ANALYSIS OF THE EFFECTIVENESS OF COMPLEXITY MEASURES IN COMPONENT-BASED INTEGRATION TESTING

BY

Fahmi Hassan Ali Qurada'a

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Science

JUNE, 2011

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS DHAHRAN,
SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **FAHMI HASSAN ALI QURADA'A** under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.

Thesis Committee

Thesis advisor Sajjad Mahmood
Dr. Sajjad Mahmood

Member
Prof. Sabri Mahmoud Mahmoud

Member
Dr. Mahmoud Elish MElish

عادل
Department Chairman
Dr. Adel Fadhl Ahmed

Dean of Graduate Studies
Dr. Salam A. Zummo

12/6/11
Date

DEDICATION

I dedicate this dissertation with all of my love to my parents, my wife, my daughter, my brothers and sisters.

ACKNOWLEDGMENT

I thank Allah (SWT) for granting me health, patient, guidance and determination to successfully accomplish this work.

I would like to express my deep appreciation to my thesis advisor Dr. Sajjad Mahmood, for his constant help, guidance, encouragement and invaluable support. Thanks are due to my thesis committee members Prof. Sabri A. Mahmoud and Dr. Mahmoud O. Elish for their cooperation, comments and support.

I would like to thank King Fahd University of Petroleum and Minerals for sponsoring me throughout my graduate studies. I also would like to thank Aden Community College which gives me the opportunity for completing my MSc degree in KFUPM.

Also, I would like to thank all my friends, mostly those who helped me in this thesis. A special thanks go to all students who participated in the experiments.

Finally, I would you like to thank my parents, my wife, daughter, brothers, and sisters who always support me with their love, patience encouragement and constant prayers throughout my study.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Thesis Abstract

NAME: Fahmi Hassan Ali Qurada'a

TITLE: Analyze the Effectiveness of Complexity Measures in Component-Based Integration Testing

MAJOR FIELD: Computer Science.

DATE OF DEGREE: June, 2011.

Component Based System (CBS) development is increasingly becoming popular for software development. Using components in developing software systems can have a potential benefits such as decrease development cost, increase software productivity, reliability, as well as improve the quality of the final products. But can also involve a series of limitations. However, black box nature of components introduces unique challenge at the integration phase of a CBS. It is well-known that there is a correlation between the number of faults found in software and its complexity. Components complexity measures have been used to identify possible faults in individual components and subsequently perform a risk assessment of the system. The aim of this work is to empirically investigate the usefulness of structure complexity measures to improve Component-Base (CB) integration testing (glue-code) in terms of defect detection effectiveness and effort. We ran three controlled experiments with students from King Fahd University of Petroleum and Minerals, to evaluate the effectiveness of structure complexity measures in CB integration testing (glue-code testing). Experiments results indicate that the adoption of structure complexity measures led to a significant better detecting of the faults during CB integration testing without requiring a significant additional effort. Finally, subject experience doesn't have any effect on the defect detection effectiveness.

خلاصة الرسالة

الاســـم : فهمي حسن علي قراضة

عنوان الرسالة : تحليل فعالية مقايس تعقيد النظام Structure Complexity Measures **خلال مرحلة الاختبارات التكاملية لنظم البرمجية المبنية علي مكونات برمجية.**

التخصص : علوم حاســب الي

تاريخ التخرج : يونيو 2011

لاقة الانظمة البرمجية المبنية علي مكونات برمجية (Component-Based systems) في الاونة الاخيرة اهمية وشعبية متزايدة في تطوير البرمجيات. فاستخدام هذة المكونات البرمجية (Components) في تطوير البرمجيات معقدة لها مزايا متعددة فمثلا تخفيض تكلفة تطوير البرمجيات و زيادة انتاجية البرمجيات وكذلك زيادة الوثوقية في البرمجيات فضلا عن تحسين جودة المنتجات النهائية. فالبرغم من هذة المزايا فانها تعاني العديد من النواقص. فالطبيعة السوداء (Black-Box nature) لهذة المكونات البرمجية تجعل عملية اجراء الاختبارات اللازمة لهذة المكونات لدمجها مع بعضها البعض اكثر تعقيدا. فمن المعلوم ان هناك علاقة مابين عدد الاخطاء الموجودة في النظام ودرجة تعقيد هذا النظام. بالاضافة لهذا فمقايس درجة تعقيد النظام (Complexity Measures) تم استخدامها لتحديد الاخطاء في المكونات البرمجية علي حدة ومن ثم في الكود ارابط مابين هذة المكونات(Glue-code) ومن ثم بناء نظام لتقيم المخاطر التي قد ترافق بناء النظام. تهدف هذة الاطروحة الي اجراء دراسة تجربية لمعرفة فؤائد استخدام مقايس التعقيد Structure Complexity Measures للمكونات البرمجية في تحسين عملية اختبار دمج هذة المكونات البرمجية (Component-Based Integration Testing) من حيث فعاليتها في اكتشاف الاخطاء التي قد ترافق عملية التفاعل فيما بين هذة المكونات البرمجية و كذلك الجهد المبذول في عملية الاختبار. خلال هذة الدراسة تم اجراء ثلاث تجارب بمشاركة طلاب من جامعة الملك فهد للبترول والمعادن لتقيم تاثيرات مقايس التعقيد Structure Complexity Measures في عملية الاختبارات التي تلازم عملية دمج المكونات. تشير النتائج التي تم الحصول عليها من خلال هذة الدراسة الي ان استخدام مقايس التعقيد Structure Complexity Measures خلال عملية الاختبارات ادت الي زيادة في عدد الاخطاء التي تم اكتشافها. ولكننا لم نلاحظ أي فارق في الجهد المبذول لاجراء الاختبارات لهذة المكونات. بالاضافة الي ذلك لم نلاحظ أي تاثير لخبرة المشاركين علي عدد الاخطاء التي تم اكتشافها.

CHAPTER ONE

INTRODUCTION

1.1 Overview

The concept of developing software components and the reuse of them gained widespread popularity and has been referred to as the next big phenomenon for software engineering. Nowadays, Component-Based System (CBS) are developed by assembling prefabricated components [1][2]. Some components may be developed in-house, while others are commercial off-the-shelf components (COTS); whose source code is usually unavailable to system integrator. Generally, CBS is an attractive approach because it has a potential to decrease development cost and increase software productivity and reliability, as well as improve the quality of the final products. All of these advantages can be achieved only when the cost of reusing and integrating these components in the new environment is lower than the cost of building system from scratch [3].

Quality of a CBS depends on its components quality, and any faulty component can lead to serious consequences on all software built on it. Hence, component validation and quality control is crucial to component providers and consumers [3].

CBS is an integration centric approach and integration testing is fundamental to its success. According to Li and Wahl [4], approximately 40% of software errors are discovered through integration testing. The main focus of integration testing is validating interactions between components of a CBS. One type of integration testing is a "big bang" approach where we test all possible interaction in a CBS. Another approach is incremental integration that uses either a top-down or bottom-up approaches. Both approaches need stubs and drivers' modules for the integration testing of a CBS [5].

In literature, interaction-related faults are categorized into three types namely: inter-component faults, interoperability faults, and traditional faults. Inter-component faults are programming-related faults. Even when component provider evaluates each component's functionality individually, there are still faults in the interaction among components. The black-box nature of a component, reusability, and heterogeneity characteristics of a CBS lead to different types of interoperability faults. These faults can be categorized into programming-language level, system-level, and specification interoperability faults. Traditional faults are related to special-execution environments or special-input [6].

1.2 Thesis Motivation

Test case prioritization techniques increase the fault detection effectiveness of testing by ordering test cases. Many prioritization techniques have been proposed and evidences show that they can be beneficial [7] [8] [9] [10].

To measure component complexity, reusability, and customizability of component; Cho et al. [11] have introduced a suite of metrics. This suite includes four metrics to measure component complexity which is component cyclomatic complexity, component plain complexity, component dynamic complexity, and component static complexity. These metrics require the complexity analysis of each method and class. The metrics also need the analysis of component's source code, to extract component cyclomatic complexity, which is rarely available. Narasimhan and Hendradjaya [12] have extracted a suite of metrics from the component interface definition language specification to measure component complexity and criticality by deriving component packing density (CPD) and component interaction density (CID) metrics. Mahmood and Richard [13] have proposed a structure complexity measures for a CBSS depending on its components by considering the interaction properties, syntactic, and semantic. They have defined three elements which are interface, constraints, and interaction, as main contribution to the CBSS complexity.

There is a correlation between the number of faults found in software and its complexity measures [14], [15], [21]. For example, complexity metrics have been used to identify fault-proneness in traditional object oriented systems [16] [17] [18] [19] [20]. Goseva et al. in [21] have used cyclomatic complexity to identify high-risk components and their connectors to improve the quality of the product. The complexity measures have been used to identify possible faults in individual components and subsequently perform a risk assessment of the system.

1.3 Aims of the work

It is well-known that there is correlation between the number of faults found in a software component and its complexity [21]. In 1976, McCabe has introduced cyclomatic complexity as a measure of a program complexity [22]. *Goseva et al.* in [21] have built a risk analysis model using the UML specifications to identify the high-risk components. In this model, they have measured a component complexity similar to McCabe's cyclomatic complexity. But, instead of using the source code control flow graph they have used the UML state charts.

Since, the CBS complexity measures enable a system analyst to identify fault-prone components and associated integration [14], [15] , [21]; it can be used as a guideline to priorities integration testing of a CBS in order to improve the performance of integration testing of CBS in terms of a number of defect detection and effort.

To the best of our knowledge, there is no work done to investigate the effectiveness of structure complexity measures in integration testing (i.e. glue-code testing) of a CBS.

In this research, we are going to investigate whether structure complexity measures (i.e. glue-code cyclomatic complexity and interface complexity) can be used as a guide to priorities integration testing of a CBS. We aim to conduct an empirical study to analysis and validate the effectiveness of these complexity measures in the CBS integration testing.

1.4 Thesis Contributions

The following subsections briefly list and summarize the contribution of this work

1.4.1 Development Of Three CBS Systems

Three distinct systems are developed using the Component-based Software Development (CBSD) process [23]. A Hotel Reservation System (HRS) is adopted from [23], while Library Management System (LMS) is adopted from [24]. Finally, Smart Office System (SOS) is adapted from [25].

1.4.2 Experiment Design And Setting

A control experiment are conducted in an academic environment (King Fahd University of Petroleum and minerals) to determine whether structure complexity measures are good indictors for priorities integration testing of a CBS.

1.4.3 Results Analysis

We use a set of well-established statistical techniques to analysis and discuss the results of the experiments.

1.5 Thesis organization

The structure of the thesis is outlined in the following previews of each of the remaining chapters:

- Chapter 2: presents the literature review. We give an overview of the traditional testing techniques, the CBS integration testing techniques, and finally, the existing component complexity metrics.
- Chapter 3: provides a unique engineering process for developing component-based software and gives more details on the component-based testing phases. Control flow testing and structural coverage criteria are discussed and it explores the structure complexity measures, glue code cyclomatic complexity and component interface complexity. Finally, it explores parametric and nonparametric tests.
- Chapter 4: describes the experimental setting. We discuss our experiment aims, experiment environment, hypothesis, experiment subjects, and the experiment materials.
- Chapter 5: discusses the experiments results and analysis.
- Chapter 6: concludes this thesis and describes a number of limitations. Possible future directions are also provided.

CHAPTER TWO

LITERATURE REVIEW

2.1 Overview

Software testing is an important activity of a software development process, which is intended at assuring the quality of the product. At the present, component provider deliver components that only include specifications of the interfaces without the source code. The essential problem is the lack of information for analysis and testing of components. A number of component integration testing approaches have introduced a solution for these problems [2].

In this chapter, we describe the work done related to the CBS integration testing, traditional testing techniques, and the complexity metrics for CBS. To our best knowledge, there is no research has been performed earlier to investigate whether structure complexity measures for CBS can be used as a guide to priorities integration testing of a CBS. Furthermore, to our best knowledge, there is no empirical study have been conducted to analysis and validate the effectiveness of these structure complexity measures in the CBS integration testing

2.2 Traditional Testing Techniques

Testing techniques can be categorized into two general approaches, black box and white box [26]. Black box testing approaches create test data without using any knowledge of the structure of the software under test, whereas white box testing approaches explicitly use the program structure to develop test data. In this section we summarize some of these techniques namely, control flow testing, data flow testing, random testing, and finally Boundary value testing.

2.2.1 Control Flow Testing

Control-flow testing ensures that program statements and decisions are fully exercised by code execution. In this technique source code of program is converted to control flow graph then we select the proper test coverage criteria and generate satisfying test cases. Control flow testing coverage criteria include node coverage, edge coverage and pair – edge coverage. Node coverage criteria requires that every statement of the program be executed at least once, edge coverage criteria requires that every branch of the program be visited at least once, and pair –edge coverage criteria requires that all pair-path in the program must be executed at least once [26]. Table 2.1 includes a list of references that are related to control flow testing.

2.2.2 Data Flow Testing

Data flow testing is a testing technique in which test cases are designed based on the flow of data within the code and within the system(s). It looks at the lifecycle of a particular piece of data (i.e. a variable) in an application. Data flow occurs when variables are declared and then accessed and changed as the program progresses [26]. Table 2.1 includes a list of references that are related to data flow testing.

2.2.3 Random Testing

Random testing is a form of functional testing that is useful when the complexity of the problem makes it impossible to test every combination. In random testing, we don't test the application sequentially; we just take the modules randomly and carry out testing whether it's functioning correctly [26]. Table 2.1 includes a list of references that are related to random testing.

2.2.4 Boundary Value Testing

Boundary value testing is carried out by creating test sets that covers the boundary values of the output and input classes identified in the specification so that both lower, upper values and the values between them of an equivalence class are exercised by test cases. Table 2.1 includes a list of references that are related to boundary value testing.

Technique	References
Control flow testing	[26] [27] [28] [29] [30] [31] [32]
Data flow testing	[26] [33] [34] [35]
Random testing	[26] [36] [37] [38]
Boundary value testing	[26] [40] [41]

Table 2.1 A list of testing techniques references

2.3 CBS Integration testing

In literature different approaches have been proposed for CB integration testing.

Wang et al. [42] proposed a Built-In Testing (BIT) technique for developing maintainable CB software. In which built-in tests are equipped to the component's code such that the component user can choose whether to execute these tests or not. The component user can run the component in "test mode (maintenance)" or "normal mode". In the test mode, the built-in tests are executed during execution of the component whereas in the normal mode, these tests are not executed.

The disadvantage of Wang's approach is increasing component size as a result of adding test cases. To get rid of this issue, Hornstein and Elder in [43] introduced the Component+ BIT approach that divides test cases from the component. The component provider creates two types of components which are a BIT component and a test-component. The first type is a component that has built-in testing capabilities while the other type includes test cases interacts with BIT-component that has testing capabilities through its interfaces.

Beydeda and Gruhn in [44] presented a self-testing approach for COTS components. They recommend appending the test component with analysis functionality and testing tools. Hence, the information that component integrator wants to produce test cases can be encapsulated in the component, or it can be produced on request.

Orso et al. in [45] recommend that every software artifacts which is used for developing component is considered metadata. These metadata should be delivered with the component to increase component testability. These metadata are also useful in conducting coverage analysis in component-based integration testing.

Wu et al. [46] suggest shipping a component UML models as metadata. These models can be used to identify context-dependent relationships between the components that can be useful for component-based integration testing.

Belli and Budnik [47] introduced a similar technique where they append the component with UML state-charts model. They used model-based tools to extract test cases from the UML state-charts. In this technique component integrator can perform coverage-based execution of the model. But the component provider has to produce the model every time the component is modified.

Counsill [48] suggested that a component should be certified by a third-party. In this technique, an independent organization tests the quality of the component and provides the test results along with the test environment to the component user.

Ma et al. [49] proposed a framework for third-party certification that consists of following these steps:

i) The third-party provides guidelines to the component developer.

ii) The component developer produces a test package using these rules.

iii) The third-party executes the test package and produces a test report.

One advantage of this approach is that it is performed by a neutral organization, and hence the results are not biased.

Gao et al. [50] introduced a testable beans technique to improve component testability. In this technique, the component developer implements testing interface (test interface) and generates test cases in the form of clients.

Jabeen and Rehman [51] proposed an approach to test object-oriented components where, the component integrator, the component supplier, and a third-party exchange test information via descriptors. These descriptors include the component requirements. A component descriptor is prepared by the component developer and fixes it to the component. The component analyst specifies the requirement of a component in another descriptor, the component requirement descriptor. The third-party creates test information via the information in the component descriptor and the component requirement descriptor.

Piel et al. [52] introduced the notion of virtual component in order to perform and manage integration testing of a CBS organized in a data flows. This approach was derived to the systems that present high availability requirements which make their runtime evaluation necessary. This means that integration and system testing will have to be performed at runtime as well. The basic idea is that every data flow to be tested corresponds to one virtual component. So the inputs of the virtual components correspond to the inputs of the data-flow and its outputs correspond to the outputs of the data-flow. Thus, integration testing of data-flow is equivalent to unit testing the virtual component.

2.4 Complexity Measures

To measure component complexity, reusability, and customizability of component; Cho *et al.* [11] introduced a suite of metrics. This suite includes four metrics to measure component complexity which is component cyclomatic complexity, component plain complexity, component dynamic complexity, and component static complexity. These metrics require the complexity analysis of each method and class. The metrics also need the analysis of component's source code, to extract component cyclomatic complexity, which is rarely available.

Narasimhan and Hendradjaya [12] extracted a suite of metrics from the component interface definition language specification to measure component complexity and criticality by deriving component packing density (CPD) and component interaction density (CID) metrics.

Mahmood and Richard [13] proposed a structure complexity measure for a CBSS depending on its components by considering the interaction properties, syntactic, and semantic. They have defined three elements which are interface, constraints, and interaction as main contribution to the CBSS complexity.

Research indicates [14], [15], [21] that there is a correlation between the number of faults found in a software component and its complexity measures. *Goseva et al.* in [21] have used cyclomatic complexity to identify high-risk components and their connectors to improve the quality of the product. The complexity measures are used to identify possible faults in individual components and subsequently perform a risk assessment of the system. However, to the best of our knowledge, none of the existing work investigates the potential benefits of complexity measures during glue code testing of a CBS.

CHAPTER THREE

BACKGROUND

3.1 Overview

Component-based software comprises a collection of self-contained and loosely coupled components that allow plug-and-play. The components may have been written in different programming languages, executed on different operational platforms, and distributed across geographic distances. Some components may be developed in-house, while others may be third-party or commercial-off-the-shelf (COTS) components, with the source code unavailable [23][39].

This chapter is organized as follows: Section 3.2 provides a unique engineering process for developing component-based software. Section 3.3 gives details on the component-based testing phases. Control flow testing and structural coverage criteria in Section 3.4. In Section 3.5 we explore the structure complexity measures, glue code cyclomatic complexity and component interface complexity. Finally, in section 3.6 we explore parametric and nonparametric tests.

3.2 Component-Based Development Process

All software development projects follow two distinct processes at the same time. The Management Process schedules work, plan deliveries, allocates resources, and monitor progress. The Development Process creates working software from requirements. Today the development process has to be subservient to the management process. This is because the management process controls project risk, and risk control is rightly viewed as paramount, even if the process is compromised as a result. The favoured management process nowadays is one based on evaluation, where the software is delivered over a number of development iterations, each refining and building on the one before. In this section we focus on the development process but we don't cover the details of the management processes. Figure 3.1 shows the overall development process. The boxes represent phases and the thin arrows represent the flow of deliverables that carry information between phases. Comparing the phases of Figure 3.1 to those found in the traditional development process, the requirements, test, and deployment phases correspond directly to those with the same names in the traditional development process. The specification, provisioning and assembly phases replace the analysis, development, and assembly phases [23][39]. For more details a complete case study for Hotel Reservation System is available in Appendix A.

3.2.1 Requirement

In this phase we give a high-level system description and describe the business process of the system. The business process description introduces a number of terms. The business concept model is built to link these terms and other key terms to create a common vocabulary among the people involved in the business. Finally, use cases are described in details in order to build the use cases model [23].

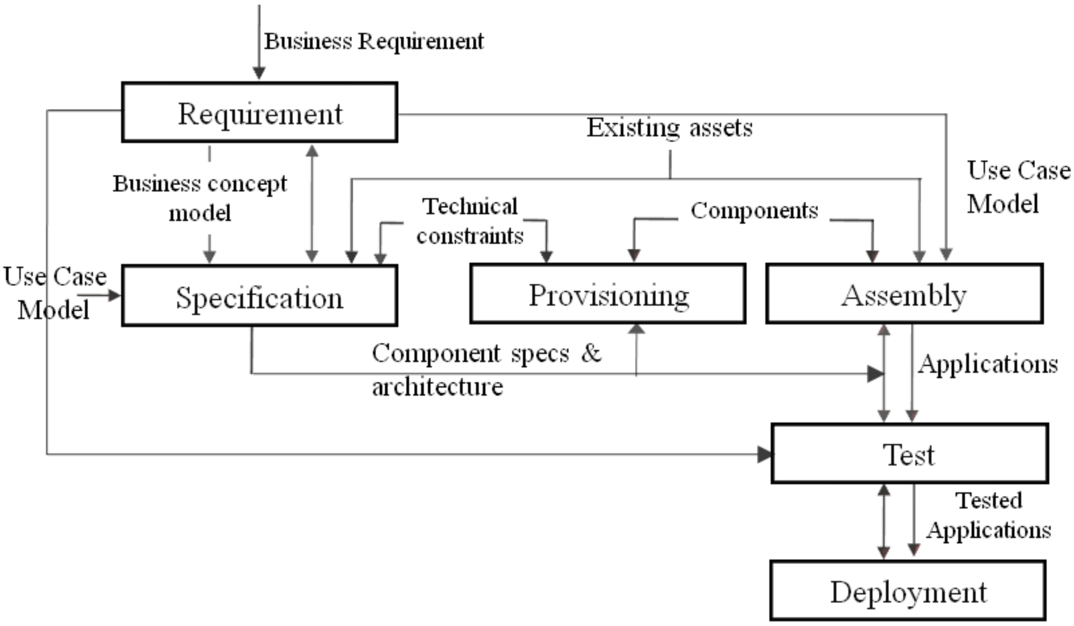


Figure 3.1 Component-based development processes

3.2.2 Specification

Component specification phase includes three stages namely, component identification, component interaction, and component specification. It takes as inputs the business concept model and the use case diagram from the requirement phase and the main outputs of this phase are components specifications, components architectures, and components interfaces [23]. Figure 3.2 shows the specification stages and the activities in each stage.

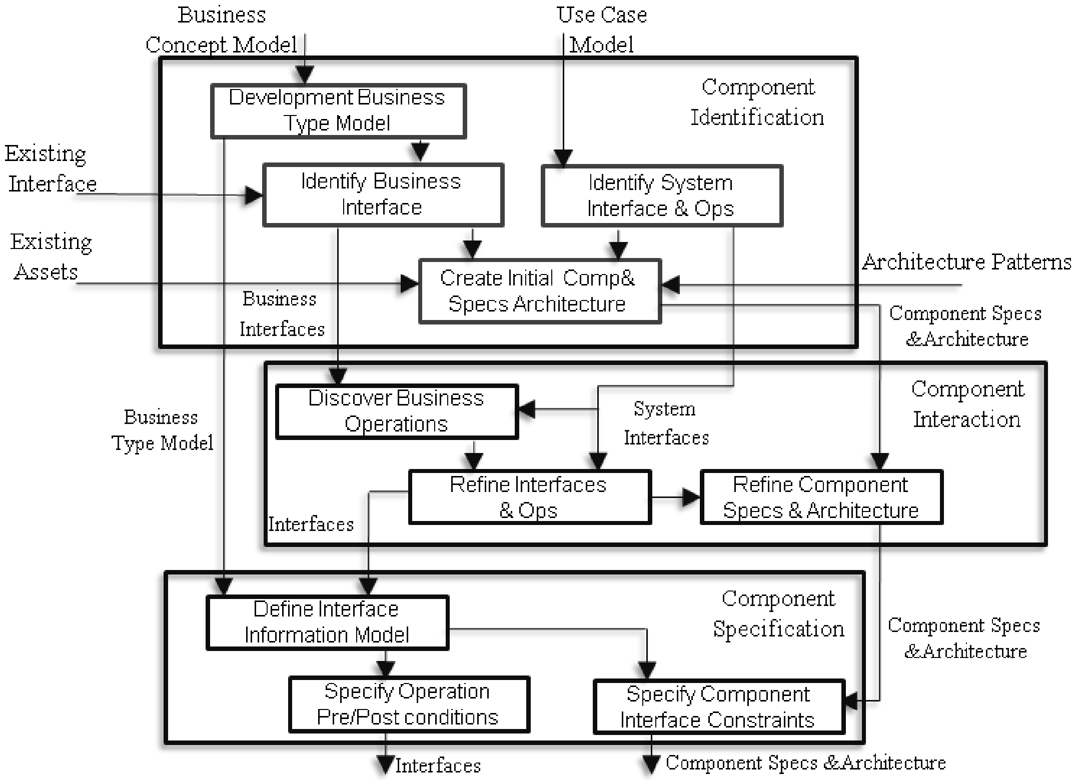

Business Concept Model
Use Case Model
Component Identification
Development Business Type Model
Existing Interface
Identify Business Interface
Identify System Interface & Ops
Existing Assets
Create Initial Comp& Specs Architecture
Architecture Patterns
Business Interfaces
Component Specs &Architecture
Component Interaction
Business Type Model
Discover Business Operations
System Interfaces
Refine Interfaces & Ops
Refine Component Specs & Architecture
Interfaces
Component Specification
Define Interface Information Model
Component Specs &Architecture
Specify Operation Pre/Post conditions
Specify Component Interface Constraints
Interfaces
Component Specs &Architecture

Figure 3.2 Specification stages

3.1.2.1 Component Identification

The purpose of this stage is to create an initial set of interfaces and component specifications, linked together into initial component architecture. It also generates an important internal specification, the business type model, which is used later to create interface information models [23].

3.1.2.2 Component Interaction

In component interaction, we decide how the components interact with each other to do the required functionality. The existing interface definitions are refined in order to discover new interfaces and operations. UML collaboration diagrams are used to model the components interactions. Component architecture and the system interfaces that are extracted in component identification stage are used to model the interaction among components. In this stage, we discover operations of the business interfaces by drawing one or more collaboration diagrams for each operation in the system interfaces. At the end of this stage, we end up with a list of business interfaces and system interfaces with its operations signatures [23].

3.1.2.3 Component Specification

In components specification we specify all interfaces supported by components or the interfaces that depends on. In this stage, we want to represent the state of the component object on which the interface depends. Since each interface has an interface information model, any changes to the state of the component object can be described in terms of this information model [23].

3.2.3 Provisioning Phase

The provisioning phase ensures that the necessary components are made available, either by building them from scratch, buying them from a third party, or reusing, integrating, mining, or otherwise modifying an existing component or other software. The provisioning phase also includes individual components testing prior to assembly [23].

3.2.4 Assembly Phase

In the Assembly phase we take all components and put them together with existing software assets and suitable graphical user interface to form an application that meets the business needs [23].

3.3 Component-Based Testing Phases

Testing is essential in the development of any software system. It is required to assess a system's functionality and quality of operation in its final environment. This is especially of importance for system being assembled from any self-contained software components. Basically testing is done to reveal faults and after detecting failures, debugging techniques are applied to isolate and remove faults. Conventionally, the development of complicated software constitutes three major testing phases: unit testing, integration testing, and system testing [3] [39]. In CBS development, these traditional testing phases must be adapted as shown in Figure 3.3.

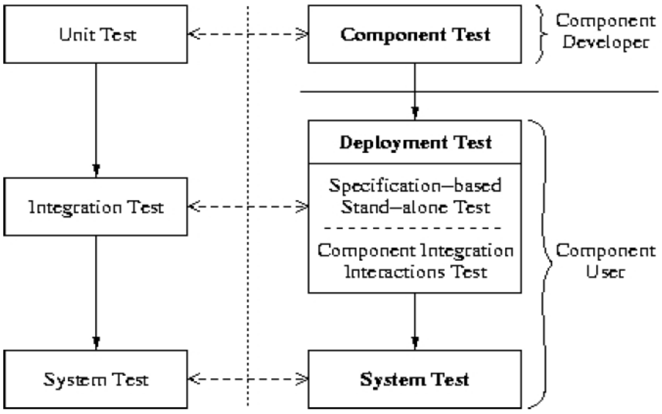


Figure 3.3 Component-Based testing process

3.3.1 Component Testing

In component testing, component developer performs testing for each component in isolation to check component functionality and uncover possible errors. Since source code of components is available for component developers, this allows them to do white-box testing for all components. In addition, they also conduct black-box testing to make sure that right specifications are attached with the component. However, component testing cannot address the component behavior when component is assembled in new environment [39].

3.3.2 Component Integration Testing

Integration testing is defined by IEEE as "testing in which software components are combined and tested to evaluate the interaction between them" [22]. Component, whether integrated individually or with other components, requires integration testing. Component integration testing is performed by a component consumer; its objective is to validate the implementation of the components that will make up the final software. The crucial problem is the lack of information for analysis of components. Integration testing has always been a challenge especially if the system under test is large with subsystems and interfaces. Several component integration-testing techniques have been proposed to provide a solution for these issues [1][39]. In this section, we give an overview to some of those techniques from the viewpoint of the component user.

S. Beydeda at el [1] have categorized CB integration testing based on the information that is provided by the component itself into five categories which are:

- Built-in Testing Approach.
- Testable Architecture Approach
- Metadata Approach
- Certification Strategy Approach
- Customer's Specification-based Testing Approach.

3.3.2.1 Built-in Testing Approach.

In this approach component developers equip component with embed tests to support self-testing. As a result, component user can run the embedded test cases in the final environment to validate the hypotheses made by the component developer. The main advantage of this approach is enhanced component testability. However, it has several drawbacks such as: the huge memory required for testing and it ignores component user.

3.3.2.2 Testable Architecture Approach

The testable architecture approach is a special case of the Built-In-Test approach; actually they share the same idea. The idea is that the component developer appends test information in the specifications form rather than embedding them in the component. This approach solves the memory consumption problem in BIT approach by separating component source code from test specification. Also in this approach, component user participates in specifying testing requirements so component provider can define test cases for component based on these specifications and this solves the second problem in the BIT approaches.

3.3.2.3 Metadata Approach

The black box nature of the components and lack of component documentation are considered as key problems that face CB testing. The objective behind the metadata approach is to equip the component with extra information so as to increase the component user's analysis capability and to simplify testing for component consumers.

3.3.2.4 Certification strategy approach

Actually, component user is always mistrustful about the information provided by the component developer until he executes the software component and checks the outcomes. Hence, to increase the trust between component user and developer, components can be certified before their reuse in CBS. Thus, a component should be certified by a third-party. In third-party certification, an independent organization tests the quality of the component and provides the test results, along with the test environment, to the component user.

3.3.2.5 Customer's specification-based testing approach

All of the previous approaches reliance on some cooperation and trust between the component developer and component user. In some approaches, component developer provides component structure and/or behavior information to users; while in some approaches producing component follows specific procedure. So user can depend on the component behavior only when he executes the test cases that he has extracted on the basis of the specifications of the component. In these techniques test cases are developed on the basis of the component user's specification and in the target environment.

3.3.3 System Testing

System testing is conducted by the component user when all components are perfectly integrated and the whole software is ready to run. The purpose of system testing is to test the whole system functionality and assess the performance of the entire system as a black box. In additional, performance and load testing are performed.

3.4 Control-Flow Testing and Structural Coverage Criteria

Control Flow Testing is a form of White-box testing. It is a testing technique which bases its tests on the structure of the code. In this technique source code of program is converted to control flow graph then a proper testing coverage criteria is selected and test cases are generated to satisfy testing coverage criteria.

Therefore, given a control flow graph, test cases can be generated accordingly and each test corresponds to a path. Unfortunately, there could be an unlimited number of paths in a control flow graph, and we need guidelines to determine how to derive test cases, and when to stop generating test cases. In testing, Test Requirement (TR) specifies things that must be satisfied or covered during testing. Test Requirements is defined by a collection of rules (Test Coverage Criterion). In this work we have three coverage criteria which are node coverage criteria, edge coverage criteria, and Pair –edge coverage criteria [26][27][28][29].

3.4.1 A node coverage criterion

A node coverage criterion requires that every statement (node) in the control flow graph be executed at least once [26]. For example the control flow graph shown in the Figure 3.4 represents a function; create a task in Smart Office System (SOS). This function receives both the number and date of the task and the following environmental variables (temperature, humidity and lighting) in addition to the start time and the end time of this task. Then the function scans the input data (temperature, humidity, and lighting) to make sure that these inputs are numbers or not. As well as examine both the task date and the start time and the end time of the task. Later the function checks whether there is any conflict in the date and time of the beginning and the end of the task with any previous tasks. According to the test result if there is any conflict, the process is canceled and the user is notified and unless the system creates a new task

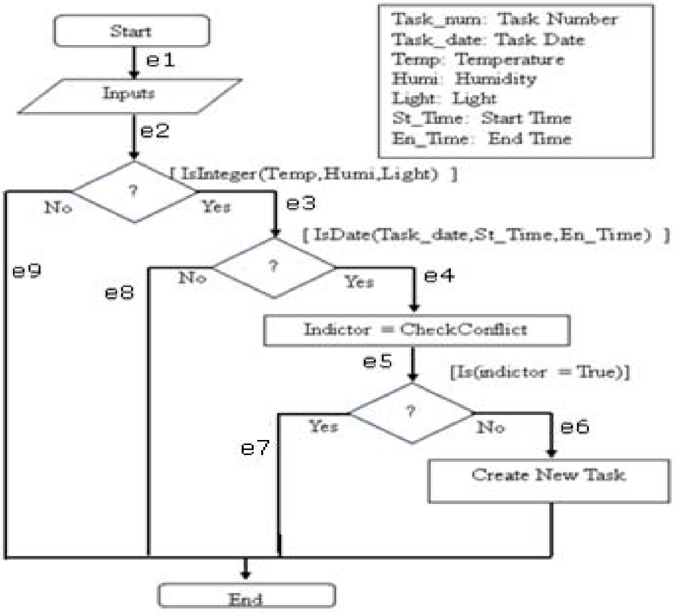


Figure 3.4 Control flow graph of create new task function

If the test requirement goal is test each statement (node) in the program at least one time, in this case we use the node coverage criteria. So, for this function we need only one test case to cover all nodes. If we apply the test case shown in the Table 3.1 in the create Task control flow graph, we see that all statements (nodes) are carried out and this is satisfied the test requirement goal.

Task number	Task date	Temperature	Humidity	Light	Start Time	End Time
1	1/2/2010	27	70	90	1:0:0 pm	5:0:0 pm

Table 3.1 Test case of the create new task function

3.4.2 Edge coverage criterion

An edge coverage criterion requires that every path (edge) in the control flow graph be visited at least once [26]. To demonstrate this coverage criterion, we use the same control flow graph for the function create task shown in Figure 3.4. In this criterion, the test requirement goal is test each path in the control flow graph at least one time. It is noted that there are 4 paths in this control flow graph ((e1,e2,e9), (e1,e2,e3,e8), (e1,e2,e3,e4,e5,e6), (e1,e2,e3,e4,e5,e7)); it means that we need 4 test cases to test this function, according to this criterion. So, If we apply the test cases in the Table 3.2 in the create task control flow graph, we observe that the first test case covers the path (e1,e2,e3,e4,e5,e6), while the second test case covers the path (e1,e2,e3,e8) and the third test case covers the path (e1,e2,e9) and finally the fourth test case covers the path (e1,e2,e3,e4,e5,e7).

Task num	Task date	Temperature	Humidity	Light	Start Time	End Time
129	2/2/2010	25	70	90	8:0:0 pm	9:0:0 pm
1	89j	25	70	90	1:0:0 pm	5:0:0 pm
1	1/2/2010	6h	70	90	1:0:0 pm	5:0:0 pm
1	2/2/2010	25	70	90	6:0:0 pm	7:0:0 pm

Table 3.2 Test cases of the create new task function

3.4.3 Pair –Edge Coverage Criterion

A pair–edge coverage criterion requires that all pair-path in the control flow graph must be executed at least once [26]. To illustrate this coverage criterion, we use the control flow graph of the function equalizeEnviroment shown in the Figure 3.5. This function is used to control the air condition and the humidifier in the office. The main inputs to this function are current temperature, last temperature, current humidity, and last humidity.

In the pair–edge coverage criterion, the test requirement goal is test each pair-edge in the control flow graph at least one time. It is noted that there are 4 paths ((e1,e2,e3,e4,e5,e6), (e1,e2,e7,e8,e9,e10), (e1,e2,e3,e4,e9,e10), and (e1,e2,e7,e8,e9,e10)) in this control flow graph. According to this criterion, we need four test cases to test this function. The test cases shown in Table 3.3 can cover all pair edges in this graph. The first test case covers the path e1,e2,e3,e4,e5,e6, while the second test case covers the path e1,e2,e7,e8,e9,e10 and the third test case covers the path e1,e2,e3,e4,e9,e10 and the fourth test case covers the path e1,e2,e7,e8,e9,e10.

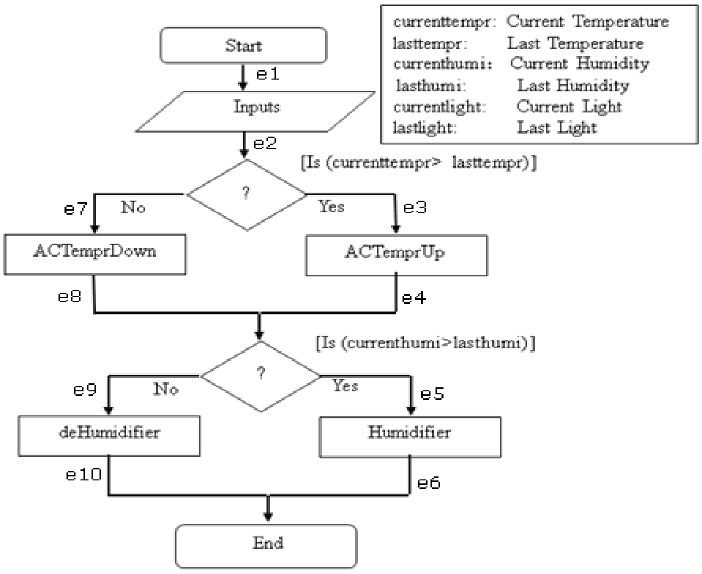

currenttempr: Current Temperature
lasttempr: Last Temperature
currenthumi: Current Humidity
lasthumi: Last Humidity
currentlight: Current Light
lastlight: Last Light
Start
e1
Inputs
e2
[Is (currenttempr> lasttempr)]
?
e7
No
Yes
e3
ACTemprDown
ACTemprUp
e8
e4
[Is (currenthumi>lasthumi)]
?
e9
No
Yes
e5
deHumidifier
Humidifier
e10
e6
End

Figure 3.5 Control flow graph of equalize environment function

Current Temperature	Last Temperature	Current Humidity	Last Humidity
26	25	75	70
25	26	70	75
26	25	70	75
26	25	75	70

Table 3.3 Test cases of the equalize environment function

3.5 Complexity Measures

In this work we have two measurements to measure the complexity of the glue-code which are:

- Component interface complexity
- Glue-code cyclomatic complexity

3.5.1 Component Interface Complexity

Fundamental to a component is its interface which describes the functionality provided. The interface defines the services provided by a component and acts as a basis for its use and implementation. It is one of the primary sources for understanding a component and often can be the only source available. An interface contains a set of operations that act as access points for an interaction with the outside environment. However, it is noteworthy that an interface is simply a collection of operations and only provides a description of them. An operation specifies how inputs, outputs and a component's state relate and the effect of calling the operations on that relationship. Mahmood et al. in [13] have developed a component interface complexity matrix based on the International Function Point User Group (IFPUG). They selected the IFPUG version of function point analysis (FPA) because it is an international standard and has been applied at design specification phase. Based on the UML interface information model, interfaces are classified as either internal logical file (ILF) or external input file (EIF). Interfaces that have operations that change the attributes of other interfaces in the data exchange are classified as ILF. All the remaining interfaces are classified as EIF. Then ILF and EIF are ranked based on the number of data element type (DET) and record element type RET, using RET/DET metrics. Since RET is a user recognizable subgroup, they count the number of operations (NO) in an interface. Similarly, DET is a unique user recognizable field; they count the number of parameters (NP) in an interface. By analogy with RET/DET metrics, they propose NO/NP complexity metrics that shows in Table 3.4, to rank candidate interface

	NP		
NO	**1-19**	**20-50**	**51+**
1	Low	Low	Average
2-5	Low	Average	High
6+	Average	High	High

Table 3.4 NO/NP Complexity Metrics

Ranked interfaces are assigned weights based on IFPUG standard weights as shown in Table 3.5.

Data Type	**Low**	**Average**	**High**
ILF_i	__ x 7=	__ x 10=	__ x 15=
ELF_i	__ x 5=	__ x 7=	__ x 10=

Table 3.5 Component interface complexity metrics

Let's go through this procedure to show how we can extract component interface complexity based on these measures. Suppose we have D.H. Controller component that has two interfaces IReadSensorValues and IEqualizeEnviroment as shown in Figure 3.6.

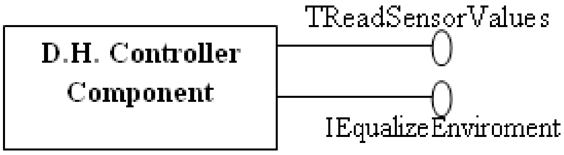


Figure 3.6 Digital Home controller component

Each interface contains a set of operations for example IReadSensorValue interface includes these operations:

- *getTemperaturenow(Out temp_Value int)*
- *getHumiditynow (Out humid_Value int)*
- *getLightnow (Out light_Value int)*
- *getappliaStatenow (Out State boolean)*

and IEqualizeEnviroment interface comprises these operations:

- *adjustControllers (in enviro_para TaskDetials, out enviro TaskDetials)*
- *setTemperature (in temp_Value int)*
- *setHumidity (in humid_Value int)*
- *setLight (in light_Value int)*
- *setappliaState (in State boolean)*

In the first step we determine the complexity of the interfaces. Component interface complexity is measured by applying this equation:

*Interface complexity = number of operation / number of parameters ……*Equation (3.1)

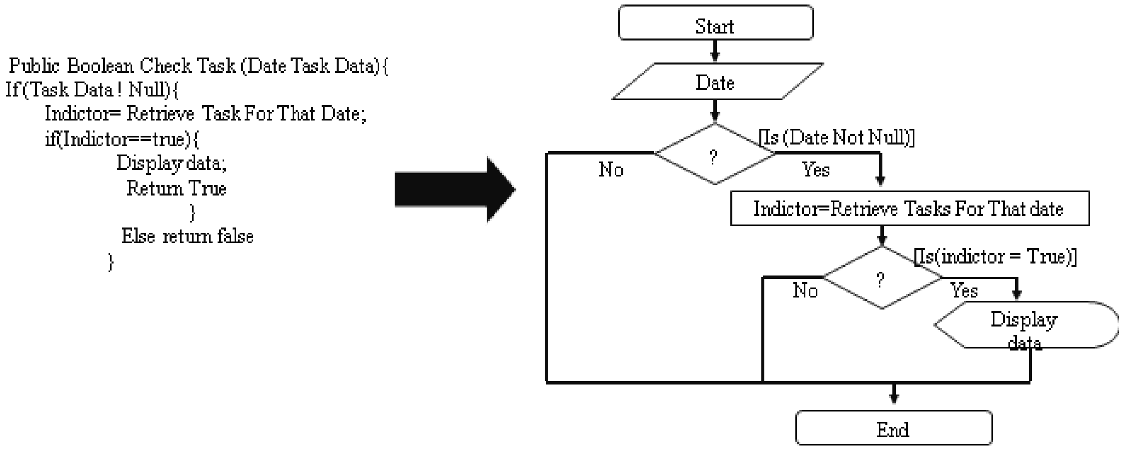
IReadSensorValue interface contains 4 operations and 4 parameters. Then based on Table 3.4 we give this interface a Low complexity as the number of operations is between 2 and 5 and the number of parameters is between 1 and 19.

However, the complexity of IEqualizeEnviroment interface is 5/21, 5 operations and 21 parameters. So we give this interface an average complexity. In the second phase we give a weight to each interface based on the metrics shown in Table 3.5. Based on the IFPUG interfaces are classified into two types ILF and EIF. IReadSensorValue interface is classified as EIF and IEqualizeEnviroment interface is classified as ILF because it contains operation (*adjustControllers*) that changes the attributes of other interfaces in the data exchange. Based on Table 3.5 we can now give the interfaces' weights. Since

IReadSensorValue has Low complexity and it is classified as ELF we give it 5 while IEqualizeEnviroment interface gets 10 because its complexity is an average and it is classified as ILF. We use these steps to extract the complexity of all components' interfaces that involved in system.

3.5.2 Glue-Code Cyclomatic Complexity

Conditional complexity or cyclomatic complexity is software metric used to measures the amount of decision logic in a single software module [26]. The source code of the program is converted into a control flow graph as shown in Figure 3.7.



```
 Public Boolean Check Task (Date Task Data){
If(Task Data ! Null){
      Indictor= Retrieve  Task For That Date;
      if(Indictor==true){
                 Display data;
                  Return True
                          }
                 Else  return false
             }
```

Figure 3.7 Source code converted into control flow graph

The cyclomatic complexity is measures by this equation:

Then Cyclomatic complexity = $E - N + 2$ *……………………….. Equation (3.2)*

Where E = the number of edges in the graph.

N = the number of nodes in the graph.

In literature, K. Goseva et al have introduced a model to measure component complexity similar to McCabe's cyclomatic complexity. But rather than use the source code control flow graph they used the UML state charts. The state chart of each component has a number of states (s) and transition (t) between these states that describe the dynamic behavior of the component. Therefore, component complexity was defined as CC=t-s+2, where t is the number of transitions and s is the number of states.

In this work we extract cyclomatic complexity for glue-code operations by using C and C++ Code Counter 'CCCC' tool.

3.5.3 Example

To illustrate the work we are going to give a simple example to show the whole picture. Suppose we have a system contains three components namely component A, component B, and component C. Each component has a set of interfaces for example component A has two interfaces IA1 and IA2 and component B also has two interfaces IB1 and IB2 while the component C has three interfaces IC1, IC2, and IC3 as shown in Figure 3.8.

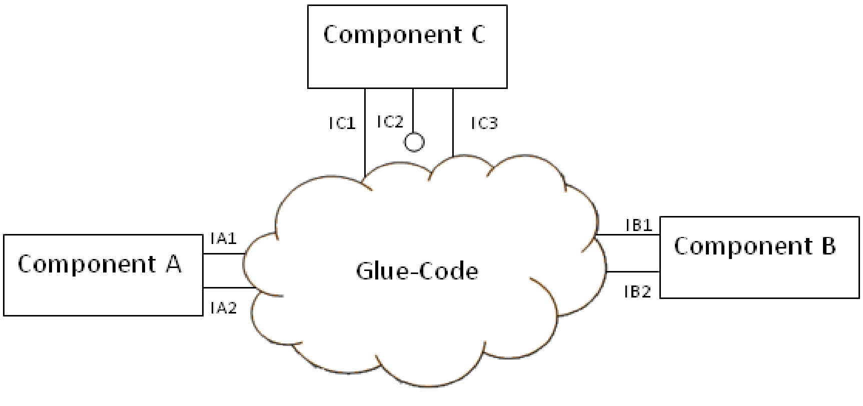


Figure 3.8 System with three components and glue-code

In addition, suppose we have already computed the interfaces complexity of all components as shown in Table 3.6 and we have used CCCC tool to extract the cyclomatic complexity of glue-code operations.

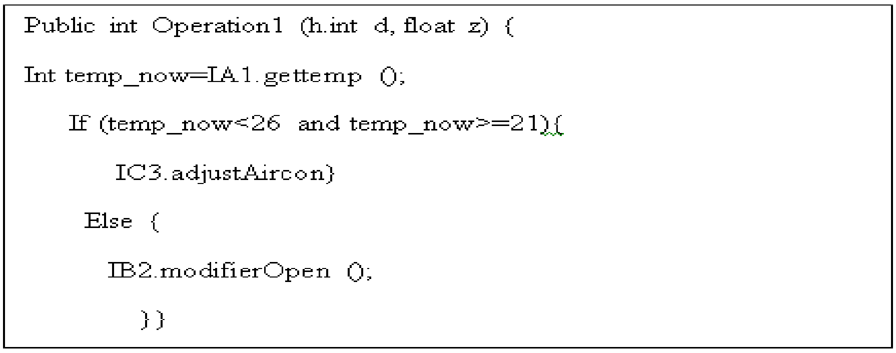
Component Interface Complexity		
Component Name	**Component interface**	**Interface Complexity**
Component A	IA1	10
Component A	IA2	5
Component B	IB1	15
Component B	IB2	10
Component C	IC1	7
Component C	IC2	7
Component C	IC3	15

Table 3.6 Components interfaces complexity

Let's now take a simple glue-code operation and compute its complexity. Figure 3.9 shows the source code of the glue-code Operation1.

Operation complexity is computed by this equation:

Operation cyclomatic complexity + interfaces complexity that are used in the operation.



```
Public int Operation1 (h.int d, float z) {
Int temp_now=IA1.gettemp ();
    If (temp_now<26 and temp_now>=21){
        IC3.adjustAircon}
     Else {
       IB2.modifierOpen ();
          }}
```

Figure 3.9 Source code of one glue-code operation

It is noted that there are three interfaces invoked in this operation which are IA1, IC3, and IB2. The complexities of these interfaces can be obtained from Table 3.6 (IA1=10, IC3=15, and IB2=10). It is clear that the cyclomatic complexity of this operation is 2. So the complexity of this operation is 2 +10+15+10=37.

We apply these steps for each operation in the glue-code. Table 3.7 shows the complexity of each glue-code operation.

Glue-Code Operations Complexity	
Operation Name	**Operation complexity**
Operation 1	2+10+15+10=37
Operation 2	20
Operation 3	68
………………..	…………

Table 3.7 Glue-code operations complexity

After the completion of calculating the complexity of each operation in the glue-code, we find the average complexity and the standard deviation. In this work we have proposed the following set of rules in order to select a suitable testing coverage criteria based on the operation complexity Figure 3.10 shows when the rules are applied.

Rule 1: Apply node coverage criteria for operation that have a complexity lower than Average – standard deviation.

Rule 2: Apply edge coverage criteria for operations that have a complexity between Average - standard deviation and average + standard deviation.

Rule 3: Apply pair edge coverage for operations that have a complexity greater than Average + standard deviation.

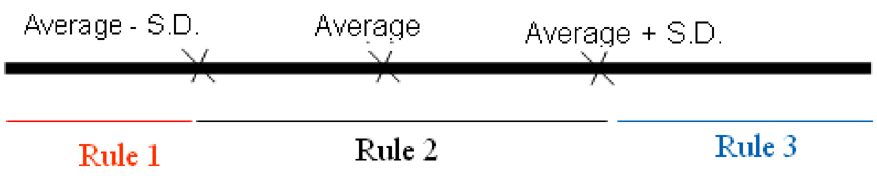


Figure 3.10 Rules that are applied in testing

Table 3.8 shows all glue-code operations and the appropriate test coverage.

Testing Rules	
Operation Name	**Testing Rule**
Operation 1	Rule 2
Operation 2	Rule 3
Operation 3	Rule 1
Operation 4	Rule 2
Operation 5	Rule 2
Operation 6	Rule 3
…………	..………………

Table 3.8 Testing glue-code operations and rules.

3.6 Parametric and Non-Parametric Test

3.6.1 Parametric Test

Parametric tests are conventional statistical procedures. In this test a sample statistic is obtained to compute the population parameter. Since this computation process includes a sample, a sampling distribution, and a population, certain parametric considerations are needed to ensure all components are compatible with each other [61]. Such as, there are three assumptions in Analysis of Variance (ANOVA) test which are:

- Observations are independent.
- The sample data have a normal distribution.

- Scores in different groups have homogeneous variances.

Examples of non-parametric tests are:

- t-test
- paired t-test
- Chi-square
- 1-Way Anova
- Pearson's r
- Factorial Anova

In the next part we will explain one parametric test which is t-test.

3.6.1.1 T-Test

The t-test estimates whether the means of two groups are statistically different from each other. The t-tests are based on the assumption that the data population is normally distributed [61]. Therefore, before we proceed with a t-test it is important to made a good estimate of our data's distribution, e.g. with Anderson-Darling test or Kolomogorov-Smirnov test. T-test also supposes that both groups share a common variance. So to check this assumption there are tests for example Bartlett's test or Levene's test. If the data do not conform to a normal distribution or don't share a common variance, the t-test will not produce reliable results and non-parametric tests are preferred.

Equation 5.1 shows how t-test is computed. The top part is the difference between the two means of the groups. The bottom part is a measure of the variability or dispersion of the scores.

$$t - test = \frac{\overline{X}_T - \overline{X}_C}{SE(\overline{X}_T - \overline{X}_C)} \qquad \text{.................. Equation (5.1)}$$

As shown in the equation 5.1 the top part could be found easily (the difference between the means) while the bottom part is called the standard error of the difference. To calculate the standard error of the difference, we take the variance for each group and divide it by the number of subjects in that group. Then we add these two values and take their square root as shown in the Equation 5.2.

$$SE(\overline{X}_T - \overline{X}_C) = \sqrt{\frac{var_T}{n_T} + \frac{var_c}{n_c}} \qquad \text{.................. Equation (5.2)}$$

The final equation for the t-test is shown below in Equation 5.3:

$$t - test = \frac{\overline{X}_T - \overline{X}_C}{\sqrt{\frac{var_T}{n_T} + \frac{var_c}{n_c}}} \qquad \text{.................. Equation (5.3)}$$

If the first mean is larger than the second the t-value will be positive and negative if it is smaller. Once we calculate the t-value we have to look it up in a table of significance to check whether the difference is large enough to say that the difference between the groups is not likely to have been a chance finding.

3.6.2 Non-Parametric Tests

Non-parametric tests have an advantage over the parametric tests as they are independent of the underlying distribution of the data population [61]. In situations where the normality of the population(s) is expect or the sample sizes are so small that checking normality is not really practicable; it is sometimes preferable to use nonparametric tests to make inferences about average value. Examples of non-parametric tests are:

- Wilcoxon signed rank test
- Whitney-Mann-UTest
- Kruskal-Wallis (KW) test
- Friedman's test

In the next part we will explain one non-parametric test which is Mann-Whitney Test.

3.6.2.1 Mann-Whitney Test

The Mann-Whitney test is used in experiments in which there are two conditions and different subjects have been used in each condition, but the assumptions of parametric tests are not required [61]. Mann-Whitney test is an alternative to the independent group t-test, when the assumption of normality or equality of variance is not met. This, like many non-parametric tests, uses the ranks of the data rather than their raw values to calculate the statistic. Since this test does not make a distribution assumption, it is not as powerful as the t-test. To compute this test we follow these steps:

Step 1: Rank data (taking both groups together) giving rank 1 to the lowest score, and so on.

Step 2: Find the sum of the ranks for the smaller sample- A in the example opposite- (if both samples are the same size, find the sum of the ranks of sample A). Call this T.

Step 3: Find $U = N_A N_B + \frac{N_A\ (N_A + 1)}{2} - T$

Where NA is the number of scores in the smaller samples (or, if both samples are the same size, the sample whose ranks were totaled to find T).

Step 4: Find $U' = N_A N_B - U$

Step 5: Look up the smaller of U and U' in the probability table. There is significant difference if the observed value is equal to or less than the probability table.

Step 6: Translate the result of the test back in terms of the experiment.

CHAPTER FOUR

EXPERIMENT SETTING

4.1 Experiment Definition

The general question of this study is analysing the effect produced by structure complexity measures on the defect detection effectiveness and effort in CB integration testing. Another important aspect is to investigate the impact of subjects experience and complexity measures on testing process. Based on this, three controlled experiments are conducted to analyze the above research question. In all experiments we use the same experiment design, procedure, and materials. Table 4.1 gives a brief overview of the most important elements of the experimentation.

Goal	Analysing the effect produced by structure complexity measures on the defect detection effectiveness and effort in CB integration testing.
Context	Academic environment.
Null hypotheses	(1) Complexity measures don't affect defect detection rate. (2) Complexity measures don't affect subjects' effort. (3) Subject experience with complexity measures don't impact defect detection effectiveness.
Main factor	Types of instruments used: three CBS (LMS, HRS, and SOS), systems specification with complexity measures vs. systems specification without complexity measures.
Dependent variables	Defect detection effectiveness, time required for testing

Table 4.1 Experimental design overview

4.2 Experiment context

Three distinct controlled experiments are conducted in an academic context. These experiments take place at King Fahd University of Petroleum and Minerals (KFUPM) in the Information and Computer Science department (ICS).

4.3 Hypothesis definition

Our experiments have one independent variable (the use of complexity measures) and two treatments (system description with complexity measures, and system description without complexity measures). The experiments have two dependent variables which are subject defect detection effectiveness and time needed for testing (effort). On the other hand, we investigate the impact of the subjects' experience on the defect detection effectiveness.

The null hypothesis for testing the effect of complexity measures on our dependent variables are as follow:

- *H_{01}: There is no difference in defect detection effectiveness of subjects who use complexity measures as compared to subjects who don’t use complexity measures.*

- H_{02}: *There is no difference in time spent on testing of subject who use complexity measures as compare to subject who don't use complexity measures.*

- H_{03}: *There is no significant difference in defect detection effectiveness of experienced subjects used complexity measures as compared to inexperienced subjects that use complexity measures.*

While the alternative hypotheses are:

- H_1: *There is difference in defect detection effectiveness of subjects who use complexity measures number as compared to subjects who don't use complexity measures.*

- H_2: *There is difference in time spent on testing of subject who use complexity measures as compare to subject who don't use complexity measures.*

- H_3: *There is significant difference in defect detection effectiveness of experienced subjects who use complexity measures as compared to inexperienced subjects who use complexity measures.*

Null hypotheses H_{01} and H_{02} are two-tailed. Ideally, complexity measures should improve the glue-code testing performance since these numbers guide subjects during testing. Even so, they might confuse the subjects making the testing harder. For this reason, H_{01} is two-tailed. Nothing can be said on subjects effort (time) during testing, that can be either increased or reduced when complexity measures is used.

Our research model is shown in Figure 4.1. This model shows our independent variable, dependent variables.

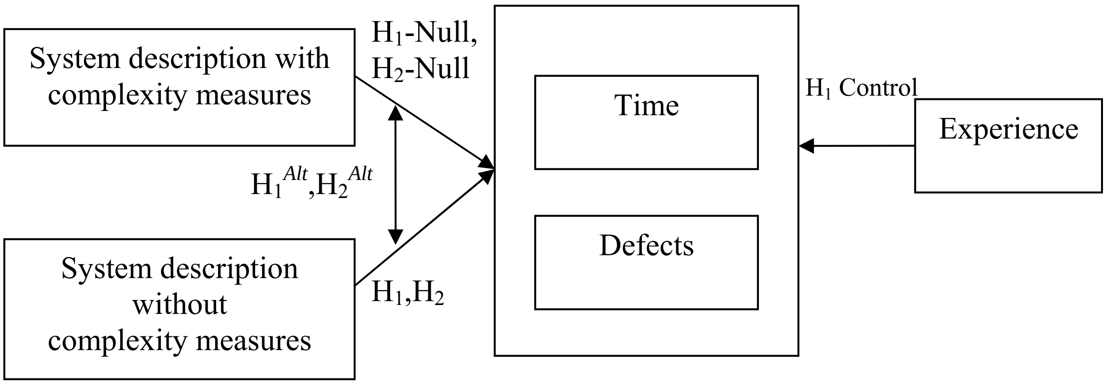

System description with complexity measures
H1-Null, H2-Null
H1Alt,H2Alt
System description without complexity measures
H1,H2
Time
Defects
H1 Control
Experience

Figure 4.1 Research model

4.4 Subject Selection

The experiments incorporated a total of 37 subjects. These experiments have 15, 10, and 12 subjects, respectively. Third and fourth year software engineering students, master and doctoral students in the information and computer science department are participated in these experiments. Table 4.2 shows subjects' details in each experiment.

They already have a firm ground in many aspects of computer science also they have previously taken different numbers and types of courses in computer science, software engineering domain and all of them had experience from a previous software engineering courses in software testing.

The subjects' experiences are captured by a pre-questionnaire, before the experiments. The collected data shows that all subjects at least take two courses in software engineering. In addition, there are 15 subjects who have an industrial experience in software development ranging from 1 year to 3 years. These data allow us to manage subjects' abilities during the experiments.

Level of Subject	*Experiment 1*	*Experiment* 2	*Experiment* 3	Total
Doctoral	5	1	-	6
Master	9	6	2	17
Bachelor	1	3	10	14
Total	15	10	12	37

Table 4.2 Levels of subjects

4.5 Preparation and Training

Since our subjects already have a firm ground in many aspects of computer science and all of them had experience from a previous software engineering courses in software testing, we conduct a presentation to refresh student's understanding of testing coverage criteria (node coverage, edge coverage, and pair edge coverage) and to show them how to extract test cases from control flow diagrams. In addition, we show them how we can extract complexity measures for glue-code operation. After that we do a demo with the Smart Office System (SOS) to show them how to use complexity measures when they test systems and how to fill and describe the detected faults. Finally, we give them a general overview about systems tested namely, Hotel Reservation System (HRS) and Library Management System (LMS) to assist them during testing.

4.6 Experiment Material

Three distinct systems are involved in these experiments: a Hotel Reservation System (HRS), a Library Management System (LMS), and a Smart Office System (SOS). These systems are adopted from [23], [24], and [25].

The HRS allows guests to make reservations in any hotel in the chain. Also it helps them to looking for room in different hotels in a certain date; and it allows them to confirm or cancelling their reservation. In addition it allows hotel chain management to do various operations such as add new hotel, add rooms to the hotel and all operations that manipulate hotels and rooms data.

The LMS helps a library employee to manage the loan of items. Members can search for items, borrow items, return items or renew items (i.e., extend a current loan). There are two types of loan items; journal and book. A member may borrow up to a maximum of 5 items. An item can be borrowed, or renewed to extend a current loan. Each of these activity has a cost in S.R. (borrow a book cost 10 S.R. while a journal only 5 S.R.; if the member performs at least 3 operations – i.e., borrow and/or renew in the same day, he receives a discount of 7 S.R.). The library system must support the facility for an item to be searched and for updating the items and members.

The SOS allows any ready computer to control an office's environment parameters (temperature, humidity, lights and the state of small appliances (e.g. refrigerator, TV, printers etc.)). It is equipped with various environment sensors (temperature sensor, light sensor, humidity sensor, power sensor, etc.) and controller devices such as Air-Condition, Humidifier, etc.… The network connection will be used to connect sensors and allows the system to manage devices. This system allows user to control office environment by applying profile for specified date. Each profile contains the environment parameters for one day. However user can mange the system manually by providing the system the environment parameters and it changes office environment based on the entered values.

HRS and LMS are used in the experiment while SOS is used in the training. Each system is enhanced with a high-level textual description of system objectives and control flow diagrams for glue-code operations in order to help subjects to understand how glue-code operations work and to make test case extraction easier. In addition to systems description and control flow diagrams each glue-code operation is assigned a complexity number. This number tells subject which coverage criteria he should use to test the operation. See Appendix B for more details about system description and control flow graph for glue code operation and its complexity numbers.

HRS contains 32 faults while LMS contains 31 faults. The injected faults are randomly seeded during the actual development of the systems by another specialist to avoid any bias in the experiments setting. These systems are built from scratch based on CBD process [23]. Table 4.3 provides relevant metrics for the LMS and HRS.

Metrics	LMS	HRS
Number of Components	4	4
Number of ILF Interfaces	4	4
Number of EIF Interfaces	4	3
Number of Low complexity interfaces	3	3
Number of Average complexity interfaces	3	3
Number of High complexity interfaces	2	1
Number of operations in Interfaces	40	31
Number of Glue-code operations	18	15
Number of Faults	31	32
Number of LOC (glue-code)	2826	3254
Number of High complexity operations	3	4
Number of Average complexity operations	11	8
Number of Low complexity operations	4	3
McCabe's Cyclomatic Number	119	189

Table 4.3 Metrics details of the LMS and HRS

In addition to the systems and their documentations, we adopt a faults collection form and two questionnaires (pre-questionnaire and post-questionnaire) from [53]. The pre-questionnaire aim at collecting subjects experiences. It composes five questions related to subject experience and its abilities. Each subject must compile them before one day of his participation. This allows us to collect subjects experience and then assigns subjects into groups based on the collected data. After the experiment, we ask the subjects to fill a feedback questionnaire (post- questionnaire). This questionnaire aims to gather subjective information about validating the internal validity of our experiment. Question 1 through question 7 are targeted the availability of sufficient time to end the testing, the simplicity of the system description and application, and the ability of subjects to understand them. Finally, the last question is devoted to evaluate the perceived usefulness of complexity measures. The post- questionnaire details are available in Appendix C.

4.7 Experiment design

Since our lab session last 4-hours, we select a very simple design which is one factor with two treatments [54] [55]. The reasons behind adopt this design are:

- Each subject will practice with both treatments of the main factor.
- This design can be used when a limited time slot is available for the experiment. [53]

In these experiments we have two objects and two treatments. The objects are HRS and LMS, and the treatments are the following:

+ System description with complexity measures.

- System description without complexity measures.

In our experiments, we use the concept of blocking to mitigate the effect of individuals and groups abilities. By analogy with [56], [57], the assigning of subjects is based on the number of the software engineering courses taken by the subjects, and the number of industrial experience years of subjects (collected before the experiments, pre-questioner). As a result, two groups are considered as shown in Table 4.4.

In Experiment 1, there are 6 subjects in the first group (have industrial experience) and in the second group 9 subjects (have taken different courses in software engineering). In Experiment 2, there are 4 subjects in the first group whereas 6 subjects are in the other group. However, in Experiment 3, there are also 5 subjects in the first group while 7 subjects are in the second group. The subjects in each group are randomly split into two groups (Red and Yellow), receiving the combination of treatments shown in Table 4.4. The effect of subjects and group abilities are mitigated by allowing subjects to test another application but this time using the alternative approach.

	Group 1	Group 2
Experiment 1	6	9
Experiment 2	4	6
Experiment 3	5	7

Table 4.4 Number of subjects in each group in the experiments

Our experiment design is ordered as two modules. In the first module, the first 45 minutes is utilized to present a brief presentation about testing coverage criteria concepts and CBS complexity concepts. The second module of the session is two exercises (one hour and half each) that form the experiment. In the first exercise, the Red group test LMS (with complexity measures) and the Yellow group test the same application (but without complexity measures). There is fifteen minutes break between the exercises. In the other exercise, subjects in groups are swapped that means subjects who are in Red group became on the Yellow group and vice versa.

For each exercise, each subject work separately on each of the two systems, using complexity measures in one case and without complexity measures in the other case.

Collaboration among subjects and swap of information are prevented and individual work could simply be monitored because testing was conducted in a laboratory setting.

4.8 Measurements

In these experiments we investigate two dependent variables which are the subject defect detection effectiveness and the effort.

Subject defect detection effectiveness refers to the number of defects reported by subject (defects that have been injected by specialist).

The time needed for testing system for each subject is measured by this equation:

$$\text{Subject\_Time}_{j=1}^{m} = \sum_{i=1}^{n} Time\,\mathrm{Re}\,quiredForTesting(Operation_i)$$

Where n is the number of operations, m is the number of subjects, and $Operation_i$ is the time that subject spend to test operation *i*. Subject have to state the start and the end time for each exercise and to be more precise we ask them to state the time that they consume in testing each operation in the system.

Subject experience is measured by the number of industrial experience years in the software development area. Table 4.5 gives description for each measure.

Dependent variable	Measurement
Defect detection effectiveness	Number of defects reported by subject
Time	$\sum_{i=1}^{n} Time\,\mathrm{Re}\,quiredForTesting(Operation_i)$
Experience	Years of industrial experience

Table 4.5 Dependents variables and its measures

CHAPTER FIVE

RESULT ANALYSIS and DISCUSSION

This chapter reports the analysis of the experiments described previously in Chapter 4. Overall 37 subjects take part to the experimentations. Two types of data are collected during these experiments, time data and defect data. Time data shows how much time each subject spent during testing. Whereas, the defect data shows the number of defects are detected by the subject. As part of our analysis we assume that hypothesis testing significance level is 0.05 ($\alpha = 0.05$). A box-plot graph is used to illustrate the analysis results for the Red and Yellow groups (with complexity measures and without complexity measures).

5.1 Defect Detection Effectiveness

To measure the effectiveness of complexity measures in detecting faults; we compare the number of faults find by subjects who are in the Red group (with complexity) and the subjects who are in the Yellow groups (without complexity).

The descriptive statistics are shown in Tables 5.1, 5.2, and 5.3. In Figure 5.1, the box-plot summaries the data collected from the first exercise (LMS). While Figure 5.2 shows the box-plot that presents the data collected from the second exercise (HRS). Figure 5.3 shows the box-plot that summaries the overall data collected for both exercises.

The mean defect detection effectiveness for the Red and Yellow groups in the first exercise (LMS) is 15.17 and 13.11 respectively as shown in Table 5.1.

Treatment	Exercise 1 (LMS)					
	Number of Subjects	Min	Med	Max	Mean	SD
Red Group	18	11	15	23	15.17	2.77
Yellow Group	19	7	13	16	13.11	2.36

Table 5.1 Descriptive statistics of defect detection effectiveness for the first exercise (LMS)

In Table 5.2 we can also observe the same results in the second exercise (HRS); the mean of defect detection effectiveness for these groups is 14.68 and 12.89 respectively.

Treatment	Exercise 2 (HRS)					
	Number of Subjects	Min	Med	Max	Mean	SD
Red Group	19	10	15	19	14.68	2.5
Yellow Group	18	8	13	17	12.89	2.4

Table 5.2 Descriptive statistics of defect detection effectiveness for the second exercise (HRS)

The overall data for both exercises are shown in Table 5.3; the mean of the Red and Yellow groups is 14.92 and 13.03 respectively.

Treatment	Both Exercise					
	Number of Subjects	Min	Med	Max	Mean	SD
Red Group	37	10	15	23	14.92	2.61
Yellow Group	37	7	13	17	13.03	2.33

Table 5.3 Descriptive statistics of defect detection effectiveness for both systems (LMS & HRS).

Parametric (t-test) and non-parametric (Mann-Whitney U-test) tests are used to test the first null hypothesis (H_{01}). The results indicate that the null hypothesis H_{01} can be rejected because p-values of the t-test and Mann-Whiteny U-test are 0.02 and 0.04 (p-value<0.05) for the first exercise (LMS). Furthermore, the p-values of the second exercise (HRS) are 0.03 for t-test and 0.03 for Mann-Whiteny U-test which also means the null hypotheses H01 can be rejected (p-value<0.05).

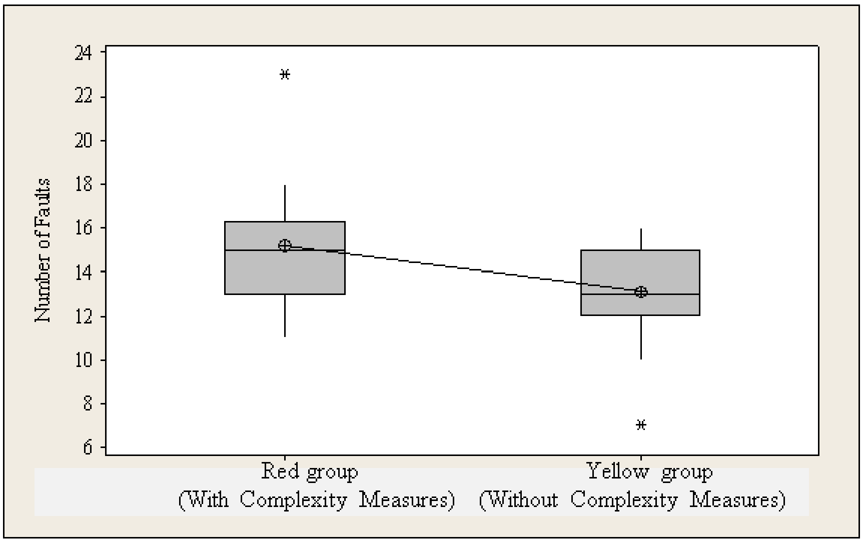


Figure 5.1 defect detection effectiveness for LMS

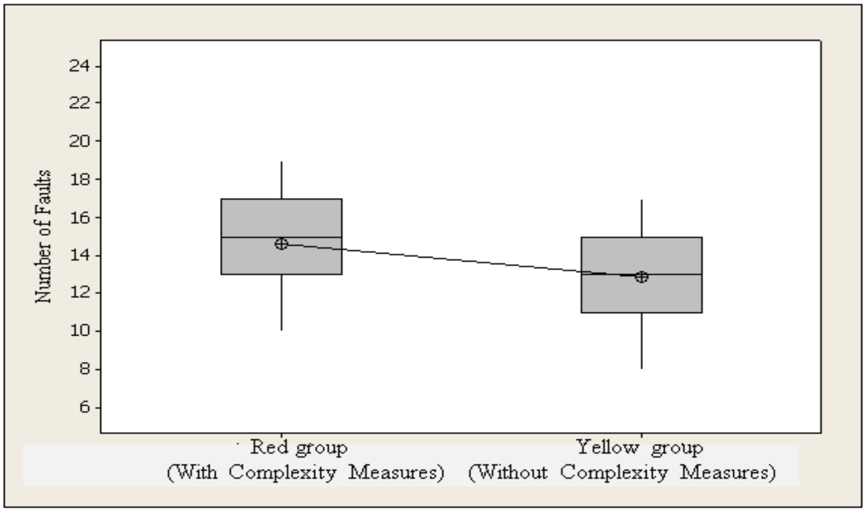


Figure 5.2 Defect detection effectiveness for HRS.

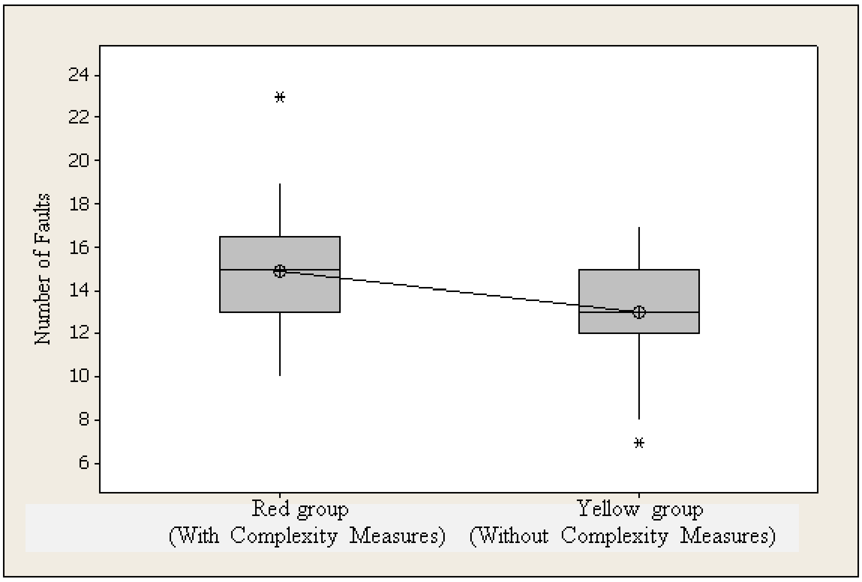


Figure 5.3 Defect Detection Effectiveness for both LMS and HRS

In other words, it is statistically significant that subjects who use complexity measures in testing detect more faults than subjects who do not use complexity measures in testing.

5.2 Time Required for Testing

We also compare the time spent on testing of subjects who are in the Red group and the Yellow group.

The descriptive statistics of the time spent on systems testing, are shown in Tables 5.4, 5.5, and 5.6. In Figure 5.4, the box-plot summaries the data collected during the first exercise (LMS). While Figure 5.5 shows the box-plot that presents the data collected during the second exercise (HRS). Figure 5.6 shows the box-plot that presents the overall data collected for both exercises.

Table 5.4 and Figure 5.4 show the mean time that is required for testing in the first exercise (LMS) which is 70.78 for the Red group and 70.58 for the Yellow group.

Treatment	Exercise 1 (LMS)					
	Number of Subjects	Min	Med	Max	Mean	SD
Red Group	18	64	70	78	70.78	3.92
Yellow Group	19	68	70	75	70.58	2.01

Table 5.4 Descriptive statistics of subjects effort (time requried for testing) for the first exercise (LMS)

The same observation is noticed in the seconded exercise (HRS) as shown in Table 5.5 and Figure 5.5. No significant difference is visible in the effort spent in testing between subjects in both groups; the means are 69.79 and 69.33 respectively.

Treatment	Exercise 2 (HRS)					
	Number of Subjects	Min	Med	Max	Mean	SD
Red Group	19	65	69	75	69.79	3.16
Yellow Group	18	65	69	74	69.33	3.01

Table5.5 Descriptive statistics of subject's effort (time required for testing) for the second exercise (HRS)

Table 5.6 and Figure 5.6 show the overall data for time in both exercises and it confirms that no significant difference is noticed between the two groups, the mean of the Red and Yellow groups is 70.27 and 69.97.

Treatment	Both Exercise					
	Number of Subjects	Min	Med	Max	Mean	SD
Red Group	37	64	70	78	70.27	3.53
Yellow Group	37	65	70	75	69.97	2.59

Table5.6 Descriptive statistics of subjects effort (time required for testing) for both exercises

Since data distribution for time, for both systems, is not normal (Anderson-darling p-value =0.03), we test the Null hypothesis H_{02} by using a non-parametric test (Mann-

Whitney U-test). The results show that the hypothesis H_{02} can't be rejected because the p-value of the Mann-Whitney U-test for the first exercise (LMS) is 0.99 (p-value >0.05) and the p-value of the Mann-Whitney U-test for the second exercise (HRS) is 0.59 (p-value >0.05).

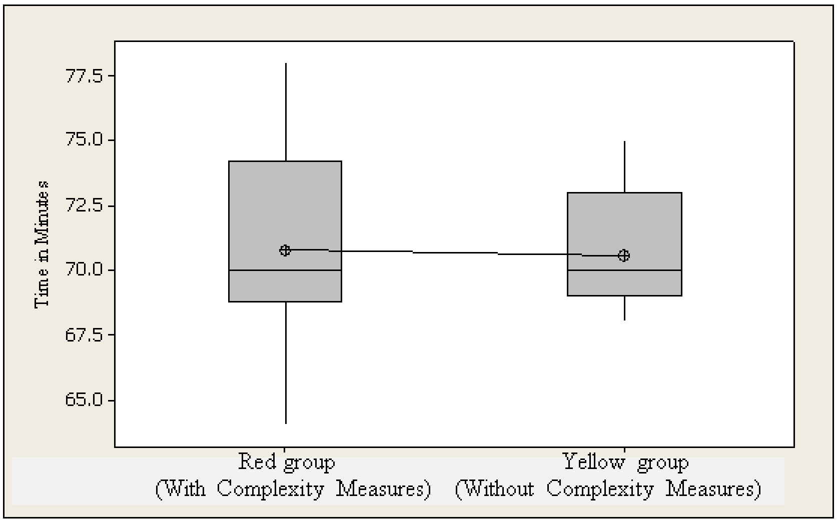


Figure 5.4 Time required for testing LMS

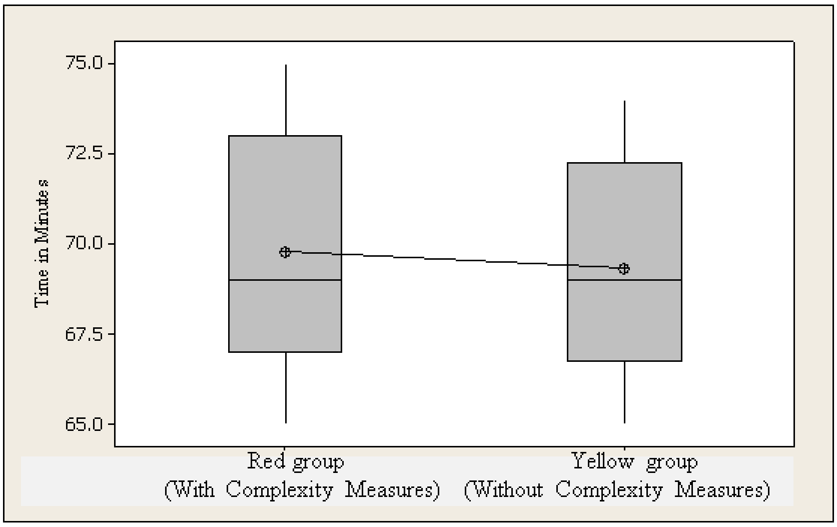


Figure 5.5 Time required for testing for HRS

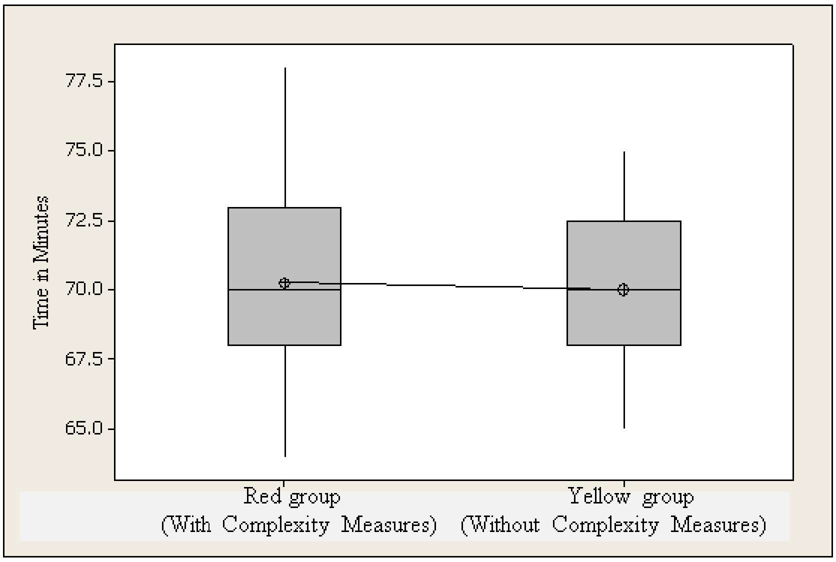


Figure 5.6 the time required for Testing LMS and HRS

In other words, There is no statistical significant difference in the time required for system testing between subjects who use complexity measures (Red group) and those who don't use complexity measures (Yellow group).

5.3 Impact of Subjects Experience and Complexity Measures on Defect Detection Effectiveness

In this section, we investigate the impact of subject's experience and the complexity measures on the defect detection effectiveness. We compare the defect detection effectiveness of experienced subjects that have used complexity measures with inexperienced subjects that have also used complexity measures during testing (Red group). The descriptive statistics of this case are shown in Tables 5.7, Table 5.8, and Table 5.9. Figures 5.7, Figure 5.8, and Figure 5.9 show the box-plots that summaries the data collected from the first exercise (LMS), the second exercise (HRS), as well as the combined data of the both exercises.

The mean of detecting faults for experienced and inexperienced subjects in the first exercise (LMS) is 15.75 and 14.7 respectively as shown in Table 5.7. It is evident that no significant difference is visible for the defect detection effectiveness between experienced subjects and inexperienced subjects.

Treatment	Exercise 1 (LMS)					
	Number of Subjects	Min	Med	Max	Mean	SD
Experience subjects	8	11	15	23	15.75	3.69
Inexperience subjects	10	11	15	17	14.7	1.83

Table 5.7 Descriptive statistics of subjects (experienced & inexperinced) with complexity measures for LMS.

Figure 5.8 and Table 5.8 show the descriptive statistics of the second exercise (HRS); the mean of defect detection effectiveness for experienced subjects and inexperienced subjects is 15.57 and 14.17 respectively.

Treatment	Exercise2 (HRS)					
	Number of Subjects	Min	Med	Max	Mean	SD
Experience subjects	7	11	17	19	15.57	3.21
Inexperience subjects	12	10	14	17	14.17	1.95

Table 5.8 Descriptive statistics of subjects (experienced & inexperinced) with complexity measures for HRS.

The mean of the overall data for experienced subjects and inexperienced subject is 15.67 and 14.41 respectively as shown in Table 5.9 and Figure 5.9.

Treatment	Both Exercise					
	Number of Subjects	Min	Med	Max	Mean	SD
Experience subjects	15	11	16	23	15.67	3.35
Inexperience subjects	22	10	15	17	14.41	1.87

Table 5.9 Descriptive statistics of subjects (experienced & inexperinced) with complexity measures for LMS & HRS.

To test the third null hypothesis H_{03} t-test is used and the results indicate that H_{03} can't be rejected as the p-value for the first exercise (LMS) is 0.5 and the p-value for the second exercise (HRS) is 0.32.

In other words, it is not statistically significant that experienced subjects who use complexity measures in testing detect more faults than inexperienced subjects who have used complexity measures in testing.

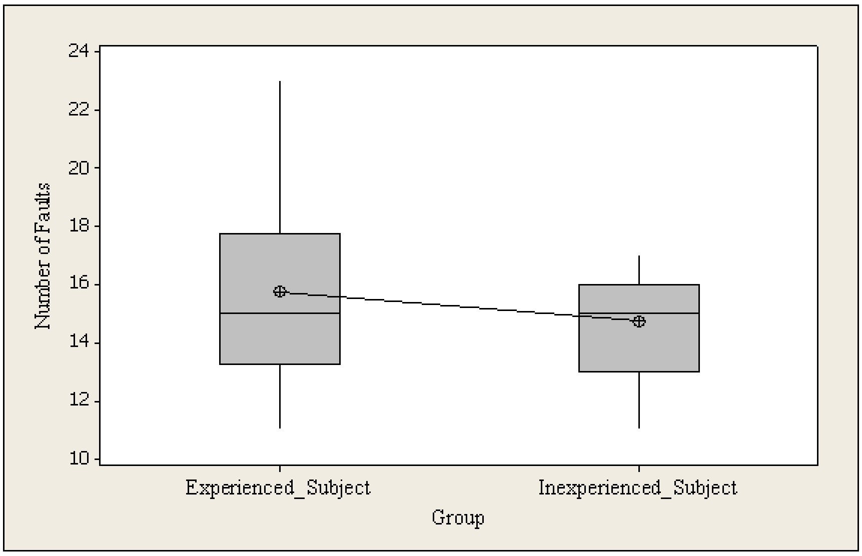


Figure 5.7 Impact of subjects experience and complexity measures on defect detection effectiveness in the first exercise (LMS)

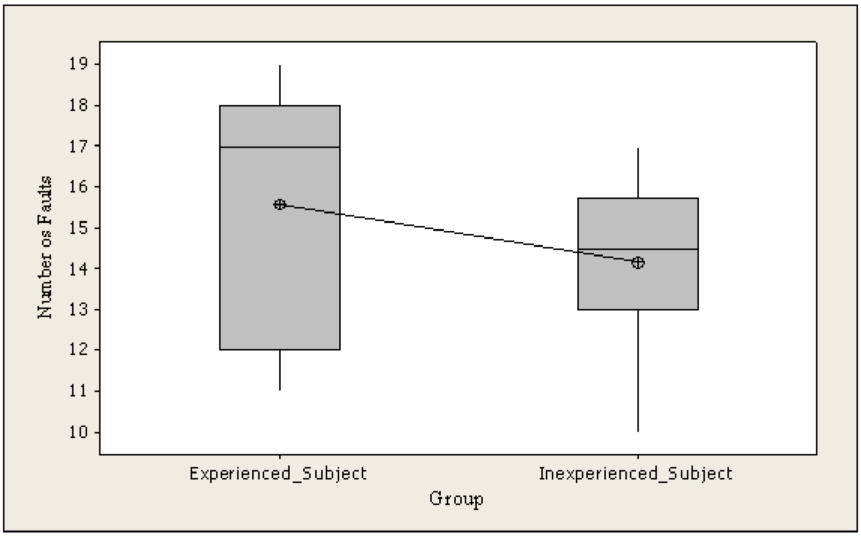


Figure 5.8 Impact of subjects experience and complexity measures on defect detection effectiveness in the second exercise (HRS)

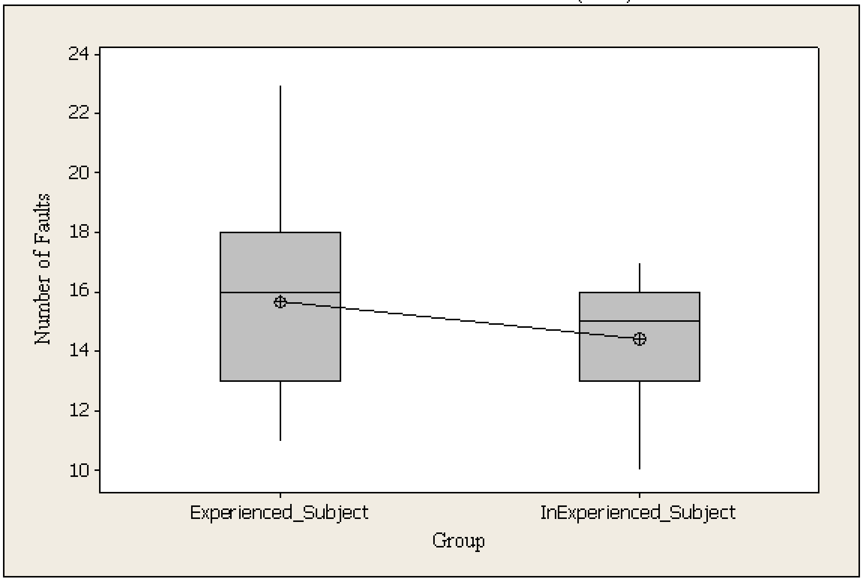


Figure 5.9 Impact of subjects experience and complexity measures on defect detection effectiveness for both LMS and HRS.

5.4 Complexity Measures and Defect Types

Instead of just investigating the effectiveness of complexity measures in CB integration testing (Glue-code testing), we examine the type of faults found. In this work 31 and 32 faults are seeded in the experiments' objects (LMS and HRS). These faults are classified into three types which are logical error, omission error, and computational error as shown in Table 5.10.

	Logical error	Omission error	Computational error
LMS	14	16	1
HRS	15	14	3

Table 5.10 Types of faults seeded in the systems

Figures 5.10 and 5.11 are frequency diagrams showing the number of subjects that found each fault respectively in the LMS and HRS. With respect to the 31 faults of the LMS, almost 1 fault is found by all subjects, 12 faults by at least one half of the subjects, and 6 faults are not found by any subject. For the HRS, no faults are found by all subjects, just 14 defects by at least one half of the subjects, and 5 faults are not found by any subject.

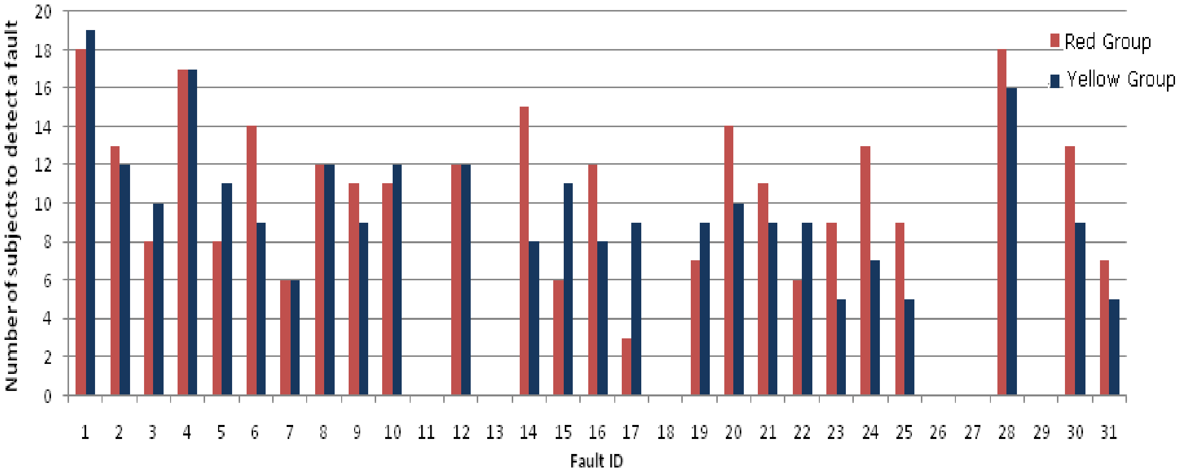


Figure 5.10 Numbers of subjects that found each LMS fault during testing

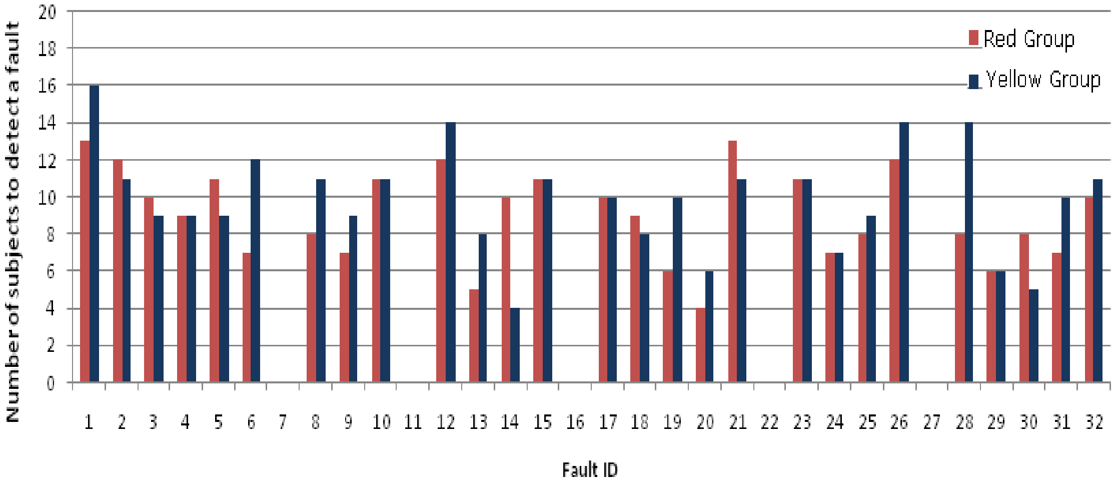


Figure 5.11 Number of subjects that found each HRS defect during testing

As mentioned previously, the faults that have been injected in the systems (LMS and HRS) are classified into three types, logical faults, omission faults and computational faults. We concentrate on the logical and the omission faults because the number of computational errors is small.

5.4.1 Result Analysis for Logical Faults

The first exercise (LMS) includes 14 logical errors and the second exercise includes 15 logical faults. Figures 5.12 and 13 are frequency diagrams showing the number of subjects that found each logical fault in the LMS and HRS. In general, the most important thing on these charts is that subjects in the Red group detected more logical errors than subjects who are in the Yellow group, except fault number 10 in the LMS and fault number 5 in the HRS because subjects used the rule 1 (node coverage criteria) to test the operations that contain those faults. It is clear that the subjects who used the structure complexity measures are able to discover many logical faults. This indicates the effectiveness of a structure complexity measures in the discovery of logical faults

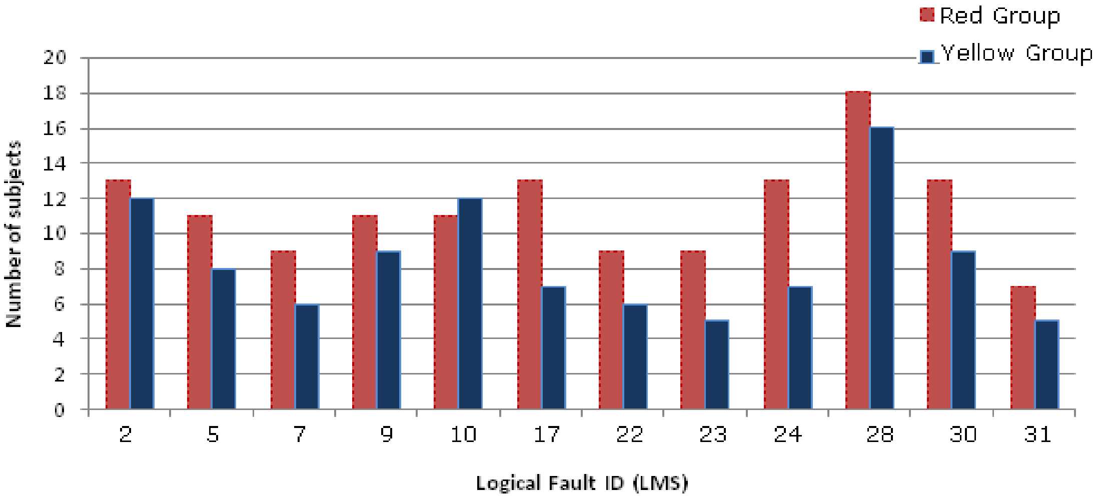


Figure 5.12 logical faults detected by subjects in the first exercise (LMS)

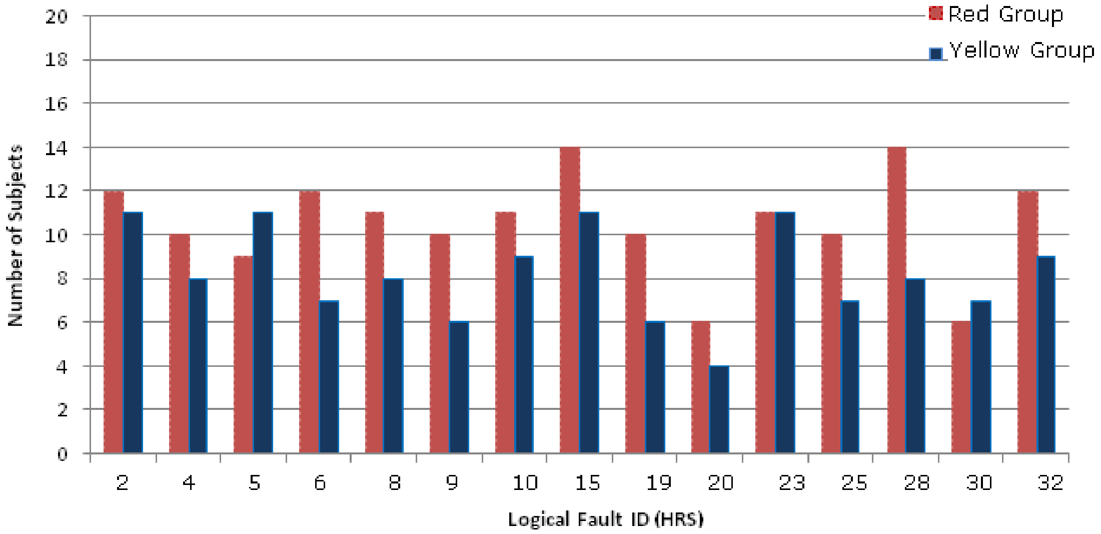


Figure 3.13 logical faults detected by subjects in the second exercise (HRS)

5.4.2 Result Analysis for Omission Faults

The first exercise (LMS) includes 16 omission faults while the second exercise (HRS) includes 14 omission faults. Figures 5.14 and 5.15 are frequency diagrams showing the number of subjects that found each omission fault respectively in the LMS and HRS. In Figure 3.14 we can see that subjects who didn't use complexity measures discover more omission faults than subjects who use complexity measures except faults number 14, 20, and 25 because subjects in the red group used the third rule (pair-Edge converge) to test the operations that include these faults. However, in the second exercise subjects that use complexity measures detect more omission faults than subjects in the other group except faults number 12, 13, and 30 because subjects that use complexity measures use the first rule (node coverage) and the second rule (edge coverage) to test the operations that include those faults.

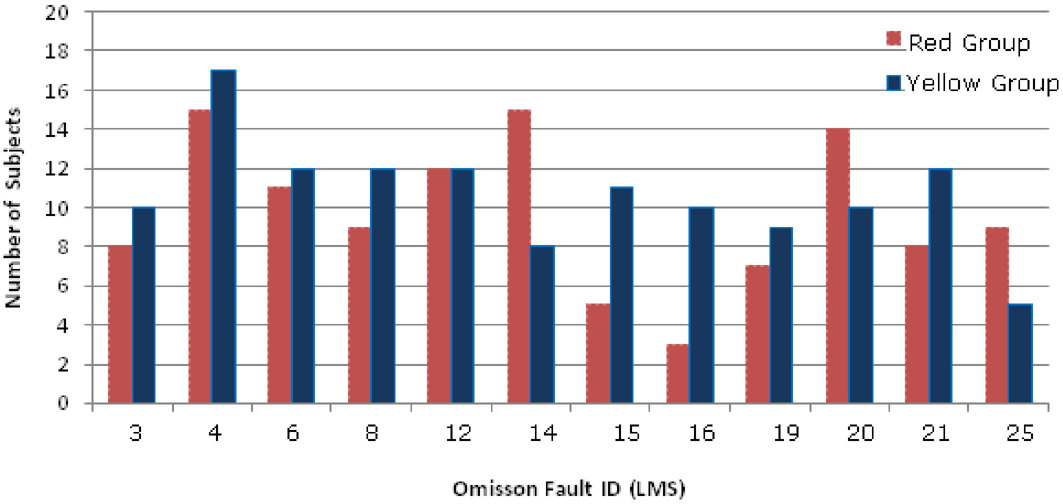


Figure 3.14 faults detected by subjects in the first exercise (LMS).

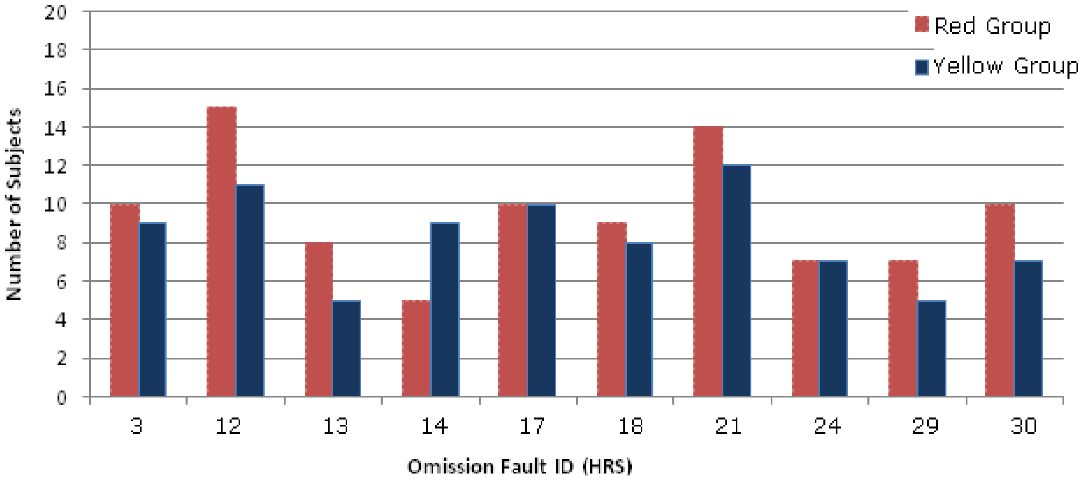


Figure 3.15 omission faults detected by subjects in the second exercise (LMS).

5.5 Debriefing questionnaire

The experiment has another source of data for analysis that is the post-questionnaire compiled by subjects after the experiments. The post-questionnaire is available in the Appendix C. Likert scale is used to rate the results; the first question through the six question are rated as follows: 5: strongly agree; 4: agree; 3: not certain; 2: disagree; 1: strongly disagree. While the last question number seven is rated in this way: 5: very much; 4: enough; 3: undecided; 2: little; 1: definitely not.

ID	Question	Mean	Max	Min	SD
Q1	I had sufficient time to do testing	3.64	5	1	1.03
Q2	The purpose of the exercises were clear to me	4.14	5	2	0.76
Q3	The description of the systems were unambiguous	4.42	5	4	0.50
Q4	The Control Flow Diagrams were clear to me	5.71	5	4	0.46
Q5	I experienced no difficulty in reading/understanding the Control Flow Diagrams	4.57	5	4	0.50
Q6	I experienced no difficulty in reading/understanding the complexity numbers	3.92	5	3	0.47
Q7	Did you find complexity (when available) useful in testing?	4.1	5	3	0.77

Table 5.11 Post-questionnaire data analysis

Table 5.11 summarizes the data collected from subjects' post-questionnaires. As we can see the answers to question 1 to question 6 confirmed that the subjects are able to understand the material provided within the time dedicated for the experiments (mean between not certain and agree for the first question) . Finally, we can observe (Q 7) that subjects deemed the structure complexity measures as a useful guideline during CBS integration testing (mean between enough and very much).

5.6 Threat to Validity

As any empirical study, these experiments exhibit a number of threats to internal validity, conclusion, construct, and external validity. In this section, we discuss the threats to the validity that can affect these experiments.

Internal validity

Internal validity investigates whether the study may be distorted by influences that impact dependent variables without the research's knowledge**.** This possibility should be minimized. The following such threats are considered:

- Subject selection effects that may happen as a result of deviation in the performance of subjects. This is minimised by creating equal ability groups as discussed in Section 4.7.
- The effect of instrumentation that may arise as a result of variation in the experimental objects used. Such variation is impossible to avoid, but we control it by having each subject test both systems (LMS AND HRS).

External validity

External validity threats can limit the ability to generalize the experiment results to a wider population. We consider the following threats:

- The biggest threat to the external validity is those students, instead of practitioners, are used as subjects. Several studies indicate that students can be used as subjects [58][59][60].
- The experiments objects represent real applications, LMS and HRS. Their complexity and size are designed to be compatible to the time available for experiments (a 4-hour laboratory session). These case studies are adopted from [23] and [24] respectively. These systems are built from scratch based on the CBSD process. Defects are randomly seeded in the systems by another specialist to avoid any bias in the faults seeding.
- Experiment time, in industrial the time required for testing is larger than what we are allocated to our experiments. We minimize the time effect on the results of our hypotheses testing by performing a training presentation before experiments; we give explanations on the testing coverage criteria and complexity measures and a guideline on how to perform testing as well as an overview for HRS and LMS.

CHAPTER SIX

CONCLUSIONS AND FUTURE WORK

In this chapter, we summarize the major contributions of the thesis. Furthermore, we present some suggestions for future research work.

As outlined in chapter one, the aim contribution of this thesis are to: construct three component-based system, conduct an empirical study to investigate the effectiveness of complexity measures in CB integration testing (glue-code testing), and finally, analysis and discuss the results with a well established statistical techniques.

Three CB systems are developed based on the component-based software development that discusses in Chessman book [23]. These systems are Hotel Reservation System (HRS), Library Management System (LMS), and Smart Office System (SOS). These systems are built from scratch by using Eclipse Plug-in 6.5 to build systems' components as plug-ins and NetBeans 6.8 to develop systems interfaces and glue these plug-ins (components).

Three controlled experiments are conducted in an academic context to investigate our research question. These experiments take place at King Fahd University of Petroleum and Minerals (KFUPM) Saudi Arabia. In all experiments the subjects are students from different levels (Bachelor, Master, and Doctoral level) in Software Engineering department and Information and Computer Science department. Each experiment is last four hours. The session is ordered as two modules. In the first module, the first 45 minutes was utilized to present a brief presentation about testing coverage criteria concepts and CBS complexity concepts. The second module of the session is two exercises (one hour and half each) that form the experiment. There is a fifteen minutes break between the two exercises. In these experiments subjects test two systems Hotel Reservation System (HRS) and Library Management System (LMS) with systems description in addition to complexity measures of glue code operations.

The results of our controlled experiments that aim at assessing whether the adoption of structure complexity measures in component-based integration testing (glue-code testing) in order to priorities integration testing improves testing performance in terms of defect detection effectiveness and effort. From the collected data we draw several conclusions:

- There is a significant difference among the subjects who receiving complexity measures during testing and those who do not use complexity measures in terms of defect detection effectiveness (p-value for t-test and Mann-Whitney U-test are < 0.05 for both exercises).
- There is no a significant difference in the time spent on testing between the two groups (Mann-Whitney U-test p-value >0.05).

- There is no significant difference among experienced subjects with complexity measures and inexperienced subjects with complexity measures in terms of defect detection effectiveness (t-test p-value>0.05).
- Structure complexity measures assist subjects to detect more logical faults than omission faults.

We may generalize these results by saying that the adoption of structure complexity measures led to a significant better detecting of the faults during CB integration testing. The effort required for testing is substantially the same. Finally, subject experience doesn't have any effect on the defect detection effectiveness.

The results of this work could have different practical implications for software practitioners. Using structure complexity measures in CB integration testing (glue-code) helps in the detecting of faults, project managers could consider using it in the CB integration testing or in development process to decide where to focus their resources. Also it has a positive effect for component integrator (integration tester) and helps them to decide where to focus their integration testing efforts.

The limitations of our work are as follow:

- Small and median systems

  In these experiments we have investigated the effectiveness of structure complexity in small and median systems. So, there is a need to further investigate and analyze potential benefits of the structure complexity measures in the large system.

- Testing technique

  In our experiments we have used the control flow testing technique. Therefore, there is a need to investigate other testing techniques such as data flow testing and mutation testing.

- Component faults

  In literature interaction-related faults are categorized into three types namely: inter-component faults, interoperability faults, and traditional faults. In our work we have concentrated on the inter-component faults and the traditional faults. So, there is a need to further analyze the potential benefits of the complexity measures to detect these types of faults.

- False positives

Beside actual defects, subjects have detected false positives (faults reported by subjects as defects, when in fact no defect exists). Only the actual defects are evaluated in this research.

In our future work, we intend to replicate this experiment in order to support our findings. Furthermore, we are interested in repeating the present experiment in alternative contexts, including professional contexts involving real requirements/systems. Finally, using other testing techniques, such as data flow testing technique or mutation testing technique.

REFERENCES

1. S. Beydeda and V. Gruhn. Testing Commercial-off-the-Shelf Components and Systems. Springer, 2005.

2. H.-G. Gross. Component-Based Software Testing with UML. Springer, Heidelberg, 2005.

3. Gao, J. Z., H.-S. J. Tsao, and Y. Wu, Testing and Quality Assurance for Component- Based Software, Artech House, Norwood, MA, 2003.

4. Yuejian Li and Nancy J. Wahl. An Overview of Regression Testing. Software Engineering Notes Vol 24, no. 1, January 1999. pp 69-73.

5. Tanja Toroi, Testing Component-Based Systems. Towards Conformance Testing and Better Interoperability, PhD dissertation, 2009.

6. Y. Wu, D. Pan, and M. Chen, Testing Component-Based Software. International Conference Software & Systems Engineering and their Applications - ICSSEA '2002. December 2002.

7. Alexander P. Conrad, Robert S. Roos, and Gregory M. Kapfhammer. 2010. Empirically studying the role of selection operators during search-based test suite prioritization. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation* (GECCO '10). ACM, New York, NY, USA, 1373-1380.
8. Chin-Yu Huang, Jun-Ru Chang, Yung-Hsin Chang, Design and analysis of GUI test-case prioritization using weight-based methods, Journal of Systems and Software, Volume 83, Issue 4, April 2010, Pages 646-659

9. Lucas Lima, Juliano Iyoda, Augusto Sampaio, and Eduardo Aranha. 2009. Test case prioritization based on data reuse an experimental study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement* (ESEM '09). IEEE Computer Society, Washington, DC, USA, 279-290.

10. S. Mirarab and L. Tahvildari. A prioritization approach for software test cases on Bayesian Networks. In *Found. App. Softw. Eng., LNCS 4422-0276*, pages 276–290, Mar. 2007.
11. Cho Es, Kim MS, Kim SD. Component metrics to measure component quality. Proceeding of 8th Asia-Pacific Software Engineering Conference, Macao, China, 2001. IEEE Computer Society Press:Los Alamitos, CA, 2001; 419-426.

12. Narasimhan VL, Hendradjaya B.A new suite of metrics for the integration of software components. Technical Report (The 1st International workshop on Object Systems and Software Architecture), University of Adelaide,Australia, 2004.

13. Mahmood, S.; Lai, R.; , "A Complexity Measure for UML component based system specification,"; software – practice and experiences, vol. 38, page 117-134; 2008.

14. Fenton NE, Ohlsson N. Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software Engineering 2000; 26(8):797–814.

15. S. Sedigh Ali, A. Ghafoor and R.A. Paul. “Software Engineering Metrics for COTS based Systems”. IEEE Computer, 34(5): 44—50, May 2001.

16. A. Marcus, D. Poshyvanyk, and R. Ferenc, “Using the conceptual cohesion of classes for fault prediction in object-oriented systems,” IEEE Transactions on Software Engineering, vol. 34, no. 2, pp. 287–300, 2008.

17. Y. Zhou and H. Leung, “Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults,” IEEE Trans. Software Eng., vol. 32, no. 10, pp. 771-789, Oct. 2006.

18. R. Subramanyan and M.S. Krishnan, “Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects,” IEEE Trans. Software Eng., vol. 29, pp. 297-310, Apr. 2003.

19. H.M. Olague, L.H. Etzkorn, S. Gholston, and S. Quattlebaum, “Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development processes,” IEEE Trans. Software Eng., vol. 33, no. 6, pp. 402-419, June 2007.

20. Ping Yu; Systa, T.; Muller, H.; , "Predicting fault-proneness using OO metrics. An industrial case study," Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on , vol., no., pp.99-107, 2002

21. K. Goseva-Popstojanova, A. E. Hassan, A. Guedem, W. Abdelmoez, D. E. M. Nassar, H. H. Ammar, and A. Mili. Architectural-level risk analysis using UML. IEEE Transactions on Software Engineering, 29(10):946–960, 2003.

22. T. McCabe, “A Complexity Metrics,” IEEE Trans. Software Eng., vol. 2, no. 4, pp. 308-320, Dec. 1976.

23. J. Cheesman and J. Daniels, UML Components – A Simple Process for Specifying Component-Based Software, ser. Component Software Series. Addison-Wesley, 2001.

24. Barclay, K., Savage, J. *Object-Oriented Design with UML and Java*. Butterworth-Heinemann, Newton, MA, USA . 2003

25. Hilburn, T.B.; Towhidnejad, M.; Nangia, S.; Li Shen; , "A Case Study Project for Software Engineering Education," Frontiers in Education Conference, 36th Annual , vol., no., pp.1-5, 27-31 Oct. 2006.

26. P. Ammann and J. Offutt, Introduction to Software Testing, Cambridge University Press, Cambridge, UK, 2008. ISBN 0-52188-038-1.

27. S. A. Vilkomir and J. P. Bowen, “From MC/DC to RC/DC: formalization and analysis of control-flow testing criteria, ”Formal Aspects of Computing, vol. 18, no. 1, pp. 42–62, 2006.

28. Vilkomir SA, Kapoor K, Bowen JP (2003) Tolerance of control-flow testing criteria. In: Proceedings of 27th IEEE annual international computer software and applications conference (COMPSAC 2003), Dallas, Texas, USA, 3–6 November 2003. IEEE Computer Society Press, pp 182–187

29. Kapoor J, Bowen JP. Experimental evaluation of the tolerance for control-flow test criteria. Software Testing, Verification and Reliability 2004; 14 (3): 167-187.
30. H. Zhu. Axiomatic assessment of control flow-based software test adequacy criteria. Software Engineering Journal, pages 194–204, September 1995.

31. C. H. Liu, D. Kung, P. Hsia, and C. T. Hsu. Structural testing of Web applications. In 11th International Symposium on Software Reliability

Engineering, pages 84–96, San Jose, CA, October 2000. IEEE Computer Society Press.

32. Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. IEEE Transactions on Software Engineering, 19(8):774–787, August 1993.

33. Phyllis G. Frankl and Yuetang Deng. Comparison of delivered reliability of branch, data flow and operational testing: A case study. In 2000 International Symposium on Software Testing, and Analysis (ISSTA '00), pages 124–134, Portland, OR, August 2000. IEEE Computer Society Press.

34. Nayak, Narmada and Mohapatra, Durga Prasad, Automatic Test Data Generation for Data Flow Testing Using Particle Swarm Optimization,Communications in Computer and Information Science volume 95,pages 1-12, ISSN 978-3-642-14825-5,2010

35. Lynn M Foreman, Stuart H Zweben, A study of the effectiveness of control and data flow testing strategies, Journal of Systems and Software, Volume 21, Issue 3, Applying Specification, Verification, and Validation Techniques to Industrial Software Systems, June 1993, Pages 215-228, ISSN 0164-1212

36. Pacheco, C.; Lahiri, S.K.; Ernst, M.D.; Ball, T.; , "Feedback-Directed Random Test Generation," Software Engineering, 2007. ICSE 2007. 29th International Conference on , vol., no., pp.75-84, 20-26 May 2007.

37. T.Y. Chen, Y.T. Yu, "On the Relationship Between Partition and Random Testing," IEEE Transactions on Software Engineering, pp. 977-980, December, 1994

38. T.Y. Chen, F.-C. Kuo, R.G. Merkel, S.P. Ng, Mirror adaptive random testing, Information and Software Technology, Volume 46, Issue 15, Third International Conference on Quality Software: QSIC 2003, 1 December 2004, Pages 1001-1010, ISSN 0950-5849.
39. I. Crnkovic and M. Larsson, editors. Building Reliable Component-based Software Systems. Artech House Publishers, 2002.

40. P. Samuel and R. Mall. Boundary Value Testing based on UML Models. In ATS'05, pages 94–99. IEEE Computer Society, 2005.

41. Nikolai Kosmatov, Bruno Legeard, Fabien Peureux, and Mark Utting Boundary coverage criteria for test generation from formal models. In 15th International Symposium on Software Reliability Engineering (ISSRE'04), pages 139–150, Saint-Malo, France, November 2004. IEEE Computer Society.

42. Wang Yingxu, G. King, and H. Wickburg, "A method for built-in tests in component-based software maintenance," Proceedings of the 3rd European Conference on Software Maintenance and Reengineering, 1999, IEEE Computer Society, pp. 186-189.

43. J. Hornstein and H. Elder, "Test reuse in CBSE using built-in tests," Proceedings of the Workshop on Component-based Software Engineering, Composing systems from components, 2002.
44. S. Beydeda and V. Gruhn, "Merging components and testing tools: the self-testing COTS components (STECC) strategy," Proceedings of the 29th Euromicro Conference, 2003, IEEE Computer Society, pp. 107-114.

45. A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do, "Using component metacontent to support the regression testing of component-

based software," Proceedings of the International Conference on Software Maintenance, 2001, IEEE Computer Society, pp. 716-725.

46. Y. Wu, M. H. Chen, and J. Offutt, "UML-based integration testing for component-based software," 2003, Springer, pp. 251-60.

47. F. Belli and C. Budnik, "Towards Self-Testing of Component-Based Software," Proceedings of the 29th Annual International Computer Software and Applications Conference, 2005, IEEE Computer Society, pp. 205-210.

48. W. Councill, "Third-Party Testing and the Quality of Software Components," *IEEE Softw.*, vol. 16, 1999, pp. 55-57.

49. Y. Ma, S. Oh, D. Bae, and Y. Kwon, "Framework for Third Party Testing of Component Software," Proceedings of the 8th Asia-Pacific on Software Engineering Conference, 2001, IEEE Computer Society.

50. J. Gao, K. Gupta, S. Gupta, and S. Shim, "On Building Testable Software Components," Proceedings of the 1st International Conference on COTS-Based Software Systems, 2002, Springer-Verlag, pp. 108-121.

51. F. Jabeen and M. Rehman, "A framework for object oriented component testing," Proceedings of the IEEE Symposium on Emerging Technologies, 2005, IEEE Computer Society, pp. 451-460.
52. Piel, É. and Gonzalez-Sanchez, A. 2009. Data-flow integration testing adapted to runtime evolution in component-based systems. In Proceedings of the 2009 ESEC/FSE Workshop on Software integration and Evolution @ Runtime (Amsterdam, The Netherlands, August 25 - 25, 2009). SINTER '09. ACM, New York, NY, 3-10.

53. Filippo Ricca, Marco Torchiano, Massimiliano Di Penta, Mariano Ceccato, Paolo Tonella, Using acceptance tests as a support for clarifying requirements: A series of experiments, Information and Software Technology, Volume 51, Issue 2, February 2009, Pages 270-283

54. B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. ElEmam, J. Rosenberg, Preliminary guidelines for empirical research in software engineering, IEEE Transactions on Software Engineering 28 (8) (2002) 721–734.

55. M. Torchiano, F. Ricca, M. Di Penta, ‘‘Talking tests: a preliminary experimental study on Fit user acceptance tests, in: IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), IEEE CS Press, September 2007, pp. 464–466.

56. Erik Arisholm, Lionel C. Briand, Siw Elisabeth Hove, Yvan Labiche, "The Impact of UML Documentation on Software Maintenance: An Experimental Evaluation," IEEE Transactions on Software Engineering, pp. 365-381, June, 2006

57. Erik Arisholm and Dag I. K. Sjoberg. 2004. Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software. *IEEE Trans. Softw. Eng.* 30, 8 (August 2004), 521-534.

58. Sauer, C., Jeffery, D., Land, L., Yetton, P., The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research; IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 11-14, 2000.
59. Höst, M., Regnell, B., Wohlin, C., Using Students as Subjects – A Comparative Study of Students and Professionals in Lead-Time Impact Assessment, Empirical Software Engineering, vol. 5, pp. 201-214, 2000.

60. Tichy, W., Hints for Reviewing Empirical Work in Software Engineering, Empirical Software Engineering: An International Journal 5: 309-312, 2001.

61. Richard J. Larsen and Morris L. Marx. An Introduction to Mathematical Statistics and Its Applications (3rd Edition). Prentice Hall, 3rd edition, January 2000.

APPENDIX A

Reservation Hotel System

The Hotel Reservation System (HRS) is a software application to help a hotel management to manage hotel room's reservation in hotel chain.

A.1 Requirement

In this section we will give a high-level system description and we will describe the business process of reserving a hotel room in hotel chain. The business process description introduces a number of terms, such *as reservation, room, customer*, which we need to get clear about hence a business concept model will be built to link these terms and other key terms to create a common vocabulary among the business people. Finally, a use cases will be described in details to build use cases model.

A.1.1 System Envisioning

To capture system requirements we will give a high-level system envisioning statement for the hotel reservation system:

Hotel reservation system (HRS) allows guests to made reservations in any hotel in the chain. It helps them to looking for room in different hotels in certain date; and it allows *them to confirm or cancelling their reservation. In addition it allows hotel chain management to do various operations such as add new hotel, add rooms to the hotel and all operations that manipulate hotels and rooms data. Guest can only reserve one room per each reservation and each reservation belongs to one hotel. When guest looking for room he must enter check-in date and check-out date, check-in date must be lower than check-out date and the difference between them is one day.*

A.1.2 Reservation System Business Process

The business process of reserving a hotel room is shown in Figure A.1. The reservation process is started by an enquiry from a guest, who states has requirements. System checks room availability and if a room is available the guest makes a reservation. Details of the reservation are sent to the guest by email. Then one of four things can occur: the guest might arrive and take up has reservation; he might cancel it, he might modified some details of it, which required another confirmation; or he might just not show up, but he is going to get a bill anyway.

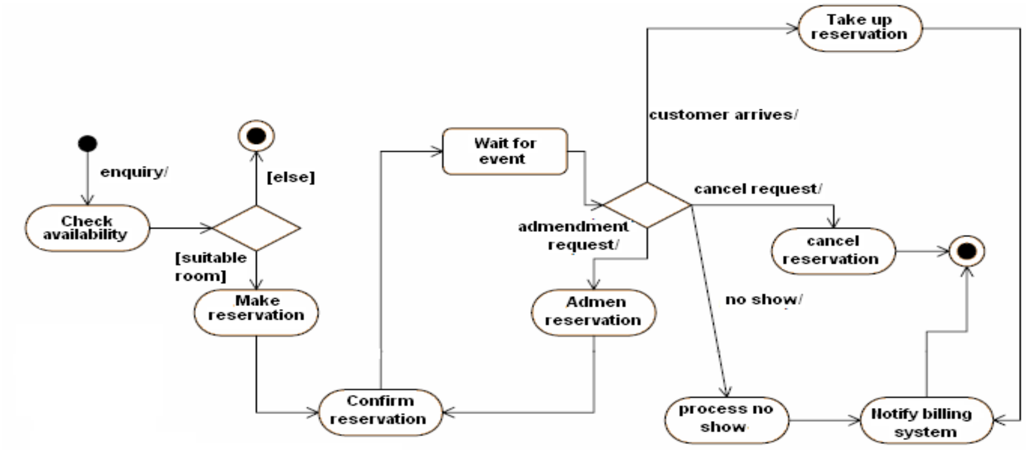


Figure A.1 Business process for hotel reservation system

A.1.3 Business Concept Model

A number of terms were introduced by the business process description, such *as reservation, room, guest*, which we want to get clear about. Consequently, business concept model will be constructed to relate these terms and other important terms to create a common vocabulary among people involved in the work. For example, if *guest* means four different things in the business, we must to get this cleared up as early as possible so that everyone is working to the same set of terms with agreed meaning. Figure A.2 shows a possible concept model for the reservation system.

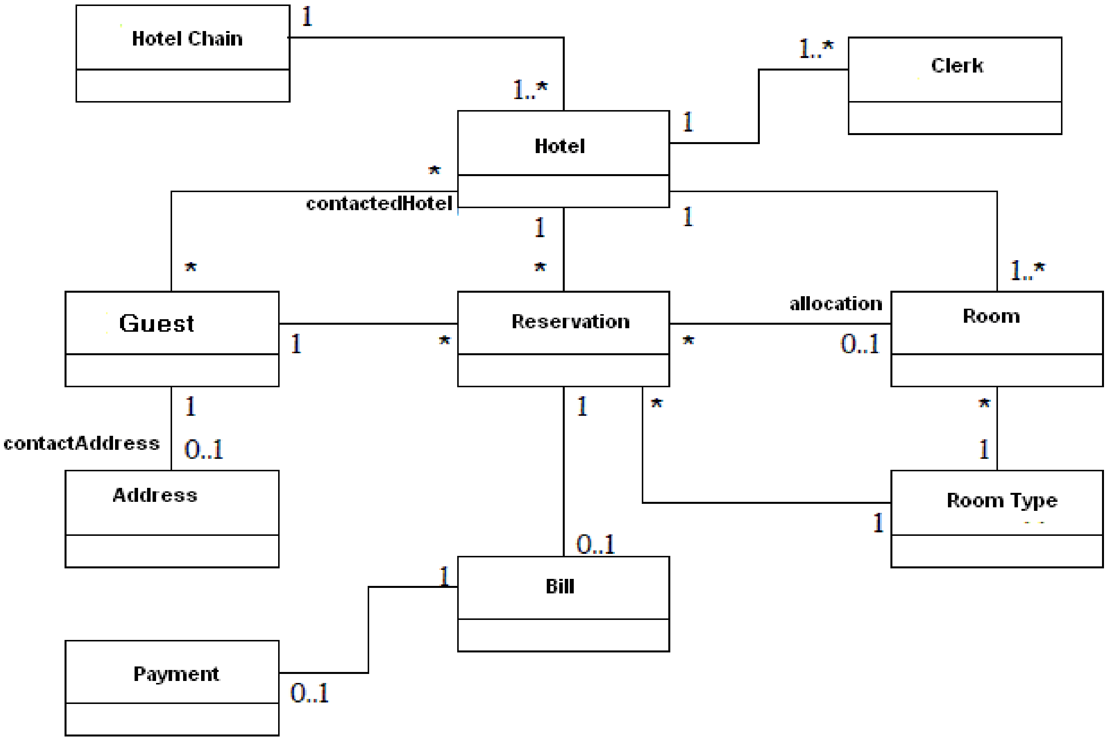


Figure A.2. Business concept model for hotel reservation system

A.1.4 Uses Cases

The expected functionalities of the hotel reservation system are presented in Figure A.3. We will describe one use cases which make a reservation. The description will show how the initiator will interact with the system.

Name: Make a Reservation

Initiator: Reservation administrator

Goal: Reserve room(s) at a hotel

Main success scenario

1. Reservation administrator asks to make a reservation
2. Reservation administrator selects a hotel, dates and room type
3. System provides available rooms and the price to the reservation maker
4. Reservation administrator asks for reservation
5. Reservation administrator provides name and postcode (zip code)
6. Reservation administrator provides contact email address
7. System makes reservation and allocates tag to reservation
8. System reveals tag to reservation maker
9. System creates and sends confirmation by email

Extensions

3. Room not available
    a) System offers alternatives
    b) Reservation maker selects from alternatives

3b) Reservation administrator rejects alternatives

a) Fail

4. Reservation administrator declines offer

a) Fail

6. Customer already on file (based on name and postcode)

a) Resume 7

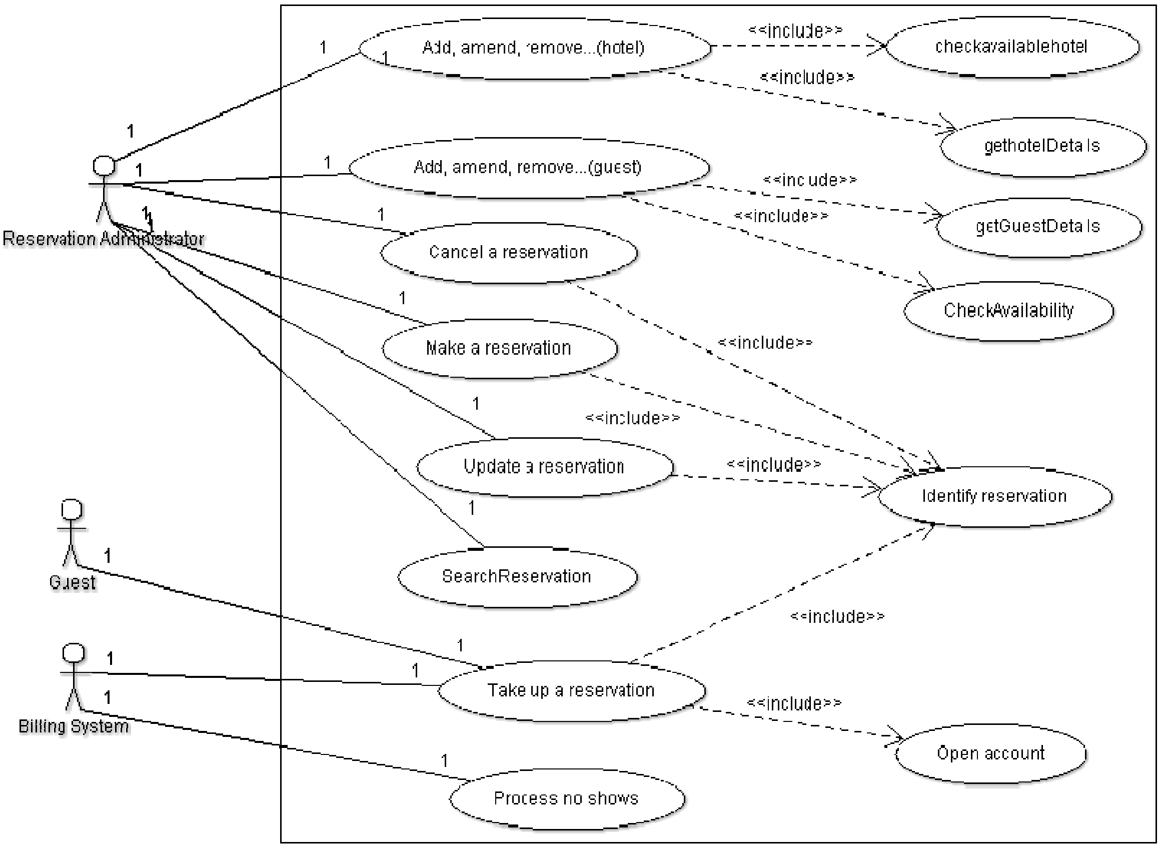


Figure A.3. Use case model

A.2 Specification

Component specification includes three stages namely, component identification, component interaction, and component specification. It takes as inputs the business concept model and the use case diagram from the requirement phase and the important outputs of this phase are components specifications, components architectures, and components interfaces.

A.2.1 Component Identification

The purpose of this stage is to create an initial set of interfaces and component specifications, linked together into initial component architecture. It also generates an important internal specification, the business type model, which is used later to create interface information models.

A.2.1.1 Identify System Interfaces and Operations

To identify system interfaces and operations from the use case model; we create one dialog type and one system interface for each use case as shown in Figure A.4. We then go through each of the use cases and for each step consider whether or not there are system responsibility that must be modeled. If we find a system responsibility needed to model, we represent them as one or more operations of the suitable system interface. An initial set of interfaces and operations are extracted and this gives us a good starting point to work from.

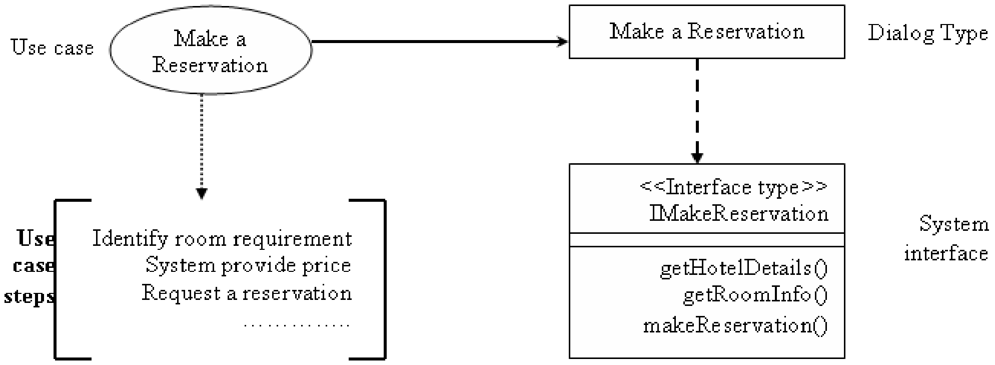


Figure A.4 Use case maps to system interface

Let's apply this on the *make a reservation* use case.

- **Make a Reservation**

In the first step, we create an initial system interface called *IMakeReservation*. The second step, we go through each step in the main success scenario, in step 2 we observe that the reservation system must permit the guest to make a reservation. In step 3 system returns the hotels details and the available rooms and the price. We call these the *getHotelDetails*() and *getRoomInfo*() operations. In step 7, we can figure out the need for an operation to create a reservation given various details and returns the reservation's reference to guest; we call this operation *makeReservation*().

Alternative behaviors under certain situations are described in the use case extensions part. In step 3, if the room not available, we can observe that the guest may select another room's types or dates; the selection of information will be handled by the user dialog logic. The interface defined so far is illustrated in Figure A.5.

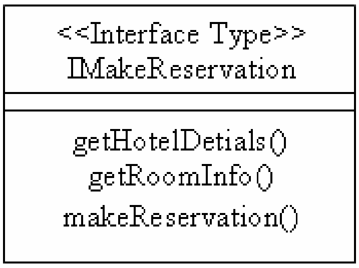


Figure 3.5 Initial system interfaces and their initial operations

At the end of this stage we end up with an initials system interfaces and a list of an initial operations.

A.2.1.2 Identify Business Interfaces

To identify the business interfaces we convert the business concept model to business type model. The business type model identifies the state/data that the enterprise requires to keep and monitor. It is also considered the main source for the business interfaces and the raw material for the development of interface information models as we can see later. After refining the business concept model, adding or removing elements until its scope is correct as you can see in Figure A.6. We can get the business type model that is shown in Figure A.7. After extracting the business type model, we can decide which types in the business type model can be considered core. The purpose of identifying core types is to start thinking about which information is dependent on which other information, and which information can stand alone. A *core* type is characterized by the following:

- A business identifier usually independent of other identifiers
- Independent existence, no mandatory associations, except to a categorizing type.

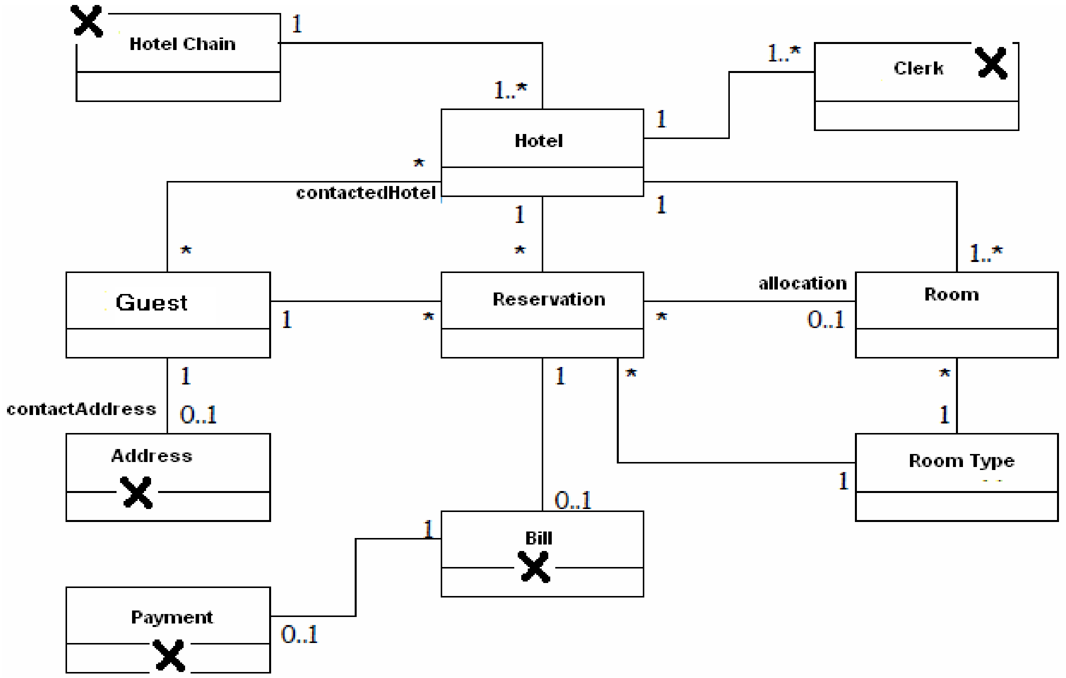


Figure A.6 refining the business concept model

By applying these rules we decide that *Hotel* and *Guest* are the core types. And all other types provide details of the core types.

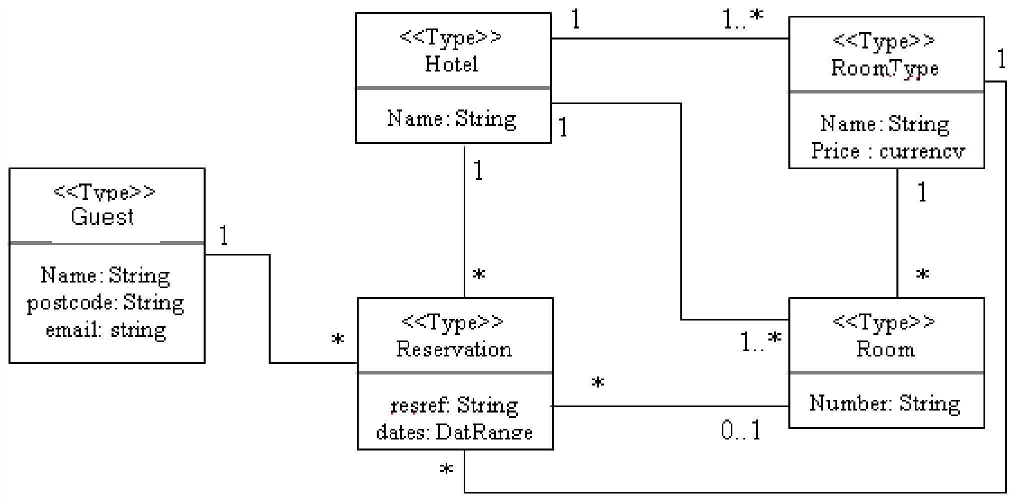


Figure A.7 Initial business type

International Function Point User Group (IFPUG) classifies interfaces into two types' internal logical file (ILF) and external logical file (ELF). Each interface includes a function that its execution affects other functions, this interface are classified as ILF. Other interfaces are classified as ELF. Therefore, for each core type we make two interfaces one ILF and one ELF. For example, the core type *Hotel* will be assigned two interfaces, *IHotelMgt* and *IADUHotel*, one to keep operations that effects other operations and the other interface to keep the other type of operations. After that we defined the core types we create two business interfaces for each core type in the business type model as shown in Figure A.8.

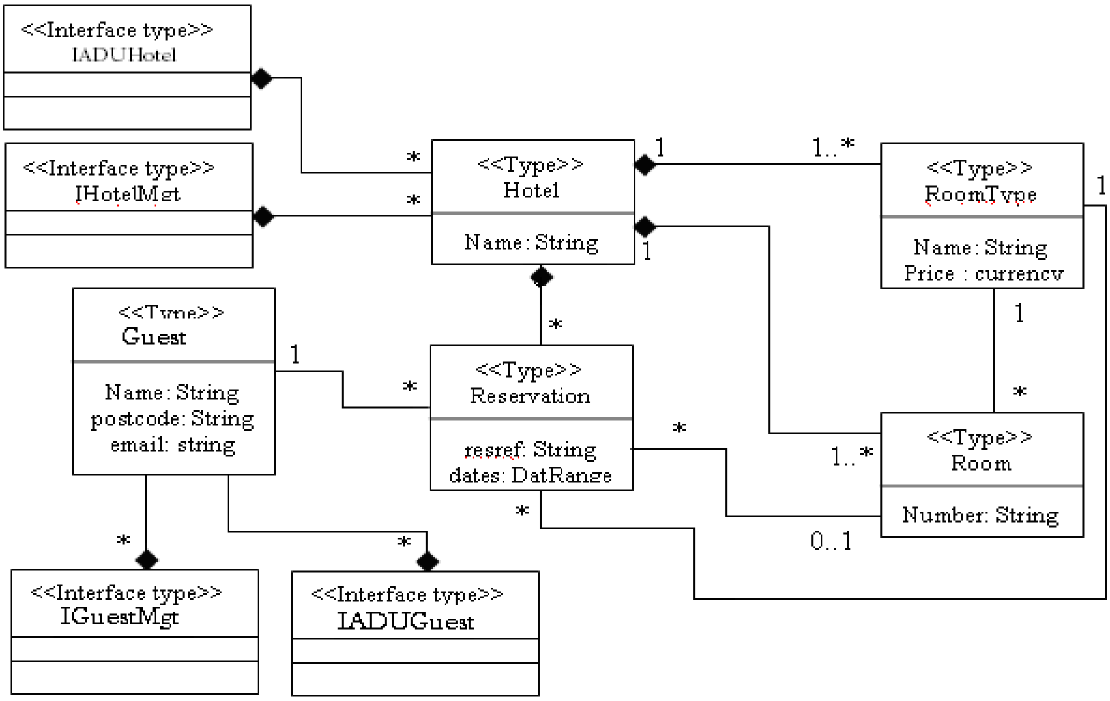


Figure A.8 Interface responsibility diagram of the business type model

A.2.1.3 Existing Interface and Systems

So far, we have extracted the initial system and business interfaces. In this step, any additional interfaces our system will be needed in the new environment must be added. In this case, we have existing billing software with a designated interface. This software has been in production for a number of years and we want to use its functionality. Therefore we include this interface to our set of system interfaces.

A.2.1.4 Component Specification Architecture

In this step an initial set of component specifications will be created and form an idea of how they might fit together.

A.2.1.5 System Component Specifications

In this system, we set IMakeReservation and ITakeUpReservation interfaces on one component specification, and we remain IBilling interface in another component specification. The reservation system uses IBilling interface, therefore we include the dependency between them. We also include in interface dependencies on IGuestMgt and IHotelMgt, although we don't know if these really exist at this stage. Figure A.9 shows the system component specifications.

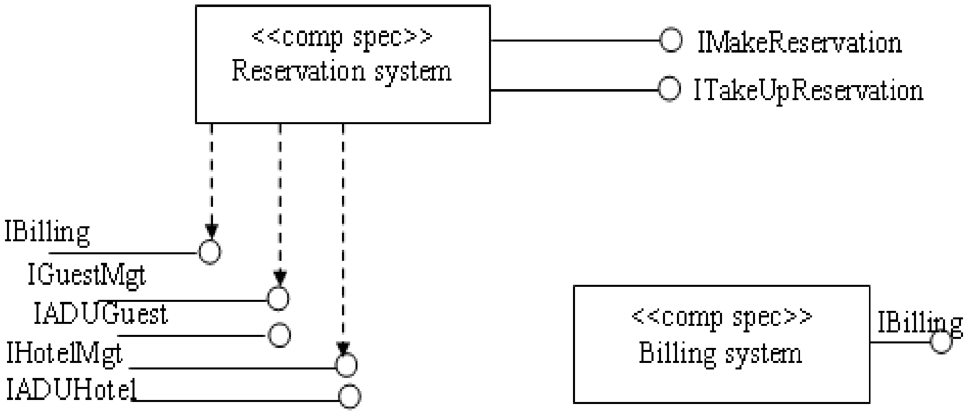


Figure A.9. System component specifications

A.2.1.4.2 Business Component Specifications

In Section 3.1.2.1.2, we determine our core types which are Hotel and Guest type and we assigned a business interfaces for *IGuestMgt*, *IADUGuestr, IHotelMgt,and IADUHotel* interfaces. So, for each core type a separate component specifications is created as shown in Figure A.10.

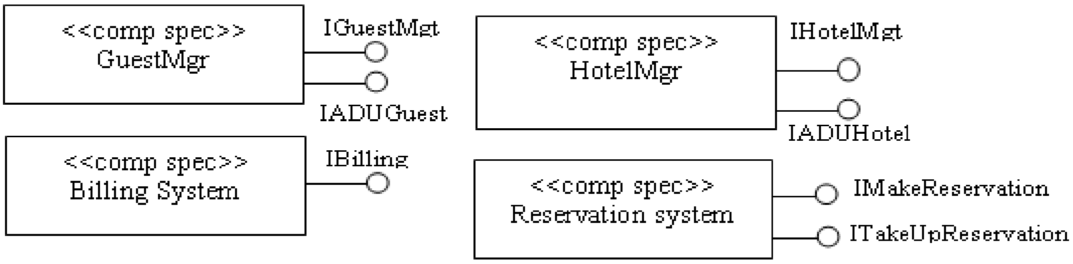


Figure A.10. Business component specification

A.2.1.6 An initial Architecture

So far we have an initial set of specification, including their supported interfaces and their interface dependences. Since we don't have any interfaces being offered by more than one component specification in this case study, we can bind the interface dependencies of the component specification directly onto their corresponding component specification interfaces, giving us the component specification architecture shown in Figure A.11.

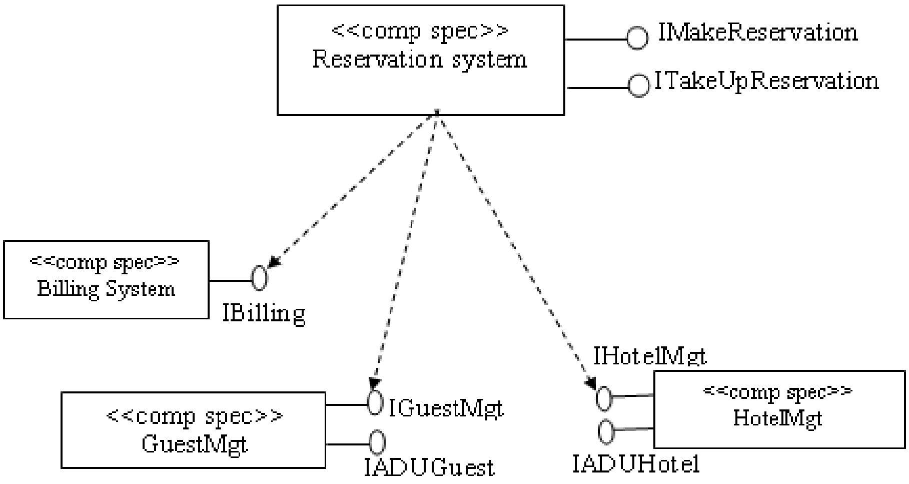


Figure A.11. Initial component specification architecture

A.2.2 Component Interaction

In component interaction, we will decide how the components will interact with each other to do the required functionality. The existing interface definitions will be refined, to define how interfaces will be used, and to determine new interfaces and operations. UML collaboration diagrams will be used to model the components interactions. Component architecture shown in Figure A.11 and the system interfaces extracted in Section A.2.1.1 are used to model the interaction among components. In this step, we will discover operations of the business interfaces by drawing one or more collaboration diagrams for each operation in the system interfaces. Let’s go through this procedure for some of the system interface operations.

- **ItakeUpReservation**
  - getReservation ( )
  - beginStay ()
- **getReservation ( )**

The purpose of *getReservation ( )* operation is to return a reservation record, so we need to pass in the reservation reference and based on that reference the system returns the reservation details back. In this operation we need to define a new data structure to store the reservation details. We call this structure *ReservationDetails* which contains the fields shown in Figure A.12. The signature of this operation becomes

ItakeUpReservation:: getReservation (in resRef: String, out rd: ReservationDetails)

At runtime, this operation is called by the dialog layer on a reservation system component object. That object is not able to fulfill the operation itself since system component doesn't store business data, so it must use a component object offering the *IHotelMgt* interface. The required interaction for getReservation is shown in Figure A.13.

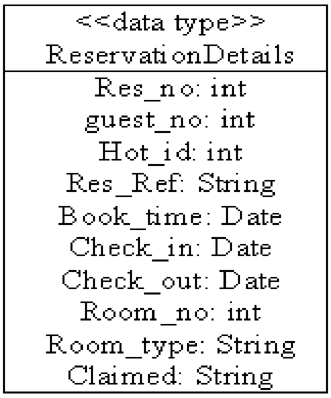


Figure A.12 Structured data type for reservation details

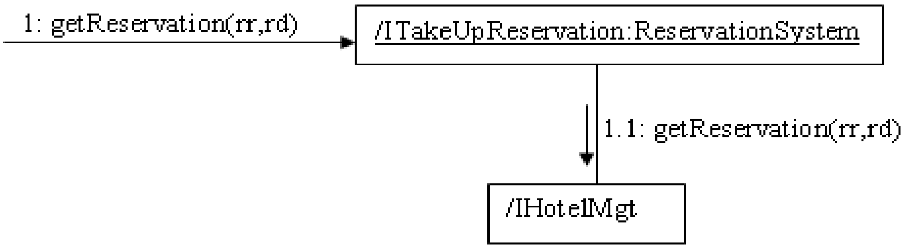

1: getReservation(rr,rd)
/ITakeUpReservation:ReservationSystem
1.1: getReservation(rr,rd)
/IHotelMgt

Figure A.13 getReservation interaction

As shown in the Figure A.13, we decide that *ItakeUpReservation* should also have a *getReservation o*peration, with the same signature. So we should now update the definition of *ItakeUpReservation* to add this operation as shown in Figure A.15.

- **beginStay**

The purpose of the *beginStay ( )* operation is inform the system when the guest is show up, change the reservation state to claimed and open new account for that guest. So we need to pass in the reservation reference and it returns back the room number that allocated to this guest. The signature for this operation becomes

ItakeUpReservation:: beginStay (in resRef: String, out roomNumber: int)

At runtime, this operation is invoked by the dialog layer on a reservation system component object. That object is not able to fulfill the operation itself because system component doesn't keep business data, so it must use component objects offering the *IHotelMgt* interface, *IGuestrMgt* interface, and *IBilling* interface. The required interaction for this operation is shown in Figure A.14.

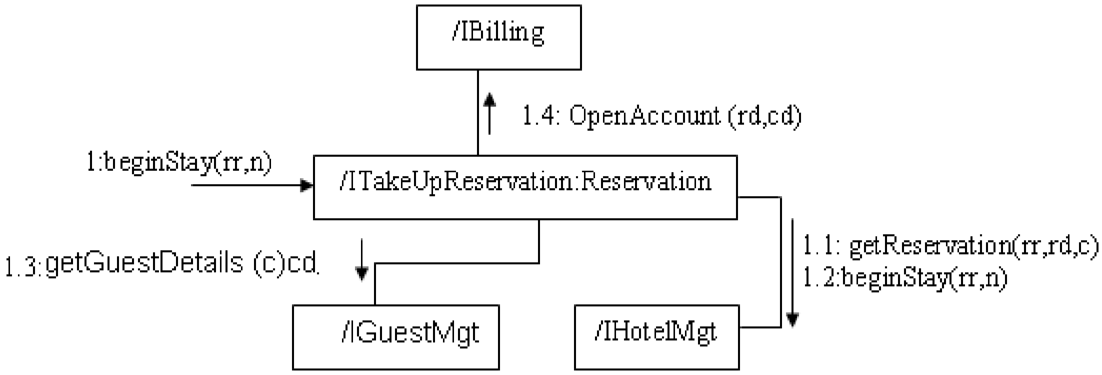
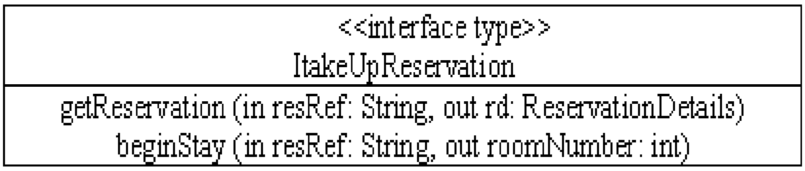


Figure A.14 Interaction for beginStay operation

As we can see in the Figure A.14, we decided that the *ItakeUpReservation* interface should also have a *beginStay* operation, with the same signature. So w*e should now update the definition of ItakeUpReservation to add this operation.*

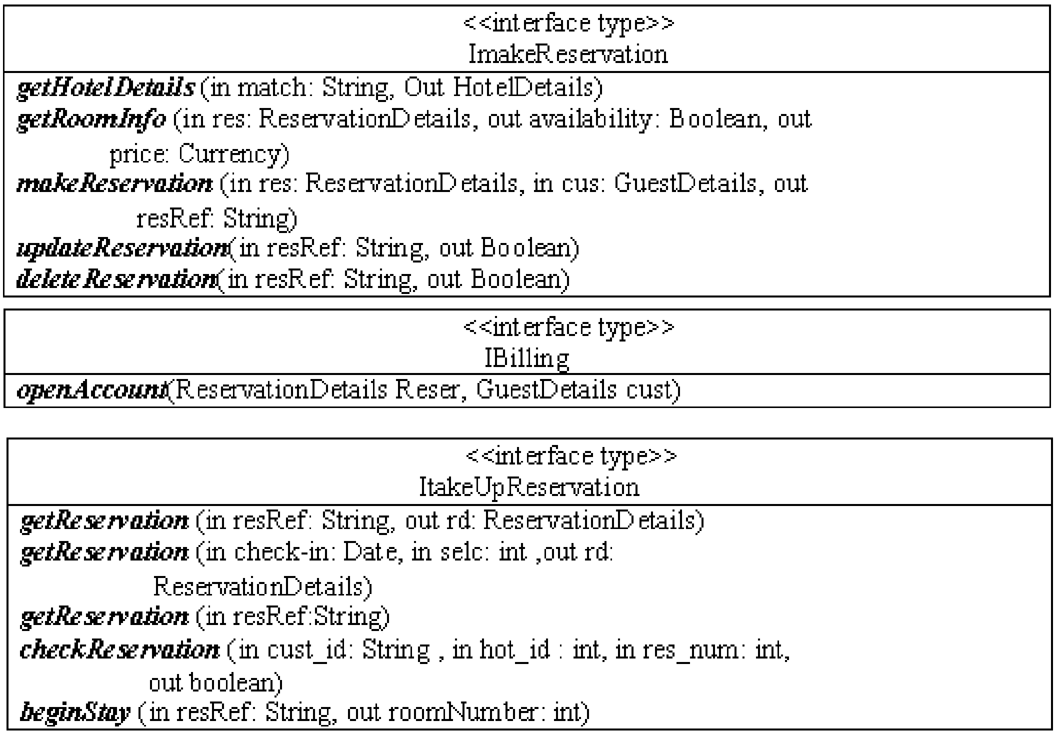


<<interface type>>
ItakeUpReservation

getReservation (in resRef: String, out rd: ReservationDetails)
beginStay (in resRef: String, out roomNumber: int)

Figure A.15 ItakeUpReservation with operation signatures

At the end of component interaction stage, we end up with a list of business interfaces and system interfaces with its operations signatures as we can see in Figure A.16 and A.17.



<<interface type>>
ImakeReservation

getHotelDetails (in match: String, Out HotelDetails)
getRoomInfo (in res: ReservationDetails, out availability: Boolean, out price: Currency)
makeReservation (in res: ReservationDetails, in cus: GuestDetails, out resRef: String)
updateReservation(in resRef: String, out Boolean)
deleteReservation(in resRef: String, out Boolean)

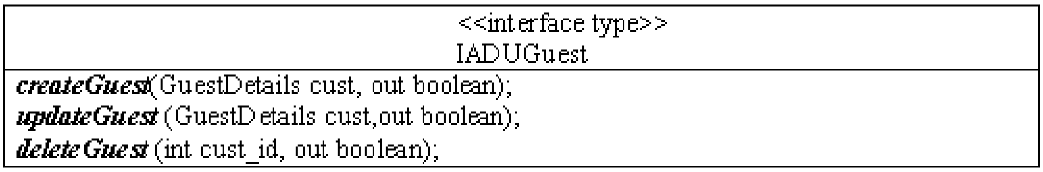
<<interface type>>
IBilling

openAccount(ReservationDetails Reser, GuestDetails cust)

<<interface type>>
ItakeUpReservation

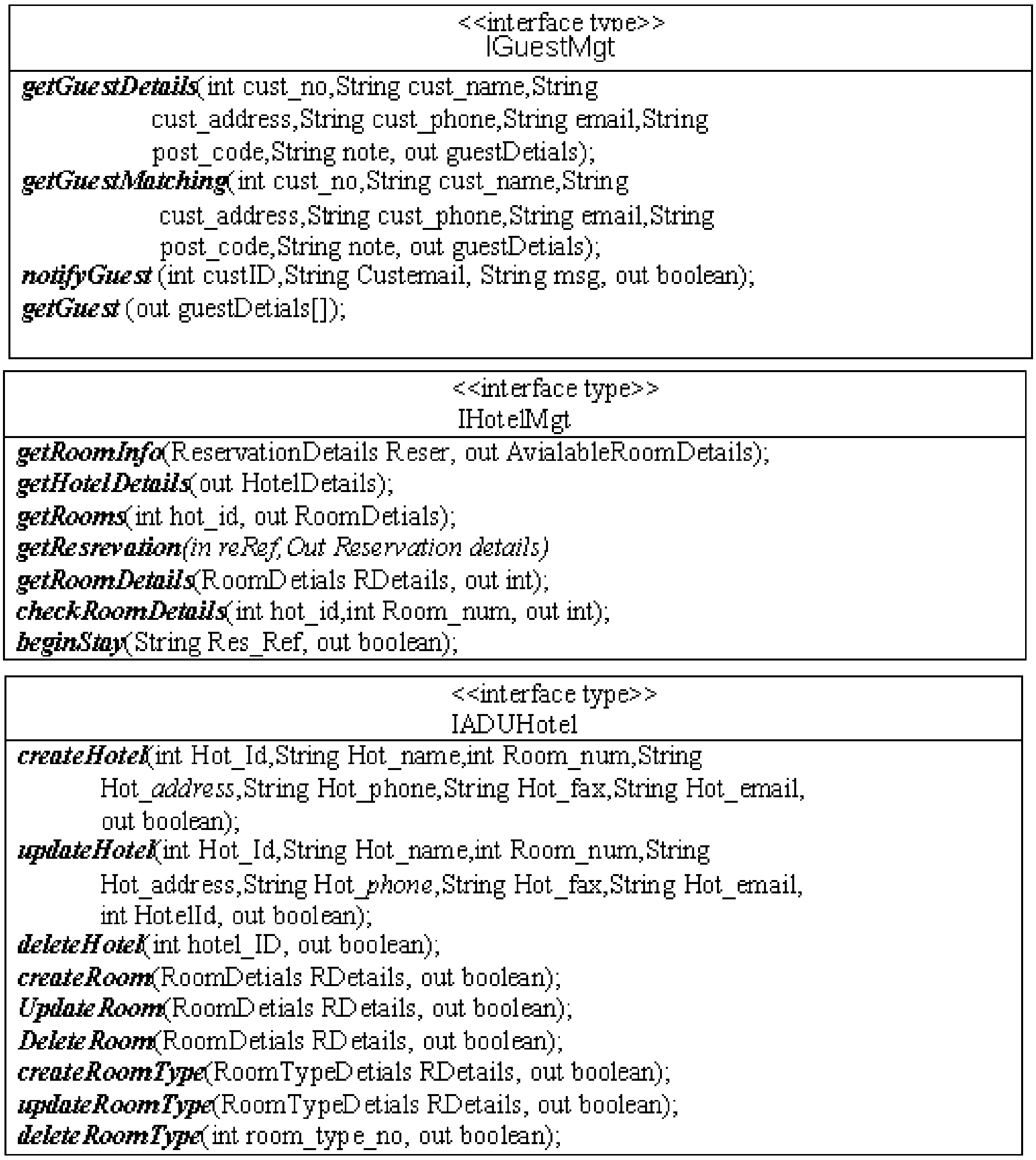
getReservation (in resRef: String, out rd: ReservationDetails)
getReservation (in check-in: Date, in selc: int ,out rd: ReservationDetails)
getReservation (in resRef:String)
checkReservation (in cust_id: String , in hot_id : int, in res_num: int, out boolean)
beginStay (in resRef: String, out roomNumber: int)

Figure A.16 System interfaces with operation signature



<<interface type>>
IADUGuest

createGuest(GuestDetails cust, out boolean);
updateGuest (GuestDetails cust,out boolean);
deleteGuest (int cust_id, out boolean);



<<interface type>>
IGuestMgt

getGuestDetails(int cust_no,String cust_name,String
cust_address,String cust_phone,String email,String
post_code,String note, out guestDetials);
getGuestMatching(int cust_no,String cust_name,String
cust_address,String cust_phone,String email,String
post_code,String note, out guestDetials);
notifyGuest (int custID,String Custemail, String msg, out boolean);
getGuest (out guestDetials[]);

<<interface type>>
IHotelMgt

getRoomInfo(ReservationDetails Reser, out AvialableRoomDetails);
getHotelDetails(out HotelDetails);
getRooms(int hot_id, out RoomDetials);
getResrevation*(in reRef, Out Reservation details)*
getRoomDetails(RoomDetials RDetails, out int);
checkRoomDetails(int hot_id,int Room_num, out int);
beginStay(String Res_Ref, out boolean);

<<interface type>>
IADUHotel

createHotel(int Hot_Id,String Hot_name,int Room_num,String
Hot_*address*,String Hot_phone,String Hot_fax,String Hot_email,
out boolean);
updateHotel(int Hot_Id,String Hot_name,int Room_num,String
Hot_address,String Hot_*phone*,String Hot_fax,String Hot_email,
int HotelId, out boolean);
deleteHotel(int hotel_ID, out boolean);
createRoom(RoomDetials RDetails, out boolean);
UpdateRoom(RoomDetials RDetails, out boolean);
DeleteRoom(RoomDetials RDetails, out boolean);
createRoomType(RoomTypeDetials RDetails, out boolean);
updateRoomType(RoomTypeDetials RDetails, out boolean);
deleteRoomType(int room_type_no, out boolean);

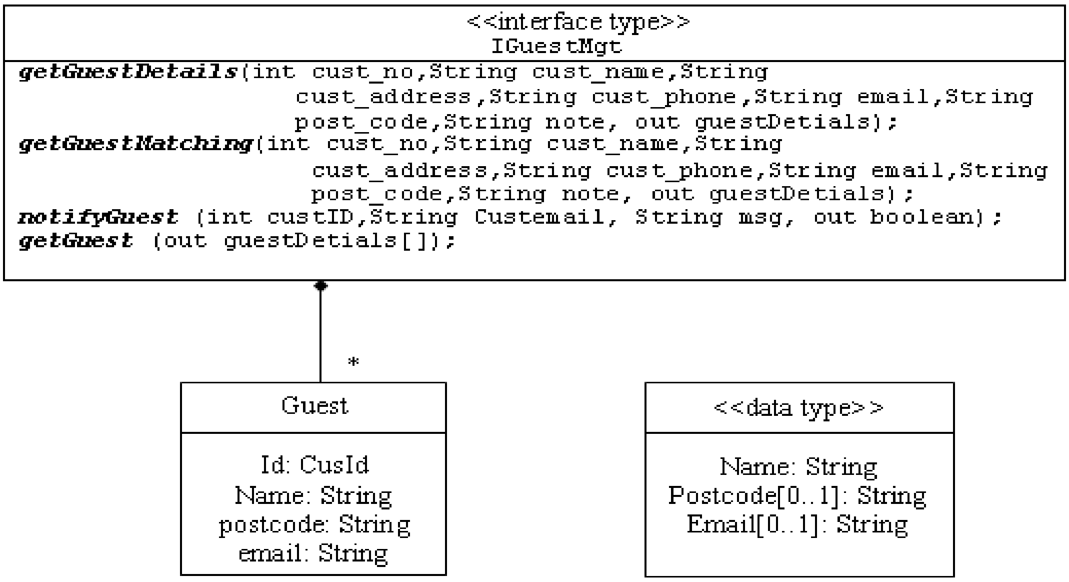
Figure A.17 Business interfaces with operation signature

A.2.3 Component Specification

In components specification we specify all interfaces supported by components or the interfaces that depends on. In this stage, we want to represent the state of the component object on which the interface depends. Since each interface has an interface information model, any changes to the state of the component object can be described in terms of this information model.

A.2.3.1 Define Interface Information Model

We use the business type model defined in Section A..2.1.2 to extract the interface information models for each interface. We decided that *IHotelMgt* interface contains operations to manage hotels data and *IGuestMgt* interface contains operations to manage guests. We Also decided that the interfaces *IMakeReservation* and *ITakeUpReservation* are responsible for managing the relationship between the guest and the hotel (reservation). Figure A.18 shows the interface information model for IGuestMgt interface that contains a *Guest* type.



```
getGuestDetails(int cust_no,String cust_name,String
                   cust_address,String cust_phone,String email,String
                   post_code,String note, out guestDetials);
getGuestMatching(int cust_no,String cust_name,String
                   cust_address,String cust_phone,String email,String
                   post_code,String note, out guestDetials);
notifyGuest (int custID,String Custemail, String msg, out boolean);
getGuest (out guestDetials[]);
```

Figure A.18 Interface specification diagram for the IGuestMgt interface

Whereas the interface information models for *IHotelMgt*, *IADUHotel*, and *IADUGuest* interfaces are shown in Figure A.19, Figure A.20, and Figure A.21.

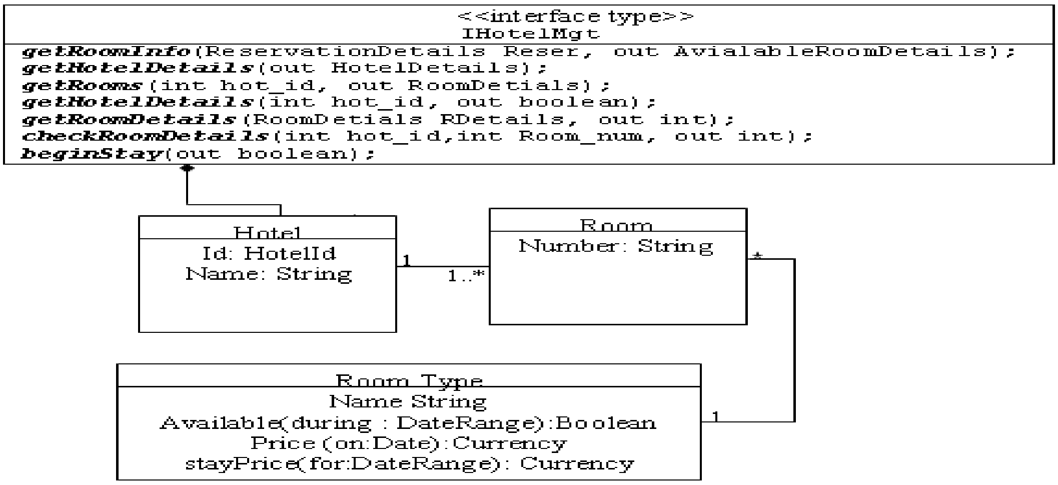

<<interface type>>
IHotelMgt
getRoomInfo(ReservationDetails Reser, out AvialableRoomDetails);
getHotelDetails(out HotelDetails);
getRooms(int hot_id, out RoomDetials);
getHotelDetails(int hot_id, out boolean);
getRoomDetails(RoomDetials RDetails, out int);
checkRoomDetails(int hot_id,int Room_num, out int);
beginStay(out boolean);
Hotel
Id: HotelId
Name: String
1
1..*
Room
Number: String
*
Room Type
Name String
Available(during : DateRange):Boolean
Price (on:Date):Currency
stayPrice(for:DateRange): Currency
1

Figure A.19 Interface specification diagram for IHotelMgt

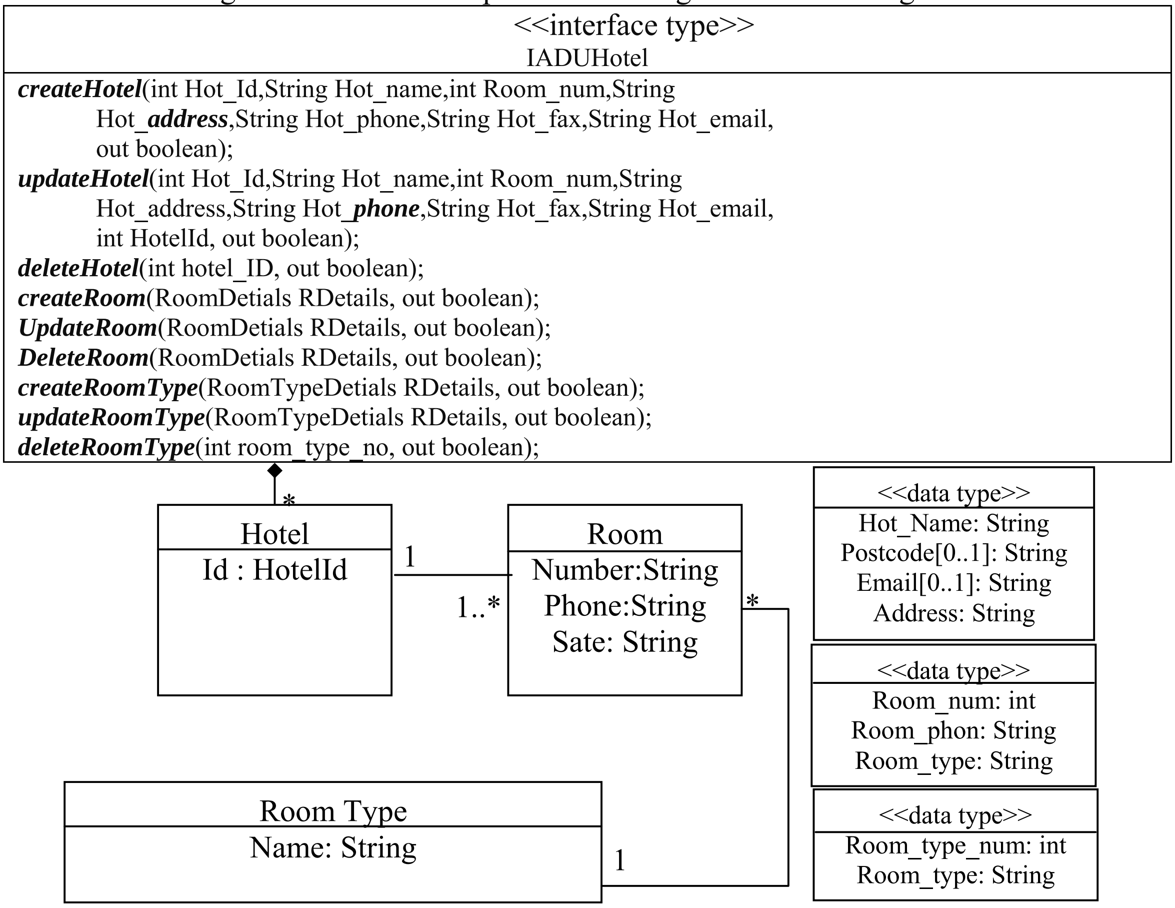

<<interface type>>
IADUHotel
createHotel(int Hot_Id,String Hot_name,int Room_num,String Hot_address,String Hot_phone,String Hot_fax,String Hot_email, out boolean);
updateHotel(int Hot_Id,String Hot_name,int Room_num,String Hot_address,String Hot_phone,String Hot_fax,String Hot_email, int HotelId, out boolean);
deleteHotel(int hotel_ID, out boolean);
createRoom(RoomDetials RDetails, out boolean);
UpdateRoom(RoomDetials RDetails, out boolean);
DeleteRoom(RoomDetials RDetails, out boolean);
createRoomType(RoomTypeDetials RDetails, out boolean);
updateRoomType(RoomTypeDetials RDetails, out boolean);
deleteRoomType(int room_type_no, out boolean);
*
Hotel
Id : HotelId
1
1..*
Room
Number:String
Phone:String
Sate: String
*
Room Type
Name: String
1
<<data type>>
Hot_Name: String
Postcode[0..1]: String
Email[0..1]: String
Address: String
<<data type>>
Room_num: int
Room_phon: String
Room_type: String
<<data type>>
Room_type_num: int
Room_type: String

Figure A.20 Interface specification diagram for IADUHotel

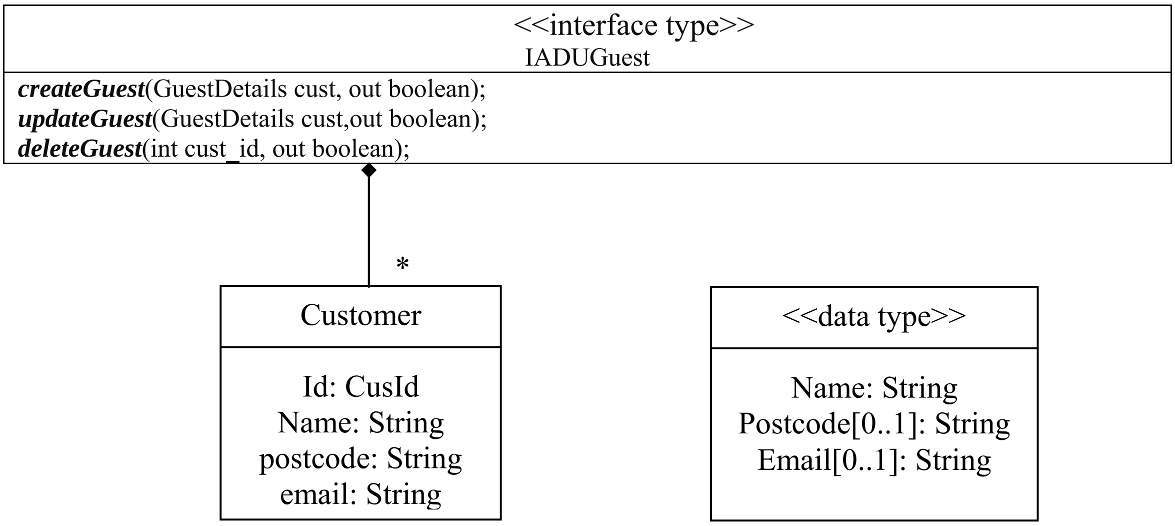


Figure A.21 Interface specification diagram for IADUGuest

A.2.3.2 Pre and Post-Conditions

Each operation has a pre- and post-conditions. These conditions identify the impact of the operation without describing an algorithm or implementation. Pre-condition is not the condition under which the operation will be called. Invocation of the operation is completely independent of the value of this condition. The post-condition specifies what are the effect of the operation. As a simple example, let's consider an operation to update a guest record.

```
▪ Context IADUGuestr:: updateGuest(in newGuestDetails, out state bolean)
     Pre:
     -- newGuestDetails is a valid Guest record
          Guest-> exist ( G | G.cust_id = GuestDetails.Cust_id)
     Post:
     -- Guest record is updated
     Guest->exists (G| G.cust_id = newGuestDetails.CUST_id,
       C.name     = newGuestDetails.Cust_name,
       C.Address = newGuestDetails.Cust_add,
```

C.Postcode= newGuestDetails.Cust_PostCode,
C.phone= newGuestDetails.Cust_phone)

- *Context IGuestMgt:: getGuestDetails(in cust_no,in cust_name,in cust_address,in cust_phone,in email,String post_code,in note, out GuestDetails);*

  Pre:
  -- Cust_no and post_code are a valid guest id and post_code
  -- Guest->exists (G|Custid= cust_no and G.Post_code= post_code))

  Post:
  -- the details returned match the details of the guest
  -- whose id is Cust_no
  -- find the guest
  Let theCust=Guest-> select(G|GC.cust_id= cust_id and
  G.post_code= post_code) in
  --Specified the result
  Result.cust_id=theCust.cust_id
  Result.cust_name=theCust.cust_name
  Result.cust_address=theCust.cust_address
  Result.cust_phone=theCust.cust_phone
  Result.email=theCust.email
  Result.post_code=theCust.post_code
  Result.note =theCust.note

A.2.3.3 Specifying System Interfaces

In this section we specify the system interfaces. In various cases the system interface operations redirect an invocation to the suitable business interface. We make a copy of everything in the business type model. In business interface, where the interface responsibility diagram gives a clear indication of which types are needed by an interface and which are not, it might not be obvious which information types you need until you specified the operations. As you can see in the Figure A.22 the information model for *IMakeReservation* doesn't require the room number attribute, so that has been removed.

On the other hand, the information model for *ITakeUpReservation* doesn't required the hotel name or the available attribute of room type as you can see in Figure A.23.

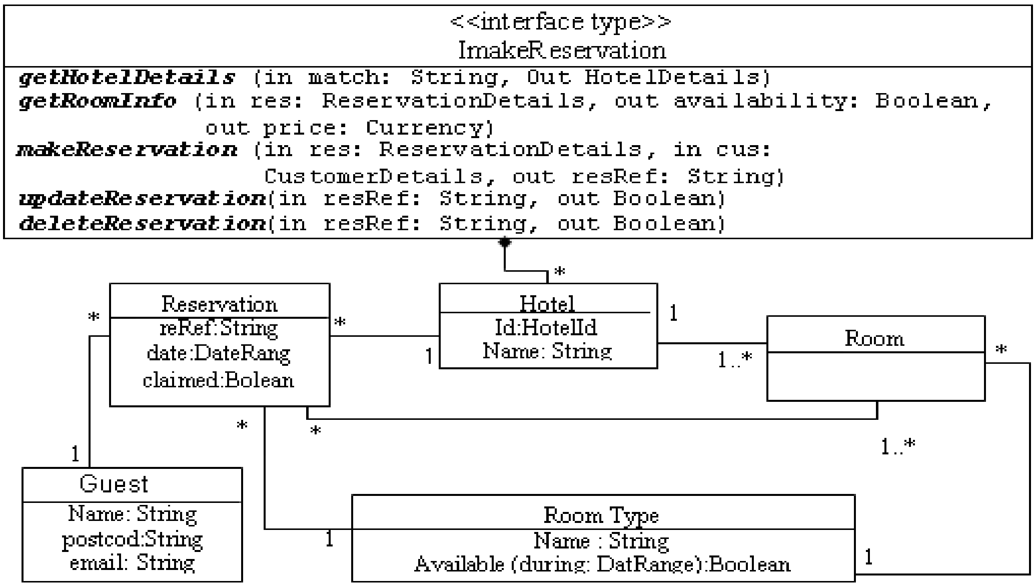

<<interface type>>
ImakeReservation
getHotelDetails (in match: String, Out HotelDetails)
getRoomInfo (in res: ReservationDetails, out availability: Boolean, out price: Currency)
makeReservation (in res: ReservationDetails, in cus: CustomerDetails, out resRef: String)
updateReservation(in resRef: String, out Boolean)
deleteReservation(in resRef: String, out Boolean)
*
Reservation
reRef:String
date:DateRang
claimed:Bolean
*
*
*
*
1
Hotel
Id:HotelId
Name: String
1
1
1..*
Room
*
1..*
1
Guest
Name: String
postcod:String
email: String
1
Room Type
Name : String
Available (during: DatRange):Boolean
1

Figure A.22 Interface specification diagram for ImakeReservation

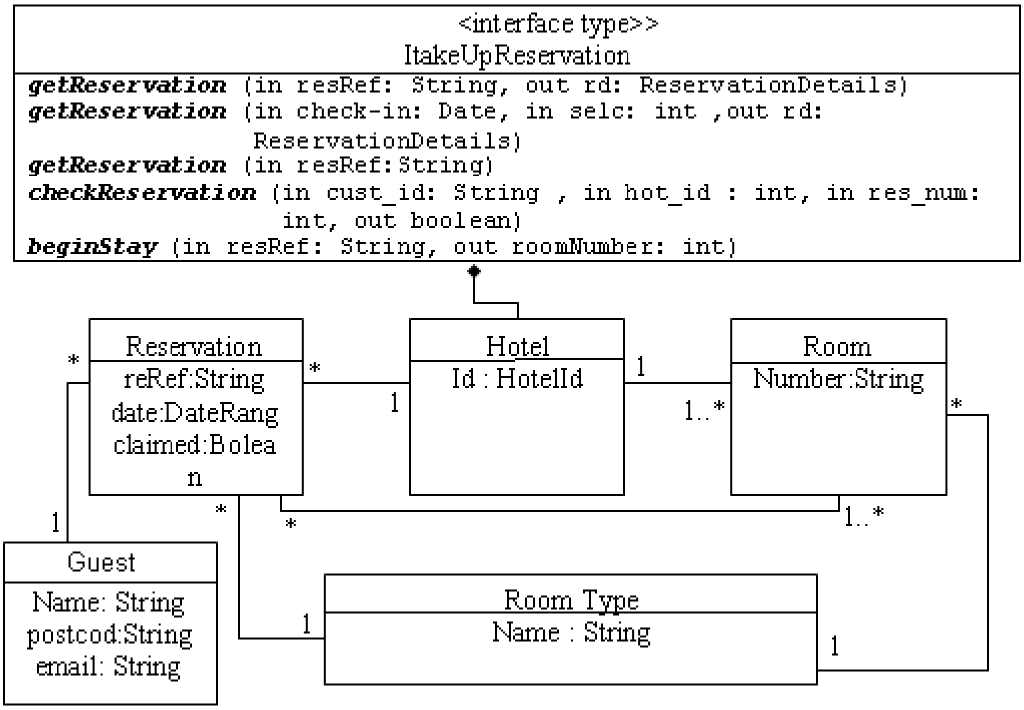

<interface type>>
ItakeUpReservation
getReservation (in resRef: String, out rd: ReservationDetails)
getReservation (in check-in: Date, in selc: int ,out rd: ReservationDetails)
getReservation (in resRef:String)
checkReservation (in cust_id: String , in hot_id : int, in res_num: int, out boolean)
beginStay (in resRef: String, out roomNumber: int)
*
Reservation
reRef:String
date:DateRang
claimed:Bolean
*
1
Hotel
Id : HotelId
1
1..*
Room
Number:String
*
*
*
1..*
1
Guest
Name: String
postcod:String
email: String
1
Room Type
Name : String
1

Figure A.23 Interface specification diagram for ItakeUpReservation

A.2.3.4 Specifying Components

So far we use the usage contract, the contract between a component object and its clients, to specify the component interfaces. Here we will clarify additional specification information that the component implementer and assembler need to be aware of, the dependencies of a component on other interfaces (realization contract).

Offered and used interfaces

We have already done this in Section A.2.1.6 in Figure A.11; where we created initial components architecture, but we must divide that diagram into pieces specific to each component. For example the hotel manager component specification shown in Figure A.24 tells that this component must offer the *IHotelMgt* and *IADUHotel* interfaces and it doesn't need any other interfaces to interact with. However, Figure A.25 shows the specification of the reservation system component and it tells that this component must offer two system interfaces and must use the three other interfaces. Actually, this specification doesn’t tell us exactly how implementations of the component must use those interfaces.

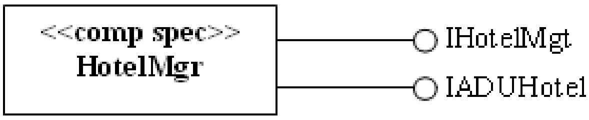


Figure A.24 Component specification diagram for HotelMgr

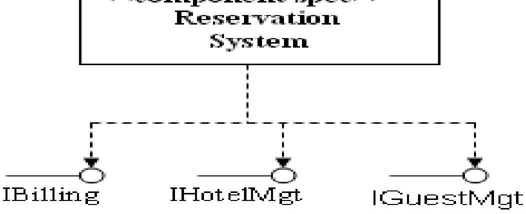


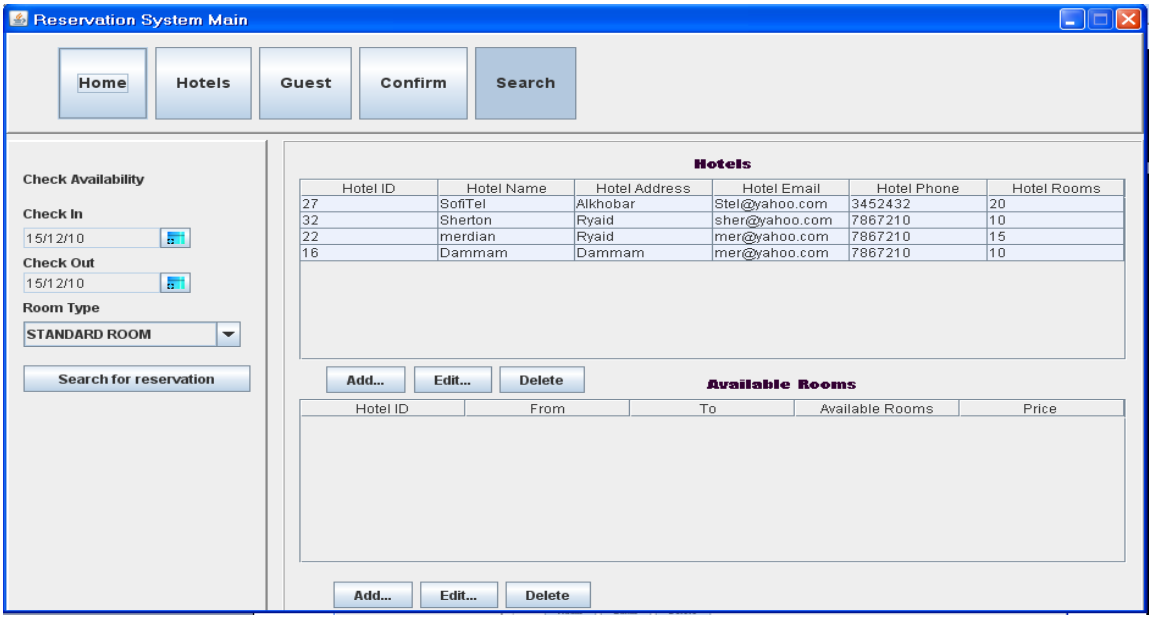
Figure A.25 Component specification diagram for ReservationSystem

A.3 Provisioning Stage

The main purpose of the provisioning phase is to find out from where components can be obtained, either by directly implementing the specification or by looking for an existing component that fulfills the specification. We searched in the internet to find components or plug-ins that fulfills our specifications but we didn't find anything related to HRS were built as components or plug-ins. We decided to build all components from scratch as plug-ins by using Eclipse 6.5.

A.4 Assembly

In this phased we hook all components and existing software assets together, implement the glue-code that tie all components and assets together, to build the system and design a user graphical interface for the system. Actually, we have used NetBeans 6.7 to glue these components together and built the user graphical interface. Figures A.26 to A.29 show you some snapshot from the H.R.S.



Check Availability

Check In: 15/12/10

Check Out: 15/12/10

Room Type: STANDARD ROOM

Search for reservation

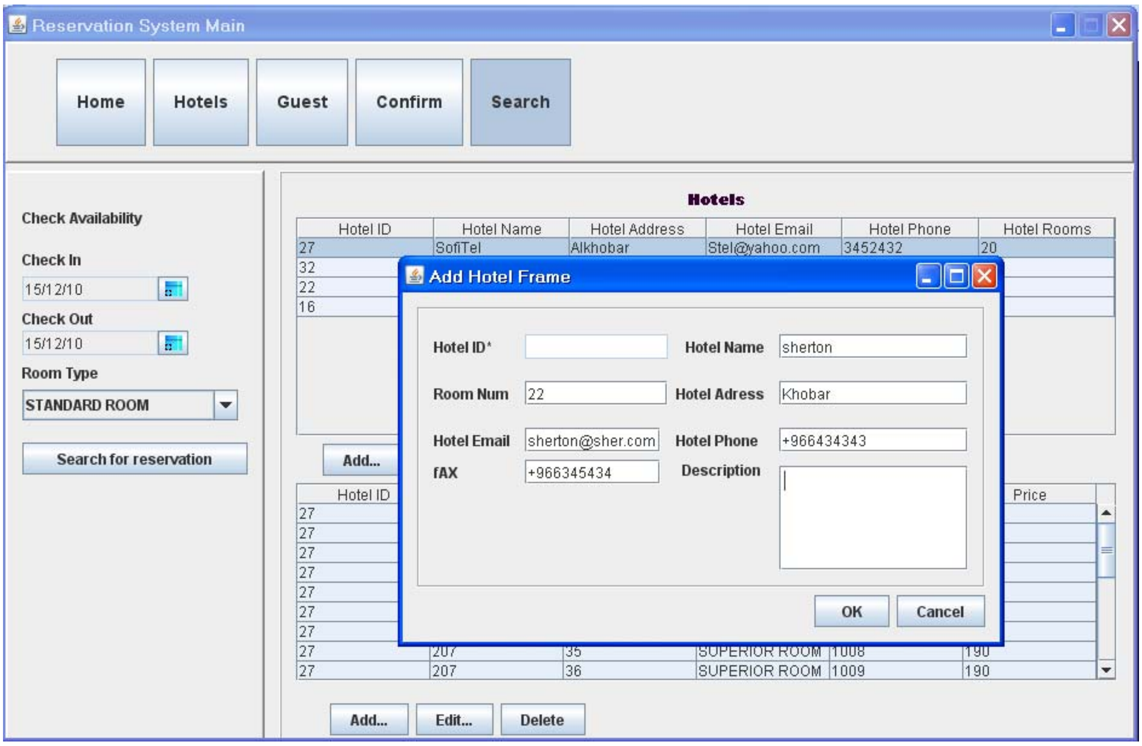
Hotels

Hotel ID	Hotel Name	Hotel Address	Hotel Email	Hotel Phone	Hotel Rooms
27	SofiTel	Alkhobar	Stel@yahoo.com	3452432	20
32	Sherton	Ryaid	sher@yahoo.com	7867210	10
22	merdian	Ryaid	mer@yahoo.com	7867210	15
16	Dammam	Dammam	mer@yahoo.com	7867210	10

Available Rooms

Hotel ID	From	To	Available Rooms	Price

Figure A.26 in this frame you can search for a reservation in specified hotel and dates



Check Availability

Check In: 15/12/10

Check Out: 15/12/10

Room Type: STANDARD ROOM

Search for reservation

Hotels

Hotel ID	Hotel Name	Hotel Address	Hotel Email	Hotel Phone	Hotel Rooms
27	SofiTel	Alkhobar	Stel@yahoo.com	3452432	20
32					
22					
16					

Add Hotel Frame

Hotel ID*:

Hotel Name: sherton

Room Num: 22

Hotel Adress: Khobar

Hotel Email: sherton@sher.com

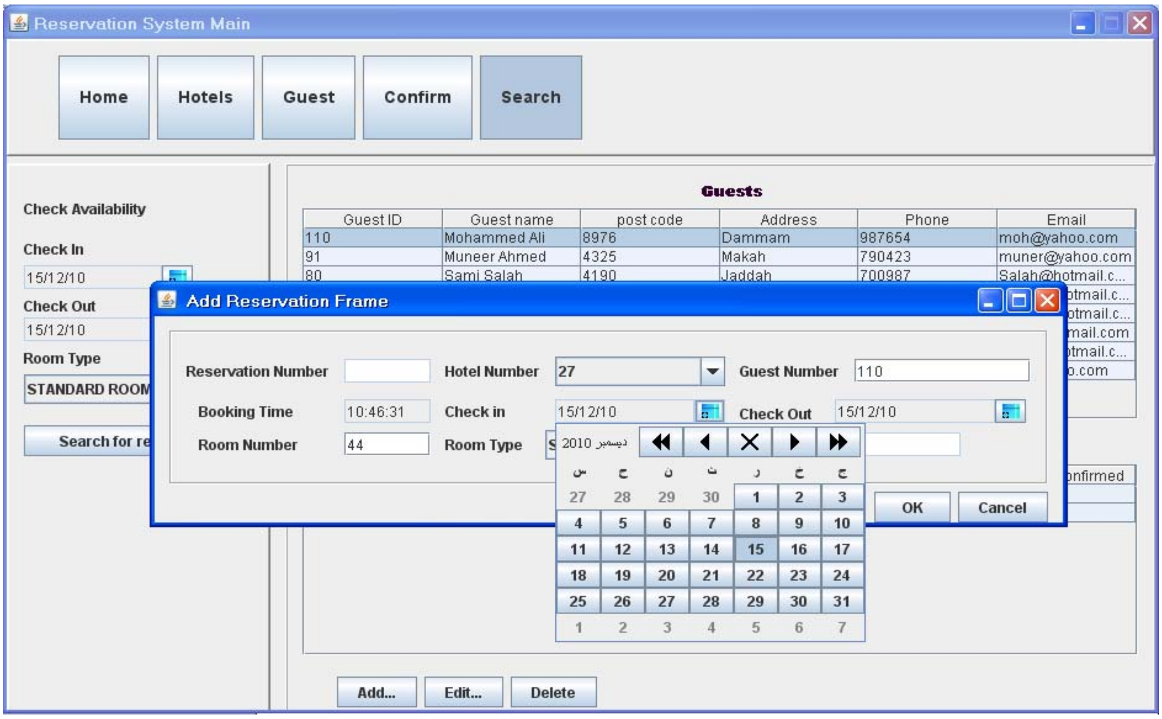
Hotel Phone: +966434343

fAX: +966345434

Description:

Hotel ID					Price
27					
27					
27					
27					
27					
27					
27					
27	207	35	SUPERIOR ROOM	1008	190
27	207	36	SUPERIOR ROOM	1009	190

Figure A.27 in this frame you can enter and manage hotels and rooms data

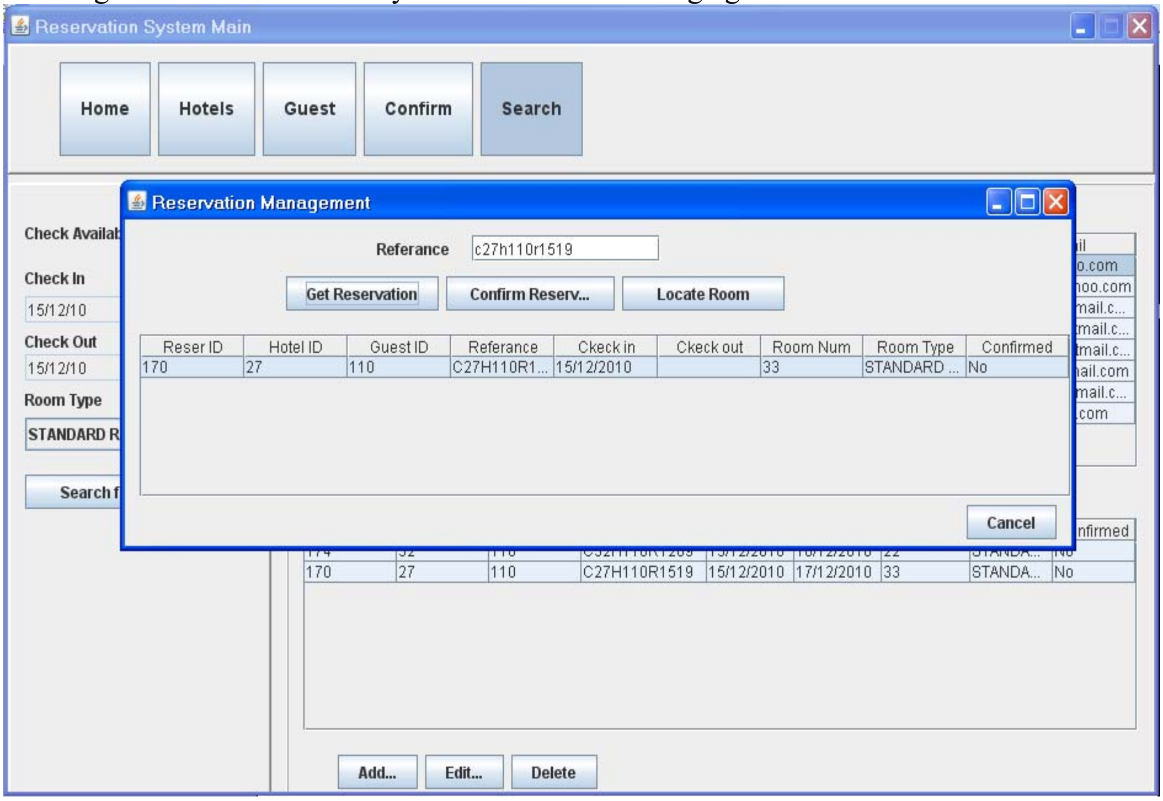


Check In: 15/12/10
Check Out: 15/12/10
Room Type: STANDARD ROOM

Guest ID	Guest name	post code	Address	Phone	Email
110	Mohammed Ali	8976	Dammam	987654	moh@yahoo.com
91	Muneer Ahmed	4325	Makah	790423	muner@yahoo.com
80	Sami Salah	4190	Jaddah	700987	Salah@hotmail.c...

Reservation Number:
Hotel Number: 27
Guest Number: 110
Booking Time: 10:46:31
Check in: 15/12/10
Check Out: 15/12/10
Room Number: 44

Figure A.28 in this frame you can enter and manage guest and reservation data



Check In: 15/12/10
Check Out: 15/12/10
Room Type: STANDARD R

Referance: c27h110r1519

Reser ID	Hotel ID	Guest ID	Referance	Ckeck in	Ckeck out	Room Num	Room Type	Confirmed
170	27	110	C27H110R1...	15/12/2010		33	STANDARD ...	No

170	27	110	C27H110R1519	15/12/2010	17/12/2010	33	STANDA...	No

Figure A.29 in this frame you can get guest reservation and confirm it

APPENDIX B

B.1 Reservation System

Red Group

Reservation System

The Hotel Reservation System (HRS) is a software application to assist a hotel management to manage reservation. It allows guests to made reservations in any hotel in the chain. Also it allows them to search hotels' rooms and confirms or cancels his/her reservations. This system allows hotel chain management to do various operations such as add new hotel, add rooms to the hotel and all operations that manipulate hotels and rooms data.

In addition, HRS helps guest to looking for room in different hotels in certain date; and it allows them to confirm or canceling their reservation. Guest can only reserve one room per each reservation and each reservation belongs to one hotel.

When guest looking for room he must enter check-in date and check-out date, check-in date must be lower than check-out date and the difference is one day.

Reservation System Main

Reservation Search/ Delete Guest/Delete hotel/Delete Reservation/Delete Room

In this frame you can do the following operation: reservation search, delete guest, delete hotel, delete reservation, and delete room. In this frame system works in three modes.

- The first mode is *when you click home button.* In this mode you can just search for room in hotel by entering check-in date, check-out date, type of room do you like and hotel ID.

- The second mode is *when you click hotel button*. In this mode you can do all operations relate to hotel management such as display hotels and display hotel rooms also call add hotel frame, delete hotel, call update hotel frame, call add room frame , delete room, and call update room frame.

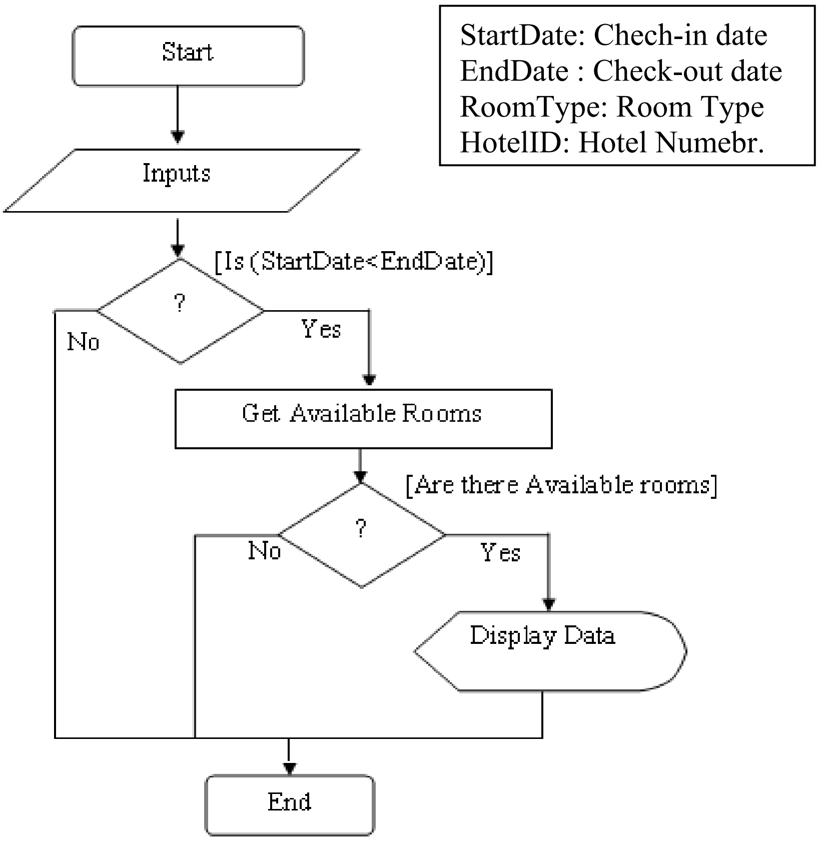
- The third mode is *when you click guest button*. In this case you can do all operations relate to guest such as display guests and display reservation for each guest also you can call add guest frame , call update guest frame , delete guest, call add reservation frame, call update reservation frame, and delete reservation.

Start Time:
End Time:

Module: Reservation System Main	**Method: Reservation Search**

Description:

This method works in the first mode (Home button). It allows you to looking for reservation by entering check-in date, check-out date, room type, and hotel number. It works as follow first it verifies whether check-in date is lower than check-out date then it retrieves and displays all available rooms in the hotel and price in that date.



StartDate: Chech-in date
EndDate : Check-out date
RoomType: Room Type
HotelID: Hotel Numebr.

Complexity

Method complexity	Average Complexity	Rule
9	**20.93**	**Rule (1)**

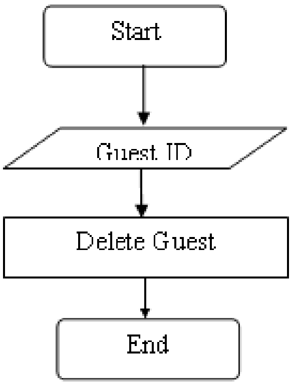
Start Time:
End Time:

Module: Reservation System Main	**Method: Delete Guest**

Description:

This method works in the third mode (Guest button). It allows you to delete guest from the system

Guest ID: Guest Number



Complexity

Method complexity	Average Complexity	Rule
7	**20.93**	**Rule (1)**

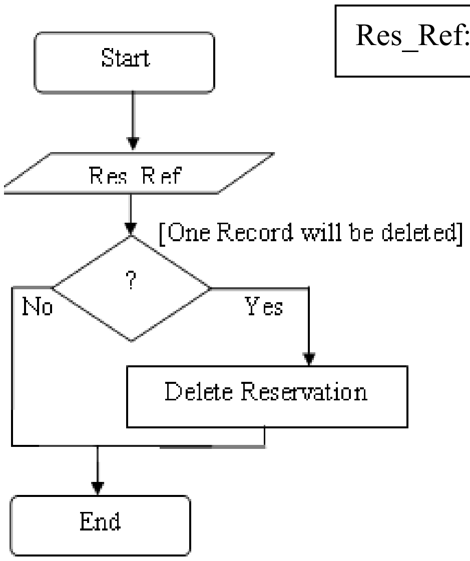
Start Time: End Time:

Module: Reservation System Main	**Method: Delete Reservation**

Description:

This method works in the third mode (Guest button). It allows you to delete guest reservation.

Res_Ref: Reservation Reference



Complexity

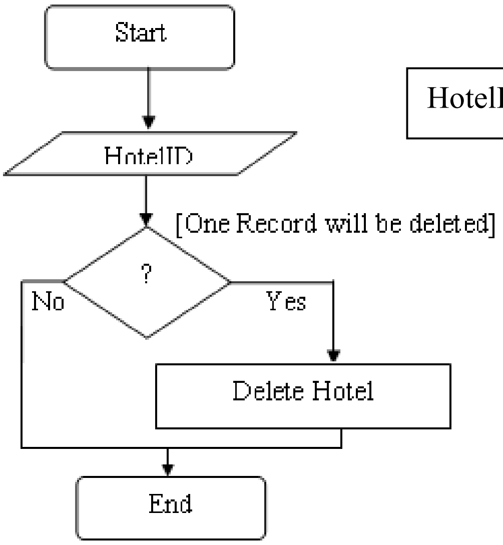
Method complexity	Average Complexity	Rule
11	**20.93**	**Rule (2)**

Start Time:
End Time:

Module: Reservation System Main	**Method: Delete Hotel**

Description:

This method works in the second mode (Hotel Button). It allows you to delete hotel from the system



HotelID: Hotel Number

Complexity

Method complexity	Average Complexity	Rule
17	**20.93**	**Rule (2)**

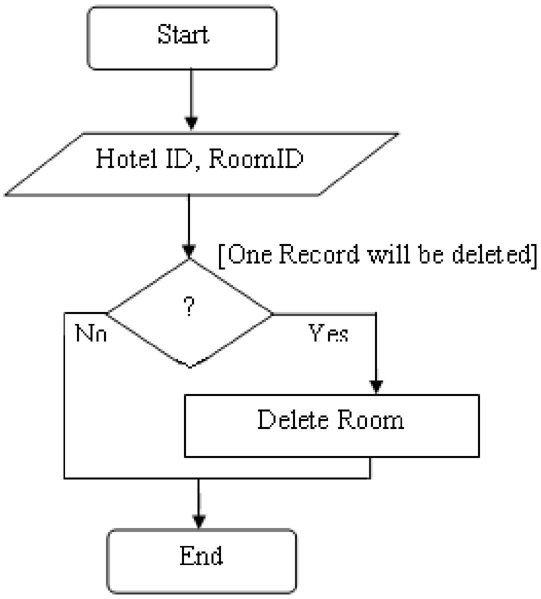
Start Time:
End Time:

Module: Reservation System Main	**Method: Delete Room**

Description:

This method works in the second mode (Hotel Button). It allows you to delete room from a hotel.

HotelID: Hotel Number
RoomID: Room Number



Complexity

Method complexity	Average Complexity	Rule
18	**20.93**	**Rule (2)**

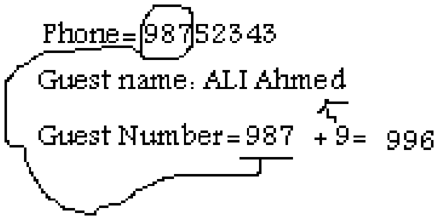
Add Guest Frame

Add_Guest

This frame works in the third mode (Guest Button). In this frame hotel administrator can insert new guest to the system. Each guest has a unique number, name, email, phone, address, and note.

Note: guest number is computed by the software as following: the first three numbers from guest phone and add to them length of guest name.

For example:



Start Time:
End Time:

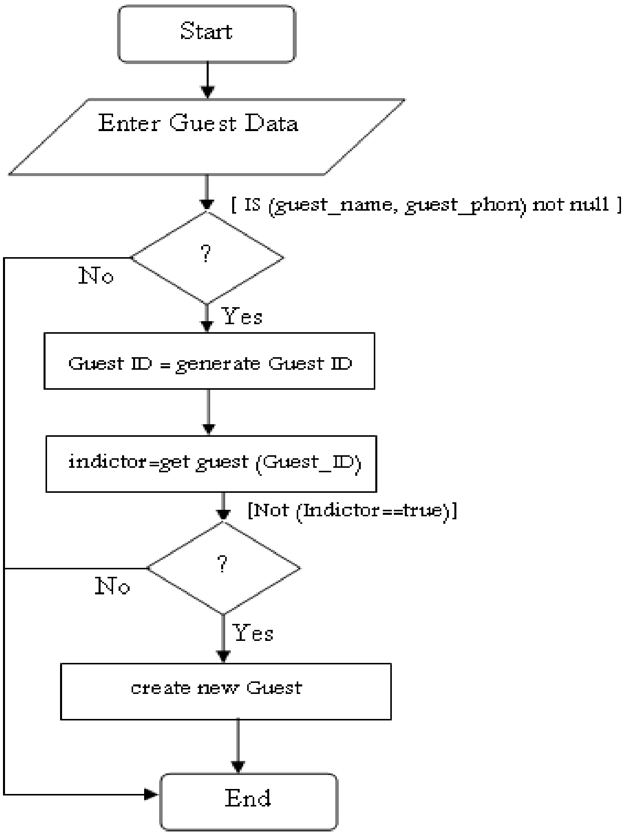
Module: Add Guest Frame	**Method: Add_Guest**

Description:

This method allows hotel reservation administrator to insert new guest to the system. Each guest has ID, name, post-code, email, phone, address, and note. All these data must be entering to the system. It checks whether guest is already exist in the system or not. So if guest doesn't exit it creates new guest otherwise it informs administrator that this guest already exists.

Note: guest number is computed automatically by the software as follows:

The first three digits of guest phone plus the length of guest name.


Start
Enter Guest Data
[IS (guest_name, guest_phon) not null]
?
No
Yes
Guest ID = generate Guest ID
indictor=get guest (Guest_ID)
[Not (Indictor==true)]
?
No
Yes
create new Guest
End

Guest_name: Guest Name
Guest_p_c: Post Code
Guest_email: Guest email
Guest_phone: Guest Phone
Guest_add: Guest address
Guest_notes: note

Complexity

Method complexity	Average Complexity	Rule
17	**20.93**	**Rule (2)**

Update Guest Frame

Update_Guest

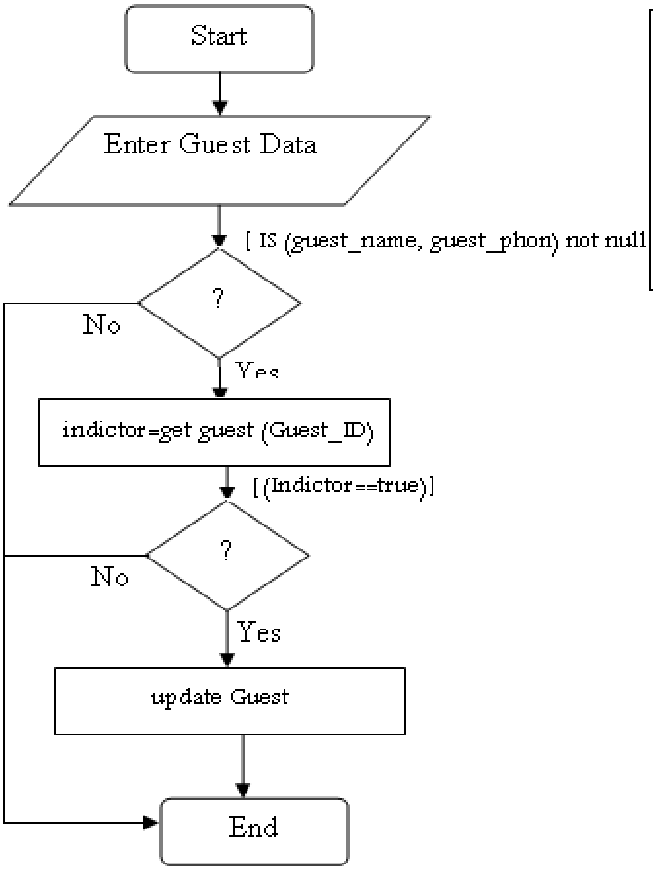
This frame works in the third mode (Guest Button). In this frame hotel administrator can update guest data.

Start Time:
End Time:

Module: Update Guest Frame	Method: Update_Guest

Description:

This method allows hotel reservation administrator to update guest record. Before updating guest data it checks whether guest already exists in the system or not. So if guest exits it updates guest record otherwise it informs administrator that this guest doesn’t exists.


Start
Enter Guest Data
[IS (guest_name, guest_phon) not null
?
No
Yes
indictor=get guest (Guest_ID)
[(Indictor==true)]
?
No
Yes
update Guest
End

Guest_ID: GuestNumber
Guest_name: Guest Name
Guest_p_c: Post Code
Guest_email: Guest email
Guest_phone: Guest Phone
Guest_add: Guest address
Guest_notes: note

Complexity

Method complexity	Average Complexity	Rule
17	**20.93**	**Rule (2)**

Add Reservation Frame

Add_Reservation

This frame allows hotel reservation administrator to add new reservation. It works as follows: first it generates reservation number then it verifies whether this reservation already exists or not. Then it checks check-in date and check-out date (check in date < check out date and the difference is one day), then it checks guest availability so if guest exits procedure is proceed but if guest not exists system informs him to enter guest data. After that it checks if there available rooms in the specified hotel or not when there is available room reservation is created and a notification message is sent to guest.

These notes must be considered when you want to add new reservation:

Note 1: guest must be enrolled in the system

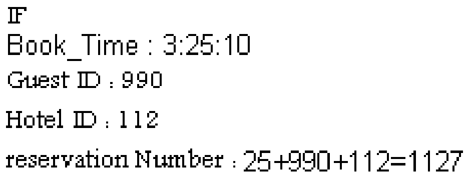
Note 2: check-in date must be lower than check-out date (e.g. 1/1/2010 & 2/1/2010)

Note 3: the difference between check-in date and check-out date must be one day.

Note 4: reservation number is computed automatically by the system as follows:

It takes minutes from booking time and adds them to guest number and hotel number.

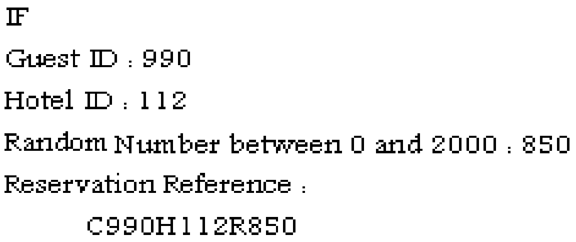
For example:



IF
Book_Time : 3:25:10
Guest ID : 990
Hotel ID : 112
reservation Number : 25+990+112=1127

Note 5: reservation reference is generated by concatenating Character "C" with guest-ID and character "H" with hotel-ID and character "R" with random number between 0 and 2000.

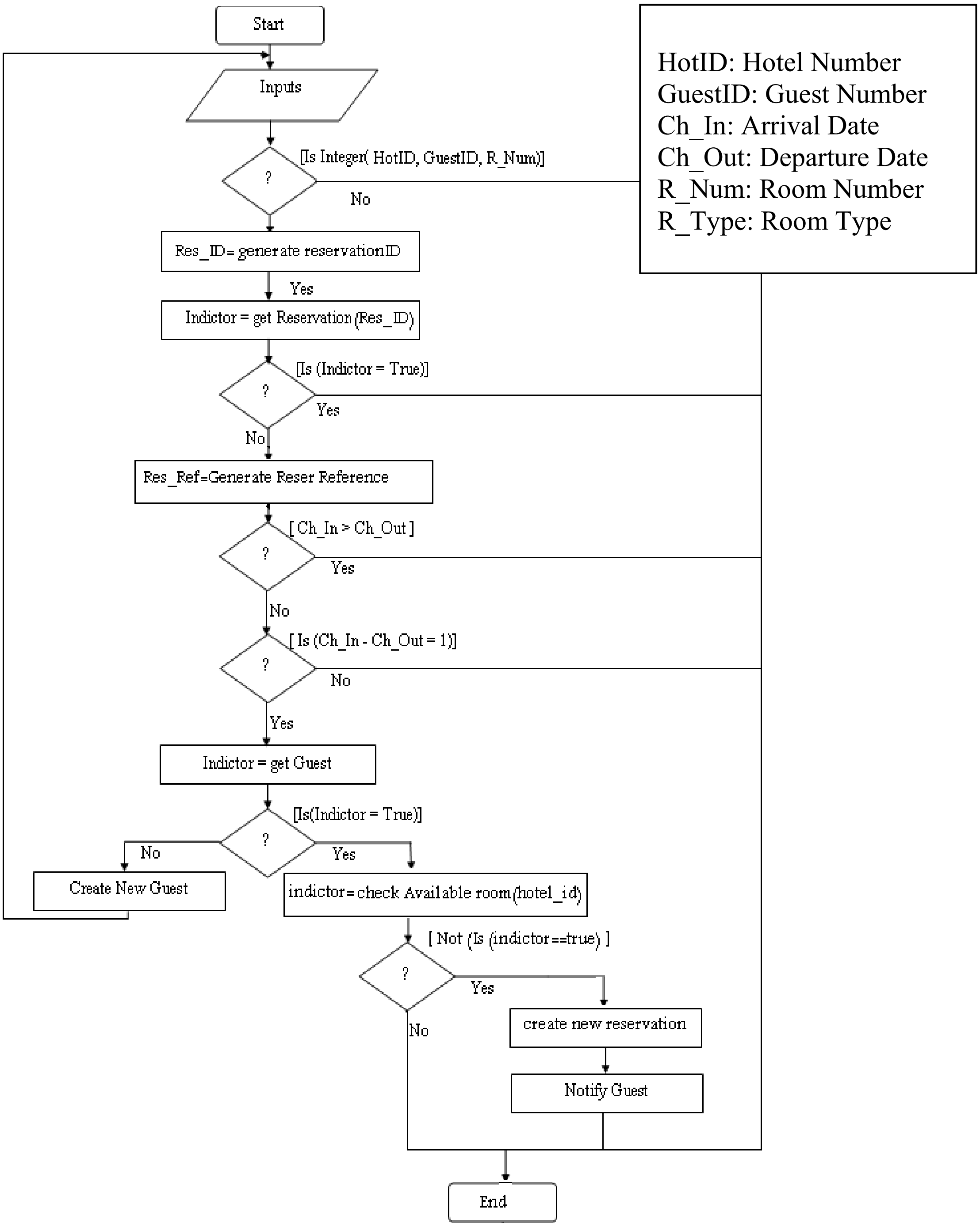
For example:



IF
Guest ID : 990
Hotel ID : 112
Random Number between 0 and 2000 : 850
Reservation Reference :
C990H112R850

Start Time:
End Time:

Module: Add Reservation Frame	**Method: Add_Reservation**


Start
Inputs
[Is Integer(HotID, GuestID, R_Num)]
?
No
Res_ID= generate reservationID
Yes
Indictor = get Reservation (Res_ID)
[Is (Indictor = True)]
?
Yes
No
Res_Ref=Generate Reser Reference
[Ch_In > Ch_Out]
?
Yes
No
[Is (Ch_In - Ch_Out = 1)]
?
No
Yes
Indictor = get Guest
[Is(Indictor = True)]
?
No
Yes
Create New Guest
indictor= check Available room(hotel_id)
[Not (Is (indictor==true)]
?
Yes
No
create new reservation
Notify Guest
End

HotID: Hotel Number
GuestID: Guest Number
Ch_In: Arrival Date
Ch_Out: Departure Date
R_Num: Room Number
R_Type: Room Type

Complexity

Method complexity	Average Complexity	Rule
38	**20.93**	**Rule (3)**

Update Reservation Frame

Update_Reservation

This frame allows hotel reservation administrator to update reservation record. It works as follows: first it checks hotel id and guest id also it checks room number to ensure that these are integer values then it verifies whether this reservation already exists or not. Then it checks check-in date and check-out date (check in date < check out date and the difference is one day), then it checks guest availability so if guest exits procedure is proceed but if guest not exists system informs him to enter guest data. After that it checks if there available rooms in the specified hotel or not when there is available room reservation is created and a notification message is sent to guest.

These notes must be considered when you want to add new reservation:

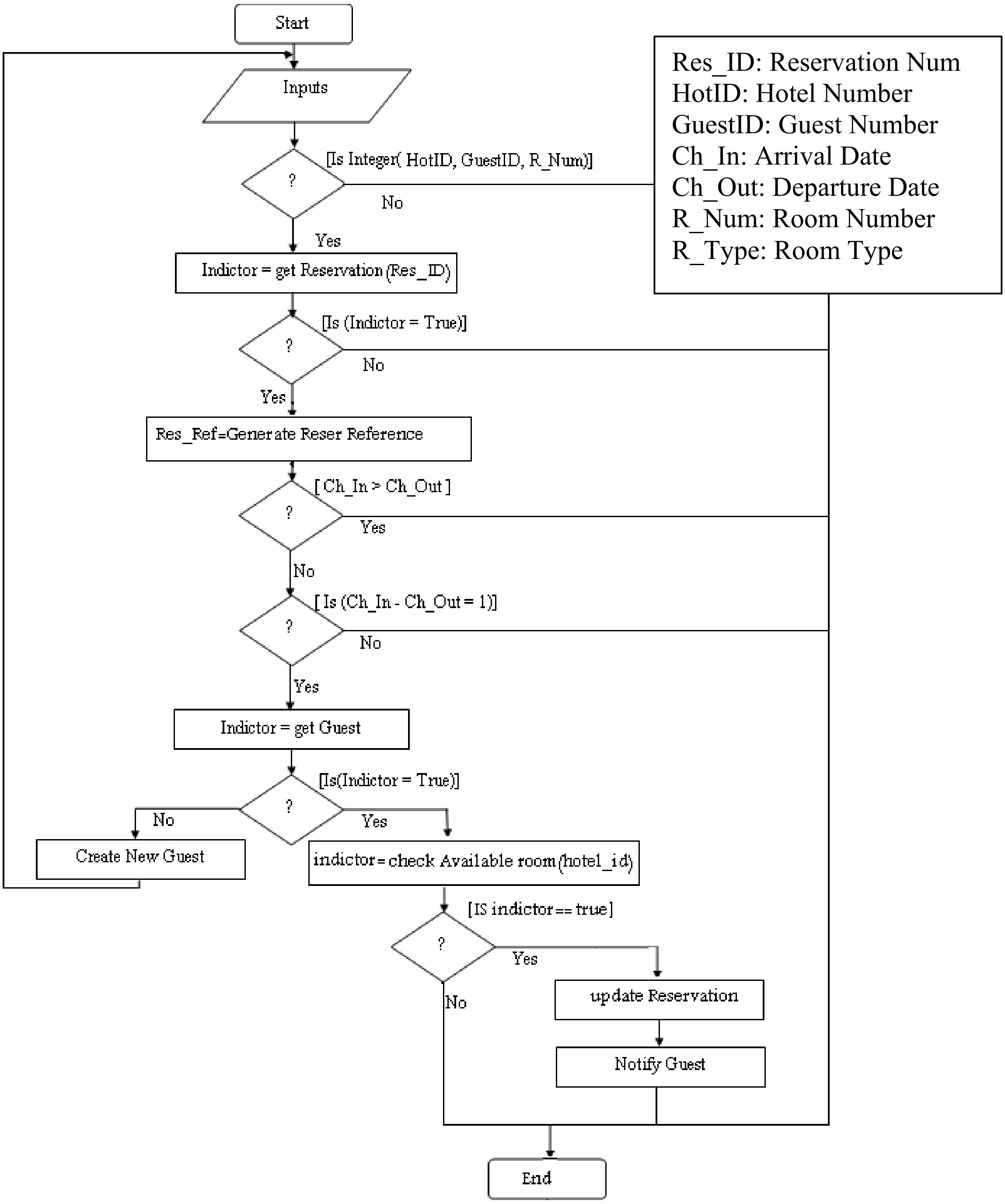
Note 1: guest must be enrolled in the system

Note 2: check-in date must be lower than check-out date (e.g. 1/1/2010 & 2/1/2010)

Note 3: the difference between check-in date and check-out date must be one day.

Start Time:
End Time:

Module: Update Reservation Frame	**Method: Update Reservation**



Complexity

Method complexity	Average Complexity	Rule
38	**20.93**	**Rule (3)**

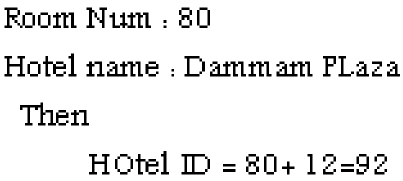
Add Hotel Frame

Add_Hotel

This frame works in second mode. In this frame hotel administrator can insert new hotel to the system. Each hotel has the following data hotel-ID, hotel name, room number, hotel address, hotel email, hotel phone, and description.

Note1: hotel number is computed automatically by the system as follows:
Number of rooms in a hotel plus the length of the hotel name.

For example:



Room Num : 80
Hotel name : Dammam FLaza
Then
HOtel ID = 80+ 12=92

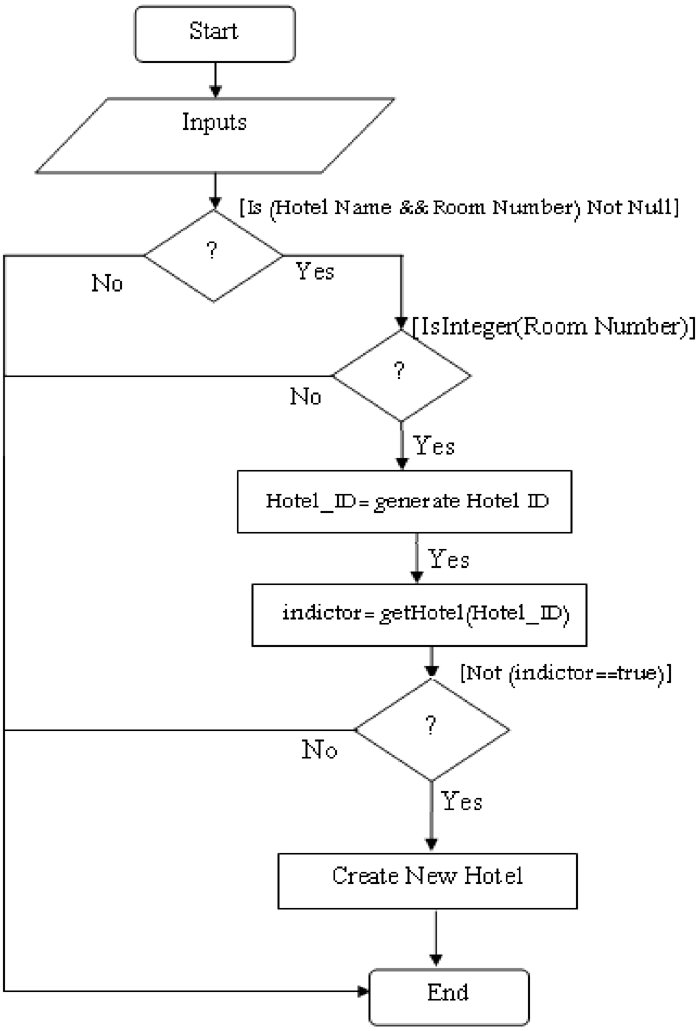
Start Time:
End Time:

Module: Add Hotel Frame	Method: Add_Hotel

Description:

This method allows hotel administrator to add new hotel to the HRS. It checks whether hotel is already exist in the system or not. So if hotel doesn't exit in the database it creates new hotel in the database otherwise it doesn't create new hotel.

Note1: hotel number is computed automatically by the system as follows:
Number of rooms in a hotel plus the length of the hotel name.



Hot_Name: Hotel Name
Hot_room: Hotel Room
Hot_Adrl: Hotel Address
Hot_Phone: Hotel Phone
Hot_fax: Hotel Fax
Description: note

Complexity

Method complexity	Average Complexity	Rule
26	**20.93**	**Rule (2)**

Update Hotel Frame

Update_Hotel

This frame works in the second mode. In this frame hotel administrator can update hotel data.

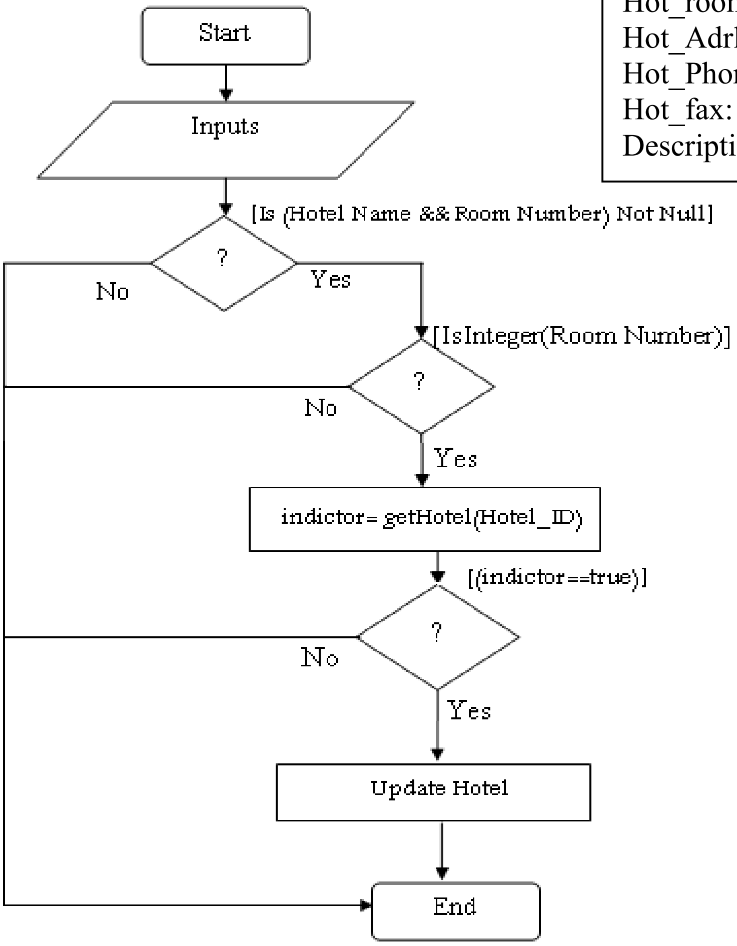
Start Time:
End Time:

Module: Update Hotel Frame	**Method: Update_Hotel**

Description:

This method allows hotel reservation administrator to update hotel record. Before updating hotel data it checks whether hotel already exists in the system or not. So if hotel exits it updates hotel record otherwise it informs administrator that this hotel doesn’t exists.

HotelID: Hotel ID
Hot_Name: Hotel Name
Hot_room: Hotel Room
Hot_Adrl: Hotel Address
Hot_Phone: Hotel Phone
Hot_fax: Hotel Fax
Description: note



Complexity

Method complexity	Average Complexity	Rule
26	**20.93**	**Rule (2)**

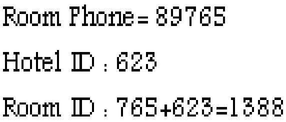
Add Room Frame

AddRoom

This frame works in the second mode (Hotel Button). In this frame hotel administrator can add new room to specified hotel. Each room has the following data hotel ID, hotel code, room number, room type, room phone, room price, and description.

Note1: room number is calculated automatically as follows:
Last three digits of room phone plus hotel ID.

For example:



Room Phone= 89765
Hotel ID : 623
Room ID : 765+623=1388

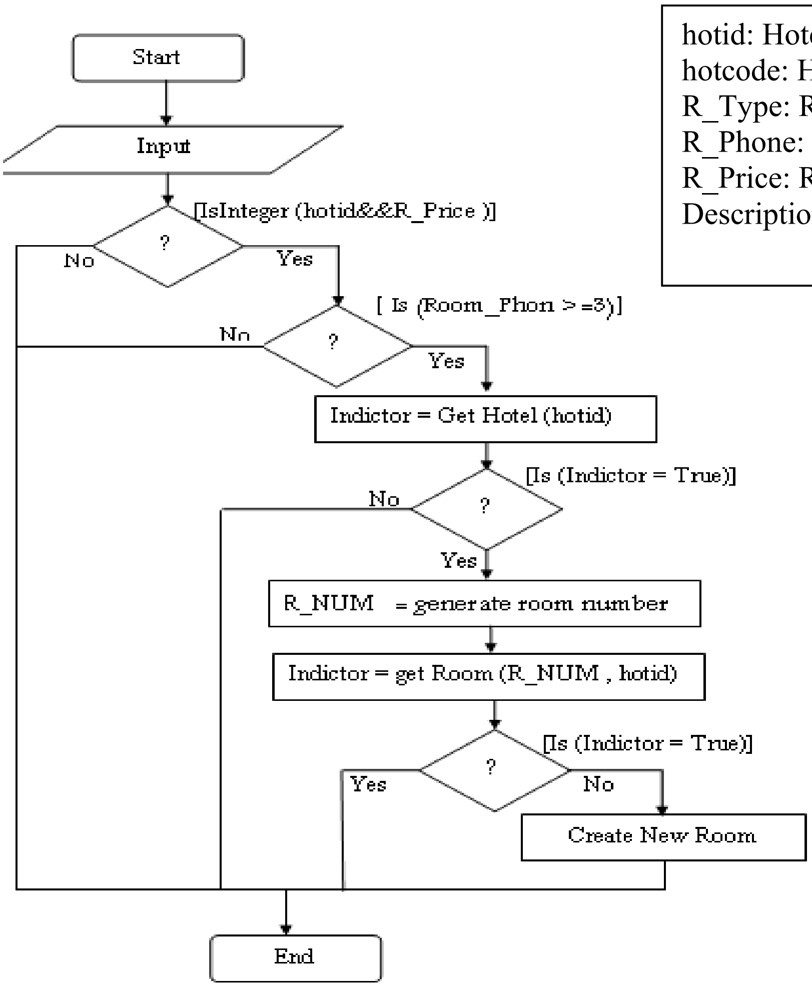
Start Time:
End Time:

Module: Add Room Frame	Method: AddRoom

Description:

This method allows hotel administrator to add new room to specified hotel. It works as follows it checks hotel-ID and price and it generates room number then it checks whether this hotel already available in the system or not also it checks whether this room exists in that hotel or not. Thus if room is not exist in the hotel it creates new room otherwise it informs user that this hotel is not available or this room already exits.

hotid: Hotel Number
hotcode: Hotel Code
R_Type: Room Type
R_Phone: Room phone
R_Price: Room Price
DescriptionNote : Note


Start
Input
[IsInteger (hotid&&R_Price)]
?
No
Yes
[Is (Room_Phon >=3)]
?
No
Yes
Indictor = Get Hotel (hotid)
[Is (Indictor = True)]
?
No
Yes
R_NUM = generate room number
Indictor = get Room (R_NUM , hotid)
[Is (Indictor = True)]
?
Yes
No
Create New Room
End

Complexity

Method complexity	Average Complexity	Rule
34	**20.93**	**Rule (3)**

Update Room Frame

Room_update

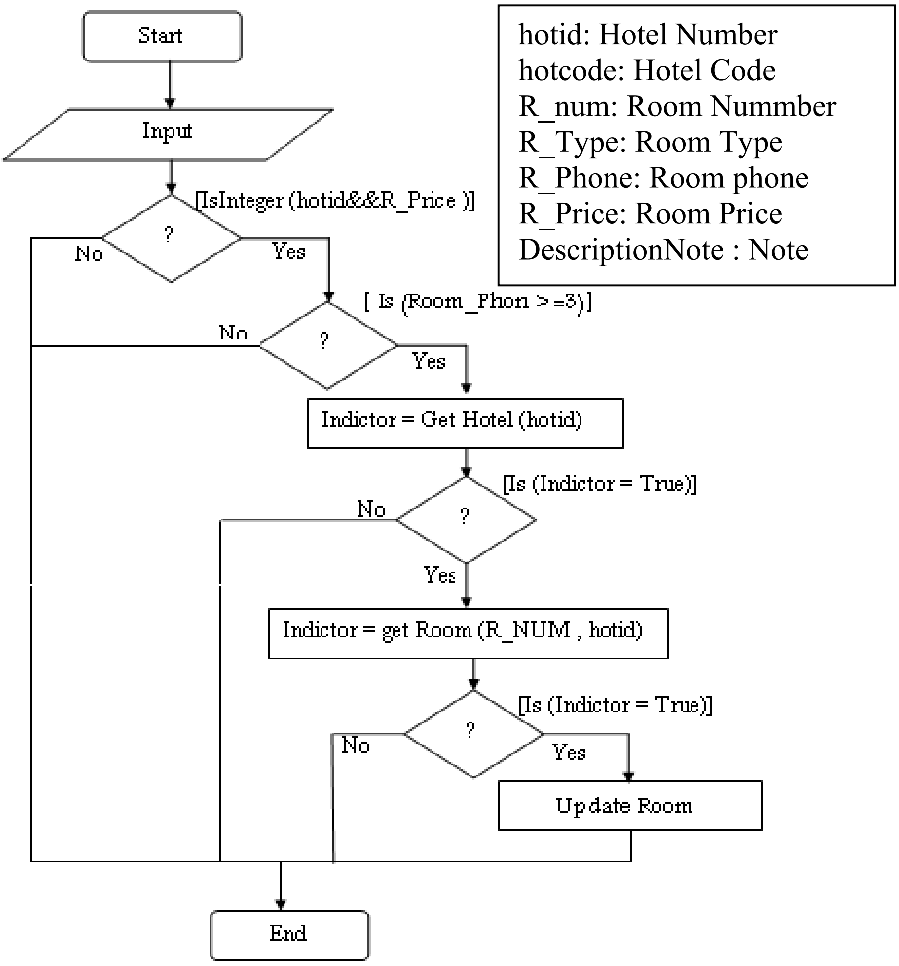
This frame works in the second mode (Hotel Button). In this frame hotel administrator can Update room record in specified hotel.

Start Time:
End Time:

Module: Update Room Frame	Method: Roomupdate

Description:

This method allows hotel administrator to update room in specified hotel. It works as follows it checks hotel-ID and price then it checks whether this hotel already available in the system or not also it checks whether this room exists in that hotel or not. Thus if room is exist in the hotel it updates room record otherwise it informs user that this hotel is not available.



hotid: Hotel Number
hotcode: Hotel Code
R_num: Room Nummber
R_Type: Room Type
R_Phone: Room phone
R_Price: Room Price
DescriptionNote : Note

Complexity

Method complexity	Average Complexity	Rule
34	**20.93**	**Rule (3)**

Reservation Management

getGuestReservation /confirm_reservation

In this frame hotel administrator can do the following operations:

- Get Guest Reservation in this method he/she can retrieve guest reservation by entering reservation reference.

- Confirm reservation in this method he/she can confirm guest reservation after retrieves its reservations.

- Locate guest room in this method he/she can allocate room for guest after confirms its reservation.

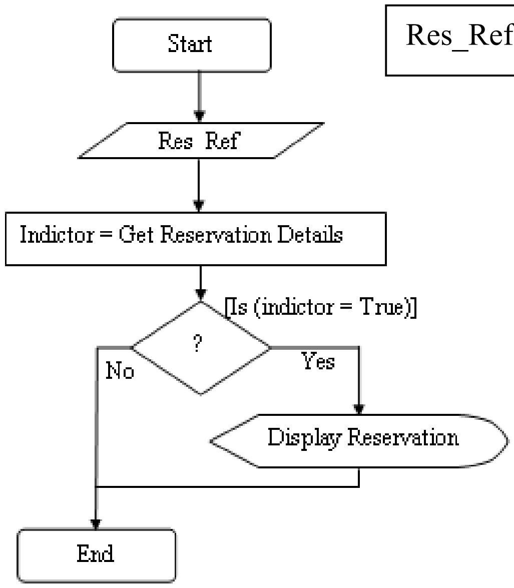
Start Time:
End Time:

Module: Reservation_management	**Method: get Reservation**

Description:

This method allows hotel administrator to retrieve guest reservation by entering guest reservation reference.

Res_Ref: Reservation Reference



Complexity

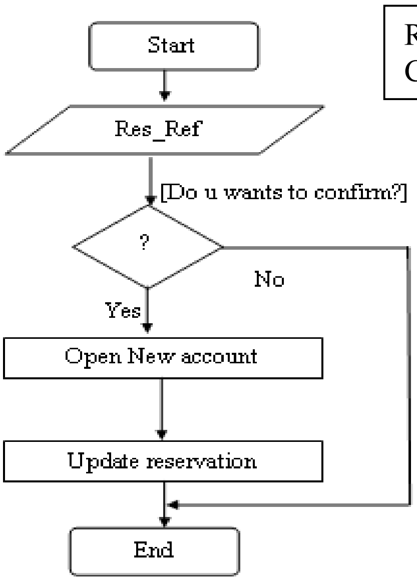
Method complexity	Average Complexity	Rule
6	**20.93**	**Rule (1)**

Start Time:
End Time:

Module: Reservation management	Method: confirm_reservation

Description:

In this method you can confirm guest reservation. So, after retrieve guest reservation you can confirm it and open new account for guest.



Res_Ref: Reservation Reference
CustID: Guest ID

Complexity

Method complexity	Average Complexity	Rule
16	**20.93**	**Rule (2)**

B.2 Library Management System

Red Group

Library System

The Library System is a "desktop application" that helps a library employee to manage the loan of books and journals. Members can borrow, return or renew (i.e., extend a current loan) books and journals.

Description

A library issues loan items to customers. Each customer is known as a member each member has a unique member number. Along with the membership number, other details on a customer must be kept such as a name, address, and date of enroll. The library is made up of a number of subject sections. Each section is denoted by a classification mark. There are two types of loan items; journal and books. A journal has a title, volume, date of issue and authors. A book has a title, authors, etc…. A customer may borrow up to a maximum of 5 items. An item can be borrowed, or renewed to extend a current loan. Each of these activity has a cost in S.R. (borrow a book cost 10 S.R. while a journal only 5 S.R.; if the member performs at least 3 operations – i.e., borrow and/or renew in the same day, she/he receive a discount of 7S.R.).

When an item is issued the borrowing customer's membership number is entered. If the number of items on loan less than 8, the procedure can proceed and the book catalog number is entered. The library must support the facility for an item to be searched and for an update of items and members.

The library employee can:

- Insert/delete/update a member
- Insert/delete/update an item in the library
- Borrow an item
- Renew an item
- Search members
- Search items

Member Management Frame

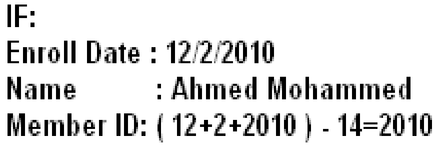
Inserting/deleting/updating/search a member

The library employee can insert a member. Each member has the following fields: *unique member number*, *name*, *date of enroll*, *email*, *address*, and *phone*. In addition, he can delete or update a member searching it by member number. The library System must support the facility "search members" by using member number.

To insert a new member the employee has to insert all data of the new member (i.e., member number, name, date of enroll, email, address, and phone).

The member number is computed automatically by the software. This value is calculated summing day, month and year of enroll date and subtracting to the result the number of letters of name.

For example:



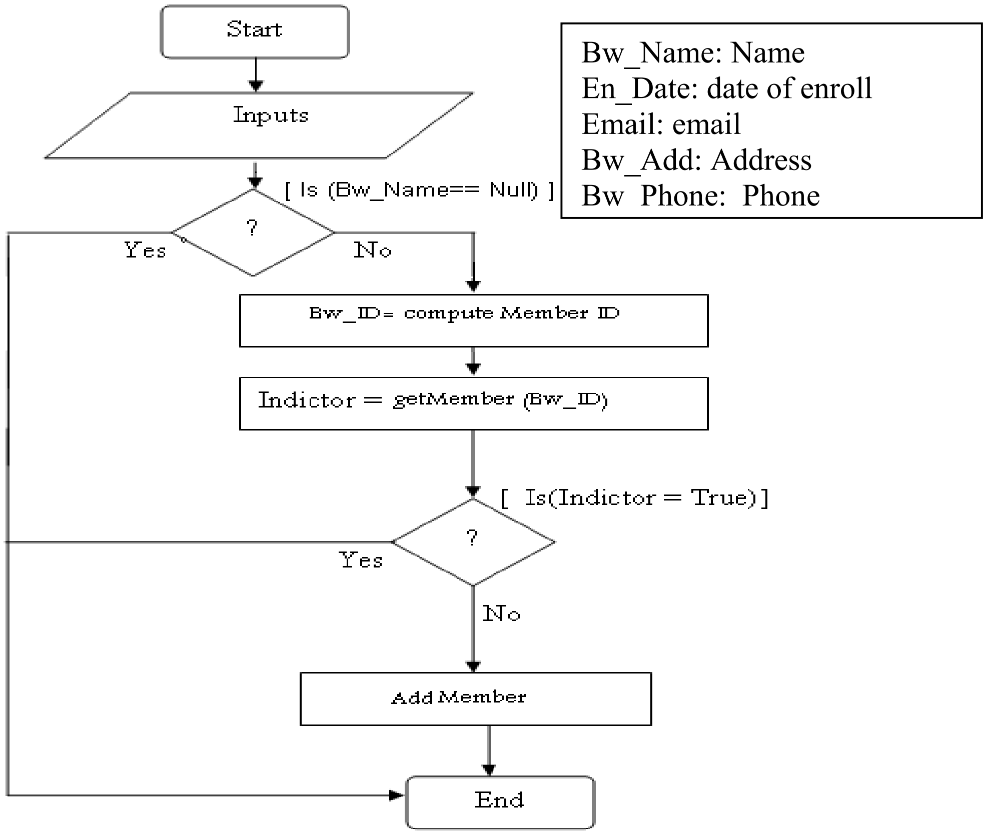
IF:
Enroll Date : 12/2/2010
Name : Ahmed Mohammed
Member ID: (12+2+2010) - 14=2010

Start Time:
End Time:

Module: Member Management Frame	**Method: add Member**

Description:

This method allows employee to add new member (i.e., member number, name, date of enroll, email, address, and phone) to the system. Before adding a new member it generates member-ID and it checks whether member is already exist in the system or not. So if member doesn't exit in the database it creates new member otherwise it doesn't create new member.



Bw_Name: Name
En_Date: date of enroll
Email: email
Bw_Add: Address
Bw Phone: Phone

Complexity

Method complexity	Average Complexity	Rule
17	**19.11**	**Rule (2)**

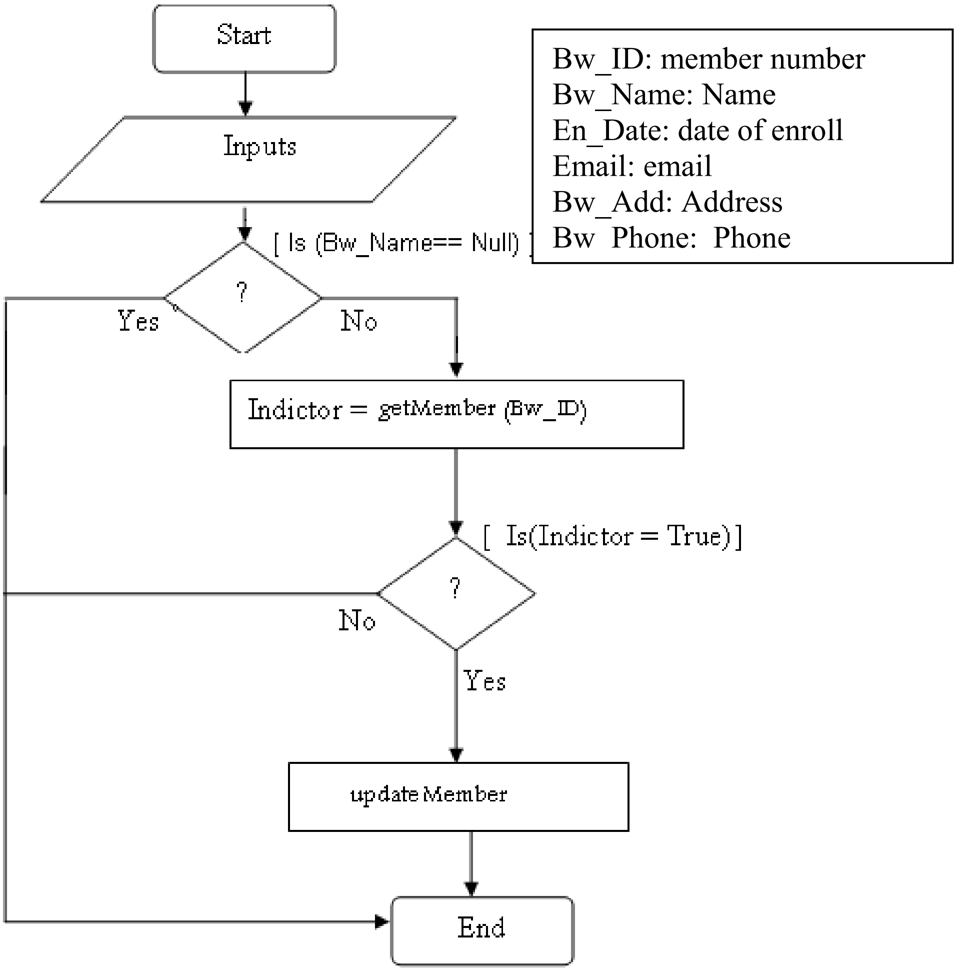
Start Time:
End Time:

Module: Member Management Frame	**Method: update Member**

Description:

This method allows employee to update member data. Also, it checks whether member is already exist in the system or not. So if member exits in the database it updates member record otherwise it informs employee that no such member is available.

Note 1: Before you update a member record you have to find it by using *search member* method.



Bw_ID: member number
Bw_Name: Name
En_Date: date of enroll
Email: email
Bw_Add: Address
Bw Phone: Phone

Complexity

Method complexity	**Average Complexity**	**Rule**
17	**19.11**	**Rule (2)**

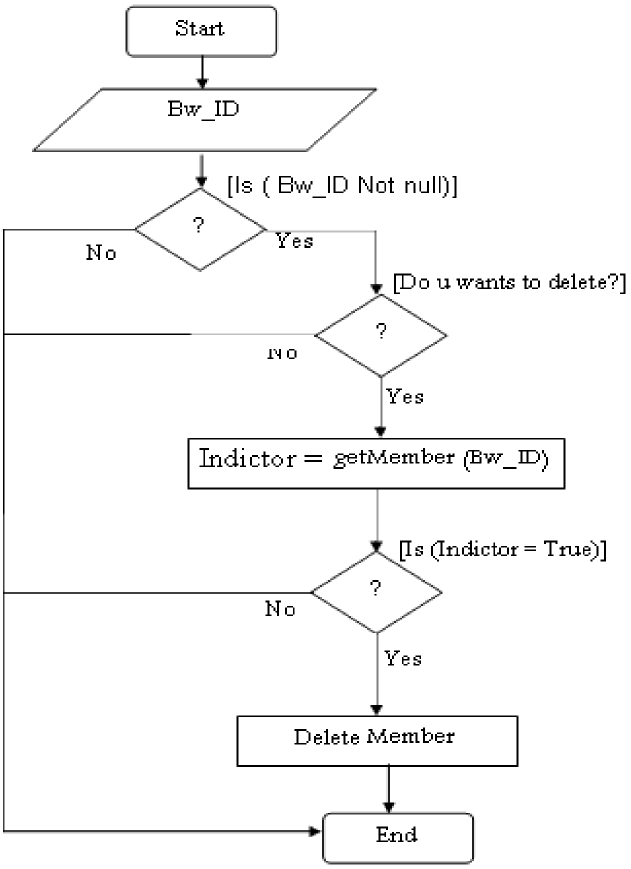
Start Time: End Time:	

Module: Member Management Frame	**Method: DeleteMember**

Description:

This method allows employee to delete member from the system. It checks whether member is already exist in the system or not. So if member exits in the database it deletes member otherwise it informs employee that no such member is available.

Note 1: Before deletes a member you have to find it by using *search member method*.



Bw_ID: member number

Complexity

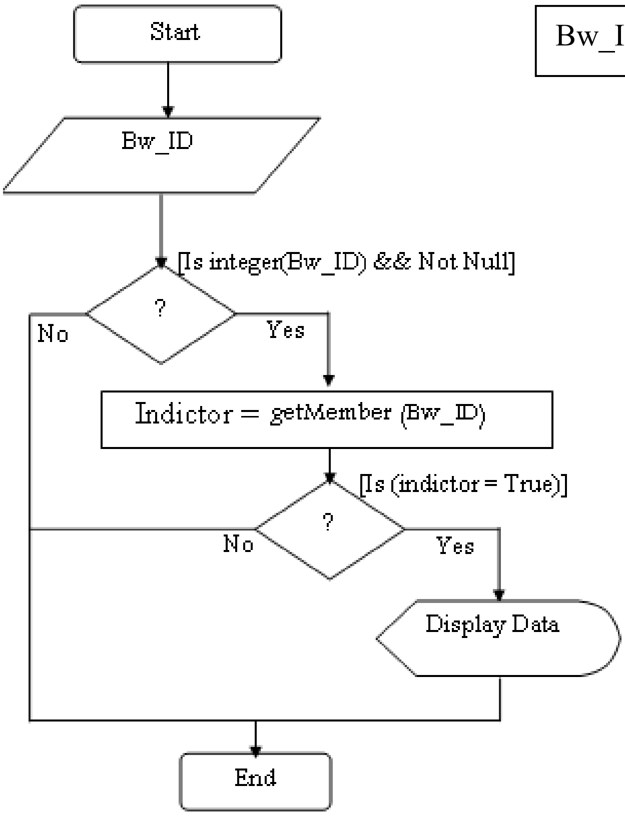
Method complexity	Average Complexity	Rule
18	**19.11**	**Rule (2)**

Start Time:
End Time:

Module: Member Management Frame	**Method: Search Member**

Description:

This method allows employee to search for member in the system. It checks whether member is already exist in the system or not. So if member exits it retrieves member record otherwise it informs employee that there is no such member.



Bw_ID: member number

Complexity

Method complexity	Average Complexity	Rule
8	**19.11**	**Rule (1)**

Item Management Frame

Inserting/deleting/updating/search an item in the library

The library employee can insert, delete, update and search an item in the library. A loan item is uniquely identified by an item ID. There are two types of loan items: journal and books.

A journal has an item ID, title, author, area (i.e., Mathematics, computer, biology, etc…), date-of-publish, location, volume and note.

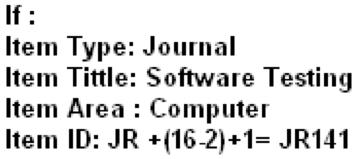
A book has an item ID, title, author, area (i.e., Mathematics, computer, biology, etc…), date-of-publish, location, edition, publication house and note.

The employee can delete or update an item searching it by item-ID. To insert a new item she/he has to insert the type item (i.e., journal or book) and all the specific fields.

The Item-ID is computed by the System as follows:

- If the item is a **journal** concatenating:
  1. 'JR'
  2. number of lower-case letters of the title
  3. '1' if the area is computer or mathematics '0' otherwise

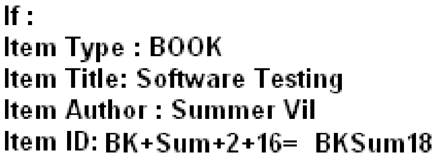
  For example:



If :
Item Type: Journal
Item Tittle: Software Testing
Item Area : Computer
Item ID: JR +(16-2)+1= JR141

- If the item is a **book** concatenating:
  1. 'BK'
  2. first, second, and third letters of author's name
  3. number of upper-case letters of the book's title plus book title length

  For example:



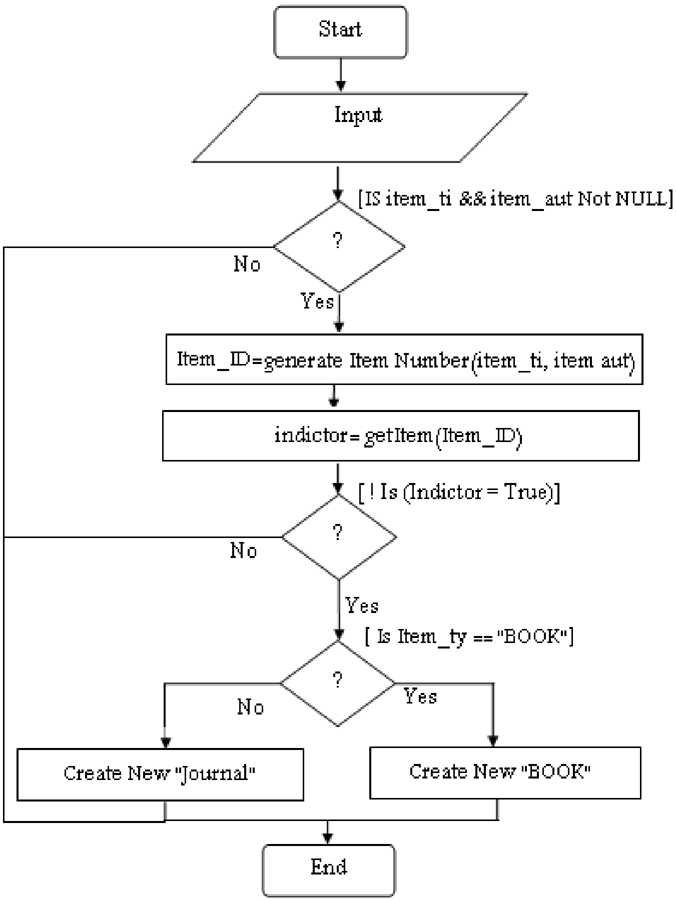
If :
Item Type : BOOK
Item Title: Software Testing
Item Author : Summer Vil
Item ID: BK+Sum+2+16= BKSum18

Start Time:
End Time:

Module: Item Management Frame	**Method: Add Item**

Description:

This method allows employee to add new item (i.e., book or journal) to the system. Each item has the following data (Item-ID, Item Type, Item Title, Item Author, Item Date, Item location, Item Edition, Item House, Item volume, and Item note). To insert a new item she/he has to insert the type item (i.e., *journal or book*) and all the specific fields. Thus, before adding a new item it first generate an item-ID then it checks whether this item is already exist in the system or not. So if the item doesn't exit in the database it creates new items otherwise it informs employee that there is conflict.


Start
Input
[IS item_ti && item_aut Not NULL]
?
No
Yes
Item_ID=generate Item Number(item_ti, item aut)
indictor= getItem(Item_ID)
[! Is (Indictor = True)]
?
No
Yes
[Is Item_ty == "BOOK"]
?
No
Yes
Create New "Journal"
Create New "BOOK"
End

Item_ty: Item Type
Item_ti: Item Title
Item_aut: Item Author
Item_Are: Item Area
Item_Pub_Date: Item Date
Item_loc: Item location
Item_Edit: Item Edition
Item_hou: Item House
Item_vol: Item Volume
Item_Pub_note: Item note

Complexity

Method complexity	Average Complexity	Rule
33	**19.11**	**Rule (3)**

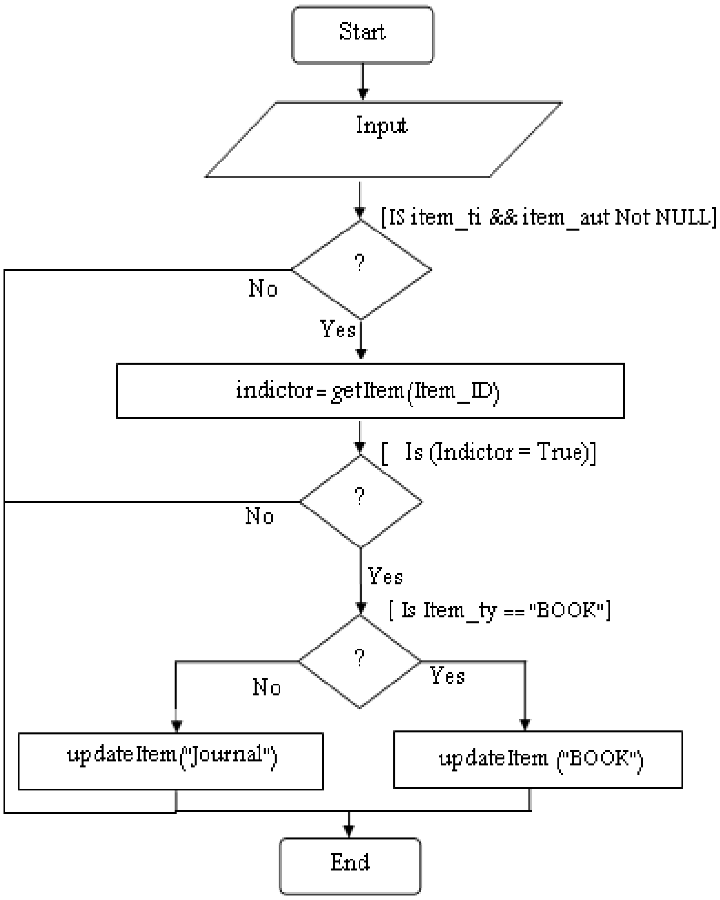
Start Time:
End Time:

Module: Item Management Frame	**Method: update Item**

Description:

This method allows employee to update item record (i.e., book or journal) in the system. Before updating an item it checks whether this item is already exist in the system or not. So if the item exits in the database it updates item record otherwise it informs employee that his item does not exist.

Note 1: Before updates an item we have to find it by using *search item method*.



Item_ID: Item ID
Item_ty: Item Type
Item_ti: Item Title
Item_aut: Item Author
Item_Are: Item Area
Item_Date: Item Date
Item_loc: Item location
Item_Edit: Item Edition
Item_hou: Item House
Item_vol: Item Volume
Item_Pub_note: Item note

Complexity

Method complexity	Average Complexity	Rule
33	**19.11**	**Rule (3)**

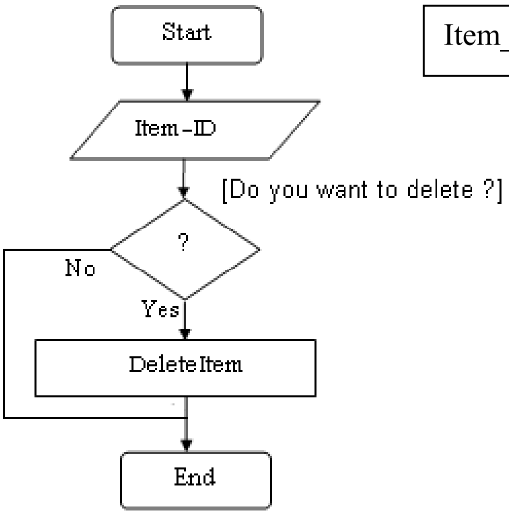
Start Time:
End Time:

Module: Item Management Frame	**Method: delete Item**

Description:

This method allows employee to delete an item from the system. It checks whether item is already exist in the system or not. So if item exits in the database it deletes item otherwise it informs employee that no such member is available.

Note 1: Before deletes an item we have to find it by using *search item method*.



Item_ID: Item ID

Complexity

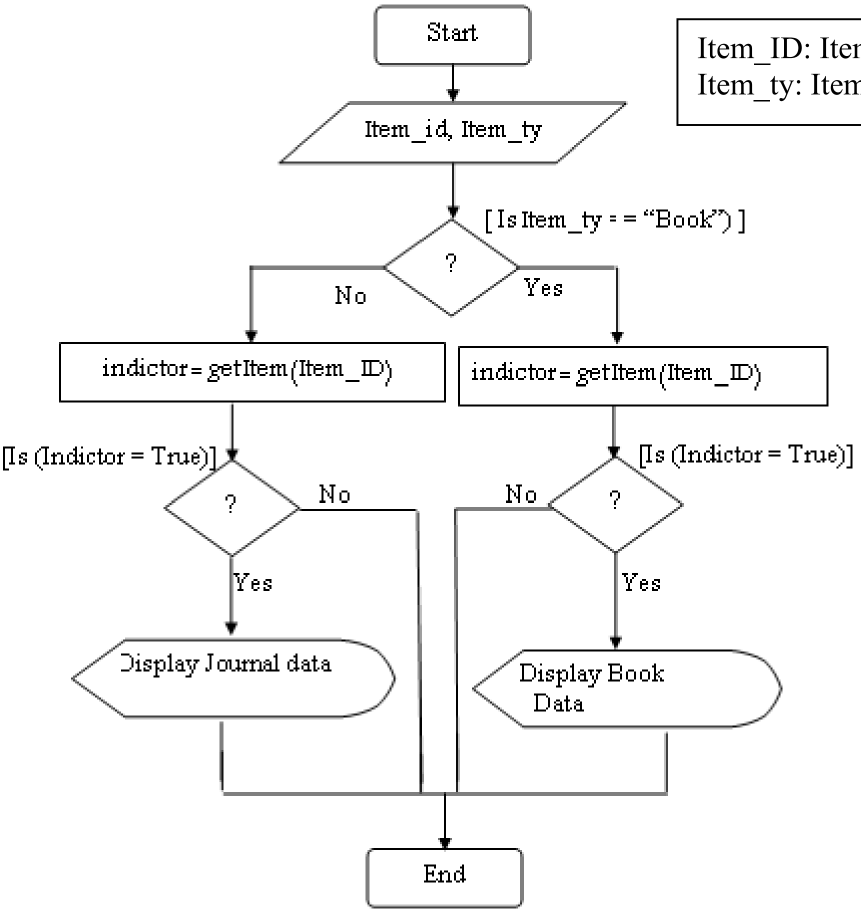
Method complexity	Average Complexity	Rule
11	**19.11**	**Rule (1)**

Start Time:
End Time:

Module: Item Management Frame	**Method: search item**

Description:

This method allows employee to search for item in the system by item-ID and item type. It checks whether item already exists or not. So if item exits it retrieves item record otherwise it informs employee that there is no such item.



Item_ID: Item ID
Item_ty: Item Type

Complexity

Method complexity	Average Complexity	Rule
23	**19.11**	**Rule (2)**

Return Items Frame

Get Member Items/Return Item/ Renew Item

The library employee can get member items that he/she has borrowed, return an item, or extend a current item loan (renew). By entering the member ID library employee can retrieve all items that have been borrowed by that member. Then he can return an item or renew loaned item.

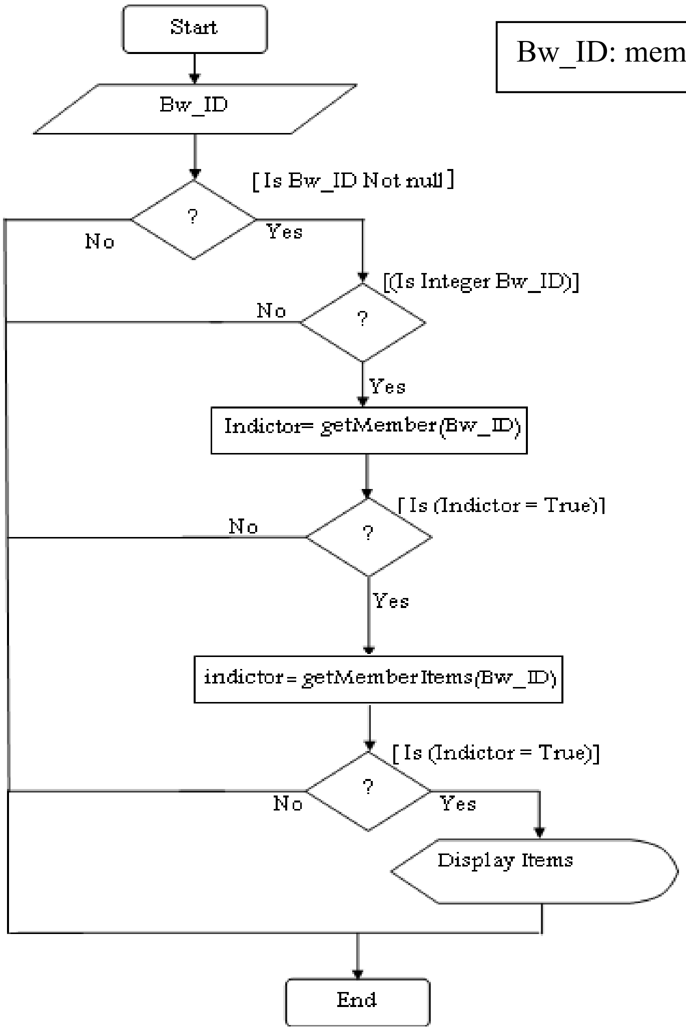
Start Time: End Time:

Module: Return Items Frame	**Method: get Member Items**

Description:

This method is allowed employee to get member items that have been borrowed by entering member-ID. First, this method checks whether the member exists in the system or not and then it retrieves and displays all items that he/she has still borrowed.

Bw_ID: member number



Complexity

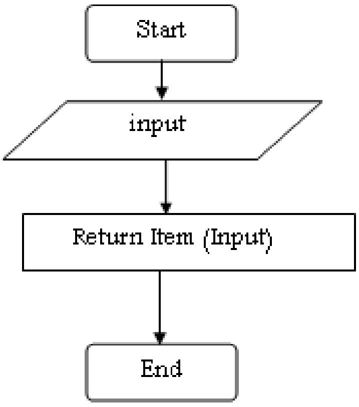
Method complexity	Average Complexity	Rule
14	**19.11**	**Rule (2)**

Start Time: End Time:

Module: Return Items Frame	**Method: Return Item**

Description:

This method allows employee to return an item to the system. You have to retrieve all items that member has borrowed.



Bw_ID: member number
Item_ID: Item ID
Br_Date: Borrow Date
Ret_Date: Return Date

Complexity

Method complexity	Average Complexity	Rule
15	**19.11**	**Rule (2)**

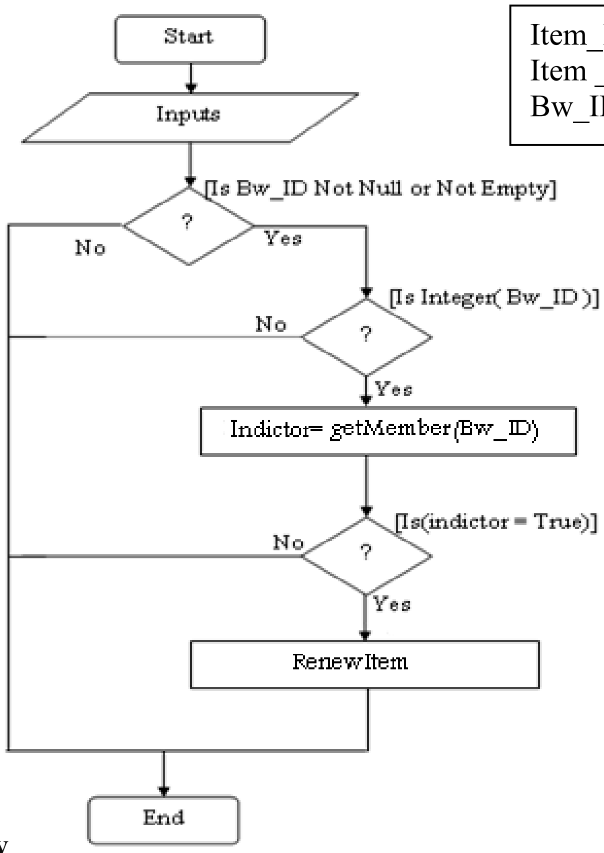
Start Time: End Time:	

Module: Return Items Frame	**Method: Renew**

Description:

This method allows employee to renew an item to a member, extend a current item loan one month (30 days). It checks whether member exists or not then it renews item.

Item_ID: Item ID
Item _ty: Item Type
Bw_ID: member number



Complexity

Method complexity	Average Complexity	Rule
24	19.11	Rule (2)

Search Item Frame

Search for Item/Loan an Item/ Items Cost

The library System must support the facility “search Items”. The library members can search for items using these terms item area, type of item, search based (e.g. title, author, Meta data), and search string. Through this facility members can look for items (books or journals) and after that if he/she wants to borrow an item the system asks for member number to validate whether he/she is member in the library or not then it checks if the number of items on loan for this member is less than 6 and the item that he/she wants is checked in then the procedure can proceed.

A member may borrow up to a maximum of 5 items. An item can be borrowed or renewed to extend a current loan. Each of these activity has a cost (borrow a book cost 10 SR while a journal only 5 SR; if the member performs at least 3 operations – i.e., borrow, and/or renew – in the same day, she/he receive a discount of 7 SR).

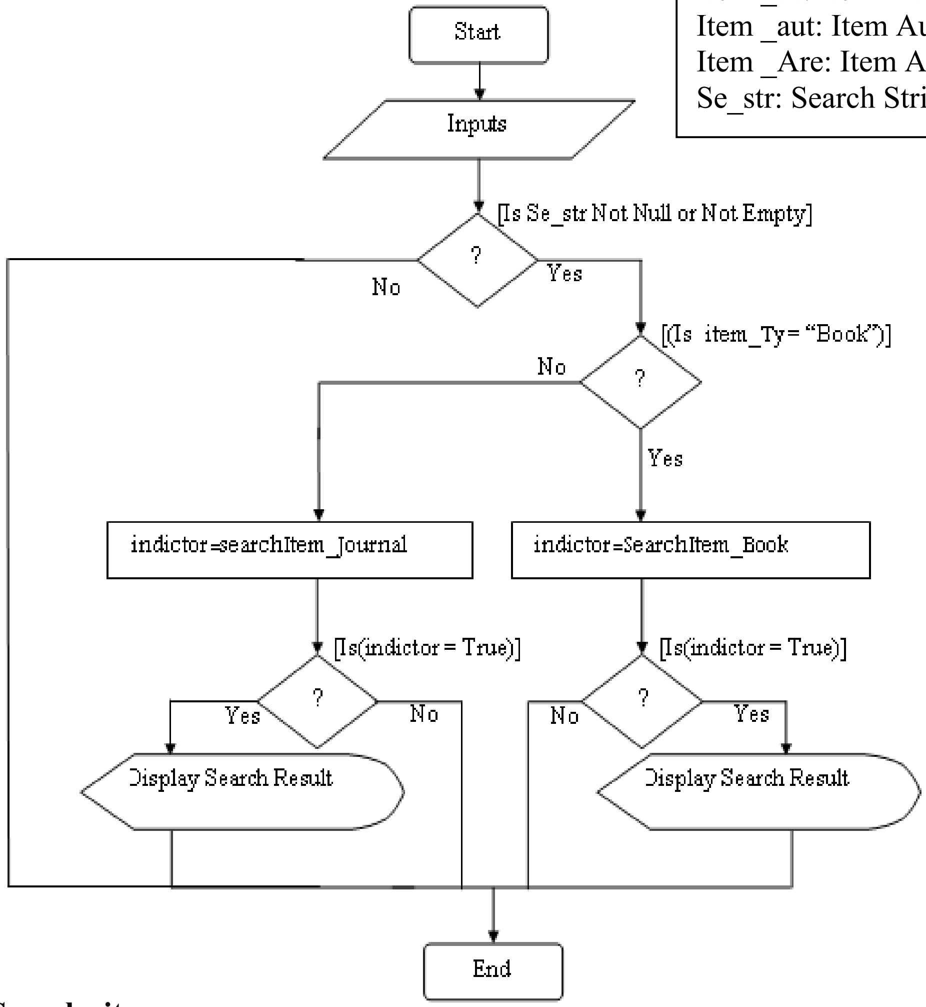
Start Time:
End Time:

Module: Search Item Frame	**Method: Search for items**

Description:

This method allows member to search for items in the library system. The library members can search for items using these terms *item area, type of item, search based (e.g. title, author, Meta data), and search string.* Member inputs must be checked before execute search query.

Item_ty: Item Type
Item _tit: Item Title
Item _aut: Item Author
Item _Are: Item Area
Se_str: Search String



Complexity

Method complexity	Average Complexity	Rule
25	**19.11**	**Rule (2)**

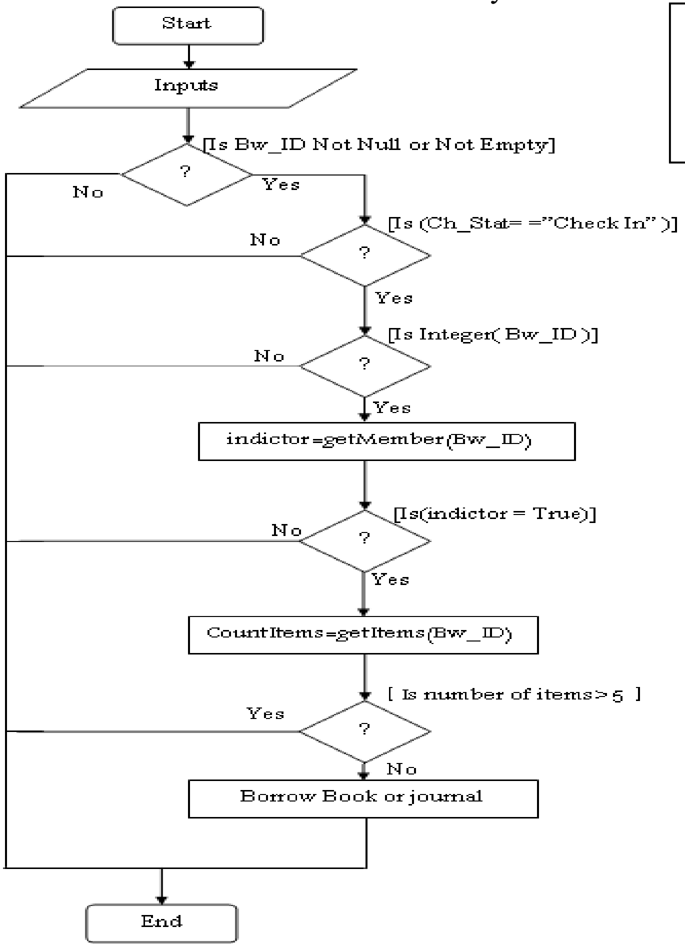
Start Time: End Time:	

Module: Search Item Frame	**Method: loan an Item**

Description:

This method allows member to borrow an item (book or journal) from the library system. After finishing search process a list of items are displayed for member. Member can borrow any item already exists in the library – (e.g. check in items, he/she can't borrow check-out items). Each member may borrow up to a maximum of 5 items for one month (30 days). This method checks the inputs and state of the item (check-in or check-out) so if item state is check-in procedure can proceed otherwise system must inform member that this item is already check-out. Then it validates whether member is allowed to borrow an item also it checks how many items member has already borrowed.

Item_ID: Item ID
Item _ty: Item Type
Bw_ID: member number
Ch Stat:Check State


Start
Inputs
[Is Bw_ID Not Null or Not Empty]
?
No
Yes
[Is (Ch_Stat= ="Check In")]
No
?
Yes
[Is Integer(Bw_ID)]
No
?
Yes
indictor=getMember(Bw_ID)
[Is(indictor = True)]
No
?
Yes
CountItems=getItems(Bw_ID)
[Is number of items> 5]
Yes
?
No
Borrow Book or journal
End

Complexity

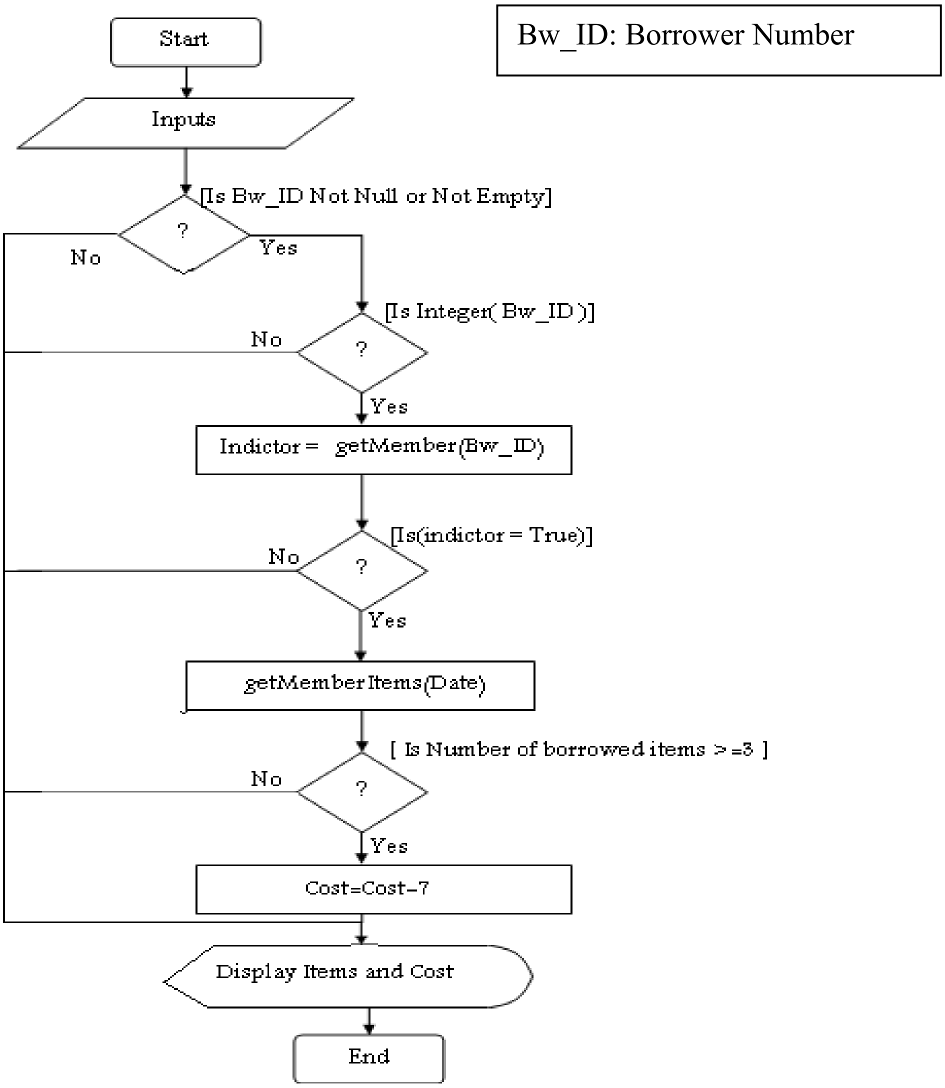
Method complexity	Average Complexity	Rule
31	**19.11**	**Rule (3)**

Start Time:
End Time:

Module: Search Item Frame	**Method: Items cost**

Description:

Member can borrow and/or renew items from library. Each of these activity has a cost in S.R. (borrow a book cost 10 S.R. while a journal only 5 S.R.); if the member performs at least 3 operation – i.e., borrow and/or renew in the same day, she/he receive a discount of 7 S.R.). So, this method checks user validity and also checks how many items he has borrowed then it computes the cost of his operations.



Complexit

Method complexity	Average Complexity	Rule
24	**19.11**	**Rule (2)**

User Setting Frame

Add user/ Update User

In this frame library employees that have manager privilege can insert add new user, update user data, and delete user from the system.

Start Time:
End Time:

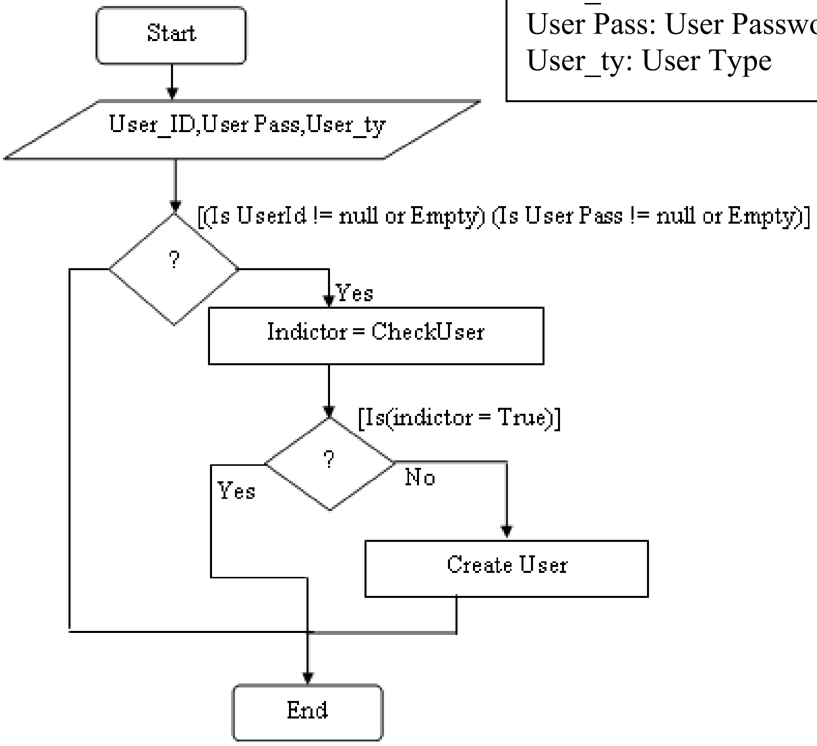
Module: User Setting Frame	**Method: add user**

Description:

This method allows a library employee that has manager privilege to add new user to the system.

In this method employee can grant the new user the privilege to be manager or user before create the new user it checks whether this user already exists or not.

User_ID: User Name
User Pass: User Password
User_ty: User Type


Start
User_ID,User Pass,User_ty
[(Is UserId != null or Empty) (Is User Pass != null or Empty)]
?
Yes
Indictor = CheckUser
[Is(indictor = True)]
?
Yes
No
Create User
End

Complexity

Method complexity	Average Complexity	Rule
17	**19.11**	**Rule (2)**

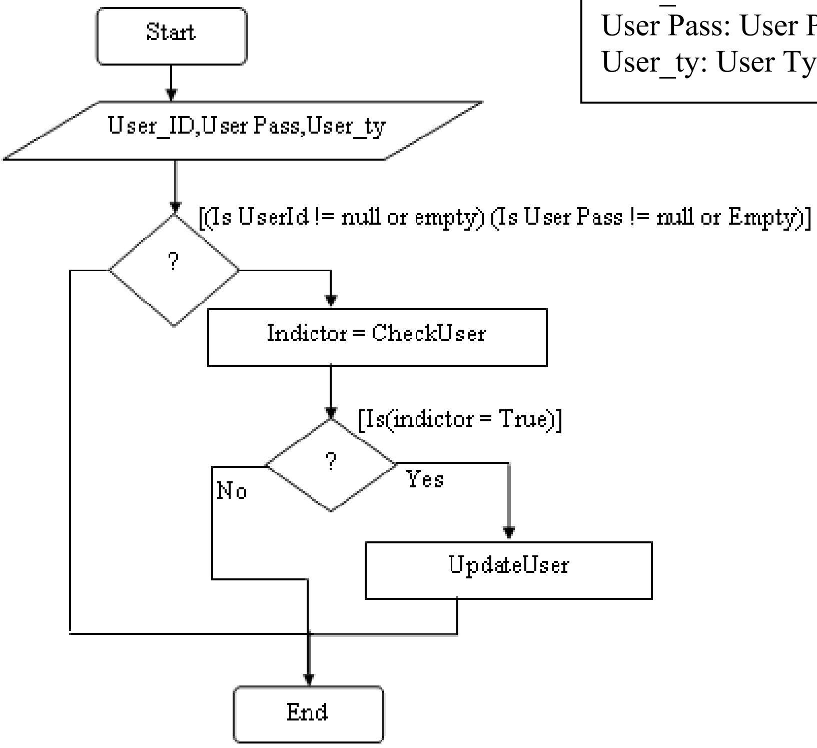
Start Time:
End Time:

Module: User Setting Frame	**Method: Updateuser**

Description:

This method allows library employee to update user record in the library system.

User_ID: User Name
User Pass: User Password
User_ty: User Type


Start
User_ID,User Pass,User_ty
[(Is UserId != null or empty) (Is User Pass != null or Empty)]
?
Indictor = CheckUser
[Is(indictor = True)]
?
No
Yes
UpdateUser
End

Complexity

Method complexity	Average Complexity	Rule
17	**19.11**	**Rule (2)**

APPENDIX C

Post Experiment Questionnaire

Name:

1. I had enough time to perform testing

☐ strongly agree ☐ agree ☐ not certain ☐ disagree ☐ strongly disagree

2. The objectives of the lab were clear to me.

☐ strongly agree ☐ agree ☐ not certain ☐ disagree ☐ strongly disagree

3. The description of the system was clear.

☐ strongly agree ☐ agree ☐ not certain ☐ disagree ☐ strongly disagree

4. The Control Flow Diagrams were clear to me.

☐ strongly agree ☐ agree ☐ not certain ☐ disagree ☐ strongly disagree

5. I experienced no difficulty in reading/understanding the Control Flow Diagrams

☐ strongly agree ☐ agree ☐ not certain ☐ disagree ☐ strongly disagree

6. I experienced no difficulty in reading/understanding the complexity numbers.

☐ strongly agree ☐ agree ☐ not certain ☐ disagree ☐ strongly disagree

7. Did you find complexity (when available) useful in testing?

☐ very much ☐ enough ☐ undecided ☐ little ☐ definitely not

Please explain the reason for your choice

Pre-Experiment Questionnaire

Name:

1. In which degree are you enrolling now?

☐ Bachelor ☐ Master ☐ PhD

2. Number of software projects with testing task (courses projects, industrial project)

☐ 5+ ☐ 3-4 ☐ 2-3 ☐ 1 ☐ Non

If more than one project please list the project name, course name and where you did it?

Project name	Course name	location

3. List software engineering courses you have studied (including this semester)?

4. Do you think you understand software testing?

☐ Completely ☐ well ☐ reasonably well ☐ not too sure ☐ not at all

5. Have you been involved in component-Based development (e.g. Using Java Beans, .Net, Net Beans, etc).

☐ Yes ☐ No

6. Have you previous experience of applying control flow testing?

☐ Yes ☐ No

Vita

- Fahmi Hassan Ali Qurada'a
- Nationality: Yemeni
- Born in Yemen on August 31, 1977.
- Completed Bachelor of Science (B.Sc.) in Computer Science from Thamar University, Yemen, in June 2001.
- Present address: King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia
- Permanent address: Yemen, Taiz; Email: qurada@yahoo.com