A FRAMEWORK FOR INTEGRATION SPECIFICATIONS FOR COMPONENT-BASED SOFTWARE

BY

MOHAMMED ABDULLAH ALI AL-QADHI

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, 2010

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **MOHAMMED ABDULLAH ALI AL-QADHI** 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

Dr. Sajjad Mahmood (Advisor)

Prof. Sabri Mahmoud (Member)

Dr. Mahmoud Elish (Member)

Dr. Kanaan Faisal
Department Chairman

Dr. Salam Adel Zummo
Dean of Graduate Studies

Date

Affectionately dedicated to
my parents
for their love, endless support
,and encouragement.

ACKNOWLEDGEMENT

In the name of Allah, the Most Beneficent, Most Merciful.

First of all, praise is due to Almighty Allah the source of all knowledge. Who in His Infinite Mercy and Grace enabled me to complete the present thesis. I bow my head with all submission and humility by way of gratitude due to Almighty Allah. May peace and blessing of Allah be upon the noble prophet Muhammad. May the mercy and forgiveness of Allah be upon the household of Rasul, his companions and all his fervent followers till the Day of Judgment.

I wish to express my heartfelt thanks and obligation to my advisor and mentor, Dr. Sajjad Mahmood who guided me with keen interest and rendered all possible help inspite of his hectic research and teaching schedules. I found no suitable words to thank him.

Thanks also are due to my thesis committee members Prof. Sabri Mahmoud and Dr. Mahmoud Elish for their invaluable comments and criticism.

I am also grateful to my family. My particular debt of gratitude is for my parents, my aunt, my brothers especially, Adnan, Ali, and my sister, for their encouragement, constant care, moral support, and love.

I also extend my thanks and gratitude to my friends: Amin Abo-monasar, Waleed Al-Zu'bi, Yousef Da'aboush, Abdirahman Daud, Emad Jaha, Basem Ala'alawi, Ali Yaseen,

Shouki Ebad, Abdulaziz Al-Baiz, Zaid Zuraigat, Mohammed Amro, Muaadh AbdulGhani, Ashraf AlShaikh, and Mohammed Yahia for their sharing of knowledge and experiences. I really had with them an unforgettable and wonderful time of my life.

Special thanks and gratitude to my friends: Mahmoud Al-Na'ami and Abdulaziz Shubaili who gave me continuous help, encouragement, and support throughout my life in KFUPM.

Also, I am thankful for KFUPM for providing the inspirational research atmosphere. The facilities provided by KFUPM were highly appreciated such as providing access to the high reputable journals.

TABLE OF CONTENTS

ACKNOWLEDGEMENT .... **iv**
TABLE OF CONTENTS .... **vi**
LIST OF FIGURES .... **viii**
LIST OF TABLES .... **x**
ABSTRACT .... **xi**
CHAPTER 1 .... **1**
INTRODUCTION **1**
1.1 Overview 1
1.2 Motivation 2
1.3 Aims of the Work 3
1.4 Contributions 4
1.4.1 Development of Glue Code Specification Framework 4
1.4.2 Applying the Framework to a Case Study 5
1.4.3 Glue Code Formal Specification 6
1.5 Thesis Organization 6
CHAPTER 2 .... **8**
BACKGROUND **8**
2.1 Introduction 8
2.2 Component-Based Software Engineering 8
2.3 Software Components 10
2.4 Component-Based Software Development 11
2.5 Components Integration 15
2.6 Adaptation (Glue Coding) 17
CHAPTER 3 .... **21**
LITERATURE REVIEW **21**
3.1 Overview 21
3.2 Individual Component's Point-of-View Approaches 21
3.3 Aspect-Oriented Approaches 24
CHAPTER 4 .... **32**
FRAMEWORK DEVELOPEMENT **32**
4.1 Introduction 32
4.2 Glue Code Specification 33
4.3 Deriving Glue Code Specification 34
4.3.1 Cheesman's Technique for Extracting System Interfaces 34
4.3.2 Conceptual and Concrete Interfaces 36
4.3.3 Deriving Conceptual Interfaces 39
4.4 The Glue Code Specification Framework 42
4.4.1 Components Documentation 42
4.4.2 Realization Stage 44
4.4.2.1 Static Realization 45
4.4.2.2 Dynamic Realization 50
4.5 Summary 56

CHAPTER 5 .... **58**
CASE STUDY FOR REALIZATION STAGE **58**
5.1 Introduction 58
5.2 Hotel Reservation System (HRS) 58
5.2.1 Applying Realization Stage 59
5.3 Summary 85
CHAPTER 6 .... **86**
GLUE CODE FORMAL SPECIFICATION **86**
6.1 Glue Code and Formal Specification 86
6.2 Object Constraint Language (OCL) 87
6.3 Composition Stage 92
6.3.1 OCL-Constrained Class Diagram (OCCD) 93
6.3.2 Validating OCCD 96
6.4 Summary 103
CHAPTER 7 .... **104**
CASE STUDY FOR COMPOSITION STAGE **104**
7.1 Applying Composition Stage 104
7.2 OCCD Validation 116
7.3 Summary 124
CHAPTER 8 .... **126**
CONCLUSION AND FUTURE WORK **126**
8.1 Summary of the Work 126
8.1.1 The Integration Framework 126
8.2 Future Research Work 128
REFERENCES **129**
APPENDIX **133**
CURRICULUM VITA **139**

LIST OF FIGURES

Figure 1: An overview for CBS development ... 1
Figure 2: An example of interfaces between two components represented in UML ... 11
Figure 3: CBS development process ... 12
Figure 4: An example of a glue code component ... 19
Figure 5: "*Make a Reservation*" UC of Hotel Reservation system. ... 35
Figure 6: Applying Chessman's technique to "*Make a Reservation*" UC ... 36
Figure 7: The generic notation for an interface ... 37
Figure 8: Conceptual interface for "*Make a Reservation*" UC ... 41
Figure 9: An overview for the glue code specification framework ... 42
Figure 10: An example of standard documentation for two components ... 43
Figure 11: An overview of the realization stage of the proposed framework ... 45
Figure 12: An example of a UCCM diagram ... 46
Figure 13: Note notation in UCCM diagram ... 46
Figure 14: CompBSD template ... 55
Figure 15: An example of scenario represented by a CompBSD ... 56
Figure 16: Selected software components to be integrated ... 59
Figure 17: UCCM diagram for *IMakingReservation* ... 63
Figure 18: CompBSD of *IMakingReservation* ... 66
Figure 19: UCCM diagram for *ICancelReservation* ... 69
Figure 20: CompBSD of *ICancelReservation* ... 71
Figure 21: UCCM diagram for *IAmendReservation* ... 74
Figure 22: CompBSD of *IAmendReservation* ... 76
Figure 23: UCCM diagram for *IProcessNoShows* ... 78
Figure 24: CompBSD of *IProcessNoShows* ... 80
Figure 25: UCCM diagram for *ITakingUpReservation* ... 82
Figure 26: CompBSD of *ITakingUpReservation* ... 84
Figure 27: Bank accounts class diagram ... 89
Figure 28: An overview of the composition stage of the proposed framework ... 93

Figure 29: An OCCD template 94
Figure 30: Class diagram for persons and a company 96
Figure 31: Generated object diagram for the model 99
Figure 32: The object model after creating the link *WorksFor* 101
Figure 33: UML sequence diagram representing sequence of operation calls 102
Figure 34: Skeleton of HRS OCCD 109
Figure 35: Attaching interfaces of the components to the OCCD skeleton 110
Figure 36: The glue component realizing "*ReservationOperations*" exposed interface 111
Figure 37: Building OCCD using ArgoUML (Tigris.org) tool 111
Figure 38 : Class diagram for HRS system 112
Figure 39: USE tool environment 117
Figure 40: A part of OCCD drawn using USE tool 117
Figure 41: Object diagram corresponding to classes in OCCD 118
Figure 42: Created objects during USE simulation for the system 119
Figure 43 : Call simulation for *askForReservation()* method 120
Figure 44: Another call simulation for *askForReservation()* method 120
Figure 45: Call simulation for *askToTakeUp()* method 121
Figure 46: Call simulation for *askToProcessNoShows()* method 121
Figure 47: Call simulation for *provideNameANDPostCode()* method 122
Figure 48: Call simulation for *getCustomerInfo()* method 122
Figure 49: Call simulation for *provideReservTag()* method 122
Figure 50: Call simulation for *getCustomerID()* method 123
Figure 51: Call simulation for *saveOldReservation ()* method 123
Figure 52: Sequence diagram for the performed calls simulations 124

LIST OF TABLES

Table 1: A summary of related work done in component integration.............................. 31

Table 2: The realization table for "*IMakingReservation*" conceptual interface 49

Table 3: Realization table for *IMakingReservation*.. 64

Table 4 : Realization table for *ICancelReservation*.. 70

Table 5: Realization table for *IAmendReservation* 75

Table 6: Realization table for *IProcessNoShows*.. 79

Table 7: Realization table for *ITakingUpReservation* 83

Table 8: List of missing functionalities 108

ABSTRACT

Full Name: Mohammed Abdullah Ali Al-Qadhi

Thesis Title: A Framework for Integration Specifications for Component-Based Software

Major Field: Information and Computer Science

Date of Degree: June, 2010

Component-Based Software (CBS) development process relies heavily on integrating individual components. Components are usually developed for general purposes and are integrated to meet the required functionality of the system-to-be. The integration code development is a complex and risk-prone process that needs to handle possible mismatches between the components' interfaces and implement missing functionalities. In this thesis, we present a framework that provides a process for deriving an integration specification for CBS. The integration specification is aimed to support missing functionalities, missing auxiliary services, mismatched interfaces, and flow of control. The framework consists of two stages, namely, realization and composition stages. The realization stage aims at specifying the mapping between the system-to-be use-cases and the selected components to identify the conceptual interfaces and the missing functionalities. The composition stage uses the Object Constraint Language (OCL) to add constraints to the integration specification. The framework will output two kinds of integration specifications. One will be in the form of Component-Based Sequence Diagrams (CompBSDs) resulting from the realization stage. The other one will be in the form of an OCL-Constrained Class Diagram (OCCD) resulting from the composition

stage. Furthermore, we present an application of the framework to a Hotel Reservation System (HRS) case study.

خلاصة الرسالة

الاسم: محمد عبدالله علي القاضي

عنوان الرسالة: إطار عمل لمواصفات التكامل للنظم البرمجية المبنية على المكونات البرمجية

مجال التخصص: علوم الحاسب و المعلومات

تاريخ التخرج: يونيو, 2009

تعتمد النظم البرمجية المبنية على مكونات برمجية بشكل كبيرعلى دمج مكونات برمجية منفردة. يتم تطوير هذه المكونات عادة لأغراض عامة و يجري تكاملها لتلبية الوظيفة المطلوبة من النظام البرمجي المزمع بناؤه. تعتبر الشفرة البرمجية التي تقوم بعملية التكامل شفرة معقدة ومعرضة للمخاطر التي تحتاج إلى معالجة حالات التباين المحتملة بين واجهات المكونات البرمجية وتنفيذ الوظائف المفقودة التي لا تقدمها المكونات الجاري تكاملها. في هذه الأطروحة ، نقدم إطار عمل لاستنباط مواصفات التكامل للبرمجيات القائمة على مكونات برمجية. هذه المواصفات سوف تقوم بتوفير الوظائف الأساسية الناقصة، و الوظائف الثانوية الناقصة، وحل التباين في مواجهات مكونات النظام، والمحافظة على سير عملية التحكم في النظام. وويتكون الإطار من مرحلتي التحقيق و الدمج. ففي مرحلة التحقيق نهدف إلى تحديد المواجهات الاولية و الوظائف المفقودة بناء على المعلومات المعطاة من حالات الإستخدام للنظام و الخدمات المقدمة من قبل المكونات البرمجية. أما في مرحلة الدمج فسوف نستخدم قيودا لغوية بواسطة لغة تقييد الكائن (OCL) لإضافة قيود على مواصفات التكامل. في إطار العمل سوف سيتم إستخراج نوعين من مواصفات التكامل. أحدهما سيكون على شكل مجموعة من مخططات التسلسل المبنية على المكونات البرمجية (CompBSDs) وهو ناتج عن مرحلة التحقيق. وسوف يكون الاخر على شكل مخطط أصناف مقيد (OCCD) وهو ناتج عن مرحلة الدمج. وعلاوة على ذلك ، فقد قمنا بتطبيق إطار العمل على نظام حجز الفنادق كدراسة حالة.

CHAPTER 1

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, complex software systems are built by integrating independent components. The reuse of software components greatly improves the software development productivity and also the quality of the final software product. An overview for Component-Based Software (CBS) development process is shown in figure 1.

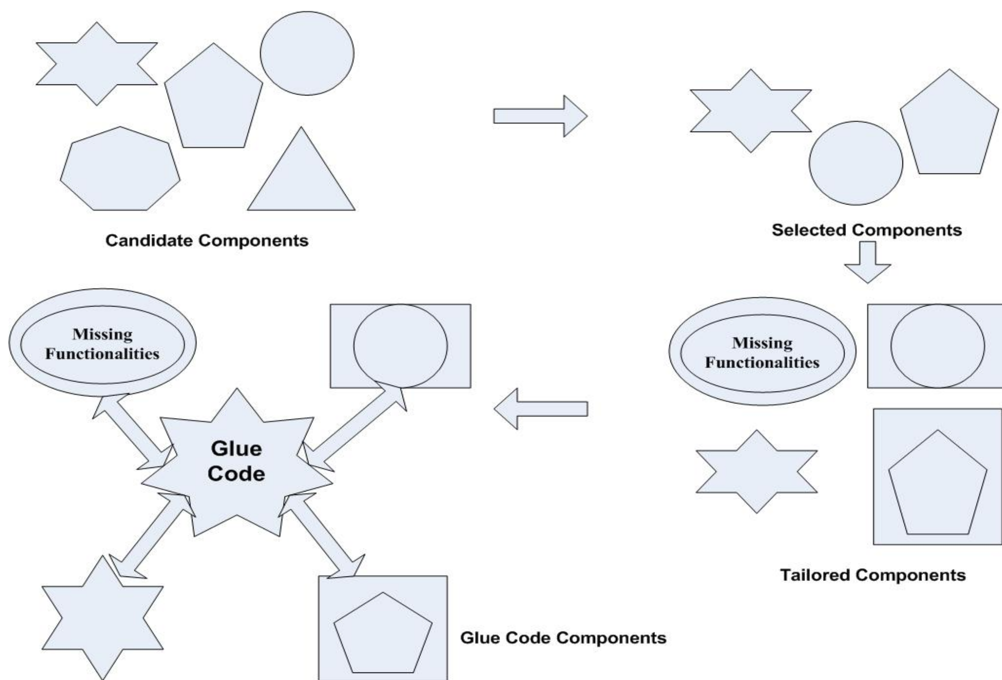


Figure 1: An overview for CBS development

1.2 Motivation

An important challenge associated with CBS integration phase is that, it is rarely the case that two components are perfectly matched. Hence, the process generally involves more than simply finding two components, which together perform the desired tasks, and then connecting their APIs (Baik, Eickelmann, & Abts, 2001). One important issue when integrating components is to deal with the interface mismatches that may occur when putting together pieces developed by different parties, usually unaware of each other (Cechich, Piattini, & Vallecillo, 2003). Also, some system functionalities required by stakeholders' requirements are not provided, so there is a need to implement them. There is also a need for a glue code specification in which mismatches between system-to-be requirements and functionalities are addressed.

(Basili & Boehm, 2001) and (Boehm & Abts, 1999) have shown that, if the components selected are not appropriate or an inappropriate model is used, or if the components are not well understood, the costs of glue code may be greater than that of the development of the components themselves (Crnkovic & Larsson, 2002). The effort for the development of glue code is usually less than 50% of the total from-scratch-development effort, but effort per line of glue code is about three times the effort per line of the application's code.

To the best of our knowledge, CBS integration (i.e. glue code) is still being developed in an ad hoc manner (Li, Conradi, Bunse, Torchiano, Slyngstad, & Morisio, 2009) which results in a brittle code (Baik, Eickelmann, & Abts, 2001). Developers rely on their on-

the-fly brainstorming when integrating components rather than a solid and well-defined engineering process. There is a need for a framework that supports the process of the glue coding in a CBS.

The progress of software development in the near future will depend very much on the successful establishment of CBS development, a point that is recognized by both industry and academia. So, we believe that this framework is important to further enhance the CBS development adoption in the industry.

1.3 Aims of the Work

The aim of this research work is to develop a framework for deriving glue code specification. This will help system analysts to (a) follow a clear sequence of steps for the derivation of glue code specification; (b) better overcoming the interface mismatches between components; and (c) better identification and implementation for missing functionalities.

The final specification is aimed to address the following issues:

- Missing primary functionalities: the final system may not meet the intended functional requirements.
- Missing auxiliary services: these are like type casting, exception handling, and temporary storage which may be needed when executing a scenario.

- Mismatched interfaces: wrong assumptions that a component makes about interfaces of other components which negatively affect the inter-operability.
- Flow of control: the lack of preserving scenarios sequences may result in exhibiting an undesirable behavior of the system's operation.

1.4 Contributions

The following subsections briefly list and summarize the contributions of this work.

1.4.1 Development of Glue Code Specification Framework

Components integration in CBS development is still being done in a traditional way (Li, Conradi, Bunse, Torchiano, Slyngstad, & Morisio, 2009). Use-cases of the system-to-be are gathered and the system design (if any) will be developed to be given to the developer. The developer has to wait till late stages of development to define system interfaces which will play an important role in the integration process.

As mentioned in the section of motivation, there is no well-defined engineering process for CBS integration rather it is done in an ad hoc manner. This may lead to mishandling many issues like: missing functionalities; mismatched interfaces; and flow of control when executing a scenario. This may result in a system that does not fulfill its requirements and\or exhibiting undesirable operation.

Moreover, the lack of documenting the CBS integration process will result in problems when maintaining the system. Maintenance is often performed by individuals who were not involved in the original design of the system being changed. Indeed, it is expected that many aspects of the system need to be understood in order to properly change it, including its functionality, architecture, and a myriad of design details.

In this research work, we developed a framework that will provide an engineering process for the components' integration in CBS development. It provides a clear derivation for the glue code specification that is going to connect the components rather than doing that in an ad hoc manner. It helps deriving the system interfaces directly from use-cases of the system-to-be instead of waiting till late development stages. It provides a means of documenting the integration process by providing a well-defined glue code specification.

1.4.2 Applying the Framework to a Case Study

We present an application of the integration framework to a case study of a Hotel Reservation System. This will give a step-by-step demonstration for the process of integration beginning with system-to-be use-cases till obtaining the final glue code specification.

1.4.3 Glue Code Formal Specification

Adding OCL formalism to glue code specification will provide us with a more precise model. This will help eliminate the problem of ambiguity in the UML model presented by glue code specification. The proposed framework provides using OCL as an option to ensure the correctness of the glue code specification. The framework can still work without using OCL and provide informal glue code specification.

1.5 Thesis Organization

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

- Chapter 2: gives a background about CBS development, Component integration, and Adaptation (Glue Coding) concepts.
- Chapter 3: provides an indicative literature review of CBS integration by presenting the research initiatives. It also shows strengths and weaknesses for each initiative.
- Chapter 4: provides a gradual demonstration of the development of the realization stage of the proposed framework till getting one form of glue code specification as output. It provides the definition of different concepts, processes, and notations involved in the stage.
- Chapter 5: presents a step-by-step application of the realization stage of the framework to a case study of a Hotel Reservation System (HRS).

- Chapter 6: presents the development of the composition stage of the framework which will provide a formal glue code specification as its output. Definition of different concepts, processes, and notations involved in the stage is provided.
- Chapter 7: shows an application of the composition stage of the framework to the output provided in chapter 5 and resulting from applying the realization stage to HRS case study.
- Chapter 8: briefly summarizes the framework as well as providing the potential strengths and weaknesses of the framework. Possible future work directions are also provided.

CHAPTER 2

BACKGROUND

2.1 Introduction

Component-Based Software Engineering (CBSE) emerged in the late 1990s as a reuse-based approach to software systems development. Its goal is to increase the productivity, quality, and time-to-market in software development. One important paradigm shift implied here is to build software systems from standard software components rather than "reinventing the wheel" each time.

Its creation was motivated by designers' frustration that object-oriented development had not led to extensive reuse, as originally suggested. Single object classes were too detailed and specific, and often had to be bound with an application at compile-time. You had to have a detailed knowledge of the classes to use them, which usually meant that you had to have the component source code. This made marketing objects as reusable components difficult. In spite of early optimistic predictions, no significant market for individual objects has ever developed.

2.2 Component-Based Software Engineering

Component-Based Software Engineering (CBSE) is the process of defining, implementing and integrating or composing loosely-coupled independent components

into systems. This has become an important software development approach because software systems are becoming larger and more complex and customers are demanding more dependable software that is developed more quickly. The only way that we can cope with this complexity and deliver better software more quickly is to reuse rather than re-implement software components.

The essentials of CBSE are (Sommerville, 2007):

1. Independent components completely specified by their interfaces. There should be a clear separation between the component interface and its implementation so that one implementation of a component can be replaced by another without changing the system.
2. Component standards that facilitate the integration of components. These standards are embodied in a component model and defined, at the very minimum, how component interfaces should be specified and how components communicate. Some models define interfaces that should be implemented by all conformant components. If components conform to standards, then their operation is independent of their programming language. Components written in different languages can be integrated into the same system.
3. Middleware provides software support for component integration. To make independent, distributed components work together, you need middleware support that handles component communications. Middleware such as CORBA handles low-level level issues efficiently and allows you to focus on application-related problems. In addition, middleware used to implement a component model may

provide support for resource allocation, transaction management, security and concurrency.

4. A development process geared to CBSE. If you try to add a component-based approach to a development process that is geared to original software production, you will find that the assumptions inherent in the process limit the potential of CBSE.

2.3 Software Components

There are many existing definitions for a software component in the literature. However, there is a general agreement in the software engineering community that a component is an independent software unit that can be composed with other components to create a software system.

Szyperski (Szyperski , Gruntz , & Murer, 2002) stated that “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” Szyperski also stated that a component has no externally observable state. This means that copies of components are indistinguishable. However, some component models, such as the Enterprise Java Beans model, allow stateful components so these clearly do not correspond with Szyperski’s definition of a component. While stateless components are certainly simpler to use, CBSE should accommodate both stateless and stateful components (Sommerville, 2007). In this research work, we adopt Szyperski’s definition.

Any software component consists of two parts: its interface and code. The interface is accessible and provides a means by which the components can talk to each other in order to exchange data and services. Interfaces should ideally contain all information about the component's operations and context dependencies. There are two types of interfaces: “Requires” and “Provides” type interfaces. A “Provides” interface defines the services provided by the component. A “Requires” interface specifies what services must be provided by other components in the system. If these are not available, then the components will not work. On the other hand, the code should be completely inaccessible (and invisible). The specification of a component therefore must consist of a precise definition of the component's operations and context dependencies (Crnkovic & Larsson, 2002). An example for two components is shown in figure 2.

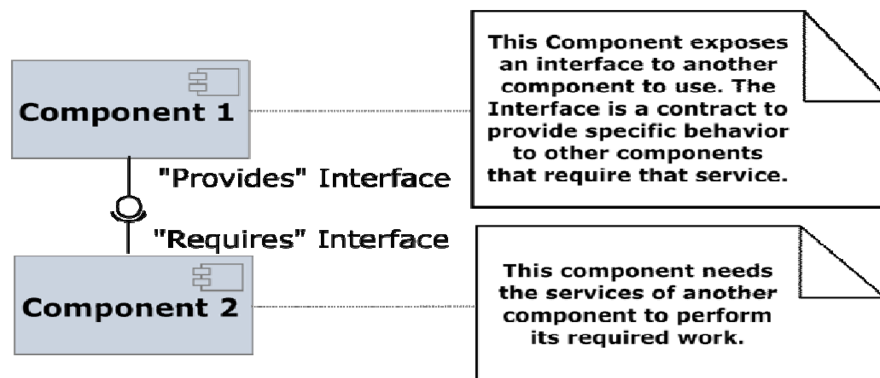


Figure 2: An example of interfaces between two components represented in UML

2.4 Component-Based Software Development

We are witnessing an enormous expansion in the reuse of software and an increasing adoption of CBS development approach by software development parties. So, software is

no longer built from scratch; instead they are built by assembling pre-existing components. This expansion is due to many factors like market competition, rapid delivery, and reduced development cost. A typical CBS development process consists of seven phases as shown in figure 3 (Crnkovic & Larsson, 2002):

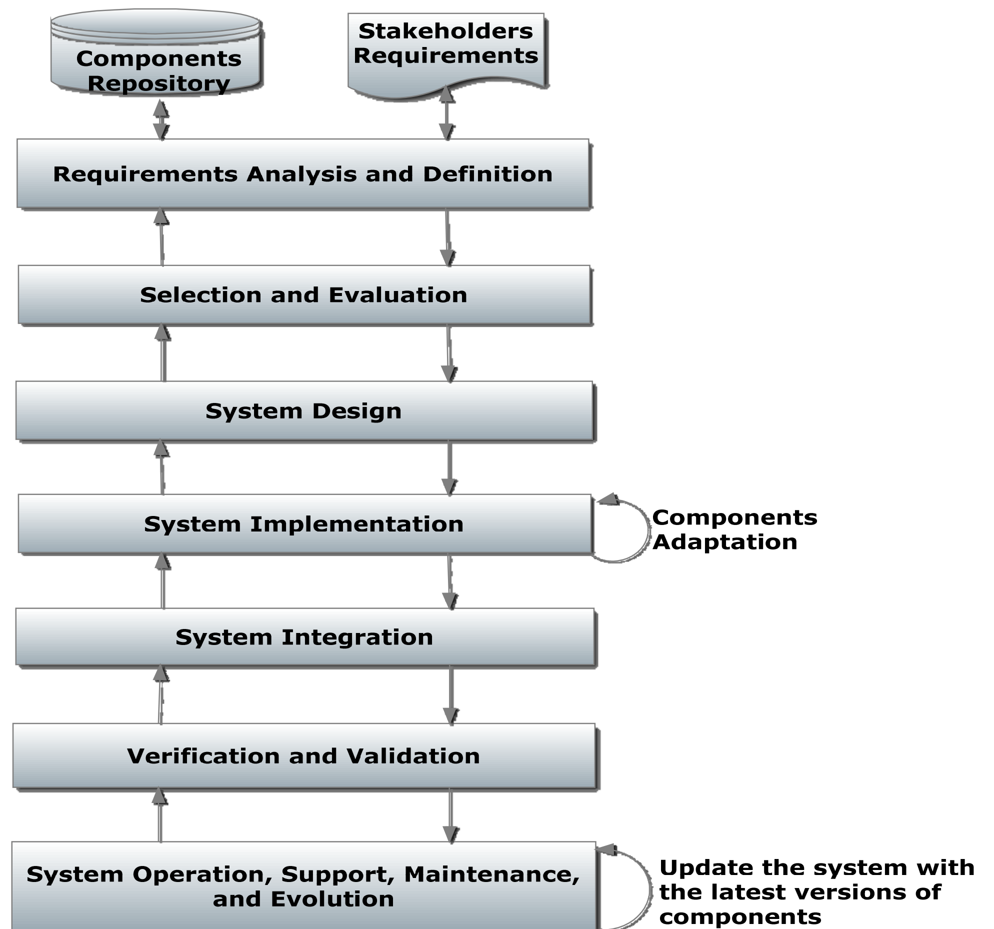


Figure 3: CBS development process

1) Requirements Analysis and Definition: The analysis activity involves identifying and describing the requirements to be satisfied by the system. In this activity, the system boundaries should be defined and clearly specified. Using a component-based approach,

an analysis will also include specifications of the components that are to collaborate to provide the system functionality. To be able to do this, the domain or system architecture that will permit component collaboration must be defined. In CBS development, the analysis is an activity with three tasks. The first task is the capture of the system requirements and the definition of the system boundaries. The second task is the definition of the system architecture to permit component collaboration, and the third task is the definition of component requirements to permit the selection or development of the required components.

2) Selection and Evaluation: To perform a search for suitable components and make their identification possible, the components must be specified, preferably in a standardized manner. Again, this may often not be the case. The component specifications will include precisely defined functional interfaces, while other attributes will be specified informally and imprecisely (no method is developed for this) if specified at all. The components selected must therefore be evaluated. The process of evaluation will include several aspects of both a technical and nontechnical nature. Technical aspects of evaluation include integration, validation, and verification. Examples of nontechnical issues include the marketing position of the component supplier, maintenance support provided, and alternative solutions.

3) System Design: System design activity typically begins with the system specification and the definition of the system architecture and continues from there. In traditional development, the design of the system architecture is the result of the system

requirements, and the design process continues with a set of sequences of refinements (for example, iterations) from the initial assumptions to the final design goal. In contrast with traditional development, many decisions related to the system design will be a consequence of the component model selected.

4) System Implementation: In an ideal CBS development process, the implementation by coding will be reduced to the creation of the "glue code" and to component adaptation. Also, it may still be necessary to design and implement some components (i.e. those that are business critical or unique to a specific solution and those that require refinement to fit into a given solution).

5) System Integration: Integration is the composition of the implemented and selected components to constitute the software system. The integration process should not require great resources, because it is based on the system architecture and the use of deployment standards defined by the component framework and by the communication standard for component collaboration. Moreover, one of the characteristics of many component-based systems is the ability to dynamically integrate components without interrupting system execution. This means that the integration activity in CBS development is present in several phases of the component-based system life cycle.

6) Verification and Validation: This last step before system delivery is similar to the corresponding procedures in a traditional development process. The system must be verified and validated. These terms can be easily confused although there is a clear

distinction between them. Verification is a process that determines whether the system meets its specified functional and nonfunctional requirements (i.e. are we building the product right?). A validation process should ensure that the system meets customer expectations (i.e. are we building the right product?).

7) System Operation, Support, Maintenance, and Evolution: The purpose of the operational support and maintenance of component-based systems is the same as that of monolithic, non-component-based systems, but the procedures might be different. One characteristic of component-based systems is the existence of components even at run time, which makes it possible to improve and maintain the system by updating components or by adding new components to the system. This makes faster and more flexible improvement possible (i.e. it is no longer necessary to rebuild a system to improve it). In a developed component market it also gives end users the opportunity to select components from different vendors. On the other hand, maintenance procedures can be more complicated, because it is not necessarily clear who is supporting the system (i.e. the system vendor or the component vendors). Moreover, CBS evolution involves providing the system with up-to-date versions of its constituent components to enhance its quality and performance.

2.5 Components Integration

Integrating components can be illustrated as a mechanical process of wiring components together. The key in integrating software components is to understand the components properties like their intended environment and the assumptions under which they were

developed. Any discrepancies must be handled in order to perform a successful integration. It is based on syntactic information such as method signatures and, when available, supplementary information supplied in a component's interface. Supplementary information will most likely include information such as a description of the function to be performed and types of exceptions thrown.

Usually, the components being integrated are developed independently, so there will be likely some interface mismatches (Garlan, Allen, & Ockerbloom, 1995). These are interface incompatibilities where the interfaces of the components that you wish to compose are not the same. Three types of incompatibility can occur (Sommerville, 2007):

1. Parameter incompatibility: The operations on each side of the interface have the same name but their parameter types or the number of parameters is different.
2. Operation incompatibility: The names of the operations in the “provides” and “requires” interfaces are different.
3. Operation incompleteness: The “provides” interface of a component is a subset of the “requires” interface of another component or vice versa.

Such interface mismatches will affect the interoperability between these integrated systems. A common representation of six major interoperability conflicts that arise through discrepancies or direct mismatches among architectural properties of interacting components has been defined by (Hepner, Gamble, Kelkar, & Davis, 2006).

The component integration process is composed of adaptation, validation and testing of the selected components.

1) **Adaptation (Glue Coding):** Here is the part where all architectural mismatches including those related to the interfaces will be solved by creating adapting programming statements called "glue code". Also, some missing system functionalities will be implemented within such code. Look at subsection 2.6 for further details.
2) **Validation and Testing:** Validating adapted components is a major task for the CBS development process. Software component testing and validation techniques focus on the expected behavior of the component to ensure that the exhibited behavior is correct. The internal structures of the components are usually unknown. The most appropriate technique for component testing and validation is black box testing. In addition to traditional software testing and validation techniques, CBD introduces a set of new challenges, for example, components must fit into the new environment and they often require real-time detection, diagnosis and handling of software faults (Mahmood, Li, & Kim, 2007).

2.6 Adaptation (Glue Coding)

Component adaptation is widely recognized to be one of the crucial problems in CBS development (Campbel, 1999) and (Heineman G. T., 1999), and it has been the subject of increasing attention in the last few years. The possibility for application builders to easily adapt off-the-shelf software components to properly work within their application is a

must for the creation of a true component marketplace and for component deployment in general (Brown & Wallnau, 1998).

Component adaptation is mainly concerned with solving mismatches between the integrated components. A component mismatch occurs, when a component, which implements a provided interface, and a component, which uses a required interface, is not cooperating as intended by the designer of the system. Using the concept of provided and required interfaces, a component mismatch can be interpreted as a mismatch between properties of required and provided interfaces, which have to be connected. Consequently, identifying mismatches between components is equivalent to identifying mismatches between interfaces.

As an example in which a glue code (a.k.a adapter) component may be used is where one component wishes to make use of another, but there is an incompatibility between the providers and requires interfaces of these components (Sommerville, 2007). This is illustrated in figure 4, where the data collector component is connected to a sensor component using an adapter. It reconciles the “requires” interfaces of the data collection component with the “provides” interfaces of the sensor component. The data collection component was designed with a generic “requires” mechanism that was not based on a specific sensor interface. It is anticipated that an adapter would always be used to connect the data collector to a specific sensor interface.

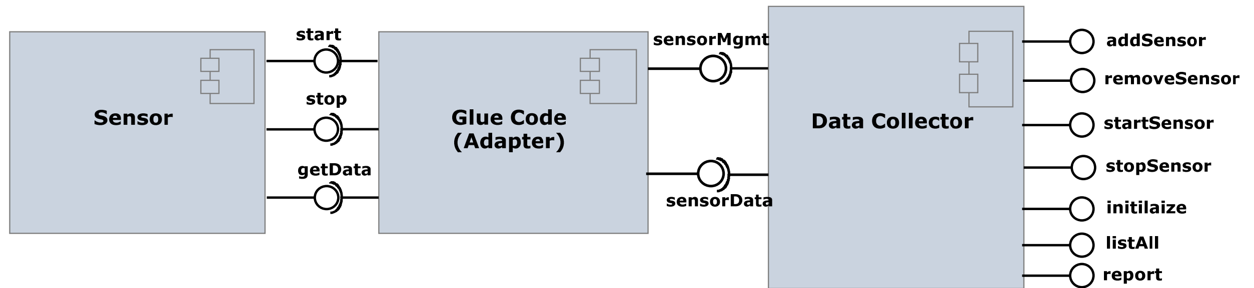


Figure 4: An example of a glue code component

Moreover, realistically speaking, usually it will not be the case that components being integrated are providing all functionalities that the final software product is expected to deliver. Therefore, a glue code also may contain the implementation of these missing functionalities.

There is no doubt that, a component is nothing but a software artifact. This means that we are performing a software adaptation. Software adaptation is the sequence of steps performed whenever a software entity is changed in order to comply with requirements emerging from the environment in which the entity is deployed. Such changes can be performed at different stages during the lifecycle. As a result, we can distinguish requirement adaptation, design-time adaptation, and run-time adaptation (Canal, Murillo , & Poizat, 2005):

- Requirement adaptation is used to react to changes during requirements engineering phase, especially when new requirements are emerging in the application domain.

- Design-time adaptation is applied during architectural design whenever an analysis of the system architecture indicates a mismatch between two constituent components.
- Run-time adaptation takes place when parts of the system offer different behavior depending on the context the parts are running in. This kind of adaptation is therefore closely related to context-aware systems.

In this research work, we are only concerned with the adaptation at the design-time.

CHAPTER 3

LITERATURE REVIEW

3.1 Overview

In CBS development, each individual component is developed based on its own requirements and context. It is usually necessary to create some sort of adaptation to reduce the conflict among the selected components. Different component integration mechanisms are presented from an individual component's point-of-view. Also, aspect-oriented integration proposals have been applied to component integration by handling crosscutting concerns.

3.2 Individual Component's Point-of-View Approaches

Filters are perhaps the oldest component integration mechanism (Garlan & Shaw, 1994). They only provide access to the data of the components without considering their functionality. The most complete mechanism provided by an individual component is an API that allows an external component complete access to all data, functions and events. An Internal Programming Language (IPL) is also used for component integration. It runs from inside the component in the form of macros (e.g. Microsoft Excel and Word). The concept of shared data repository is also discussed as a mechanism of component integration (Rader, 1997). This method is based on the idea that multiple components share a common data repository, while reading and writing the same data object.

The use of adapters to integrate components has been proposed by (Rine, Nada, & Jaber, 1999). Adapters are introduced to interconnect components containing interaction's protocol and manage component interactions. In this technique, each component has an associated adapter. Components request services from each other through their associated adapters. The associated adapters are responsible for solving the syntactic interfaces mismatch. The individual adapters communicate with each other to fulfill their components interaction.

An element of architecture for integration and rules that facilities the integration has been defined (Vigder & Dean, 1997). They have identified a set of component integration components that are required for the component integration. Wrappers isolate the underlying components from other components of the system while a glue code provides the functionality to combine the components and component tailoring (i.e. the ability to enhance the functionality of a component).

A language concept that facilitates the integration of components into applications is used to declare the type of components using the notation of collaboration interface (Mezini & Ostermann, 2002). Collaboration interfaces facilitate the bidirectional expression of potential contexts (i.e. client-from-server contract and server-from-client contract) in which they might be integrated. The decoupling of component implementation from binding via re-modularization allows to mix and match re-modularization and components on demand. The decoupling of components combined with integration of collaboration interfaces provides reuse in the proposed model.

(Dietrich, Patila, Sundermiera , & Urbana, 2006) have used active rules to design and generate wrappers to adapt components. The wrappers are automatically generated as enterprise Java Bean components and they act as proxy objects. These proxy objects intercept method calls and provide functionality required by the overall component-based system.

(Kouroshfar, Shahir, & Ramsin, 2009) have proposed a generic process framework for component-based development CBSDP consisting of four phases (each considered a phase process pattern): Analysis, Design, Provision, and Release. The integration takes place at the Release phase, where components are assembled together to form the system proper.

(Kim , Park, Yun, & Lee, 2008) had established an integration procedure which encourages the developers frequent and small releases. They also created an automated integration system which continuously runs an integration process in an incremental way so as to create and maintain an up-to-the-minute reasonably stable version of the product release candidate.

(Gomez, et al., 2008) proposed a platform independent model and architecture based on the component-based software composition paradigm. This architecture was intended to be used as a reference to develop integration applications through its deployment over a specific platform or technology.

(Canal, Poizat, & Salaun, 2008) have presented an approach for software adaptation which relies on an abstract notation based on synchronous vectors and transition systems for governing adaptation rules. Their proposal is supported by dedicated algorithms that generate automatically adapter protocols. These algorithms have been implemented in a tool.

(Zitouni, Seinturier, & Boufaida, 2008) present a contract-based approach to analyze and model the properties of components and their composition in order to detect and correct composition errors. With this approach they characterize the structural, interface and behavioral aspects, and a specific form of evolution of these components. Enabling this, they propose the use of the LOTOS language as an Architecture Description Language (ADL) for formalizing these aspects.

(Chi, 2009) has proposed software components composition compatibility checking based on behavior description. He defined the signature view and the behavior view of the software component then designed the modeling method that transfers the component behavior into a π calculus process expression and proposed the algorithm that makes the transfer automatically.

3.3 Aspect-Oriented Approaches

The concepts of aspect-oriented software development have also been incorporated into the CBS integration process. For example, (Assman, 2000) presents the concept of invasive composition and uses self-generated glue codes to integrate components.

Similarly, (Suvee, Vanderperren, & Jonckers, 2003) presents the JAsCo language for CBS integration. The language is designed to be used with Java Beans component model and introduces concepts of aspect beans and connectors. An aspect bean describes behavior that interferes with the execution of a component. Further, a connector is used to deploy the aspect beans in a given context.

(Batista, Chavez, Garcia, Kulesza, Sant'Anna, & Lucena, 2006) present the concept of the Aspectual Connector (AC), a special kind of architectural connector, as the only necessary enhancement to an ADL in order to support a seamless integration of AOSD and Software Architecture. They also present AspectualACME, an extension to ACME that incorporates ACs and additional facilities to modularize architectural crosscutting concerns.

(Kvale, Li , & Conradi , 2005) conducted a case-study to investigate whether AOP can help to build an easy-to-change COTS-based system. The case study was performed by comparing changeability between an object-oriented application and its aspect-oriented version. Results from this study have shown that integrating a COTS component using AOP may help to increase the changeability of the COTS component-based system, if the cross-cutting concerns in the glue code are homogenous (i.e., consistent application of the same or very similar policy in multiple places). Extracting heterogeneous or partial homogenous cross-cutting concerns in glue-code as aspects does not provide benefits.

(Lee & Bae, 2004.) have proposed an Aspect-Oriented Development Framework (AODF) in which functional behaviors are encapsulated in each component and connector, and particular non-functional requirements are flexibly tuned separately in the course of software composition. To support the modularity for non-functional requirements in component-based software systems, they devised Aspectual Composition Rules (ACR) and Aspectual Collaborative Composition Rule (ACCR). AODF makes component-based software built to provide both supports of modularity and manageability of non-functional requirements such as synchronization, performance, physical distribution, fault tolerance, atomic transaction, and so on. With the Collaboration-Based architectural style, AODF explicitly enables one to deal with nonfunctional requirements at the intra-component and inter-component levels.

(Truyen, J¨orgensen, Joosen, & Verbaeten, 2000) have examined a middle ground between aspect-oriented programming and computational reflection that improves the dynamics of this gluing process such that interaction between components can be refined at run-time. They have shown how this middle ground may be used to dynamically integrate into the architecture of the middleware systems some new services that support nonfunctional aspects such as security, transactions, and real-time.

Obviously, all initiatives done in component integration do not completely address the issues of missing functionalities, mismatched interfaces and preserving flow of control. We are addressing these issues in this research work. A summary of this literature review is shown in table 1.

Reference	Initiative	Strengths	Weakness
(Garlan & Shaw, 1994)	Proposing filters.	Easy understanding of the system's behavior as the composition of filters; they support reuse, easy to maintain and enhance; they support specialized analysis (throughput deadlock analysis); and they support concurrent execution.	Increased complexity; Lowered performance due to communication overhead; poor for interactive applications; and can be difficult to maintain synchronization between two related but separate streams.
(Vigder & Dean, 1997)	Presented wrappers and glue for integration and has defined rules that facilitate the integration.	More reliable software that can evolve over time.	May become as (or more) complex than the encapsulated element; Some undesirable functionality may exceed the wrapper's ability to buffer the system from the component (Hardy, 2000).
(Rine, Nada, & Jaber, 1999)	Proposed the use of adapters to integrate components.	Isolating and managing the components' interactions outside the components using adapters decrease components' complexity, increase their reusability, and eases their integration.	Adapters incur latency overhead and introduce performance penalty. The overhead stems from excessive data copying and non-optimized data structures for data buffering (Chiang & Ford, 2005).
(Mezini & Ostermann, 2002)	Proposed a language concept that facilitates the integration of components into applications using the notation of collaboration interface.	Decoupling of component implementation from bindings via remodularizations, to mix-and-match remodularizations and components on demand.	No support for reusable crosscutting concerns.

(Dietrich, Patila, Sundermiera , & Urbana, 2006)	They have used active rules to design and generate wrappers to adapt components based on EJB component model.	The metadata for the components to be integrated is used to adapt and enhance the black-box components through the use of wrappers; the process is an automated one.	Creating wrappers within the container proved to be challenging because EJB standard restricts components from creating new threads. Carrying a transactional context between containers also proved to be difficult.
(Kim , Park, Yun, & Lee, 2008)	They had established an integration procedure which encourages the developers' frequent and small releases in an incremental way.	Being an automated integration system; continuously runs integration process; create and maintain an up-to-minute reasonably stable version of the product release candidate.	Minimum unit is not a source code line but a package itself, so package maintainers are requested to have heavier responsibility than the developers in traditional way of software integration.
(Gomez, et al., 2008)	They proposed a platform independent model and architecture based on the component-based software composition paradigm.	Platform independent.	No attention has been paid to control and data flow issues during composition.
(Canal, Poizat, & Salaun, 2008)	They have presented an approach for software adaptation which relies on an abstract notation based on synchronous vectors and transition systems for governing adaptation rules.	Based on simple adaptation contract notation that could be used even to express correspondences (possibly involving mismatching messages) between complex adaptation scenarios.	It is global approach that is inapplicable when the system evolve, with components entering or leaving it at any time, e.g., for pervasive computing.

(Zitouni, Seinturier, & Boufaida, 2008)	They present a contract-based approach to analyze and model the properties of components and their composition in order to detect and correct composition errors.	Allows finding errors in the design composition early in the development process.	Being ADL-based integration make it restricted by ADL syntax.
(Chi, 2009)	He proposed software components composition compatibility Checking Based on Behavior Description. He defined the signature view and the behavior view of the software component then designed the modeling method based on π calculus.	Powerful in term of check component composition compatibility.	Difficulty in learning such method being mathematical formalized.
(Kouroshfar, Shahir, & Ramsin, 2009)	Proposed a generic process framework for component-based development CBSDP consists of four phases: Analysis, Design, Provision, and Release. The integration takes place at the Release phase, where components are assembled together to form the system proper.	It is a generic framework so; it can be used for tailoring process for CBS development.	More details for phase and stage levels but, no details have been mentioned for task patterns.
(Assman, 2000)	Presents the concept of invasive composition and uses self-generated glue codes to integrate components.	Being handling crosscutting concerns.	Not yet scalable to run time.

(Truyen, J¨orgensen, Joosen, & Verbaeten, 2000)	A middle ground between aspect-oriented programming and computational reflection that improves the dynamics of gluing process.	Defines application-specific global composition strategies that govern the gluing of aspect wrappers in a semantically correct way.	Being ad hoc.
(Suvee, Vanderperren, & Jonckers, 2003)	They present the JAsCo language for CBS integration. The language is designed to be used with Java Beans component model.	Introduces a component mode that incorporates the necessary traps to enable dynamic aspect application and removal. Also, component developers are still able to guarantee QoS for their components.	JAsCo is Java Beans component model in particular; its dynamicity and flexibility imposes a large performance overhead so, it is unsuitable in environments where resources are limited.
(Lee & Bae, 2004.)	They have proposed an Aspect-Oriented Development Framework (AODF) in which functional behaviors are encapsulated in each component and connector, and particular non-functional requirements are flexibly tuned separately in the course of software composition.	Supports modularity and manageability of non-functional requirements such as synchronization, performance, physical distribution, fault tolerance, atomic transaction, and so on.	Since that AODF depends on reflective architecture at runtime there has been neither a component writing standard nor a programming language which completely provides such reflective architecture at runtime, except for several Java extensions such as Javassist.

(Kvale, Li , & Conradi , 2005)	They have shown that integrating COTS component using AOP may help to increase the changeability of the COTS component-based system, if the cross-cutting concerns in the glue-code are homogenous.	When adding or replacing a COTS component, the main benefit of using AOP in COTS-based is that fewer classes need to be changed than using OOP.	It depends on whether glue code is homogenous or not. Using AOP when glue code is (partly) heterogeneous may not bring benefits. A careful analysis on cross-cutting concerns in the glue-code is therefore needed before the decision of using a certain COTS component.
(Batista, Chavez, Garcia, Kulesza, Sant'Anna, & Lucena, 2006)	They Presents the concept of the Aspectual Connector (AC), a special kind of architectural connector, and introduced AspectualACME ADL to support a seamless integration of AOSD and Software Architecture.	Avoid complexity by enriching the composition semantics supported by architectural connectors instead of introducing new abstractions.	Being ADL-based integration make it restricted by ADL syntax.

Table 1: A summary of related work done in component integration

CHAPTER 4

FRAMEWORK DEVELOPMENT

4.1 Introduction

In CBS development, like any other development paradigm, system requirements will be gathered in the form of use-cases (UCs). At the same time, we do have a set of selected components that provide a part of the functionality intended to be delivered by the system-to-be. The selected components are either COTS components or in-house developed ones. Generally, components will be encapsulated in such a way that the only revealed information is about their interfaces. This information includes the interfaces' methods signatures associated with a description about its functionality.

The glue code will be developed during the integration phase. Glue code will facilitate communication among components by handling possible interface mismatches between components. Also, components do not completely meet the desired functionality of the system-to-be. Missing functionalities that are not provided by the components will be developed in the glue code. Therefore, one can claim that glue code is the core element of the final system.

Prior knowledge about the system UCs is vital to develop the glue code for a CBS to meet the required system scenarios. At the same time, the information accompanying components will help the glue code developer understand which method of which

interface of which component needs to be called in with which scenario. A study has shown that companies still use traditional processes enriched with OTS-specific activities to integrate Off-The-Shelf (OTS) components (Li, Conradi, Bunse, Torchiano, Slyngstad, & Morisio, 2009). Also, it has been shown that some companies were immature and lacked a well-documented development process while using their ad hoc development processes. Also, it has been shown by (Vigder & Dean, 1997) and (Baik, Eickelmann, & Abts, 2001) that glue coding is done in an ad hoc manner.

4.2 Glue Code Specification

Intuitively, one can claim that a glue code is the code where all scenarios of the CBS are implemented. This is done by specifying the sequence of the methods to be invoked to implement a specific scenario. Ideally, the system developer will have the components specification and the development will be just a matter of plugging and playing them.

In reality, plug and play is difficult to achieve due to:

- Complete components specification might be missing.
- Components interfaces are incompatible to each other.
- Some of the required functionality might be missing altogether.
- Furthermore, some components provide additional functionality which are not required being in the system-to-be.
- Lack of a clear integration process results in on-the-fly decisions that may result in a brittle glue code (Baik, Eickelmann, & Abts, 2001).

Presence of a suitable specification to represent the glue code specification will help the developer in better implementing the system functionalities.

4.3 Deriving Glue Code Specification

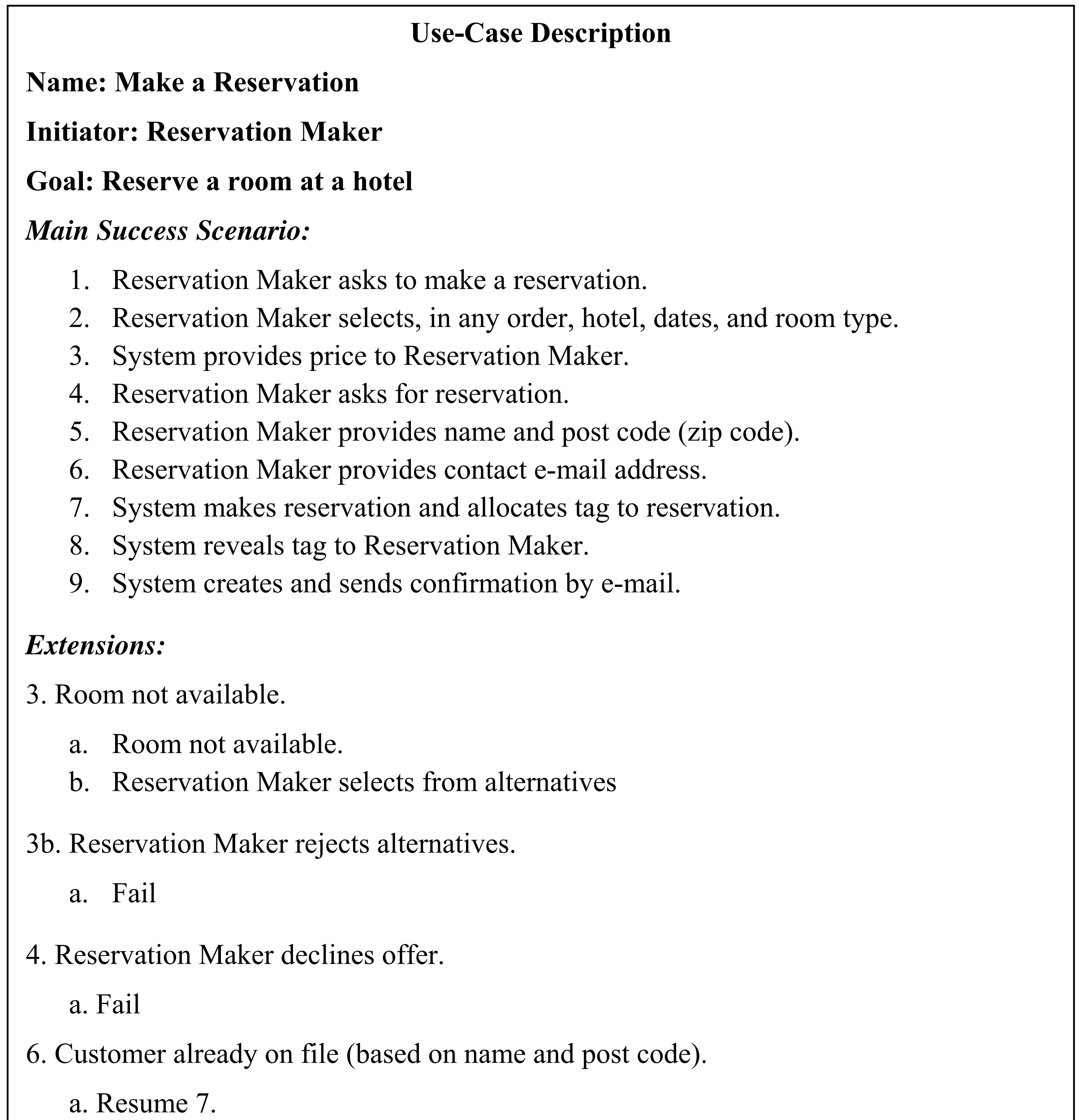
To handle all pre-mentioned issues, we find that there is a need to have an integration framework for a CBS. The framework is discussed as follows:

4.3.1 Cheesman's Technique for Extracting System Interfaces

In this technique, (Cheesman & Daniels , 2000) defined as a first-cut approach a dialogue type and one system interface per use-case. The technique works by going through each of the use-cases and for each step of a use-case one considers whether or not there are system responsibilities that must be modeled. If so, they are going to be represented as one or more operations of the appropriate system interface. This gives us an initial set of interfaces and operations to work from. If there are several consecutive use-case steps that are all system responsibilities, these can be collapsed into a single operation but this will not be done if these steps might later need to be split (e.g. by an extension). The example shown in figure 6 illustrates the technique by applying it to "*make reservation*" use-case of a hotel reservation system described in figure 5.

For "make reservation" use-case, we define an initial system interface called *IMakingReservation*. In the main success scenario, in step 2 we see that the system must allow the person making the reservation the ability to get details of different hotels, then

for a given selection provide (in step 3) a price and available ability for a given request. We'll call these the *getHotelDetails()* and *getRoomInfo()* operations. In step 7 we can deduce the need for a *makeReservation()* operation that creates a reservation given various details, returns a reference number, and confirms the reservation.



Use-Case Description

Name: Make a Reservation

Initiator: Reservation Maker

Goal: Reserve a room at a hotel

Main Success Scenario:

1. Reservation Maker asks to make a reservation.
2. Reservation Maker selects, in any order, hotel, dates, and room type.
3. System provides price to Reservation Maker.
4. Reservation Maker asks for reservation.
5. Reservation Maker provides name and post code (zip code).
6. Reservation Maker provides contact e-mail address.
7. System makes reservation and allocates tag to reservation.
8. System reveals tag to Reservation Maker.
9. System creates and sends confirmation by e-mail.

Extensions:

3. Room not available.

   a. Room not available.
   b. Reservation Maker selects from alternatives

3b. Reservation Maker rejects alternatives.

   a. Fail

4. Reservation Maker declines offer.

   a. Fail

6. Customer already on file (based on name and post code).

   a. Resume 7.

Figure 5: "*Make a Reservation*" UC of Hotel Reservation system.

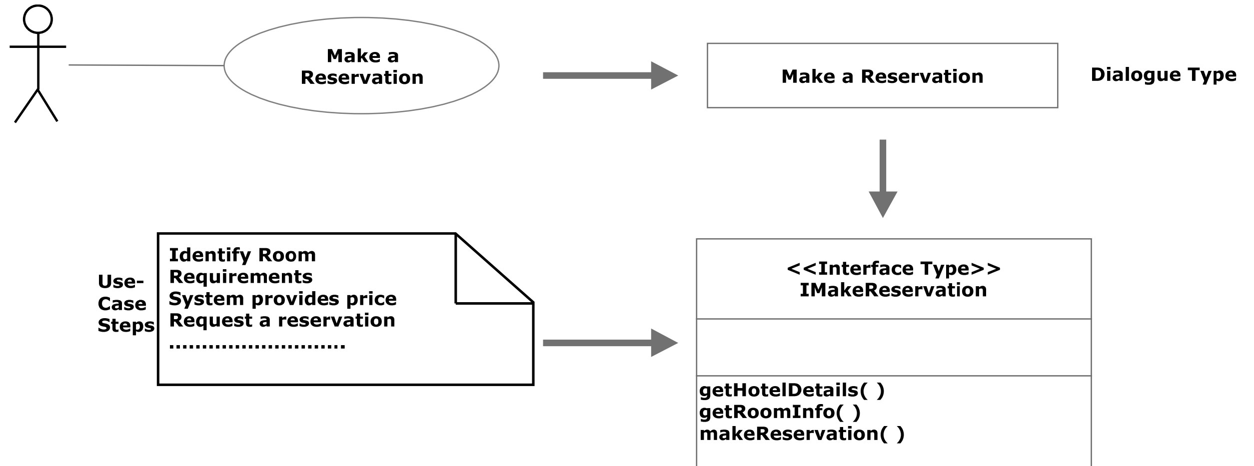


Figure 6: Applying Chessman's technique to "*Make a Reservation*" UC

The use-case extensions describe alternative behavior under certain situation. From the "room not available" extension we can see that the user may select alternative dates or room types. However, this is not an operation of the system (i.e. the display and selection of information will be handled by the user dialogue logic).

4.3.2 Conceptual and Concrete Interfaces

We built a framework that derives high-level interfaces from system use-cases to begin with by adapting Cheesman's technique. We are going to map the functionality provided by a use-case to a set of "Conceptual Interfaces". This approach is an alternative to waiting to approach late development stages to get system interfaces. We have added the modifier "conceptual" to the interface stereotype because, we are still talking about a high-level interface that does not exist in reality as a real component interface and we want to distinguish them from another type of interfaces we called "Concrete Interfaces".

For the sake of clarity, interfaces provided by the components are going to be called "Concrete Interfaces". The "concrete" modifier indicates the low-level of these interfaces as being real programming interfaces provided by classes constituting a software component. So far we have these two categories of interfaces:

I) ***Conceptual Interfaces:*** These are interfaces we came up with by deriving them from their corresponding use-cases of the system we are intending to develop. Such that, we convert each use-case into its corresponding interface.

II) ***Concrete Interfaces:*** These are the ones that are well-defined and exposed by the selected software components.

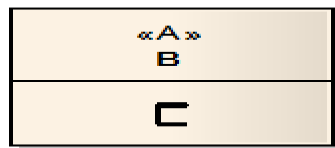


Figure 7: The generic notation for an interface

As shown in figure 7, an interface notation should contain two compartments with three parts as follows:

A. ***Stereotype of the Interface*** which will take one of these two stereotypes:

- <<Conceptual Interface Type>>: for conceptual interfaces.
- <<Concrete Interface Type>>: for concrete interfaces.

B. ***Name of the Interface:*** It is better to give a meaningful and easy-to-understand name for the interfaces but it will vary between conceptual and concrete as follows:

- Conceptual interface which only requires any given name of the interface

- Concrete interfaces which will take the following syntactical form:

Component Name* : *Interface Name

Interface Name indicates the very specific name for concrete interface participating in the scenario to fulfill the conceptual interface. On the other hand, ***Component Name*** indicates the component to which the concrete interface belongs. There is a separate colon between ***Component Name*** and ***Interface Name***.

C. ***Interface's Operation(s):*** Corresponds to the list of the operations providing services by the interface. The operation definition specification is common for both conceptual and concrete interfaces and consists of three parts that take the following syntactical form:

Operation Name (I/O Parameters): Return Type

The three parts are***:***

- ***Operation Name***: must be a meaningful one.
- ***I/O Parameters***: represents the input and output parameters to be passed during an interface scenario. They will be enclosed by two parentheses and separated by a comma. They should take one of two following syntactical forms:
    - ***(Data type Parameter1 Name, …,…,…..)***
    - ***(Parameter1 Name: Data type, …,…,…..)***

Parameter Name indicates the name to be used to make later one-to-one correspondence between the interface specification and the hidden internal implementation of the operation provided within the component to which the interface pertains. ***Data Type*** indicates the data type of the parameter. It may take the values of int, float, char…etc.

- ***Return type***: indicates the data type of the value retuned by the operation. It may take the values of: int, float, char…etc.

4.3.3 Deriving Conceptual Interfaces

We have to derive conceptual operations for each conceptual interface. Natural language plays a vital role to extract the conceptual operations for a conceptual interface. Linguistic analysis will be applied to this description with help of domain modeling approaches (Larman, 2004) as follows:

- Identify all actions within the description: by identifying the verbs in the main flow scenario and extensions. Identify only "real" actions by eliminating those that are just synonyms, repetition, etc. Also, weed-out actions that do not represent an interaction between the system and the actor like what appears in the following statement:

Guest arrives at hotel and claims the reservation

Here, we should consider only ***“claims the reservation”*** as the real action and discarding ***“arrives”*** action.

- Generalize the actions that can be generalized.

So, for our example, for “make a reservation” use-case we can see that from the use-case description, we may have the following list of “real” actions from the main success scenario and extensions underlined:

1. Reservation Maker asks to make a reservation.
2. Reservation Maker selects, in any order, hotel, dates, and room type.
3. System provides price to Reservation Maker.

4. Reservation Maker asks for reservation.
5. Reservation Maker provides name and post code (zip code).
6. Reservation Maker provides contact e-mail address.
7. System makes reservation and allocates tag to reservation.
8. System reveals tag to Reservation Maker.
9. System creates and sends confirmation by e-mail.
10. System will notify the billing system for the payment.

Also, extensions statements should be analyzed:

3. Room not available.

- Room not available.
- Reservation Maker selects from alternatives.

3b. Reservation Maker rejects alternatives.

a. Fail

4. Reservation Maker declines offer.

a. Fail

6. Customer already on file (based on name and post code).

a. Resume 7.

Then we can generalize the extracted real actions to get the final conceptual operations:

- askForReservation( ): to represent asks to make a reservation. This operation role is to initiate the whole scenario of making a reservation.
- selectReservation ( ): to represent selects, in any order, hotel, dates, and room type.
- providePrice( ): to represent provides price.
- getCustomerInfo( ): to represent provides name and post code (zip code) and provides contact e-mail address.

- reservation( ): to represent makes reservation and allocates tag to reservation and reveals tag to Reservation Maker and creates and sends confirmation by e-mail.
- refineReservDeatils( ): refining entered reservation details to provide more possible options with different timings and rooms.
- provideAlternatives( ): to represent Reservation Maker selects from alternatives when there is no room with the criteria entered by the customer.
- acceptOrReject( ): to represent both Reservation Maker rejects alternatives and Reservation Maker declines offer.
- failure( ): for issuing failure messages.
- notifyBillingSys( ): to represent System will notify the billing system for the payment.

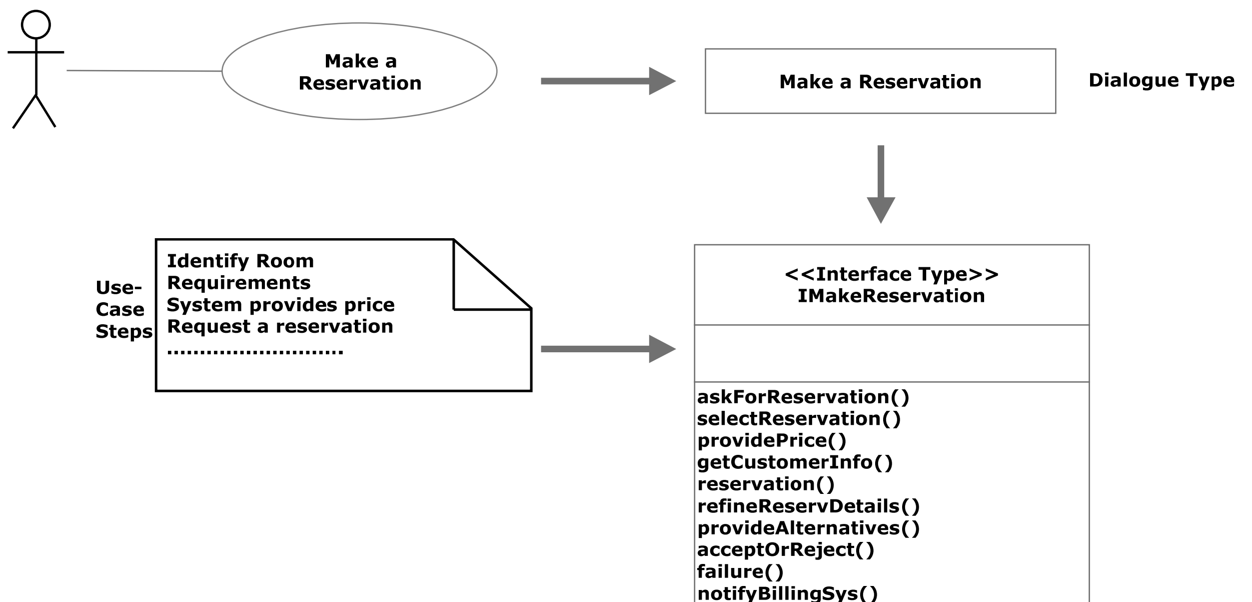


Figure 8: Conceptual interface for "*Make a Reservation*" UC

4.4. The Glue Code Specification Framework

The proposed framework constitutes two main stages: realization and composition, as shown in figure 9. It will lead us from the system use-case to the final specification of the glue code.

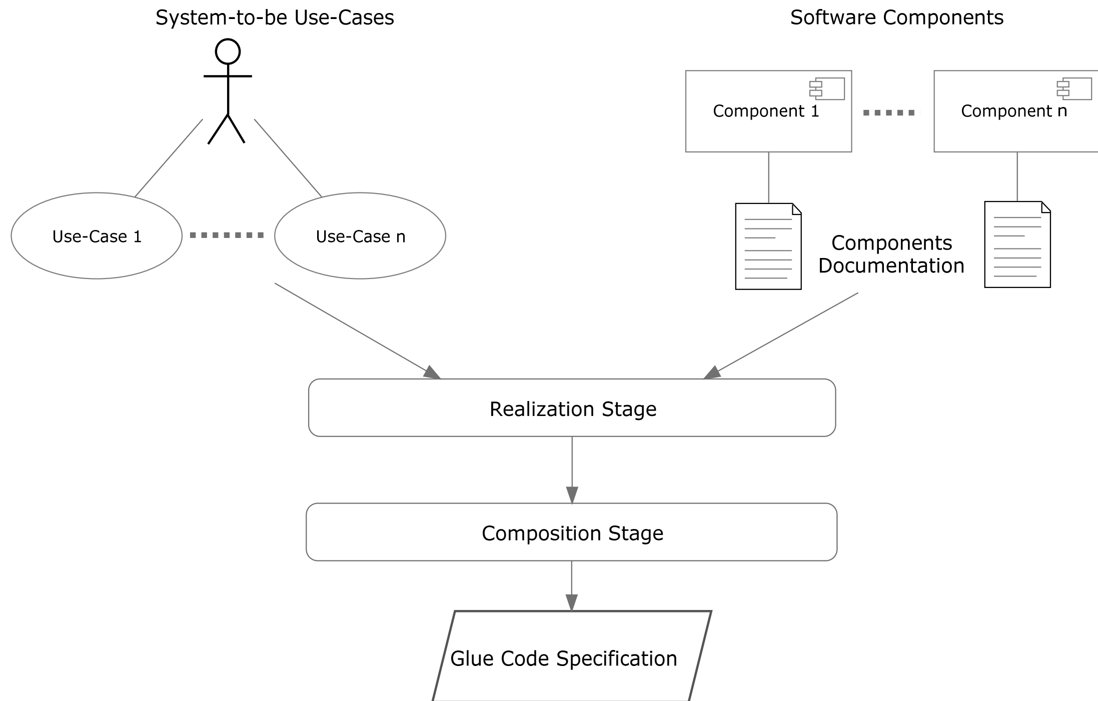


Figure 9: An overview for the glue code specification framework

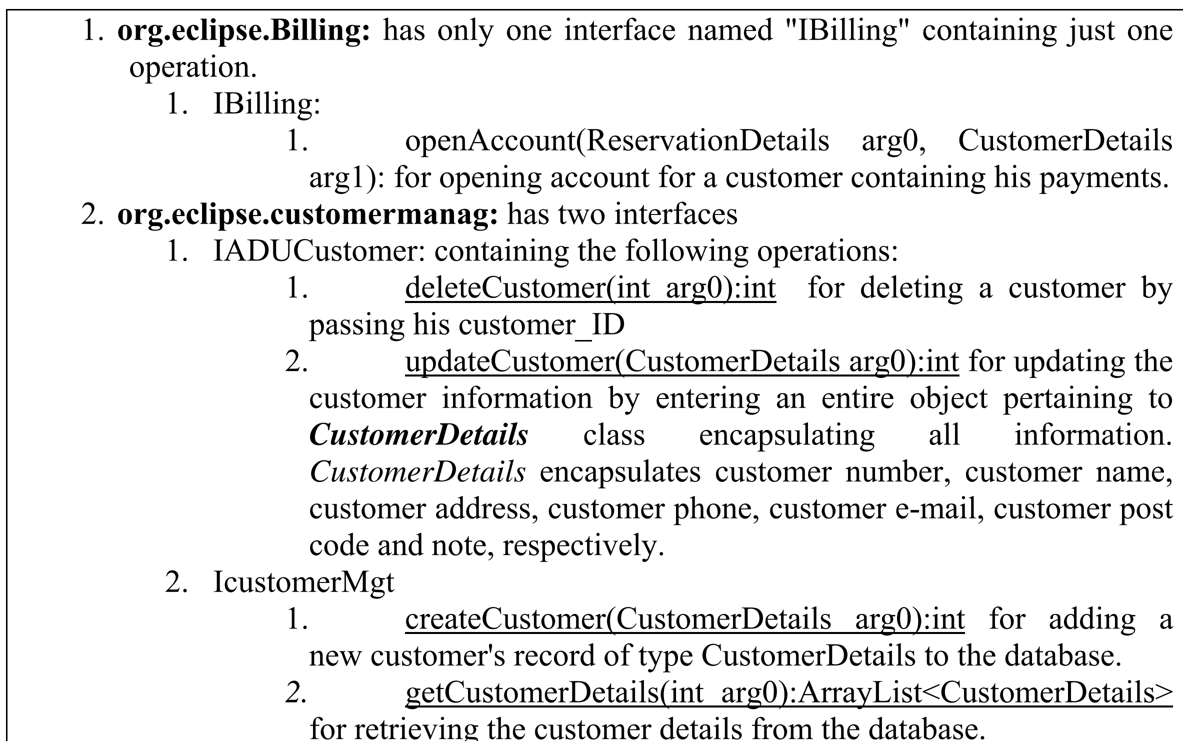
4.4.1 Components Documentation

We believe that components constituting a CBS adopting our integration framework should be associated with a documentation that meets the following conditions:

- It should provide the reader with all interfaces exposed by the component.
- It should provide an informative description of the functionality provided by each interface.
- It should explicitly contain the description about the complete set of the operations implemented by each interface.

- It should provide an informative description about each interface operation in terms of the following:
    - Operation's name.
    - Operation's signature: the method parameters that should be passed and the data type of each one of them.
    - Its return type.

For the sake of standardization and quickly referencing a concrete operation, we may put the components, their interfaces, and concrete operations in the form represented in the example provided by figure 10.



1. **org.eclipse.Billing:** has only one interface named "IBilling" containing just one operation.
    1. IBilling:
        1. openAccount(ReservationDetails arg0, CustomerDetails arg1): for opening account for a customer containing his payments.
2. **org.eclipse.customermanag:** has two interfaces
    1. IADUCustomer: containing the following operations:
        1. deleteCustomer(int arg0):int for deleting a customer by passing his customer_ID
        2. updateCustomer(CustomerDetails arg0):int for updating the customer information by entering an entire object pertaining to ***CustomerDetails*** class encapsulating all information. *CustomerDetails* encapsulates customer number, customer name, customer address, customer phone, customer e-mail, customer post code and note, respectively.
    2. IcustomerMgt
        1. createCustomer(CustomerDetails arg0):int for adding a new customer's record of type CustomerDetails to the database.
        2. getCustomerDetails(int arg0):ArrayList<CustomerDetails> for retrieving the customer details from the database.

Figure 10: An example of standard documentation for two components

This will provide us with a way to refer to each concrete operation with its code. ***Concrete Operation Code*** uniquely identifies a concrete operation. It takes the following form:

(ComponentName-InterfaceNumber-OperationNumber)

For instance, the code **(B-2-2)** means that this is the second operation of the second interface of the component named **B** which is ***getCustomerDetails(int arg0) in*** the description example in figure 10.

4.4.2 Realization Stage

Firstly, use-cases of the system intended to be developed and the documentation of the candidate components being integrated will be presented as inputs into the realization stage. Then, a use-case will be picked at time. Use-Case Conceptual Mapping (UCCM) diagram which is considered as static realization for the use-case is generated. Tabular form of UCCM diagram may be generated to help better viewing the interfaces involved in realization of the use-case. Then, Component-Based Sequence Diagram (CompBSD) will be generated to represent the dynamic realization of that use-case. Realization phase is shown in figure 11.

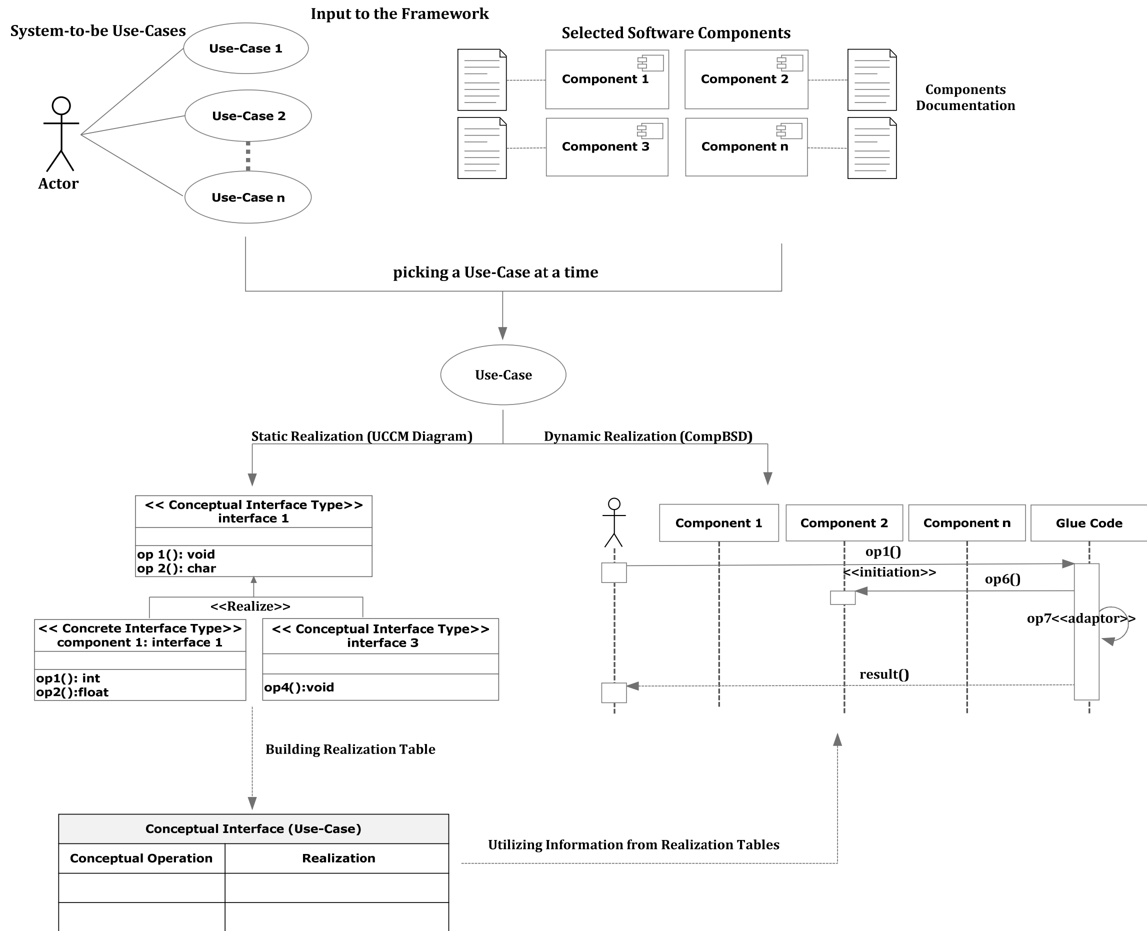


Figure 11: An overview of the realization stage of the proposed framework

4.4.2.1 Static Realization (Use-Case Conceptual Mapping)

We have referred to such a process as a static one because it does not describe "How" conceptual interfaces will be implemented (i.e., it does not look to the realization from a dynamic point-of-view). Rather, it describes the realization from a domain-model-like point-of-view but, in terms of concrete and\or conceptual interfaces instead of domain classes by means of a ***UCCM diagram*** and ***realization tables***.

UCCM diagram is one that shows all interfaces (i.e. concrete and conceptual interfaces) participating to fulfill a conceptual interface derived for a use-case of the system-to-be developed. An example of UCCM diagram is shown in figure 12. Obviously, we can see that the relationship between the conceptual interface and its underlying interfaces is stereotyped as "<<Realize>>".

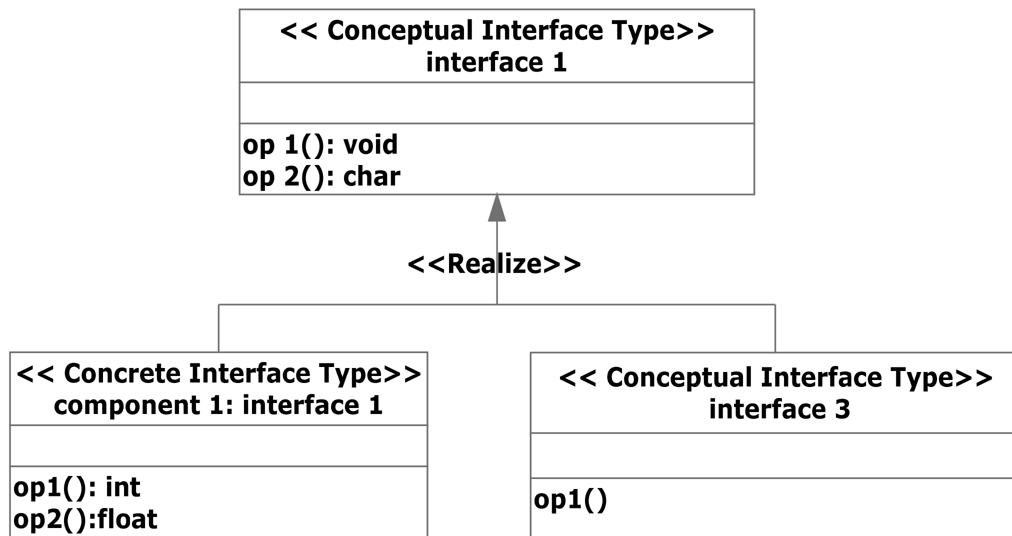


Figure 12: An example of a UCCM diagram

Also, notes may be added to the UCCM diagram components to provide illustrative comments. Such comments may include a brief of the functionality and relationship of the component. Notes notation will take the shape in figure 13.

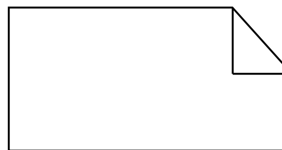


Figure 13: Note notation in UCCM diagram

UCCM process will help us to identify the following:

1) Which components may participate in realizing a use-case?
2) Which interfaces of which components are fulfilling the use-case interface that is being represented in the diagram.
3) The set of missing operations of a conceptual interface that are completely not fulfilled (i.e. realized) by the participating interfaces.
4) The set of operations of a conceptual interface that are partially fulfilled by the participating interfaces.
5) The additional interfaces of the involved components that have to be stored for further analysis.

When applying a Use-Case Conceptual Mapping (UCCM) process, we will have a system use-case model and the documentation of the components interfaces as inputs to the process. Then, the following steps should be followed to get a UCCM diagram for a use-case as an output.

1- Derive a corresponding conceptual interface for each use-case of the system-to-be.
2- Interface Involvement Identification Process (IIIP): lists all concrete and conceptual operations involved in fulfilling each operation for each conceptual interface. We have to write each concrete or conceptual interface associated with all its operations including even the operations which are not participating in the fulfillment.
3- Building a UCCM diagram for each conceptual Interface.
4- Generate what we called a ***"realization table"*** for each conceptual Interface.

Realization table is a tabular representation for each conceptual interface's UCCM diagram to realize each conceptual operation belonging to that interface by means of ***realization statements***. However, it contains a more detailed description about the realization of each conceptual operation belonging to the conceptual interface. Table 2 shows an example of the realization table realizing ***"IMakingReservation"*** conceptual interface. It contains an upper caption containing the name of the conceptual interface and two underlying fields: one for the conceptual operations belonging to it and another one for the realization statement for each conceptual operation.

Realization Statement is one that describes the execution flow of the concrete operations involved in realization of a conceptual operation. It is a combination of the concrete operation codes and\or conceptual operation names and relational operators: ***AND*** and ***OR***. Also, square brackets can provide a way of enclosing more than one operation. The execution flow will begin from the left-hand-side of the statement to right-hand-side and from inner to outsider brackets. For example, the following is a realization statement for the conceptual operation ***function( )***:

[(D-1-8) OR (E-3-8)] AND [(B-1-1) OR (E-1-1)]

This realization statement means that to fulfill ***function( )*** we should firstly, involve either operation coded as (D-1-8) or the one coded as (E-3-8). Then, we should involve either operation coded as (B-1-1) or the one coded as (E-1-1).

Sometimes, we may find that a conceptual interface is needed to be involved to fulfill a scenario. In other words, an entire system scenario is needed by another one. In this case, we should indicate that in the realization table by putting the name of the scenario involved. This scenario should be fully shown in the dynamic realization (i.e. in the CompBSD) by embedding all messages represented by the involved scenario's CompBSD within the CompBSD of the involving one.

IMakingReservation	
Conceptual Operations	**Realization**
askForReservation( )	Missing
selectReservation ( )	Missing
providePrice( )	Partially{(D-2-6) OR (E-5-3)}
getCustomerInfo( )	Missing
reservation( )	(A-1-1) AND [(D-2-8) OR (E-5-4)] AND [(B-2-5) OR (E-2-4)]
refineReservDetails( )	Missing
provideAlternatives( )	refineReservDetails( ) AND [(D-2-6) OR (E-5-3)]
acceptOrReject( )	Missing
failure()	Missing
notifyBillingSys( )	Missing

Table 2: The realization table for "*IMakingReservation*" conceptual interface

The realization table will be fed later into the process of dynamic realization. It will provide the system analyst with an initial clue about the concrete operations that may contribute to the fulfillment of a specific scenario to be represented, as well as, the initial flow of methods invocation to be applied.

4.4.2.2 Dynamic Realization (Component-Based Sequence Diagram)

We referred to this realization as a dynamic one because each conceptual interface that represents a system scenario will be represented as a specific-purpose sequence diagram showing the run-time sequence of actions. This diagram is called Component-Based Sequence Diagram (CompBSD). This stage reveals significant information that could not be caught at static realization like:

1. Missing functionalities: these are non-existing functionalities but are needed to provide the complete intended scenario that fulfills the overall functionality which should be provided by a conceptual interface. Even this has been taken into account in the static realization stage; this provides an additional assurance to avoid incomplete scenarios.
2. Data mismatch: this happens when a sequence of methods calls occurs and there is data-dependency between them. Such dependency may happen when, for example, an acquiring method needs the entire or part of the return value of another acquired method as argument to pass to it.
3. Execution precedence: checking the order of execution and method calls such that it matches the same execution flow pre-defined in the realization of the conceptual interface.

4. Auxiliary services: a scenario may need some libraries, header files, or temporary storage. For example, we can include a library containing the definition for some data types by importing it from the component to which it belongs at the beginning of a scenario.
5. Exception handling: there will arise some situations where we have to handle some potential situations that may lead to a system failure, for example, a scenario including division on zero.

In a CBS, glue code is the part of the code where all scenarios of the system-to-be will be implemented. Sequence diagrams (SDs) provide a means of representing system scenarios. However, a generic SD is restrictive for representing system scenario for CBS. The generic SD represents a scenario as an interaction between the objects of domain model classes while in the CBS the interaction will be between the integrated components. This means we need a sequence diagram that represents a scenario from a components point-of-view. Therefore we introduced what we called Component-Based Sequence Diagram (CompBSD).

This CBS-specific sequence diagram differs from the generic SD in the following aspects:

1) In CompBSD, lifelines are stereotyped with ***“<<Component>>”*** indicating that messages are sent between the constituent components’ lifelines while they are sent between the constituent classes’ objects lifelines in a generic SD.

2) The name of a lifeline should be the same name of the component which it represents as it appears in the component documentation.
3) Including a glue code component lifeline is a must.
4) In CompBSD, a message must be annotated with the name, signature, and the stereotype of the component method to be invoked by it.
5) In CompBSD, the first message is usually sent from the actor to the glue code component to trigger the sequence of messages implementing the scenario that it represents.

As mentioned, any message representing an invocation for a missing methods in a CompBSD must be annotated with a stereotype (i.e. invoked concrete operation will not be given a stereotype). In order to enrich the expressive power provided by CompBSD, we introduce here seven types of stereotypes for a method:

1) <<Initiation>>: a stereotype for methods, usually first sent messages, triggering scenarios.
2) <<Missing>>: some standalone functionalities while realizing a use-case.
3) <<LibraryImporting>>: for methods importing header files and built-in libraries from components.
4) <<Adapter>>: here an adapter code in the form of a method added to resolve an existing data mismatch. Such a method will remedy the mismatch through getting the designated return value of the acquired method and converting it to a compatible version to be passed to the acquiring method.
5) <<TemporaryStorage>>: for methods providing temporary variables to store long-life needed values (e.g. values to be used more than once at a scenario).

6) <<Utility>>: for methods which just provide input or output data from the user or a database.
7) <<ExceptionHandling>>: for a method providing the service of exception handling.

Building CompBSDs should be based on the information provided by use-case description and realization tables obtained from the UCCM process. A realization table provides a very beneficial starting point for knowing the methods calls and its sequence. When looking from a run-time perspective, one could find how necessary it is to implement some methods that are not provided by components within the glue code component.

The template for CompBSD is shown in figure 14. Messages are represented using arrows whose head is always targeting the component being meant for the invocation. Any missing method being invoked and annotated to a message must have the following syntactical form:

<<Stereotype>> Missing Method Name (List of Parameters): Return Type

- **Stereotype:** will take one of the above mentioned stereotypes based on the type of functionality that will be provided by the operation.
- **Missing Method Name:** should be a meaningful name given to the missing method.

- **List of Parameters**: list of parameters (if any) to be passed that may contribute to the final resulting computation. It will take one of two following syntactical forms:
    - ***(Data type Parameter1 Name, …,…,…..)***
    - ***(Parameter1 Name: Data type, …,…,…..)***

Parameter Name indicates the name to be used to make later one-to-one correspondence between the interface specification and the hidden internal implementation of the operation provided within the component to which the interface pertains. ***Data Type*** indicates the data type of the parameter. It may take the values of int, float, char…etc.

- **Return Type:** indicates the data type of the value retuned by the operation. It may take the values of: int, float, char…etc.

Any missing operation should be stereotyped and implemented later within the glue code. The first message always is a scenario initiating method with being stereotyped with "Initiating". Messages invoking a concrete operation\method from a specific component other than glue code will have the same syntactic form except being not stereotyped and the name begins with the operation code as follows:

Concrete Operation Code) Concrete Method Name (List of Parameters): Return Type

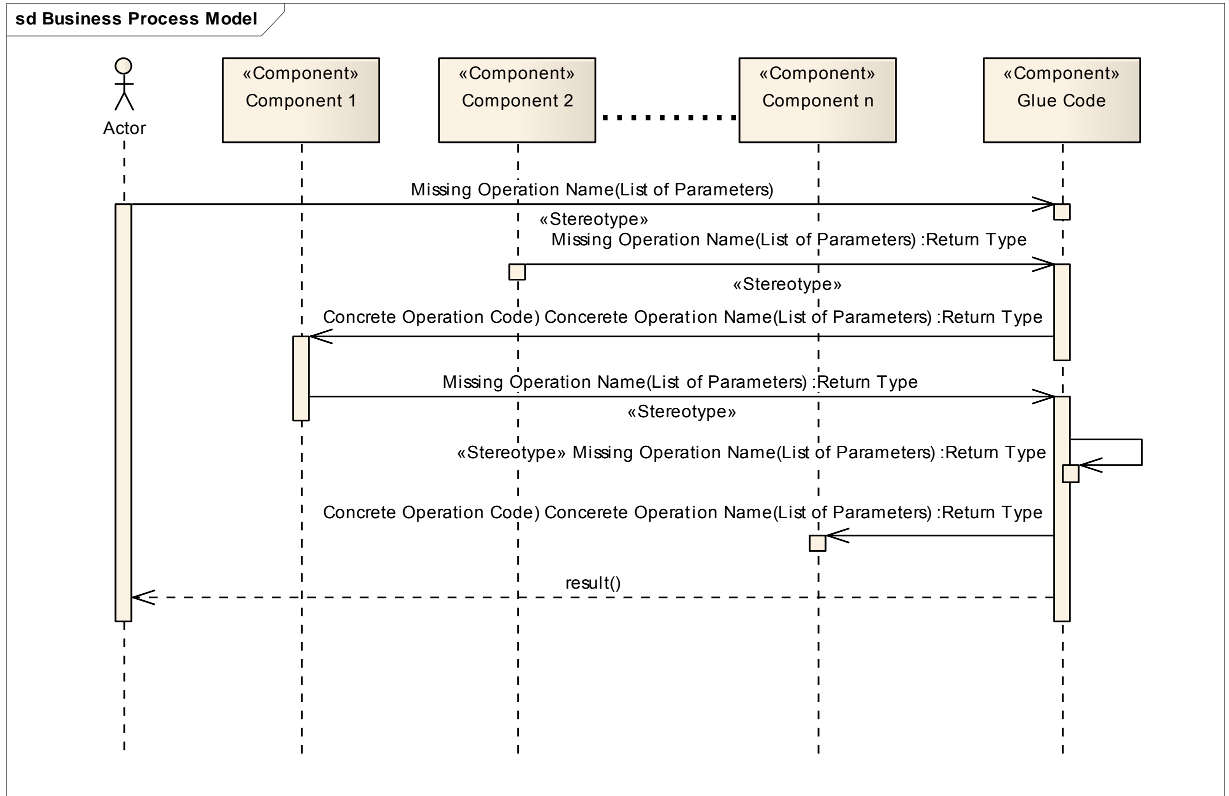


Figure 14: CompBSD template

As an example, let's take the scenario represented by CompBSD shown in figure 15. First, the actor will trigger the scenario by invoking missing method *op1( )* with being stereotyped with "Initiation" and of "void" return type. Then, the missing operation *op2( )* returning a value of "string" data type will be invoked. *op2( )* performs a standalone functionality applied to *Component 2* so, it is stereotyped with "Missing". After that, *op3( )* missing operation being returning an integer value will be invoked . *op3( )* is being stereotyped with "Adapter" indicating that it is doing a data mismatch resolving functionality needed for some data to be used by the scenario. Afterward, *1-2-5)op5( )* will be invoked from *Component 1*. From the concrete operation code indicated in the beginning of the name of the methods we can conclude that it is the

method numbered 5 implemented by the component numbered 1 existing in its implemented interface numbered 2 in the components documentation that meets our convention that has been mentioned earlier.

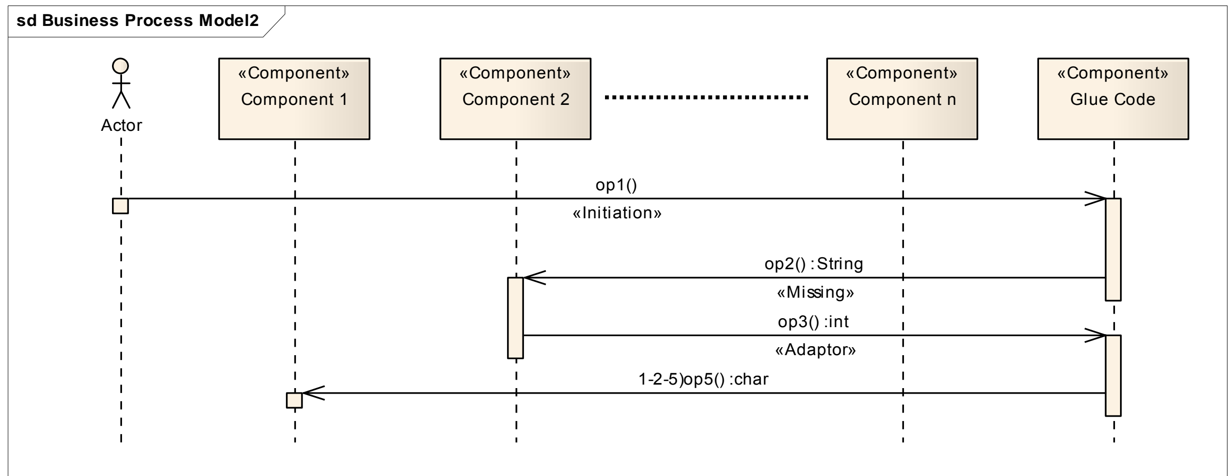


Figure 15: An example of scenario represented by a CompBSD

4.5 Summary

In this chapter, we have developed the realization stage of our integration framework. We have introduced concepts of static and dynamic realization. UCCM and CompBSD diagrams, as well as, realization table have been well-defined in terms of their definitions, notations, and processes for deriving them. Examples have been given for all of the introduced concepts.

We have demonstrated how we can derive the conceptual interfaces directly from UCs of the system-to-be by applying the UCCM process to them. This approach provides us with an initial set of interfaces of the system to begin with. This will help getting rid of the

burden of waiting till the coding phase to derive them. Also, we have shown how we can apply the dynamic realization process to the same set of UCs with help of realization tables obtained from the UCCM process to derive CompBSDs.

CHAPTER 5

CASE STUDY FOR REALIZATION STAGE

5.1 Introduction

In this chapter, the realization stage of the proposed integration specification framework will be applied to a case study. An application of it to a case study of Hotel Reservation System (HRS) will be presented (Cheesman & Daniels , 2000). This helps to give the reader a better clue about the framework. It also helps to validate the framework and shows its usefulness and success, as well.

5.2 Hotel Reservation System (HRS)

This system provides the automated support for a hotel management and staff. It helps managing the reservations-related services like: making reservations, cancelling reservations, amending reservations, process no-show reservations, and taking up reservations.

Here we have group of five selected software components to be integrated. All of them are implemented in Java in form of plug-ins as shown in figure 16.

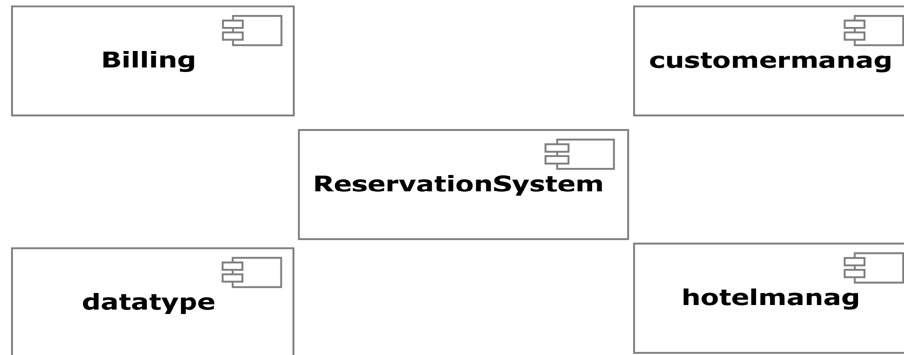


Figure 16: Selected software components to be integrated

5.2.1 Applying Realization Stage

We will build here a conceptual interface for each use case. This means we will have five conceptual interfaces corresponding to the system-to-be use-cases as follows:

1) **IMakingReservation**
2) **ICancelReservation**
3) **IAmendReservation**
4) **IProcessNoShows**
5) **ITakingUpReservation**

Let's apply static and dynamic realizations processes for each of the five conceptual interfaces individually:

1) IMakingReservation

Static Realization:

From the use-case description, we may have the following list of "real" actions from the main success scenario and extensions underlined:

1. Reservation Maker asks to make a reservation.

2. Reservation Maker selects, in any order, hotel, dates, and room type.

3. System provides price to Reservation Maker.

4. Reservation Maker asks for reservation.

5. Reservation Maker provides name and post code (zip code).

6. Reservation Maker provides contact e-mail address.

7. System makes reservation and allocates tag to reservation.

8. System reveals tag to Reservation Maker.

9. System creates and sends confirmation by e-mail.

10. System will notify the billing system for the payment.

Also, extensions statements should be analyzed:

3. Room not available.

   a. Room not available.
   b. Reservation Maker selects from alternatives

3b. Reservation Maker rejects alternatives.

   a. Fail

4. Reservation Maker declines offer.

   a. Fail

6. Customer already on file (based on name and post code).

   a. Resume 7.

Then, we can generalize the extracted real actions to get the final conceptual operations:

- askForReservation( ): to represent asks to make a reservation. This operation role is to initiate the whole scenario of making a reservation.

- selectReservation( ): to represent selects, in any order, hotel, dates, and room type.

- providePrice( ): to represent provides price.
- getCustomerInfo( ): to represent provides name and post code (zip code) and provides contact e-mail address
- reservation( ): to represent makes reservation and allocates tag to reservation and reveals tag to Reservation Maker and creates and sends confirmation by e-mail.
- refineReservDeatils( ): refining entered reservation details to provide more possible options with different timings and rooms.
- provideAlternatives( ): to represent Reservation Maker selects from alternatives when there is no room with the criteria entered by the customer.
- acceptOrReject( ): to represent both Reservation Maker rejects alternatives and Reservation Maker declines offer.
- failure( ): for issuing failure messages.
- notifyBillingSys( ): to represent System will notify the billing system for the payment.

Then, we are going to apply IIIP process to find out which concrete and/or conceptual operation(s) may contribute to fulfill each conceptual operation within each conceptual interface. Each interface contributing to the fulfillment should be shown in the UCCM diagram. Matching between concrete and conceptual operations should be based on the matching between the functionality that should be provided by the conceptual operation and the functionality provided by concrete operations as described by the components' documentation.

Let's here apply IIIP to conceptual operations of **IMakingReservation** one by one:

- **askForReservation( )**: By going through the component documenetation, we can see there are no cocncrete operations that are providing either complete or partial fulfillment. So, it is considered as a fully missing functioanlity that should be implemenetd within the glue code.
- **selectReservation ( ):** Also, it is considered as a fully missing functionality that should be implemenetd within the glue code.
- **providePrice( ):** we can clearly see that there are two concrete operations coded (D-2-6) or (E-5-3) that can partially provide the functionality.
- **getCustomerInfo( ):** it is considered as a fully missing functionality that should be implemenetd within the glue code.
- **reservation( ):** There is a sequence of concrete operations that should apply to provide the final functionality. The concrete operations that could be involved are: (A-1-1), (D-2-8), (E-5-4), (B-2-5), and (E-2-4).
- **refineReservDeatils( ):** it is missing.
- **provideAlternatives( ):** it is considered as a fully missing functionality that should be implemenetd within the glue code.
- **acceptOrReject( ):** it is considered as a fully missing functionality that should be implemented within the glue code.
- **failure( ):** missing.
- **notifyBillingSys( ):** missing.

Then, we get the UCCM diagram in figure 17 and realization table shown in table 3.

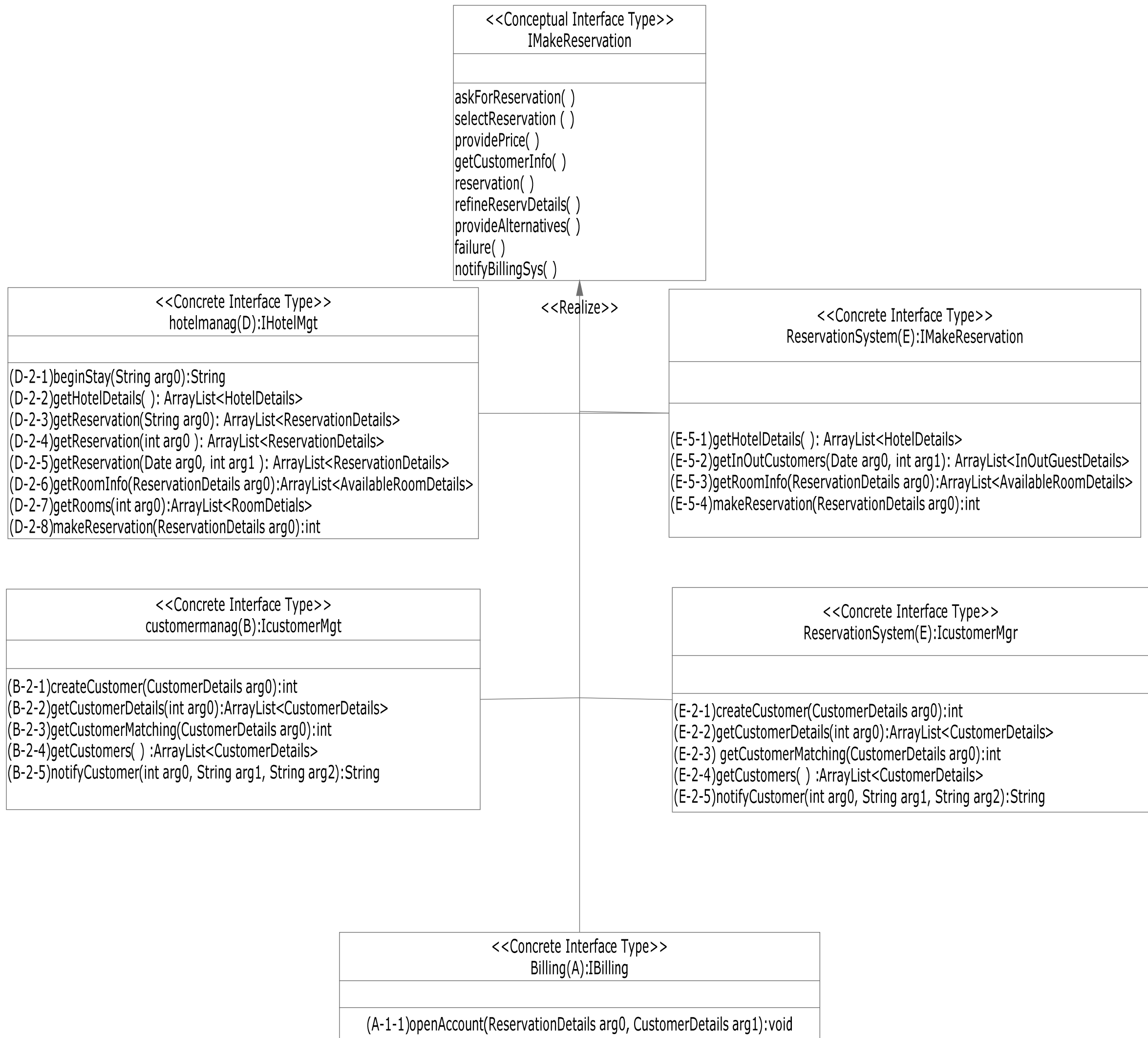


Figure 17: UCCM diagram for *IMakingReservation*

IMakingReservation	
Conceptual Operations	**Realization**
askForReservation( )	Missing
selectReservation ( )	Missing
providePrice( )	Partially{(D-2-6) OR (E-5-3)}
getCustomerInfo( )	Missing
reservation( )	(A-1-1) AND [(D-2-8) OR (E-5-4)] AND [(B-2-5) OR (E-2-4)]
refineReservDetails( )	Missing
provideAlternatives( )	refineReservDetails( ) AND [(D-2-6) OR (E-5-3)]
acceptOrReject( )	Missing
failure()	Missing
notifyBillingSys( )	Missing

Table 3: Realization table for *IMakingReservation*

Dynamic Realization:

Here we are going to build a CompBSD corresponding to **IMakingReservation**. We have a realization table with an initial set of conceptual operations associated with their realization statements. Figure 18 shows the CompBSD of **IMakingReservation.** Missing conceptual operations will be added as complete methods to be implemented in the glue code components whereas; realized ones are substituted with their realizing concrete operations when representing the scenario in the CompBSD. So, for instance, *reservation( )* operation has been substituted with its realizing cocncrete operations. However, *askForReservation( )* operation has been mentioned in the CompBSD with being stereotyped with <<Initaition>> stereotype.

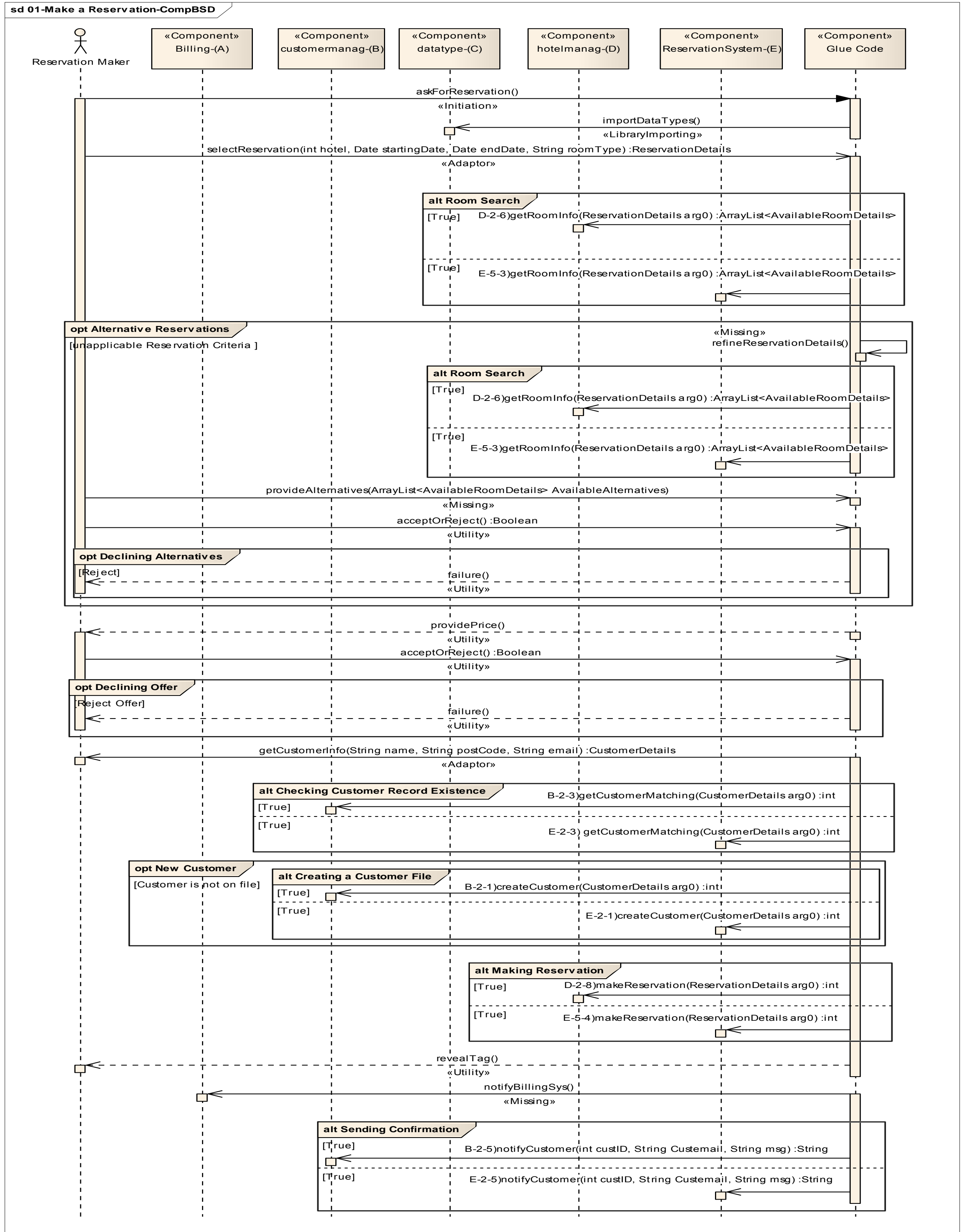


Figure 18: CompBSD of *IMakingReservation*

2) ICancelReservation

Static Realization:

From the use-case description, we may have the following list of "real" actions from the main success scenario and extensions underlined:

1. Reservation Maker asks to cancel a reservation.
2. System asks Reservation Maker to enter the reservation tag.
3. System could find the reservation.
4. System cancels the reservation.
5. The system will notify billing system about refund procedure.
6. System displays the cancellation confirmation and sends it by e-mail.

Extensions:

2. Reservation tag is not available.

   a. System asks for reservation name and post code.
   b. Get all active reservations for the customer.
   c. The customer will select the designated reservation.

3a. System could not find the reservation.

   a. Fail

4. System could not find the reservation.

   a. Fail

Then, we can generalize the extracted real actions to get the final conceptual operations:

- askForCancellation( ): This operation role is to initiate the whole scenario of cancelling a reservation.
- provideReservationTag( ): for entering reservation tag.
- provideNameANDPostCode( ): for entering name and the post code of the customer.
- displayANDSelectActiveReservations( ): for displaying and selecting from active reservations found for a customer.
- cancelReserv( ): for actual procedure of cancelling a reservation.

- failure( ): for issuing failure messages.
- notifyBillingSys( ): for notifying the Billing system about the refunded and cut money.
- confirmCancellation( ): an operation for confirming the cancellation and sending it by e-mail.

Let's here apply IIIP to conceptual operations of **ICancelReservation** one by one:

- **askForCancellation( ):** no cocncrete operation available to realize it so it is missing and needs to be implemented in the glue code.
- **provideReservationTag ( ):** missing.
- **provideNameANDPostCode( ):** missing.
- **displayANDSelectActiveReservations( ):** it could be partially fulfilled with the following concrete operations: (B-2-3), (E-2-3), (D-2-4), and (E-4-3). This partial fulfillement should be indicated in the realization table. As shown in table 4 for this cocnceptual interface we have indicated that by enclosing the realization statement within curley braces.
- **cancelReserv( ):** The concrete operations could be involved are: (D-1-8), (E-3-8), (B-1-1), and (E-1-1).
- **failure( ):** missing.
- **notifyBillingSys( ):** missing.
- **confirmCancellation( ):**cocncrete operations may involve: (B-2-5) and (E-2-4).

Then, we get the UCCM diagram in figure 19 and realization table shown in table 4.

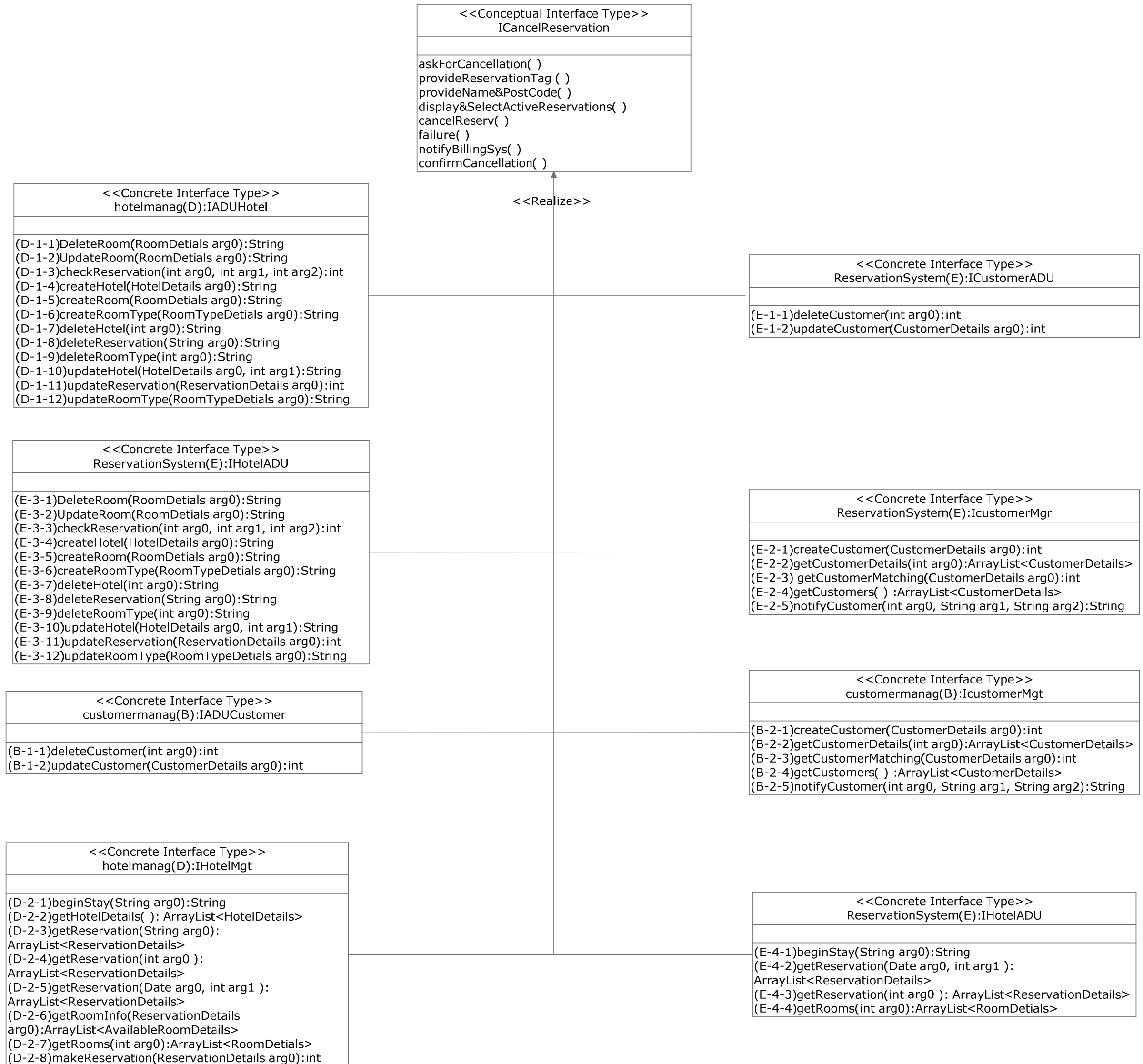


Figure 19: UCCM diagram for *ICancelReservation*

ICancelReservation	
Conceptual Operations	**Realization**
askForCancellation( )	Missing
provideReservationTag ( )	Missing
provideNameANDPostCode( )	Missing
displayANDSelectActiveReservations( )	Partially{[(B-2-3)OR(E-2-3)] AND [(D-2-4)OR (E-4-3)]}
cancelReserv( )	[(D-1-8) OR (E-3-8)] AND[(B-1-1) OR (E-1-1)]
failure( )	Missing
notifyBillingSys( )	Missing
confirmCancellation( )	(B-2-5) OR (E-2-4)

Table 4 : Realization table for *ICancelReservation*

Dynamic Realization:

Here we are going to build a CompBSD corresponding to **ICancelReservation** as shown in figure 20.

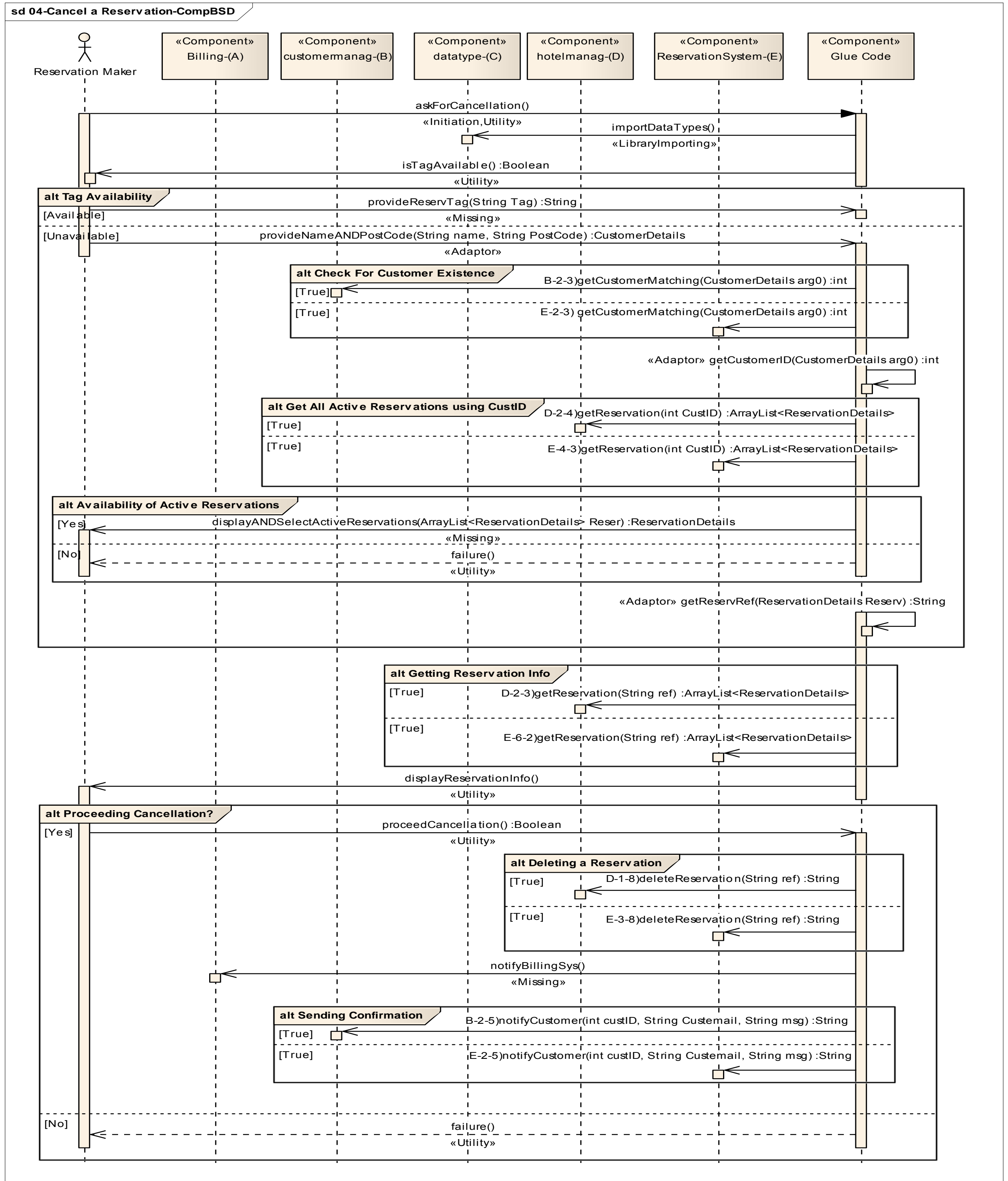


Figure 20: CompBSD of *ICancelReservation*

3) IAmendReservation

Static Realization:

From the use-case description, we may have the following list of "real" actions from the main success scenario and extensions underlined:

1. Reservation Maker asks to amend a reservation.
2. System asks to enter the reservation tag.
3. System could find the reservation.
4. System will go through process of a new reservation scenario.
5. The system will display the price of the new reservation.
6. The system will ask the Reservation Maker to pay the difference in price between the old and new reservations.
7. The system will generate a new reservation tag to the Reservation Maker.
8. The system sends the confirmation by e-mail.

Extensions:

2. Reservation tag is not available.

   a. System asks for reservation name and post code.
   b. Get all active reservations for the customer.
   c. The customer will select the designated reservation.

2. System could not find the reservation.

   a. Fail

Then, we can generalize the extracted real actions to get the final conceptual operations:

- askForAmending( ): This operation role is to initiate the the whole scenario of amending a reservation.
- provideReservationTag ( ): for entering reservation tag.
- provideNameANDPostCode( ): for entering name and the post code of the customer.
- displayANDSelectActiveReservations( ): for displaying and selecting from active reservations found for a customer.

- calculatePriceDifference( ): for calculating the diffrence between the old and new reservation to be paid by\returned to the customer.
- failure( ): for issuing failure messages.
- notifyBillingSys( ): for notifying the billing system about the new payement.
- confirmCancellation( ): an operation for confirming the amending and sending it by e-mail.
- <Make Reservation Scenario>: to carry out a new reseravtion

Let's here apply IIIP to conceptual operations of **IAmendReservation** one by one:

- **askForAmending( ):** missing.
- **provideReservationTag ( ):** missing.
- **provideNameANDPostCode( ):** missing.
- **displayANDSelectActiveReservations( ):** it could be partially fulfilled with the following cocncrete operations: (B-2-3), (E-2-3), (D-2-4), and (E-4-3).
- **calculatePriceDifference( ):** missing.
- **failure( ):** missing.
- **notifyBillingSys( ):** missing.
- **confirmCancellation( ):** cocncrete operations may involve: (B-2-5) and (E-2-4).
- **<Make Reservation Scenario>:** realized by ***IMakingReservation*** conceptual interface.

Then, we get the UCCM diagram in figure 21 and realization table shown in table 5.

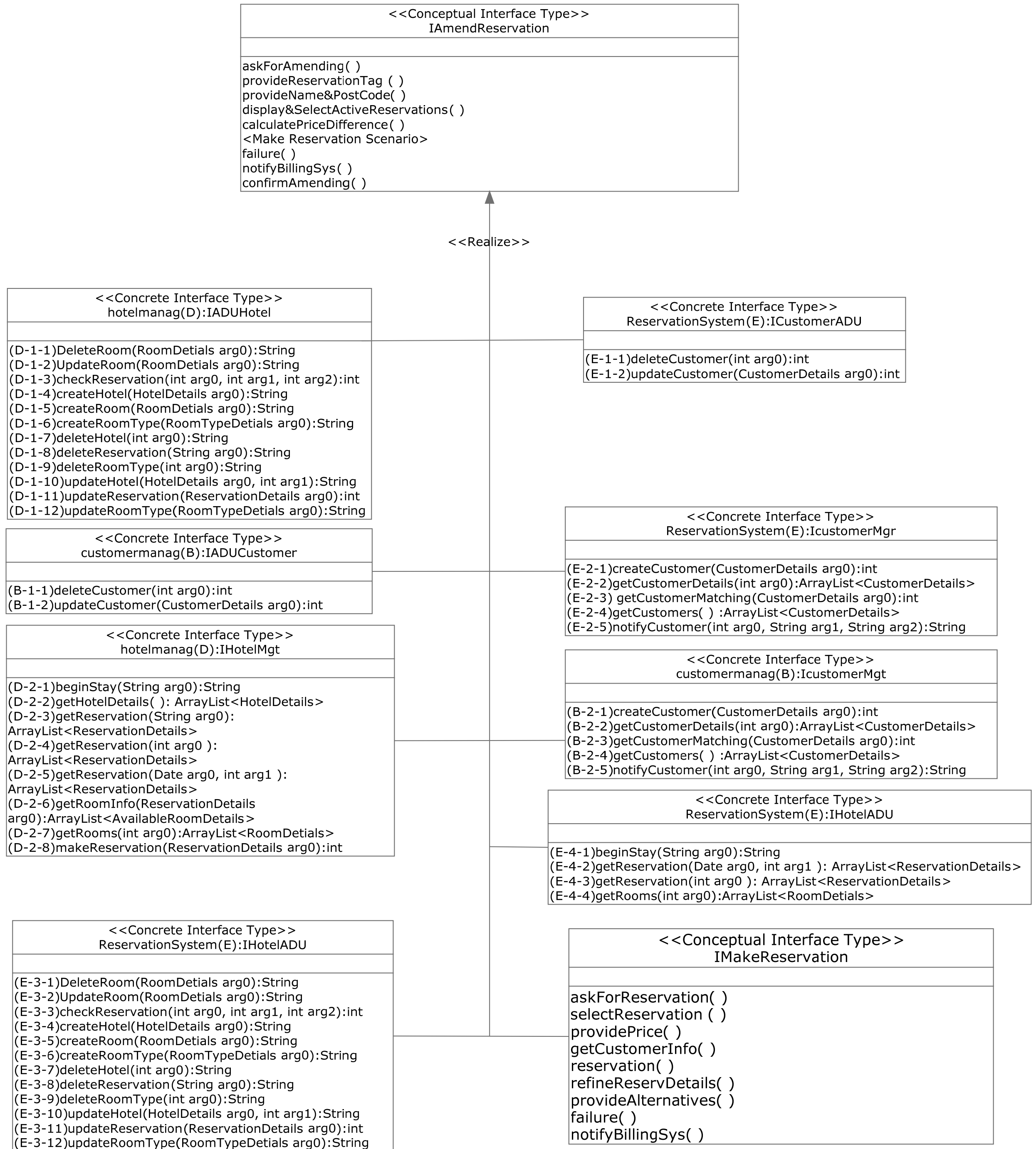


Figure 21: UCCM diagram for *IAmendReservation*

IAmendReservation	
Conceptual Operations	**Realization**
askForAmending( )	Missing
provideReservationTag ( )	Missing
provideNameANDPostCode( )	Missing
displayANDSelectActiveReservations( )	Partially{[(B-2-3)OR(E-2-3)] AND [(D-2-4)OR (E-4-3)]}
calculatePriceDifference( )	Missing
< Make Reservation Scenario>	IMakingReservation
failure( )	Missing
notifyBillingSys( )	Missing
confirmAmending( )	(B-2-5) OR (E-2-4)

Table 5: Realization table for *IAmendReservation*

Dynamic Realization:

Here we are going to build a CompBSD corresponding to **IAmendReservation** as shown in figure 22.

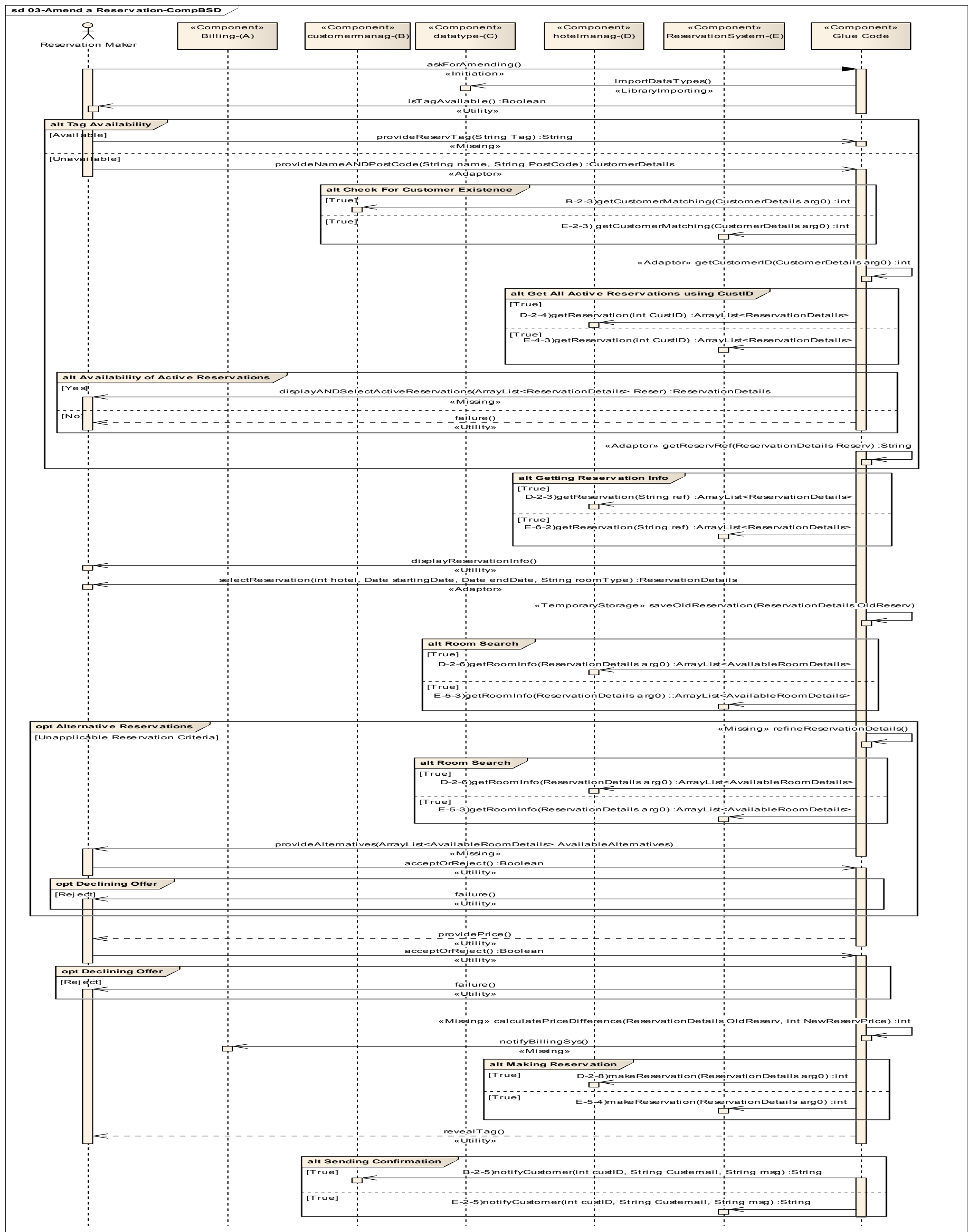


Figure 22: CompBSD of *IAmendReservation*

4) IProcessNoShows

From the use-case description, we may have the following list of real actions from the main success scenario underlined:

Main Success Scenario:

1. Administrator asks the system to display the list of reservations whose holders did not show or take them up.
2. The system will display the list of no shows.
3. The system administrator will select all or a subset of the no-shows list.
4. The system will apply the payment-cut to all of the selected no-shows.
5. The system will notify billing system about the cut and remaining amount.
6. The system will send an e-mail to the customer informing him about the applied procedure.

Then, we can generalize the extracted real actions to get the final conceptual operations:

- askToProcessNoShows( ): for initiating the whole scenario.
- displayNoShows( ): for display of all available no shows.
- selectFromNoShows( ): for selecting the group of no shows to which the no shows procedure will be applied.
- applyNoShowProcedure( ): for applying no show procedure (i.e. payment-cut).
- notifyBillingSys( ):notifying the billing system about the applied no show procedure regarding cut money.
- confirmNoShow( ):for sending confirmation of no show procedure to the corresponding customers.

Let's here apply IIIP to conceptual operations of **IProcessNoShows** one by one:

- **askToProcessNoShows( ):** missing.

- **displayNoShows( ):** missing.
- **selectFromNoShows( ):** missing.
- **applyNoShowProcedure( ):** missing.
- **notifyBillingSys( ):**missing.
- **confirmNoShow( ):** cocncrete operations that may be involved: (B-2-5) and (E-2-4).

Then, we get the UCCM diagram in figure 23 and realization table shown in table 6.

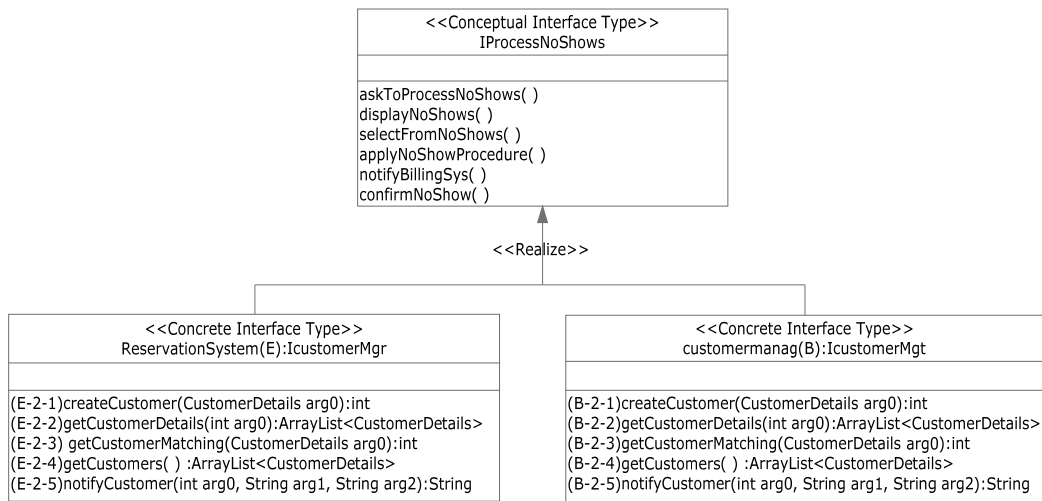


Figure 23: UCCM diagram for *IProcessNoShows*

IProcessNoShows	
Conceptual Operations	**Realization**
askToProcessNoShows( )	Missing
displayNoShows( )	Missing
selectFromNoShows( )	Missing
applyNoShowProcedure( )	Missing
notifyBillingSys( )	Missing
confirmNoShow( )	(B-2-5) OR (E-2-4)

Table 6: Realization table for *IProcessNoShows*

Dynamic Realization:

Here we are going to build a CompBSD corresponding to **IProcessNoShows** as shown in figure 24.

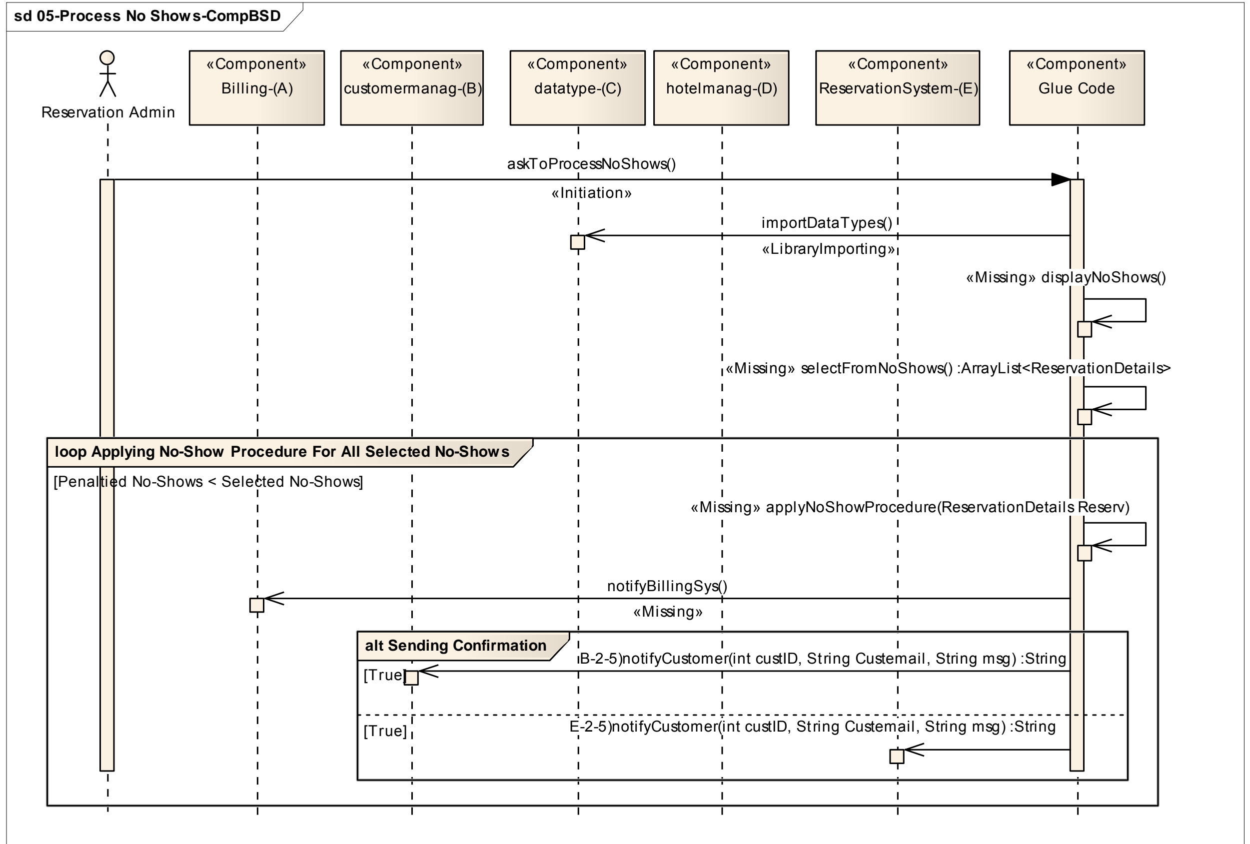


Figure 24: CompBSD of *IProcessNoShows*

5) ITakingUpReservation

From the use-case description, we may have the following list of real actions from the main success scenario and extensions underlined:

Main Success Scenario:

1. Guest arrives at hotel and claims a reservation.
2. Guest provides reservation tag.
3. Guest confirms details of stay duration, room type.
4. System allocates room.
5. System notifies billing system that a stay is starting.

Extensions:

3. System cannot find a reservation with the given tag.
   a. Guest provides name and post code.

b. System identifies guest and displays active reservations for the customer.
c. Guest selects the reservation.
d. Resume 4.

3. The reservation tag refers to a reservation at a different hotel.

a2. Fail.

3c. No active reservations at this hotel for this customer.

a. Fail

Then, we can generalize the extracted real actions to get the final conceptual operations:

- askToTakeUp( ): for initiating the whole scenario.
- provideReservationTag ( ): for entering reservation tag.
- provideNameANDPostCode( ): for entering name and the post code of the customer.
- displayANDSelectActiveReservations( ): for displaying and selecting from active reservations found for a customer.
- notifyBillingSys( ):notifying the billing system that a stay has begun.
- failure( ):for issuing failure message.

Let's here apply IIIP to conceptual operations of **ITakingUpReservation** one by one:

- **askToTakeUp( ):** missing.
- **provideReservationTag ( ):** missing.
- **provideNameANDPostCode( ):** missing.
- **displayANDSelectActiveReservations( ):** missing. It could be partially fulfilled with the following cocncrete operations: (B-2-3), (E-2-3), (D-2-4), and (E-4-3).

- **notifyBillingSys( ):**missing.
- **failure( ):** missing.

Then, we get the UCCM diagram in figure 25 and realization table shown in table 7.

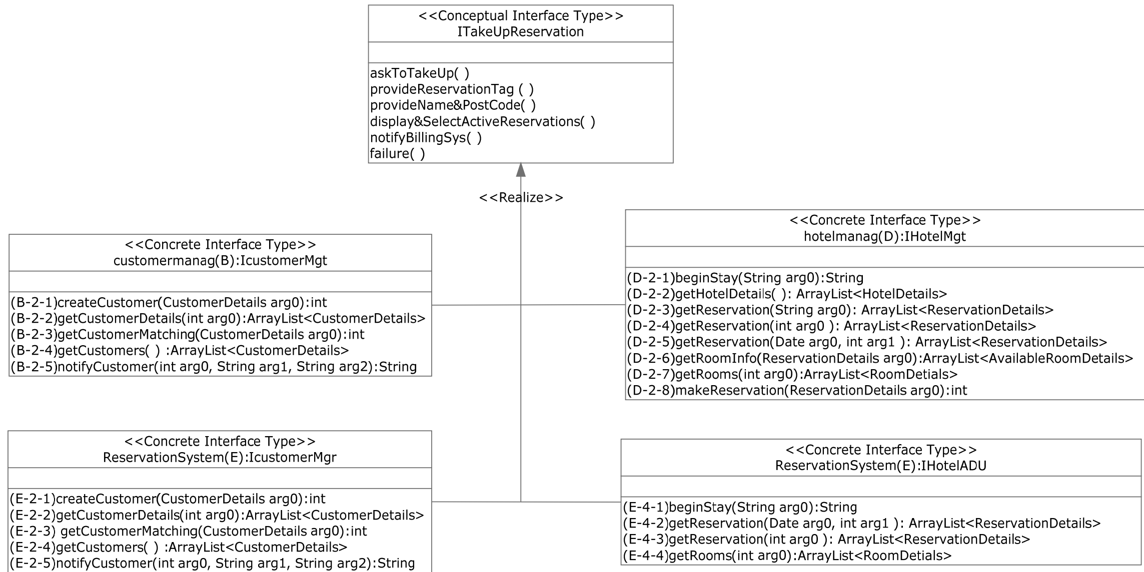


Figure 25: UCCM diagram for *ITakingUpReservation*

ITakingUpReservation	
Conceptual Operations	**Realization**
askToTakeUp( )	Missing
provideReservationTag ( )	Missing
provideNameANDPostCode( )	Missing
displayANDSelectActiveReservations ( )	Partially{[(B-2-3)OR(E-2-3)] AND [(D-2-4)OR (E-4-3)]}
notifyBillingSys( )	Missing
failure( )	Missing

Table 7: Realization table for *ITakingUpReservation*

Dynamic Realization:

Here we are going to build a CompBSD corresponding to ***ITakingUpReservation*** as shown in figure 26.

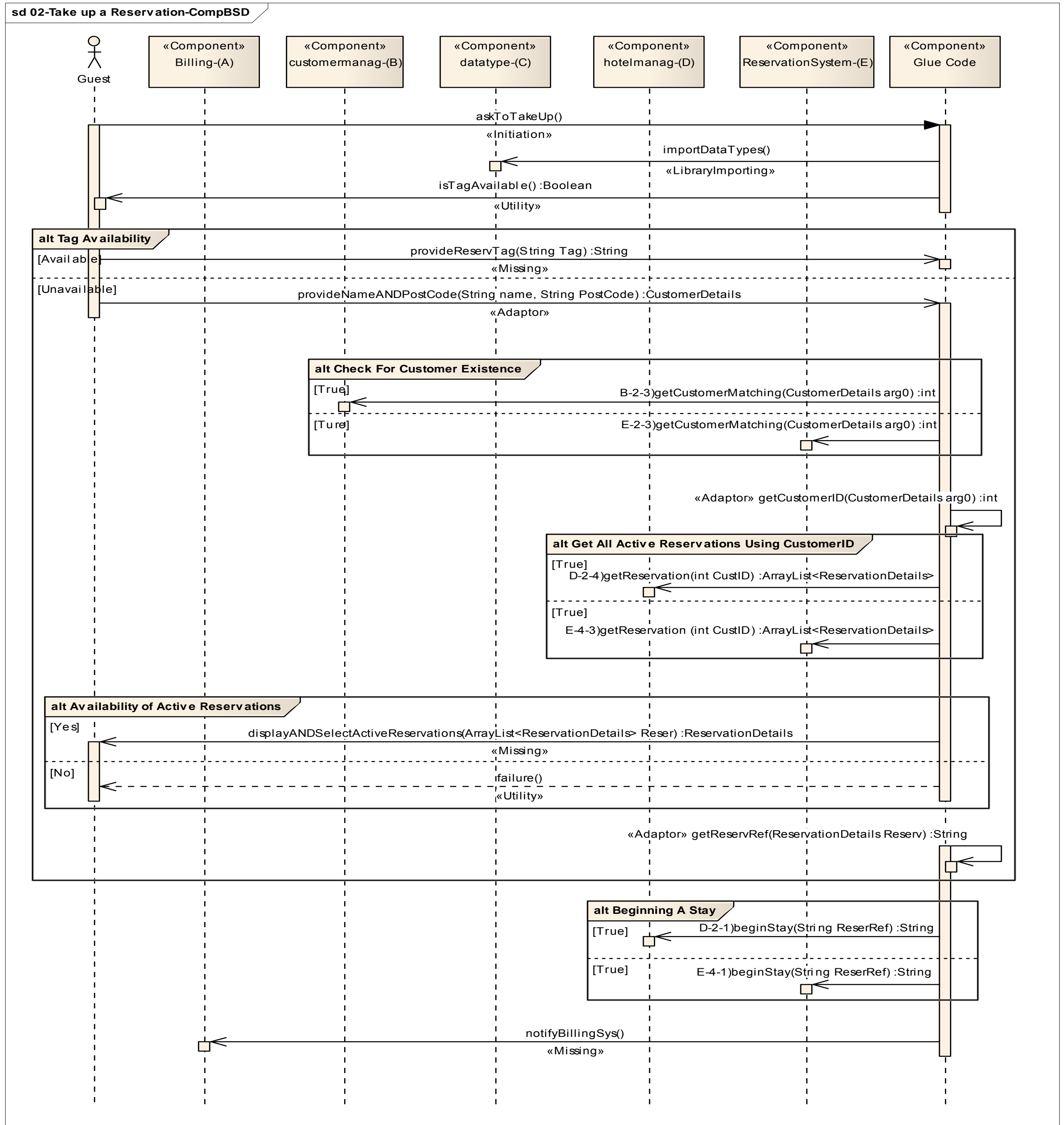


Figure 26: CompBSD of *ITakingUpReservation*

5.3 Summary

In this chapter, the realization stage of the proposed framework has been applied to a case study of a Hotel Reservation System (HRS). We had five selected software components written in Java in advance. Also, we had the UCs of the system-to-be in the form of UC descriptions. We started by applying the UCCM technique to all of the system UCs to derive the UCCM diagrams and realization tables. Afterward, we have applied dynamic realization process to UCs along with their corresponding realization tables to derive the first form of integration specification which were in the form of CompBSDs for HRS system.

CHAPTER 6

GLUE CODE FORMAL SPECIFICATION

6.1 Glue Code and Formal Specification

Generally speaking, a formal specification is the expression in some formal language and at some level of abstraction, of a collection of properties some system should satisfy. Formality helps in obtaining higher-quality specifications; it also provides the basis for their automated support. They remove areas of doubt in a specification.

In chapter 4, we have demonstrated how we can derive one form of specification for the glue code. This specification was on a form of a set of CompBSDs representing required system scenarios. This specification is suspected to be informal, imprecise, and ambiguous. In order to get a more precise and correct glue code specification, more formality should be applied to the glue code specification.

In this chapter, we present the composition stage of the framework. This stage produces a formal glue code specification. This specification will be in form of OCL-Constrained Class Diagram (OCCD). It gives the system developer a better definition of the properties that should be implemented by the system-to-be.

6.2 Object Constraint Language (OCL)

A UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is, among other things, a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints, so-called formal languages have been developed. The disadvantage of traditional formal languages is that they are usable to persons with a strong mathematical background, but difficult for the average business or system modeler to use (OMG, 2005).

OCL is a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. Note that when the OCL expressions are evaluated, they do not have side effects; i.e. their evaluation cannot alter the state of the corresponding executing system.

OCL expressions can be used to specify operations\actions that, when executed, do alter the state of the system. UML modelers can use OCL to specify application-specific constraints in their models. UML modelers can also use OCL to specify queries on the UML model, which are completely programming language independent.

The general form of an OCL expression is as follows:

package <packagePath>

context <contexualInstanceName> : <modelElement>

<expressionType><expressionName> : <expressionBody>

<expressionType><expressionName> : <expressionBody>

...

endpackage

The following are some basic rules for any OCL expression:

- The package context is optional.
- The expression context is mandatory.
- One or more expressions.
- Every OCL constraint has a context, the element that is being constrained (operation, class).
- A constraint can be written in a textual form (data dictionary) or attached to model elements.
- Keyword **context** in bold type as well as constraint stereotypes.
- The keyword self in the textual form of the constraint simply refers to the instance of the context class (not always needed but it aids readability).

As an example, a class may have invariants to preserve a specific property as shown for classes in figure 27. There are four business rules about *CheckingAccounts* and *DepositAccounts* (Arlow & Neustadt, 2005).

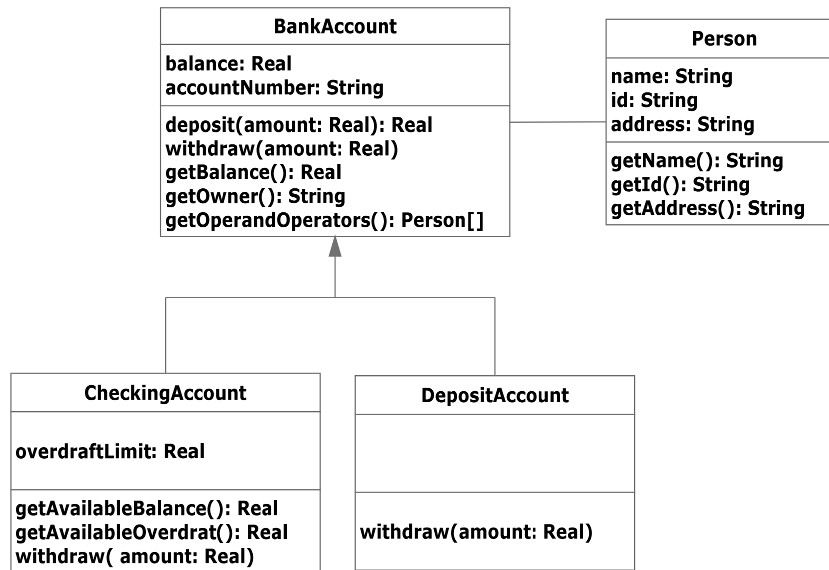


Figure 27: Bank accounts class diagram

1. No Account shall be overdrawn by more than $1000.00. We can express this first rule as an invariant on the *BankAccoun*t class because it must be true for all instances of *BankAccount*.

context BankAccount
inv balanceValue:
self.balance >= (-1000.0)
/* A bank account shall have a balance > -1000.0 */

This invariant is inherited by the two subclasses, *CheckingAccount* and *DepositAccount.* These subclasses can strengthen this invariant but can never weaken it. This is to preserve the substitutability principle.

2. *CheckingAccounts* have an overdraft facility. The account shall not be overdrawn to an amount greater than its overdraft limit. We can express these rules as invariants on the *CheckingAccount* class:

```
context CheckingAccount
inv balanceValue:
        self.balance >= (-overdraftLimit)
inv maximumOverdraftLimit:
      self.overdraftLimit <= 1000.0
```

See how the *CheckingAccount* subclass has strengthened the *BankAccount::balance* class invariant by overriding it.

3. *DepositAccounts* shall never be overdrawn. We can express this rule as follows:

```
context DepositAccount
inv balanceValue:
        self.balance >= 0.0
                        /* Have a balance of zero or more */
```

This also is a strengthening of *BankAccount::balance* class invariant.

4. Each *accountNumber* shall be unique. This constraint is expressed as an invariant on the *BankAccount* class.

```
context BankAccount
        inv uniqueAccountNumber:
                BankAccount::allInstances()->
              isUnique( account | account.accountNumber)
```

Also, pre and postconditions apply to operations. Their contextual instance is an instance of the classifier to which the operations belong. Refer to the bank accounts example in figure 27 and consider the *deposit()* operation that both *CheckingAccount* and *DepositAccount* inherit from *BankAccount*. There are two business rules for this:

1. The amount to be deposited shall be greater than zero.

2. After the operation executes, the amount shall have been added to the balance.

- These rules can be expressed concisely and accurately in preconditions and postconditions on the *BankAccount::deposit()* operation as follows:

```
context BankAccount::deposit( amount: Real): Real
        pre amountToDepositGreaterThanZero:
                amount > 0
        post depositSucceeded:
                self.balance = self.balance@pre + amount
```

Notice the use of the @pre keyword. This keyword can be used only within postconditions. The expression balance@pre refers to the value of balance before the operation executes.

- For completeness, here are the constraints on the *BankAccount::withdraw()* operation

```
context BankAccount::withdraw(ampunt: Real)
        pre amountToWithdrawGreaterThanZero:
                amount > 0
        post withdrawalSucceeded:
                self.balance@pre – amount
```

- When an operation is redefined by a subclass, it gets the preconditions and postconditions of the operation it redefines. It can only change these conditions as follows:

- The redefined operation may only weaken the precondition.

- The redefined operation may only strengthen the postcondition.

6.3 Composition Stage

In chapter 4, we have seen how one can derive the integration (i.e. glue code) specification for a CBS beginning from the UCs of the system-to-be by applying the realization stage of our proposed framework. The derived specification was in form of CompBSDs.

In the composition stage, we are going to derive a formal specification for the glue code. This formal specification will be in form of OCL-Constrained Class Diagrams (OCCDs). An overview for the composition stage is shown in figure 28.

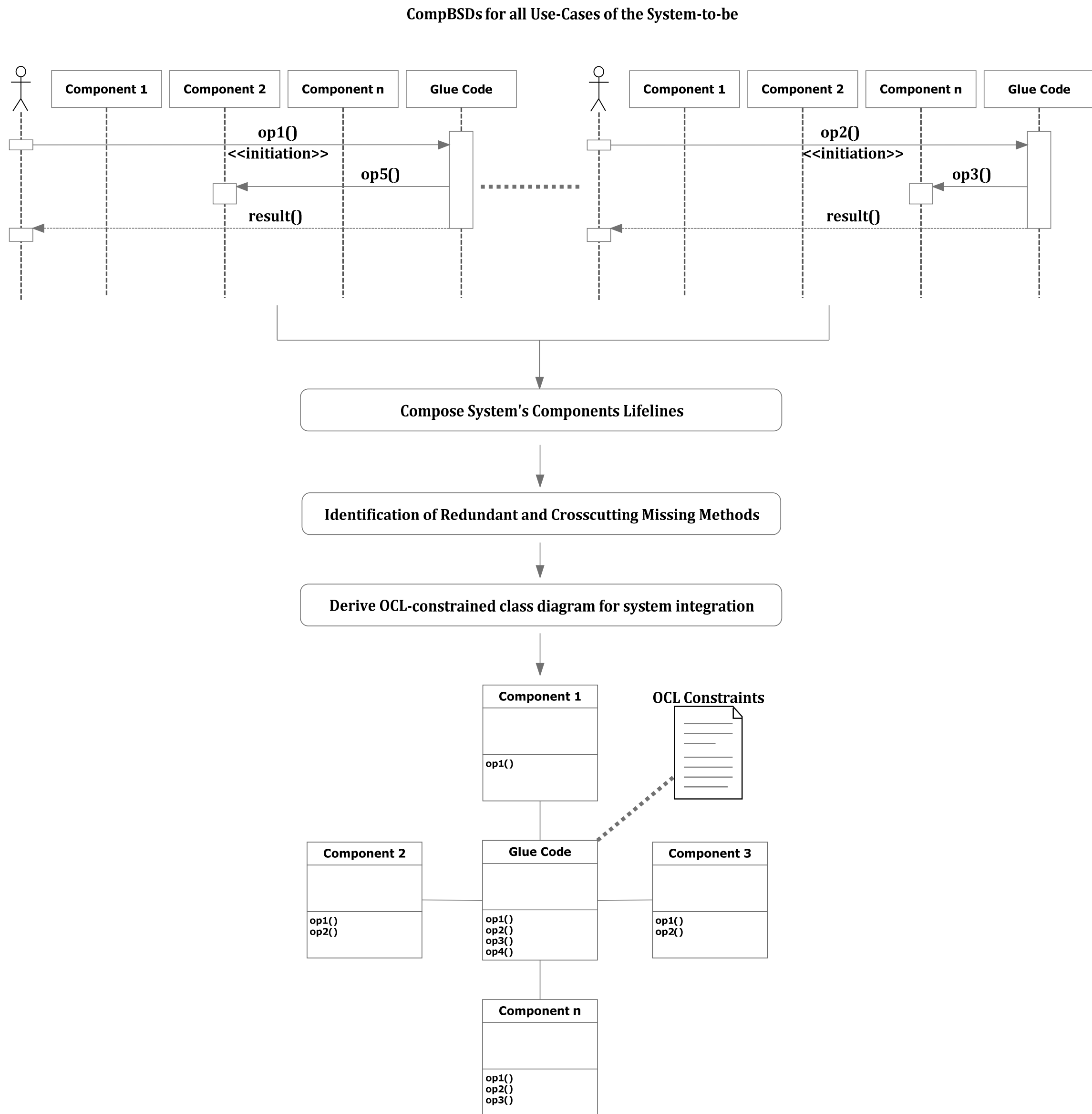


Figure 28: An overview of the composition stage of the proposed framework

6.3.1 OCL-Constrained Class Diagram (OCCD)

OCCD is a class diagram in which each class represents a constituent component that realizes some programming interface(s). Such interfaces have participated in system scenarios represented by CompBSDs. It is noticed here that we began adding the

modifier “programming” to the interfaces due to approaching “coding” arenas. We used to call the interfaces of the system-to-be as “conceptual interfaces” due to high level interfaces derived from UCs in the framework. Here we are going to map such interfaces into programming interfaces implemented by the constituent components. A template of OCCD is shown in figure 29.

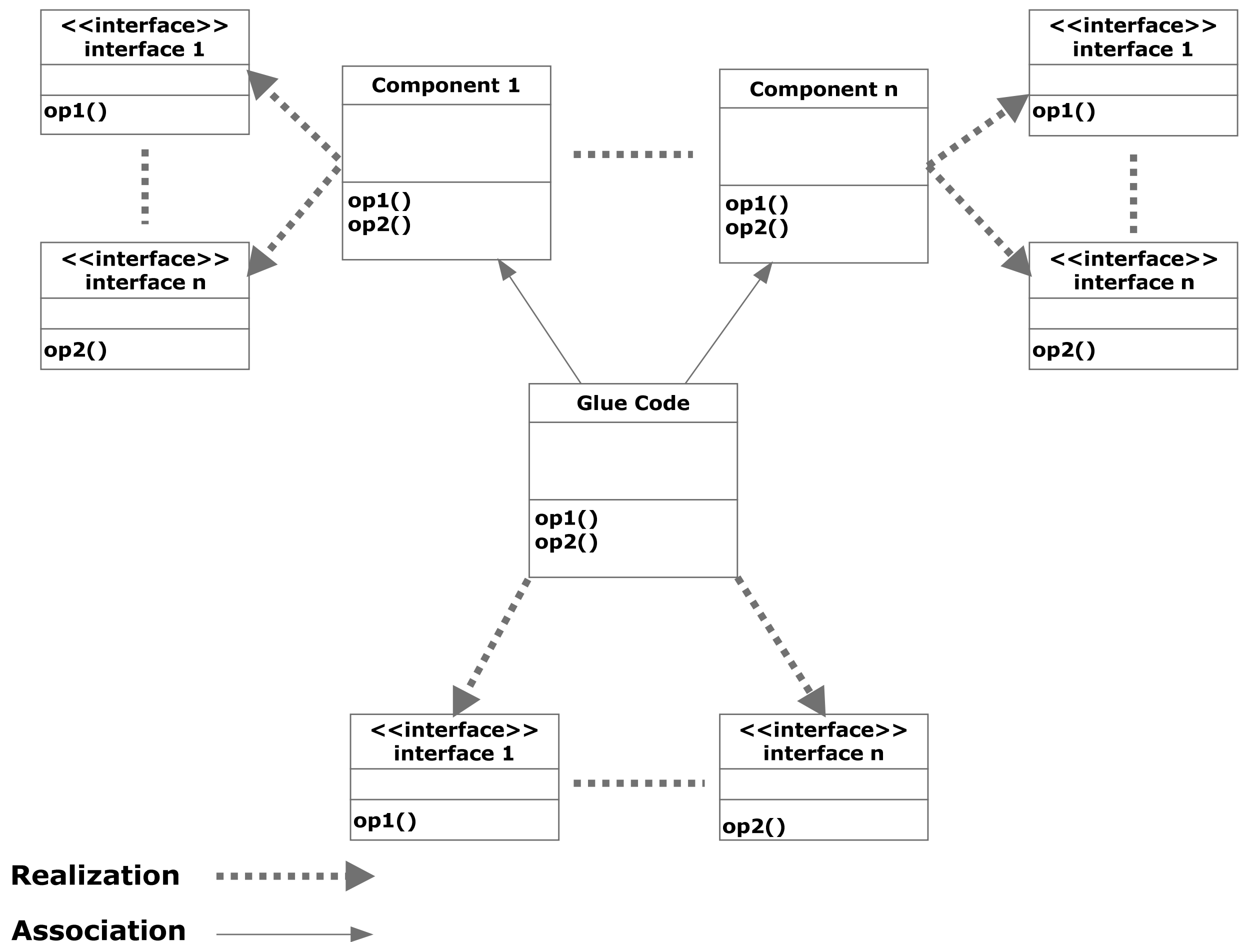


Figure 29: An OCCD template

After passing the realization stage, the composition stage will take the set of CompBSDs resulting from the realization stage as input. Redundant and crosscutting operations of the glue code lifelines of all CompBSDs will be identified. Then, all glue code's scenarios represented by all CompBSDs will be collapsed in a single OCL-Constrained Class Diagram (OCCD) providing the formal specification of the glue code.

The process of building the OCCD will begin by creating a corresponding class for each component in the system including glue code. Then, we are going to build the interface(s) for the glue code component containing all the scenario initiating methods (i.e. missing methods stereotyped with "Initiation"). The glue code class's method compartment will be filled with only the missing methods shown by CompBSDs (i.e. concrete operations provided by the pre-existing selected components will not be added to it). The glue code interface(s) will be attached to the glue code by a hollow-headed dashed arrow and stereotyped with "Realize". The head of the arrow will be attached to the interface whereas its tail is attached to the glue code component.

After all glue code's interface(s) being created and filled with the designated operations, we have to switch consideration to the other components (i.e. the selected components) one by one. We are going here to do the same procedure by creating only the interfaces which contain the operations that have been invoked during the system scenarios. Since an interface may contain many methods, methods' compartments of the interfaces will be filled with only the methods that have been invoked by system scenarios at least once.

Each interface will be attached to its corresponding component using the same arrow used to connect glue code component and its interfaces.

Then, OCL constraints will be annotated to OCCD classes that will be in the form of class invariants and operations' pre and postconditions. Such annotation will provide us with a more precise class diagram for the glue code. Actually, these constraints will be annotated to the glue code component only because we have nothing to do with the other selected components composing the system.

6.3.2 Validating OCCD

Validating OCL constraints annotated to OCCD can be performed using one of the existing automated tools. In this thesis work, we have used UML-based Specification Environment (USE) tool for validating the resulting formal specification for the glue code (Richters, 2001). Let's here give an example provided by (Richters, 2001) on the validation of the model shown in figure 30.

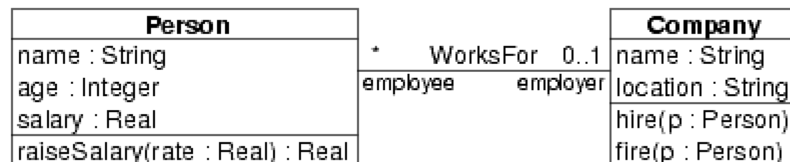


Figure 30: Class diagram for persons and a company

In USE tool the validation process will be as follows:

1) Building the model by mapping the model into USE-based specification as follows

```
model Employee
-- classes

class Person
attributes
name : String
  age : Integer
  salary : Real
operations
  raiseSalary(rate : Real) : Real
end

class Company
attributes
  name : String
  location : String
operations
  hire(p : Person)
  fire(p : Person)
end

-- associations

association WorksFor between
  Person[*] role employee
  Company[0..1] role employer
end
```

2) Adding pre and postconditions as follows:

```
constraints
context Company::hire(p : Person)
  pre  hirePre1: p.isDefined()
  pre  hirePre2: employee->excludes(p)
  post hirePost: employee->includes(p)

context Company::fire(p : Person)
  pre  firePre:  employee->includes(p)
  post firePost: employee->excludes(p)
```

The first precondition of the `hire` operation is named `hirePre1` and makes sure that the operation can only be called with a well-defined person object (Note that the operation `isDefined` is a USE extension. It is not possible to express this constraint with standard OCL). The second precondition (`hirePre2`) makes sure that the person passed as parameter `p` is not already an employee of the company. The postcondition (`hirePost`) guarantees that after the operation has exited, the person actually has been added to the set of employees. The constraints for the operation `fire` work just the other way round.

3) Importing the model into USE tool.
4) Creating the initial system state: this is achieved by creating all needed objects corresponding to classes pre-specified in the model. Assigning values to the data members of the objects to be passed in the simulation of the system scenarios. For instance, we will create two objects for *Person* and *Company* classes using "!create" command. Then, we will set the values of the data members of both of them using "!set" command as follows:

```
use> !create ibm : Company
use> !create joe : Person
use> !set joe.name := 'Joe'
use> !set joe.age := 23
```

The current system state can be visualized with an object diagram view shown in figure 31.

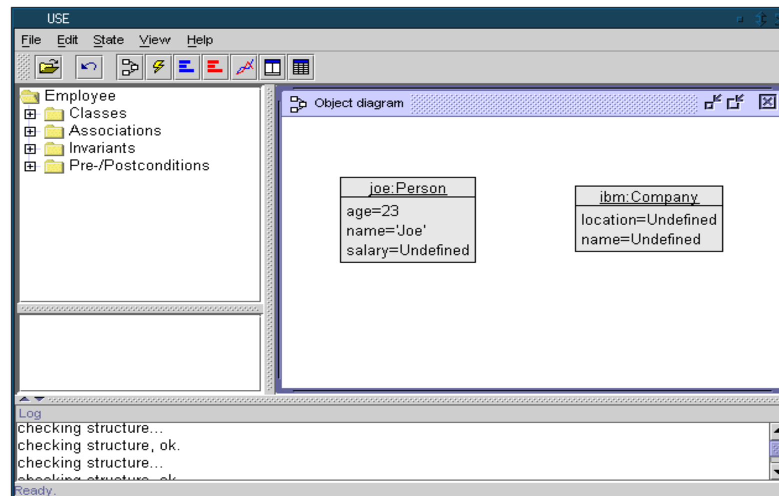


Figure 31: Generated object diagram for the model

5) Calling Operations and Checking Preconditions: operation calls are initiated with the command "!openter". The syntax is as follows:

!openter <source-expression> <operation-name> ([<argument-expression-list>])

Where <source-expression> must be an OCL expression denoting the receiver object of the operation named <operation-name>. Depending on the operation's signature an <argument-expression-list> has to be passed to the operation.

The following command shows the top-level bindings of variables. These variables can be used in expressions to refer to existing objects.

```
use> info vars
ibm : Company = @ibm
joe : Person = @joe
```

We invoke the operation `hire` on the receiver object `ibm` and pass the object `joe` as parameter.

```
use> !openter ibm hire(joe)
precondition `hirePre1' is true
precondition `hirePre2' is true
```

The `!openter` command has the following effects:

1. The source expression is evaluated to determine the receiver object.
2. The argument expressions are evaluated.
3. The variable `self` is bound to the receiver object and the argument values are bound to the formal parameters of the operation. These bindings determine the local scope of the operation.
4. All preconditions specified for the operation are evaluated.
5. If all preconditions are satisfied, the current system state is saved and the operation call is saved on a call stack.

6) Exiting Operations and Checking Postconditions: we can simulate the execution of an operation with the usual USE primitives for changing a system state. The postcondition of the `hire` operation requires that a *WorksFor* link between the person and the company has to be created as shown in figure 32. We also set the salary of the new employee. This can be created as follows:

```
use> !insert (p, ibm) into WorksFor
use> !set p.salary := 2000
```

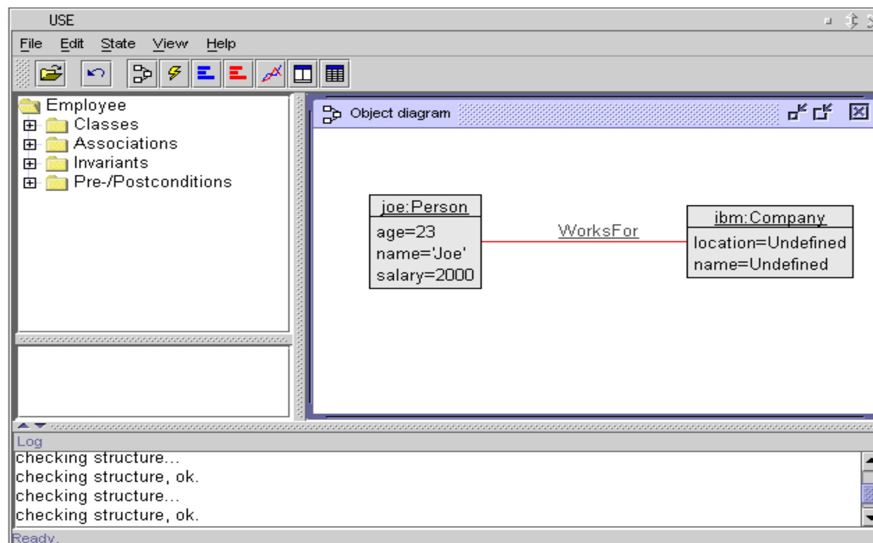


Figure 32: The object model after creating the link *WorksFor*

After generating all side-effects of an operation, we are ready to exit the operation and check its postconditions. The command “`!opexit`” simulates a return from the currently active operation. The syntax is:

!opexit [<*result-expression*>]

The optional <*result-expression*> is only required for operations with a result value. An example will be given later. The operation `hire` specifies no result, so we can just issue

```
use> !opexit
postcondition 'hirePost' is true
```

The `!opexit` command has the following effect:

1. The currently active operation is popped from the call stack.
2. If an optional result value is given, it is bound to the special OCL variable "result".

3. All postconditions specified for the operation are evaluated in context of the current system state and the pre-state saved at operation entry time.
4. All local variable bindings are removed.

In our example, the postcondition `hirePost` is satisfied.

7) Visualization as Sequence Diagram: The USE tool can visualize a sequence of operation calls as a UML sequence diagram. The screenshot shown in figure 33 shows the sequence of calls for the example. During a validation session the diagram is automatically updated on each operation call.

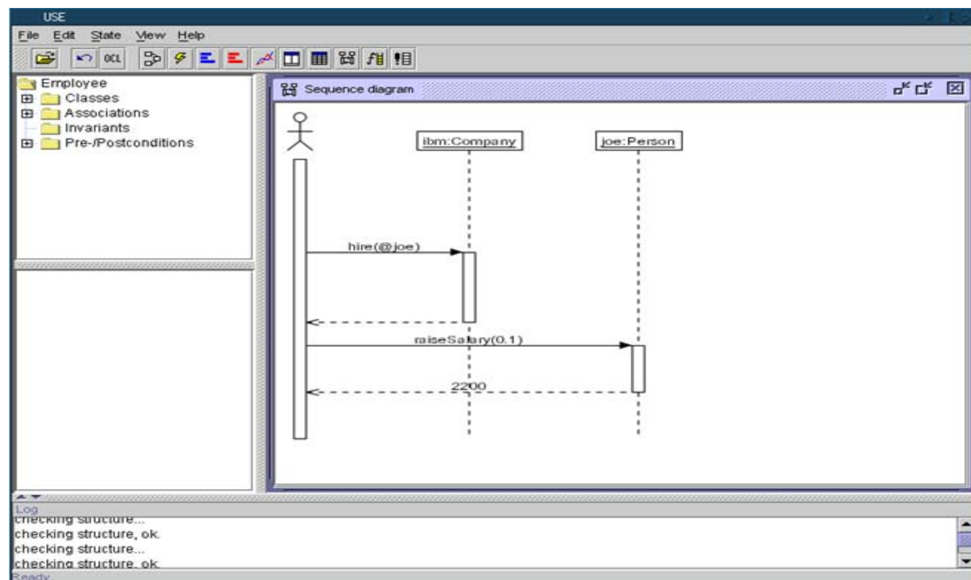


Figure 33: UML sequence diagram representing sequence of operation calls

6.4 Summary

In this chapter, the composition stage of the proposed framework has been well-explained. We proposed here a formal specification for the glue code in the form of an OCL-Constrained Class Diagram. Notations and definitions for OCCD have been established. We have demonstrated the process of deriving OCCD by collapsing the resulting CompBSDs from the realization stage of the framework. Then, the process of annotating OCL constraints to OCCD has been illustrated, as well as, the process of validating it to check the correctness of the system integration model represented by OCCD by means of the USE tool.

CHAPTER 7

CASE STUDY FOR COMPOSITION STAGE

7.1 Applying Composition Stage

In chapter 5, we have derived a set of CompBSDs as a result of the realization stage that has been applied to the UCs of Hotel Reservation System. We are here going to map all the resulting CompBSDs to a single OCCD to represent the glue code formal specification for the HRS system. Firstly, let's list all the missing methods identified by CompBSDs as shown by table 8.

Missing Functionality	Stereotype	Role
askForReservation() : void	Initiation	For triggering making reservation scenario
askToTakeUp() : void	Initiation	For triggering taking up reservation scenario
askForAmending() : void	Initiation	For triggering amending reservation scenario

askForCancellation() : void	Initiation	For triggering cancelling reservation scenario
askToProcessNoShows() : void	Initiation	For triggering processing no show ups scenario
importDataTypes():void	LibraryImporting	For importing customized data types
selectReservation(hotel :int, startingDate: Date, endDate: Date, roomType: String): ReservationDetails	Adapter	To get reservation data from user and adapt it to the "ReservationDetails" data type
refineReservationDetails():void	Missing	to change unapplicable user entered reservation information to get the closest reservation available
provideAlternatives(AvailableAlternatives: ArrayList<AvailableRoomDetails>): void	Missing	Shows all alternative reservations
acceptOrReject():Boolean	Utility	Prompting the user to answer with Yes or No

failure():void	Utility	Showing a failure message regarding a specific transaction
providePrice():void	Utility	Displaying the price of a reservation
getCustomerInfo(name: String, postcode: String, e-mail: String): CustomerDetails	Adapter	To get entered customer data and convert it to the “CustomerDetails” data type
notifyBillingSys():void	Missing	For notifying the billing system about the financial-related aspect of a transaction
revealTag():void	Utility	Displaying the reservation tag to the user
isTagAvailable():Boolean	Utility	Asking the user for the availability of tag, the answer will be either Yes or No.
provideReservTag(Tag : String):String	Missing	Getting reservation tag from the user

provideNameANDPostCode(String name:String, PostCode: String): CustomerDetails	Adapter	Getting other information of customer when tag is unavailable and converting it to the “CustomerDetails” data type
getCustomerID(arg0: CustomerDetails): int	Adapter	Extracting customer ID from the corresponding field in the customer’s data of “CustomerDeatils” data type
displayANDSelectActiveReservations(Reser: ArrayList<ReservationDetails>): ReservationDetails	Missing	Display all current active reservation to the customer to choose from
getReservRef(Reserv: ReservationDetails): String	Adapter	Extracting reservation reference from the corresponding field in the reservation’s data of “ReservationDetails” data type
saveOldReservation (OldReserv: ReservationDetails): void	TemporaryStorage	Temporarily storing reservation and customer data

calculatePriceDifference(OldReserv: ReservationDetails, NewReservPrice:int):int	Missing	Calculating the difference between two reservations prices
proceedCancellation():Boolean	Utility	Asking the user whether he want to continue cancellation process
displayNoShows():void	Missing	Displaying all available customers who haven’t shown up yet
selectFromNoShows():ArrayList<ReservationDetails>	Missing	Selecting a group of no shows
applyNoShowProcedure(Reserv: ReservationDetails):void	Missing	Applying the intended procedure to a group of selected no shows

Table 8: List of missing functionalities

Also, a list of all concrete operations has appeared to be invoked in CompBSDs. It is enough just to list them by their concrete operation codes so that it could be referred quickly from the components documentation as follows:

(D-2-6), (E-5-3), (B-2-3), (E-2-3), (B-2-1), (E-2-1), (D-2-8), (E-5-4), (B-2-5), (E-2-5), (D-2-4), (E-4-3), (D-2-3), (E-6-2), (D-1-8), (E-3-8), (D-2-1), (E-4-1).

Then, we will begin building the OCCD skeleton by creating a class for each component involved in the scenarios including the glue code component as shown in figure 34.

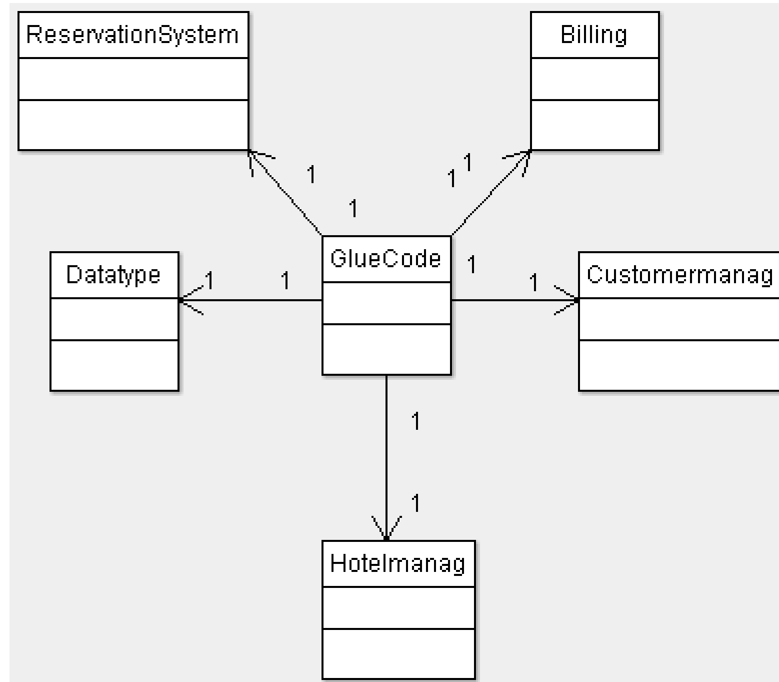


Figure 34: Skeleton of HRS OCCD

Then, we are going to attach the interfaces to its corresponding components. For, the components other than glue code we are going to attach only the interfaces and operations that contributed to the scenarios represented by CompBSDs to their corresponding components being stereotyped with "*realize*" (i.e. interfaces having nothing to do with the system-to-be scenarios will be discarded). This is shown in figure 35.

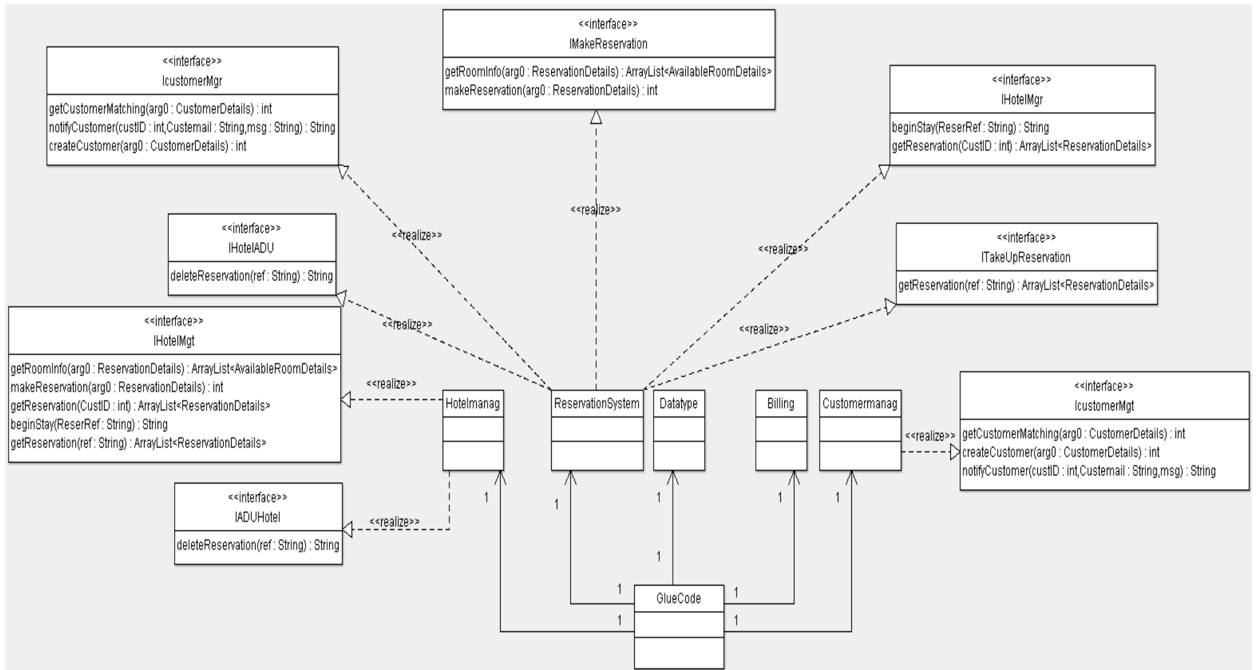


Figure 35: Attaching interfaces of the components to the OCCD skeleton

Then, we will add all missing methods to the method compartment of the glue code component class. Also, we should attach an interface named "*ReservationOperations*" to the glue code class. The missing methods stereotyped with "Initiation" are surrounded by a red colour frame. They will be put as exposed methods in the method compartment of this interface being implemented within the glue code component as shown in figure 36. The final OCCD will appear as shown in figures 37 and 38.

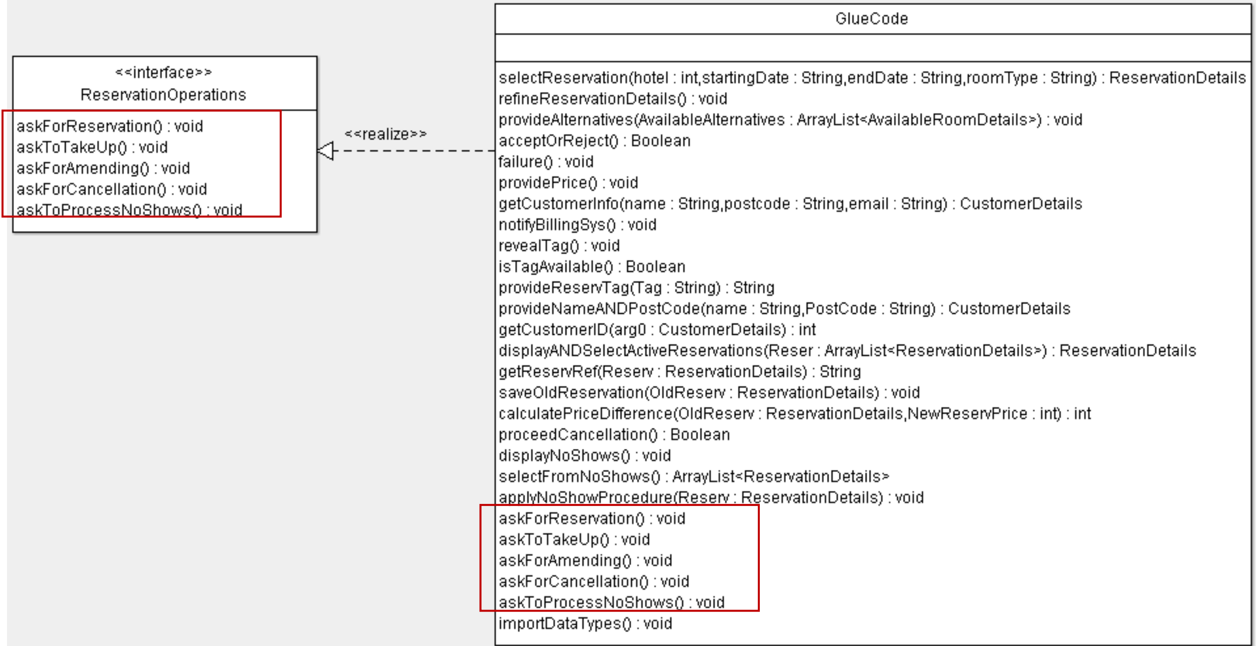


Figure 36: The glue component realizing "*ReservationOperations*" exposed interface

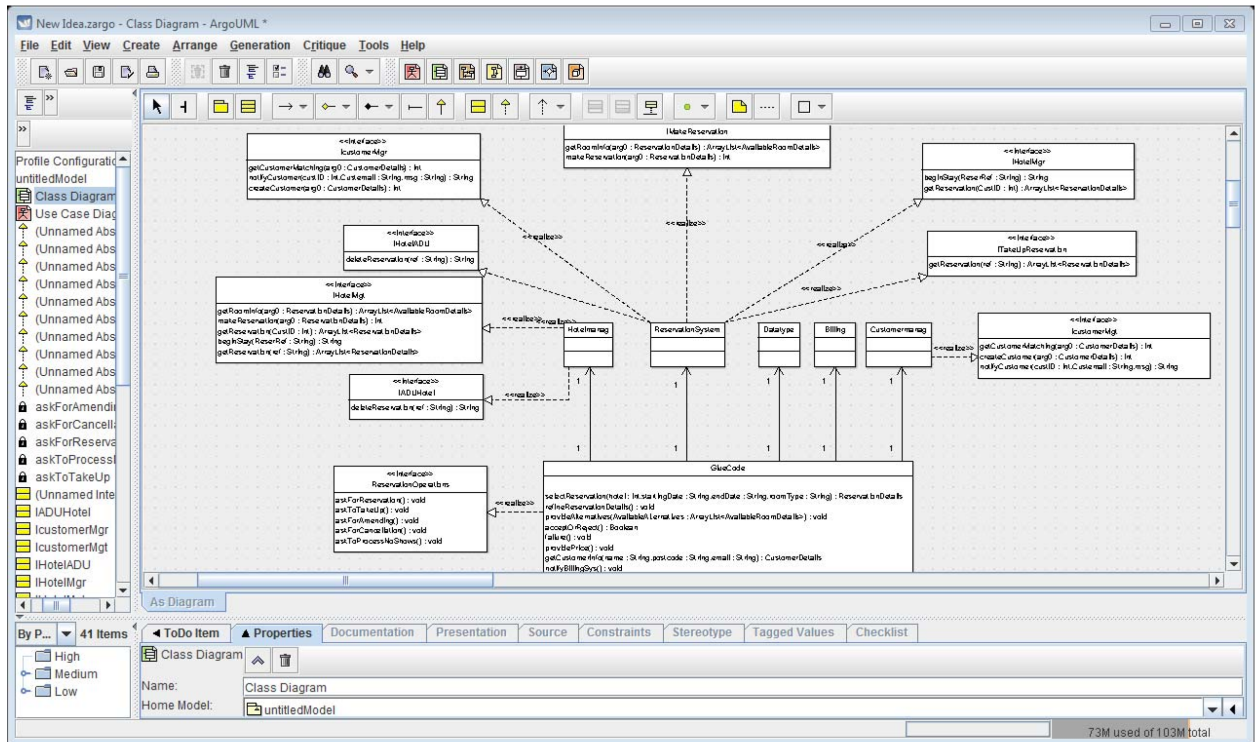


Figure 37: Building OCCD using ArgoUML (Tigris.org) tool

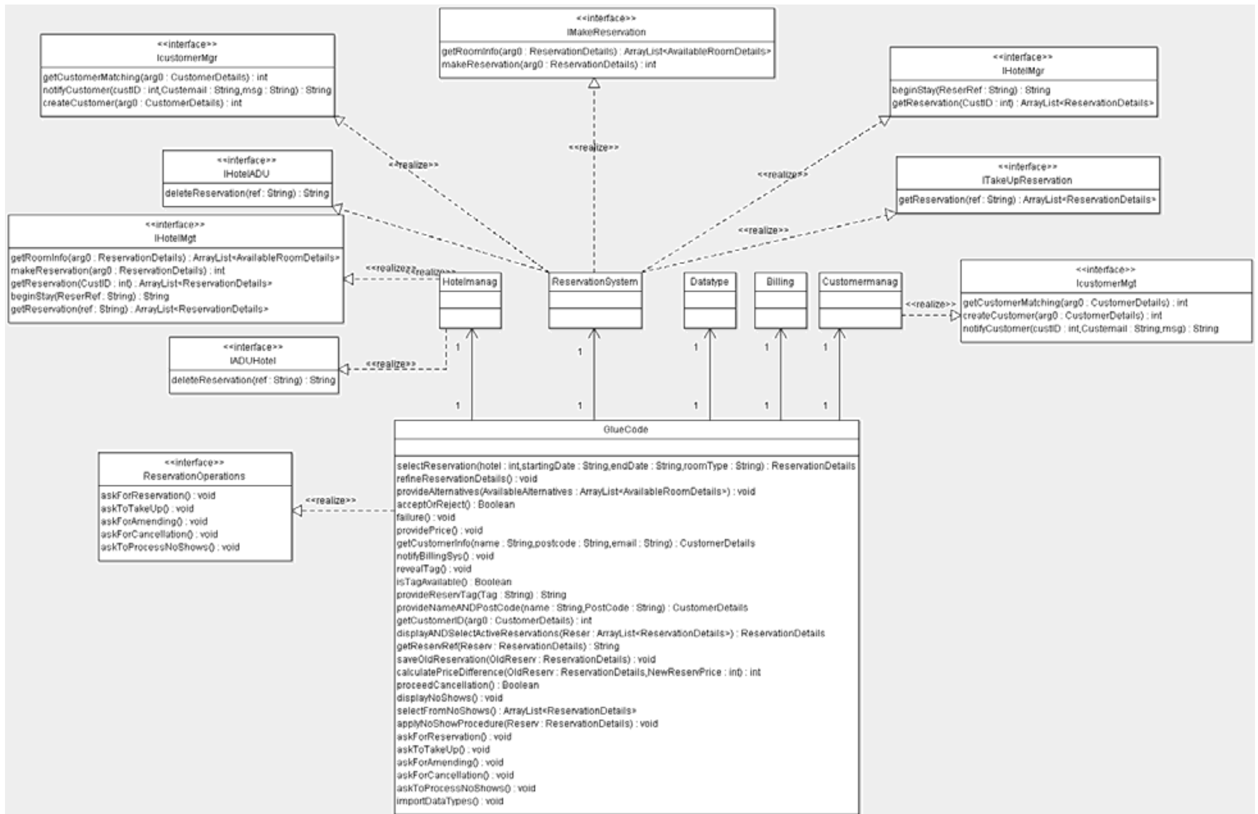


Figure 38 : Class diagram for HRS system

Then, we will add OCL constraints for the resulting class diagram by them to the glue component. We are going to provide some examples:

Let's begin with the glue code component (i.e., class). For instance, we can add a class invariant to it to check the validity of any run-time instance of the glue code component. We can check that using the following invariant:

```
context GlueCode
inv validGlueCode: self.oclIsTypeOf(GlueCode)
```

In USE tool, the class invariants as well as the entire structure of the system will be automatically checked whenever the system state changes. So, there is no need to check the invariants manually.

Also, we can add pre and/or postconditions to any of the missing operations to be implemented within the glue code. For instance, the following pre and postconditions will be annotated to the scenario initiating methods *askForReservation()*:

```
context GlueCode::askForReservation()
pre:self.room->exists(r|r.available=true)
post:(self.room->select(r|r.available=true)->size=
self.room@pre->select(r|r.available=true)->size   -1)   and
self.reservation->size=self.reservation@pre->size+1
```

The precondition will help to make sure that there is still at least one available room that has not been booked, yet. This means that this method will not be executed unless its precondition is satisfied. While in the postcondition, it will check the correctness of the functionality to be provided by checking that there is an increase in the number of the reserved rooms within the hotel.

askToTakeUp() method will have a precondition to check whether there are some reservation to be claimed or not. On the other hand, its postcondition will check the correctness of the method execution by making sure that the number of rooms that have not been claimed yet is decreased by one.

```
context GlueCode::askToTakeUp()
pre:self.reservation->exists(Res|Res.claimed=false)
post:       self.reservation->select(res|res.claimed=false)-
>size=self.reservation@pre->select(res|res.claimed=false)-
>size-1
```

askForCancellation() will have a precondition that is always true but its postcondition will check whether the number of reservation has been decreased by one or not before exiting the method.

```
context GlueCode::askForCancellation()
pre:true
post:self.reservation->size=self.reservation@pre->size -1
```

askToProcessNoShows() will have the following pre and postconditions to check the availability of non-claimed reservations on the entry to the method and to check for the unavailability of any non-claimed reservation before exiting the methods.

```
context GlueCode::askToProcessNoShows()
pre:self.reservation->exists(res|res.claimed=false)
post:self.reservation->exists(res| res.claimed=true)
```

getCustomerInfo() will have an always true precondition. Also, it will have a postcondition for checking the existence of the customer based on his name and post code to make sure that it has been added to the customer database.

```
context  GlueCode::getCustomerInfo(name:  String,  postcode:
String,e-mail:String):CustomerDetails
pre: true
post:self.customer->exists(cust|    cust.name<>name)    and
self.customer->exists(cust|   cust.postCode<>postcode)   and
result.cust_name=name and result.post_code=postcode
```

provideReservTag() have a precondition that will check the existence of the provided reservation tag. Its postcondition returns the value 1.

```
context GlueCode::provideReservTag(Tag : String):String
pre:self.reservation->exists(res|res.resRef=Tag)
post:result=1
```

getCustomerID() checks the availability of the customer in the database and return his id if it is on record.

```
context GlueCode::getCustomerID(arg0: CustomerDetails ):
Integer
pre:self.customer->exists(cust|cust.name=arg0.cust_name)
post: result=arg0.cust_no
```

getReservRef() checks for the existence of the reservation on the entry to method and returns the reservation reference on the exit.

```
context GlueCode::getReservRef(Reserv: ReservationDetails
): String
pre:self.reservation->exists(res|res.resRef
=Reserv.Res_Ref)
post: result=Reserv.Res_Ref
```

*saveOldReservation()*checks for the existence of the reservation on the entry to method and returns the value of 1 at exit.

```
Context GlueCode::saveOldReservation(OldReserv:
ReservationDetails)
pre: self.reservation->exists(res|res.resRef
=OldReserv.Res_Ref)
post: result=1
```

7.2 OCCD Validation

OCL provides special syntax for specifying pre and postconditions on operations in a UML model. Pre and postconditions are constraints that define a contract that an implementation of the operation has to fulfil. A precondition must hold when an operation is called, a postcondition must be true when the operation returns. As mentioned earlier, we will validate the resulting formal glue code specification (i.e. OCCD) using USE tool.

USE can be employed to animate the model and thus validate it according to the system's requirements. These are represented through OCL constraints, which can be evaluated and appraised during the animation of the model. New constraints can also be introduced in OCL, and applied to the model loaded. The USE tool allows validating pre and postconditions by simulating operation calls.

One specification of USE contains a textual description (classes, associations, attributes, operations and constraints) of the previously loaded UML class diagram. The textual description of the model is proper to the tool and does not adhere to any standard. A screenshot of USE tool is shown in figure 39.

Figure 39: USE tool environment

We should create some additional classes to help us in simulating the system. Classes of *Hotel*, *Reservation*, *Room*, and *Customer* will be created. The USE-based specification for the OCCD classes, associations, operation and OCL constraints should be written to show the resulting class diagram drawn by the USE is shown at figure 40.

Figure 40: A part of OCCD drawn using USE tool

Then, we will create run-time objects corresponding to the classes of OCCD as shown by object diagram generated by the USE tool in figure 41.

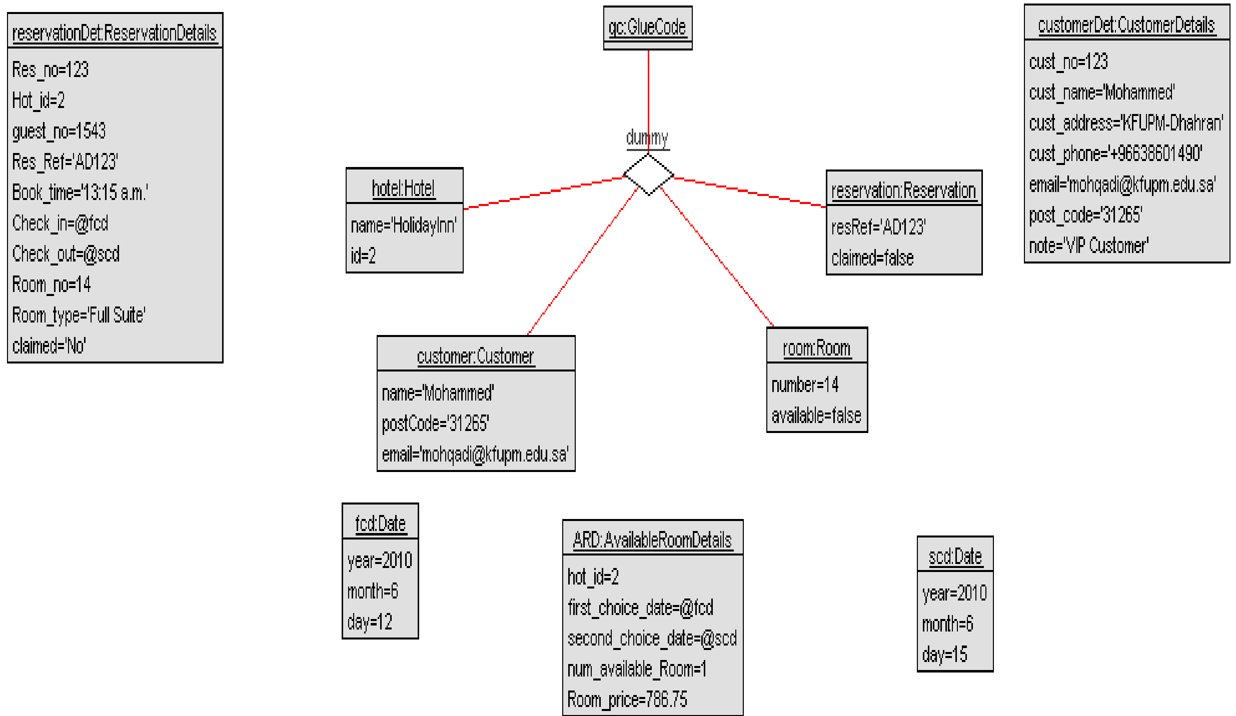


Figure 41: Object diagram corresponding to classes in OCCD

We can make a query using the command “info vars” to check the available objects at the moment as shown in figure 42.

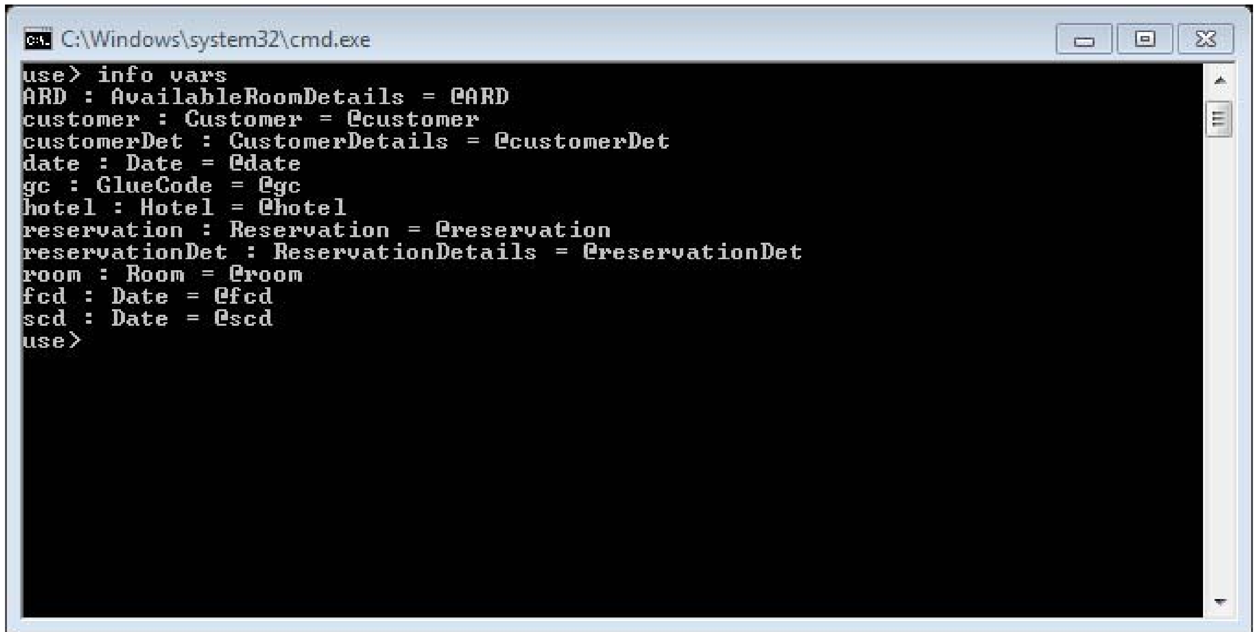


Figure 42: Created objects during USE simulation for the system

Let's now begin simulating the call of *askForReservation()* method as shown in figure 43. We will set the attribute of the single room attribute called "available" to "true" indicating the availability of the room. The simulation will be performed by typing the command "!openter" for checking the precondition and the command "!opexit" for postconditions. The Command "!openter" will take the following form:

!openter <the object of the class to which method belong> <method name signature>

```
C:\Windows\system32\cmd.exe
use> !set room.available:=true
use> !openter gc askForReservation()
precondition `pre127' is true
use> !opexit
postcondition `post155' is false
evaluation results:
  self : GlueCode = @gc
  self.room : Set(Room) = Set{@room}
  r : Room = @room
  r.available : Boolean = true
  true : Boolean = true
  (r.available = true) : Boolean = true
  self.room->select(r : Room | (r.available = true)) : Set(Room) = Set{@room}
  self.room->select(r : Room | (r.available = true))->size : Integer = 1
  self : GlueCode = @gc
  self.room@pre : Set(Room) = Set{@room}
  r : Room = @room
  r.available : Boolean = true
  true : Boolean = true
  (r.available = true) : Boolean = true
  self.room@pre->select(r : Room | (r.available = true)) : Set(Room) = Set{@room
}
  self.room@pre->select(r : Room | (r.available = true))->size : Integer = 1
  1 : Integer = 1
  (self.room@pre->select(r : Room | (r.available = true))->size - 1) : Integer =
```

Figure 43 : Call simulation for *askForReservation()* method

We have seen in figure 43 that the result of precondition was "true" because the precondition has been satisfied. However, the postcondition was "false" due to that there is no reservation has been performed. In figure 44, we have reset the availability of the room to be unavailable and we called precondition which this results in as "false".

```
C:\Windows\system32\cmd.exe
use> !set room.available:=false
use> !openter gc askForReservation()
precondition `pre127' is false
```

Figure 44: Another call simulation for *askForReservation()* method

The following are screenshots for call simulation for *askToTakeUp(), askToProcessNoShows(), provideNameANDPostCode(), getCustomerInfo(), provideReservTag(), getCustomerID(), saveOldReservation(),* respectively.

```
C:\Windows\system32\cmd.exe
use> !set reservation.claimed:=false
use> !openter gc askToTakeUp()
precondition `pre128' is true
use> !opexit
postcondition `post156' is false
evaluation results:
  self : GlueCode = @gc
  self.reservation : Set(Reservation) = Set{@reservation}
  res : Reservation = @reservation
  res.claimed : Boolean = false
  false : Boolean = false
  (res.claimed = false) : Boolean = true
  self.reservation->select(res : Reservation | (res.claimed = false)) : Set(Rese
rvation) = Set{@reservation}
  self.reservation->select(res : Reservation | (res.claimed = false))->size : In
teger = 1
  self : GlueCode = @gc
  self.reservation@pre : Set(Reservation) = Set{@reservation}
  res : Reservation = @reservation
  res.claimed : Boolean = false
  false : Boolean = false
  (res.claimed = false) : Boolean = true
  self.reservation@pre->select(res : Reservation | (res.claimed = false)) : Set(
Reservation) = Set{@reservation}
  self.reservation@pre->select(res : Reservation | (res.claimed = false))->size
```

Figure 45: Call simulation for *askToTakeUp()* method

```
C:\Windows\system32\cmd.exe
use> !set reservation.claimed:=false
use> !openter gc askToProcessNoShows()
precondition `pre130' is true
use> !opexit
postcondition `post158' is false
evaluation results:
  self : GlueCode = @gc
  self.reservation : Set(Reservation) = Set{@reservation}
  res : Reservation = @reservation
  res.claimed : Boolean = false
  true : Boolean = true
  (res.claimed = true) : Boolean = false
  self.reservation->exists(res : Reservation | (res.claimed = true)) : Boolean =
 false
use> !set reservation.claimed:=true
use> !openter gc askToProcessNoShows()
precondition `pre130' is false
use>
```

Figure 46: Call simulation for *askToProcessNoShows()* method

```
C:\Windows\system32\cmd.exe
use> !set customerDet.cust_name:='Mohammed'
use> !set customerDet.post_code:='31265'
use> !openter gc provideNameANDPostCode('Mohammed','31265')
precondition `pre6' is true
use> !opexit customerDet
postcondition `post6' is true
use>
```

Figure 47: Call simulation for *provideNameANDPostCode()* method

```
C:\Windows\system32\cmd.exe
use> !set customerDet.cust_name:='Mohammed'
use> !set customerDet.post_code:='31265'
use> !openter gc getCustomerInfo('Mohammed','31265','mohqadi@kfupm.edu.sa')
precondition `pre7' is true
use> !opexit customerDet
postcondition `post7' is false
evaluation results:
  self : GlueCode = @gc
  self.customer : Set(Customer) = Set{@customer}
  cust : Customer = @customer
  cust.name : String = 'Mohammed'
  name : String = 'Mohammed'
  (cust.name <> name) : Boolean = false
  self.customer->exists(cust : Customer | (cust.name <> name)) : Boolean = fa

  self : GlueCode = @gc
  self.customer : Set(Customer) = Set{@customer}
  cust : Customer = @customer
```

Figure 48: Call simulation for *getCustomerInfo()* method

```
C:\Windows\system32\cmd.exe
use> !set reservation.resRef:='AD123'
use> !openter gc provideReservTag('AD123')
precondition `pre8' is true
use> !opexit 'true'
postcondition `post8' is true
use>
```

Figure 49: Call simulation for *provideReservTag()* method

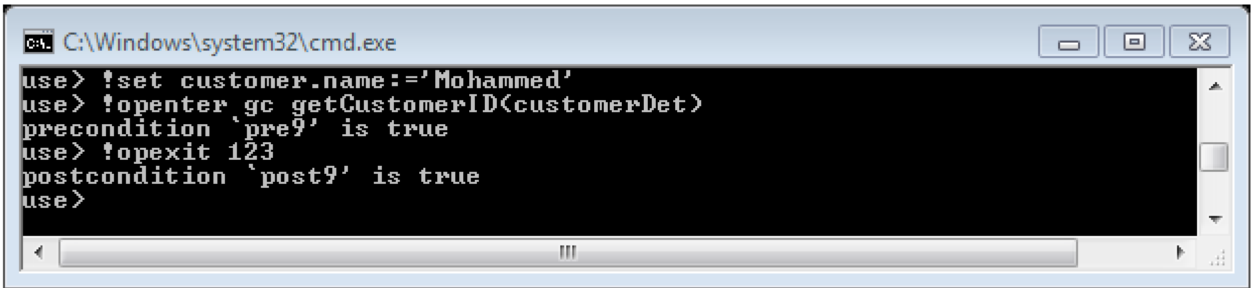


Figure 50: Call simulation for *getCustomerID()* method

Figure 51: Call simulation for *saveOldReservation ()* method

Figure 52: Sequence diagram for the performed calls simulations

7.3 Summary

In this chapter, we have presented an application of the composition stage of our proposed framework to the Hotel Reservation System. CompBSDs of the system resulted from the realization stage that has been presented to the composition stage. We have collapsed them into a single OCCD. Then, we annotated OCL constraints to the diagram.

We converted the final OCCD into USE tool proper specification so that it’s methods pre and postconditions as well as the class invariants could be validated by simulating methods invocations.

CHAPTER 8

CONCLUSION AND FUTURE WORK

In this chapter, the work which has been done is summarized and evaluated. We also present some suggestions for future research work.

8.1 Summary of the Work

As outlined in chapter 1, the aims of this thesis were to:

- Develop a process for CBS integration.
- The framework will address the following issues during integration:
    - Missing primary functionalities.
    - Missing Auxiliary services.
    - Mismatched interfaces.
    - Flow of control.

8.1.1 The Integration Framework

The proposed framework is mainly constituted of two stages: realization and composition. Firstly, use-cases of the system intended to be developed and the documentation of the selected components being integrated will be presented as input into the realization stage. Then, a use-case will be picked at a time. Use-Case Conceptual Mapping (UCCM) diagrams which are considered as static realization for the use-cases

are generated. A tabular form of UCCM diagram may be generated to help in better viewing the interfaces' operations involved in realization of the use-case. Also, the Component-Based Sequence Diagram (CompBSD) will be generated to represent the dynamic realization of that use-case. CompBSDs will show the pre-existing methods (i.e. those provided by components) and missing methods associated with its proper stereotypes. Afterward, the composition stage will take the set of CompBSDs resulting from the realization stage as input. Redundant and crosscutting operations of all lifelines of all CompBSDs will be identified. Then, all system scenarios represented by all CBSDs will be collapsed in a single OCL-Constrained Class Diagram (OCCD) providing the formal specification of the glue code.

One of strengths of the framework is in its being automated. CompBSD and OCCD can be developed by means of one of the available CASE tools by taking advantage of the stereotyping feature. Validation of the OCCD is automated using USE tool. Using OCL is optional depending on the application domain which makes the framework more flexible.

One limitation of the framework is that it has been applied to only one case study. So, there is a need to further analyze potential benefits of the framework to different application domains. In the scope of this work we have not addressed an important research issue of integrating components that have been implemented in a range of programming languages.

8.2 Future Research Work

Based on the evaluation of the work done, we propose future research work as follows:

- Refactoring for glue code interfaces: Since glue code may have more than one interface, things can be simplified by refactoring the interfaces, especially by introducing new abstract interfaces. These will act as super-types for other interfaces, holding common interface information model elements, and, sometimes, definitions of common operations.
- Integration testing for CBS.
- There is a need to investigate the potential benefit of the framework on maintenance and the evolution of CBS application.

REFERENCES

Arlow, J., & Neustadt, I. (2005). *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design* (2 ed.). Addison-Wesley Professional.

Assman, U. (2000). A component model for invasive composition. *ECOOP 2000 Workshop on Aspects and Dimensions of Concerns.* Cannes, France.

Baik, J., Eickelmann, N., & Abts, C. (2001). Empirical software simulation for COTS glue code development and integration. *25th Annual International Computer Software and Applications Conference COMPSAC 2001* (pp. 297-302). Chicago,Illinois,USA : IEEE Computer Society.

Basili, V. R., & Boehm, B. (2001). COTS-Based Systems Top 10 List. *IEEE Computer, 34*(5), 91-95.

Batista, C., Chavez, C., Garcia, A., Kulesza, U., Sant'Anna, C., & Lucena, C. (2006). Aspectual Connectors: Supporting the Seamless Integration of Aspects and ADLs. *ACM SIGSoft XX Brazilian Symposium on Software Engineering (SBES'06).* Florianopolis, Brazil: ACM.

Birmingham, U. o. (n.d.). *UML2Alloy tool*. Retrieved from http://www.cs.bham.ac.uk/~bxb/UML2Alloy/

Boehm, B., & Abts, C. (1999). COTS integration: plug and pray? *IEEE Computer, 32*(1), 135-138.

Brown, W. A., & Wallnau, C. K. (1998). The current state of CBSE. *IEEE Software, 5*(5), 37–47.

Campbel, G. H. (1999). Adaptable components. *21st international conference on Software engineering (ICSE)* (pp. 685-686). Los Angeles, California, United States: IEEE Press.

Canal, C., Murillo , J. M., & Poizat, P. (2005). Report from the ECOOP 2004 Workshop on Coordination and Adaptation Techniques for Software Entities. *Lecture Notes in Computer Science, 3344*, 133-147.

Canal, C., Poizat, P., & Salaun, G. (2008). Model-Based Adaptation of Behavioral Mismatching Components. *IEEE Transactions on Software Engineering, 34*(4), 546-563.

Cechich, A., Piattini, M., & Vallecillo, A. (2003). Assessing Component-Based Systems. *Lecture Notes in Computer Science, 2693*, 1–20.

Cheesman , J., & Daniels , J. (2000). *UML Components: A Simple Process for Specifying Component-Based Software* . Boston,MA,USA: Addison-Wesley.

Chi, Z. (2009). Software components composition compatibility checking based on behavior description. *IEEE International Conference on Granular Computing (GRC '09)* (pp. 757-760). Nanchang,China: IEEE.

Chiang, C.-C., & Ford, C. W. (2005). Maintainability and reusability issues in CORBA-based systems. *43rd annual southeast regional conference. 2*, pp. 275 - 280. Kennesaw, Georgia: ACM.

Crnkovic, I., & Larsson, M. (2002). *Building Reliable Component-Based Software Systems.* Norwood, MA: Artech House, Inc.

Crnkovic, I., Chaudron, M., & Larsson, S. (2006). Component-Based Development Process and Component Lifecycle. *International Conference on Software Engineering Advances (ICSEA'06)* (p. 44). Tahiti, French Polynesia: IEEE.

Dietrich, S. W., Patila, R., Sundermiera , A., & Urbana, S. D. (2006). Component adaptation for event-based application integration using active rules. *Journal of Systems and Software, 79*(12), 1725-1734.

Garlan, D., & Shaw, M. (1994). *An Introduction to Software Architecture.* Carnegie Mellon University, School of Computer Science .

Garlan, D., Allen, R., & Ockerbloom, J. (1995). Architectural mismatch: Why reuse is so hard? *IEEE Software*, 17–26.

Gomez, J. M., Alor-Hernandez, G., Posada-Gomez, R., Rivera, I., Mencke , M., Chamizo , J., et al. (2008). An Approach for Component-Based Software Composition. *Electronics, Robotics and Automotive Mechanics Conference, 2008. (CERMA '08)* (pp. 195-200). Cuernavaca, Morelos, Mexico: IEEE.

Hardy, M. G. (2000). COTS Components in Software Development. *Proceedings of the Computer Science Discipline Seminar Conference (CSCI 3901).* Minnesota, Morris,USA.

Heineman , G. T., & Councill , W. T. (2001). *Component-Based Software Engineering: Putting the Pieces Together.* Reading: Addison-Wesley Professional.

Heineman, G. T. (1999). An evaluation of component adaptation techniques. *ICSE’99 Workshop on CBSE.*

Hepner, M., Gamble, R. F., Kelkar, M., & Davis, L. A. (2006). Patterns of conflict among software components. *Journal of Systems and Software, 79*(4), 537-551.

Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis.* Cambridge, MA.: MIT Press.

Kim , S., Park, S., Yun, J., & Lee, Y. (2008). Automated Continuous Integration of Component-Based Software: An Industrial Experience. *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)* (pp. 423-426). L'Aquila, Italy: IEEE/ACM.

Konstantas, D. (1995). Interoperation of object oriented application. In O. Nierstrasz, & D. Tsichritzis, *Object-oriented software composition* (pp. 69–95). Prentice Hall.

Kouroshfar, E., Shahir, H. Y., & Ramsin, R. (2009). Process Patterns for Component-Based Software Development. *12th International Symposium on Component-Based Software Engineering (CBSE'09). LNCS 5582*, pp. 54-68. Springer.

Kvale, A. A., Li , J., & Conradi , R. (2005). A case study on building COTS-based system using aspect-oriented programming. *ACM symposium on Applied computing* (pp. 1491-1498). Santa Fe, New Mexico,USA: ACM.

Lamsweerde, A. V. (2009). *Requirements Engineering: From System Goals to UML Models to Software Specifications.* Wiley.

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (Third ed.). Addison Wesley Professional.

Lee, J.-S., & Bae, D.-H. (2004.). An aspect-oriented framework for developing component-based software with the collaboration-based architectural style. *Information & Software Technology, 46*(2), 81-97.

Li, J., Conradi, R., Bunse, C., Torchiano, M., Slyngstad, O., & Morisio, M. (2009). Development with Off-The-Shelf Components: 10 Facts. *IEEE Software, 26*(2).

Mahmood, S., Li, R., & Kim, Y. (2007). Survey of component-based software development. *IET Software*, 57-66.

Mezini , M., & Ostermann, K. (2002). Integrating independent components with on-demand remodularization. *17th ACM SIGPLAN Conference Object-Oriented Programming, Systems, Languages, and Appl.* (pp. 52-67). Seattle, Washington, USA : ACM.

OMG. (2005). *OCL 2.0 Specification.* OMG.

Rader, J. A. (1997). Mechanisms for Integration and Enhancement of Software Components. *Fifth International Symposium on Assessment of Software Tools and Technologies (SAST'97)* (pp. 24–31). Pittsburgh: IEEE Computer Society.

Richters, M. (2001). *UML-based Specification Environment tool*. Retrieved from http://www.db.informatik.uni-bremen.de/projects/USE/

Rine, D., Nada, N., & Jaber, K. (1999). Using Adapters to Reduce Interaction Complexity in Reusable Component-Based Software Development. *Symposium on Software Reusability* (pp. 37-43). Los Angeles,Cailfornia,USA: ACM.

Sommerville, I. (2007). *Software Engineering 8.* Boston, USA: Addison-Wesley.

Suvee, D., Vanderperren, W., & Jonckers, V. (2003). JAsCo: an aspect-oriented approach tailored for component based software development. *Second International Conference on Aspect-Oriented Software Development* (pp. 21-29). Boston, Massachusetts, USA: ACM.

Szyperski , C., Gruntz , D., & Murer, S. (2002). *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley/ACM Press.

Tigris.org. (n.d.). *ArgoUML.* Retrieved from ArgoUML: http://argouml.tigris.org/

Truyen, E., J¨orgensen, B. N., Joosen, W., & Verbaeten, P. (2000). Aspects for run-time component integration. *ECOOP 2000 Workshop on Aspects and Dimensions of Concerns.* Sophia Antipolis and Cannes, France: Springer.

Vigder, M. R., & Dean, J. (1997). An architectural approach to building systems from COTS software components. *Conference of the Centre for Advanced Studies on Collaborative research* (p. 22). Toronto, Ontario, Canada: IBM Press.

Wegner , P. (1996). Interoperability. *ACM Computing Surveys, 28*(1), 285 - 287.

Zitouni, A., Seinturier, L., & Boufaida, M. (2008). Contract-Based Approach to Analyze Software Components. *13th IEEE International Conference on Engineering of Complex Computer Systems, 2008. (ICECCS 2008)* (pp. 237-242). Belfast, Northern Ireland: IEEE Computer Society.

APPENDIX

Selected Components Documentation for HRS Case Study

1. **org.eclipse.Billing:** has only one interface named "IBilling" contains just one operation
    1. IBilling:
        1. *openAccount(ReservationDetails arg0, CustomerDetails arg1):void for opening account for a customer containing his payments.*
2. **org.eclipse.customermanag:** has two interfaces
    1. IADUCustomer: containing the following operations:
        1. deleteCustomer(int arg0):int for deleting a customer by passing his customer_ID
        2. updateCustomer(CustomerDetails arg0):int for updating the customer information by entering an entire object pertaining to ***CustomerDetails*** class encapsulating all information. *CustomerDetails* encapsulates customer number, customer name, customer address, customer phone, customer e-mail, customer post code and note, respectively.
    2. IcustomerMgt
        1. createCustomer(CustomerDetails arg0):int for adding a new customer's record of type CustomerDetails to the database.
        2. getCustomerDetails(int arg0):ArrayList<CustomerDetails> for retrieving the customer details from the database.
        3. getCustomerMatching(CustomerDetails arg0):int for checking whether a customer do exist in the database or not using his details
        4. getCustomers( ) :ArrayList<CustomerDetails> for showing up all existing registered customers within the hotel.
        5. notifyCustomer(int arg0, String arg1, String arg2):String for sending a message for notifying the user.

3. **org.eclipse.datatype**: has no interface but contains the definition for the data types used through the application as follows:

AvialableRoomDetails

```
int hot_id;
Date first_choic_date;
Date second_choic_date;
int num_available_Room;
float Room_price;
```

CustomerDetails

```
int cust_no;
String cust_name;
String cust_address;
String cust_phone;
String e-mail;
String post_code;
String note;
```

HotelDetails

```
int Hot_Id;
String Hot_name;
int Room_num;
String Hot_address;
String Hot_phone;
String Hot_fax;
String Hot_e-mail;
```

InOutGuestDetails

```
int G_num;
int hot_id;
String G_name;
Date Check_in=null;
Date Check_out=null;
String referen;
String Cliamed;
```

ReservationDetails

```
int Res_no;
int Hot_id;
int guest_no;
String Res_Ref;
String Book_time;
Date Check_in=null;
Date Check_out=null;
int Room_no;
String Room_type;
String claimed;
```

Roomdetails
int room_acount;
float room_price;

RoomDetials
int hot_id;
String code;
int Room_number;
String Room_type;
String phone;
String available;
String note;
int price;

RoomTypeDetials
int room_type_no;
String room_type;
String note;

4. **org.eclipse.hotelmanag:** has two interfaces
    1. IADUHotel
        1. DeleteRoom(RoomDetials arg0):String for deleting a room from the list of room in a hotel from the database.
        2. UpdateRoom(RoomDetials arg0):String update the information about a specific existing room in the database.
        3. checkReservation(int arg0, int arg1, int arg2):int to check the existence of a reservation.
        4. createHotel(HotelDetails arg0):String creating a new hotel in hotels chain.
        5. createRoom(RoomDetials arg0):String create a new room in a specific hotel.
        6. createRoomType(RoomTypeDetials arg0):String for introducing a new type of rooms within the hotels chain.
        7. deleteHotel(int arg0):String for deleting one of the hotels of the chain from the database.
        8. deleteReservation(String arg0):String for deleting a record for a reservation from the database.
        9. deleteRoomType(int arg0):String for deleting a type of rooms within the hotels chain from the database.
        10. updateHotel(HotelDetails arg0, int arg1):String for modifying the information about a specific hotel in the database.

        11. updateReservation(ReservationDetails arg0):int to modify information about a specific reservation.
        12. updateRoomType(RoomTypeDetials arg0):String for modifying the information about a type of rooms within the hotels chain.
    2. IHotelMgt
        1. beginStay(String arg0):String updating the reservation record's field named "claimed" with "True" value indicating that the stay began.
        2. getHotelDetails( ): ArrayList<HotelDetails> return the information about all hotels within the chain.
        3. getReservation(String arg0): ArrayList<ReservationDetails> get the information of a specific reservation by querying using its reservation reference.
        4. getReservation(int arg0): ArrayList<ReservationDetails> get the information of a specific reservation by querying using the customer ID.
        5. getReservation(Date arg0, int arg1): ArrayList<ReservationDetails> querying about a specific reservation by displaying a list of reservation to be looked for.
        6. getRoomInfo(ReservationDetails arg0):ArrayList<AvailableRoomDetails> retrieving the available unreserved rooms within the hotels chain.
        7. getRooms(int arg0):ArrayList<RoomDetials> showing the full list of rooms of a specific hotel within the chain by passing its Hotel ID.
        8. makeReservation(ReservationDetails arg0):int reserving a room for a customer after entering his full details.

5. **org.eclipse.ReservationSystem**: has six interfaces
    1. ICustomerADU**:** All operations here are doing the same functionality of their counterparts in **org.eclipse.customermanag** component by delegating the functionality to them.
        1. deleteCustomer(int arg0):int
        2. updateCustomer(CustomerDetails arg0):
    2. IcustomerMgr: All operations here are doing the same functionality of their counterparts in **org.eclipse.customermanag** component by delegating the functionality to them.
        1. createCustomer(CustomerDetails arg0):int

2. getCustomerDetails(int arg0):ArrayList<CustomerDetails> getCustomerMatching(CustomerDetails arg0):int
3. getCustomers( ) :ArrayList<CustomerDetails>
4. notifyCustomer(int arg0, String arg1, String arg2):String

3. IHotelADU**:** All operations here are doing the same functionality of their counterparts in **org.eclipse.hotelmanag** component by delegating the functionality to them.
    1. DeleteRoom(RoomDetials arg0):String
    2. UpdateRoom(RoomDetials arg0):String
    3. checkReservation(int arg0, int arg1, int arg2):int
    4. createHotel(HotelDetails arg0):String
    5. createRoom(RoomDetials arg0):String
    6. createRoomType(RoomTypeDetials arg0):String
    7. deleteHotel(int arg0):String
    8. deleteReservation(String arg0):String
    9. deleteRoomType(int arg0):String
    10. updateHotel(HotelDetails arg0, int arg1):String
    11. updateReservation(ReservationDetails arg0):int
    12. updateRoomType(RoomTypeDetials arg0):String
4. IHotelMgr**:** All operations here are doing the same functionality of their counterparts in **org.eclipse.hotelmanag** component by delegating the functionality to them.
    1. beginStay(String arg0):String
    2. getReservation(Date arg0, int arg1): ArrayList<ReservationDetails>
    3. getReservation(int arg0): ArrayList<ReservationDetails>
    4. getRooms(int arg0):ArrayList<RoomDetials>

5. IMakeReservation
    1. getHotelDetails( ): ArrayList<HotelDetails> doing the same functionality of its counterpart in **org.eclipse.hotelmanag** component by delegating the functionality to them.
    2. getInOutCustomers(Date arg0, int arg1): ArrayList<InOutGuestDetails> it displays all checked in and out customers within the hotels chain.
    3. getRoomInfo(ReservationDetails arg0):ArrayList<AvailableRoomDetails> doing the same functionality of its counterpart in **org.eclipse.hotelmanag** component by delegating the functionality to them.
    4. makeReservation(ReservationDetails arg0):int doing the same functionality of its counterpart in **org.eclipse.hotelmanag** component by delegating the functionality to them.

6. ITakeUpReservation
   1. beginStay(String arg0,int arg1):String begin the Stay by acknowledging the system of the customer attendance then RMI will be done to call beginStay( ) method of **org.eclipse.hotelmanag.**
   2. getReservation(String arg0): ArrayList<ReservationDetails> doing the same functionality of its counterpart in **org.eclipse.hotelmanag** component by delegating the functionality to them.

CURRICULUM VITA

Name: Mohammed Abdullah Ali Al-Qadhi

Date/ Place of Birth of Birth: 16 Feb 1984/ Jazan, Samita, Saudi Arabia

Nationality: Yemeni

Marital Status: Single

Present and Permanent Address: P.O. Box 607, Samita 45922, Jazan, Saudi Arabia

Contact: +966(0)590784224

E-mail: mqadhi@hotmail.com

Education:

- **Master of Science in Computer Science from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia 2010.**
- **Bachelor of Science Computer Science from King Khalid University, Abha, Saudi Arabia 2007, with being ranked 1st in my batch.**
- **High School Diploma from Najamia High School, Jazan, Saudi Arabia 2001, with a percentage of 95.76/100.**