, http://fmpp.sourceforge.net/freemarker/dgui_template_overallstructure.html.
23. Gamma E, Helm R, Johnson R, and Vlissides J. Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley Publishing Company, 1995.
24. Granlund A, Lafreniere D, Carr DA. A Pattern Supported Approach to the User Interface Design Process. *Proceedings of HCI International 2001 9th International Conference on Human-Computer Interaction*, 2001.
25. Huang H. Business Rule Extraction from Legacy Code. *Proceedings of the 20th Conference on Computer Software and Applications, 1996*; IEEE Computer Society; 162.
26. Hung M, Zou Y. A Framework for Exacting Workflows from E-Commerce Systems, *Proceedings of Software Technology and Engineering Practice,* 2005; IEEE Computer Society; 43-46.
27. Jiang J, Koskinen J, Ruokonen A, Systa T. Constructing Usage Scenarios for API Redocumentation. *Proceedings of the 15th IEEE International Conference on Program Comprehension,* 2007; IEEE Computer Society; 259-264.
28. Keller RK, Schauer R, Robitaille S, Pagé S. Pattern-based reverse-engineering of design components. *Proceedings of the 21st international conference on Software engineering*, 1999; IEEE Computer Society Press; 226-235.
29. Krämer C., Prechelt L. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. *Proceedings of the 3rd Working Conference on Reverse Engineering*, 1996; IEEE Computer Society; 208-215.
30. Lehman MM. Laws of Software Evolution Revisited. *Proceedings of the 5th European Workshop, 1996*; Springer Verlag; 108-124.
31. Marcus A, Maletic JI. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. *Proceedings of International Conference on Software Engineering*, 2003; IEEE Computer Society; 125 – 135.
32. Memon A, Banerjee I, Nagarajan A. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. *Proceedings of the 10th Working Conference on Reverse Engineering,* 2003; IEEE Computer Society; 260-269.
33. Moore M, Moshkina L, Migrating Legacy User Interfaces to the Internet: Shifting Dialogue Initiative. *Proceedings of Working Conference on Reverse Engineering*, 2000; IEEE Computer Society; 52-58.
34. Moore M, Rugaber S, Seaver P, Knowledge Based User Interface Migration. *Proceedings of International Conference on Software Maintenance*, 1994; IEEE Computer Society; 72-79.
35. Muchnick S. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.
36. Opentaps Open Source ERP + CRM. http://www.opentaps.org/.
37. Patterns in Interaction Design, http://www.welie.com/.
38. Poo DCC. Explicit Representation of Business Policies. *Proceedings of Asia Pacific Software Engineering Conference*, 1998; IEEE Computer Society; 136-143.

39. Ricca F, and Tonella P. Understanding and Restructuring Web Sites with ReWeb. *Journal of IEEE Multimedia,* 2001; 8(2): 40-51.
40. Richner T, Ducasse S. Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. *Proceedings of IEEE International Conference of Software Maintenanc*e. 2002; 34-43.
41. Shao J, Pound CJ. Extracting Business Rules from Information Systems, *BT Technology Journal*. 1999; Kluwer Academic Publishers: Hingham, 17(4): 179-186.
42. Sinnig D. The Complicity of Patterns and Model-Based UI Development. *Mater thesis in Department of Computer Science*, University of Concordia, Montréal, Canada, 2004.
43. SMaCS. http://sourceforge.net/projects/casualstaffhr/.
44. Sneed HM, Erdos K. Extracting Business Rules from Source Code. *Proceedings of 4th International Workshop on Program Comprehension, 1996*; IEEE Computer Society; 240.
45. Sneed HM, Jandrasics G. Inverse Transformation of Software from Code to Specification. *Proceedings of the Conference on Software Maintenance*, 1988; 102-109.
46. Sneed HM, Nyary E. Extracting object-oriented specification from procedurally oriented programs. *Proceedings of Working Conference on Reverse Engineering, 1995*; IEEE Computer Society; 217–226.
47. Sneed HM. Extracting Business Logic from Existing COBOL Programs as a Basis for Redevelopment. *Proceedings of 9th International Workshop on Program Comprehension*, 2001. IEEE Computer Society; 167.
48. Sneed HM. Software Renewal: A Case Study. *IEEE Software,* 1984. 1(3): 56-63.
49. Stroulia E, El-Ramly M, Iglinski P, Sorenson P. User Interface Reverse Engineering in Support of Interface Migration to the Web. *Journal of Automated Software Engineering*, 2003; Springer, 10(3):271–301.
50. Sequoia Open Source ERP, http://www.sequoiaerp.org/
51. The Apache Open for Business Project, http://www.OFBiz.org/.
52. Tidwell J. Designing Interfaces. O'Reilly Media, 2005.
53. Tonella P, Potrich A. Reverse Engineering of the Interaction Diagrams from C++ Code. *Proceedings of International Conference on Software Maintenance*, 2003; IEEE Computer Society; 59-168.
54. Tucker K, Stirewalt K. Model based User Interface Reengineering. *Proceeding of Working Conference on Reverse Engineering,* 1999; IEEE Computer Society; 56-65.
55. Web Patterns. http://harbinger.sims.berkeley.edu/ui_designpatterns/webpatterns2/webpatterns/home.php
56. WebSphere Business Modeler. http://www-306.ibm.com/software/integration/wbimodeler/index.html.
57. van Welie M. Web Design Patterns. http://www.welie.com/patterns/.
58. Workflow Process Definition Interface – XML Process Definition Language. http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf.
59. Xing Z, Stroulia E. Understanding the Evolution and Co-Evolution of Classes in Object-Oriented Systems. *Journal of Software Engineering and Knowledge Engineering,* 2006; 16(1): 23-52.
60. Zou Y, Lau TC, Kontogiannis K, Tong T, McKegney R. Model-Driven Business Process Recovery. *Proceedings of the 11th Working Conference on Reverse Engineering*, 2004. IEEE Computer Society; 224-233.