A BUSINESS PROCESS DRIVEN APPROACH FOR AUTOMATIC GENERATION OF BUSINESS APPLICATIONS

by

Xulin Zhao

A thesis submitted to the Department of Electrical and Computer Engineering

in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

(January, 2011)

Abstract

Business processes describe a set of tasks for accomplishing business objectives of an organization. Business applications automate business processes to improve the productivity of business users. Nowadays, the business environment changes fast due to rapid market growth and technological innovations. Business processes are continuously updated to reflect new business initiatives. Business applications are frequently evolved to add features to meet new requirements and fix defects. Quite often, business processes and business applications evolve independently without direct reference to each other. Over time, it becomes more and more challenging to maintain the consistency between a business application and the corresponding business processes. Moreover, the existing development approaches rely on software developers' craftsmanship to design and implement business applications. Such a development paradigm is inefficient and leads to inconsistency between business processes and business applications.

To facilitate the alignment between business applications and business processes, we present an approach that automatically generates software architecture and code skeletons of business applications from business processes. We identify architectural components from business processes by analyzing dependencies among tasks. To verify the achievement of quality requirements, we extend a set of existing product metrics to automatically evaluate the quality of the generated software architecture designs. Eventually, we apply refactoring strategies, such as software architectural styles or design patterns, to optimize the generated software architecture designs and resolve identified quality problems. Moreover, we also present an approach to automatically refine software architecture to design models and code skeletons of business applications. The effectiveness of our proposed approach is illustrated through case studies.

Acknowledgements

First of all, I would like to express my gratefully gratitude to my supervisor Dr. Ying Zou. She has all the traits of an excellent research supervisor. Through her guidance and encouragement, I have accomplished what I have done to date. I am grateful for the constructive comments and suggestions in research that she provided.

Next, I would like to thank my committee members: Dr. Tzerpos, Dr. Dean, Dr. Hassanein, and Dr. Yousefi, for their valuable comments and feedback on this thesis.

To members of the Software Reengineering Research Group (SERG) laboratory: Qi Zhang, Alex Hung, Rongchao Chen, Jeff Beiko, Hua Xiao, Jin Guo, Derek Foo, Brian Chan, Lionel Marks, Mohamed Al Guindy, Lili Barbour, Ran Tang, Kobe Yuan, Gehan Kamel, and Bipin Upadhyaya, for their help in my research and for making SERG laboratory a pleasant place to work in. I would also like to acknowledge Tack Tong, Bhadri Madapusi and Jen Hawkins from IBM Canada Ltd., for their expertise and kind assistance in my research.

To all my friends in Kingston, you are the ones who made my stay enjoyable and memorable. Finally, I would like to thank my wife, parents, and sister for their unwavering love and moral support.

Table of Contents

List of Figures

List of Tables

▪ List of Acronyms

ACDC:	Algorithm for Comprehension-Driven Clustering
AIO:	Abstract Interface Object
ALMA:	Architecture Level Modifiability Analysis
API:	Application Programming Interface
BDD:	Business Driven Development
BPEL:	Business Process Execution Language
BPMN:	Business Process Modeling Notation
CLA:	Complete Linkage Algorithm
CRM:	Cusomter Relation Management
DIT:	Depth of Inheritance Tree
DoDAF:	Department of Defense Architecture Framework
FEAF:	Federal Enterprise Architecture Framework
FPA:	Function Point Analysis
FSM:	Functional Size Measurement
GUI:	Graphical User Interface
IDE:	Integrated Development Environment
IDEF:	Integrated Definition Mehtods
LOC:	Line of Content
MB-UID:	Model Based UI Development
MQ:	Modularization Quality
MRS:	Mean Recognition Score
NFR:	Non-Functional Requirement
OCL:	Object Contraint Language
SA:	Software architecture
SLA:	Simple Linkage Algorithm
RCP:	Rich Client Platform
RM-ODP:	Open Distributed Processing-Reference Model
TOGAF:	The Open Group Architecture Framework
UCD:	User Centered Design

UI:	User Interface
UIMS:	UI Management System
UML:	Unified Modeling Language
WCA:	Weighted Combined Algorithm
WIMP:	Windows, Icons, Menus, and Pointing devices
WSC:	WebSphere Commerce

Chapter 1

Introduction

Business processes describe a set of tasks for accomplishing business objectives of an organization. Business applications provide automated support for business users to perform tasks and help improve the productivity of business users. In the business domain, business processes undergo constant changes due to the addition of new business requirements or business process optimization activities (e.g., removing bottlenecks and inconsistency). Business applications are also frequently updated to adapt technological advances and maintain the competitive edge of an organization. Quite often, business processes and business applications evolve independently without direct reference to each other. Over time, it becomes more and more challenging to maintain the consistency between business applications and business processes. As a consequence, it is reported that over 50% of business applications fail to address their corresponding business requirements [29].

1.1 Background

1.1.1 Business Process

Business processes are usually created by business analysts to improve the performance of an organization's businesses. Business processes describe activities, resources, and workers that produce products and services. Such descriptions provide fundamental support for business analysts to identify and remove defects of an organization's businesses. In addition, business processes facilitate business analysts to understand and analyze existing businesses of an organization and to design and test renewed businesses.

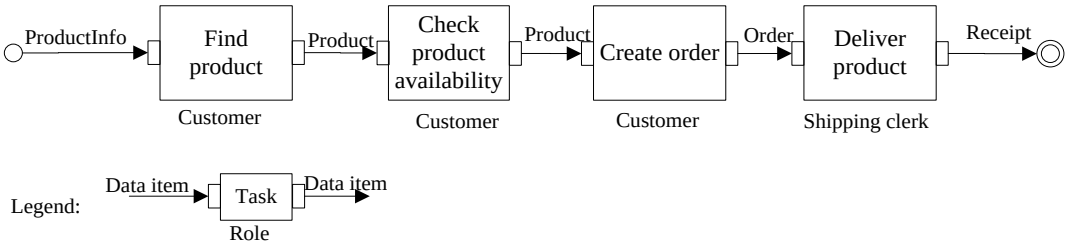


Figure 1-1: An example *Product purchasing* business process

Typically, a business process is composed of a set of interrelated tasks which are joined together by data flows and control flow constructs. Data flows describe inputs into tasks and outputs generated from tasks. Data items are abstract representations of information flowed among tasks. Control flows specify valid execution orders of tasks. Control flow constructs specify the order (e.g., sequential, parallel, or alternative) of the execution of tasks. For instance, Figure 1-1 shows an example business process for purchasing products online. The business process consists of four tasks: *Find product*, *Check product availability*, *Create order*, and *Deliver product*. The task, *Create order*, is performed by customers to specify kinds and quantities of products that they want to purchase. The role, *Customer*, is an abstract representation of customers. The data item, *Order*, captures the output of the task.

1.1.2 Software Architecture

Software architecture is typically the first product in a software development process and captures high level design decisions. In light of this, software architecture is widely used to bridge the gap between business requirements specified in business processes and the implementation of the underlying business application. In the early design stage, software architecture serves as a means for documenting design decisions and enables the evaluation of quality attributes. In subsequent design stages, software architecture is used as a blueprint to guide system construction and composition. Software architecture is often described as a collection of components, connectors,

and constraints. A component captures a unit of functionality in a business application. Connectors describe the interactions and data transitions among components. Constraints specify properties and topologies of components and connectors. Figure 1-2 shows an example that describes the structure of a model-view-controller software architecture. The software architecture is composed of three correlated components, *Model*, *View*, and *Controller*, which are represented as boxes. Connectors among components are displayed as arrows.

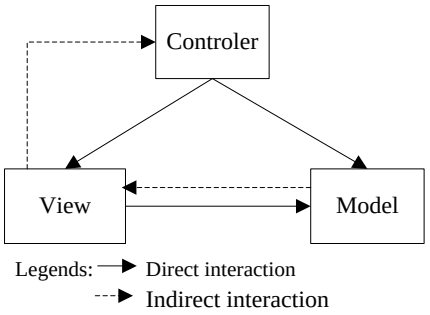


Figure 1-2: Example software architecture

Designing software architecture typically consists of two steps: 1) identify functional components to fulfill functional requirements; and 2) apply design principles to address quality requirements. Functionality of a software system is often decomposed to architectural components in three ways:

1) *Functional decomposition* recursively refines high-level functional requirements into a collection of more concrete functional requirements. Eventually, each software component captures one concrete functional requirement.
2) *Data oriented decomposition* identifies a set of essential data structures from the requirements. Each software component is intended to implement one data structure.

3) *Object oriented design* combines both approaches by forming packages that are composed of data structures and their associated functions. Each package corresponds to one software component.

Software architectural styles capture principles and best practices for designing software architecture. A software architectural style determines the vocabulary of components and connectors for describing the architecture of a family of software systems. Garlan and Shaw [28] and Buschmann et al. [40] summarize typical software architectural styles from different domains and describe the pros and cons of each software architectural style. During a software architecture design process, software architects choose appropriate software architectural styles by analyzing specified requirements and create software architecture using components and connectors defined in chosen software architectural styles.

1.1.3 User Interface (UI) Design

UIs of a business application provide an interactive environment for users to communicate with the business application. The design of UIs is often separated from the design of the software architecture of the business application for two reasons:

1) The design of UIs usually starts earlier than the design of software architecture. UI prototyping is widely used as a requirement elicitation approach in the requirement analysis stage [68][87][124] whereas software architecture is typically designed during the design stage.
2) UIs change more frequently than software architecture. To produce business applications with satisfactory usability, UIs should be designed in conformance with users' characteristics, capabilities, and expectations. However, the number of users of a business application is vast. Users often have diversified backgrounds. Therefore, it is difficult to

determine representative users of a business application. Even when representative users are determined, users may not be clear about their expectations and designers also have difficulty in understanding users' expectations [10]. Therefore, prototypes of UIs need to be iteratively tested and redesigned throughout the development process of a business application. Software architecture represents the gross structure of the business application and therefore is expensive to change once the development process launches. Once the design of software architecture is completed, little or no changes on the software architecture are expected in the subsequent development stages.

UI design patterns capture principles and best practices of UI design. Two types of patterns are commonly used during the design of UIs: task patterns and usability patterns. A task pattern provides a template for implementing the UIs of a family of tasks. For instance, the *Find* task pattern [123] defines widgets for implementing tasks that encapsulate data inquiry operations. Usability patterns describe reusable solutions to address usability problems. For instance, the *Forgiving Format* pattern [69] helps to reduce error rates of end-users.

1.2 Problem Definition

Aligning business applications with business requirements has been a critical issue for organizations for almost three decades [58]. In the rapid changing business environment, business processes undergo constant changes and organizations need to swiftly adapt their business applications to maintain their competitive edge. However, existing software development processes suffer from a lack of agility to keep up with the pace of business changes. Mainstream design approaches [3][20][81] rely on software architects' craftsmanship to derive software architecture from business requirements based on their past experience. However, real-world, large scale business applications need to satisfy hundreds of business requirements that contain

numerous intrinsic dependencies [116]. Quite often, a manual design approach is inefficient and leads to inconsistency between business requirements and business applications. Moreover, limited techniques are available to evaluate the achievement of quality requirements in software architecture without the detailed designs and implementations [83]. In software industry, the degree to which a software system addresses quality requirements is often determined by testing implementations of the software system. The disadvantage of such an evaluation approach is that a large amount of effort is wasted in building system implementations without conformance with quality requirements. In addition, limited approaches are available to ensure that problems identified in the testing processes can be correctly addressed in the subsequent redesign activities.

The usability of UIs of business applications is crucial for the survival of a business and its success. The key to attract customers is applications with high quality content, ease of use, quick response and frequent updates. However, usability engineering techniques are generally not used in practice because they are considered overwhelmingly complex, time-consuming, and expensive [63]. Consequently, studies indicate that the majority of business applications suffer from usability problems [2][98]. For example, applications often provide more functionalities than necessary. It may not be obvious for users to navigate through such UIs in order to fulfill business tasks. Approaches have been proposed for automating the design and implementation of UIs [61][88]. However, these approaches create the designs of the UIs from the perspectives of the software development domain. Without the consideration of users' capabilities and expectations, the UIs might not satisfy the usage patterns of business users. Therefore, it is crucial to provide an approach that incorporates users' requirements into the UI design process to support intensive interactions between users and business applications.

1.3 Challenges

To facilitate the alignment between business requirements and business applications, we present an approach that automatically generates software architecture, UI designs, and code skeletons from business processes. In this thesis, we aim to address the following challenges:

1) *derive software architecture from business processes.* To fulfill functional requirements, we need to identify the primary functionalities from business processes and model them as architectural components. Nevertheless, functionalities are specified by task names which are typically descriptive texts, such as *Create customer order*. Due to the lack of details, it becomes infeasible to identify primary functionalities by analyzing task names. Moreover, modularity plays a pervasive role in early design stages and determines the achievement of a wide spectrum of quality attributes including understandability, maintainability, and testability [121]. It is shown that modularity is primarily achieved during the creation of architectural components of a software system [23]. Functionalities can be distributed to architectural components in a variety of ways. Different distribution methods result in software architectures with different modularity. Hence, we need an approach to compare different distribution methods and choose an optimal one. However, the modularization of a software system can be considered as partitioning the graph that is composed of functionalities in the software system and their dependencies. Such a problem is known to be NP-hard. Therefore, it is not practical to exhaustively compare all possible modularization results.
2) *evaluate the quality of software architecture.* It is widely recognized that quality requirements are primarily addressed during the design of software architecture. A variety of product and process metrics are proposed in the literature to evaluate quality attributes

of software systems. The majority of such metrics, such as *line of code (LOC)* and *depth of inheritance tree (DIT)* of classes, require detailed designs or source code of a software system. However, such detail information is not available at the architectural level. Therefore, these metrics cannot be directly used to assess software architecture.

3) *generate UIs from business processes.* A necessary step in generating UIs is to determine functionalities of UIs in order to align business requirements specified in business processes with functionalities in UIs. Functionalities are specified by tasks in business processes. Nevertheless, the granularity of tasks generally varies considerably. For example, primitive tasks, such as *Add product to shopping cart* and *Submit order* represent the lowest level of detail. Other tasks may convey functional requirements that can be composed of a sequence of primitive tasks. Hence, it is not feasible to produce a UI window for every individual task. In addition, usability of UIs is crucial to the success of a business application. The focus of usability evaluation is to determine optimal layouts of UIs [96] that are computed by analyzing properties (e.g., size and position) of widgets in UIs. However, business processes are created by business analysts in the business domain and do not contain artifacts for implementing functionalities or UIs of the business application. To generate UIs with specified usability features, an approach to identify widgets of UIs from business processes is in demand.

1.4 Solution

Business processes describe daily work rhythms within an organization and encapsulate functional requirements of business applications. Tasks and data items in business processes convey low level details of functionalities and data structures. In this thesis, we strive to address

the challenges in aligning business applications with business requirements by analyzing business processes. More specifically, the major contributions of our research include:

1) *automatically derive the functional software architecture of a business application from business processes.* We group functionally similar tasks to specify functionalities for architectural components. Functionalities of a business application are specified by tasks in business processes. Tasks for fulfilling the same functionality are tightly coupled to each other. Hence, we analyze tasks and data items defined in business processes to identify the internal structures of components and connectors. Clustering is a technique to group entities in such a way that entities within a group are more similar to each other than entities in different groups. In software engineering, clustering techniques are commonly used to modularize a software system by grouping functionally similar artifacts (e.g., procedures or objects). In our work, we apply clustering algorithms to produce groups of functionally similar tasks.
2) *evaluate quality attributes of software architecture by analyzing properties of tasks and data items in business processes*. In our generated software architecture, the functionality of a component is specified by a set of tasks, data items and their interactions derived from business processes. We extend a collection of existing product metrics using the properties of tasks and data items to assess the quality attributes of the software architecture. For instance, we evaluate the dependency strength among architectural components by analyzing task transitions across architectural components.
3) *automatically generate UIs with specified usability features from business processes.* Instead of considering every task as delivering individual functional requirements, we analyze data dependencies among tasks and data items in the business processes to group

tasks into functional segments. A functional segment captures a functional requirement to be implemented by a UI window. We propose a set of segmentation rules that group relevant tasks and data into meaningful functional segments. In addition, we use a collection of intermediate UI models with increasing details to bridge the gap between business processes and UI code skeletons. We define model transformation rules to automatically generate UI models and code skeletons. To generate UIs with desired usability features, we interpret usability requirements to model constraints and use the identified model constraints to guide the generation of UI models and code skeletons.

1.5 Thesis Overview

The remaining chapters of this thesis are organized as follows:

Chapter 2 introduces existing work related to our approach.

Chapter 3 gives a brief overview of our approach for aligning business applications with business requirements.

Chapter 4 describes our method for generating functional software architecture designs from business processes.

Chapter 5 presents our approach for optimizing the quality of the generated software architecture designs.

Chapter 6 introduces our approach for generating UIs from business processes.

Chapter 7 depicts our approach for incorporating usability requirements into the UI generation process.

Chapter 8 gives a brief introduction of our prototype tools.

Chapter 9 presents case studies that evaluate the feasibility and effectiveness of our approach.

Chapter 10 concludes this thesis and proposes our future work.

Chapter 2

Related Work

In this chapter, we give a brief overview of existing work related to our research. More specifically, our work is related to three major areas, namely, software architecture generation, software architecture evaluation and optimization, and UI development and optimization.

2.1 Software Architecture Generation

2.1.1 Bridging the Gap between Requirements and Software Architecture

Although the conceptual differences between requirements and software architecture are well understood and articulated, evolving and elaborating requirements to a viable software architecture satisfying those requirements remains a challenging task in software engineering. Mainstream design approaches [43][137] rely on software architects to decompose requirements to architectural components by analyzing functionalities, data structures, and objects specified in requirements. However, business requirements evolve frequently due to rapid changes in the business environment. Manual software architecture design approaches are inefficient in addressing rapid changing business requirements and lead to inconsistency between business requirements and business applications. A variety of approaches have been presented to address changes in requirements during software architecture design. Some researchers try to address the problem by introducing synchronization steps into the software architecture design process. For example, Nuseibeh [8] adapts the spiral life cycle model to incrementally develop software architecture and synchronize software architecture with requirements. Greunbacher et al. [112] enhance Nuseibeh's approach by providing an intermediate model to assist the synchronization of software architecture and requirements. However, little support is available to assist software

architects deriving architectural components from requirements in these approaches. Other researchers present pattern based approaches to guide software architects deriving software architecture from requirements. Jackson [95] uses problem frames to record typical system partitions in different domains. The created problem frames are then used to guide software architects creating software architecture from requirements. Fowler [89] presents a collection of analysis patterns to help software architects analyzing requirements and constructing architectural components. An analysis pattern reflects a conceptual structure (e.g., Customer and Accounting) of business processes. Problem frames and analysis patterns keep knowledge in requirement analysis and provide helpful guidance for software architects to construct software architecture. However, dramatic effort is required to build collections of problem frames and analysis patterns before these approaches can be used. In addition, problem frames and analysis patterns are created and used by software architects based on their knowledge and past experience. Hence, the effectiveness of these approaches depends heavily on the expertise of software architects. In our work, we present an approach that automatically generates software architecture from business processes. When business processes are changed, we can easily reconcile business processes and software architecture with the assistance of our proposed approach.

2.1.2 Software Clustering

Clustering is a technique for grouping objects in such a way that intra-group dependency is high and inter-group dependency is low. Clustering is widely used in a variety of disciplines (e.g., image processing, information retrieval, and pattern recognition) to address a wide spectrum of problems. In the software engineering domain, clustering is often used to facilitate the modularization of software systems. For instance, Neighbors [75] presented a clustering approach to form subsystems by grouping tightly coupled components. Coupling among components are

evaluated by analyzing static and dynamic interconnections among components. Anquetil and Lethbridge [102] proposed an approach to extract subsystems by grouping source files of a software system. The authors assess dependencies among source files by comparing names of source files. Hutchens and Basili [35] performed clustering to identify subsystems based on data interactions among procedures. Tzerpos and Holt presented a collection of subsystem patterns and proposed an ACDC algorithm to produce decompositions of a software system by applying these subsystem patterns [133]. Mancoridis et al. [126] applied genetic algorithms to partition software systems and optimize modularity of software systems. Different from our proposed approach, these approaches take existing code or documentations as input and produce software architecture designs for legacy systems. Our proposed approach generates software architecture designs for new business applications from requirements encapsulated in business processes.

Lung et al. [26] use clustering techniques to decompose requirements to subsystems. The authors manually identify dependencies among requirements and apply the hierarchical agglomerative clustering method to produce a hierarchy of clusters. However, the approach does not consider quality requirements. In our approach, we reuse business processes in the business domain as inputs and automatically generate software architecture designs with desired quality.

2.1.3 Business Driven Development (BDD)

Business requirements change fast due to rapid market growth and technological advances. Existing software development processes lack the agility required to keep up with market trends and competition. BDD improves the agility of business application development by using business processes to model business requirements and guide the development process. Mitra [128] illuminated essential tenets of BDD and described an overview of steps involved in a BDD process. However, little guidance is provided for software architects to convert business processes

to business applications. In [57], Koehler et al. classified business processes to analysis models and design models. Analysis models are created by business analysts to describe activities for accomplishing business requirements. Design models are derived from analysis models to support the development and integration of business applications. The presented approach focuses on transformations among business processes and relies on software architects to map business processes to components (or services). Inaganti and Behara [125] presented a service identification approach to map business processes to components (or services) of business applications. In the presented approach, software architects manually adjust and map tasks in business processes to establish one-to-one mappings between tasks and existing business applications. Such a manual approach works well for small scale business applications but is inefficient for large scale business applications. Zimmermann et al. [109] proposed a semi-automatic approach that uses reusable architectural decision models to bridge the gap between business processes and business applications. Architectural decision models are represented as decision trees. Components can be identified by trimming created decision trees. However, decision trees for large business applications are quite large and difficult to understand and use. Our proposed approach automatically generates software architecture designs from business processes. Results of our proposed approach can be used as blueprints to guide the subsequent design and development activities.

2.2 Software Architecture Evaluation and Optimization

2.2.1 Software Architecture Analysis

Software architecture analysis is essential to determine the extent to which a software architecture design meets specified quality requirements. However, such an analysis is difficult for two reasons: 1) high level quality requirements are often tentative and contradictory and 2) it is

difficult to measure quality attributes due to the lack of low level details. A great deal of effort has been devoted to help software architects clarifying quality requirements. For instance, standards, such as ISO 9126 [54] and ISO/IEC 14598 [51], are presented to guide the specification of quality requirements. Mylopoulos et al. [60] proposed a non-functional requirement (NFR) framework that prompts the use of soft-goals to represent quality requirements and guide software design activities. The NFR framework is used and validated in many different systems [82][84][85][140]. In our work, we also use the NFR framework to specify application specific quality requirements.

The fitness of a software architecture design is often evaluated by two categories of approaches: quantitative measurement and qualitative analysis. The software metric community has presented numerous product metrics [45][73][104][126] to quantify the quality of a software design. However, the majority of these metrics requires code level artifacts or detailed designs. Due to the lack of such low level details, quantitative measurement approaches are rarely used in the design of software architecture. Recent efforts toward software architecture evaluation are focused on using scenarios [83][117][118] to qualitatively detect pitfalls of software architecture designs. However, these techniques rely on evaluators' manual work to evaluate software architecture designs and are expensive in an iterative design process. In our generated software architecture, components are described using data items and tasks derived from business processes. We adapt existing software metrics to automatically assess software architecture designs using properties of tasks and data items. Our approach reduces the cost of designing software architecture by automatically generating software architecture with desired quality attributes from business processes.

2.2.2 Software Architectural Styles and Frameworks

Software architectural styles provide reusable solutions to resolve problems during software architecture design. A software architectural style defines the vocabulary of components and connectors for describing the software architecture of a family of software systems. Garlan and Shaw [28] and Buschmann [40] summarized common paradigms in software architecture design and presented a number of software architectural styles. Software architectural styles provide significant support for software architects to address quality requirements. However, little support is provided to assist software architects comparing and choosing software architectural styles with respect to requirements. To address this problem, Klein et al. [99] correlate software architectural styles to quality models to assist software architects qualitatively compare and choose software architectural styles. Chung et al. [82] present a goal model based approach to qualitatively evaluate contributions of various software architectural styles to specified quality requirements. These approaches are demonstrated to be effective in addressing quality requirements. However, little effort is devoted to help software architects modularizing functionalities in requirement specifications. Our proposed approach automatically generates software architecture designs with desirable quality attributes from business processes. We analyze dependencies among tasks to derive architectural components from business processes and apply software architectural styles to optimize the quality of the generated software architecture.

Software architecture frameworks provide structured and systematic approaches for designing software architecture. Software architecture frameworks use views to represent different perspectives of a software architecture design and provide solutions to a wide spectrum of software architecture design problems. Software architecture frameworks, such as the Open Distributed Processing–Reference Model (RM-ODP) [52] and the 4+1 View Model [113], emphasize on software architecture design and provide support for software architecture

modeling and analysis. The Open Group Architecture Framework (TOGAF) [132], the Department of Defense Architecture Framework (DoDAF) [130], and the Federal Enterprise Architecture Framework (FEAF) [5] focus on enterprise architecture design and provide support for facilitating the definition, understanding and standardization of software architecture design activities in an enterprise. A major problem of software architecture frameworks is that they put little attention on the description of design rationales. Hence, it is difficult to verify result software architecture designs. In addition, designing software architecture is still costly since software architects need to manually derive architectural components from requirements in these frameworks. In our proposed approach, we combine clustering techniques and software architectural styles to automatically generate software architecture with desired quality attributes. Design rationales are encapsulated in clustering algorithms and transformation rules in the generation process. Hence, our proposed approach can help to reduce the cost of software architecture design and provides support for the analysis and verification of generated software architecture designs.

2.3 UI Development and Optimization

2.3.1 UI Development Approaches

UIs offer an environment for people to communicate with software applications. A graphical user interface (or GUI) is a kind of UIs that takes advantage of graphical images and widgets to make programs easier to use. Widgets are graphical elements that are designed following the WIMP (windows, icons, menus and pointing devices) paradigm [88]. Common widgets include windows, menus, buttons, boxes, scroll bars and icons. In our work, we leverage widgets provided by the Eclipse Rich Client Platform (RCP) to implement UIs.

UIs are hard to develop by nature [6]. One of the major problems is that UI developers cannot accurately comprehend domain tasks and end users' needs. User centered design (UCD) and task analysis are two ways that are commonly used to improve the understanding of domain tasks and end users' needs. UCD is a design approach in which end users are actively involved in the design and evaluation of UIs [18]. A typical UCD process starts from a thorough understanding of end users and the selection of a group of representative users. However, such a goal is often difficult to achieve in practice. End users of a business application may be composed of people with diversified backgrounds and scatter over the world. In addition, the body of end users of a business application is usually huge and cannot be fully determined before the development of the business application. Therefore, UCD is not often used in practice [79]. Task analysis is a process to determine the way in which interactive tasks are accomplished by end users. Task analysis typically results in a task model that describes goals of end users, elements for accomplishing interactive tasks, and temporal relationships among interactive tasks. Limbourg et al. [115] presented a review of common task analysis approaches. However, these task analysis approaches rely on developers to manually create task models. Real life, large scale business applications consist of hundreds or thousands of interrelated tasks. A manual task analysis approach is inefficient and error prone in understanding end users' needs for these business applications. In our work, we design heuristic rules to automatically analyze business processes and derive task models. Our proposed approach improves the productivity and reduces the cost of UI development.

Another problem is that UIs are complex to develop. Traditionally, UI development is treated as a creative activity rather than a systematic engineering process. In many interactive systems, UIs are the major reason of complexity. Traditional UI development approaches are time-

consuming and expensive. Studies show that about 50% of the development time is spent in the developent of UIs [11]. To improve the efficiency of UI development, UI design tools and generation approaches are presented. There are two types of UI design tools: UI builders and UI management systems (UIMSs). UI builders are provided in many mainstream integrated development environments (IDEs) [105][135] and provide toolkits to assist developers to design the appearance of UIs. Little support is provided for constructing the controller between UIs and the remainder of an application in a UI builder. UIMSs [31][86] provide means for constructing both the appearance and the controller of an application. However, UIMSs require developers to construct addition models beyond normal programming languages and are considered cumbersome. Hence, UIMSs are rarely used in industrial environment. Numerous UI generation approaches have also been proposed. Typical automatic UI generation approaches include model based user interface development (MB-UID) [46] and UML based UI prototyping. These approaches can automatically generate UIs from pre-defined UI models. However, the creation of initial UI models and the usability evaluation of generated UIs are conducted by end users and domain experts. Such human centric UI modeling and evaluation approaches are still inefficient. In our work, we automatically generate UI models from business processes and transform UI models to UI code skeletons. Furthermore, we apply UI design guidelines and patterns to guide model transformations to generate UIs in conformance with end users' expectations.

2.3.2 Usability Evaluation

Usability measures the effort required to understand and use an interactive application. The success of an interactive application has heavy dependence on the usability of its UIs [33]. Surveys [91][114] show that UIs with poor usability are the major causes of frustration.

A great deal of effort has been devoted to the development of UIs with desired usability. Gajos et. al. treated UI generation as an optimization problem and presented a rendering algorithm to generate UIs with the minimal user effort [78]. Sottet et. al. proposed an approach for preserving usability during UI adaption [76]. The presented approach stands for the use of advisor tools and abstracts model transformations to two types of transformation models: predictive models and descriptive models. Predictive models can reform UI models without ambiguity (e.g. $f(x)=x+2$). Descriptive models are qualifiers (e.g. $f(x)>x$). The reservation of usability is achieved by well designed transformation models. In our work, we incorporate usability requirements into model based UI design and automatically generate UI models and code skeletons from business processes. In addition, we leverage usability goal graphs to represent users' usability expectations and guide the UI generation process. Eventually, we can automatically generate UIs with desired usability features from business processes.

A variety of usability evaluation methods have also been presented to identify usability problems in designed UIs. Ivory and Hearst [101] classified usability evaluation methods to five categories:

1) *Usability testing*: Evaluators determine usability problems by observing users' interactions with UIs. User testing approaches [1][67][71] provide trusty evaluation of the usability of UIs since evaluators assess usability through samples of real users. However, it is difficult to select a correct sample of real users and the establishment of a testing environment is expensive. In addition, these approaches can only be applied after UIs or prototypes of UIs are developed.
2) *Usability inspection*: Evaluators examine UIs with respect to their conformance to predefined criteria, guidelines, or heuristics. Inspection approaches [39][66][138] do not

involve users nor require any equipments. Hence, inspection approaches can be applied from early stages of UI design and are widely used in industrial environments. However, these approaches are subjective and rely heavily on skills of evaluators.

3) *Usability inquiry*: Evaluators identify usability problems using inquiry methods, such as interviews, surveys, and questionnaires, to gather feedback from users. Inquiry approaches [12][56][131] can only be applied after UIs or prototypes of UIs are developed too. The cost of inquiry approaches varies based on the number of users involved in the evaluation process.
4) *Analytical modeling*: Evaluators predict usability of UIs by analyzing user models and interface models. A number of modeling approaches [13][59][100] have been presented to model user preferences and actions, interactive tasks, and structures of UIs. Analytical modeling approaches can be applied in early stages of UI design. Nevertheless, considerable effort is required to construct models.
5) *Simulation*: Evaluators mimic interactions between users and UIs in artificial settings and report results of the interactions. Simulation approaches [37][44] provide automated support for usability evaluation. Evaluators can run simulation with different parameters and observe responses of UIs. However, the establishment of the simulation environment is costly and time-consuming.

In our work, we present an approach that automatically generates UIs from business processes. Moreover, we create goal graphs to map users' preferences to constraints on UI models and automatically evaluate the usability of generated UIs.

2.3.3 UI Design Guidelines and Patterns

Capturing UI design knowledge is essential to help developers to make right design decisions during UI design. UI design guidelines and patterns are two commonly used approaches for recording and consolidating UI design knowledge. UI design guidelines describe UI design principles, rules and recommendations for improving usability [22]. UI design guidelines can be found in many different formats, ranging from books [27][62][64][139], organizational style guides, to ISO standards [55]. These guidelines cover all aspects of UI design, such as features, styles, look and feel, and code structure, on various platforms. UI guidelines are usually applied in two steps: 1) a collection of UI guidelines are chosen by analyzing requirements of a business application and 2) interpret chosen UI guidelines to application specific rules or constraints. The accomplishment of both of the two steps involves numerous subjective judgments. Moreover, applying UI design guidelines is time-consuming and labor intensive due to the large amount of available UI guidelines and the vagueness inherent in abstract descriptions of UI design guidelines.

UI design patterns are presented to resolve the problems of UI guidelines. UI design patterns record reusable solutions of recurring problems during UI design. Beside design knowledge, UI design patterns also provide descriptions of design problems, rationales, and typical examples so that designers can apply patterns accurately. In addition, a pattern may be implemented by a number of other patterns. As a result, UI patterns can be organized to a hierarchy structure similar to class diagrams. Designers can easily find required patterns following relations between them. A variety of UI design patterns and pattern based UI design approaches are presented in literature [32][69][92]. As shown in these approaches, UI design patterns provide better guidance for the design of UIs than UI design guidelines do. Nevertheless, UI design patterns lack definition of low level details of UI designs such as widget selections and window layouts. For example, a

Searching [92] pattern defines that a *Find* task will be implemented by input fields, buttons, and output fields. But an input field can have multiple implementations such as a text field, a combo box or a check box. The definition of the *Searching* pattern doesn't have formal description about how it is implemented on a specific platform. Therefore, UI design patterns cannot totally replace UI design guidelines. In our work, we use both UI design guidelines and UI design patterns to guide the generation of UIs. We apply UI design patterns to determine contents of windows and use UI design guidelines to guide the selection of widgets and window layouts.

2.4 Summary

In this chapter, we review existing work related to software architecture generation, software architecture evaluation and optimization, and UI development and optimization. In addition, we also highlight our contributions in these areas by comparing our approach with existing work.

Chapter 3

Overview of Our Approach

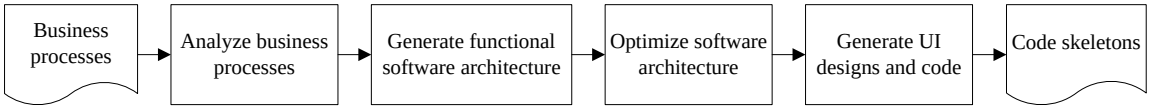


Figure 3-1: Business process driven business application generation

Figure 3-1 gives an overview of our approach that generates business applications from business processes. The generation process consists of four major steps: 1) analyzing business processes to extract the artifacts relevant to tasks and their dependencies; 2) generating functional software architecture to fulfill functional requirements specified in business processes; 3) optimizing the generated software architecture to address quality requirements; and 4) generating UI designs and code skeletons from software architecture.

3.1 Analyzing Business Processes

Business processes are typically described with diagrammatic modeling techniques. Such techniques allow business analysts to discuss and validate business processes with users who are often not willing to invest their time in understanding complex representations. A variety of techniques have been proposed to model business processes. BPM languages, such as integrated definition methods (IDEF) [119], business process modeling notation (BPMN) [14], and business process execution language (BPEL) [16], define standard notations to represent entities in business processes. To provide a generic solution to handle business processes described in different languages, we create a meta-model to capture commonalities among these BPM languages as shown in Figure 3-4. Figure 3-2 illustrates a business process for purchasing books

online. A task represents a primitive activity that accomplishes a business objective. The specification of a task consists of an action and a number of pins that represent data flows among tasks. The action of a task is often designated by the name of the task using a descriptive text, such as *Sign in account* and *Find book*. We classify tasks into three categories according to their functionalities:

- *Human tasks* are interacted with users to take a user's inputs and display the results of a user's requests. The roles of human tasks are often human, such as *Customer*;
- *Automatic tasks* carry out the computation. The role of automatic tasks is often software components or devices.
- *Data-access tasks* are special case of automatic tasks and encapsulate the operations for database manipulation such as *Insert*, *Update*, *Query*, and *Delete*.

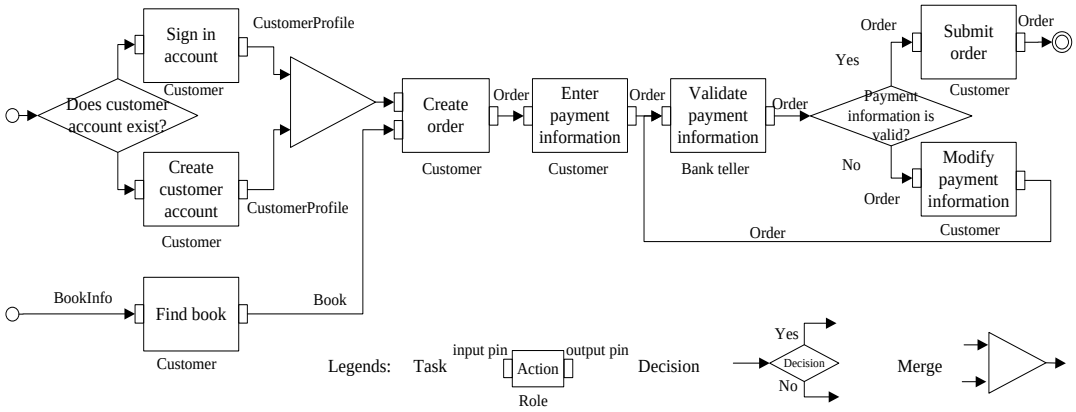


Figure 3-2: The *Purchase book* business process

Input pins hold input data used by the task. Output pins hold output data of the task. For instance, the task, *Find book*, receives the data item, *BookInfo*, from its input pin and returns the data item, *Book*, through its output pin. A data model is used to describe structures of data items and dependencies among them. A data item contains a set of attributes of various types, such as

primitive types (e.g., String, Integer, or Float) or user defined types. For instance, Figure 3-3 shows the definition of the data item, *Order* (shown in Figure 3-2), and the attributes of the data item. A data item can also inherit attributes from other data items.

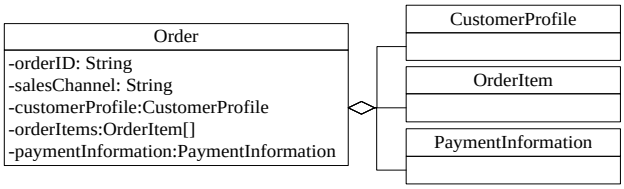


Figure 3-3: The definition of the data item, *Order*

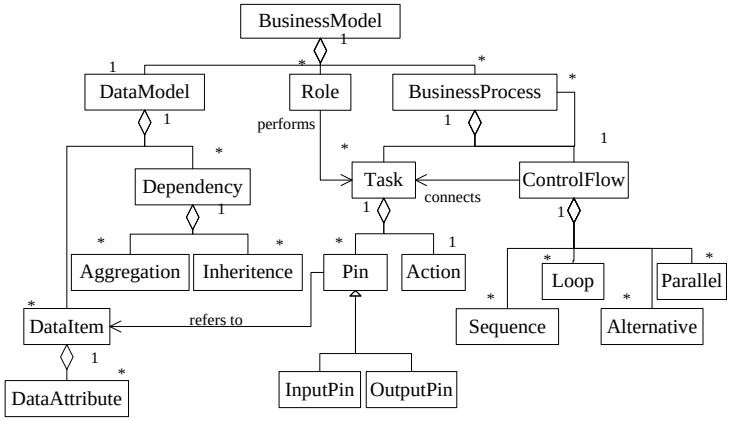


Figure 3-4: The meta-model for business processes

A data item can be transferred between tasks or between a task and an external entity (e.g., user or device) in the operational environment. An inter-task data interaction happens when the output data item of a task is served as the input data item of another task. Environment-task data interaction occurs when a data item is passed between a task and the operational environment of the task. A variety of data items can be exchanged between tasks and the operational environment. For instance, a task might accept requests from users, send a surface mail to users, retrieve data from an external database, or download data from an external server.

Control flows specify execution orders of tasks. The control flow of a business process can contain four types of control flow constructs: *Sequence*, *Loop*, *Alternative*, and *Parallel*. Tasks in a sequence are accomplished one after another. A loop is composed of a set of tasks executed iteratively. Alternative allows one execution path to be chosen from multiple possible execution paths with respect to the output of the preceding task, the value of a data item, or the result of a user decision. A parallel defines multiple execution paths that can be executed simultaneously.

Roles specify subjects that perform tasks in order to accomplish business objectives. A role can be a software component, a device, or an abstract representation of a category of users that have common responsibilities. Responsibilities or services of a role are specified by the associated tasks. For example shown in Figure 3-2, the role, *Customer*, performs all tasks in the example business process except the task, *Validate payment information*.

3.2 Generating Functional Software Architecture

In a business process specification, tasks specify functionalities of the underlying business application and data items designate data structures of the business application. To fully use the information encapsulated in business processes, we apply clustering techniques to analyze data interactions specified in the business processes and produce clusters of data items with strong dependencies. Each cluster aggregates data items to data groups with different levels of granularity and inter-dependencies. We choose a set of data groups with satisfactory modularity to form sketches of architectural components. To describe functionalities for components, we assign tasks specified in business processes to data groups by analyzing the dependency strength between tasks and data groups. Eventually, we produce a functional software architecture that is described as a set of connected software components. The functionality of a software component is described using data items and tasks in business processes. Our proposed approach provides

automated support to help software architects to analyze dependencies among tasks and improves the consistency between business requirements and business applications.

3.3 Optimizing Software Architecture

To generate software architecture with desired quality attributes, we need to evaluate the quality of generated software architecture and provide mechanisms to resolve identified quality problems.

Quality attributes can be measured by software product metrics which require the details of software artifacts, such as lines of code and the number of method calls. However, these software artifacts are usually not available in the software architecture design stage. In our generated software architecture, components are described using data items and tasks derived from business processes. Definitions of data items encapsulate data structures of components. Tasks and transitions among tasks convey functionalities as well as control and data dependencies among components. Moreover, dependencies among tasks and roles indicate the interactions between various users and components. With the information embedded in tasks and data items, we extend existing product metrics to assess quality attributes of software architecture.

When quality problems are identified in the evaluation process, we choose appropriate software architectural styles or design patterns to optimize the generated software architecture designs. Mainstream design approaches [20][81] rely on software architects to manually restructure the software architecture to apply software architectural styles or design patterns. Such a manual optimization approach is inefficient and error prone. In our work, we define transformation rules to describe refactoring operations and perform the defined transformation rules to automatically restructure software architecture.

3.4 Generating UI Designs and Code Skeletons

To reduce the development and maintenance effort of a business application, we present an approach that automatically generates designs and code skeletons from business processes. However, business processes do not contain enough information to produce designs and code. For example, a business process does not describe the threading model or the database schemas to be used in the implementation. We use intermediate models with increasing details towards the final code to bridge the gap between business processes and code. In addition, we use model transformation techniques to automatically generate and synchronize intermediate models and code. In this thesis, we use the generation of UI designs and code skeletons as an example to demonstrate our approach.

3.5 Summary

In this chapter, we give an overview of the major steps in our approach and discuss how we address challenges presented in Section 1.3.

Chapter 4

Generating Functional Software Architecture

To facilitate the alignment of business requirements with business applications, we propose an approach that automatically generates functional software architecture from business processes. We apply clustering algorithms to automatically group functionally similar tasks and distribute functionalities to architectural components. In the following sections, we first give an overview of clustering algorithms and then describe our approach for generating architectural components and their interactions.

4.1 An Overview of Clustering Algorithms

Clustering algorithms group entities with strong dependencies to form clusters. Similarity measures evaluate the strength of dependencies between entities by assessing the number of common features or connections between entities. For example, the unbiased Ellenberg measure [107] evaluates the degree of similarity between components by calculating the percentage of common features shared by both components, such as data members, previous components, and subsequent components.

$$MQ = \begin{cases} \dfrac{\sum_{i=1}^{k} A_i}{k} - \dfrac{\sum_{i,j=1}^{k} E_{i,j}}{k(k-1)/2} & k > 1 \\ A_1 & k = 1 \end{cases} \qquad A_i = \frac{\mu_i}{N_i^2} \qquad E_{i,j} = \begin{cases} 0 & i = j \\ \dfrac{\varepsilon_{i,j}}{2N_iN_j} & i \neq j \end{cases} \tag{4-1}$$

k is the number of clusters. A_i assesses to intra-connectivity and $E_{i,j}$ evaluates inter-connectivity. μ_i is the sum of connections between entities within the cluster C_i. $\varepsilon_{i,j}$ is the sum of connections between entities in the cluster C_i and entities in the cluster C_j. N_i and N_j are the number of entities in the cluster C_i and the cluster C_j respectively.

Partitional algorithms [127] and hierarchical algorithms [24][25][35][102][108] are two categories of commonly used clustering algorithms to group entities using similarity measures.

More specifically, partitional algorithms define heuristics to optimize a set of initial clusters which can be a set of randomly grouped entities or the result of other clustering algorithms. For example, Mancoridis et al. [126] generate initial clusters by randomly grouping a set of entities, and then apply hill climbing algorithms and genetic algorithms to optimize the initial clusters using the modularization quality (MQ) metric. The MQ metric measures the cohesion and coupling of software components by evaluating inter-connectivity between components and intra-connectivity within components. The definition of MQ is shown in equation (4-1). In general, the value of the MQ metric is bounded between -1 and 1. -1 means that a software system has no cohesion and 1 means that the software system has no coupling. Neither of the extreme values can be achieved in practical software systems. The exact range of the MQ value is determined by the intrinsic dependencies within a software system. If requirements have strong dependencies to each other, the MQ value tends close to -1. If requirements can be divided into multiple independent groups, the MQ value tends close to 1.

Agglomerative algorithms and divisive algorithms are hierarchical algorithms which form a hierarchy of clusters. Agglomerative algorithms are bottom-up approaches that generate clusters by grouping entities in the lowest level of the granularity and moving up to coarser grained entities in a stepwise fashion. Divisive algorithms are top-down approaches that produce clusters by gradually dividing the coarser grained entities into more fine grained entities. Using an agglomerative algorithm, the most similar pair of entities is selected to form a new cluster. When more than two entities have the same similarity, the algorithm makes arbitrary decisions by randomly merging two entities. However, arbitrary decisions are harmful to clustering quality and should be avoided in the clustering process [4]. The weighted combined algorithm (WCA) [107] is used to reduce arbitrary decisions and information loss. A study [108] shows that clustering

results of WCA are more consistent with expert decompositions than other hierarchical algorithms. Therefore, we choose to use WCA to produce software architecture in conformance with the one designed by software architects.

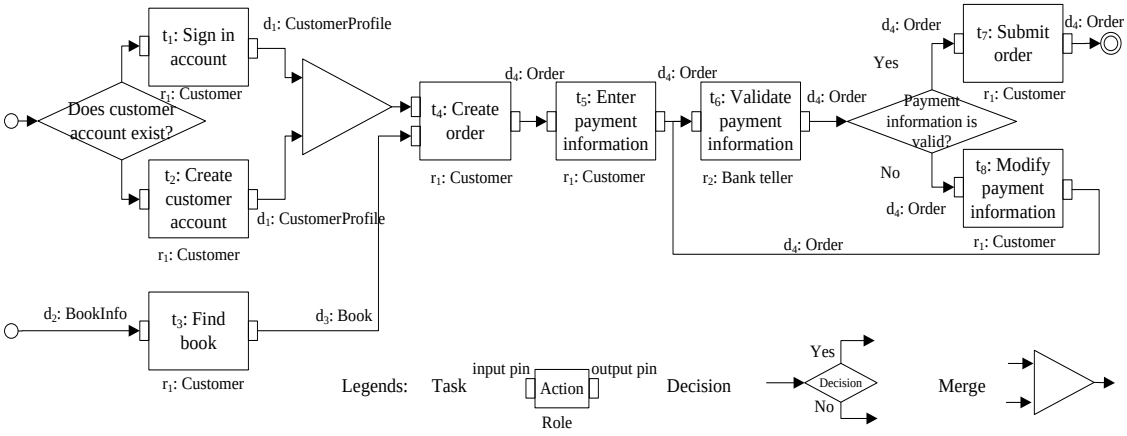


Figure 4-1: The *p_1: Purchase book* business process

4.2 Generating Components

In this section, we describe our approach for generating components. We first identify data groups that form data structures of components and then assign tasks to components to specify operations on data structures.

4.2.1 Grouping Data Items

To improve the cohesion within a software component, we strive to identify strongly inter-dependent data items to form the data structure used in a component. To analyze the dependencies among data items, we create a data dependency graph to analyze data flows within business processes. Essentially, a data dependency graph contains a set of nodes and connectors. A node denotes a data item in a business process. A connector represents a transition from an input data item to a task to an output data item from the task. For example shown in Figure 4-1, the data item, *d_2:BookInfo* is the input data item of the task, *t_3: Find book*; and the data item,

d_3*:Book*, is the output from the task. Therefore, a connector is created between data items, d_2 and d_3. Figure 4-2 illustrates the data dependency graph generated from the example business process.

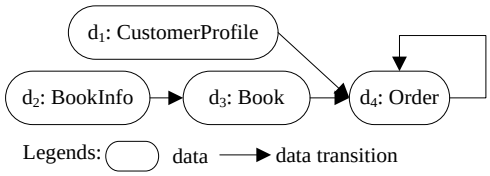


Figure 4-2: The example data dependency graph

DataDependency=<PreviousDataItem, SubsequentDataItem, ContainingBusinessProcess>;

PreviousDataItem =<d_1, d_2, …, d_m>;

SubsequentDataItem =<d_1, d_2, …, d_m>;

ContainingBusinessProcess =<p_1, p_2, …, p_v>;

Subscripts *m* and *v* are the number of data items and business processes respectively.

Figure 4-3: The format of data dependency vectors

We conduct cluster analysis on the data items in a data dependency graph. In particular, we choose the WCA clustering algorithm to reduce arbitrary decisions. The WCA algorithm produces a number of data groups at different levels of granularity. To select an optimal grouping result, we use the MQ metric to evaluate the quality of data groups. We aim to achieve high cohesion within a data group and low coupling among data groups. The MQ metric only concerns direct dependencies among components. Therefore, we analyze dependencies among data items and their adjacent data items and do not consider transitivity of dependencies.

To group data items, we examine three features of data items for describing dependencies of a data item: previous data items, subsequent data items, and the containing business processes. These features are organized as a dependency vector (i.e., *DataDependency* shown in Figure 4-3), which consists of three data components: *PreviousDataItem*, *SubsequentDataItem*, and

ContainingBusinessProcess. Furthermore, each data component in a dependency vector is also defined as a vector. More specifically, *PreviousDataItem* for a current data item is represented as *PreviousDataItem* =<d_1, d_2, ..., d_i, ..., d_m>, where d_i represents one data item defined in a business process, and *m* is the total number of data items defined in the business processes. d_i is set to 1 when d_i is the incoming data item of the current data item. Otherwise, d_i is set to 0. Similarly, the *SubsequentDataItem* vector marks the data items that appear as the outgoing data items of the current data item as 1. The *ContainingBusinessProcess* vector, i.e., <p_1, p_2, ..., p_i,..., p_v>, represents a collection of business processes that need to be implemented in a business application. *v* is the total number of business processes. p_i refers to a business process. It is set to 1 when p_i uses the current data item; otherwise, p_i is set to 0. For example, Table 4-1 illustrates the values of the vectors for the data dependency graph shown in Figure 4-2. Each row in the table represents a dependency vector of a data item. For example shown in Figure 4-2, the data item, d_1*: CustomerProfile*, has no previous data item, one subsequent item, d_4*: Order*, and one containing business process, p_1*: Purchase book*. Therefore, we set d_4 in the *SubsequentDataItem* vector and p_1 in the *ContainingBusinessProcess* vector of d_1 to 1 illustrated in Table 4-1.

Table 4-1: The data dependency table

	PreviousDataItem				**SubsequentDataItem**				**ContainingBusinessProcess**
Data item	**d_1**	**d_2**	**d_3**	**d_4**	**d_1**	**d_2**	**d_3**	**d_4**	**p_1**
d_1	0	0	0	0	0	0	0	1	1
d_2	0	0	0	0	0	0	1	0	1
d_3	0	1	0	0	0	0	0	1	1
d_4	1	0	1	1	0	0	0	1	1

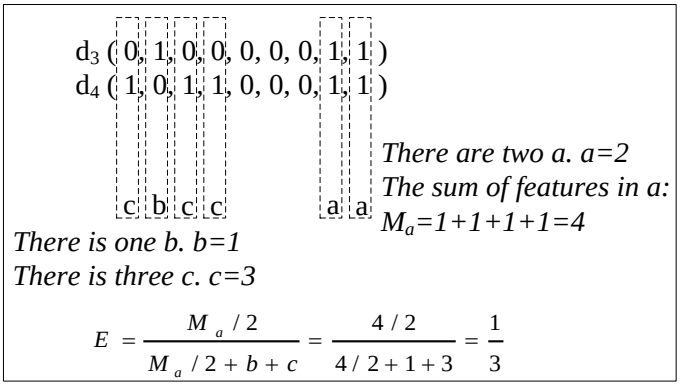


Figure 4-4: The process for calculating similarity

$$E = \frac{M_a / 2}{M_a / 2 + b + c} \tag{4-2}$$

Given two data items D_x and D_y ($x \neq y$), M_a denotes the sum of features that present for both data items. b represents the number of features that presents for D_x and absent for D_y. c represents the number of features that present for D_y and absent for D_x.

Using the data dependency table, we evaluate the similarities among data items using the unbiased Ellenberg measure which computes the degree of the similarity of two data items by evaluating the percentage of common features (i.e., previous data items, subsequent data items, or containing business processes). The definition of the unbiased Ellenberg measure is shown in equation (4-2) [107]. For example, we demonstrate the calculation of the similarity between d_3 and d_4 in Figure 4-4. d_3 and d_4 are two feature vectors whose values are adapted from Table 4-1. The similarities among any pair of data items in the example business process shown in Figure 4-1 are illustrated in Table 4-2. The more common features the two data item present, the more similar they are.

Using the similarities of any pairs of data items, we iteratively cluster data items using the five steps listed in Figure 4-5.

1) Initialize each data item as a data group;
2) Merge the two data groups that have the highest similarity value to form a new data group. For example, we choose d_1 and d_3 to create *DataGroup <1>* shown in Figure 4-6 (a);
3) Calculate features of the newly formed data group using equation (4-3) [20]. For example, we calculate features of *DataGroup <1>* using the feature vectors of data items, d_1 and d_3, illustrated in Figure 4-7. For a given feature in the merged group *DataGroup<1>*, we calculate the sum of the corresponding feature values in the data items, d_1 and d_3. For example, the second features of d_1 and d_3 are 0 and 1 respectively. Therefore, the sum of the second feature is 0+1=1. We also normalize feature values with the total number of data items in the newly formed data group. *DataGroup <1>* contains two data items (i.e., d_1 and d_3). Hence, we divide the second feature value *DataGroup<1>* by 2 and use ½ as the normalized feature value in the feature vector of *DataGroup<1>*;
4) Calculate similarities between the newly formed data group and other data groups using equation (4-2); and
5) Repeat step 2) to 4) until only one data group is left.

Figure 4-5: The data grouping algorithm

$$f_i = \frac{f_{i1} + f_{i2}}{n_1 + n_2} \tag{4-3}$$

f_i, f_{i1} and f_{i2} refer to the ith feature of the newly formed data group and its two constituent data groups respectively. n_1 and n_2 are the number of data items in the two constituent data groups.

Table 4-2: The data similarity table

Data item	**d_1**	**d_2**	**d_3**	**d_4**
d_1	-	1/3	2/3	2/5
d_2	-	-	1/4	1/6
d_3	-	-	-	1/3
d_4	-	-	-	-

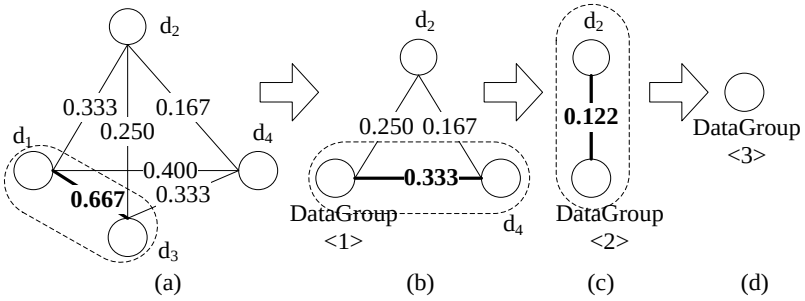


Figure 4-6: The data grouping process for the example business process

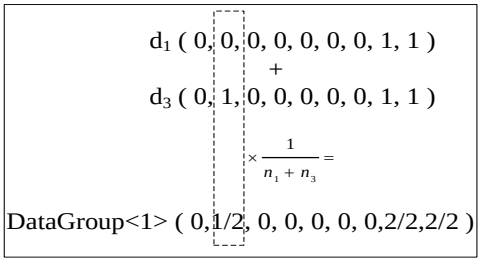


Figure 4-7: The process for calculating features for *DataGroup <1>*

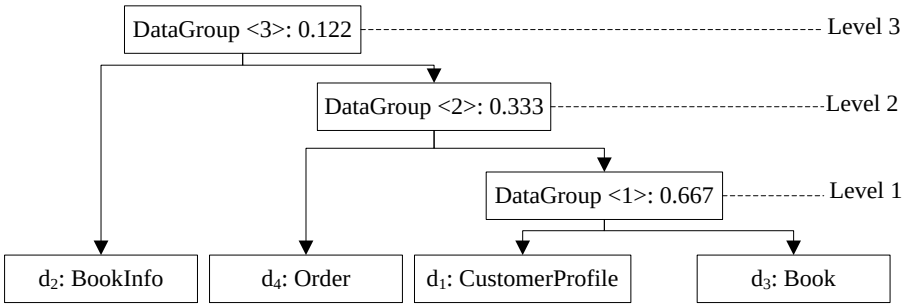


Figure 4-8: The result hierarchy of data groups

For example, Figure 4-6 shows the three iterations for grouping data items defined in the example business process (shown in Figure 4-1). One data group is generated in each iteration. We organize the generated data groups in a hierarchy as shown in Figure 4-8. Leaves of the hierarchy are data items in the business process. Internal nodes represent data groups created

from each iteration in a bottom-up fashion. The level in the hierarchy denotes the order of the data items being merged in the clustering process. For example, the data group *DataGroup<1>* (i.e., *{d_1: CustomerProfile, d_3: Book}*) in level 1 is generated in the first iteration. In the clustering algorithm, the most similar data items are grouped first. Therefore, data items merged (or data groups) in a lower level are more similar than those merged in a higher level of the hierarchy. For example shown in Figure 4-8, the similarity between *d_1: CustomerProfile* and *d_3: Book* is 0.667 in the data group, *DataGroup <1>*, while the similarity between *d_4: Order* and *DataGroup <1>* is 0.333 in the data group, *DataGroup <2>*. A high similarity value indicates a strong dependency. Therefore, data groups in lower levels have stronger intra-dependencies (i.e., high cohesion) while data groups in higher level have looser inter-dependencies (i.e., low coupling). To generate software architectures with high modularity, we use the MQ metric defined in equation (4-1) to balance cohesion and coupling and choose an optimal data grouping result.

Table 4-3: MQ values of the three data grouping results

Cutting level	**Grouping result**	**MQ**
1	{d_2: BookInfo}, {d_4: Order}, {d_1: CustomerProfile}, {d_3: Book}	-0.358
2	{d_2: BookInfo}, {d_4: Order}, {d_1: CustomerProfile, d_3: Book}	-0.219
3	{d_2: BookInfo}, {d_4: Order, d_1: CustomerProfile, d_3: Book}	-0.172

{$dataitem_1$, $dataitem_2$, …} represents a data group.

We can get multiple grouping results by cutting the hierarchy of data groups at different levels. Cutting the hierarchy of data groups at a certain level generates a data group that collects all data items appearing in the lower levels. For example, cutting the hierarchy of data groups shown in Figure 4-8 at level 2 results in three data groups: *{d_2: BookInfo}*, *{d_4: Order}*, and *DataGroup <1>*. Furthermore, the data group, *DataGroup <1>*, contains two data items, *d_1:*

CustomerProfile and d_3*: Book*. Therefore, the data grouping result can also be represented as *{*d_2*: BookInfo}*, *{*d_4*: Order}*, and *{*d_1*: CustomerProfile,* d_3*: Book}*. We can obtain three different data grouping results by cutting the hierarchy of data groups in each of the three levels, as listed in Table 4-3.

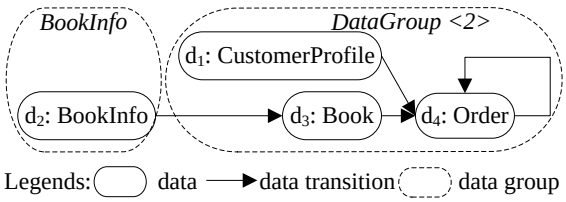


Figure 4-9: The selected data grouping result

The MQ metric, uses only the interactions between components to calculate their dependencies, as specified in equation (4-1). It is not sufficient to assess the cohesion and coupling using multiple quality attributes of components. To compare the modularity of the different data grouping results, we extend the definition of MQ to evaluate cohesion and coupling using data dependency vectors, defined in Figure 4-3. We use the intra-dependency of data items within a data group *i* to evaluate the cohesion of the data group and extend the definition of μ_i to the sum of similarities between all data items inside the data group. We use the inter-dependency between two data groups *i* and *j* to assess their coupling and extend $\varepsilon_{i,j}$ to the sum of similarities of all data items in the two data groups. The value range of the extended MQ is between -1 and 1. For example, we apply the extended MQ metric to evaluate the modularity of the three data grouping results listed in Table 4-3. Result MQ values are also shown in Table 4-3. Comparing these MQ values, we find that the data grouping result generated from level 3, *{*d_2*: BookInfo}*, *{*d_4*: Order,* d_1*: CustomerProfile,* d_3*: Book}*, has the highest MQ value which indicates the best modularity. As a consequence, we separate data items into two groups as illustrated in Figure 4-9.

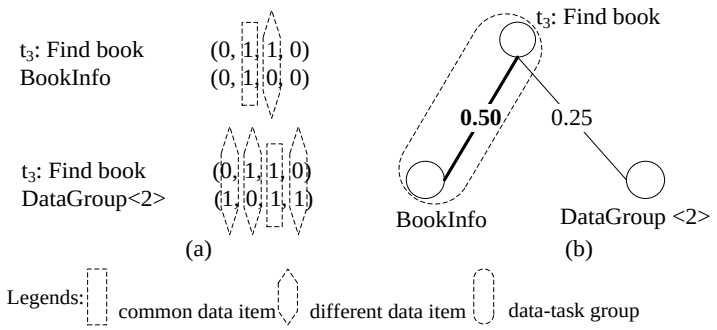


Figure 4-10: The data vectors and the component selection method

4.2.2 Associating Tasks with Data Groups

To identify functionalities for components, we assign tasks to data groups using the dependencies between data groups and tasks. We classify tasks into two categories: inner tasks and outer tasks.

Inner tasks depend on data items within a data group. More specifically, the data group contains both input data items and output data items of the task. All information required to performing this type of tasks is contained in one data group. Therefore, we group inner tasks with their dependent data groups into one component. For example shown in Figure 4-9, we can create two components from the data grouping result. One component contains the data item, *d_2: BookInfo*. The other component contains three data items: *d_1: CustomerProfile*, *d_3: Book*, and *d_4: Order*. The task, *t_4: Create order*, takes data items, *d_1: CustomerProfile* and *d_3: Book*, as input and generates data item, *d_4: Order*, as output. All these data items belong to the data group, *DataGroup <2>*. Therefore, we merge the task, *t_4: Create order*, to the latter component.

Outer tasks depend on data items distributed in multiple data groups. To assign an outer task to an appropriate data group, we evaluate the dependency strength between a task and the related data groups using the unbiased Ellenberg measure. Similar to measuring the similarity among data items, we create data vectors to describe the features of tasks and data groups. A data vector

describes the dependency of a task or a data group on all data items defined in the business processes. Therefore, a data vector is represented as *DataVector* =<d_1, d_2, ..., d_m>, where *m* is the total number of data items in business processes and d_i *(1≤i≤m)* denotes one data item. We set d_i to 1 if it is depended by a task or included in a data group. Otherwise, we set d_i to 0. For the example shown in Figure 4-1, the task, t_3*:Find book*, takes the data item, d_2*: BookInfo*, as its input and generates the data item, d_3*: Book*, as its output. Therefore, we set d_2 and d_3 to 1 in the data vector of the task, t_3*: Find book*, illustrated in Figure 4-10 (a). Similarly, the data group, *BookInfo*, contains one data item, d_2*: BookInfo*. Therefore, we set d_2 to 1 in the data vector, *BookInfo* as illustrated in Figure 4-10 (a). The calculated unbiased Ellenberg values in Figure 4-10 (b) indicate that the task, t_3*: Find book*, depends more strongly on the data group, *BookInfo*, than on the data group, *DataGroup<2>*. To achieve high cohesion, we assign the task, t_3*: Find book*, to the component corresponding to the data group, *BookInfo*.

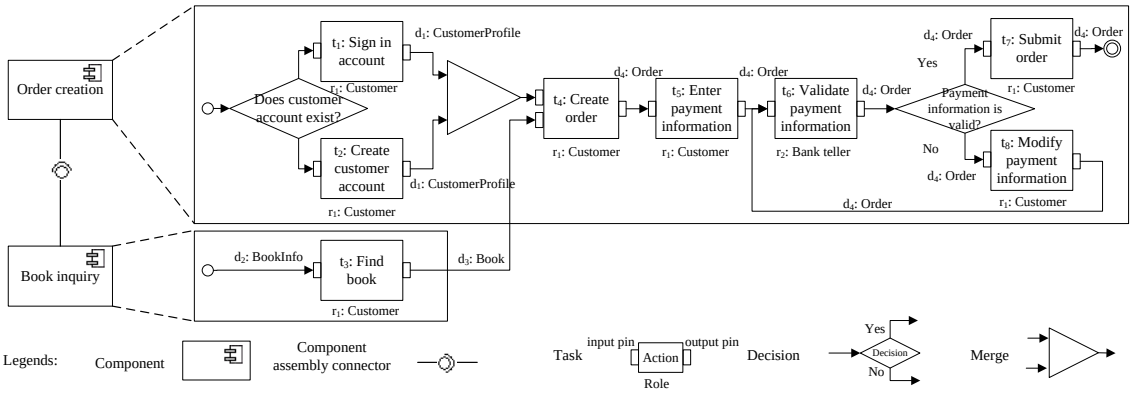


Figure 4-11: The generated functional software architecture

To produce a logical view of the software architecture, we further identify connectors between components by analyzing task transitions among components. For example, Figure 4-11 shows the two components generated from the example business process shown in Figure 4-1. The first component contains data items in the data group *DataGroup<2>* and tasks associated to them.

The second component is composed of the data item in the data group *BookInfo* and the task associated to it. The connector between these two components are identified from the transition between the task t_3*:Find book* and the task t_4*:Create order*.

4.3 Summary

In this chapter, we discuss our approach for generating functional software architecture from business processes. We first give a brief overview of clustering algorithms and then describe our approach for identifying architectural components from business processes. We describe the functionality of the generated architectural components using tasks and data items in business processes.

Chapter 5

Optimizing Software Architecture

In this chapter, we introduce our approach for evaluating and optimizing the generated software architecture designs with respect to specified quality requirements. Quality requirements specify desired quality attributes of a business application, such as easy to modify and good performance. These quality requirements can be interpreted differently in various domains. For example, the performance requirement can mean short response time in real-time systems or high throughput in web applications. Therefore, the meanings of quality requirements need to be elaborated for a given business application before a quality evaluation process. A variety of approaches [36][82] have been proposed to elicit concrete quality requirements. The strength of the goal oriented approach [60] is well recognized in representing and reasoning quality requirements. In our work, we use the goal oriented approach to specify application specific quality requirements.

Modifiability refers to the easiness to which a business application can be modified. Studies indicate that 50-70% of the cost in the lifecycle of a software system is devoted to modifications after the initial development [36][77]. In the following sections, we propose a collection of model transformation rules to automatically improve the modifiability of software architectures. We extend a set of existing product metrics to assess the modifiability impact of the proposed model transformation rules and guide the quality improvement process.

5.1 Describing Modifiability Requirements

Modifiability being a high-level quality attribute, a direct measurement is difficult. Therefore, we treat an abstract modifiability requirement as a high level goal and iteratively refine this high level goal into a set of low level goals that can be directly measured using metrics. As pointed out

by Bachmann et al. [38]. Modifiability can be improved by reducing effort required to modify components and by avoiding propagation of modifications. We break down the goal, *Improve modifiability*, into two sub-goals: *Reduce effort required to modify components* and *Avoid propagation of modification*. These sub-goals are further refined to more concrete goals as illustrated in Figure 5-1. The three-tier software architectural style improves cohesion and reduces coupling among components by distributing the functionality of a business application into three tiers and standardizing interfaces between tiers [40]. Hence, we use the three-tier software architectural style as a method to address the modifiability requirement as shown in Figure 5-1.

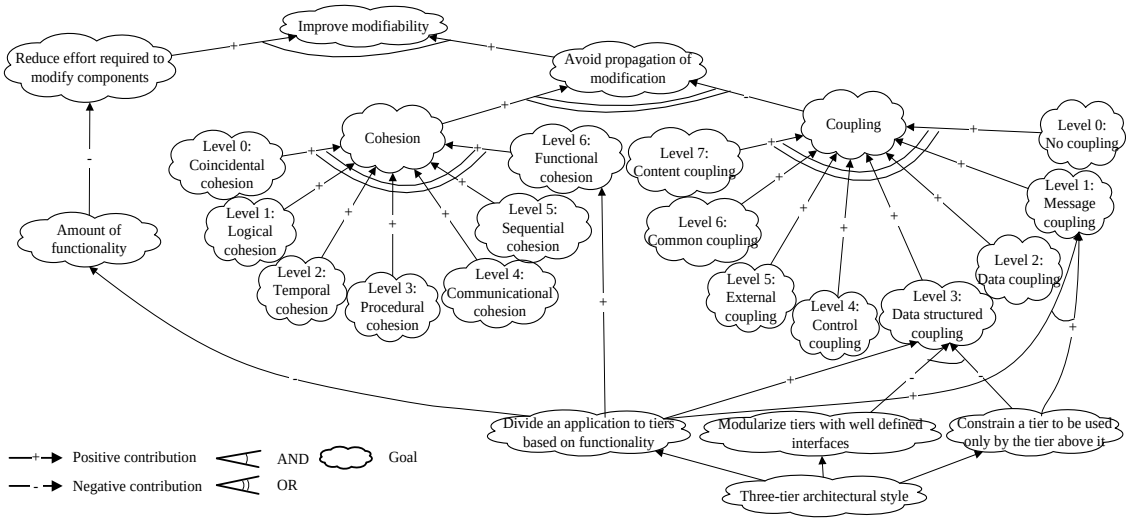


Figure 5-1: A goal model for modifiability

We use the soft goal graph [60] to represent the elaboration of modifiability goal. In the soft goal graph, goal relationships are represented by AND and OR links. An *AND* relation represents that all sub-goals need to be satisfied in order to achieve the parent goal. For example, sub-goals of the goal, *Improve modifiability* (shown in Figure 5-1), contribute to the goal with an *AND*

relation. An *OR* relation denotes that the satisfaction of one of the sub-goals is sufficient to achieve their parent goal.

The lowest level goals can be directly measured using software metrics. Changes of metric results reflect the quality impact of transformation rules. The changes are propagated to upper goals through links. Positive changes in the low level goals indicate the improvement of the upper goals. We label such links with "+" sign. Similarly, negative changes of the low level goals shows the negative impact on the upper level goals. We mark such links with "-" sign.

5.2 Metrics for Modifiability Evaluation

As shown in Figure 5-1, the modifiability can be evaluated using metrics for evaluating function points, cohesion and coupling. The traditional metrics take software entities, such as variables, statements, and methods, as input for computation. However, the generated software components are described using business process entities, such as tasks and data items. We extend traditional metrics to take business process entities as inputs for measuring the amount of functionality, cohesion and coupling of the generated architectural components.

5.2.1 Measuring Amount of Functionality

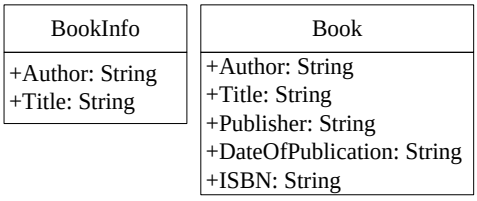


Figure 5-2: The definition of *BookInfo* and *Book*

If a component contains more functionality, it becomes more complex to modify [38]. Function point analysis (FPA) is used to evaluate the amount of functionalities in components and software

systems. We calculate the number of function points of a component using the approach defined in the ISO functional size measurement (FSM) standard [53] which evaluates functionalities using files, interfaces and interactions among applications. In our generated software architectures, detailed designs of components are described using data items and tasks derived from business processes. We analyze dependencies among data items and tasks to compute function points for each component. Specifically, following FSM, we re-mapped functionalities into the following five functional factors:

- *Internal logic files* hold data items used within a component. For the example shown in Figure 4-11, the component, *Order creation*, creates and edits the data item, *Order*. Therefore, the internal logic file of the component is *Order*.

- *External interface files* contain external data received from the operational environment. For the example shown in Figure 4-11, the component, *Book inquiry*, receives the data item, *BookInfo*, as input. The data item, *BookInfo*, specifies the data received from the operation environment and is identified as the external interface file of the component.

- *External inputs* refer to input pins of tasks that hold external data. For the example shown in Figure 4-11, the input pin of the task, *Find book*, holds the input data item, *BookInfo*, for the component, *Book inquiry*. Hence, we identify the input pin of the task, *Find book*, as the external input of the component.

- *External outputs* correspond to output pins of tasks that return data to the operational environment. For the example shown in Figure 4-11, the output pin of the task, *Find book*, holds the return data of the component, *Book inquiry*. Therefore, we identify the output of the task, *Find book*, as the external output of the component.

- *External inquiries* are tasks that capture data access actions. For the example shown in Figure 4-11, the task, *Find book*, captures the data access action in the component, *Book inquiry*. Hence, we identify the task, *Find book*, as the external inquiry of the component.

Table 5-1: Weights of functional factors in FSM

Functional factor	**Low**	**Average**	**High**
Internal logic files	7	10	15
External interface files	5	7	10
External inputs	3	4	6
External outputs	4	5	7
External inquiries	3	4	6

The function points of a component are the weighted sum of different types of functional factors within this component. For each functional factor, the FSM approach defines three levels of complexity and specifies a weight for the functional factor at each complexity level. Table 5-1 shows the complexity levels and corresponding weights for the five functional factors. In the *Book inquiry* component shown in Figure 4-11, inputs from customers are captured in the data item, *BookInfo*, which contains 2 primitive data attributes as shown in Figure 5-2. As specified in the FSM standard, the complexity of a data item with no more than 50 attributes is *Low*. Hence, the *External interface file* of the component, *Book inquiry*, has a *Low* complexity. Similarly, the complexity levels of the other three functional factors are *Low*. As shown in Table 5-1, weights for the four factors are 5, 3, 4, and 3, respectively. Therefore, the number of function points of the component, *Book inquiry*, is 15 as illustrated in Table 5-2.

Table 5-2: Function point count for *Book inquiry*

Functional factor	Number	Weight	Function point
Internal logic files	0	7	0×7
External interface files	1	5	1×5
External inputs	1	3	1×3
External outputs	1	4	1×4
External inquiries	1	3	1×3
Total			15

Two alternative architecture designs tend to have the same number of function points when they are generated from the same set of business processes. To evaluate the functionality distribution among components, we use the median value of function points from the components in an architecture design to quantify the overall cost of modifying components in a software architecture design. The median value can avoid the impact of the extreme values of functional points of components.

5.2.2 Measuring Cohesion and Coupling

Cohesion measures the strength of intra-dependencies within a component while coupling assesses the strength of inter-dependencies among components. Cohesion is divided into 6 distinct levels [137] and coupling is broken down to 7 distinct levels [120][136]. Figure 5-1 illustrates levels of cohesion and coupling in the order from the worst (low cohesion, high coupling) to the best (high cohesion, low coupling). Table 5-3 and Table 5-4 show the characteristics of data items and tasks in components for determining the cohesion levels and

coupling levels of components. Components with low cohesion and high coupling result in much stronger ripple effects of changes than others [120].

Table 5-3: Cohesion levels and their characteristics

Category	Cohesion level	Characteristics
High level	Functional cohesion	A component performs a single task.
Moderate levels	Sequential cohesion	A data item is sequentially transferred across tasks within a component.
	Communicational cohesion	All tasks within a component share the same input or output data items.
	Procedural cohesion	Tasks within a component are connected by control connectors.
Low levels	Temporal cohesion	Tasks within a component are correlated by temporal relations.
	Logical cohesion	Tasks that are logically grouped to perform the same type of functionalities.

We determine the cohesion level of a component by analyzing tasks and data items within the component. If they present characteristics corresponding to a cohesion level *h*, the cohesion level of the component is *h*. The level of coupling between two components is determined by analyzing dependencies between the two components. When this dependencies present characteristics of a coupling level *u* the coupling level of the two components is *u*. Once the levels of cohesion and coupling are determined, we calculate the dependency strength at each cohesion (or coupling) level. A number of metrics [45][104][126] can be used to calculate dependency strength among

components. We use the connectivity metrics defined in [126] to calculate dependencies among components. We chose these metrics because they are designed to evaluate cohesion and coupling based on dependency graphs that have similar structure to business processes; and they have been applied and validated on many systems [126].

Table 5-4: Coupling levels and their characteristics

Category	Coupling level	Characteristics
High level	Content coupling	A component uses data or control maintained by another component.
	Common coupling	Components share global data items.
	External coupling	Components are tied to external entities such as devices or external data.
Moderate levels	Control coupling	Controls flow across components.
Low levels	Data structured coupling	Structured data are transferred among components.
	Data coupling	Primitive data or arrays of primitive data are passed among components.
	Message coupling	Components communicate through standardized interfaces.

The cohesion strength of a component is evaluated by analyzing the number of dependencies within this component as shown in Equation (5-1). This cohesion strength increases as the number of dependencies within the component grows. The cohesion strength of a software

architecture is the average cohesion strength of all components within the software architecture. For our example from Figure 4-11, we analyze the dependencies introduced by task transitions to evaluate the strength of the procedural cohesion of the architecture. The component, *Book inquiry*, contains no task transition. Therefore, the strength of procedural cohesion of the component, *Book inquiry*, is 0. The component, *Order creation*, includes 7 task transitions and 7 tasks. Hence, the strength of the procedural cohesion of the component, *Order creation*, and the overall cohesion strength of the functional software architecture are calculated as follows:

$$A_{OrderCreation} = \frac{7}{7^2} = \frac{1}{7}\ ;\ S_{Cohesion} = \frac{0 + \frac{7}{7^2}}{2} = \frac{1}{14}\ .$$

$$S_{Cohesion} = \frac{\sum_{i=1}^{m} A_i}{m}, A_i = \frac{\mu_i}{n_i} \qquad (5\text{-}1)$$

A_i is the strength of dependency within the i^{th} component. μ_i refers to the number of dependencies within the i^{th} component. n_i is the number of tasks within the component. m is the number of components within the generated software architecture.

$$S_{Coupling} = \frac{\sum_{i,j=1}^{m} E_{i,j}}{m(m-1)/2}, E_{i,j} = \begin{cases} 0 & i = j \\ \frac{\varepsilon_{i,j}}{2n_i n_j} & i \neq j \end{cases} \qquad (5\text{-}2)$$

$E_{i,j}$ is the inter-dependency between the i^{th} component and the j^{th} component. $\varepsilon_{i,j}$ is the total number of dependencies between the two components. n_i and n_j are the number of tasks in the two components. m is the total number of components in the generated software architecture.

We use the inter-connectivity metric defined in equation (5-2) to assess the strength of coupling of a software architecture design by analyzing the number of dependencies among components. In equation (5-2), the strength of coupling between two components is assessed by examining the number of dependencies between the two components. The coupling strength

between two components increases as the number of dependencies between the two components grows. The coupling strength of a software architecture design is the average coupling strength among components within the software architecture design. For example shown in Figure 4-11, 1 control connector is used to connect the two components: *Book inquiry* and *Order creation*. The component, *Book inquiry*, contains 1 task; and the component, *Order creation*, has 7 tasks. Therefore, the control coupling of the software architecture is $S_{Coupling} = \frac{1}{\frac{2 \times 1 \times 7}{(2 \times 1)/2}} = \frac{1}{14}$.

We denote the cohesion and coupling of a software architecture design using the pair value (*h*, *s*) and *(u, s)*. *h* and *u* designates the levels of cohesion and coupling, respectively; *s* refers to the strength of this cohesion or coupling at these levels. To evaluate the overall cohesion of an architecture, we produce a paired value for each level of cohesion or coupling since components may have different levels of cohesion or coupling in a software architecture.

5.3 Model Transformation Rules for Architecture Improvement

In Chapter 4, we generate software architecture from business processes. Such software architectures, referred to as functional software architecture, describe the initial distribution of functionality without the conformance to the three-tier architectural style. The structure of these software architectures is illustrated in the meta-model shown in Figure 5-3. We define a set of model transformation rules that automatically restructures software architectures into three-tier software architecture. Our transformation rules distribute these functionalities into components in different tiers. We use object constraint language (OCL) [106] to specify each transformation rule. In the following subsections, we first describe the transformation rules that produce three-tier architecture. Then we describe the generation of connectors among the components in

different tiers. Finally, we discuss our technique for evaluating the quality impact of each transformation rule.

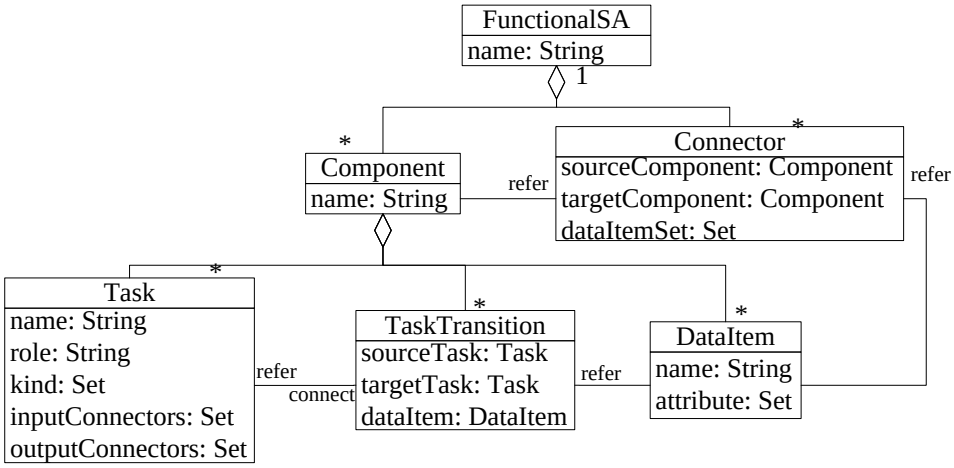


Figure 5-3: The meta-model of functional software architecture

5.3.1 Generation of Three Tiers

Functionalities of software architectures generated from business processes are described by the tasks specified in business processes. These tasks capture different functionalities as discussed in Section 3.1. In our work, we take components in the functional software architecture as source components and define transformation rules to automatically generate target components in the three-tier software architectural style. One source component can be broken down into fine grained components which contain unique functionality, and are distributed into different tiers in the target components. In the rest of this subsection we discuss the transformation rules by analyzing tasks and their transitions in source components.

- **Rule 1: Presentation Tier Generation Rule**

$$T_p = \{t \mid t.kind\text{->includes}(\text{'Human task'})\}$$

$$C_p = \{c \mid c.sourceTask.kind\text{->includes}(\text{'Human task'}) \textbf{ and } c.targetTask.kind\text{->includes}(\text{'Human task'})\}$$

where t is an instance of *Task* and c is an instance of *TaskTransition* as defined in Figure 5-3.

This rule identifies the presentation tier by grouping human tasks in a source component into a UI component and moving the UI component into the presentation tier. If the role of a task refers to a group of users, we classify such a task as a human task. The expression, T_p, selects tasks classified as human tasks from a source component to form one UI component. Moreover, we generate connectors among UI components by analyzing transitions among human tasks that are distributed in different UI components. The expression C_p, identifies task transitions whose source tasks and target tasks are both human tasks. Eventually, components in the presentation tier capture the functionality to handle communications between users and business logics.

- **Rule 2: Business Logic Tier Generation Rule**

T_b={*t*| *t.kind*->includes(*'Automatic task'*)}

C_b={*c*|*c.sourceTask.kind*->includes(*'Automatic task'*) **and** *c.targetTask.kind*->includes(*'Automatic task'*)}

where t is an instance of *Task* and c is an instance of *TaskTransition* as defined in Figure 5-3.

To provide support for users, we generate components in the business logic tier to handle requests from users and produce the corresponding responses. As specified in the expression, T_b, we select automatic tasks and their transitions from a source component to generate components and connectors within the business logic tier. The expression, extracts automatic tasks from a source component to form a component in the business logic tier. As described in the expression, C_b, we generate connectors among components in the business logic tier by analyzing transitions among automatic tasks. More specifically, the expression, C_b, checks the types of source and target tasks of task transitions and returns those whose source and target tasks are both automatic tasks.

- **Rule 3: Data Tier Generation Rule**

T_d={*t*| *t.kind*->includes(*'Data-access task'*)}

where t is an instance of *Task* as defined in Figure 5-3.

We create data access components in the data tier to support the communications between data access tasks and data repositories. We identify data access tasks from a source component to generate data access components by matching verbs in task names with the data access operations (e.g., insert, update and delete). Such operations are abstractions of fundamental database operations. Table 5-5 lists the heuristic mappings between task names and data access operations. The expression, T_d, extracts data access tasks from a source component. Data access components provide support for accessing persistent data without passing data between each other. Hence, we do not generate connectors among data access components.

Table 5-5: Huristic mappings between tasks names and data access operations

Data access operations	**Verbs in task names**
Insert	Create, Add, Insert
Update	Update, Modify
Query	Find, Search, Select, Get
Delete	Delete, Remove

5.3.2 Generation of Connectors between Tiers

Beside connectors among components within each tier, three-tier software architectures also contain connectors between tiers:

- *Presentation-Business* connectors that describe communications between components in the presentation tier and components in the business logic tier.
- *Business-Data* connectors that capture communications between components in the business logic tier and components in the data tier.

- **Rule 4: Presentation-Business Connectors Generation Rule**

$T_{p\text{-}b}$={*t*| *t.kind*->includes('*Human task*') **and** *t.kind*->includes('*Automatic task*')}

where t is an instance of *Task* as defined in Figure 5-3.

Presentation-Business connectors transfer information between human tasks in the presentation tier and the automatic tasks in the business logic tier. If a task captures users' interactions and requires automated support, it is broken down to a human task and an automatic task. For tasks with this feature, we generate a presentation-business connector to transit requests and responses between the presentation tier and the business logic tier as specified in the expression $T_{p\text{-}b}$.

- **Rule 5: Business-Data Connectors Generation Rule**

$T_{b\text{-}d}$={*t*| *t.kind->includes('Automatic task')* **and** *t.kind->includes('Data-access task')*}

where t is an instance of *Task* as defined in Figure 5-3.

Business-Data connectors pass information between automatic tasks in the business logic tier and data access tasks in the data tier. If an automatic task encapsulates data access operations, it is divided into a data access task and an automatic task. For tasks with this feature, we generate a business-data connector to transmit data between the business logic tier and the data tier. The expression $T_{b\text{-}d}$ extracts automatic tasks with data access operations for generating business-data connectors.

5.3.3 Application of Transformation Rules

To assess the quality impact of a model transformation rule, we compare the quality of software architectures before and after applying the transformation rule. If the impact confirms to specified quality requirements, *i.e.*, the modifiability, we deem that the model transformation is effective. Otherwise, we choose other transformation rules to improve the generated software architecture. Such an evaluation and improvement process is iterated till modifiability is maximized.

To compare the coupling of the source software architecture, SA_{before} and the transformed software architecture, SA_{after}, we compare the coupling level by level, in the order from high to low and consider that SA_{after} has lower coupling if:

- its coupling strength is less than that of SA_{before} when both are in the same level of coupling,
- a high level of coupling is present in SA_{before} but absent in SA_{after} or, a lower level of coupling is absent in SA_{before} but present in SA_{after}.

Similarly, we compare the cohesion of SA_{before} and SA_{after} to evaluate the impact of a transformation rule on the cohesion of a software architecture. The comparison of function points of SA_{before} and SA_{after} is performed using median values as discussed in Section 5.2.1. We strive to generate software architecture design with high cohesion, low coupling, and low function points to maximize the modifiability..

5.4 Summary

In this chapter, we describe our approach for optimizing software architecture. We first introduce our approach for elaborating quality requirements and then depict our method for evaluating and optimizing software architecture. Our proposed approach provides automated support for software architecture evaluation and optimization. We extend existing product metrics to automatically evaluate the quality of software architecture and perform model transformation rules to automatically restructure software architecture and resolve quality problems.

Chapter 6

Generating User Interfaces

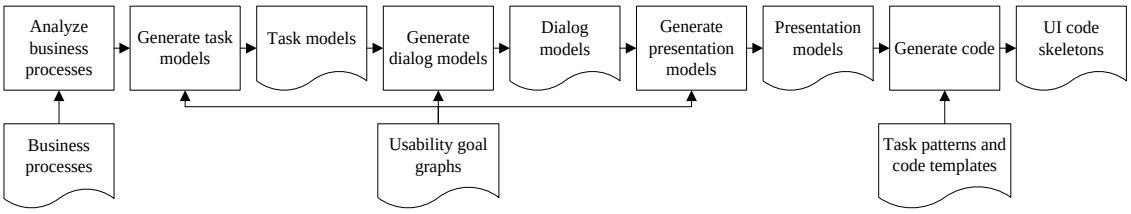


Figure 6-1: UI generation framework

In this chapter, we discuss our approach for producing UI designs and code skeletons from generated software architecture designs. The structure of generated software architecture designs is described in the meta-model shown in Figure 5-3. We use intermediate models with increasing details to bridge the gap between business processes and UI code skeletons. The intermediate models include task models, dialog models, and presentation models. Figure 6-1 illustrates the overall steps in our UI generation framework. In the following sections, we describe our approaches for generating task models, dialog models, and presentation models.

6.1 Generating Task Models

Task models recognize the structural and temporal information and describe how users execute various tasks defined in business processes. To derive such functional requirements, we need to analyze business processes and examine tasks that embrace functionalities of a business application. To handle tasks with various granularities, we analyze dependencies among tasks and data items in the business processes to group tasks into functional segments. We propose a collection of heuristic rules to group tasks on the basis of their dependencies. Intuitively, grouping relevant tasks divides a business process into a set of smaller segments of task groups.

Each segment is mapped to a window in the generated UI. The segments are linked by the transitions among tasks in the business process. The meta-model of task models is depicted in Figure 6-2. A business process consists of a set of *Segments* and *Connectors*. Each segment is composed of a collection of tasks and data items. In the following subsections, we discuss our heuristic rules for producing segments from business processes.

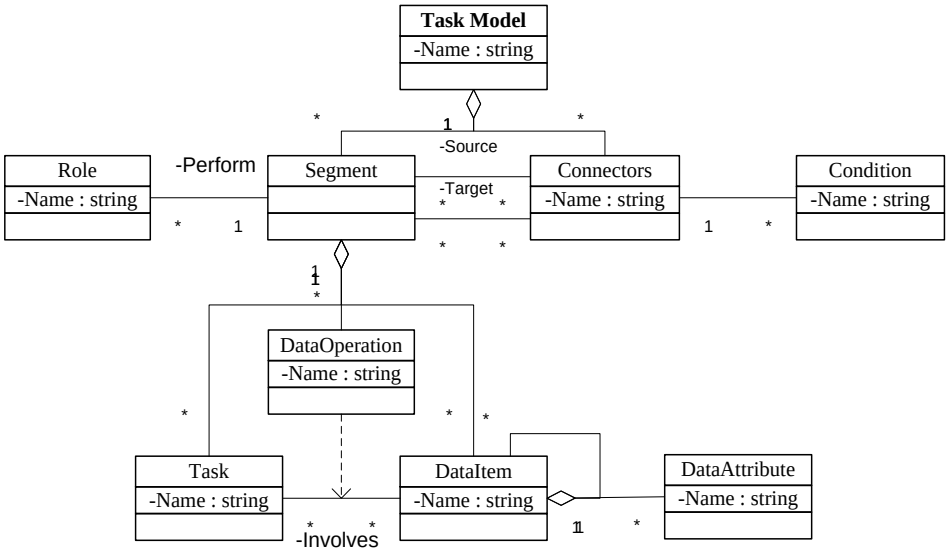


Figure 6-2: The meta-model of the task model

6.1.1 Role Rule

$$B_1=\{c|c.sourceTask.Role \neq c.targetTask.Role\},$$

where *c* is an instance of *TaskTransition* as defined in Figure 5-3

Different users use a business application to achieve different business goals and have different usability requirements. Users with different characteristics are modeled as different roles in business processes. In our work, we generate personalized UIs for each individual role in a business process. A personalized UI provides the necessary functional features and UI widgets with respect to a certain role. A user can focus on their own work and easily select the widgets from fewer widget sets in the UI. We define a role rule that divides a business process into a few

groups as specified in B_1. Each group of tasks is performed by the same role. We identify a set of connectors that link tasks performed by different roles as delimiters of segments. More specifically, connectors in B_1 divide a business process into *Segments*. Each *Segment* contains the tasks and related data items performed by one role. For example, as illustrated in Figure 6-3, the business process is segmented by connectors *{5, 6, 7, 9}* by applying the role rule.

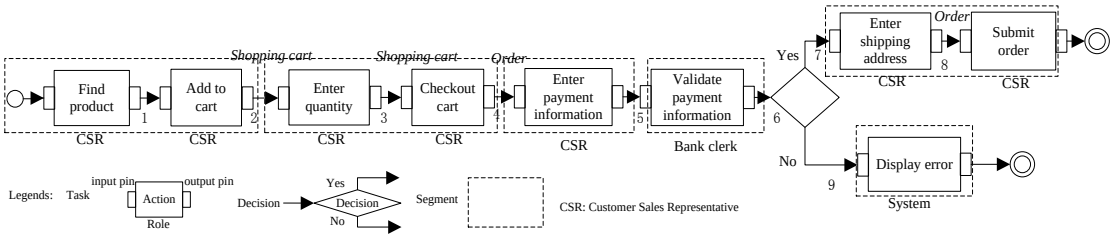


Figure 6-3: An example business process

6.1.2 Primitive Task Rule

$$B_2=\{c|c.sourceTask.kind->includes(\text{“Primitive task”})\}$$

where *c* is an instance of *TaskTransition* as defined in Figure 5-3.

Primitive tasks, such as *Add to cart* and *Submit order*, describe the lowest level of details in a business process. Such tasks are usually implemented as button widgets and used in combination with other tasks that share the same UI window. For example, when purchasing a product, we first find the product using attributes of the product and add the found product to a shopping cart. To improve the efficiency of the UI, the *Add to cart* task is often combined with the *Find product* task and implemented in one window, as illustrated in Figure 6-3. We identify primitive tasks using naming conventions. For example, we look for tasks with a name containing *submit*, *save*, *add*, and *delete*. A primitive task alone does not produce a segment in a business process.

6.1.3 Manual Task Rule

$$B_3=\{c|c.sourceTask.kind->includes(\text{“Manual task”})\ \textbf{xor}\ c.targetTask->includes(\text{“Manual task”})\}$$

where *c* is an instance of *TaskTransition* as defined in Figure 5-3.

Manual tasks, such as sending surface mail, are manually accomplished by humans without using business applications. Therefore, no UI windows are required for performing the manual tasks. The manual task rule is defined by B_3 and is used to exclude the manual tasks from the rest of the tasks in a business process. Therefore, we can avoid generating unnecessary widgets or windows for manual tasks in UIs. We identify connectors whose source tasks or target tasks are manual tasks to separate manual tasks from the other tasks.

6.1.4 Optional Task Rule

$$B_4=\{c|c.sourceTask.kind->includes(\text{“Optional task”})\ \textbf{xor}\ c.targetTask->includes(\text{“Optional task”})\}$$

where *c* is an instance of *TaskTransition* as defined in Figure 5-3.

To improve their work efficiency, users would rather click fewer buttons. In this case, the developers may choose to provide default values or settings for UI widgets, such as combo boxes and text boxes. We use the default values specified in the attributes of data items to specify the default values for widgets that are used to implement tasks. Such tasks are considered as optional tasks that are performed only when the user would like to change the default values. For example, credit card is defined as the default payment method. The task, *Select payment method*, can be skipped if users make payment using credit cards. Optional tasks separate the business processes into segments. This rule can improve efficiency since users perform optional tasks only when they cannot use default values. We use the expression, B_4, to identify connectors that link to optional tasks. We detect connectors whose source tasks or target tasks are optional tasks to enclose segments that are composed of optional tasks.

6.1.5 Branch Rule

$$B_5=\{c|\ c.sourceTask.inputConnectors.size()>1\ \textbf{xor}\ c.targetTask.outputConnectors.size()>1\}$$

where *c* is an instance of *TaskTransition* as defined in Figure 5-3.

If a task has more than one outgoing connector, this structure indicates alternative branches or parallel paths. We include all the tasks in one branch as a segment. For example, the business process in Figure 6-3 is divided into three segments using connectors, *{6, 7, 9}*, identified by the branch rule.

6.1.6 Data Sharing Rule

B6={c|(c.sourceTask.inputConnectors->notEmpty()

and *c.sourceTask.inputConnectors.exists(ic:TaskTranstion|ic.dataItem≠c.dataItem)}*

where *c* is an instance of *TaskTransition* as defined in Figure 5-3.

If a sequence of tasks operates on the same data item, these tasks share the same data information. In the UI design, it would be inefficient if a user had to enter the same information multiple times when performing different tasks. This data sharing rule improves the efficiency of the UI design by grouping the tasks and their shared data item in a segment. As specified in B_6, if the data item of the connector *c* and the data item of the connector *ic* are not equivalent, *c* is a connector that divides the business process into different segments. For example as shown in Figure 6-3, two segments are identified using the data sharing rule. The tasks, *Enter quantity* and *Checkout cart*, are included in the same segment since both tasks share the data item, *Shopping cart*. The tasks, *Enter payment information* and *Submit order*, share the data item, *Order* and therefore are grouped into one segment.

6.2 Application of Rules

Each rule has preconditions that specify the context for applying the rule. For example, the branch rule would not be applied for sequentially ordered tasks. In the process of dividing a business process into a collection of segments in a task model, we identify an applicable set of rules using the preconditions of each rule. The role rule, data sharing rule, branch rule, manual

task rule, and optional task are independent from each other. Therefore, the result of the segments is independent from the order of applying these rules. The primitive tasks, such as the *Add to cart* and *Submit order*, are not associated with a separate window. As the result, the primitive task rule identifies primitive tasks, and merges the identified primitive task with the prior task. In this case, the primitive task and their prior task are treated as one merged task. Therefore, we apply the primitive task rule before any other rules. Figure 6-3 illustrates a segmentation example. The primitive task rule is applied before any other rules and it identifies the connector set B_2. Moreover, the role rule, branch rule, and data sharing rule are applied independently in any orders. These will identify the connector set, $B_1 \cup B_5 \cup B_6$. At last, we derive six connectors that divide the business process into segments (i.e., $B_2 \cup (B_1 \cup B_5 \cup B_6) = \{2, 4, 5, 6, 7, 9\}$) shown in Figure 6-3.

6.3 Dialog Model

The dialog model is composed of a set of linked windows, as illustrated in Figure 6-4. Each window is generated from a segment in the task model. Window transitions are generated from connectors in the task model. In the dialog model, task windows provide widgets that allow users to interact with the application in order to fulfill tasks following predefined orders as specified in the business processes. The support windows provide contextual support that assists users in performing tasks. A task window contains task and data operations. A support window can only include data operations inferred from data items. An example of data operations can be viewing the product specification before performing the task, *Add product to shopping cart*.

Our rules for transforming task models to dialog models are inspired by [17][70], which derive navigation structures and window structures from task structures. In our work, window transitions are directly derived from the connectors in the task models. That is, if there is a connector

between tasks in two different segments, we will generate a transition between the two task windows generated from the two segments. We design the following three rules to transform segments to windows.

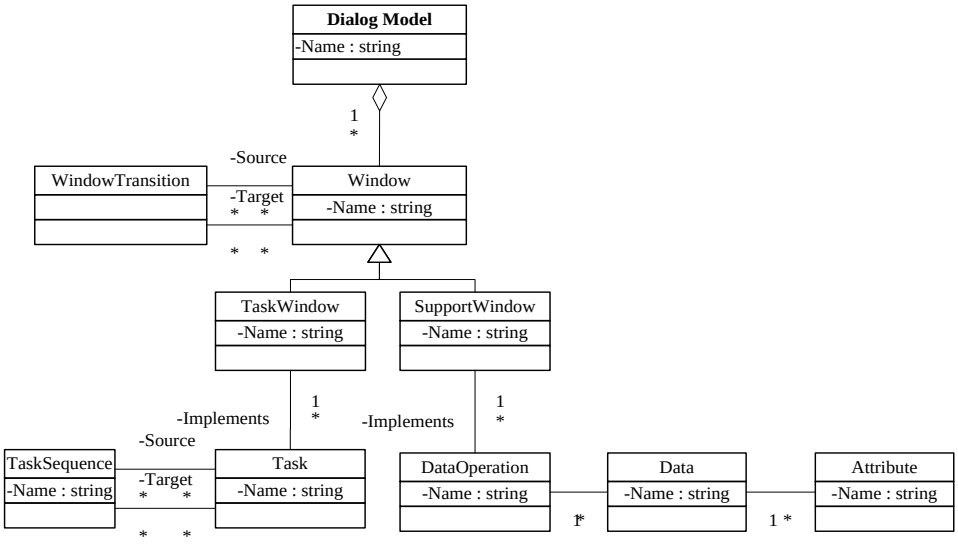


Figure 6-4: The meta-model of the dialog model

6.3.1 Single Window Rule

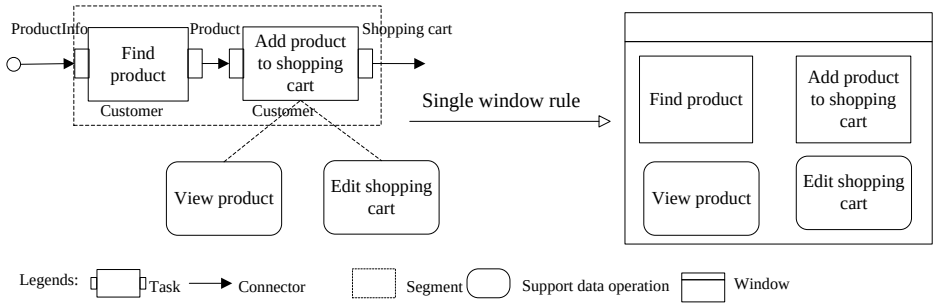


Figure 6-5: An example of the single window rule

The single window rule transforms all tasks and data operations in one segment into a task window. This rule results in UIs with high efficiency of use, since the users can complete tasks using fewer clicks. But the UI may be difficult for novice users who need to learn and remember

the functionality and order of use of all the widgets in one window. Figure 6-5 shows an example transformation from a segment to a window.

6.3.2 Data Window Rule

The data window rule transforms a segment to a task window and multiple support windows. The task window implements all tasks in the segment. The support windows implement the data operations in the segment. A support window contains all the data operations related to a specific data item. In this case, support windows can be reused in many other segments that contain data operations for the same data item. This rule reduces the number of tasks and data operations implemented in one window. Moreover, users' knowledge about a support window can be reused in other contexts when the same data items are used. Figure 6-6 illustrates an example transformation from a segment to a task window and two data windows.

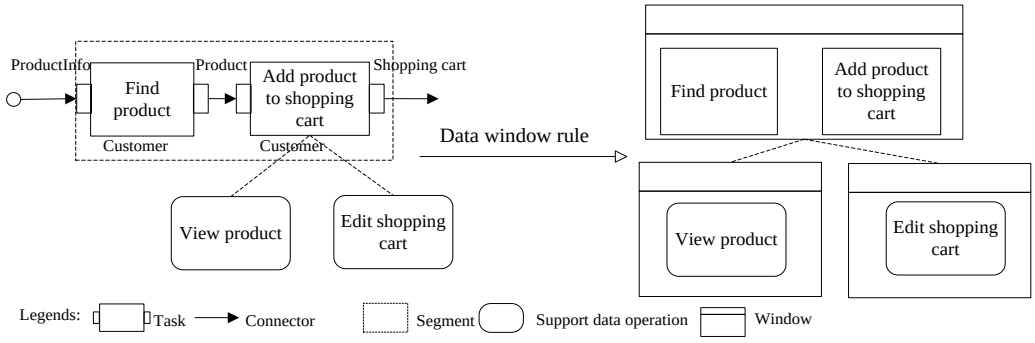


Figure 6-6: An example of the data window rule

6.3.3 Functional Window Rule

This rule transforms all tasks in a segment into a window and all data operations in the segment into a support window. This rule decreases the number of tasks and data operations in a window. The generated UI is easier to learn than the one generated using the single window rule. However, it may be more difficult to learn than the one generated using the data window rule. This rule is a

trade-off between the single window rule and the data window rule. Figure 6-7 shows an example transformation from a segment to a task window and a support window.

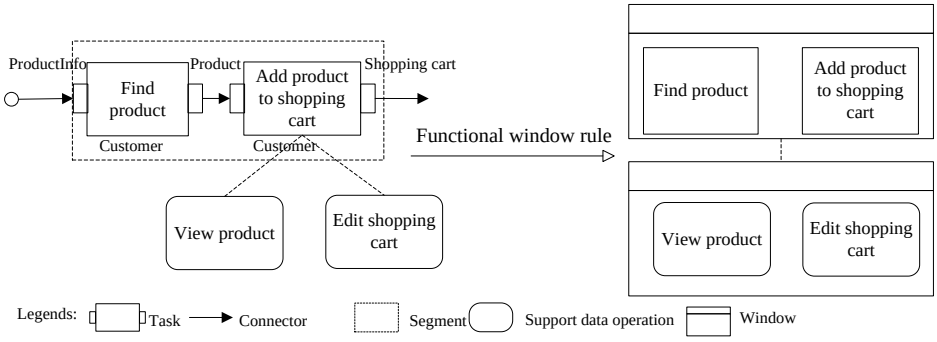


Figure 6-7: An example of the functional window rule

6.4 Presentation Model

The presentation model transforms the dialog model into a collection of platform-independent abstract interface objects (AIOs). As illustrated in Figure 6-8, a window in the presentation model is associated with a collection of AIOs, such as input fields, output fields, selectors and action invokers. The generation of the presentation model is accomplished by matching tasks with task patterns, which are collections of widely used operations in the design of UIs, such as find operation, display contents operation. Two well-known collections of UI task patterns are summarized in [69][92]. For example, Figure 6-9 shows the structure of a search task pattern, adapted from [32][41]. Specifically, a search task can be decomposed into two sub-tasks including entering search criteria and displaying search results. Furthermore, an input field (e.g., text field) and a search action invoker (e.g., button) allow a user to enter search criteria. Similarly, an output field (e.g., a table) and a select action invoker (e.g., button) display the returned result. We use naming similarities to match tasks in the business processes with task patterns used in the

UI designs. The screenshot for our generated UI after performing the search task pattern on the task, *Find product*, is shown in Figure 6-10.

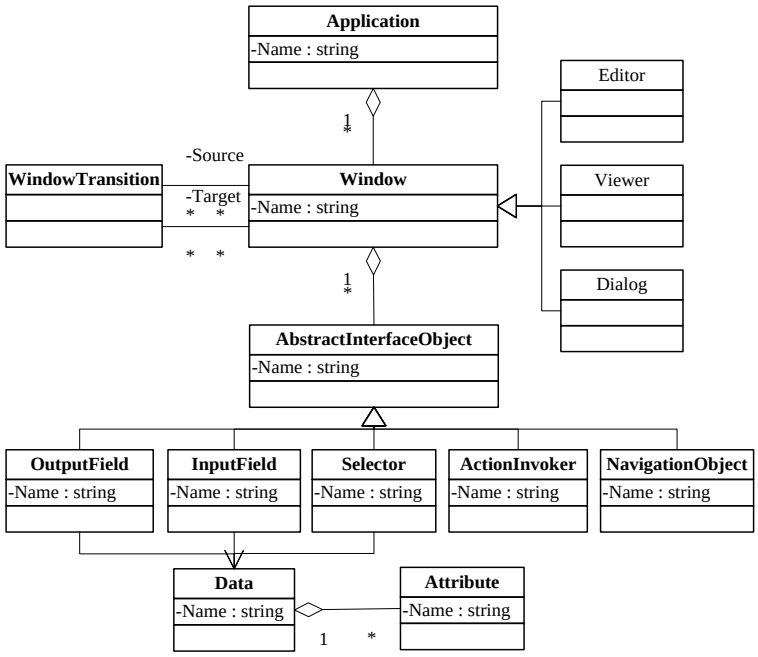


Figure 6-8: The meta-model of the presentation model

To transform a platform-independent presentation model into platform-specific UI code skeletons, a set of platform-specific code templates are used to specify code snippets for implementing AIOs. The selection of a concrete widget is determined by the tasks and data items specified in the business process. For example, when a task generates output data, the window that implements this task is set to be an editor, for the reason that a user can modify the intermediate result when performing the task. When a data item, such as a *Picture* of a *Product*, is used as an input to a task, the content of this data item can be previewed using a viewer assuming that we can't modify the content of the input data item. We map the types of the data attributes of a data item with different widgets that are used to display the context of a data item. For example, the data item, *Product*, contains string typed attributes, such as *Product ID*, *Product name*, and

Product type. The string typed data attributes are interpreted into text fields, as illustrated in Figure 6-10. A data item that can appear multiple times (e.g., an array of resulting products), is converted into a table, as shown in Figure 6-10.

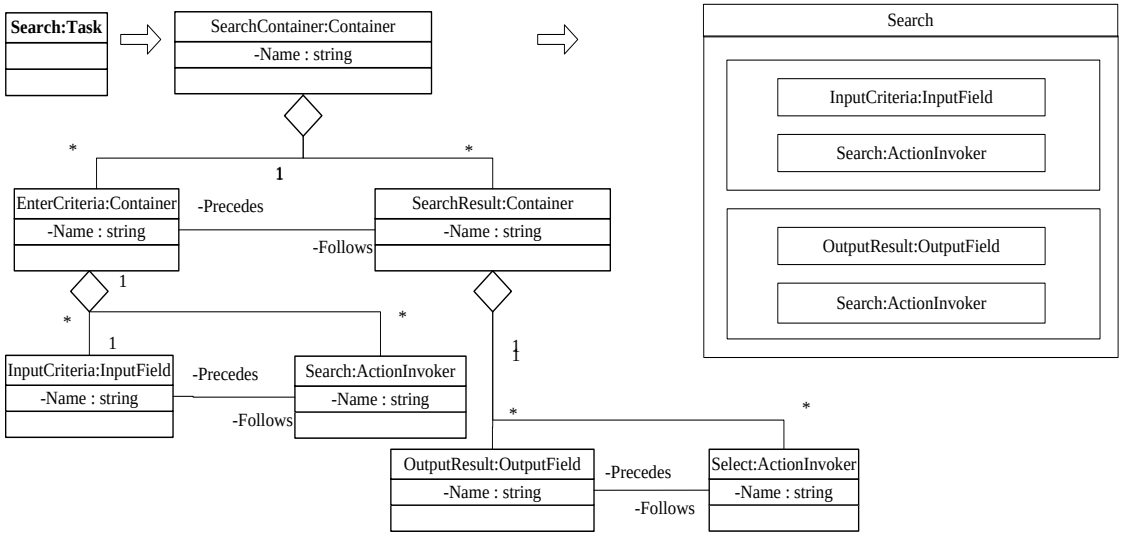


Figure 6-9: The structure of the *Find* task pattern and an example presentation model

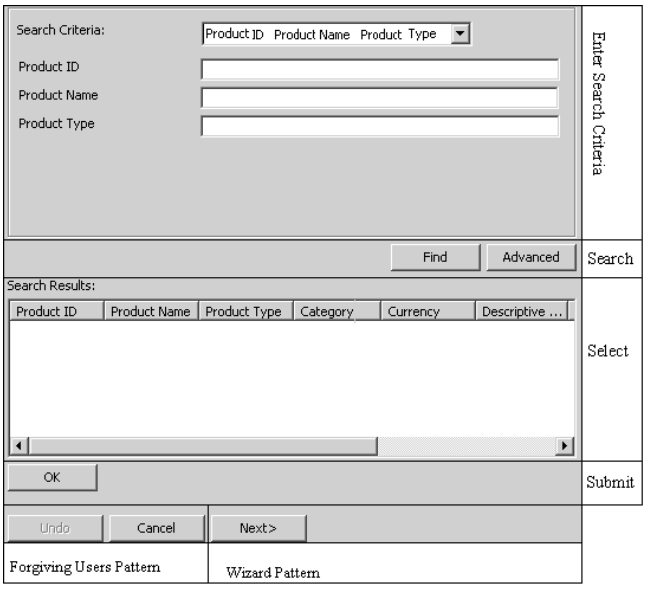


Figure 6-10: The generated UI

Moreover, we incorporate UI design guidelines and patterns into the generation of the UI to improve the usability of generated UIs. For example as shown in Figure 6-10, three buttons (i.e., *Undo*, *Cancel*, and *Next*) at the bottom are generated to improve the usability of this window. The *Undo* button and the *Cancel* button allow user to correct their mistakes by adapting the *Forgiving users pattern* usability pattern [69][92]. The *Next* button is created using the *Wizard* pattern [69][92], which guides users to perform tasks in a step-by-step fashion. Users can use this button to transition to the next page to perform the next task.

6.5 Summary

In this chapter, we introduce our approach for generating UI designs and code skeletons from business processes. We use a collection of intermediate models to bridge the gap between business processes and code skeletons and then define model transformation rules to generate and synchronize models and code skeletons. Our approach provides automated support for creating UI designs and code skeletons from business processes.

Chapter 7

Optimizing Usability of User Interfaces

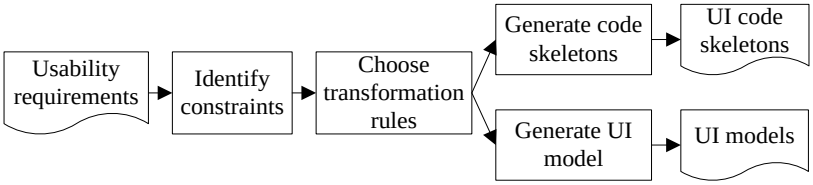


Figure 7-1: A framework for incorporating usability goals into model transformations

To generate UIs with desired usability features, we use usability requirements to guide the selection of transformation rules presented in Chapter 6. As shown in Figure 7-1, we interpret usability requirements to constraints on UI models and use the identified constraints to guide the selection of transformation rules. Eventually, we perform the chosen transformation rules to generate UI models or code skeletons.

7.1 Identifying Constraints

As discussed in Chapter 5, we use goal models to elaborate concrete usability requirements. Usability measures the extent to which UIs can be understood and used. Nielson [64] divides usability to five sub-attributes including:

1) **Learnability** assesses the easiness for end-users to accomplish tasks when they use the UIs for the first time.
2) **Memoriability** evaluates the easiness for end-users to complete tasks using the UIs after a period of not using them.
3) **Low error rate** measures the number of errors that end-users make during the use of the UIs.

4) **Efficiency** measures the performance of end-users after they have learned the UIs.

5) **Satisfaction** assesses users' impression of the UIs.

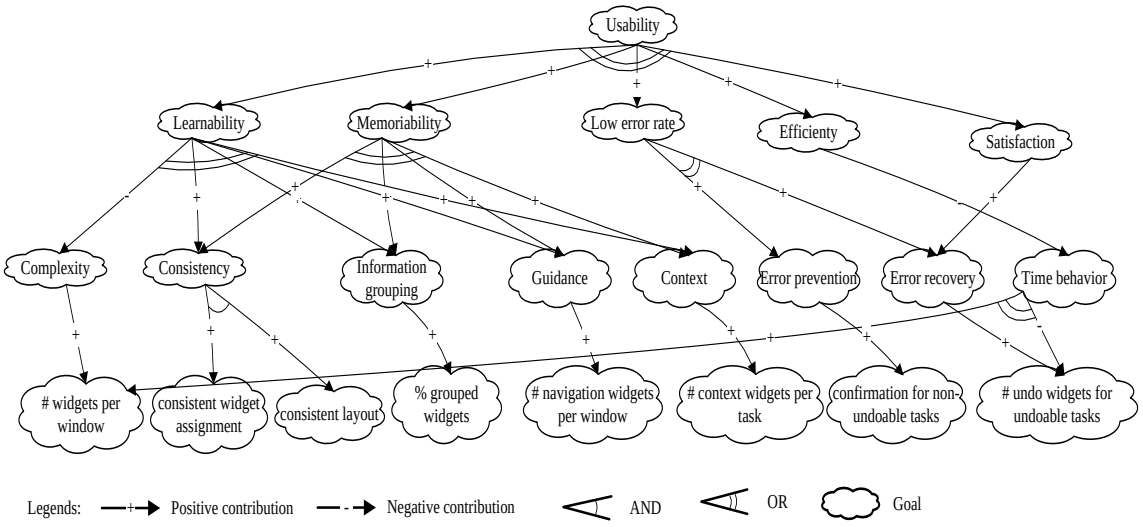


Figure 7-2: An example usability goal graph

Figure 7-2 depicts the relationships between usability and the five sub-attributes. These sub-attributes are abstract and cannot be directly measured. To measure the usability of UI models, we decompose the five sub-attributes to a set of measurable attributes by referring to usability models [64][93][97][122] in the literature. The identified measurable attributes are shown in Figure 7-2.

1) The complexity of UIs contributes to **learnability** negatively [97][122]. A simple UI requires little cognitive effort and is therefore easy to learn.

2) The consistency of UIs contributes **learnability** and **memoriability** positively [122]. With consistent UI designs, users can reuse their past experience to learn new UIs.

3) Grouping items in UIs help to improve **learnability** and **memoriability** [93]. Items in one group provide hints for users to learn and use the other items in the same group.

Table 7-1: Metrics for evaluating structural attributes

Metric	Description
Number of widgets per window	This metric measures the average number of widgets in UIs of a business application.
Consistent widgets assignment	This metric evaluates whether the same type of tasks are implemented by the same set of widgets.
Consistent layout	This metric assesses whether the same layout are applied to all tasks of the same type.
Percentage of grouped widgets	This metric measures the average percentage of grouped widgets in UIs of a business application.
Number of navigation widgets per window	The metric examines the average number of navigation widgets in UIs of a business application.
Number of context widgets per task	The metric evaluates the average number of context support widgets for tasks of a business application.
Confirmation for non-undoable tasks	This metric checks whether undoable tasks are protected by confirmation dialogs.
Number of undo widgets for undoable tasks	This metric assesses the average number of undo widgets for undoable tasks.

4) Guidance in UIs contributes to **learnability** and **memoriability** positively [93]. Guidance provides explicit instructions to reduce users' effort in learning and using UIs.

5) Context support in UIs helps to improve **learnability** and **memoriability**[139]. Context sensitive support provides necessary information for users to perform tasks and reduces users' effort required to learn and use UIs.
6) Error prevention and error recovery both contribute positively to **low error rate** [64]. Error prevention mechanisms validate users' inputs and prevent errors from happening. Error recovery helps users' returning a valid status once errors occur. In addition, error recovery can reduce frustration of users and improve the **satisfaction** of users [64].
7) Time behaviors of UIs contribute negatively to **efficiency** [64]. Simple UIs often have long interaction paths and reduce the efficiency of users. Undoable UIs reduce the time for users to fix mistakes and improve the efficiency of users.

Each measurable attribute can be measured by one or more metrics as illustrated in Figure 7-2. Table 7-1 shows descriptions of theses metrics.

7.2 Choosing Transformation Rules

We choose transformation rules with respect to specified usability requirements to generate UIs with desired usability features. Usability impact of a transformation rule can be evaluated by comparing UIs before and after the transformation rule is applied. We measure the impact of transformation rules using the metrics listed in Table 7-1. If the impact confirms to specified usability requirements, we deem that the model transformation is effective. Otherwise, we choose other transformation rules to optimize the generated UIs. Such an evaluation and optimization process is iterated multiple times till all quality problems are resolved.

For example, we strive for generating UIs with high *number of widgets per window* to improve the efficiency of advanced users. We choose transformation rules to produce UIs with high *number of widgets per window* in the following steps:

1) Choose a rule from the predefined rule set, we denote the UIs before and after the transformation rule is applied as UI_{before} and UI_{after}, respectively;
2) If the *number widgets per window* in UI_{after} is higher than that in UI_{before}, we consider the impact of the transformation rule is in conformance to the given usability requirements and add the transformation rule to the chosen transformation rule set;
3) Otherwise, we go to step 1) and try other rules.

7.3 Summary

In this chapter, we describe our approach to generate UIs with desired usability features. We use goal models to elaborate concrete usability requirements and map these requirements to structural attributes of UI models. We derive a collection metrics from UI design guidelines to evaluate structural attributes of UI models and guide the selection of transformation rules. Our approach provides automated support to address usability requirements during UI design.

Chapter 8

Overview of Prototype Tools

8.1 Software Architecture Generation Tool

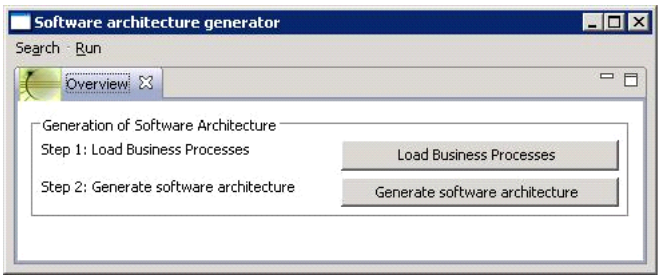


Figure 8-1: The screenshot of the software architecture generation tool

We develop a software architecture generation tool prototype on the Eclipse RCP platform. Figure 8-1 is the screen shot of our software architecture generation tool. Using the software architecture generation tool, we generate software architecture in two steps:

1) Analyze business processes to identify required information according to the meta-model defined in Figure 3-4.
2) Generate software architecture designs following the approach presented in Chapter 4 and Chapter 5.

8.1.1 Features of the Software Architecture Generation Tool

The current version of our software architecture generation tool offers the following features: automatic requirement extraction, automatic software architecture generation, automatic software architecture optimization, and automatic generation of software architecture descriptions.

Automatic Requirement Extraction: To generate software architecture, we need to identify business requirements from business processes.

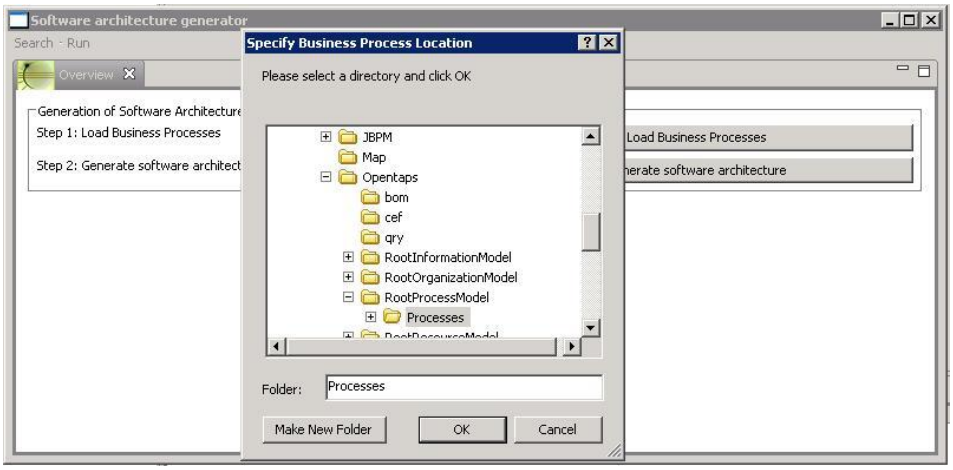


Figure 8-2: The dialog for loading business processes

In our work, business processes are modeled using IBM WebSphere Business Modeler and are stored as XML documents. We provide a dialog for users to load business processes from specified directories as shown in Figure 8-2. We develop a business process parser to parse the loaded business processes. The parser provides a set of application programming interfaces (APIs) for identifying required information from business processes created using WBM. With these APIs, we extract requirements specified in business processes according to the meta-model defined in Figure 3-4.

Automatic Software Architecture Generation: Our prototype tool automatically generates software architecture by analyzing requirements specified in business processes. We first group data items in business processes by invoking the WCA clustering algorithm and then associate tasks with data groups to form architectural components. Figure 8-3 illustrates a screenshot of the clustering result. Clusters are organized in a hierarchy. Leaves of the hierarchy are data items in business processes. Branch nodes of the hierarchy are clusters at different granularity levels.

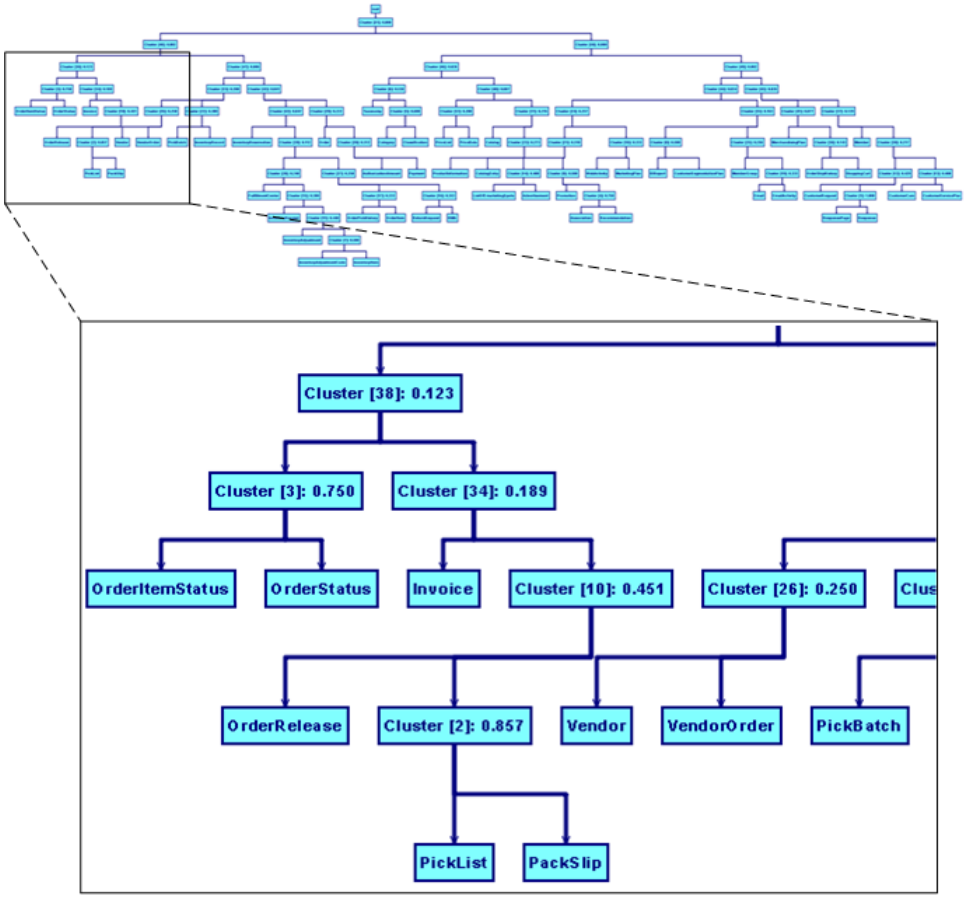


Figure 8-3: The screenshot of the clustering result

Automatic Modifiability Optimization: To reduce the effort required for quality optimization, we design model transformations to restructure the generated software architecture. In the prototype tool, we provide a collection of predefined model transformation rules that can be performed to automatically generate three-tier software architecture from functional software architecture. The transformation is guided by the extended product metrics to maximize the modifiability of the generated three-tier software architectures as discussed in Chapter 5. Figure 8-4 shows a functional software architecture generated from the example business process shown

in Figure 4-1 and the optimized version of the software architecture after applying the three-tier software architectural style.

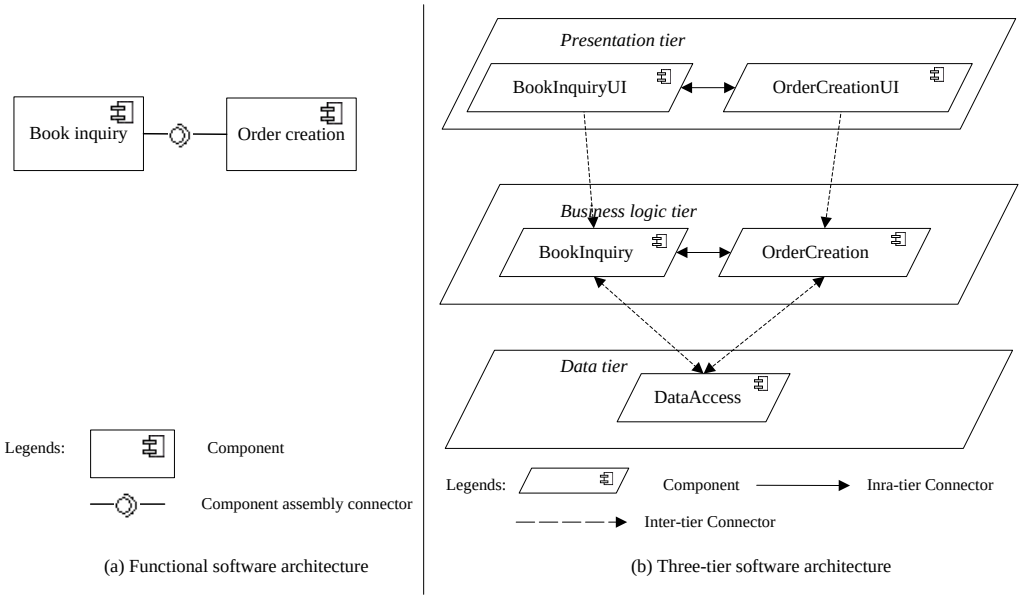


Figure 8-4: Example software architecture designs

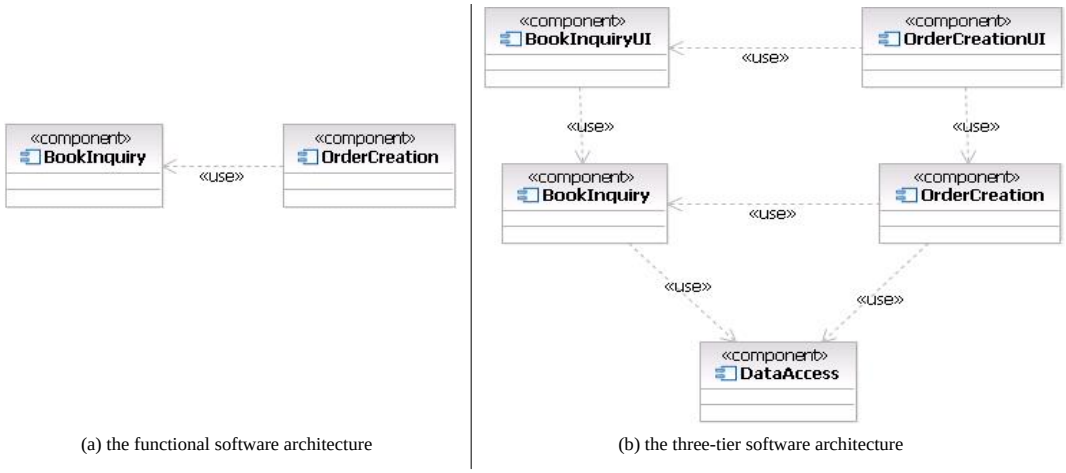


Figure 8-5: Descriptions of generated software architecture designs

Automatic Generation of Software Architecture Descriptions: To help users understanding and using generated software architecture designs, we use UML notations to describe generated

software architecture designs. Since UML is the de facto software modeling language, such descriptions are easy for users to understand and use. Figure 8-5 (a) and Figure 8-5 (b) illustrate generated descriptions of the functional software architecture and the three-tier software architecture shown in Figure 8-4, respectively.

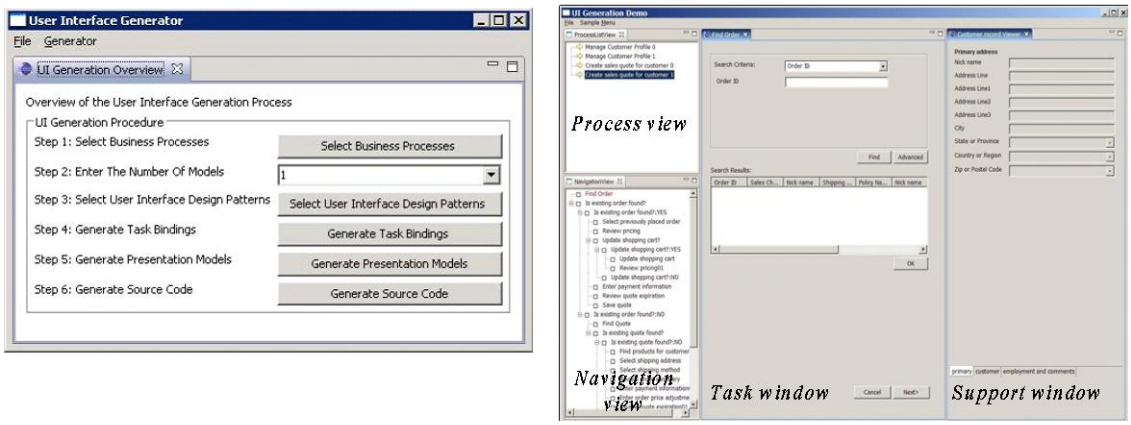


Figure 8-6: Screenshots of UI generation and testing tools

8.2 UI Generation Tool

We implement a UI generation tool prototype to generate UI designs and code skeletons from business processes. Figure 8-6 (a) shows the UI of our UI generation tool prototype. Moreover, we also implement a testing environment to preview generated UIs as illustrated in Figure 8-6 (b). The process view displays the main navigation of UI prototypes. Users can select and initiate their preferred UI alternatives from the process view. Once users click a UI alternative, the structure of the business process is displayed in the navigation view. The task window of the initial task is launched in the task window area. Meanwhile, the support data operations of the task are opened in the support window area.

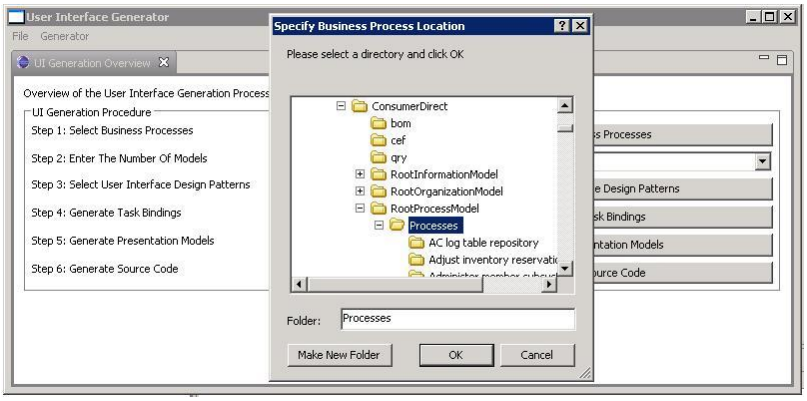


Figure 8-7: The dialog for loading business processes

8.2.1 Generating UIs

Using the UI generation tool, we generate UIs in 4 steps:

1) Analyze business processes to infer task models that are independent from various business process modeling languages.
2) Generate dialog models from task models by performing transformation rules presented in Chapter 6.
3) Generate presentation models from dialog models by applying task patterns and UI design patterns as discussed in Chapter 6.
4) Generate UI code skeletons from presentation models by matching AIOs in presentation models with concrete widgets in a specified platform.

8.2.2 Features of the UI Generation Tool

The current version of our UI generation tool offers the following features: automatic requirement extraction, automatic model generation, usability guided model transformation, and automatic code generation.

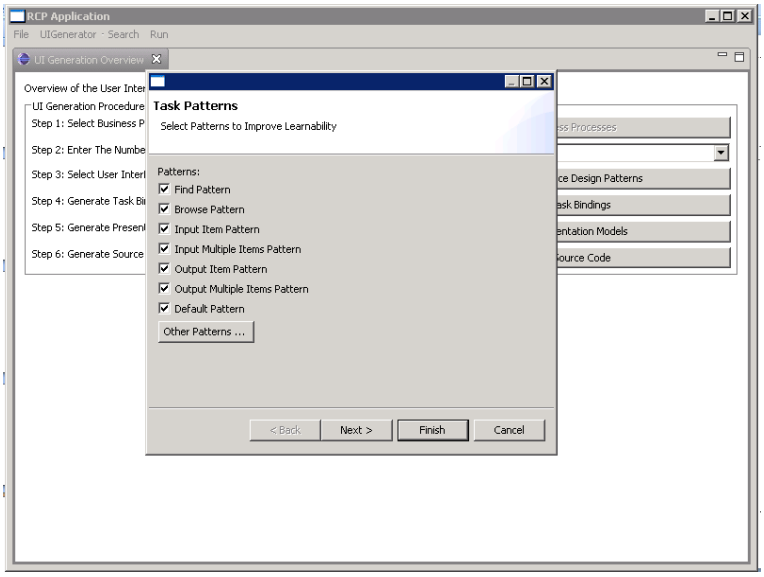


Figure 8-8: The wizard for customization

Automatic Requirement Extraction: To generate UIs, we need to identify business functions to be implemented from business processes. The UI generation tool automatically extracts requirements from business processes as discussed in Section 8.1.1. We provide a dialog for users to load business processes from specified directory as shown in Figure 8-7.

Automatic Model Generation: Our UI generation tool automatically generates UI design models by applying predefined model transformation rules, task patterns, and UI design patterns. In addition, we allow users to use their own design patterns to generate UIs that are in conformance with application specific requirements. We provide a wizard for users to customize transformation rules and design patterns as shown in Figure 8-8.

Usability Guided Model Transformation: To generate UIs in conformance with users' expectations, we incorporate users' expectations to our UI generation process. We first convert users' usability requirements to constraints on UI models and then use the identified constraints to evaluate impact of transformation rules on the usability of UI models. The evaluation results are

shown in a tree view for developers to preview and assess as shown in Figure 8-9. Using the tree view, developers can check usability evaluations of each business process. Once usability problems are identified, users can easily locate the causes of usability problems by reviewing the corresponding business processes.

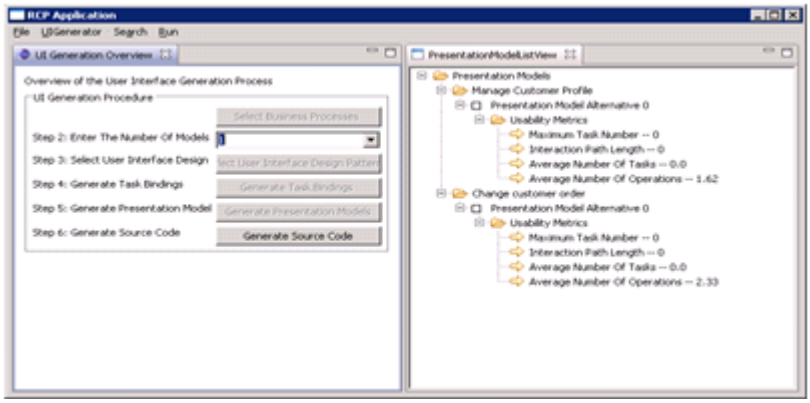


Figure 8-9: The tree view for showing usability evaluation results

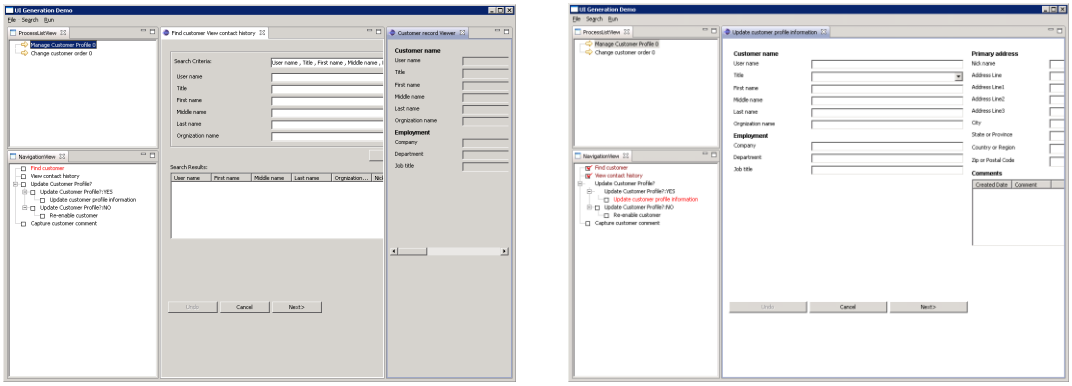


Figure 8-10: Examples of generated UIs

Automatic Code Generation: Using presentation models, our UI generation tool can automatically generate UI code skeletons on a particular platform. In our prototype tool, we provide a collection of code templates for generating UIs on the Eclipse RCP platform. Once presentation models are generated, we combine the AIOs in the generated presentation models

with predefined code templates to generate UI code skeletons. The generated UI skeletons can be previewed using the testing tool shown in Figure 8-6. Figure 8-10 illustrates two examples of generated UI skeletons.

8.3 Summary

In this chapter, we walkthrough our prototype tools for generating software architecture and UIs from business processes. These prototype tools are used in our case studies to evaluate the effectiveness of our approach.

Chapter 9

Case Studies

In this chapter, we present our case studies to evaluate the effectiveness of our proposed approaches. Our case studies consist of three parts: 1) evaluate the effectiveness of our approach in generating functional software architecture; 2) assess the effectiveness of our approach in optimizing the quality of software architecture; and 3) examine the effectiveness of our approach in improving usability of UIs.

9.1 Evaluation of Functional Software Architecture

9.1.1 Objectives

Our approach uses clustering techniques to generate software architecture designs from business processes. A promising software clustering process needs to produce meaningful and consistent system decompositions [72]. The objectives of the case study are to evaluate the following aspects:

1) *Authoritativeness* of the generated architecture. It regards the structural similarity between the generated architecture and an authoritative architecture design. One source of authoritative architecture designs is the architecture designed by expert architects. However, descriptions of such architecture designs are often difficult to find. A common source of authoritative architecture designs is to recover as-implemented architecture designs from documentations or existing source code. In our work, we use the as-implemented architecture designs as authoritative designs and compare the generated

architecture with as-implemented software architecture designs to evaluate the authoritativeness of the generated architecture.

2) *Stability* of our proposed approach. It concerns the persistence of the generated architecture when the business processes are evolved to reflect the changing requirements. We want to assess if our proposed approach can produce persistent architecture designs when the business processes are slightly evolved.
3) *Modularity* of the generated architecture. Modularity is one of the internal quality attributes which can directly or indirectly affect the external quality attributes, such as maintainability and reusability [142]. We aim to produce architecture designs with high modularity which can leads better external quality attributes.
4) *The reduction of arbitrary decisions* of the WCA algorithm. Arbitrary decisions are harmful to the quality of the clustering result and should be avoided. We record the number of arbitrary decisions that are made in the WCA algorithm and compare the number with those of the single linkage algorithm (SLA) and the complete linkage algorithm (CLA) to assess the effectiveness of the WCA algorithm in reducing the number of arbitrary decisions.

To achieve the aforementioned objectives, we conduct the case study in five major steps:

1) apply the WCA algorithm to generate software architecture designs from business processes of subject business systems and record the number of arbitrary decisions that are made in the generation process;
2) apply SLA and CLA to generate software architecture and record the numbers of arbitrary decisions made;

3) compare the number of the arbitrary decisions made in WCA with those in SLA and CLA to evaluate the effectiveness of the WCA algorithm in reducing the number of arbitrary decisions;
4) recover as-implemented software architecture designs from various sources (e.g., documentations or source code) of subject business systems;
5) compare the generated software architecture with the as-implemented software architecture to assess the authoritativeness of our generated architecture;
6) analyze the extent to which the generated software architecture is affected by changes in business processes to examine the stability of our proposed approach;
7) evaluate the modularity of the generated software architecture.

9.1.2 Subject Business Applications

We evaluate our proposed approach on two large scale business systems: IBM WebSphere Commerce (WSC) server [48] and Opentaps [110]. The subject business systems are selected as representatives of different development domains: commercial systems and open source systems.

9.1.2.1 IBM WSC Server

IBM WSC server is a commercial platform for building e-commerce web sites and applications. This system implements business processes to support both B2B (business-to-business) and B2C (business-to-consumer) transactions. The business processes for the system are available online [48]. These business processes are classified to 5 categories. Table 9-1 shows descriptions of the functionalities encapsulated in business processes of the 5 categories. Table 9-2 lists the number of tasks, data items, roles, and business processes in each category of business processes specified in IBM WSC. All these business processes are modeled using WebSphere Business Modeler [47]

and stored as XML documents. We developed a business process parser to parse the XML documents and extract entities specified in our meta-model (shown in Figure 3-4).

Table 9-1: Functionality of IBM WebSphere Commerce

Category	**Description of functionality**
Marketing	Facilitate marketing campaigns and activities
Merchandise management	Create, load, and manage products in online stores
Order management	Manage state of orders and components of orders
Customer management	Create and manage customer profiles and member resources
Customer service	Deliver services to customers

Table 9-2: Numbers of entities in IBM WebSphere Commerce

Category	**# Task**	**# Data item**	**# Role**	**# Business process**
Marketing	69	14	3	8
Merchandise management	66	24	6	24
Order management	204	48	14	25
Customer management	17	8	6	7
Customer service	22	13	5	10

9.1.2.2 Opentaps

Opentaps is an open source system for enterprise resource planning (ERP) and customer relationship management (CRM). The system consists of 13 business applications for supporting various organizational business activities such as customer relationship management, warehouse and inventory management, supply chain management, and financial management. The business

processes of Opentaps are not published. In our case study, we automatically recover business processes from the source code of Opentaps using the Business Process Explorer (BPE) developed by Zou et al. [141]. The recovered business processes are specified using the format supported by WebSphere Business Modeler. The recovered business processes also maintain the mappings between tasks in the business processes and the corresponding source code. Table 9-3 summarizes the functionalities of the 13 business applications. Table 9-4 lists the numbers of entities in each business application.

Table 9-3: Functionality of Opentaps

Category	**Description of functionality**
Accounting	Manage agreements, invoices, fixed assets, and general ledger
Content	Administrate product contents, websites, blogging, and forums
CRM	Offer customer services
Ecommerce	Manage shopping cart
Financial	Handle account receivables and account payables
Manufacturing	Manage bill of materials, production tasks, and work orders
Marketing	Process customer segments and marketing campaigns
Order	Deal with orders, quotes, and customer requests
Party	Provide general party management
Product	Support product and catalog management
Purchasing	Manage supply chain
Warehouse	Track inventory
Work effort	Handle time sheets, events, and tasks

Table 9-4: Numbers of entities in Opentaps

Category	# Task	# Data item	# Role	# Business process
Accounting	143	47	5	69
Content	126	34	4	53
CRM	88	41	5	33
Ecommerce	10	10	2	8
Financial	52	24	2	18
Manufacturing	34	20	2	21
Marketing	31	9	3	19
Order	78	41	5	22
Party	85	24	5	23
Product	119	65	4	60
Purchasing	19	4	2	14
Warehouse	47	14	3	24
Work effort	49	7	2	21

9.1.3 Experiment Design

In this section, we describe the experiment design for evaluating authoritativeness, stability, and modularity of generated software architecture designs.

9.1.3.1 Evaluation of Authoritativeness of the Generated Architecture

To assess the authoritativeness of the generated architecture, we compare our generated software architecture with the as-implemented software architecture. We manually recover architectural components to construct the as-implemented architecture. As studied by Tzerpos and Holt [133], the directory structure of an application designates its functional decomposition. We analyze directory structures of the subject systems and map each top level directory to an architectural component. We derive the directory structure of IBM WSC server from the documentations [48]. The directory structure of Opentaps is directly obtained from the source code [110]. In IBM WSC, a class identifier consists of various segments including the information for identifying the system, the packages, and the class name. For example, *com.ibm.commerce.order.commands.OrderCreateCmdImpl* designates a class used in IBM WSC. *com.ibm.commerce* identifies the system itself. *order* denotes a top level package that captures a major functionality of the system. Hence, we identify *order* as an architectural component. *commands* designates a sub-package of the *order* package and can be mapped to a sub-component of the *order* component. *OrderCreateCmdImpl* is the name of a class within the *order* component.

In our generated software architecture, functionalities of components are described by tasks in business processes. To compare our generated software architecture with the as-implemented software architecture, we need to describe functionalities of as-implemented components in the same way. Therefore, we further analyze relationships among directories, classes, and tasks to identify tasks captured in as-implemented software components. As aforementioned, IBM WSC server is a commercial platform without available source code. By consulting the architects from IBM WSC server, each task in a business process is implemented by a command class in the code. We use the naming convention of the classes to recover the corresponding tasks in the business processes. For the example of a class, *OrderCreateCmdImpl*, *Create Order* corresponds

to the name of the task, *Create Order*; and *CmdImpl* indicates the class is the implementation of a task command. Opentaps is an open source software system with available source code. We use the BPE tool [141] to automatically recover tasks from source code of Opentaps.

$$MoJoFM = (1 - \frac{mno(A,B)}{\max(mno(\forall A,B))}) \times 100\ \% \quad (9\text{-}1)$$

A and B are two software architecture designs. mno(A,B) represents the minimum number of *move* and *join* operations required for changing A to B. max(mno(∀ A, B)) denotes the maximum distance to the partition B.

To evaluate the authoritativeness of the generated architecture, we use the MoJoFM metric [143] to compare the generated software architecture designs and the as-implemented architecture designs. The metric counts the number of operations required to change from one software architecture design to another. The definition of the MoJoFM metric is shown in equation (9-1). The metric counts the number of operations required to make one architecture design the same as another. Large numbers of operations indicate high refactoring effort and reveal a low structural similarity between the two versions of software architecture. Two types of operations are used to change software architecture designs: move and join. A move operation moves one task from a component to another. A join operation merges two components to form a new component. The value of MoJoFM is bounded between 0 and 100%. 0 means that two software architecture designs are completely different and 100% indicates that two software architecture designs are exactly the same.

9.1.3.2 Evaluation of Stability of the Generated Architecture

In the business domain, business processes are subjected to changes due to new business requirements or business process optimization activities (e.g., removing bottlenecks and

inconsistency) in the business process design. The minor changes, such as task addition, removal or merging two separated tasks into one, do not have significant impact on the structure of software architecture. In another words, the software architecture needs to be persistent after the changes applied to business processes. Therefore, we examine the stability of the architecture designs by comparing the architecture generated from the initial business processes with the ones generated from the changed business processes.

$$Difference = 1 - MoJoFM = \frac{mno(A,B)}{\max(mno(\forall A,B))} \times 100\ \% \quad (9\text{-}2)$$

A and B are two software architecture designs. mno(A,B) represents the minimum number of *move* and *join* operations required for changing A to B. max(mno(∀ A, B)) denotes the maximum distance to the partition B.

To conduct the stability study, we introduce changes to business processes. It is challenging to enumerate all possible changes in business processes. Instead, we use a random process to introduce disturbances to business processes. To design such a random process, we need to know the types of potential changes and the appropriate amount of the changes for each type. As studied by Tzerpos and Holt [134], a clustering result (i.e., the generated architecture) can be affected by four types of changes: task addition, task deletion, connector (i.e., the control and data transition between tasks) addition, and connector deletion. Other changes, such as modification of task names, and modification of data items have no impact on the generated software architecture, since our clustering process is independent of task names and the internal structures of data items. Moreover, the four types of changes happen with different frequencies. To ensure that the random process correctly reflects practical changes, we assign different weights (shown in Table 9-5) to the four types of changes as suggested by Tzerpos and Holt [134].

Table 9-5: A summary of changes in the stability study

Change type	Weight	Number of Changes of IBM WSC (Total number of tasks=378)	Number of Changes of Opentaps (Total number of tasks=881)
Task addition	50%	$378 \times 1\% \times 50\% \approx 2$	$881 \times 1\% \times 50\% \approx 5$
Task deletion	25%	$378 \times 1\% \times 25\% \approx 1$	$881 \times 1\% \times 25\% \approx 3$
Connector addition	15%	$378 \times 1\% \times 15\% \approx 1$	$881 \times 1\% \times 15\% \approx 1$
Connector deletion	10%	$378 \times 1\% \times 10\% \approx 1$	$881 \times 1\% \times 10\% \approx 1$
Total	100%	5	10

N is the number of tasks in a business application.

In the stability study, we consider only slight changes in business processes. Similar to the work by Tzerpos and Holt [134], we use 1% of changes in the functionally as a magnitude of slight changes. The tasks in business processes capture the functionality. Therefore, the total functionality is evaluated using the total number of tasks in business processes. If our approach for generating architecture is stable, then 1% of functionality changes in the business processes result in no more than 1% differences in the generated software architecture [134]. To compare software architecture designs generated before and after the changes, we use the MoJoFM metric to compare the structural differences between both architecture designs (as illustrated in equation (9-2)). Table 9-5 summarizes the weights for each type of changes and the number of changes made to the business processes. Changes in different parts of business processes can affect the result software architecture designs differently. Hence, we repeated 1000 times of the study. Each time, we randomly introduce different changes.

Dramatic changes, such as removal of entire business processes, are not considered in our case study. When dramatic changes are made, a new architecture would be needed to accommodate

the changes. In this case, we concern the authoritativeness of the generated software architecture, rather than the persistency of the generated software architecture.

9.1.3.3 Evaluation of Modularity, Cohesion, and Coupling of the Generated Architecture

Cohesion and coupling have been used as indicators of modularity for a long time [7][49][50][120][104]. Fenton and Melton [104] presented a metric to measure coupling of software designs. The equation (9-3) shows the metric for evaluating the coupling of two software components, *x* and *y*. In the equation, *i* is a number that denotes the level of coupling and *n* is the number of inter-connections between the two components, *x* and *y*. Since the average of coupling levels does not have a practical meaning, Fenton and Melton suggest using the median of component coupling values to evaluate the coupling of a software architecture. In our work, the generated software architectures are enriched with information on tasks and data items specified in business processes. This information provides detailed description of the functionalities and data structures of software components. Hence, we apply the metric shown in Equation (9-3) to assess the coupling of the generated software architecture by analyzing task and data transitions among generated software components.

$$M(x, y) = i + \frac{n}{n+1} \qquad (9\text{-}3)$$

x and *y* are two software components. *i* is the level of coupling. *n* is the number of inter-connections between the two components *x* and *y*.

We apply the *Module Cohesiveness* (MC) metric presented by Bieman and Kang [74] to measure cohesion of software components. The metrics assess cohesion of a software component by analyzing dependencies among interface data (i.e., inputs and outputs) of a software component. In our work, connectors among software components are generated from transitions

among tasks in business processes. Data items capture information exchanged among tasks. Hence, we identify inputs and outputs for software component by analyzing task transitions across software components and data items along with these task transitions. For the inter-component task transition shown Figure 9-1, we identify *d* as an output of the component, *Component A*, and as an input of the component, *Component B*.

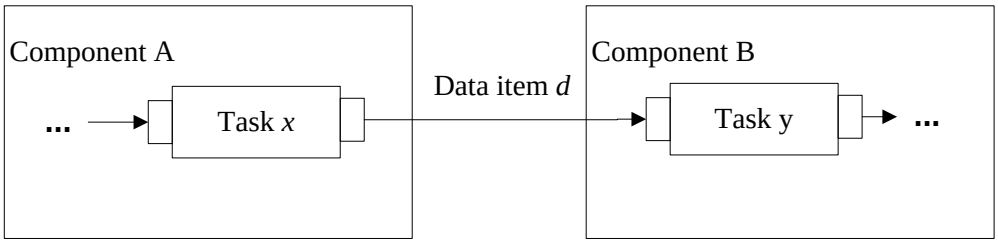


Figure 9-1: An inter-component task transition

$$MC(m) = \frac{\sum_{i=1}^{T} C_i}{T}, C_i = \begin{cases} \frac{N_i - 1}{O - 1} & O > 1 \\ 1 & O <= 1 \end{cases} \qquad (9\text{-}4)$$

C_i is the connectivity of the i^{th} interface data item. N_i is the number of output data items with which the i^{th} interface data item has dependence relation. O is the total number of output data items of the software component.

The metric MC assesses the average connectivity among interface data in a software component as illustrated in Equation (9-4). In the equation, C_i is the connectivity of the i^{th} interface data. C_i is defined as the ratio of the number of outputs that the i^{th} interface data affect and the total number of outputs of the software component *m*. A high MC value indicates that interface data of a software component are strongly correlated. Such a software component presents high cohesion. In this case study, we use the MC metric to compare the cohesion of the generated software architecture and the as-implemented software architecture. To acquire an

overall evaluation of the cohesion of a software architecture, we calculate the mean value of component MC values to provide a summary of the MC values.

9.1.3.4 Evaluation of the Reduction Arbitrary Decisions

In this case study, we compare the WCA algorithm with the SLA algorithm and the CLA algorithm. SLA and CLA are two commonly used hierarchical clustering algorithms in software engineering [129]. The difference between SLA and CLA lies in the rule that is used to update the similarity between the newly generated clusters and existing clusters. We denote the similarity between two clusters C_x and C_y as sim(C_x, C_y). If a cluster C_{xy} is created by merging two clusters C_x and C_y, the similarity between an existing cluster C_i and the newly created cluster C_{xy}, sim(C_i, C_{xy}), is the larger value of sim (C_i, C_x) and sim (C_i, C_y) in SLA. sim(C_i, C_{xy}) is the smaller value of sim (C_i, C_x) and sim (C_i, C_y) in CLA. Table 9-6 shows a summary of the updating rules of SLA and CLA. In this case study, we apply SLA and CLA to group data items in business processes and record the numbers of arbitrary decisions made in the two algorithms. Then, we compare the number of arbitrary decisions in WCA with those in SLA and CLA to evaluate the effectiveness of WCA in reducing the number of arbitrary decisions.

Table 9-6: Updating rules of SLA and CLA

Clustering algorithm	**Updating rule**
SLA	sim (C_i, C_{xy})=Max(sim(C_i, C_x), sim(C_i,C_y))
CLA	sim (C_i, C_{xy})=Min(sim(C_i, C_x), sim(C_i,C_y))

The similarity between two clusters can be calculated using numerous similarity measures as summarized in [129]. It has been observed that the Jaccard measure [144] shown in equation (9-5) give good results for software clustering [107][129]. For two clusters C_x and C_y, a is the

number of features (e.g., an input data item) that both software components present. *b* is the number of features that present in C_x but are absent in C_y. *c* is the number of features that absent in C_x but present in C_y. In this case study, we use the Jaccard measure to calculate the similarities among data items in business processes.

$$J(C_x, C_y) = \frac{a}{a + b + c} \qquad (9\text{-}5)$$

C_x and C_y are two clusters. *a* is the number of features that both C_x and C_y present. *b* is the number of features that C_x presents but C_y absents. *c* is the number of features that C_x absents but C_y presents.

9.1.4 Comparison of As-implemented and Generated Software Architecture

In this section, we give a brief introduction of software architecture designs recovered and generated in our case study.

9.1.4.1 Software Architecture Designs of IBM WSC

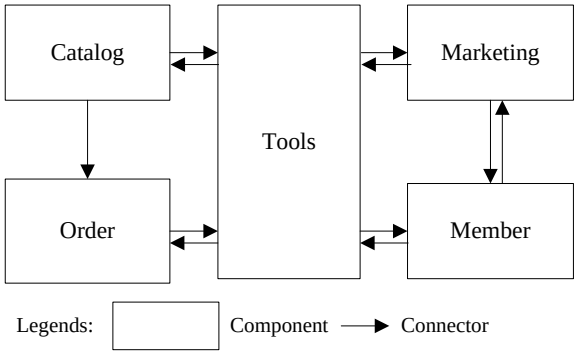


Figure 9-2: The as-implemented software architecture of IBM WSC

To identify the as-implemented software architecture, we study the documentation for the IBM WSC and the APIs provided by IBM WSC server. The documentation describes the functional decomposition of the entire system. Figure 9-2 illustrates the subsystems in IBM WSC and their

relations. We identify components for each subsystem by studying the documentation for the packages and the classes within a package. Figure 9-3 shows more detailed architecture by grouping functionally similar packages into a component within a subsystem. As shown in Figure 9-3, each component captures a primary functionality of IBM WSC server. The name of each component is summarized by studying the functionality of software packages. For example, the *Tools* subsystem provides a set of utility operations to find data items from the databases or create input data items. We have developed a prototype tool to generate software architectures using business processes of IBM WSC. The generated software architecture is shown in Figure 9-4.

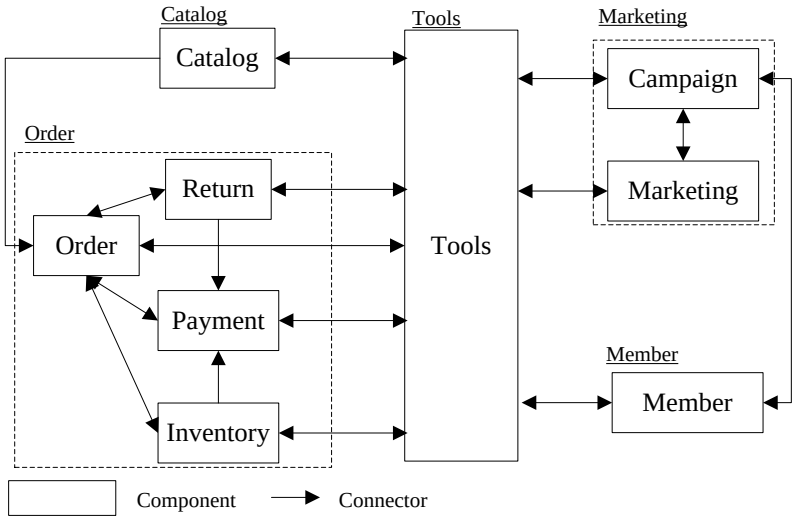


Figure 9-3: An in-depth view of the as-implemented software architecture of IBM WSC

The name of a generated component is identified by studying the description of tasks specified in business processes. An underlined label in Figure 9-3 and Figure 9-4 illustrates the corresponding subsystem which contains the functionally similar components. We assign the same name to the similar components in both architectures. For example, both Figure 9-3 and Figure 9-4 have a component named, *Order*. This indicates that the two components capture the same primary functionality. However, this does not indicate that tasks contained in both

components are identical. We attach a prime sign on the name of each generated component to differentiate generated components from as-implemented components. For example, order processing is closely related to payment handling in IBM WSC, therefore, we group payment handling tasks and order processing tasks into one component, *Order'* in the generated architecture to improve cohesion and reduce coupling in the generated software architecture. Moreover, the differences in the two architectures can be observed by comparing connectors in Figure 9-3 and Figure 9-4. The generated software architecture shown in Figure 9-4 contains fewer connectors than the as-implemented software architecture shown in Figure 9-3. Hence, we envision that our generated software architecture present lower coupling than the as-implemented software architecture.

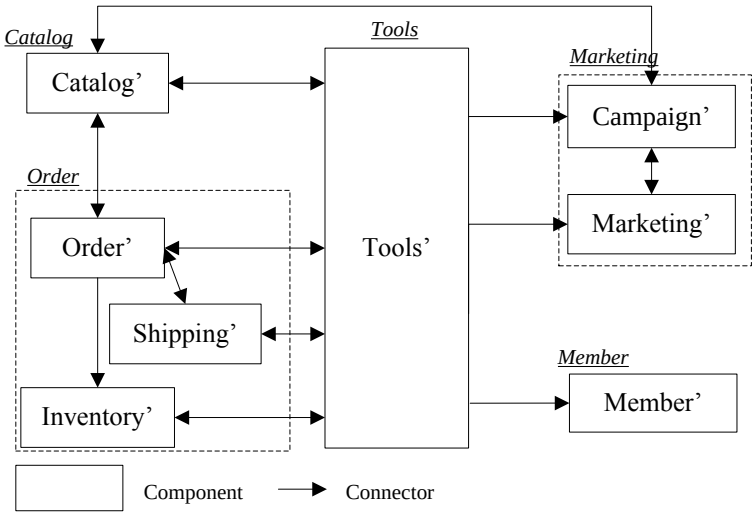


Figure 9-4: The generated software architecture of IBM WSC

9.1.4.2 Software Architecture Designs of Opentaps

We use our prototype tool to generate a software architecture design from the recovered business processes of Opentaps. Figure 9-5 shows the as-implemented software architecture of Opentaps.

Connectors between components are identified from transitions among tasks. Figure 9-6 illustrates the generated software architecture. Comparing Figure 9-5 and Figure 9-6, we find that the generated software architecture contains fewer components than the as-implemented software architecture. Two pairs of components, *(Party, CRM)* and *(Ecommerce, Order)*, in the as-implemented software architecture are merged in the generated software architecture. A new component, *Invoice*, is created from tasks related to invoice processing within the as-implemented component, *Accounting*. Tasks related to shipment are grouped to form a new component, *Shipment*. Tasks in the component, *Purchasing*, in the as-implemented software architecture are distributed to the *Catalog* component and the *Shipment* component in the generated software architecture.

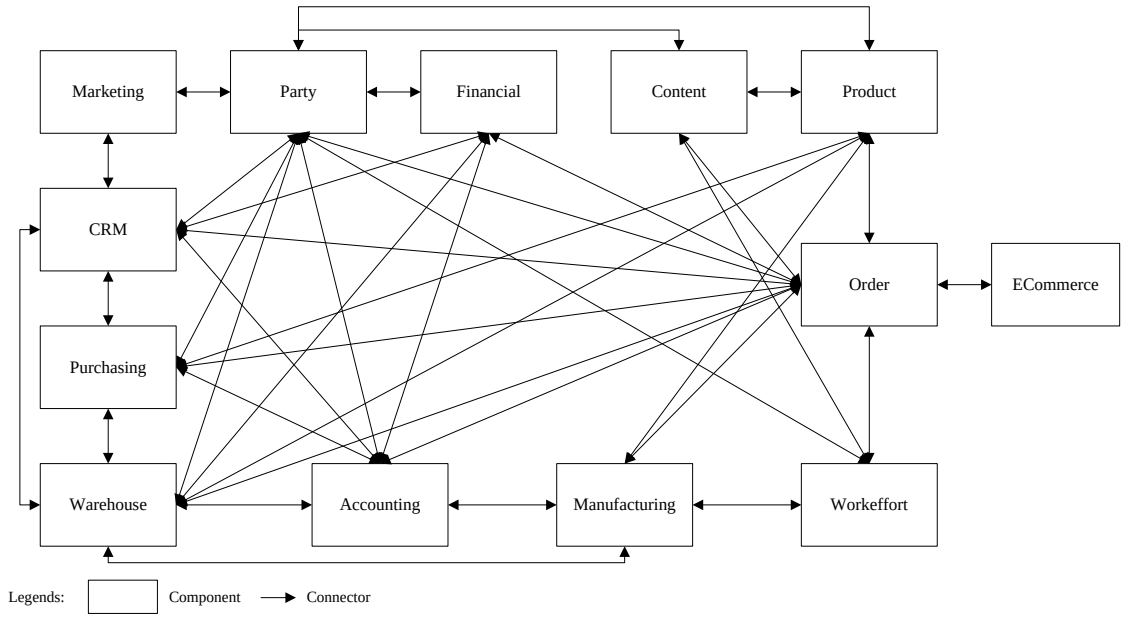


Figure 9-5: The as-implemented software architecture of Opentaps

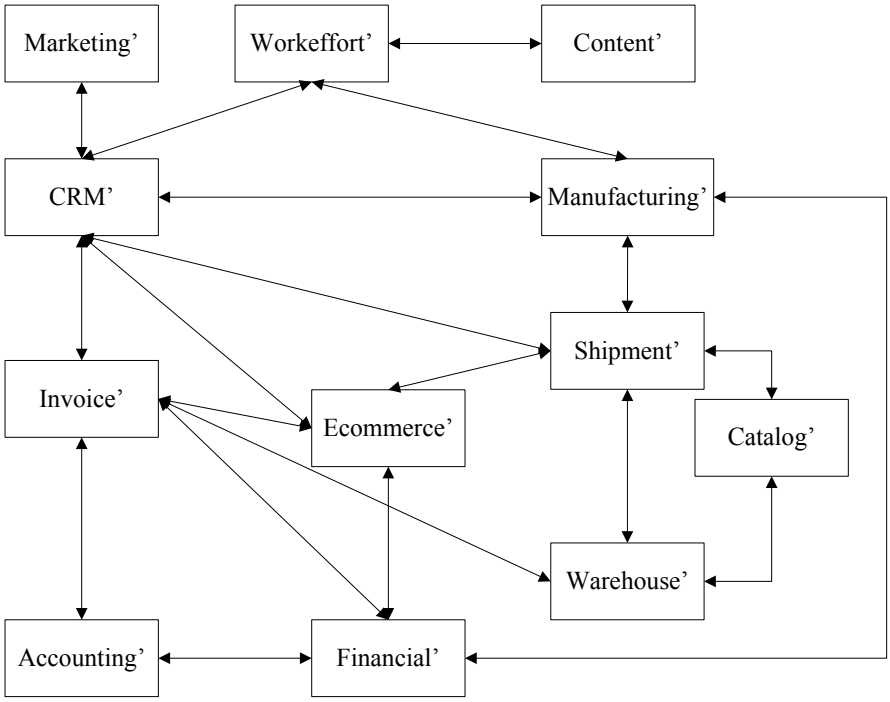


Figure 9-6: The generated software architecture of Opentaps

9.1.5 Analysis of Experiment Results

9.1.5.1 Result of Authoritativeness Evaluation of the Generated Architecture

We calculate the MoJoFM value to evaluate the structural similarity between as-implemented architecture designs and generated software architecture designs. Table 9-7 lists the results of structural similarity of both architecture designs for each subject application. As shown in Table 9-7, the MoJoFM value is 72% for IBM WSC. It indicates that 28% of tasks in the as-implemented software architecture are moved to form the generated software architecture. The MoJoFM value is 76% for Opentaps. This value shows that 24% of tasks in the as-implemented architecture are moved to produce the generated software architecture. As discussed by Wen and

Tzerpos [143], such values are desirable and show that the components generated by our proposed approach are consistent with the as-implemented architectures.

Table 9-7: Results of authoritativeness and stability studies

Application	Authoritativeness	Stability
IBM WSC	72%	96%
Opentaps	76%	93%

9.1.5.2 Result of Stability Evaluation of the Generated Architecture

The stability of the generated architecture is evaluated and the results are listed in Table 9-7. As reported in [134], the clustering process can generate persistent software architecture when at least 80% software architecture designs are structurally similar. In our study, the stability value is 96% and 93% among 1000 times of random changes for IBM WSC and Opentaps, respectively. The results indicate that our approach can produce persistent software architecture designs given 1% of functionality changes in business processes.

9.1.5.3 Result of Modularity Evaluation of the Generated Architecture

Table 9-8: Results of the modularity study

	IBM WSC		**Opentaps**	
	As-implemented	**Generated**	**As-implemented**	**Generated**
Cohesion	0.49	0.74	0.34	0.90
Coupling	3.83	3.75	3.86	3.86

We measure cohesion and coupling to evaluate the modularity of as-implemented and generated software architectures. Table 9-8 lists coupling and cohesion values of IBM WSC and Opentaps.

From Table 9-8, we can observe that our approach can significantly improve the cohesion of software architecture. The mean cohesion values are 0.49 and 0.34 for the as-implemented software architectures of the two subject systems. Such values indicate that the majority of interface data do not have dependence relation with outputs of software components. Hence, the cohesion in both as-implemented software architectures is lower. The mean cohesion values are 0.74 and 0.90 for the two generated software architecture, respectively. This indicates that the majority of interface data have higher dependence relation with outputs of software components. Therefore, applying our approach can greatly improve the cohesion of software architecture. The median coupling values are 3.83 and 3.86 for the as-implemented software architectures of the two subject systems. 3 indicates that the level of coupling is control coupling. 0.83 and 0.86 are strengths of control coupling in as-implemented software architectures. The coupling strength is reduced in the generated software architecture of IBM WSC but is remained the same in that of Opentaps. Considering that the number of components is reduced in the generated software architecture of Opentaps, we deem that such a coupling value is still promising and our approach can reduce the coupling of software architecture.

With high cohesion and low coupling, components can be easily understood, implemented, and changed with little effect on other components. Therefore, increasing cohesion and decreasing coupling can reduce the development cost for applications and improve external quality attributes, such as maintainability, reusability, and flexibility [15][45].

9.1.5.4 Result of the Reduction of Arbitrary Decisions

We record the numbers of arbitrary decisions that are made in the three clustering algorithms as listed in Table 9-9. As shown in Table 9-9, we observe that WCA makes fewer arbitrary decisions than SLA and CLA. WCA makes 4 arbitrary decisions for both subject systems. SLA makes 11

arbitrary decisions for IBM WSC and 5 arbitrary decisions for Opentaps. CLA makes 19 and 22 arbitrary decisions for IBM WSC and Opentaps, respectively. Therefore, we can conclude that WCA is more effective in reducing information loss and decreasing arbitrary decisions.

Table 9-9: Arbitrary decisions made in the three algorithms

Subject system	**WCA**	**SLA**	**CLA**
IBM WSC	4	11	19
Opentaps	4	5	22

9.1.6 Threats to Validity

We discuss the influences that might threat the validity of our case study. We assess three types of threats to validity: construct validity, internal validity, and external validity.

Construct validity concerns whether the selected metrics are appropriate for the purpose of our case study [34]. A common technique to evaluating a clustering algorithm is to compare its output with an authoritative clustering result [143]. MoJoFM is a metric specifically designed to compare the structural similarity of two clustering results. Cohesion and coupling are the major concerns in the modularization stage of a software system. The MQ metric is used to evaluate the modularity of software architecture. However, both metrics are relative measures and their effectiveness depends heavily on the quality of the benchmark software architecture. Although the success of IBM WSC and Opentaps ensures that their software architectures are well-designed, we cannot assure that the software architectures are optimal. In the future, we plan to further evaluate the effectiveness of our proposed approach by comparing the generated software architecture with expert decompositions created by other software architects. Moreover, we assume that the possible changes in business processes are less than 1% in a couple of days. The

1% magnitude is suggested by Tzerpos and Holt [134] based on their experience. We plan to study more changes and figure out the threshold of the changes that would affect the stability.

Threats to **internal validity** are factors that can affect the accuracy of our observations [34]. We predict interactions among components from transitions among tasks in our case study. However, the mapping between task transitions and component interactions may not necessarily one to one mappings. In the implementation stage, one task transition might be mapped to multiple interactions among components (e.g., a handshake process in network protocols). Moreover, one component interaction can be implemented as multiple interactions among tasks. In the future, we aim to further assess the effectiveness of our approach by clarifying the coherence between task transitions and component interactions.

Threats to **external validity** are factors that can hinder the generalization of our conclusion [34]. We conduct our experiment on IBM WSC and Opentaps. The success of the two systems can ensure that they can be considered as representatives from the business application domain. We envision that our approach can be customized to generate software architecture for other business applications. However, we plan to further assess the generality of our approach by investigating the feasibility of our approach in generating software architectures from the business processes specified for other domains.

9.2 Evaluation of Optimized Software Architecture

9.2.1 Objective and Experiment Design

In this case study, we aim to examine the effectiveness of our proposed approach in optimizing the modifiability of software architecture. To achieve this objective, we conduct our case study in four major steps:

1) generate functional software architecture designs by analyzing dependencies among tasks and data items in the subject business processes described in Section 9.1.2;
2) apply the three-tier software architectural style to optimize the modifiability of the functional software architecture designs;
3) evaluate the modifiability of the two versions of software architecture;
4) compare the modifiability before and after the optimization.

9.2.2 Comparison of Functional and Optimized Software Architecture

In this section, we give a brief introduction of functional and optimized software architecture designs in our case study.

9.2.2.1 Software Architecture Designs of IBM WSC

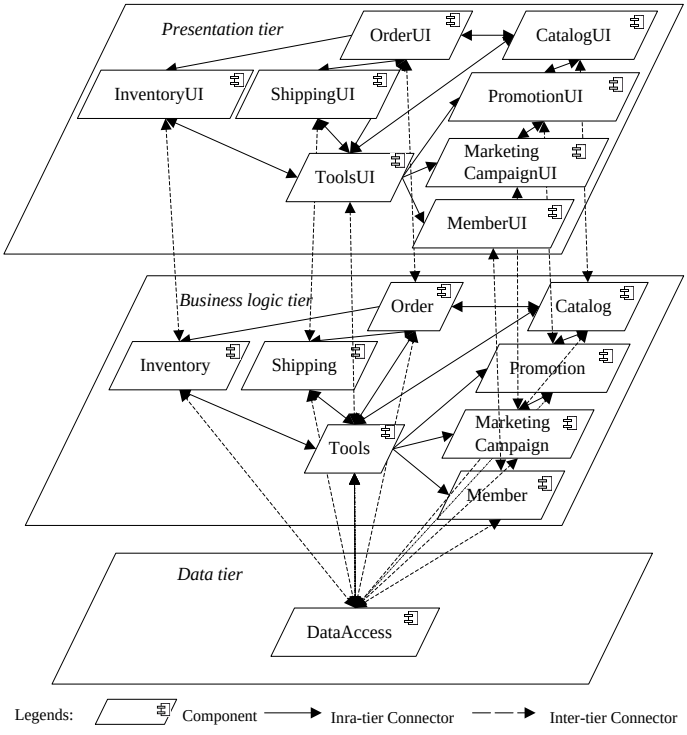


Figure 9-7: The three-tier software architecture of IBM WSC

Figure 9-3 and Figure 9-7 show the two versions of software architecture of IBM WSC, respectively. Figure 9-3 illustrates the initial functional software architecture that is generated by decomposing the functionality specified in subject business processes. Each component captures a set of functionally similar tasks. Interactions among the components are identified from the control flows and data flows between tasks. For example, the component, *Tools*, encapsulates a collection of utility tasks for accessing data repositories or external data. The component, *Order*, encloses the tasks for creating and manipulating orders. Figure 9-7 shows the three-tier software architecture produced by applying the three-tier architectural style to the initial architecture shown in Figure 9-3. By breaking the architecture into three tiers, we aim to reduce couplings among components and improve the modifiability of the business application.

9.2.2.2 Software Architecture Designs of Opentaps

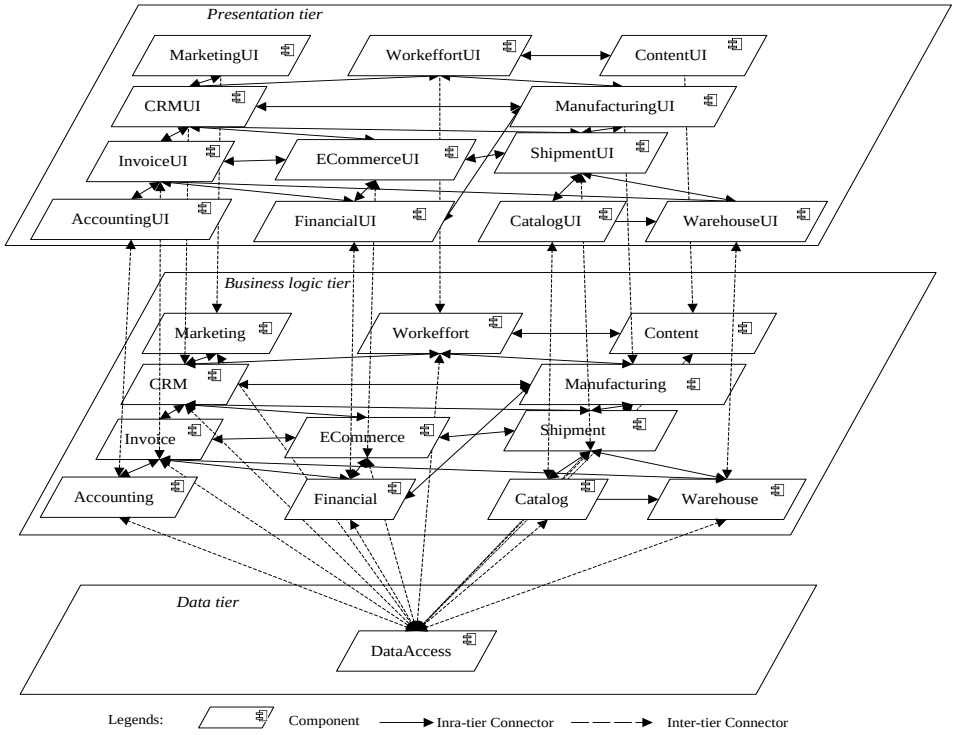


Figure 9-8: The three-tier software architecture of Opentaps

Figure 9-6 and Figure 9-8 illustrate the functional software architecture and the three-tier software architecture of Opentaps, respectively. The functional software architecture shown in Figure 9-6 is generated by analyzing dependencies among tasks and data items specified in business processes of Opentaps. The three-tier software architecture in Figure 9-8 is produced by applying the three-tier software architecture to optimize the modifiability of the functional software architecture shown in Figure 9-6.

Table 9-10: Metric values of IBM WSC

Metric	**Functional SA**	**Three-tier SA**	**Improvement**
Median function point	368	231	37.2%
Cohesion	(Procedural cohesion, 0.0467)	(Procedural cohesion, 0.0646)	38.3%
Coupling	(Common coupling, 0.0111)	(Common coupling, 0.0023)	79.4%
	(Control coupling, 0.0024)	(Control coupling, 0.0009)	62.5%
	(Data structured coupling, 0.0024)	(Data structured coupling, 0.0005)	79.2%
	N/A	(Message coupling, 0.0238)	N/A

9.2.3 Analysis of Experiment Results

Table 9-10 and Table 9-11 show the modifiability values of the two versions of software architecture of IBM WSC and Opentaps, respectively. Each row of the two tables shows the results of a metric. Column 2 and Column 3 illustrates metric results corresponding to the initial

functional software architecture and the three-tier software architecture, respectively. Column 4 shows the improvement rate in the metric result after applying the quality optimization transformation. As shown in Table 9-10 and Table 9-11, we observe the following:

Table 9-11: Metric values of Opentaps

Metric	Functional SA	Three-tier SA	Improvement
Median function point	281	165	41.3%
Cohesion	(Procedural cohesion, 0.1134)	(Procedural cohesion, 0.1581)	39.4%
Coupling	(Common coupling,0.0111)	(Common coupling, 0.0034)	69.4%
	(Control coupling, 0.0022)	(Control coupling, 0.0007)	68.2%
	(Data structured coupling, 0.0018)	(Data structured coupling, 0.0004)	77.8%
	N/A	(Message coupling, 0.0119)	N/A

1) The numbers of function points in the three-tier architecture designs are reduced. We use the median of function points to assess the functionality distribution among components. Components in the three-tier software architecture designs have fewer function points than those in the functional software architecture designs. It indicates that components in the three-tier software architecture designs are simpler than those in the functional software architecture designs.

2) The cohesion levels in both versions of software architecture are procedural cohesion. However, the strength of procedural cohesion of the three-tier architecture is 0.0646 in IBM

WSC and 0.1581 in Opentaps. Such values are higher than those of the functional software architecture designs.

3) Both versions of software architecture designs present common coupling, control coupling, and data structured coupling. The coupling strengths of IBM WSC are 0.0023, 0.0009, and 0.0005, respectively. The coupling strengths of Opentaps are 0.0034, 0.0007, and 0.0004, respectively. All of these values are lower than those of the functional software architecture designs. This indicates that the three-tier software architecture designs have lower coupling than the functional software architecture designs. In addition, the three-tier software architecture designs further reduce coupling among components by introducing a new low level coupling, message coupling. We use N/A to denote that the functional software architecture designs do not present the message coupling. The introduction of message coupling can greatly improve the modifiability of the three-tier software architecture [40].

As shown in Table 9-10 and Table 9-11, we conclude that the three-tier software architecture designs are more modifiable than the functional software architecture designs.

9.3 Usability Evaluation

9.3.1 Objectives

The objectives of this case study are two-fold:

1) Evaluate the effectiveness of our approach in improving the usability of existing UIs. UIs of business application are often designed from the perspective of the IT personnel and lack navigational and contextual support [145]. Such UIs are difficult for users to learn and use. Our approach helps to improve the usability of existing UIs by generating navigational and contextual guidance information from business processes. In this case

study, we conduct a usability study to examine the effectiveness of our approach in improving the usability of existing UIs.

2) Evaluate the effectiveness of our approach in generating UIs with desired usability. Our approach provides assistance in generating UIs to support new business processes. Our approach identifies functional requirements to be implemented by UIs and use intermediate models with increasing details to generate UIs from business processes as discussed in Chapter 6. To generate UIs in conformance with specified usability requirements, we apply usability goal models to guide the generation process and apply UI design guidelines and patterns to optimize the usability of generated UIs. In this case study, we conduct a usability study to assess the effectiveness of our approach in generating UIs with desired usability.

9.3.2 Subject Business Application

The business application evaluated in our usability studies is a sales center application that is an early version of a tool being developed for commercial release. This application provides full functionality for a customer service representative (CSR) to interact with customers and sell products over the phone. However, usability of the application's UIs is not optimal, especially to novice users. Generally, CSRs have limited computer experience. Call centers usually experience high turnover rates among CSRs. Therefore, many CSRs can be considered as novice users. In this case, the usability of the UI is a critical issue due to the high cost of training the large number of novice users.

9.3.3 Study Subjects

As pointed out by Virzi [146] and Jacobsen et al. [147], four to five users are sufficient to a usability study. We recruit 12 users with different levels of experience for the first usability study.

Table 9-12 gives an overview of the study subjects. All of the subjects use computers on a regular basis and are familiar with graphical UIs since they use such interfaces to perform daily activities such as checking e-mails, browsing the Internet, and shopping online.

Table 9-12: Breakdown of users

Group name	Experience	Number of users
Expert	Experience users who have used the existing UIs for over one year.	2
Novice-Tutorial	Novice users who take a 15 minutes tutorial.	5
Novice-NoTutorial	First time users who did not receive any tutorial or guidance.	5

Since users with different backgrounds can evaluate UIs differently, we invite both expert users and novice users to compare our generated UIs with existing UIs. More specifically, we invite two users who have used the existing UIs for over 1 year as expert users. The expert users were also involved in our research project and understood the features of the generated UIs. We invite 10 graduate students in computer engineering and computer science as novice users. The novice users have no prior experience of using either of the UIs. We randomly select five users (Novice-Tutorial) in the novice group and give them a 15 min tutorial on both the generated and the existing UIs. The remaining five novice users do not receive a tutorial (Novice-NoTutorial). In the tutorial, we step through both versions of the UI and demonstrate how we fulfill all of the scenarios used in our study to the Novice-Tutorial group. At each step, we explain the information required for the user to perform a task and demonstrate the results expected after the fulfillment of each task. The tutorial is designed to give the five novice users more guidance in using the UIs. During the experiments, an observer sits beside the users and records the major

problems that they encountered and the length of time required for performing the various scenarios.

In the second usability study, we invite 6 graduate students in computer science to evaluate generated UIs using a standard questionnaire. These users have the same backgrounds to the novice users in the first usability study. 6 usability evaluation sessions are conducted to evaluate the usability of the generated UIs. Each user evaluates the generated UIs in an individual session. At the beginning of each session, we give a description of tasks that users need to accomplish and explain questions in the questionnaire to users. Users are given 1 hour to accomplish specified tasks using the generated UIs. Then, users are asked to evaluate the generated UIs using the IsoMetrics^{s} [80] questionnaire as shown in the Appendix.

9.3.4 Experiment Procedures

In the first usability study, we ask the users to carry out three different scenarios using the two versions of UIs. Users need to perform one or two business processes in each scenario. A summary of the scenarios is listed in Table 9-13. We use the *Manage customer profile* and *Create order* business processes in the scenarios. The *Manage customer profile* is a simple process which contains five tasks. The *Create order* is a more complex scenario which contains 15 tasks. A detailed description of these two processes is specified as follows:

- The *Manage customer profile* business process describes the tasks for managing a customer's profile. To keep a customer's profile up to date, a CSR often conducts this business process while performing other business processes related to the customer. Specifically, a CSR first finds a customer profile and views the past contact history of the customer. Depending on the customer's status, a CSR can update a customer's profile. In the end, a CSR may attach comments to the profile.

- The *Create order* business process specifies tasks for a CSR to create an order for a customer. First, a CSR obtains the customer's data during a phone conversation and searches for the customer's record. If the customer has a record in the system, the CSR will follow a sequence of tasks to place an order for the customer. Otherwise, the CSR creates a new record for the customer and places the order for the customer. Placing the order involves finding products, adding products to the shopping cart, and selecting a shipping address.

Table 9-13: Summary of tasks performed in each scenario

Number of scenario	Used business processes	Number of tasks
1	Manage customer profile	5
2	Create order	15
3	Manage customer profile, Create order	20

These two business processes were combined to create three different scenarios:

Scenario 1. Users perform a simple business process, namely, the *Manage customer profile* process. To fulfill this business process in the existing UIs, users need to switch between the menus and the UI components a few times to locate the UI components for the different tasks in the process. We want to evaluate whether users prefer the strict processing order in the generated UIs. The strict order guides users in performing a business process while limiting them to specific alternative paths.

Scenario 2. Users perform a complex business process, namely, the *Create order* process, which contains more tasks and control flow structures. Using this business process, we want to evaluate the effect of dynamically hiding, viewing, opening, and closing UI components, and the usage of the "next" button for guiding users through tasks in a complex business process.

Scenario 3. The user simultaneously performs two different instances of the *Manage customer profile* and *Create order* processes. For the generated UIs, supporting viewers (that is, a non-editable window) such as promotions viewers are only visible when the user is working on an instance of the *Create order* business process. The promotions viewer is hidden when the user is working on an instance of the *Manage customer profile* process. Similarly, the customer's profile information viewer is only displayed when the user is working on an instance of the *Manage customer profile* process. Therefore, the correct UI components are displayed according to the context of the running process instances. In this scenario, we want to evaluate if our generated UIs can help users manage multiple process instances simultaneously.

In the second usability study, we apply a standard questionnaire to evaluate the usability of generated UIs. A pilot study shows that the majority of windows are reused across business processes. Hence, it is possible to evaluate all windows by choosing a collection of representative business processes. Moreover, the pilot study illustrates that it takes more than one day to conduct an exhaustive usability study on UIs of the sales center application. Such evaluation involves a large amount of duplicated evaluation of reused windows. To save users' efforts, we choose a collection of representative business processes as subject business processes to conduct the questionnaire based usability study. More specifically, our choice of subject business processes is motivated by the following factors:

1) UIs of these business processes covers instances of all task patterns;
2) The subject business processes are frequently used to deliver services to customers;
3) The subject business processes represent business processes at various complexity levels.

Eventually, we select the four business processes shown in Table 9-14 as subject business processes of this experiment. UIs of the chosen business processes can be evaluated in 4 hours.

Table 9-14 summarizes the number of tasks in each subject business process. Table 9-15 shows the rules and patterns that we used for generating UIs with high learnability.

Table 9-14: The numbers of tasks in chosen business processes

Business process	**Name**	**Number of tasks**
BP_1	Change customer order	3
BP_2	Manage customer profile	5
BP_3	Place order	13
BP_4	Create quote	24

1) The business process, *Change customer order*, describes tasks for CSRs to remove errors in orders. CSRs first locate the order with errors and then update the order as required to remove errors or obsolete data. Finally, CSRs submit the order for processing.
2) The business process, *Manage customer profile*, contains tasks for CSRs to update existing customers' personal information and status. CSRs first find a customer using the name of the customer and then check the customers' contact history to learn what kind of update is required. CSRs can update customers' personal information or customers' status.
3) The business process, *Place order,* is composed of tasks for CSRs to buy products. At first, CSRs find products according to customers' requirement. Then, CSRs create an order and enter customers' payment information and shipping information. Finally, CSRs submit the order for processing. CSRs can also record customers' comments using tasks in this business process.
4) The business process, *Create quote,* is comprised of tasks for CSRs to construct quotes for customers. The business process allows CSRs to transform orders to quotes, create quotes

by adapting previous quotes, and create quotes from scratch. Created quotes are saved in the database.

Table 9-15: A summary of rules and patterns for generating UIs with high learnability

Process	Task model	Dialog model	Presentation model
BP_1	Role rule, Manual task rule, Data sharing rule	Data window rule	Input task pattern, Find task pattern, Action task pattern
BP_2	Role rule, Branch rule, Manual task rule	Data window rule	Input task pattern, Output task pattern, Find task pattern, Action task pattern
BP_3	Role rule, Data sharing rule, Optional task rule	Data window rule	Input task pattern, Output task pattern, Find task pattern, Browse task pattern, Action task pattern
BP_4	Role rule, Branch rule, Data sharing rule, Manual task rule, Optional task rule	Data window rule	Input task pattern, Output task pattern, Find task pattern, Browse task pattern, Action task pattern

9.3.5 Evaluating Usability

9.3.5.1 Evaluation Criteria for Usability

For the first usability study, we apply our approach to enhance the usability of UIs of a business application which is fully developed. Users can perform tasks using functionalities provided in the business application. Hence, we use a set of usability metrics to quantitatively assess each of these usability attributes. More specifically, usability is measured in terms of the following attributes: error rate, memorability, efficiency, user satisfaction, and learnability.

Error rate measures the rate of cancellation and system rejection. We record the number of times that a user cancels a UI component such as a window or a dialog when trying to locate the UI component needed to perform a task in a business process. We also track the number of times that the application rejects a user's selection of a UI component. Each rejection is indicated by a warning message. For example, if a user submits an order without providing the payment information, then the system would reject the submission and shows a warning message. We measure the number of cancellations and system rejections for each user after a scenario is completed. To compare different scenarios, we normalize the error rate over the total number of tasks performed in a scenario. UIs with low error rates are desirable.

Memorability can be measured by using retention over time [9]. In practice, help and feedback prompts are used for improving the memorability of a UI. We adapt two metrics to evaluate memorability: the percentage of tasks that a user requests for help while fulfilling the business processes (Help%) and a mean recognition score (MRS) [21][94]. The Help% metric is derived by measuring the number of times that an application displays a context specific help messages that is triggered by the user requesting help. The number is divided by the total number of tasks in a scenario. UIs with high memorability have low Help%. MRS determines the accuracy with which an application can automatically prompt users with the correct dialogs or windows while users are performing consecutive tasks. MRS is produced by counting the number of correctly prompted dialogs or windows. We normalize the MRS by dividing it by the total number of tasks in a scenario. A high MRS indicates that UIs have high memorability since users need to memorize fewer UI features.

Efficiency can be measured directly by using the time needed for a user to complete a business process (that is, elapsed time) or indirectly by measuring the performance of a user. Kamm et al.

[21] propose that high learnability, low memorability, and low error rate have a positive impact on the performance. For example, requesting help has a negative impact on the performance. We measure the performance of the user by using equation (9-7), which is proposed by Kamm et al. [21]:

$$Performance=0.25*MRS+0.33*SuccessRate+0.33*Help\% \quad (9\text{-}7)$$

MRS is the Mean Recognition Score, SuccessRate refers to the rate of successfully completed tasks, and Help% measures the percentage of tasks that a user requests help while performing a scenario.

User satisfaction is a subjective measure of the comfort and acceptability of use by users. To provide data on user satisfaction, users in our study completed the following short survey. We indicate the rationale for asking each question in brackets. The users were not shown the rationale for the questions. The survey contains the following questions:

- Were the UI components needed to launch a business process easy to locate? (process list performance)
- Did the system provide the information that you want to see? (context awareness performance)
- In the interaction, was it easy to locate the next UI component to perform the task that you want? (ease of navigation)
- During the interaction, did you know what you can do at each point of the interaction? (user's knowledge of the UI)
- Did the system work as you expect? (expected system behavior)
- Would you use the system regularly in the future? (future use)

The survey has multiple-choice Likert scale responses ranging over values such as almost never, rarely, sometimes, often, and almost always. To produce a numerical score, each survey response is mapped into the range of 1 to 5, with five representing the greatest degree of

satisfaction. All of the responses are summed up to produce a user satisfaction measure which ranges from 6 to 30 for each scenario.

Learnability measures the time needed to learn a UI [9]. We are interested in understanding the ease of learning our business process driven UIs in comparison to the existing UIs. To measure the learnability of a UI, we compare the time needed for an expert to perform the same scenario relative to the time needed for a novice user to perform a scenario. For example, if an expert takes, on average, 50 sec to perform a scenario, while a novice user takes 100 sec to perform the same scenario, then the learnability of the UI is 0.5. UIs with high learnability are desirable for novice users such as CSRs.

9.3.5.2 Questionnaire

For the second usability study, we generate UI prototypes from business processes of the subject application. The generated UIs can accept inputs from users but do not produce responses to users' inputs. Hence, it is challenging to measure the usability of generated UIs using the metrics discussed in Section 9.3.5.1. Questionnaire is a commonly used approach for evaluating usability. The IsoMetrics questionnaire [80] presents questions that map the 7 characteristics (shown in Table 9-16) of ISO 9421 part 10 to properties of an application. There are two versions of the IsoMetrics questionnaire: IsoMetrics^{s} and IsoMetrics^{l}. The IsoMetrics^{s} supports summative evaluations that use numeric scores to quantitatively assess the usability of an application. The IsoMetrics^{l} provides support for formative evaluations that collect weaknesses of UIs to qualitatively assess the usability of an application. To enable objective comparison of evaluation results, we choose the IsoMetrics^{s} questionnaire as shown in Appendix to quantitatively evaluate the usability of generated UIs. During the usability study, users give a score to each question in

the scale from 1 to 5. 1 means predominately disagree and 5 denotes predominately agree. A no-opinion option is provided to avoid random marking.

Table 9-16: Usability characteristics from ISO9421 part 10

Characteristics	Evaluation guidelines
Suitability for the task	The UI of a task is suitable for the task if it supports users to perform the task effectively and efficiently.
Self-descriptiveness	A UI is self-descriptive if it is understandable in an intuitive way and provides sufficient context support.
Controllability	UIs are controllable if users can easily start UIs and can determine the direction and speed of their operations.
Conformity with user expectations	UIs are in conformance with user expectations when they have consistent look and feel and comply with the backgrounds of users.
Error tolerance	UIs are error tolerant when users can accomplish tasks with little or no additional effort in resolving errors.
Suitability for individualization	UIs are suitable for individualization when users can customize UIs with respect to tasks, capabilities of users, and preferences of users.
Suitability for learning	UIs are suitable for learning when users can easily understand and use UIs.

9.3.6 Analysis of Experiment Results

9.3.6.1 Comparison with Existing UIs

Table 9-17, Table 9-18, and Table 9-19 are the descriptive statistics for the three scenarios of the three groups of users (that is, Expert, Novice-Tutorial, and Novice-NoTutorial). For each user

group, the table shows the means of the various usability metrics for that particular user group. Simply comparing the means of the various metrics is not sufficient since the differences may be due to the natural variability of the data and may not be statistically significant. Instead, we statistically test the results of our study. In particular, we formulate the following hypotheses for our study:

1. Usability and type of UI. The usability of the generated UIs is higher in comparison to the usability of the existing UIs for novice users. However, the usability of the generated UIs is lower for expert users.

2. Usability and tutorial. A tutorial does not increase the usability of the generated UIs for novice users. However, a tutorial increases the usability of the existing UIs for novice users.

Table 9-17: Results of usability evaluation for scenario 1

Usability attribute	**Metrics**	**Expert users**		**Novice-tutorial**		**Novice-NoTutorial**	
		UI_e	**UI_g**	**UI_e**	**UI_g**	**UI_e**	**UI_g**
Error rate	**Error rate**	0	0	0	0	0	0
Memoriability	**Help%**	0	0	0	0	0.4	0
	MRS	0.2	0.8	0.2	0.8	0.2	0.8
Efficiency	**Elapsed time (sec)**	70	76	153	122	247	127
	Success rate	1	1	1	1	1	1
	Performance	1.05	1.20	1.05	1.20	0.92	1.20
User satisfaction	**Subjective satisfaction**	23	28	21	27	16	26
Learnability	**Learnability**	1	1	0.46	0.62	0.28	0.60

UI_e represents existing UIs. UI_g represents generated UIs.

Table 9-18: Results of usability evaluation for scenario 2

Usability attribute	Metrics	Expert users		Novice-tutorial		Novice-NoTutorial	
		UI_e	UI_g	UI_e	UI_g	UI_e	UI_g
Error rate	**Error rate**	0	0	0.07	0	0.13	0
Memoriability	**Help%**	0	0	0.13	0.07	0.27	0
	MRS	0.20	0.87	0.20	0.87	0.20	0.87
Efficiency	**Elapsed time (sec)**	224	245	480	360	825	380
	Success rate	1	1	1	1	1	1
	Performance	1.05	1.22	1.01	1.19	0.96	1.22
User satisfaction	**Subjective satisfaction**	24	29	21	27	17	27
Learnability	**Learnability**	1	1	0.47	0.68	0.27	0.64

UI_e represents existing UIs. UI_g represents generated UIs.

Table 9-19: Results of usability evaluation for scenario 3

Usability attribute	Metrics	Expert users		Novice-tutorial		Novice-NoTutorial	
		UI_e	UI_g	UI_e	UI_g	UI_e	UI_g
Error rate	**Error rate**	0	0	0.05	0	0.10	0
Memoriability	**Help%**	0	0	0.13	0.07	0.50	0
	MRS	0.20	0.85	0.20	0.85	0.20	0.85
Efficiency	**Elapsed time (sec)**	266	291	721	500	1200	520
	Success rate	1	1	1	1	1	1
	Performance	1.05	1.21	1.05	1.21	0.89	1.21
User satisfaction	**Subjective satisfaction**	22	20	20	28	16	28
Learnability	**Learnability**	1	1	0.37	0.58	0.22	0.56

UI_e represents existing UIs. UI_g represents generated UIs.

We test our hypotheses by using statistical tests which assume a significance level of 5 percent (that is, α=0.05). We perform all of our statistical analyses on the data for novice users. Due to the small number of expert users (only two users), we cannot statistically test our results for expert users. However, for reference, we report the metrics for the two expert users in Table 9-17, Table 9-18 and Table 9-19. The metrics for both expert users are consistent, with little variability between both users. For each hypothesis for the novice users, we compare the means of the usability metrics discussed in Section 9.3.5.1.

For our statistical analysis, we conduct parametric and nonparametric tests. We use a t-test [30] for the parametric test and a paired Wilcoxon signed-rank test [30] for the nonparametric test. We study the results of both types of tests to determine whether there are any differences between the results reported by both types of tests. In particular, for non-significant differences reported by the parametric tests, we check if the differences are significant according to the nonparametric Wilcoxon test. The Wilcoxon test helps ensure that non-significant results are not simply due to the departure of the data from the assumptions of the t-test. For the results presented in the following, the results of both types of tests are consistent and we only show the P values for the t-tests.

- **Usability and Type of UI**

For the first hypothesis, we compare the mean of the usability metrics for novice users using the existing UIs $\mu(\text{Metric}_{\text{Existing}})$ and the mean of the usability metrics for novice users using the generated UIs $\mu(\text{Metric}_{\text{Generated}})$. We believe that the generated UIs would increase the usability of the call center application, in particular for novice users. For example, we expect that the learnability metric would increase. Therefore, we expect that the difference in means for the learnability metric would improve. We can then formulate the following test hypothesis:

H_0: $\mu(\text{Learnability}_{\text{Existing}}\text{-Learnability}_{\text{Generated}})=0$,

H_A: $\mu(\text{Learnability}_{\text{Existing}}\text{-Learnability}_{\text{Generated}})=0$.

$\mu(\text{Learnability}_{\text{Existing}}\text{-Learnability}_{\text{Generated}})$ is the population mean of the difference between the learnability metric for each UI. If the null hypothesis H_0 holds (that is, the derived P value>α=0.05), then the difference in mean is not significant and is likely due to the variability of the data. If H_0 is rejected with a high probability (that is, the derived P value≤0.05), then we are confident about the performance improvements of the generated UIs.

Table 9-20: Statistical analysis for usability versus type of UIs

Usability attribute	**Metric**	**$\mu(\lvert\text{Metric}_{\text{Existing}}\text{-Metric}_{\text{Generated}}\rvert)$**	**P(H_0 holds)**
Error rate	**Error rate**	0.0583(+100%)	0.018
Memorability	**Help%**	0.205(+95%)	0.001
	MRS	0.6388(+319%)	0.013
Efficiency	**Elapsed time**	18.819(+43%)	0.04
	Performance	0.227(+23%)	0.017
User satisfaction	**Subjective satisfaction**	8.7(+47%)	<0.001
Learnability	**Learnability**	0.27(+78%)	0.047

We perform similar hypotheses testing on all of the usability metrics. Table 9-20 shows the results of the t-test for the hypotheses of the usability metrics. The table shows the difference in means and the percentage of improvement. To present all of the metrics in a consistent matter, we show the absolute value of the difference since, for some metrics (for example, Help%), a decrease in metric value represents an improvement, whereas, for other metrics (for example, Performance), an increase in value represents an improvement. All statistically significant cells

are colored gray. The results shown in Table 9-20 indicate that the differences in the metrics between both UIs are significantly different from zero. An analysis of the metrics reveals that all metrics have improved in the generated UIs. We can conclude that the usability of the generated UIs is statistically better than the usability of the existing UIs.

Examining Table 9-17, Table 9-18 and Table 9-19, we notice that the expert users take more time to complete all three scenarios for the generated UIs. The existing UIs are designed around a data flow. That is, a UI component such as a window or a dialog is designed for each data object such as a customer, an order, or a quote. Users can enter all of the information related to that data object in a UI component. Inside each UI component, users can use buttons or tabs to switch between different pages in the UI component. An expert user who is familiar with the UI can perform business processes with high efficiency. When two business processes have common tasks, an expert user can fulfill a common task once in the corresponding UI component. The result of the task can be shared with other business processes. However, in the generated UIs, data from common tasks in different business processes cannot be shared. Therefore, common tasks have to be performed repeatedly in each business process. For example, if two separate business processes both contain the task, *Enter User Shipping Information*, then, in the generated UIs, the expert user must perform this task twice instead of performing both processes in parallel and sharing the task data. Due to the inability of the expert user to share task data between processes, the elapsed time for expert users using the generated UIs is longer than the time for the existing UIs. In short, our experimental results show that the generated UIs may slow down expert users since users have to follow a strict order to conduct their tasks. The strict order is not desirable for expert users who may have acquired different workarounds and speedups. Unfortunately, we cannot statistically test this finding due to the limited number of expert users.

Nevertheless, we confirm our findings with the expert users who pointed to their frustration in having to re-enter data and in having to follow the strict order. The expert users expressed their frustration as well in the subjective survey by giving the generated UIs a lower satisfaction rate in scenario 3 (20 for generated UIs versus 22 for existing UIs). Both expert users indicated that they do not prefer using the generated UIs in the future.

For the generated UIs, we have received positive feedbacks from the novice users. The novice users appreciate the strict order in the generated UIs. Users note that they like that the navigation sequence shows only mandatory tasks and that it dynamically adapts to a user's selection. The navigation sequence reduces the complexity of presenting the progress of a process instance and gives users an overview of the remaining steps. The navigation sequence also records the progress and the status of the process instances and is valuable in helping users resume their work, even if they took a break during the evaluation. The context awareness is considered to be desirable, especially for supporting UI components that are mixed with other UI components or which require users to manually initiate them. Novice users do not need to search for UI components themselves. Instead, they simply click on the "next" button and the UI dynamically opens and closes the appropriate UI components. Moreover, the "next" button replaces multiple mouse clicks with one simple click when searching for the subsequent UI components. The "next" button reduced the time required by users to locate UI components and improved their work efficiency.

As pointed out in [90], usability attributes such as ease of learning, maximum efficiency of use, low error rate, and subjective satisfaction conflict with one another. It is challenging to balance the expectations of novice users and expert users in one UI design. A compromise is usually reached in the design of a UI to meet the expectation of the majority of the users of an

application (that is, novice or expert). An application that has a user base with high turnover is best served with a UI designed for novice users. Conversely, an application that is used frequently by a stable user base would require a UI designed for expert users to improve the efficiency.

We believe that supporting novice users has precedence since a large number of users of e-commerce applications tend to be novice users [65]. As business processes evolve over time, the underlying implementation such as the UIs of an e-commerce application must change. Many expert users are likely not to have expert knowledge of all business processes offered by large complex applications. Moreover, an expert may not be an expert after the UIs are updated to accommodate evolving business processes. In our approach, the content of the UI is automatically regenerated from the updated process definitions. Using our approach, the changes to a business process are immediately reflected in the UIs. The strict order of the generated UIs ensures that expert users and novice users do not miss new changes in old processes. Moreover, the generated UIs can make optimal use of screen space by automatically opening and closing the supporting UI components when no data is available. In short, our business process driven UI is beneficial since it reduces the probability of errors to novice and expert users (especially if they are overworked).

- **Usability and Tutorial**

To familiarize users with the UI, half of the novice users were given a tutorial about both UIs. We would like to examine the benefits of conducting such a tutorial to the existing UIs and to the generated UIs. Due to the high cost associated with training users, we hope that our generated UIs can reduce the need for tutorials.

We study the benefits of the tutorial on the usability attributes by comparing the usability metrics for novice users with tutorial and the metrics for novice users without tutorial. We first

compare the metrics for the novice users using the existing UIs to determine if the tutorial was helpful for the existing UIs. We then compare the metrics for the generated UIs to determine whether the tutorial is helpful for the generated UIs. For example, for the learnability metric, we formulate the following test hypotheses to study the benefits of the tutorial on the existing UIs for novice users:

$$H_0: \mu(\text{Learnability}_{\text{ExistingUI\_Tutorial}}\text{-Learnability}_{\text{ExistingUI\_NoTutorial}})=0,$$

$$H_A: \mu(\text{Learnability}_{\text{ExistingUI\_Tutorial}}\text{-Learnability}_{\text{ExistingUI\_NoTutorial}})\neq 0.$$

Table 9-21: Statistical analysis of the effect of tutorial

Usability attribute	**Metric**	**μ(Metric$_{\text{Tutorial}}$-Metric$_{\text{NoTutorial}}$)**	
		Existing UIs	**Generated UIs**
Error rate	**Error rate**	-0.038	0
Memorability	**Help%**	-0.34	0.02
	MRS	0	0
Efficiency	**Elapsed time**	-21.91	-1.11
	Performance	0.113	-0.0007
User satisfaction	**Subjective satisfaction**	4.33	0.33
Learnability	**Learnability**	0.123	0.0201

$\mu(\text{Learnability}_{\text{ExistingUI\_Tutorial}}\text{-Learnability}_{\text{ExistingUI\_NoTutorial}})$ is the population mean of the difference between the learnability metric for novice users with and without tutorial for the existing UIs. If the null hypothesis H_0 holds (that is, the derived P value$>\alpha=0.05$), then the difference in mean is not significant and is likely due to the variability of the data. If H_0 is rejected with a high probability (that is, the derived P value≤ 0.05), then we are confident that the

tutorial is of little value. This analysis is done for the existing UIs and the generated UIs. Table 9-21 is the summary of both analyses. Cells in gray are statistically significant, whereas the other cells are not. The table indicates that the tutorial is valuable for the existing UIs since the tutorial improves the usability for novice users for all usability metrics except for Error Rate and MRS metrics. However, the tutorial does not have any statistically significant effect on the usability of the generated UIs for novice users and is likely not needed.

9.3.6.2 Questionnaire Based Evaluation

Results of the questionnaire based evaluation are listed in Table 9-22. Each row of the table shows scores from one user. Column 2, column 3, column 4, and column 5 depict scores of the four characteristics: *Suitability for the task*, *Self-descriptiveness*, *Conformity with user expectations*, and *Suitability for learning*, respectively. Column 6 shows the average scores in the IsoMetrics[s] questionnaire. All users mark the generated UIs above average as illustrated in Table 9-22. Hence, we believe that the usability of the generated UIs is acceptable.

Table 9-22: The usability evaluation result

User	Suitability for the task	Self-descriptiveness	Conformity with user expectations	Suitability for learning	Average
User 1	***4.933***	***4.833***	***4.649***	***4.875***	3
User 2	***4.467***	***4.500***	***4.353***	***4.750***	3
User 3	***4.867***	***4.750***	***4.630***	***4.750***	3
User 4	***4.533***	***4.500***	***4.405***	***4.375***	3
User 5	***4.800***	***4.833***	***4.575***	***4.875***	3
User 6	***4.733***	***4.833***	***4.583***	***4.875***	3

9.3.7 Threats to Validity

We now discuss the different types of threats that may affect the validity of the results of our experiment.

External validity tackles the issues related to the generalization of the result. Our study asks users to conduct three scenarios on a single e-commerce application. The scenarios are selected in consultation with the developers of the application. The developers feel that these scenarios are the most frequently used scenarios. The e-commerce application is a large and complex application that is currently deployed in an industrial setting. Nevertheless, we should, in the future, study the benefits of our approach by using other applications and other scenarios to determine if our results hold for other applications and domains.

In our experiment, a limited number of expert users participated in our usability study since it is difficult to find users who have used the systems for more than 1 year other than the original developers. In practice, the users of the studied application are novice users with limited computer experience. The novice user groups in our usability study were graduate student volunteers. These volunteers likely have more experience in using computers than a regular CSR. Nevertheless, we believe that the additional experience would make them more comfortable in using and navigating through both versions of the UI. Hence, the usability improvement demonstrated in our experiment is more likely to be a lower bound rather than an upper bound.

Internal validity is a concern with the issues related to the design of our experiment. In particular, we need to account for the order in which users use the UI and the order in which they perform the scenarios. In our experiment, all three user groups complete the scenarios in the same order and always start with the generated UIs. To avoid any bias, the novice users are not informed whether the UIs that they are using are the existing or the generated UIs. We feel that

this setup should not reflect positively on our generated UIs. This setup will also not reflect negatively on the existing UIs. The guidance provided by the generated UIs will likely help users in performing the scenarios using the existing UIs. The users' experience in using the generated UIs should make the users more familiar with the components of the existing UIs.

Users performed the scenarios in the order of their complexity. We feel that the order of scenarios from the easiest (that is, scenario 1 was done first) to the most complex is more realistic since users in an industrial setting are likely to be mentored through easy scenarios before being given more complex scenarios. Scenarios 1 and 2 have no common UI components. However, scenario 3 contains the UI components from scenarios 1 and 2. The two earlier scenarios help users in performing scenario 3.

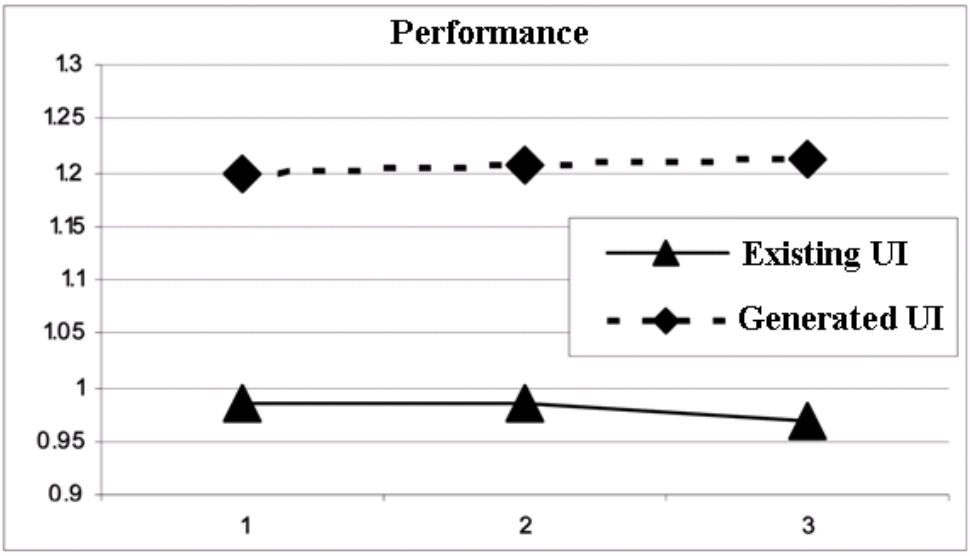


Figure 9-9: Mean values of the performance metric across the three scenarios

To quantify the learning effect, we compared the means of the various usability metrics across the three scenarios for all novice users for both existing and generated UIs. Figure 9-9 shows the means for the performance metrics for all novice users on both UIs. Figure 9-9 shows that there is

a slight improvement in performance going from scenarios 1 to 2, but the performance drops for scenario 3 in the existing UIs. However, the improvements are trivial. To determine if there is a learning effect, we perform a two-way ANOVA [30],α=0.05, which uses a scenario (1, 2, or 3) and a UI type (generated UIs or existing UIs) as independent variables and the usability metrics as dependent variables. The ANOVA results show that the differences in means of all of the usability metrics are statistically significant for the UI type, but the differences are not significant for the scenarios. The differences are not significant for the interaction between scenarios and UI types. These results indicate that the learning effect is not statistically significant. We performed one-way Kruskal-Wallis [30] nonparametric tests by using scenarios as independent variables and the usability metrics as dependent variables. The results of the Kruskal-Wallis tests confirmed that the differences of means for the usability metrics across scenarios are not significant. Although our statistical tests show that there is no visible learning effect between scenarios, we expect, in practice, that there would be a learning effect. We do not think that the learning effect will affect our results since novice users are commonly trained on simple scenarios before being moved to more complex ones.

Users performed the three scenarios by using the generated UIs before performing the scenarios again by using the existing UIs. To prevent users' fatigue affecting the results of our study, users were given a 10 min break between the experiments on the different UIs.

Construct validity is a concern as to the meaningfulness of the measurement. In our study, we measured the usability of a UI by using a variety of metrics from the literature. However, usability is a subjective issue in many cases and we account for this through the use of a subjective survey rather than the use of usability metrics.

9.4 Summary

In this chapter, we present our case studies to evaluate the effectiveness of our approach for generating functional software architecture, optimizing generated software architecture, and optimizing usability of UIs. The results of these case studies demonstrate that:

1) our approach can stably generate meaningful functional software architecture with satisfactory modularity;
2) our approach is effective in optimizing the modifiability of software architecture;
3) our approach can generate UI skeletons with desired usability features.

Chapter 10

Conclusion

In this thesis, we present an approach that provides automated support in aligning business applications with business requirements specified in business processes. The effectiveness of our approach is demonstrated through case studies. In this chapter, we summarize the contributions of this thesis and propose our future work.

10.1 Thesis Contributions

Our work has the following contributions:

1) *automatically generate software architecture from business processes*. We apply clustering algorithms to automatically derive architectural components from business processes and use the MQ metric to choose architectural components with satisfactory modularity. Eventually, our approach can automatically generate meaningful software architecture with desired modularity. Our proposed approach improves the efficiency of software architecture design and can assure the consistency between business requirements and software architecture.
2) *extend existing product metrics to evaluate quality attributes of software architecture*. We use tasks and data items in business processes to describe internal structures of architectural components. In addition, we extend the existing product metrics to measure the quality of software architecture based on properties of tasks and data. Our approach provides a fast prototyping technique for software architecture design and helps software architects to automatically verify the achievement of quality requirements and evaluate quality impact of transformation rules.

3) *automatically generate UIs with desired usability features from business processes*. Business processes are used as business requirement models that capture business knowledge. The task models are directly derived from business processes. The UIs are generated using two other intermediate models: the dialog models and the presentation models with increasing levels of details. In addition, our proposed approach enables usability evaluation in the early stage of UI design instead of deferring the usability evaluation when the UI is fully developed. We use UI design guidelines and patterns to guide the transformations between models. As a result, the generated UIs have strong usability support such as consistent look and feel and transition guidance. Moreover, any changes to business processes can be automatically propagated to the UIs by regenerating the code skeletons using our prototype tool. It is easy for developers to integrate their feedback to the generation process and to fine-tune their design.

10.2 Future Work

In the future, we plan to enhance our proposed approach from the following aspects:

1) *extend our approach to generate designs and code skeletons from business process in other domains*. In our work, we apply our proposed approach to generate business applications from business processes in the business domain. To generalize our proposed approach to other domains, we plan to investigate the feasibility of our approach in generating applications from business process in other domains.
2) *investigate the effectiveness of our proposed approach in improving other quality attributes*. Our proposed approach is designed to generate business applications with desired quality attributes. We apply the three-tier software architectural style to improve the modifiability of software architecture and use UI design patterns to optimize the

usability of UIs. Business applications often need to satisfy a variety of quality requirements from different stakeholders. In the future, we plan to investigate the effectiveness of our proposed approach in improving other quality attributes, such as reusability, portability and security. In addition, we plan to investigate the effectiveness of different software architectural styles and design patterns in improving these quality attributes.

3) *invite end-users and software architects to evaluate the quality of generated UIs and software architecture designs*. In our work, we recruit graduate students to evaluate the usability of generated UIs and apply a scenario based approach to evaluate the quality of generated software architecture designs. In the future, we plan to invite end-users and software architects to evaluate the quality of the generated UIs and software architecture designs. The feedbacks from end-users and software architects can help refine the quality goal graphs and the selection of transformation rules. We will also improve the customizability of our prototype tool, so that developers can integrate their own design patterns, quality requirements, metrics, and UI widgets into the generation process.

Bibliography

[1]. A. Dix, J. Finlay, and GD. Abowd, *Human Computer Interaction*, Prentice Hall, 1998.

[2]. A. Guruge, *Corporate Portals Empowered with XML and Web Services*, Butterworth-Heinemann, 2002.

[3]. A. Tang, J. Han, and P. Chen, A Comparative Analysis of Architecture Frameworks, *In Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pp. 640-647, 2004.

[4]. A. van Deursen, and T. Kuipers, Identifying objects using cluster and concept analysis, *In Proceeding of the 21st International Conference on Software Engineering*, pp. 246–255, 1999.

[5]. A Practical Guide to Federal Enterprise Architecture (FEA), http://www.enterprise-architecture.info/Images/Documents/Federal%20Enterprise%20Architecture%20Guide%20v1a.pdf [2010].

[6]. B. Myers, Why are Human-computer Interfaces Difficult to Design and Implement. *Technical Report CMU-CS-93-183*, Carnegie Mellon University, 1993.

[7]. B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, 1997.

[8]. B. Nuseibeh, Weaving Together Requirements and Architectures, *Computer*, vol.34 (3): 115-117, 2001.

[9]. B. Shneiderman, Designing the User Interface. Addison-Wesley, 1998.

[10]. BA. Myers, Challenges of HCI Design and Implementation, *Interactions*, vol. 1(1): 73-83, 1994.

[11]. BA. Myers and MB. Rosson, Survey on user interface programming, *In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 195–202, 1992.

[12]. BD. Harper and KL. Norman, Improving User Satisfaction: The Questionnaire for User Satisfaction Interaction Version 5.5. *In Proceedings of the 1st Annual Mid-Atlantic Human Factors Conference*, pp. 224–228, 1993.

[13]. BE. John and DE. Kieras, The GOMS Family of User Interface Analysis Techniques: Comparison and Contrast, *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 3(4): 320-351, 1996.

[14]. BPMN, http://www.bpmn.org/. [2010]

[15]. BS. Mitchell, A Heuristic Search Approach to Solving the Software Clustering Problem, *PhD Thesis*, Drexel University, 2002.

[16]. Business Process Execution Language (BPEL) for Web Services version 1.1, http://www-128.ibm.com/developerworks/library/specification/ws-bpel/ [2010].

[17]. BV. Zanden and BA. Myers, Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces. *In Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*, pp. 27-34, 1990.

[18]. C. Abras, D. Maloney-Krichmar, and J. Preece, User-Centered Design. *In Bainbridge, W. Encyclopedia of Human-Computer Interaction*. Thousand Oaks: Sage Publications, 2004.

[19]. C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[20]. C. Hofmeister, P. Kruchten, RL. Nord, H. Obbink, A. Ran, and P. America, Generalizing a Model of Software Architecture Design from Five Industrial Approaches, *In Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pp. 77-88, 2005.

[21]. C. Kamm, D. Litman, and M. Walker, *From Novice to Expert: The Effect of Tutorials on User Expertise with Spoken Dialogue Systems*, In Proceeding of the 5th International Conference of Spoken Language Processing, pp. 1211-1214, 1998.

[22]. C. Mariage, J. Vanderdonckt, and C. Pribeanu, State of the Art of Web Usability Guidelines, *The Handbook of Human Factors in Web Design*, 2005.

[23]. C. Sant'Anna, E. Figueiredo, A. Garcia, and C. Lucena, On the Modularity of Software Architectures: A Concern-Driven Measurement Framework, *In Proceeding of the European Conference on Software Architecture (ECSA)*, 2007.

[24]. CH. Lung, Software Architecture Recovery and Restructuring through Clustering Techniques, *In Proceeding of the 3rd International Workshop on Software Architecture*, pp. 101-104, 1998.

[25]. CH. Lung, M. Zaman, and A. Nandi, Applications of Clustering Techniques to Software Partitioning, Recovery and Restructuring, *Journal of Systems and Software*, vol. 73(2), pp. 227-244, 2004.

[26]. CH. Lung, X. Xu, and M. Zaman, Software Architecture Decomposition Using Attributes. *International Journal of Software Engineering and Knowledge Engineering, Special Issue on Selected Papers from ICSEKE 2005*, 2005.

[27]. CM. Brown, Human-Computer Interface Design Guidelines, Intellect, 1998

[28]. D. Garlan and M. Shaw, An Introduction to Software Architecture, *In Advances in Software Engineering and Knowledge Engineering*, vol. 1, 1993.

[29]. D. McDavid, The Business-IT Gap: A Key Challenge, *IBM Research Memo*, http://www.almaden.ibm.com/coevolution/pdf/mcdavid.pdf [2010].

[30]. D. Montgomery and G. Runger, *Applied Statistics and Probability for Engineers*, 3rd ed. John Wiley & Sons, 2003.

[31]. D. Olsen, *User Interface Management Systems: Models and Algorithms*, Morgan Kaufmann Publishers Inc., 1992.

[32]. D. Sinnig, A. Gaffar, D. Reichart, P. Forbrig, and A. Seffah, Patterns in Model-Based Engineering, *In Proceedings of CADUI 2004*, pp. 197-210, 2004.

[33]. D. Stone, C. Jarrett, M. Woodroffe and S. Minocha. *User Interface Design and Evaluation*, Morgan Kaufmann, 2005.

[34]. DE. Perry, AA. Porter, and LG. Votta, Empirical Studies of Software Engineering: A Roadmap, *In Proceedings of the Conference on The Future of Software Engineering*, pp: 345-355, 2000.

[35]. DH. Hutchens and VR. Basili, System Structure Analysis Clustering with Data Bindings, *IEEE Transactions on Software Engineering*, vol. 11(8), pp. 749-757, 1985.

[36]. EF. Ecklund Jr., LML. Delcambre, and MJ. Freiling, Change Cases: Use Cases that Identify Future Requirements. *In Proceeding the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 342–358, 1996.

[37]. EH. Chi, P. Pirolli, and J. Pitkow, The Scent of a Site: A System for Analyzing and Predicting Information Scent, Usage, and Usability of a Web Site. *In Proceedings of the Conference on Human Factors in Computing Systems*, pp. 161–168, 2000.

[38]. F. Bachmann, L. Bass, and R. Nord, Modifiability Tactics, *Technical report CMU/SEI-2007-TR-002*. Software Engineering Institution, 2007.

[39]. F. Bodart, AM. Hennebert, JM. Leheureux, and J. Vanderdonckt, Towards a Dynamic Strategy for Computer-aided Visual Placement. *In Proceedings of the Workshop on Advanced Visual Interfaces*, pp. 78-87, 1994.

[40]. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture*: *A System of Patterns*, John Wiley & Sons Inc., 1996.

[41]. F. Paterno, Model-Based Design and Evaluation of Interactive Application, *Intelligence*, vol. 11 (4): 26-38, 2000.

[42]. FO. Buck, Indicators of Quality Inspections, *Technical Report 21.802*, IBM, 1981.

[43]. G. Booch, Object-oriented Design, *ACM SIGAda Ada Letters*, vol. 1(3): 64-76, 1982.

[44]. G. Brajnik, Automatic web usability evaluation: Where is the limit? *In Proceedings of the 6th Conference on Human Factors & the Web*, 2000.

[45]. H. Dhama, Quantitative Models of Cohesion and Coupling in Software, *Journal of Systems and Software*, vol. 29 (1): 65-74, 1995.

[46]. H. Traetteberg, Model-based User Interface Design, *Ph.D. thesis*, Norwegian University of Science and Technology, 2002.

[47]. IBM WBI Modeler, http://www-01.ibm.com/software/integration/wbimodeler/ [2010].

[48]. IBM WebSphere Commerce infocenter, http://publib.boulder.ibm.com/infocenter/wchelp/v6r0m0/index.jsp [2010].

[49]. I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering-A Use Case Driven Approach*. Addison-Wesley/ACM Press, 1992.

[50]. I. Sommerville, *Software Engineering*, 6th ed:Addison-Wesley Longman, 2000.

[51]. ISO/IEC 14598, http:// www.iso.org/iso/catalogue_detail.htm?csnumber=24902 [2010].

[52]. ISO/IEC 15414, Reference Model of Open Distributed Processing (RM-ODP), http://www.iso.org/iso/catalogue_detail.htm?csnumber=43767 [2010].

[53]. ISO/IEC 20926, IFPUG Functional Size Measurement Method, http://www.iso.org/iso/catalogue_detail.htm?csnumber=51717 [2010].

[54]. ISO 9126, http://www.iso.org/iso/iso_catalogue/catalogue_tc/ catalogue_detail. htm ?csnumber=22749 [2010].

[55]. ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs)- Part 11: Guidance on usability, http://www.iso.org/iso/catalogue_detail.htm?csnumber=16883 [2010].

[56]. J. Kirakowski and N. Claridge, Human Centered Measures of Success in Web Site Design. *In Proceedings of the 4th Conference on Human Factors & the Web* , 1998.

[57]. J. Koehler, R. Hauser, J. Küster, K. Ryndina, J. Vanhatalo, and M. Wahler, The Role of Visual Modeling and Model Transformations in Business driven Development, *In 5th International Workshop on Graph Transformation and Visual Modeling Techniques*, pp. 1-10, 2006.

[58]. J. Luftman and R. Kempaiah, An Update on Business-IT Alignment: "A Line" Has Been Drawn, *MIS Quarterly Executive*, vol. 6 (1): 165-177, 2007.

[59]. J. May and PJ. Barnard, Supportive Evaluation of Interface Design. *In Proceedings of the 1st Interdisciplinary Workshop on Cognitive Modeling and User Interface*, 1994.

[60]. J. Mylopoulos, L. Chung, and B. Nixon, Representing and Using Nonfunctional Requirements: a Process-oriented Approach, *IEEE Transactions on Software Engineering*, vol. 18(6): 488-497, 1992.

[61]. J. Nichols and A. Faulring, Automatic Interface Generation and Future User Interface Tools, *In Proceedings of the Workshop on the Future of User Interface Design Tools at CHI*, 2005.

[62]. J. Nielsen, *Designing Web Usability. The Art of the Simplicity*, New Riders Publishing, 2000.

[63]. J. Nielsen, *Guerrilla HCI: Using Discount Usability Engineering to Penetrate the Intimidation Barrier, Cost-Justifying Usability*, Academic Press, Inc., 1994.

[64]. J. Nielsen, *Usability Engineering*, Morgan Kaufmann, 1993.

[65]. J. Nielsen, *Novice vs. Expert Users*, Jakob Nielsen's Alertbox, 2000.

[66]. J. Nielsen and RL. Mack, *Usability Inspection Methods*, John Wiley & Sons, New York, NY, 1994.

[67]. J. Preece, Y. Rogers, H. Sharp, D. Beyon, S. Holland and T. Carey, *Human-Computer Interaction*, Addison Wesley, 1994.

[68]. J. Raskin, *The Human Interface: New Directions for Designing Interactive Systems*, ACM Press/Addison-Wesley Publishing Co., 2000.

[69]. J. Tidwell, *Designing User Interfaces*, http://designinginterfaces.com/ [2010]

[70]. J. Vanderdonckt, Knowledge-Based Systems for Automated User Interface Generation: the TRIDENT Experience. *Technical Report RP-95-010*, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, Namur, 1995.

[71]. J. Whiteside, J. Bennet and K. Holtzblatt, Usability Engineering Our Experience and Evolution, *Handbook of Human-Computer Interaction*, Elsevier Science, 1988.

[72]. J. Wu, AE. Hassan,RC. Holt, Comparison of clustering algorithms in the context of software evolution, IN Proceedings of the 21st IEEE International Conference on Software Maintenance 2005, pp. 525 - 535, 2005

[73]. JK. Blundell, ML. Hines, and J. Stach. The measurement of software design quality, *Annals of Software Engineering*, vol. 4(1): 235–255, 1997.

[74]. JM. Bieman, BK. Kang, *Measuring design-level cohesion*, IEEE Transaction on Software Engineering, vol. 24(2), pp. 111-124, 1998.

[75]. JM. Neighbors, Finding reusable software components in large systems, *In Proceedings of the 3rd Working Conference on Reverse Engineering*, pp. 2-10, 1996.

[76]. JS. Sottet, G. Calvary, and JM. Favre, Mapping Model: A First Step to Ensure Usability for sustaining User Interface Plasticity, *In Proceedings of the MoDELS 2006 Workshop on Model Driven Development of Advanced User Interfaces*, vol. 214, 2006.

[77]. JT. Nosek and P. Palvia, Software Maintenance Management: Changes in the Last Decade. *Journal of Software Maintenance: Research and Practice*, vol. 2 (3): 157–174, 1990.

[78]. K. Gajos and DS. Weld, SUPPLE: Automatically Generating User Interfaces, *In Proceedings of the 9th International Conference on Intelligent User Interfaces*, pp. 93-100, 2004.

[79]. K. Vredenburg, JY. Mao, PW. Smith, and T. Carey, A Survey of User-centered Design Practice, *In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Changing Our World, Changing Ourselves*, pp. 471-478, 2002.

[80]. KC. Hamborg, B. Vehse, and HB. Bludau, Questionnaire Based Usability Evaluation of Hospital Information Systems, *Electronic Journal of Information Systems Evaluation*, vol. 7 (1), pp. 21-30, 2004.

[81]. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, *2nd edition*, Addison-Wesley, 2003.

[82]. L. Chung, D. Gross, and E. Yu, Architectural Design to Meet Stakeholder Requirements. *In Proceeding of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pp. 545–564, 1999.

[83]. L. Dobrica and E. Niemelä, A Survey on Software Architecture Analysis Methods, *IEEE Transactions on Software Engineering*, vol. 28 (7), 2002.

[84]. L. Liu and E. Yu, From Requirements to Architectural Design-Using Goals and Scenarios, *Software Requirements to Architectures Workshop*, 2001.

[85]. L. Tahvildari, K. Kontogiannis, and J. Mylopoulos, Quality-driven software reengineering. *Journal of Systems and Software*, vol. 66(3):225–239, 2003.

[86]. LJ. Bass, LJ. Bass, and J. Coutaz, *Developing Software for the User Interface*, Addison-Wesley Longman Publishing Co., Inc., 1991.

[87]. LL. Constantine and LAD. Lockwood, *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*, ACM Press/Addison-Wesley Publishing Co., 1999.

[88]. M. Elkoutbi and RK. Keller, User Interface Prototyping Based on UML Scenarios and High-Level Petri Nets, *In Proceedings of the 21st International Conference on Application and Theory of Petri Nets*, pp. 166-186, 2000.

[89]. M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley Longman Publishing Co., Inc., 1996.

[90]. M. Padilla, *Strike a Balance: User's Expertise on Interface Design*, IBM white paper, http://www-128.ibm.com/developer works/web/library/wa-ui/#2, 2003.

[91]. M. Ramsay and J. Nielsen, *WAP Usability Report*, Nielsen Norman Group, 2000.

[92]. M. van Welie, *Patterns in Interaction Design*, http://www.welie.com/ [2010].

[93]. M. van Welie, GC. Van Der Veer, and A. Eliëns, Breaking down Usability, *In Proceedings of INTERACT 99*, pp. 613-620, 1999.

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

[95]. MA. Jackson, *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*, ACM Press. Addison-Wesley, 1995.

[96]. MA. Ould, *Business Processes: Modeling and Analysis for Re-engineering and Improvement*, John Wiley and Sons, 1995.

[97]. MF. Bertoa, JM. Troya, and A. Vallecillo, Measuring the Usability of Software Components, *Journal of Systems and Software*, vol. 79(3), pp. 427-439, 2006.

[98]. MF. Costabile, Usability in the Software Life Cycle, *Handbook of Software Engineering and Knowledge Engineering*, pp. 179-192, 2000.

[99]. MH. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and HF. Lipson, Attribute-Based Architecture Styles, *In Proceedings of the TC2 First Working IFIP Conference on Software Architecture*, p.225-244, 1999.

[100]. MJ. Tauber, ETAG: Extended Task Action Grammar-A Language for the Description of the User's Task Language. *In Proceedings of the IFIP TC13 Third International Conference on Human-Computer Interaction*, pp. 163-168, 1990.

[101]. MY. Ivory and MA. Hearst, The State of the Art in Automating Usability Evaluation of User Interfaces, *ACM Computing Surveys (CSUR)*, vol. 33 (4): 470-516, 2001.

[102]. N. Anquetil and TC. Lethbridge, Recovering Software Architecture from the Names of Source Files, *Journal of Software Maintenance: Research and Practice*, vol. 11 (3): 201-221, 1999.

[103]. N. Anquetil and TC. Lethbridge, Experiments with Clustering as a Software Re-Modularization Method, *In the 6th Working Conference on Reverse Engineering (WCRE'99)*, pp. 235–255, 1999.

[104]. NE. Fenton and A. Melton, Deriving Structurally Based Software Metrics, *Journal of Systems and Software*, vol. 12 (3): 177-187, 1990.

[105]. Netbeans, http://netbeans.org/ [2010].

[106]. Object Constraint Language (OCL), http://www.omg.org/spec/OCL/2.2/PDF [2010].

[107]. O. Maqbool and HA. Babri, The Weighted Combined Algorithm: a Linkage Algorithm for Software Clustering, *In Conference on Software Maintenance and Re-engineering (CSMR'04)*, pp. 15–24, 2004.

[108]. O. Maqbool and HA. Babri, Hierarchical Clustering for Software Architecture Recovery, *IEEE Transactions on Software Engineering*, vol. 33(11), pp. 759-780, 2007.

[109]. O. Zimmermann, J. Koehler, and F. Leymann, Architectural Decision Models as Micro-Methodology for Service-Oriented Analysis and Design, *In Workshop on Software Engineering Methods for Service Oriented Architecture*, 2007.

[110]. Opentaps, http://www.opentaps.org/ [2010].

[111]. P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level Modifiability Aanalysis (ALMA), *Journal of Systems and Software*, vol. 69 (1-2): 129–147, 2004.

[112]. P. Grünbacher, A. Egyed, and N. Medvidovic, Reconciling Software Requirements and Architectures with Intermediate Models, *Software and System Modeling*, vol. 3(3):235--253, 2004.

[113]. P. Kruchten, Architectural Blueprints-The 4+1 View Model of Software Architecture, *IEEE Software*, vol.12 (6): 42-50, 1995.

[114]. P. Jackson, J. Favier and I. Stagia, Segmenting Europe's Mobile Consumers, *Forrester Research Technographics Report*, 2002.

[115]. Q. Limbourg, C. Pribeanu, and J. Vanderdonckt, Towards Uniformed Task Models in a Model-Based Approach, *In Interactive Systems: Design, Specification, and Verification, Proceedings of the 8th International Workshop*, pp.164-182, 2001.

[116]. R. Briggs, and P. Gruenbacher, EasyWinWin: Managing Complexity in Requirements Negotiation, *In Proceeding of the 35th Annual Hawaii International Conference on System Sciences*, vol.1, 2002.

[117]. R. Kazman, G. Abowd, L. Bass, and P. Clements, Scenario-Based Analysis of Software Architecture, *IEEE Software*, vol. 13(6): 47-55, 1996.

[118]. R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, The Architecture Tradeoff Analysis Method, *In Proceeding 4th IEEE International Conference on Engineering Complex Computer Systems*, pp. 68-78, 1998.

[119]. RS. Aguilar-Saven, Business Process Modeling: Review and Framework, *International Journal of Production Economics*, vol. 90(2): 129-149, 2004.

[120]. RS. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill Higher Education, 5th edition, 2001.

[121]. RW. Schwanke, An Intelligent Tool For Re-engineering Software Modularity, *In Proceedings of the 13th International Conference on Software Engineering*, pp. 83-92, 1991.

[122]. S. Abrahão and E. Insfran. Early Usability Evaluation in Model Driven Architecture Environments, *In Proceedings of the 6th International Conference on Quality Software*, pp.287-294, 2006.

[123]. S. Ahmed and G. Ashraf, Model-based User Interface Engineering With Design Patterns, *Journal of Systems and Software*, pp. 1408–1422, 2007.

[124]. S. Ambler, User Interface Development Throughout the System Development Lifecycle, *Human Computer Interaction: Issues and Challenges*, pp. 11-28, 2001.

[125]. S. Inaganti, GK. Behara, Service identification: BPM and SOA Handshake, *Technical Report. Business Process Trends*, 2007.

[126]. S. Mancoridis, BS. Mitchell, C. Rorres, and Y. Chen, Using Automatic Clustering to Produce High Level System Organizations of Source Code, *In Proceeding of the 6th International Workshop on Program Comprehension*, pp. 45, 1998.

[127]. S. Mancoridis, BS. Mitchell, Y. Chen, and ER. Gansner, Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures, *In Proceeding of the IEEE International Conference on Software Maintenance (ICSM'99)*, pp. 50, 1999.

[128]. T. Mitra, Business-driven Development, *IBM developerWorks Article*, 2005.

[129]. T. Wiggerts. *Using clustering algorithms in legacy systems remodularization*. In Proceeding of Working Conference on Reverse Engineering, 1997.

[130]. The DoDAF Architecture Framework, http://cio-nii.defense.gov/sites/dodaf20/ [2010].

[131]. The NetRaker suite, http://www.netraker.com/nrinfo/products/nrer.asp [2010].

[132]. The Open Group Architecture Framework (TOGAF), http://www.opengroup.org/togaf/ [2010].

[133]. V. Tzerpos, and RC. Holt, ACDC: An Algorithm for Comprehension-Driven Clustering, *In Proceeding of the 7th Working Conference on Reverse Engineering (WCRE'00)*, pp. 258, 2000.

[134]. V. Tzerpos and RC. Holt, On the Stability of Software Clustering Algorithms, *In Proceedings of the 8th International Workshop on Program Comprehension*, pp. 211–218, 2000.

[135]. Visual Studio, http://www.microsoft.com/visualstudio/en-us [2010].

[136]. W. Li and S. Henry, Object-oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, vol. 23(2): 111–122, 1993.

[137]. W. Stevens, G. Myers, and L. Constantine, Structured Design, *IBM Systems Journal*, vol. 13 (2): 115-139, 1974.

[138]. WC. Kim and JD. Foley, Providing High Level Control and Expert Assistance in the User Interface Presentation Design. *In Proceedings of the Conference on Human Factors in Computing Systems*, pp. 430–437, 1993.

[139]. WO. Galitz, The essential guide to user interface design: an introduction to GUI design principles and techniques, 3rd ed., Wiley Pub., 2007

[140]. Y. Zou, Techniques and Methodologies for the Migration of Legacy Systems to Object Oriented Platforms, *Ph.D. thesis*, University of Waterloo, 2003.

[141]. Y. Zou, J. Guo, KC. Foo, and M. Hung, Recovering Business Processes from Business Applications, *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21 (5), pp. 315-348, 2009

[142]. Y. Zou, K. Kontogiannis, Migration to Object Oriented Platforms: A State Transformation Approach, *In the Proceeding of IEEE International Conference on Software Maintenance (ICSM)*, pp.530-539, 2002.

[143]. Z. Wen, and V. Tzerpos, An Effective Measure for Software Clustering Algorithms, *In Proceeding of the IEEE International Workshop on Program Comprehension*, pp. 194-203, 2004.

[144]. P Jaccard. Etude comparative de la distribution orale dans une portion des Alpes et des Jura. In *Bulletin del la Socit Vaudoise des Sciences Naturelles*, volume 37, pages 547-579.

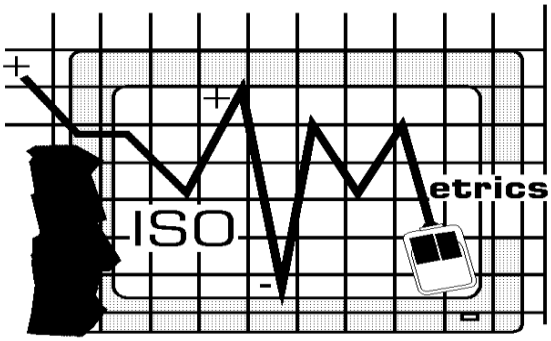
[145]. Y. Zou,Q. Zhang, and X. Zhao, Improving the Usability of e-Commerce Applications Using Business Processes, IEEE Transactions on Software Engineering, vol. 33(12), pp. 837 - 855, 2007

[146]. RA. Virzi, Refining the Test Phase of Usability Evaluation: How Many Subjects Is Enough? Human Factors: The Journal of the Human Factors and Ergonomics Society, vol. 34 (4), pp. 457-468(12), 1992

[147]. NE. Jacobsen, M. Hertzum, and BE. John, The Evaluator Effect in Usability Tests, Conference summary on Human Factors in Computing Systems, CHI 1998, ACM Press.

Appendix A

IsoMetric^{s} questionnaire



IsoMetrics^{S}

Questionnaire
for the evaluation of graphical user interfaces
based on ISO 9241/10

(short version)

Version: 2.01e October 1998

Contact:
Universität Osnabrück
Fachbereich Psychologie
Seminarstr. 20
D - 49069 Osnabrück
Germany

email: isometric@luce.psycho.uni-osnabrueck.de

About the questionnaire 'IsoMetrics'

Dear study participant:

The purpose of this questionnaire is to assess the usability of software products. By completing it, you are enabling us to identify and remedy any shortcomings, with the aim of enhancing its user-friendliness.

The questionnaire contains statements about the user-friendliness of software products. Please indicate the extent to which you agree or disagree with each of these statements, making use of the scale provided in each case. Here is an example:

		Pre-dominantly disagree		So - so		Pre-dominantly agree	
Index	**dialogue principle**	**1**	**2**	**3**	**4**	**5**	No opinion
0	Computers are useful work aids.					X	

If you think the statement "Computers are useful work aids" is true, then mark column "5" for "**Predominantly agree**" (as shown) with an X. If you find you cannot agree with this statement, then mark column "1" for "**Predominantly disagree**". You can also indicate various degrees of agreement between these two poles by marking the corresponding numbers (column 2 or 4). If for some reason you cannot or do not wish to reply, you should mark the last column "**no opinion**" with an X.

Thank you very much for your cooperation.

IsoMetrics^S

		Pre-dominantly disagree		So - so		Pre-dominantly agree	
Index	**suitability for the task**	**1**	**2**	**3**	**4**	**5**	No opinion
A.1	The software forces me to perform tasks that are not related to my actual work.						
A.3	The software lets me completely perform entire work routines.						
A.4	The functions implemented in the software support me in performing my work.						
A.6	The way in which data is entered is suited to the tasks I want to perform with the software.						
A.7	I perceive the arrangement of the fields on-screen as sensible for the work I do with the software.						
A.8	Too many different steps need to be performed to deal with a given task.						
A.9	The way in which data is output is suited to the tasks I want to perform with the software.						
A.10	The software is well suited to the requirements of my work.						
A.11	In a given screen, I find all of the information I need in that situation.						
A.12	The terminology used in the software reflects that of my work environment.						
A.14	The software provides me with a repeat function for work steps that must be performed several times in succession.						
A.15	I can easily adapt the software for performing new tasks.						
A.16	The important commands required to perform my work are easy to find						
A.17	I am able to adjust the presentation of results (on the screen, to printer, plotter etc.) to my various work requirements.						
A.18	The presentation of the information on the screen supports me in performing my work.						

IsoMetrics^S

Index	self-descriptiveness	Pre-dominantly disagree 1	2	So - so 3	4	Pre-dominantly agree 5	No opinion
S.2	I can call up specific explanations for the use of the system, if necessary.						
S.3	I understand immediately what is meant by the messages displayed by the software						
S.5	It is easy to retrieve information about a certain entry field.						
S.6	When menu items are not available in certain situations, this fact is visually communicated to me.						
S.7	If I want, the software will display not only general explanations but also concrete examples to illustrate points.						
S.8	The explanations the software gives me clearly refer to the specific situations in which they are output.						
S.9	If I want, the software displays basic information about conceptual aspects of the program.						
S.10	The software provides me with enough information about which entries are permitted in a particular situation.						
S.11	I can tell straight away which functions are invoked by the various menu items.						
S.12	The terms and concepts used in the software are clear and unambiguous.						
S.13	The software always visually marks the current entry location (e.g. by a highlight, a contrasting color, a blinking cursor, etc.).						
S.14	I can easily tell the difference among feedback messages, requests to confirm inputs or commands, warnings, and error messages.						

IsoMetrics^S

Index	controllability	Pre-dominantly disagree 1	2	So - so 3	4	Pre-dominantly agree 5	No opinion
T.2	The possibilities for navigating within the software are adequate.						
T.3	The software makes it easy for me to switch between different menu levels.						
T.4	The software lets me return directly to the main menu from any screen.						
T.5	I can interrupt any dialog at any time.						
T.6	It is always easy for me to evoke those system procedures that are necessary for my actual work.						
T.7	It's easy for me to move back and forth between different screens.						
T.8	The software allows me to interrupt functions at any point, even if it is waiting for me to make an entry.						
T.10	The navigation facilities of the software support optimal usage of the system functionality.						
T.12	In order to perform my tasks, the software requires me to perform a fixed sequence of steps.						
T.13	When selecting menu items, I can speed things up by directly entering a letter or a command code.						
T.15	It is always possible to abort a running procedure manually.						

IsoMetrics^S

Index	conformity with user expectations	Pre-dominantly disagree **1**	**2**	So - so **3**	**4**	Pre-dominantly agree **5**	No opinion
E.8	The software is inconsistently designed, thus making it more difficult for me to do my work.						
E.1	I can anticipate which screen will appear next in a processing sequence.						
E.2	I have no difficulty in predicting how long the software will need to perform a given task.						
E.3	The designations are used consistently in all parts of the software I am familiar with.						
E.4	I find that the same function keys are used throughout the program for the same functions.						
E.5	When executing functions, I have the feeling that the results are predictable.						
E.6	My impression is that the same possibilities are consistently available for moving within and between different parts of the software.						
E.7	The messages output by the software always appear in the same screen location.						

IsoMetrics^S

Index	error tolerance	Pre-dominantly disagree 1	2	So - so 3	4	Pre-dominantly agree 5	No opinion
F.1	When working with the software, even small mistakes have sometimes had serious consequences.						
F.2	Even if I make a mistake, the information (e.g. data, text, and graphics) which I have just entered is not lost.						
F.3	If I make a mistake while completing a form, I can easily restore everything to its previous state.						
F.4	When I attempt to perform a destructive operation (e.g. deletion of data etc.), I am always first prompted to confirm the action.						
F.5	My impression is that very little effort is involved in correcting mistakes.						
F.6	When I make entries, they are first checked for correctness before further processing is initiated.						
F.7	No system errors (e.g. crashes) occur when I work with the software.						
F.8	If I make a mistake while performing a task, I can easily undo the last operation.						
F.9	I have never made an entry that caused a software error (e.g. a system/program crash or an undefined dialog state).						
F.10	The software includes safety features to help prevent unintended actions (e.g. critical keys spaced well apart, highlights, designations that are not easily confused).						
F.12	The software provides me with useful information on how to recover from error situations.						
F.13	I perceive the error messages as helpful.						
F.14	In some situations the software waits too long before calling attention to wrong entries.						
F.15	The software warns me about potential problem situations.						
F.16	The software lets me keep the original data even after it has been changed.						

IsoMetrics^S

Index	suitability for individualization	Pre-dominantly disagree 1	2	So - so 3	4	Pre-dominantly agree 5	No opinion
I 1	The software lets me adapt forms, screens and menues to suit my individual preferences.						
I 4	The software can be easily adapted to suit my own level of knowledge and skill.						
I 6	I am able to adjust the amount of information (data, text, graphics, etc.) displayed on-screen to my needs.						
I 7	The software lets me change the names of commands, objects and actions to suit my personal vocabulary.						
I 8	I can adjust the attributes (e.g. speed) of the input devices (e.g. mouse, keyboard) to suit my individual needs.						
I 11	I can adjust the software's response times to my own personal working speed.						

Index	suitability for learning	Pre-dominantly disagree 1	2	So - so 3	4	Pre-dominantly agree 5	No opinion
L 1	I needed a long time to learn how to use the software.						
L 2	It is easy for me to relearn how to use the software after a lengthy interruption.						
L 3	The explanations provided help me understand the software so that I become more and more skilled at using it.						
L 4	So far I have not had any problems in learning the rules for communicating with the software, i.e. data entry.						
L 5	I was able to use the software right from the beginning by myself, without having to ask coworkers for help.						
L 6	I feel encouraged by the software to try out new system functions by trial and error.						
L 7	In order to use the software properly, I must remember a great many details.						
L 8	I find it easy to use the commands.						