An Approach for Estimating Code Changes in E-Commerce Applications

Brian Chan[1*], Lionel Marks[2], and Ying Zou[1]

Dept. of Electrical and Computer Engineering[1]
Queen's University
Kingston, Ontario, Canada
{2byc, ying.zou}@queensu.ca

School of Computing[2]
Queen's University
Kingston, Ontario, Canada
marks@cs.queensu.ca

Abstract

E-commerce applications automate various business processes within an organization. To maintain a competitive edge, business analysts frequently modify business processes. Determining the effort for modifying a business process is not a trivial task, since the changes to the business process will result in changes to the source code for which the business analyst has limited knowledge. In this paper, we propose an approach for tracing the propagation of a change from the business process level to the code. We derive a change impact metric which estimates the code to be modified as a result of a business process change. A case study, using a large e-commerce application from the OFBiz open source project, demonstrates the effectiveness of our proposed change impact metric.

1. Introduction

Most organizations rely upon e-commerce applications for automating their daily operations and processes, such as planning enterprise resources, managing customer relationships and conducting sales using e-commerce websites. A business process describes how a set of logically related tasks are executed, ordered and managed by following business rules to achieve business objectives. For instance, a bookstore's website is an e-commerce application that follows a business process and executes a set of tasks, such as searching for books, adding the selected books into a shopping cart if the books are available, checking the items out and validating the buyer's credit card.

Rapid changes to business requirements are forcing organizations to adapt their business processes and to evolve their supporting e-commerce applications. In the business domain, business processes are constantly reengineered by business analysts to improve organizational performance measures such as reducing cost, increasing revenue and enhancing the quality of services. When a business process is changed (e.g., adding a task, or removing a task) by a business analyst, the source code which implements the business processes must be updated in order to reflect the new business requirements. These code changes may cause unexpected side-effects to other business processes. For example, a simple change to a business process such as the addition of a welcome screen or an order summary page may seem trivial but may require a substantial amount of code changes. The code changes may be too complex when the business value of such a summary page is considered. It is important for business analysts to evaluate the impact of business process changes to the implementation, especially when considering various alternatives. The complexity of the changed code increases the cost for the maintenance. Rough estimates for the changed code range are also critical when customizing a vendor-specific e-commerce application to suit the needs of an organization during a consulting engagement.

Impact analysis attempts to quantify the amount of work that is needed to implement a specific change. However, determining the impact of a business process change is not trivial since business analysts are not experts in understanding source code and the scattering of the changes across various classes and packages may be too complex. In this paper, we develop an approach that gives business analysts a rough and quick estimate on the code change effort for making a business process change. Our approach integrates results from two levels of abstraction: change impact analysis at the business process level and change impact analysis at the source code level. To quantify a proposed business process change, we build a change propagation graph that describes data and call dependencies of potentially affected source code entities. The propagation graph is built by first locating the code entities directly related to the changed business process components, then propagating the change in the source code while filtering code entities based on their business relevance. To provide an accurate estimate for the development effort, we consider a decay model which reduces the probability of changing a code entity based on its distance in the propagation graph from the initial affected code entity. For a proposed change in a business process, our approach:

1) Identifies the business process components which must be changed and stores them in a *business component impact set*;

978-1-4244-2790-1/08/$25.00 ©2008 IEEE

2) Uses established links between the business process components and code entities [20][33] to map all the elements in *business component impact set* to their corresponding code entities;
3) Stores the corresponding code entities in a *source code impact set*;
4) Analyzes the data and control dependencies of the code entities in the *source code impact set* and generates a propagation graph for each code entity in the impact set; and
5) Calculates a *change impact metric*. The metric uses a decay model to capture the development effort of a change.

Organization of the Paper

The rest of the paper is organized as follows. Section 2 presents a motivating example to cover the needed background and motivate our work. Section 3 discusses the impact analysis at the business process level. Section 4 discusses impact analysis at the source code level. We propose a change impact metric for estimating the development effort for business analysts. Section 5 presents a case study to demonstrate the effectiveness of the proposed metric. Section 6 gives an overview of the related work. Finally, Section 7 concludes the paper and discusses future work.

2. A Motivating Example

Business analysts create, visualize, and analyze business processes using business process modeling tools, such as IBM WebSphere Business Modeler (WBM)[30]. Business process definitions are often described in proprietary formats used by particular business process modeling tools. Business processes can also be specified using standards, such as XPDL (XML Process Definition Language) [32]. A business process consists of three major components:

- Tasks describe the steps needed to achieve business objectives. A task can be described by a set of internal properties (e.g., time to execute, automatic task, and resource requirements) and external properties (e.g., input data and output data). For example, the *Purchase Order* process as shown in Figure 1 handles the checkout process and validates the customer payment information. The tasks *Checkout Cart, Enter Shipping Address,* and *Enter Payment Information* are performed by the customer by entering the relevant information into the website. Consequently, the payment information is automatically validated by a credit card processing system.
- Control flows determine the execution path of tasks. A set of tasks that can be executed in different orders, such as sequence, parallel, decision and repetition. For example, the tasks depicted in Figure 1 are executed in sequential order.
- Data flows describe the input/output of a task. For example as shown in Figure 1, *Enter Shipping Address* task accepts *Shopping Cart* as input data and generates *Order* as output data.

We developed a business process explorer tool which automatically recovers business processes from e-commerce applications and establishes the links between the business process components (e.g., tasks, data and control flows) and the source code entities (e.g., methods, and variables)[16]. In an e-commerce application, a variable keeps the data fetched from the database. Such a variable is used as an input for a task operation. Moreover, the content of a variable is saved into the database as an output of task operations. The input and output of a task in a business process are called business data. Such data conveys high level business concepts, such as *Order*, and *Shopping Cart* and are stored in databases. Our tool automatically identifies business data from local variables by checking the direct or transitive data dependencies on the database accesses. The techniques for recovering business processes from e-commerce applications, identifying business data, and maintaining a mapping between (i.e., linking) business process components and source code entities are detailed in [16][20][33]. We have applied these techniques successfully on several large open source and commercial software applications. Figure 1 illustrates the linked code entities corresponding to the tasks in the recovered business process.

A business analyst could examine the process diagram and analyze the impact of a change using a business process modeling tool. For example to provide more secure on-line payment methods for the customers, the business analyst may add *PayPal* as an additional payment method. This decision results in changing the *Validate Payment Info* task in the *Purchase Order* process. The corresponding method *validationController::validatePayment()*, needs to be changed to handle the *PayPal* payment. The changed task and its corresponding code are shown in bold in Figure 1. Consequently, changes to *validationController::validatePayment()* may have an impact on the prior or subsequent methods in the code, which in turn affect the properties of the corresponding tasks in the business process. For example, the method *OrderController::getPaymentInfo()* that implements *Enter Payment Information* task may require restructuring to produce the input data required by the *validationController::validatePayment()*.

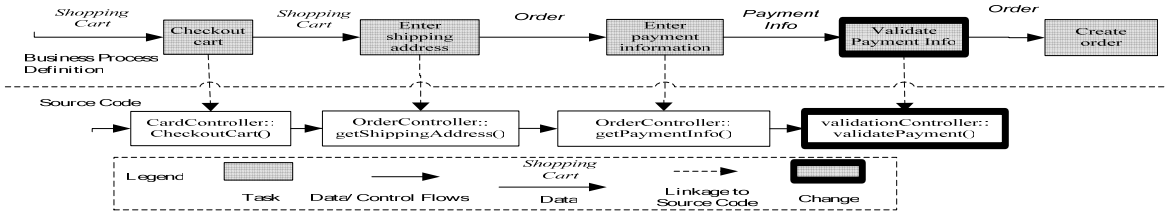

Figure 1: A Purchase Order Business Process with a Change in the *Validation Payment Info* Task to Support PayPal

Table 1 – Summary of Primitive Changes in a Business Process

Case	Name	Primitive Business Process Change	Description	Business Component Impact Set
1	Inner Property Modification	Task 1 → Task 2 (chg) → Task 3 Output(Task$_1$) = Input(Task$_2$) && Output(Task$_2$) = Input(Task$_3$)	Modify the implementation of a task, without changing the interfaces.	S_{input}={ ø } S_{output}={ ø } $S_{logical}$={Task$_2$}
2	Input data modification	Task 1 ⇢ Task 2 (chg) → Task 3 Output(Task$_1$) ≠Input(Task$_2$)	Modify the input interface of a task	S_{input}={Task$_2$ } S_{output}={Task$_1$} $S_{logical}$={ø}
3	Output data modification	Task 1 → Task 2 (chg) ⇢ Task 3 Output(Task$_2$) ≠Input(Task$_3$)	Modify the output interface of a task.	S_{input}={Task$_3$} S_{output}={Task$_2$} $S_{logical}$={ø}
4	Task addition with matched interfaces	Task 1 → Task 2 (new) → Task 3 Output(Task$_1$) = Input(Task$_2$) && Output(Task$_2$) = Input(Task$_3$)	The input/output of the added task match the interfaces of the neighbors.	S_{input}={ ø } S_{output}={ ø } $S_{logical}$={ø}
5	Task addition with mismatched interfaces	Task 1 ⇢ Task 2 (new) ⇢ Task 3 Output(Task$_1$) ≠ Input(Task$_2$) ‖ Output(Task$_2$) ≠ Input(Task$_3$)	Add a new task. The input/output interfaces do not match the interfaces of the neighbors.	S_{input}={Task$_3$} S_{output}={Task$_1$} $S_{logical}$={ø}
6	Task deletion with mismatched interfaces	Task 1 → Task 2 (del) → Task 3 Output(Task$_1$) ≠ Input(Task$_3$)	Delete an existing task. The input/output interfaces of its neighbors are not matched after the deletion.	S_{input}={Task$_3$} S_{output}={Task$_1$} $S_{logical}$={ø}
7	Task deletion with matched interfaces	Task 1 → Task 2 (del) → Task 3 Output(Task$_1$) = Input(Task$_3$)	Delete an existing task. The input/output interfaces of its neighbors are matched after the deletion.	S_{input}={ ø } S_{output}={ ø } $S_{logical}$={ø}
		Task; Data/ Control Flow; Changed Task; Changed Data/Control Flow; Deleted Task; Deleted Data/Control Flow; (chg) = Change; (add) = Add; (del) = Delete		

However, changes to the return type of the method would cause changes in the output of the corresponding task (i.e., the *Enter Payment Information* task) and would produce ripple effects as the required code change can propagate to other methods, classes or packages in the code. Such a change is scattered into a larger scope, and potentially affects other tasks in the same process or other processes in the application.

To estimate the effort needed to make this change, we propose a change impact metric. This metric quantifies the impact of a business process change on the business process and source code. The metric also factors in the propagation of the change at the business process and code levels. Using this metric, business analysts can perform cost-benefit analysis with limited knowledge about the source code which implements the changed business processes.

3. Impact Analysis of Business Processes

Business analysts are familiar with business processes, so they specify changes to business processes at the business process level. We identify seven primitive changes in a business process [31]. Complex changes can be treated as a combination of one or more primitive changes in a business process.

3.1. Primitive Changes in a Process

In a business process, a primitive change is limited to the granularity of a task. Addition, removal or modification of a task are examples of primitive changes in a process. For each primitive change listed in Table 1, we analyze the propagation of the change, and identify a set of changed components in a business process. More specifically, a primitive change of a task may lead to changes in the interfaces of its neighbors. For example, as shown in Case 2 of Table 1, the input data of a task (i.e., Task 2 in Case 2

of Table 1) is modified, then the change propagates to the prior tasks (i.e., Task 1 in Case 2). The propagation of the change indicates that the output data of prior tasks may need to be modified in order to match the modified input data of the changed task (i.e., Task 2 in Case 2). The changed components caused by a primitive change consist of either a change in the input of a task, the output of a task, or the internal logic of a task. For each primitive change, we collect the *business component impact set* by identifying three subsets: S_{input} , S_{output} and $S_{logical}$. S_{input} and S_{output} refer to the sets of tasks that have an interface mismatch in their input or output respectively. For example, in Case 2 listed in Table 1, the *business component impact set* includes the modified task caused by the change of the input (i.e., S_{input}={$Task_2$}) and the prior tasks caused by the change of the output (i.e., S_{output}={$Task_1$}). $S_{logical}$ refers to the set of tasks where the change modifies the internal logic of a task without affecting the input and output of the task. For example, in Case 1 listed in Table 1, the internal properties of a task, such as its execution time, are modified. This change is limited to that particular task. Therefore the *logical set* (i.e., $S_{logical}$ = {$Task_2$}) of such a change contains the changed task.

The *business component impact set* (BCIS) for each primitive change is listed in Table 1. Each entry represents a singular change to a business process. However there are cases when more than one primitive change is made. To find the impact of multiple changes, we must factor in all those primitive changes together. Multiple business process changes are defined as the union of individual primitive changes to the business process as defined in Equation (1). The impact domain of a compound change is composed of its primitive changes. Such changes are the combination of Case 1 (i.e., $S_{logical}$={$Task_2$}) and Case 2 (i.e., S_{input}={$Task_2$} and S_{output}={$Task_1$}). The BCIS for this case is {$Task_2$, $Task_1$}.

$$BCIS = \cup_{ij} (S^i_{input} \cup S^i_{output} \cup S^i_{logical}) \quad (1)$$

BCIS refers to a business component impact set. i indicates a case listed in Table 1, $i \in [1,7]$. S^i_{input} represents a set of tasks with changes in the input in Case i. S^i_{output} represents a set of tasks with changes in the output in Case i. $S^i_{logical}$ represents a set of tasks has logical changes in Case i. The symbol $\cup_{ij}$ is the union over the primitive change j of Case i.

3.2. Tracing A Primitive Change to Code

Following the links between tasks and code entities, we collect methods which implement the tasks in the *business component impact set* and create a *source code impact set*. Most cases as listed in Table 1 involve changes in the input or output of a task in a business process. Furthermore, the inputs of a task are mapped to the incoming business data of the linked method. For example, the *Order getShippingAddress(ShoppingCart cart*) method is linked to the *Enter Shipping Address* task as shown in Figure 1. The *Shopping Cart* as an input to the task is linked to the *cart* variable, defined as a parameter of the method. The output of a task is associated with the variables as outgoing business data from a method. For example, *Order* is the return type of the *Order getShippingAddress(ShoppingCart cart)* method. The return variable in the method is associated with the output of *Enter Shipping Address* task shown in Figure 1.

For Case 1 as listed in Table 1, the changes for the inner properties affect the business logic and rules in the code without modifying the signatures of the method. The business logic is composed of code blocks which operate on business data and deliver tasks specified in a process. A business rule makes decisions according to the state of the business data in the code. A business rule is often implemented using control statements (e.g., if statements, for and while loops). Changes in the business logic and rules are governed by the business data passed as incoming or outgoing data of the method (e.g., using parameters or the return values). For example, as shown in Figure 1, payment information is business data. The state of the payment information determines if the product should be delivered. Adding *Paypal* as a new payment method would change the implementation of the corresponding business rules and logic that operate on the payment business data. Although no interface change is needed in Case 1, it is important to trace the uses of business data as incoming or outgoing data of the linked method. Therefore, the input and output of a task with internal logical changes are mapped to the parameters and return variables of the linked method in the code.

4. Impact Analysis at the Code Level

We perform a static analysis on the abstract syntax tree (AST) of the code and trace method invocations starting from the initial method linked to a changed task. To predict the change impact in the source code, we propose a propagation graph that traces the dependencies of the business data as parameters or return variables derived from the initial changed method. We derive a mathematical formula to give a quantitative estimation of the effort needed to fulfill a changed task in the *business impact component set*. In the following subsections, we elaborate on the impact analysis at the source code level.

4.1. Creating a Propagation Graph

When an initial method linked to a task in a business process is changed, methods which are directly or indirectly invoked by the initial method may be impacted. A call graph captures the control flow relation among methods. In a call graph, a node represents an individual method, and edges represent call sites. To estimate the impact of performing changes in the source code, our early

work [31] expands the call graph by including all the method invocations initiated from the method linked to a changed task. Figure 3 depicts an example of a call graph of the program as listed in Figure 2. In the example call graph, *m1()* calls *m2()* and *m3()*. In turn, *m2()* invokes *m4()*.

In a complex situation, a call graph for a method may enclose all the methods defined in a program. However, not all the methods in the call graph contain business logic and rules. In particular, the call graph includes utility methods that are used as building blocks to implement business logic. When a task is changed, the likelihood for modifying utility methods which do not directly deliver business tasks is low. To provide more accurate estimation on the change effort, the work in this paper builds a propagation graph by filtering the methods that are not relevance to high-level business task. We include the methods that use business data and their derived variables. The heuristics and rules used to determine the business relevance of a code and data entities are detailed in our prior work [16][20][33].The main intuition behind this prior work is the tracing of the flow of business data from user inputs and databases. Figure 4 illustrates a propagation graph for the program in Figure 2.

The propagation of the incoming and outgoing business data is presented as dependency paths in the built propagation graph. We trace the uses of the incoming business data passed from the initial method, and check if the local variables are dependent on the business data in each method along the call paths. We derive business relevant local variables by checking the assignment statements, which assign the information directly or indirectly obtained from the business data to a local variable. For example, as shown in Figure 2, *m1()* is the initial method mapped to a business task in a business process. The parameter *v0* is linked to the incoming business data as the input of the changed task. We trace the uses of *v0* within the method body of *m1()* to derive local variables dependent on *v0*. In this example, the local variables, *v1*, *v2*, and *v3*, are derived as business data due to their dependencies on *v0*. The dependencies are established using assignment statements in lines 3 to 5.

Similar to tracing the propagation of the incoming business data, the propagation of the outgoing business data is traced from the definition of the outgoing business data in the method. We locate the definition of an outgoing business data in a method and analyze the uses of the outgoing business data in defining other local variables. For example in Figure 2, we assume that *v5* is a return variable (i.e., an outgoing business data) in line 12 and defined in line 10. The uses of *v5* include lines 11 and 12 in Figure 2.

```
1.    m1(Type0 v0)
2.    {
3.        Type1 v1= v0.getValue();
4.        Type2 v2 = v1.getValue() + 1;
5.        Type3 v3 = m2(v2);
6.        m3(v1);
7.    }
8.    Type3  m2(Type2 v4)
9.    {
10.       Type3 v5= v4.getValue();
11.       m4(v5);
12.       return v5;
13.   }
14.   void m3(Type1 v6)
15.   {
16.       Type4 v7= v6.getX();
17.   }
```

Figure 2: Example Program

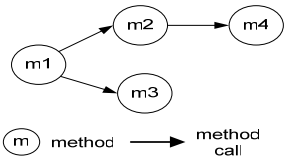

Figure 3: An Example Call Graph

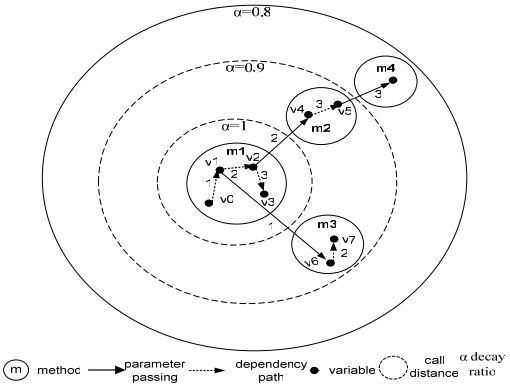

Figure 4: An Example Propagation Graph

The business data can be passed through parameters in method invocations. We expand a dependency path when business data is passed as parameters to an invoked method. We examine if the local variables defined in an invoked method are business relevant using assignment statements. For example as shown in Figure 4, *v4* is business data since *v2* (i.e., business data) is passed to *v4*. Similarly, *v5* is derived as business data due to the data dependence on *v4* (i.e., business data). The expansion of a propagation graph stops under the following conditions: 1) an invoked method is linked to another task in the *business component impact set*; 2) the method calls itself; and 3) the end of the program. In the propagation graph, we omit local variables that are not business relevant, since such local variables are more implementation specific and are not likely impacted by changes to a task.

As shown in Figure 4, the propagation graph contains the business relevant methods associated by the parameter passing in call paths, and the dependency paths of business data. The incoming business data and the definition of the outgoing business data are the sources of dependency paths. For example, *v0* is the source of three dependency paths (i.e., (*v0*, *v1, v6, v7*), (*v0, v1, v2, v3*), and (*v0, v1, v2,*

v4, v5, ...)). Whenever a change in the source occurs, it will affect other variables on the dependency paths. If more than one dependency path is possible to a variable, the shortest path will be taken into account.

4.2. Estimating Change Impact Value

The method in the center of a propagation graph represents the initial method that is linked to a changed task. As an invoked method is further away from the method in the center, the likelihood of such a method to be impacted is minimal. Similarly, when business data (i.e., variable) are further away from the source (i.e., the initial changed business data) along a dependency path, the potential for changing the code that uses the variable is decreased. Moreover, the complexity of a method also determines the effort needed to understand and modify its code. To estimate the impact for changing business logic and rules, we measure the complexity of the control statements that involve business data.

$$ImpactValue = \sum_{business\ data} DependencyDistance * (Complexity + 1) \quad (2)$$

ImpactValue is an indicator for the change impact in the code. DependencyDistance refers to the distance of business data away from the source node of a dependency path. Complexity estimates the difficulty in changes control statements using business data.

We propose Equation (2) to calculate the impact value which predicts the change impact. Using the propagation of business data identified in a propagation graph, we quantify the impact by considering the dependency distance of each business data in a dependency path and the complexity of control statements involving each business data in the dependency paths. When no business data are used in control statements, the complexity value assigned for changing business logic and rules is zero. In the following sub-sections, we derive the measurement for dependency distance as defined in Equation (4). The complexity of the control statements that business data are involved in is also derived later in Equation (5).

4.2.1. Measuring the Dependency Distance

To measure the propagation decay of a change to an initial method, we measure the call distance between the initial method and each invoked method in the propagation graph. For example, as illustrated in Figure 4, the *m1()* in the center represents the initially changed method. The methods (e.g., *m2()* and *m3()*) are one distance away from *m1()*. We introduce, α, as the decay ratio of a method, using equation (3). The example values of α (e.g., *k=0.1)* are illustrated in Figure 4. The choice of the *k* value controls the change impact to the methods further down in a call path. The larger the *k* value, the less impact a method with a large call distance would be affected by a business process change.

$$\alpha = 1 - k * callDistance,\ 0 \le k \le 1,\ 0 \le \alpha \le 1 \quad (3)$$

Each method in a propagation graph has a collection of business data as described in various dependency paths (shown in Figure 4). We count the dependency distance that each business data is away from the source of a dependency path. Business data with a greater dependency distance in a dependency path has less chance to be modified. The source business data mapped to the interface of a task (e.g., *v0* in Figure 4) has a dependency distance of 0. For example as shown in Figure 4, *v1* is defined using *v0*. Then, *v1* is 1 distance away from *v0*. When a variable is passed to a parameter of a method, the dependency distance of the parameter of the invoked method retains the same dependency distance as the passing variable. For example as shown in Figure 4, *v4* as a parameter of *m2()*, has a dependency distance of 3 same as *v3*, a variable in *m2()*.

$$DependencyDistance = varCount \times \frac{\alpha}{disc + 1} \quad (4)$$

varCount counts the number of instances for business data within a method in a propagation graph. disc is the call distance of business data along a dependency path. α is the decay ratio of the method that contains the business data.

Although business data in different methods may have the same dependency distance (not call distance (disc)), the business data in the central method (e.g., *m1()* in Figure 4) is more likely to be changed than the business data in the methods further away from the center. For example, *v2* and *v7* have the distance of 2 in the dependency path. However, *v2* in *m1()* is more likely to be modified than *v7* in *m3()*. Therefore, we integrate the decay ratio of a method, which contains the business data in a dependency path, into the computation of the dependency distance of business data. Moreover, the same business data can be used multiple times in a method. The overall dependency distance of one business data counts the frequency of uses within a method. As a summary, the dependency distance of business data is computed using Equation (4).

4.2.2. Measuring the Complexity of a Change

We propose Equation (5) to measure the complexity of a change associated with control statements, which examine business data collected in a propagation graph. Business data may be used in the conditional expression of different control statements in the code. To this effect, we compute the complexity of a single control statement, which examines the number of variables used in the conditional expressions, the nested condition depth, and the depth of the business data in the control statement.

$$Complexity = \sum_{statements} NumVar + ConditionDepth - OccuringDepth \quad (5)$$

NumVar refers to the total number of variables used in a control statement. ConditionDepth refers to the total number of nested conditions within the control statement. OccuringDepth is the smallest depth of business data used in conditional expressions.

A conditional expression gets complex when more variables are used. As defined in Equation (5), *NumVar* counts the number of variables used in the expressions of a control statement. This includes variables that are not directly related to business data. For example, *if(a<b)* has a *NumVar* value of 2 regardless of how *a* and *b* are created as long as the control statement uses business data in a conditional expression. As stated in the McCabe complexity metric [25], the larger the number of nested conditions in a control statement, the more complicated of the control logic. *ConditionDepth* is the total number of nested conditional structures within a control statement. For example as shown in Figure 6, the *ConditionDepth* is 4. *OccuringDepth* is the level that business data is occurred in the nested conditional structure. Business data at a shallower nested condition would cause a greater change to the control logic than a more deeply nested condition so the shallowest depth is considered. In Figure 6, the *OccuringDepth* of the business data corresponds to 3.

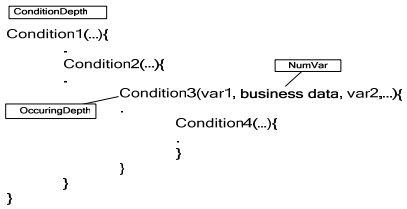

Figure 6: Example Conditional Structure

5. Case Study

We conducted a case study to examine the effectiveness of the proposed change impact metric in predicting change impact in the code. In this section, we describe the setup of the case study, and the criteria for evaluating the results. We also discuss the results and the threats to their validity.

5.1. Setup

The Apache Open For Business Project (OFBiz) [3] is a large open source project that offers a suite of e-commerce applications for ERP (Enterprise Resource Planning), CRM (Customer Relationship Management) and E-Commerce. The Business Process Explorer Tool [16] is used to recover the business processes implemented in OFBiz and establishes the links between the business process components and source code entities in the OFBiz project. To assist a business analyst in reviewing and analyzing the business processes, we visualize the recovered processes using a commercial business process modeling tool, IBM WebSphere Business Modeler (WBM) [30]. Changes in the business processes are captured and traced into the linked source code entities. To assess the impact of a proposed change in a process, we developed a prototype that traces the linked source code entities and calculates the proposed change impact value for primitive changes in business processes.

5.2. Criteria for Evaluating the Change Impact Metric

To measure the effectiveness of the proposed impact metric, we want to examine how well the change impact metric can be used to predict the amount of time a developer needs to make code changes as a result of particular change in a business process. We assume that the developers take more time to complete their code analysis for changes that have a greater impact on the source code. The prediction of the change impact metric should relate to the amount of time a developer takes. For example if the change impact metric returns a small value, it should take a developer a short time to make the changes in the code. However if a developer devoted a large amount of time to accommodate a primitive change in a process, the impact metric should return a larger value. If the correlation is not strong, no meaningful information can be garnered using our proposed change impact metric.

We also use LOC as a base metric to predict the amount of time (i.e., time-to-change) that a developer spent in changing code for the same primitive change in a process. We want to compare the performance of LOC and the proposed change impact metric. Intuitively the larger the affected LOC is, the more time a developer spends in understanding and modifying the code. To measure the LOC, we sum the LOC of each method in a propagation graph. Prior research shows that most complexity metrics correlate well with LOC [23].

5.3. Data Collection

To compare the performance of the two metrics (i.e., LOC and the proposed change impact metric) in predicting time-to-change, we recruited five developers (graduate students) to implement specific business changes. To ensure unbiased results, the five developers are not stakeholders of the work. They are not familiar with the source code of the OFBiz project. The five developers have a similar level of proficiency in Java, the implementation language of the OFBiz project.

In the recovered processes from the OFBiz project, the tasks are implemented using HTML and Java. In our study, we focus on predicting the change impact on the Java source code. We randomly picked 35 primitive changes in the tasks implemented using Java from 7 different business processes. Each developer was assigned 7 primitive changes. We measure the actual time for each developer to locate the code and perform the change. Specifically, the time is measured as how long it takes the developers to study, implement, debug, and test the change.

$$y = 0.756x + 0.490 \quad (6)$$

x is an impact value and y is the predicted time-to-change

5.4. Analysis of the Results

Figure 14 shows the linear relation between the impact value and the actual time-to-change. The R^2 value of 0.88 reflects a strong relation between the impact value and the time-to-change. In the OFBiz project, the call distances are on average of 3 invocations away from the initial methods linked to the 35 primitive changes. We set *k* (i.e., equation (3)) to 0.1 to allow a change to be propagated to 9 method calls along a call path. This value permits us to factor in the methods in the longest call paths.

The results indicate that a larger impact value on the code incurs longer time for developers to analyze. Given an impact value, we use the generated trend function as specified in Equation (6) to predict the time required for a developer to implement the business level change in the code. The results enforce the argument that the change impact metric predicts accurate change effort in regards to the amount of changes to the code. We examine the linear relation between the time-to-change and the LOC. The results are summarized in Figure 15 for the same 35 changes conducted by the five developers. The R^2 value of 0.56 shows that the LOC would not give a good indication of the effort that a developer needs to implement the changes. The LOC measure does not consider the decrease (i.e., decay) in the likelihood of a change being performed. So 10 LOCs in the initial changed methods are worth the same as 10 LOCs in a method that is 4 edges away in the propagation graph from the initially changed method. It should be noted that decreasing k beyond 0.1 will increase the impact value unnecessarily, insinuating a small change will falsely cause a large impact that can be altered with little time. Similarly increasing k beyond 0.1 will cause the impact value increment too slowly, suggesting larger changes could actually be done in too little time.

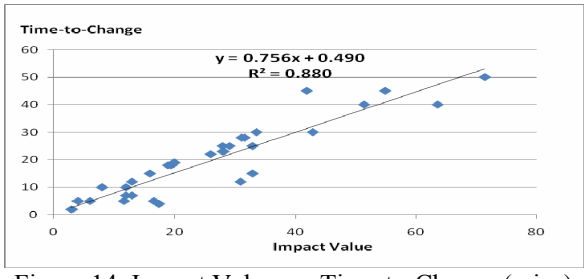

Figure 14: Impact Value vs. Time-to-Change (mins)

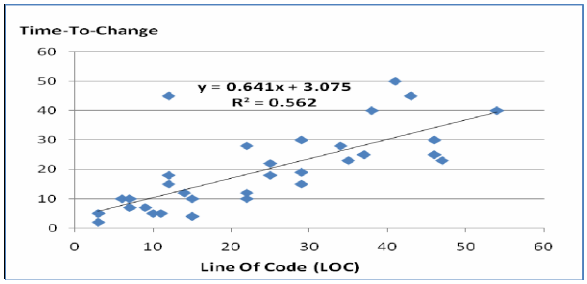

Figure 15: LOC vs. Time-to-Change (mins)

5.5. Threats to Validity

In the case study, we evaluated the change impact metric using e-commerce applications contained in the OFBiz project. To study the generality of the change impact metric, we should evaluate our work in more projects different from OFBiz project.

Five graduate students were recruited for performing the changes to the code. All students are not familiar with the code of the applications. However, the time-to-change is largely dependently on the coding style, and the developers' knowledge on the code. The time needed to perform a change may drop as the developers get more familiar with the code as making more changes. To avoid the learning effect to obtain the actual time-to-change, each developer is only limited to make 7 changes in the case study. Further research should be focused on whether the knowledge of the developers would affect the relations of the change impact values and the time-to-change. We also plan to conduct a larger case study with more participants.

Our metric is based on static analysis of the code, therefore when propagating a change we may miss propagating a change due to dynamic dependencies. Our metric could make use of dynamic dependencies however such dependencies are more time consuming to calculate. It would be interesting to measure the additional improvement in the accuracy of our metric estimates and to compare the improvement against the additional overhead.

6 Related Work

Software metrics are used to measure software characteristics in order to predict software quality attributes, such as maintainability, reusability and modifiability [6][7][12][13][14][18][21][35][36]. For example, Bilal and Black [6][7] presents a ripple-effect algorithm that measures the complexity of an object-oriented software. Similar to our work, the ripple-effect algorithm traces the data flow of a variable in the program [35][36]. In our research, we consider the diminishing effect of a variable on a change based on the distance of that variable from the initial change.

Techniques are proposed to perform impact analysis based on higher level abstraction than code [1] [8][10][11][29]. Ajila[1] investigates the change impact among four phases in the software development: requirements, specification, design and implementation. The changes performed in the earlier phases are analyzed to understand their impact on later phases. Instead of using business processes recovered from existing systems, de Boer et al. [8] study the ripple effect on software architecture description by modifying the relations in a software architectural design. In [10] [11], impact analysis is conducted using UML models. Soffer [29] provides techniques to analyze the change impact on business processes without considering the code that implements the business processes. In our work, we propagate the change

propagation from a higher level of abstraction (i.e., business processes) of software systems to the code level.

A plenary of research performs impact analysis on the source code. Bohner et al. [9] focus on the source code changes that would affect the security aspect of the system. Hassan and Holt [17] conducted an empirical study on how changes to source code entities affect an entire system. Hoffman [19] predicts modules to be affected based on maintenance changes. Lee [22] investigates the impact on legacy software and the challenges for modifying older code. Mendes et al. [26] provide impact analysis techniques for web applications. In [28], dynamic analysis is used to evaluate the impact of changes in software systems. The static analysis used in our approach cannot identify dynamic method invocation. Sneed proposed various models for estimating costs for performing maintenance tasks [37][38]. Our work focuses on estimating the impact on the code. The impact on the code is one of the many factors affecting the overall cost of performing a maintenance task.

Traceability of code to the business process level was needed for our work. We use the techniques garnered in [18][31] which discussed the techniques to obtain business processes. Other research on traceability focuses on tracing techniques from code to documentation [2][5][24][27].

7. Conclusion and Future Work

We present an approach which propagates business process changes to the linked code entities. We propose a change impact metric which quantifies the impact in the code in response to changes at the business process level. Our work assists business analysts in determining the needed effort for performing business process changes without having to study the code. However, a modification in a business process impacts also other components different from the software system: human resources, technological resources and logistic components [34]. In our research, we focus on analyzing the impact caused by business process changes on software systems and providing rough estimates. The rough estimates are particularly important when customizing a vendor-specific e-commerce application to suit various needs of an organization during consulting engagements.

In the future, we plan to perform a more in-depth empirical case study to evaluate our approach on more e-commerce applications. This includes further justification of variables such as propagation decay. We want to investigate if developers with different level of knowledge on a subject application have an effect on the performance of our proposed change impact metric. More attention must be given to the business level analysis, studying the impact and cost of changes in software systems on the business process.

References

[1] S. Ajila "Software Maintenance: Approach to Impact Analysis of Objects Change", *Software-Practice and Experience*, Vol. 25(10), 1995, pp. 1155-1181.

[2] G. Antoniol, G. Canfora, A. De Lucia, Maintaining Traceability During Object-Oriented Software Evolution: a Case Study, *International Conference on Software Maintenance (ICSM),* 1999.

[3] The Apache Open for Business Project, http://www.OFBiz.org/.

[4] M. Asmild, J. C. Paradi and A. Kulkarni, "Using Data Envelopment Analysis in Software Development Productivity Measurement", *Software Process Improvement and Practice*, 2006, pp. 561-572.

[5] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, E. Merlo, "Recovering Traceability Links between Code and Documentation", *IEEE Transactions on Software Engineering*, Vol. 28(102002), pp. 970-983.

[6] H. Bilal, S. Black, "Ripple Effect: A Complexity Measure for Object Oriented Software", *European Conference on Object-Oriented Programming*, 2007.

[7] S. Black, "Computing the Ripple Effect" in *Journal of Software Maintenance and Evolution: Research and Practice*, 2001, pp. 263-279.

[8] F.S. de Boer, M.M. Bonsangue, L.P.J Groenewegen, A.W.Stam, S.Stevens and L. Van Der Torre, "Change Impact Analysis of Enterprise Architectures", *Information Reuse and Integration Conference*, 2005.

[9] S. Bohner and D. Gracanin, C. Dong, B. George, J. Ying and Y. Zhou, "Software Security Impact Analysis", *Goddard Space Flight Center - Advanced Architectures and Automation Branch*, 2002.

[10] L. C. Briand Y. Labiche G. Soccar, "Automating Impact Analysis and Regression Test Selection. Based on UML Designs", *ICSM*, 2002.

[11] L. C. Briand, Y. Labiche, L. O'Sullivan, "Impact Analysis and Change Management of UML Models", *ICSM 2003.*

[12] S. R. Chidamber and C. F. Kemerer. "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, Volume 20(6), June 1994, pp. 476-493.

[13] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, "The effect of inheritance on the maintainability of object oriented software: An empirical study", *ICSM* 1995.

[14] S. L. Fleeger, S. A. Bohner, "A Framework for Software Maintenance Metrics", ICSM 1990.

[15] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History", IEEE Transactions on Software Engineering, 25(7), July 2000.

[16] J. Guo, K. Foo, L. Barbour, Y. Zou, "A Business Process Explorer: Recovering and Visualizing E-Commerce Business Processes", International Conference on Software Engineering, 2008.

[17] A. E. Hassan and R. C. Holt, "Predicting Change in Propagation in Software Systems", *ICSM*, 2004.

[18] S. Henry and D. Kafura, "The Evaluation of Systems' Structure using Quantitative Metrics", *McGraw-Hill International Series*, 1993, pp.99-111.

[19] M. A. Hoffman, "Automated Impact Analysis of Object-Oriented Software Systems", *International Conference on*

*Object-Oriented Programming, Systems, Languages, and Application*s, 2003, pp.72-73.

[20] M. Huang, Y. Zou. "Recovering Workflows from Multi Tiered E-commerce Systems", *International Conference on Program Comprehension,* 2007, pp.198-207.

[21] H. Kagdi and J. I. Maletic, "Combining Single Version and Evolutionary Dependencies for Software Change Prediction", *International Conference on Software Engineering*, 2007.

[22] M. Lee, "Change Impact Analysis of Object Oriented Software", *George Mason University,* 1998.

[23] M. Lipow, Number of Faults per Line of Code, IEEE Transactions Volume SE-8, Issue 4, July 1982, Pages 437-439.

[24] A. Marcus, JI. Maletic, Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. ICSM, 2003.

[25] T. McCabe, and C. Butler, "Design Complexity Measurement and Testing", Communications of the ACM 32, 12 (December 1989), pp. 1415-1425.

[26] E. Mendes, N. Mosley and S. Counsell, "Web Metrics – Estimating Design and Authoring Effort", *IEEE Multimedia*, 2001, pp. 50-57.

[27] A. De Lucia, R. Oliveto, G. Tortora, "Recovering Traceability Links using Information Retrieval: a Controlled Experiment", *International Symposium on Grand Challenges in Traceability*, 2007, pp. 46-55.

[28] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold, "An Empirical Comparison of Dynamic Impact Analysis Algorithms", *International Conference of Software Engineering*, Scotland, UK, 2004.

[29] P. Soffer, "Scope Analysis – Identifying Impact of Changes in Business Process Models", *Software Process Improvement and Practice*, 2005, pp.393-402.

[30] WebSphere Business Modeler. http://www-306.ibm.com/software/integration/wbimodeler/index.html.

[31] H. Xiao, J. Guo, Y. Zou, "Supporting Change Impact Analysis for Service Oriented Business Applications", *International Conference on Software Engineering Workshops*, 2007.

[32] XML Process Definition Language (XPDL), http://www.wfmc.org/standards/docs.htm#XPDL_Spec_Final. April 2008.

[33] Y. Zou, T. C. Lau, K. Kontogiannis, T. Tong, R. McKegney. "Model Driven Business Process Recovery", *Working Conference on Reverse Engineering,* 2004.

[34] L. Aversano, T. Bodhuin, and M. Tortorella, "Assessment and Impact Analysis for Aligning Business Processes and Software Systems", Proc. of ACM Symposium on Applied Computing, SAC 2005.

[35] S. Barros, T. Bodhuin, A. Escudie, J.P. Queille, J.F. Voidrot, "Supporting Impact Analysis: a Semi-Automated Technique and Associated Tool", ICSM, 1995.

[36] J. O'Neal, D. Carver, "Analyzing the Impact of Changing Requirements", ICSM, 2001.

[37] H. Sneed, "Estimating the Costs of software maintenance tasks", ICSM, 1995.

[38] H. Sneed, "Impact Analysis of Maintenance Tasks for a Distributed Object-oriented System", ICSM, 2001.