INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

Robustness Analysis

A Bridging Technique
From System Requirements to Design

By

Antonio Salameh

A thesis submitted to

The School of Graduate Studies and Research

In partial fulfillment of the degree of

Master in Computer Science

School of Information Technology and Engineering

University of Ottawa

(Ottawa-Carleton Institute for Computer Science)

Robustness Analysis

A Bridging Technique From System Requirements to Design

By
Antonio Salameh

ABSTRACT

Robustness analysis, a technique proposed by Jacobson, can be used in a use-case-driven process to help close this gap between requirements and design by examining in depth the requirements found in the narrative text of each use case and structuring it into a *robustness diagram*.

In this thesis, we present a step by step procedure for performing *robustness analysis*. In particular, we describe how to construct and use *robustness diagrams* to discover analysis and design errors early in the process, and to verify the correctness of the use case text. Moreover, we propose a new metric, *robustness analysis points*, to estimate project size and effort and we compare our results to *function points* and *use case points* in three case studies. Finally, we assess and evaluate the RA technique and conclude by giving recommendations for future work.

Acknowledgements

I would like to thank the many individuals who have made this thesis possible. I am especially graceful to my supervisor Dr. Robert L. Probert who helped choose the thesis topic and provided guidance, valuable support and encouragement during this research.

I would like also to thank Dr. Micheal Weiss from Carleton University and Dr. Timothy Lethbridge from the University of Ottawa for their valuable suggestions and comments. I would also like to thank Louise Defrochers for her unconditional help.

My deepest thanks go to my family (my father Jean and my sisters Rita, Micheline and Patricia), whose support and encouragement have helped me pursue my education.

Last but not least, my deepest love go to my wife Maya for her love, support and understanding during the whole period that went into conducting this research and writing the thesis.

To my wife Maya and my daughter Marie-Josée

&

To the memory of my mother Josephine

TABLE OF CONTENTS

ABSTRACT

Over the past decades, object-oriented analysis and design has grown and received great attention in government, industry and academia. This is due to the rise of various underlying object-oriented methodologies that facilitate the modeling, understanding and reuse of software. However, the main challenge has been to ensure the consistent and correct transition from analysis (*what* must be done) to design (*how* it should be done). The main impediment is a semantic gap between requirements expressed in customer language (*use-case model*) and requirements expressed in developer language (*design model*).

Robustness analysis, a technique proposed by Jacobson, can be used in a use-case-driven process to help close this gap between requirements and design by examining in depth the requirements found in the narrative text of each use case (customer language) and structuring it into a *robustness diagram* (developer language). However, a detailed set of guidelines for conducting robustness analysis has not yet been published.

In this thesis, we present a step by step procedure for performing *robustness analysis*. In particular, we describe how to construct and use *robustness diagrams* to analyze infeasible specifications, to discover missing domain analysis classes, to find design errors early in the process, to verify the correctness of the use case text, and to assess the completeness of the analysis phase. This is illustrated by the analysis and partial design of an on-line stock broker case study. Moreover, we propose a new metric, *robustness analysis points*, to estimate project size and effort. We compare our results to *function points* and *use case points* in three case studies. We assess and evaluate the RA technique and conclude by giving recommendations for future work. The results of our case study support the use of the methodology for improving robustness of high-level design. However, more study should be done particularly to test the suitability of *Robustness Analysis Points* as a software effort-estimation metric.

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ASP	Active Server Pages
DET	Data Element Type
EF	Environmental Factor
EI	External Input
EIF	External Interface File
EFactor	Environment Factor
EO	external Output
EQ	External Inquiry
Flevel	Environment Level
FP	Function Points
FTR	File Type Referenced
IFPUG	International Function Point Users Group
ILF	Internal Logical File
ILF	Internal Logical File
IUP	ICONIX Unified Process
JSP	Java Server Pages
MTL	Model Traceability Links
MVC	Model View Controller
NAB	North American Broker
NW	National Widget
NYSE	New York Stock Exchange
OMG	Object Management Group
OMT	Object Modeling Technique
OOA	Object Oriented Analysis
OOP	Object Oriented Programming
OOSE	Object-Oriented Software Engineering

ORB	Object Request Broker
OSB	On-line Stock Broker
RA	Robustness Analysis
RAP	Robustness Analysis Points
RD	Robustness Diagram
RET	Record Element Type
RFP	Request For Proposal
RP	Robustness Points
RUP	Rational Unified Process
SBM	Solution Based Modeling
TCF	Technical Complexity Factor
TFactor	Technical Factor
TLevel	Technical Level
TMS	Time Management System
TSE	Toronto Stock Exchange
UAW	Unadjusted Actor Weights
UBW	Unadjusted Boundary Weights
UCP	Use Case Points
UCW	Unadjusted Control Weights
UEW	Unadjusted Entity Weights
UFP	Unadjusted Function Point
URAP	Unadjusted Robustness Analysis Points
UML	Unified Modeling Language
UUCP	Unadjusted Use Case Points
UUCW	Unadjusted Use Case Weight
VAF	Value Adjustment Equation
WSP	Warehouse Software Portfolio

CHAPTER 1

INTRODUCTION TO THESIS

1.1 BACKGROUND & MOTIVATION

*I*t was not until the late 80's that Object Oriented Programming (OOP) became popular with programming languages such as C++ and Smalltalk. With the success and booming of OOP, the need for methods to support software development led to the creation of dozens of methodologies and notations in the early 90's. The most popular methods were the Booch Method [BOOCH 94], Object Modeling Technique (OMT) [RUMBAUGH 91], Object-Oriented Software Engineering (OOSE) [JACOBSON 92], Object Oriented Analysis (OOA) by Coad and Yourdon, and Solution Based Modeling (SBM) by Goldstein and Alger.

In an effort to end the *method wars*, Rational Software, the intellectual home of Grady Booch, accelerated this process by hiring Jim Rumbaugh in October 94, and by merging with Ivar Jacobson's firm Objectory AB in fall 95 [DBMS 96]. The result was the Rational Objectory Process, a predecessor of the *Rational Unified Process* (RUP).

Robustness Analysis (RA), which occurs during the elaboration phase, links analysis to design by examining in depth the requirements found in the narrative text of each use case (1) to identify and classify classes into b*oundary*, c*ontrol* and e*ntity* classes, and (2) to construct the associated Robustness Diagram (RD) from the identified classes.

[JACOBSON 92] first introduced the concept of RA in his original OOSE process. He differentiates between the process of requirements analysis, which has as input a requirements specification and produces a requirements model, and the process of RA, which investigates this model concerning its robustness, and produces the analysis model.

In RUP [JACOBSON 99A], RA is simply called analysis and produces a high-level analysis model that has two purposes. The first purpose is to refine the use cases in more detail. The second purpose is to make an initial allocation of the behavior of the system to a set of classes that provides the behavior. However, it is unclear how the analysis model is derived, and how it refines use cases.

[ROSENBERG 99] adopted RA as an integral part of his *ICONIX Unified Process* (IUP) [ICONIX 01]. In IUP, RA analyzes the narrative text found in each use case and produces RD. Although the RA technique has well been defined and described, the clarity, and procedural details required to conduct RA and to produce RD are lacking.

In this thesis, we favor the use of the RA technique and terminology over analysis for several reasons. RA (1) refines use cases, (2) validates and verifies user requirements, (3) uncovers errors early in the process, (4) assesses the completeness of the analysis phase, and (5) results in a robust object structure and design. In addition, we prefer IUP to RUP because RA is an essential part of IUP; however, RA can apply equally well to RUP.

The problem addressed by this thesis is:

1. The ability to make a consistent and correct transition from analysis (*what* must be done) to design (*how* it should be done). The main impediment is a semantic gap between requirements expressed in customer language (use-case model) and requirements expressed in developer language (design model).

2. To show in details the steps needed to perform RA and construct its related RD for each use case.

3. To demonstrate the usefulness of applying this technique during the analysis phase to better understand and maintain the requirements, to view the structure of the whole system, and to verify and validate requirements.

1.2 CONTRIBUTIONS OF THE THESIS

The major contributions of this thesis are:

1. We distilled out and made explicit the construction rules in [ROSENBERG 99] and we added a new rule to simplify RDs. Also we gave a complete methodology description, including complete guidelines for constructing RDs.

2. We made explicit the RA steps implied by Rosenberg's description and gave an example cost-benefit analysis of RA based validation. Our result can be used as a foundation for explaining and teaching the RA technique.

3. We assessed the quality of the RA approach by a realistic case study on-line stock broker.

4. We proposed and illustrated a new software effort-estimation technique, Robustness Analysis Points (RAP). RAP was calculated for three case studies and the results were compared to *function points* (FP) and *use case points (UCP).*

5. We showed a Quality Assessment approach and illustrated it by demonstrating how RA can detect three serious errors in requirements: infeasible specifications, class omissions, and design errors.

6. We evaluated the RA technique from a cost-benefit analysis viewpoint. We showed the relevant contributions of this technique to Software Quality and the architecture pattern that results.

1.3 ORGANIZATION OF THE THESIS

This thesis is organized as follows:

- **Chapter 2:** Introduces the *Unified Modeling Language* (UML) and the two unified UML processes RUP and IUP that are considered in this thesis.

- **Chapter 3:** Defines RA within the context of RUP and IUP and demonstrates how to conduct it and how to construct RD for each use case.

- **Chapter 4:** Introduces the on-line stock broker case study along with required preliminary requirements prior to performing RA.

- **Chapter 5:** In this chapter, we continue with our case study from Chapter 4. We introduce and categorize the type of errors that RA can detect. We conduct RA and construct RD for three selective use cases that validate and verify use case requirements.

- **Chapter 6:** In this chapter, we assess the RA technique from a cost-benefit analysis viewpoint.

- **Chapter 7:** In this chapter, we propose a new metric, RAP, to better estimating software size and effort, and we compare our results to FP and UCP with three case studies.

- **Chapter 8:** Concludes the work and proposes future research topics in the area of O-O analysis and design.

CHAPTER 2

BACKGROUND: UML AND UNIFIED UML PROCESSES

*T*oday, many object-oriented software development methodologies have been developed and used; however, when they are viewed from a sufficiently high-level viewpoint, they appear similar. In this chapter, we give an overview of the UML, and we review two popular development processes that are considered in this thesis: RUP and IUP.

2.1 UML

In this section, we give the definition and a brief history of the UML then we review UML's primary design goals and artifacts.

2.1.1 DEFINITON

UML is the new standard for describing artifacts of an object-oriented development process [JACOBSON 99B], [JACOBSON 99C]. The UML definition was led by Rational Software's industry-leading three guru methodologists as the natural successor to object modeling languages of three previously leading object-oriented methods (Booch [BOOCH 94], Object Modeling Technique (OMT) [RUMBAUGH 91], and Object Oriented Software Engineering (OOSE) [JACOBSON 92]).

> *"UML is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. UML represents a collection of the best engineering practices that have proven successful in the modeling of large and complex systems." [RATIONAL 01B]*

In our opinion, UML is an important language for specifying, visualizing, constructing, and documenting the artifacts of software systems because it helps the development team to communicate unambiguously their decisions using the same language.

2.1.2 HISTORY OF UML

UML evolved from various second-generation object-oriented methods during the 1990s, starting with Rational Software Corporation and three of the most prominent methodologists in the information systems and technology industry: Grady Booch, James Rumbaugh, and Ivar Jacobson. The language has gained significant industry support from various organizations via the UML Partners Consortium and was approved by the Object Management Group (OMG) [OMG] as a standard in September 1997 as shown in Figure 2.1.

The development of UML began in October 1994 when Grady Booch and Jim Rumbaugh of Rational Software began their work on unifying the Booch and OMT methods. The result was a draft version 0.8 of the Unified Method in October 1995.

In Fall 1995, Ivar Jacobson and his Objectory company joined Rational Software bringing in the OOSE method. This unification effort of Booch, Rumbaugh, and Jacobson resulted in the release of UML 0.9 and 0.91 documents in June and October of 1996.

When OMG issued a Request For Proposal (RFP), Rational Software established the UML Partners consortium with leading computer and software companies such as HP, Microsoft,

Oracle, Unisys and IBM that produced UML 1.0, a modeling language that was well defined, expressive, powerful and widely applicable. It was submitted to the OMG in January 1997 as an initial RFP response.

In January 1997 separate RFP responses by IBM, ObjectTime, Platinum Technology, Ptech, Taskon, Reich Technologies and Softeam were incorporated by the consortium into UML 1.1. This improved the clarity of UML 1.0 semantics. It was submitted to the OMG for their consideration and adopted in November 1997.

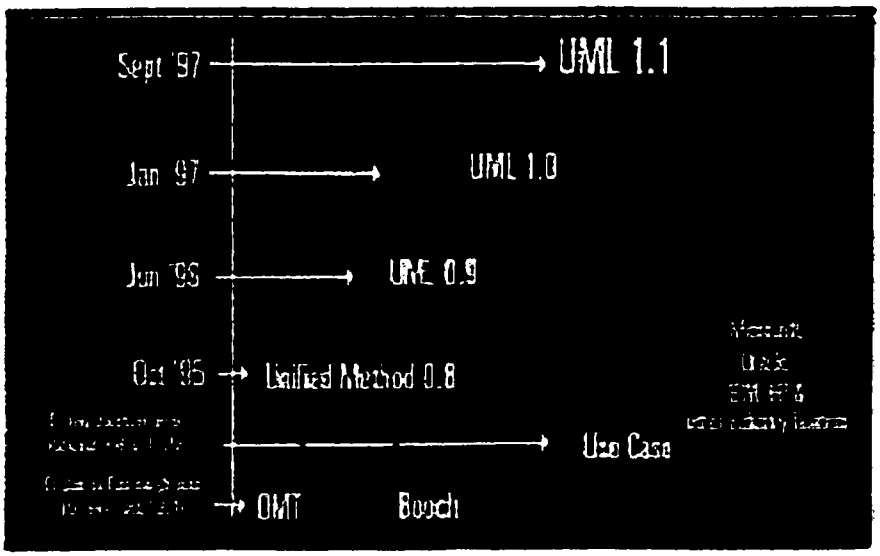


Figure 2.1: UML History

From [RATIONAL 01B]

The OMG chartered a revision task force to accept comments from the general public and to make revisions to the specifications to handle bugs, inconsistencies, ambiguities, and minor omissions that could be handled without a major change in scope from the original proposal. The result was UML 1.3, the current version at the time of this writing.

2.1.3 UML GOALS

The design goals of UML include:

- Provide users with a ready-to-use, expressive visual modeling language to develop and exchange meaningful models;
- Furnish extensibility and specialization mechanisms to extend the core concepts;
- Support specifications that are independent of particular programming languages and development processes;
- Provide a formal basis for understanding the modeling language;
- Encourage the growth of the object tools market;
- Support higher-level development concepts such as components, collaborations, frameworks and patterns; and,
- Integrate best practices in object-oriented development.

2.1.4 UML ARTIFACTS

An artifact is a piece of information that is produced by the workers of a process. Artifacts can be models, model elements or documents. The primary artifacts of the UML are viewed from two different perspectives: the UML definition itself (UML Semantics, UML Notation Guide, and UML Standard Profiles) and how it is used to produce project artifacts as shown in Table 2.1:

a) Use Case diagram	b) Class diagram
c) Behavior diagrams 1) Statechart diagram 2) Activity diagram 3) Interaction diagrams: i) Sequence diagram ii) Collaboration diagram	d) Implementation diagrams: 1) Component diagram 2) Deployment diagram

Table 2.1: UML Artifacts

2.2 RATIONAL UNIFIED PROCESS

In this section, we give the definition of RUP in literature, and we overview its lifecycle, phases and core workflow.

2.2.1 DEFINITION

RUP is a software engineering process that provides a disciplined approach to assigning tasks and responsibilities within a development organization [KRUCHTEN 98]. It is a component-based, use-case-driven, architecture centered, iterative and incremental developmental process. It uses the UML to represent models of the software system to be developed. It describes, apart from the unified generic process and the different activities in developing a software system, the different models developed and evolved during the lifecycle of a system [JACOBSON 99A].

2.2.2 RUP LIFE CYCLE

RUP iterates over a series of cycles making up the life of a system. Each cycle consists of four phases: inception, elaboration, construction, and transition. Each phase is further subdivided into iterations. A typical iteration goes through all Core Workflows and consists of requirements, analysis, design, implementation and test as shown in Figure 2.2.

On the other hand, [ROYCE 98] defines two stages, an engineering stage and a production stage, that make up the life cycle of a system, then decomposes them into the four RUP phases:

1. The Engineering Stage, driven by less predictable but smaller teams doing design and synthesis activities, is decomposed into two distinct phases: inception and elaboration.

2. The Production Stage, driven by more predictable but larger teams doing construction, test, and deployment activities, is decomposed into the two other phases: construction, and transition.

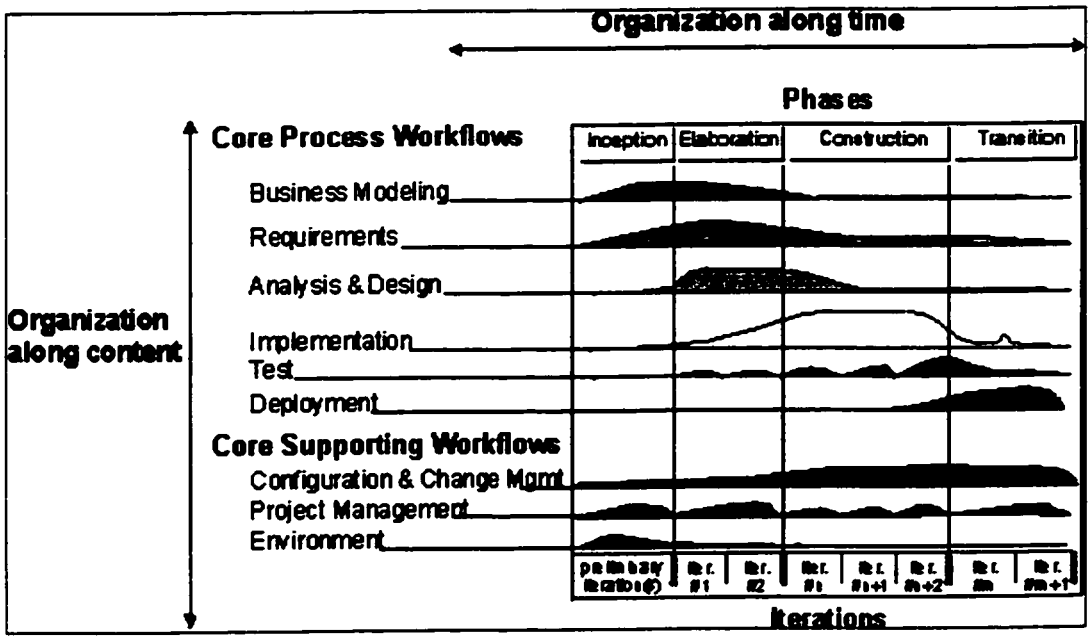


Figure 2.2: RUP Life Cycle [RATIONAL 00]

2.2.2.1 RUP PHASES AND CORE WORKFLOWS

The four phases that make up the life cycle of the system are described below:

1. **Inception Phase:** Establishes the business rationale for the project and achieves initial understanding and agreement of the product definition. In this phase, the customer is involved in delivering requirements and an agreement is reached upon developing a set of use cases and other artifacts that specify the behavior of the desired system.

2. **Elaboration Phase:** Contains a collection of detailed requirements, high-level analysis and design to establish a baseline architecture achieving initial understanding and agreement of

the product's design. By using the techniques of RA, a conceptual domain model and an analysis model will be constructed along with the next iteration of use cases.

3. **Construction Phase:** Creates the initial fully functional product build. The system is built in a series of iterations (mini-projects). Each mini project has analysis, design, coding, testing (unit testing by developer & system testing by testers), and integration.

4. **Transition Phase:** Delivers a product that meets the initial goals. The focus is on the transition of the code from the development organization to the customer.

2.2.2.2 RUP MODELS

The RUP models mapped to the Core Workflows are described below and shown in Figure 2.3.

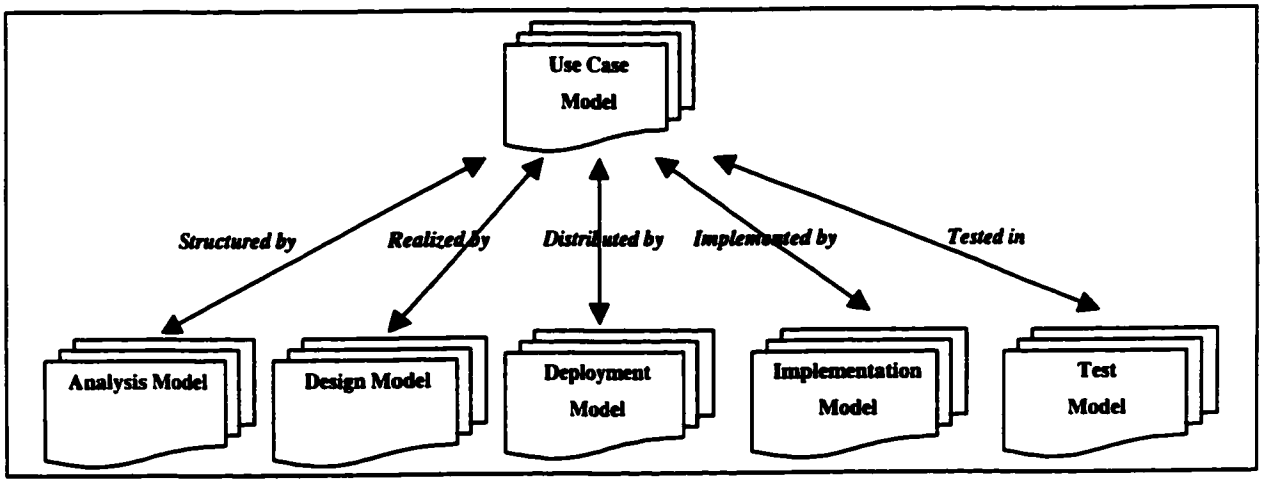


Figure 2.3: RUP Models

1. **Domain and Use-Case model:** The domain model represents an understanding and classification of information within the domain by capturing the functional requirements used

to construct use case and analysis models. It characterizes the problem space vs. the solution space of architectures, class hierarchies and object models.

2. **Analysis model:** The analysis model explores the implications of the requirements for a particular application by refining the use cases in more detail, and making an initial allocation of the behavior of the system to a set of classes that provides the behavior.

3. **Design Model:** The design model is an object model that describes the physical realization of use cases by focusing on how functional and nonfunctional requirements impact the system under consideration. It represents both the information in the domain objects and the behavior in the use cases.

4. **Deployment Model:** The deployment model shows the physical relationships among software and hardware components in the delivered system. It shows how components and objects are routed and move around a distributed system, and where each package is running on the system.

5. **Implementation Model:** The implementation model describes how elements in the design model, such as design classes, are implemented in terms of components such as source code files, executables, and so on.

6. **Test Model:** The test model primarily describes how executable components (such as builds) in the implementation model are tested by integration and system tests. It is a collection of test cases, test procedures, and test components.

The intended reader can consult [JACOBSON 99A], [KRUTCHEN 98], [CANTOR 98], and [ROYCE 98] for more details about RUP and, project management with UML and RUP.

2.3 ICONIX UNIFIED PROCESS

As with RUP, IUP is a refinement of other object-oriented processes and methodologies based on the work of Grady Booch, Ivar Jacobson and Jim Rumbaugh.

Use case modeling, RA and object interactions (sequence diagrams, collaboration diagram) are the main work of Jacobson (OOSE). Booch's main area is detailed design (class diagram, sequence diagrams, collaboration diagrams) and implementation (code) while Rumbaugh's work (OMT) is more concentrated in the area of exploring the problem space (domain model) as shown in [Figure 2.4].

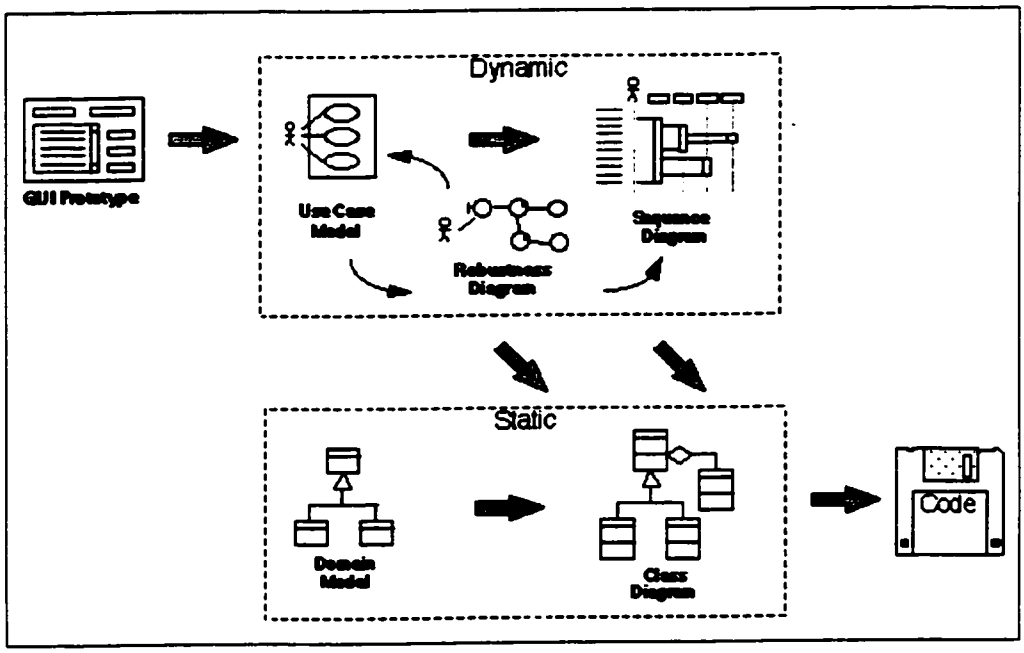


Figure 2.4: IUP Approach [ROSENBERG 00]

2.3.1 IUP APPROACH

Here we present the main properties of the IUP development process. The details will be provided later as needed.

There are three main fundamental features in this approach. It is iterative and incremental, provides a high-level of traceability, and offers streamlined usage of UML:

1. It is **iterative and incremental:** Multiple iteration occurs between developing the domain model and analyzing use cases. The static model (composed of the domain model and class diagram) is refined incrementally during the iterations of the dynamic model (composed of use-case model, RD and sequence diagram) [Figure 2.4].

2. It provides a high-level of **traceability.** At each step, a reference can be traced back into the requirements. In other words, classes can be traced back and forth from one model or diagram into another model. For example, we can trace from use-case model to RD to sequence diagram.

3. It **uses UML** i.e. all artifacts produced by IUP are those defined in UML.

2.3.2 IUP MILESTONES

The IUP approach is achieved in four milestones: Requirements analysis, analysis and preliminary design, detailed design, and implementation/delivery as shown in Figure 2.5. These milestones are very similar to the milestones in RUP.

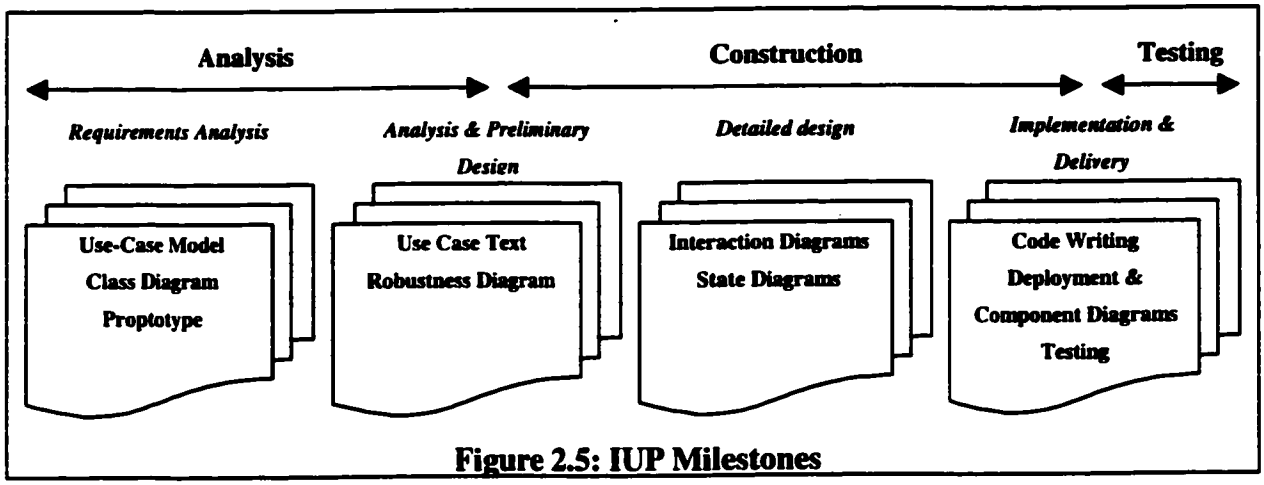


Figure 2.5: IUP Milestones

2.3.2.1 REQUIREMENTS ANALYSIS

The requirements analysis phase consists of creating a high-level class diagram from classes identified in the domain model, and possibly performing rapid prototyping of the proposed system. Also at this stage, a use case diagram is created and organized into package diagrams.

2.3.2.2 ANALYSIS AND PRELIMINARY DESIGN

The analysis and preliminary design phase consists of writing the descriptive text of each use case identified in the requirements analysis phase. Next, RA is performed by identifying classes participating in the stated scenario of each use case. Identified classes are then classified into three stereotypes: *boundary*, *entity* and *control* classes and a RD is constructed for each use case. An update to the domain model and class diagram will be required as classes are discovered in this phase.

2.3.2.3 DETAILED DESIGN

The design phase consists of developing detailed interaction diagrams (sequence and collaboration diagrams). This is done in identifying the messages between different classes for each use case. State diagrams can also be used to show the real-time behavior of each use case. The phase is ended by adding detailed design information such as visibility, values and patterns to the static model.

2.3.2.4 IMPLEMENTATION/DELIVERY

The implementation/delivery design phase starts with code writing. Deployment and component diagrams can be used to facilitate the implementation phase. Application testing, which consists of unit, integration, system, and user-acceptance testing, is also performed.

The intended reader can consult [JACOBSON 92], [ROSENBERG 99], and [ROSENBERG 00] for more details about IUP.

2.4 SUMMARY

In this chapter, we reviewed two object-oriented software development processes and methodologies, RUP and IUP, that are the focus and baseline of this thesis. Both processes have a lot in common. Both are a refinement of earlier object-oriented processes and methodologies developed by the three amigos [BOOCH 94], [RUMBAUGH 91], and [JACOBSON 92]. Both are also iterative and incremental, use-case-driven and based on a strong architectural foundation. The minor difference is that RUP suggests a use case style that is elaborate and encompassing while IUP strongly encourages the use of formal RA and advocates a larger number of smaller use cases [CONALLEN 00]. In this thesis, given the minor differences between the two processes, we favor the IUP process because RA is an integral part of the process. However, RA can apply equally well to RUP.

CHAPTER 3

EXTENSION OF RA TO UNIFIED PROCESSES

*I*n this chapter, we give the context of RA within UML. First, we state necessary preliminary requirements to perform RA followed by a detailed definition of RA with a simple example on how to construct RD. Then we discuss the transition process from the analysis phase into the detailed design phase.

3.1 PRELIMINARIES TO ROBUSTNESS ANALYSIS

There are two prerequisites to performing RA: domain model and use-case model. Both domain model and use-case model are developed in **parallel** as requirements are captured. They are maintained and updated throughout the life cycle of the project as requirements are changed and refined.

3.1.1 DOMAIN ANALYSIS

A domain model captures the most important types of classes in the context of the system. The domain objects represent the "things" that exist or events that transpire in environment in which the system works [RUMBAUGH 91].

Finding classes is a central decision in building an object-oriented software system [BLAHA 98]. Unfortunately, there is not a precise method for finding classes , however, the best sources of classes can be found from the *problem statement* -a set of high-level requirements that describes the main functionality and features of the system-, the set of requirements and, the expert knowledge of the problem space.

Hence, we start analyzing each sentence of the problem statement or requirements document and we extract *candidate classes* by analyzing **verbs** and **nouns** of each sentence. In function-oriented design we would concentrate on the verbs, which correspond to actions and in object-oriented design we underline the nouns, which describe classes [Meyer 97].

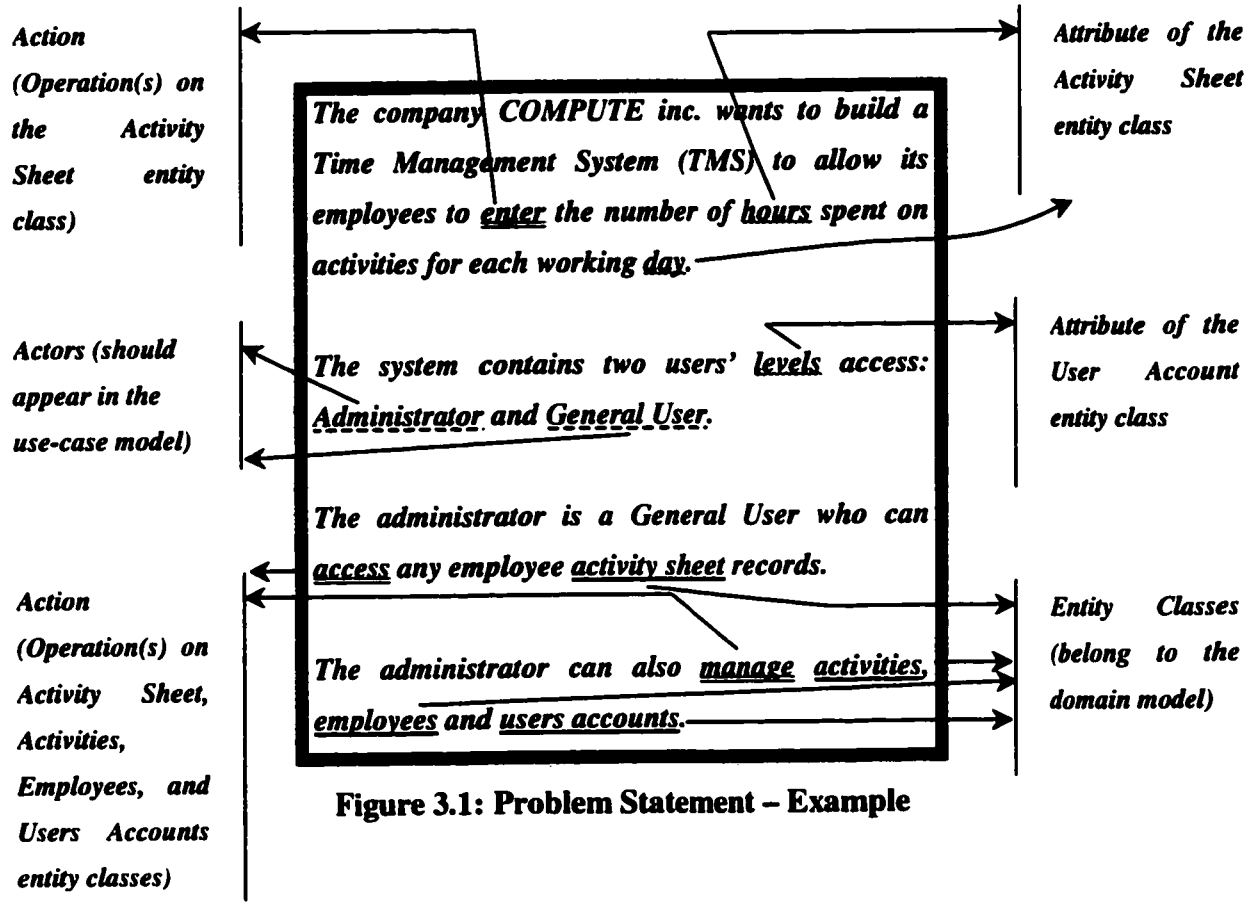


Figure 3.1: Problem Statement – Example

The problem statement example in Figure 3.1 contains a set of requirements. After removing any duplicate, redundant and ambiguous terms, we classify our findings as follows:

1. ***Activity***, ***Employee***, ***Activity Sheet*** and ***User Account*** are entity classes. Therefore, they should be added to the domain model.
2. ***Administrator*** and ***General User*** are actors and will be used in our use-case model in the next section.
3. ***Hours*** and ***day*** are attributes of the ***Activity Sheet*** entity class and will be identified as ***hours*** and ***date*** respectively in the class diagram. In addition, the attribute ***levels*** will be added to as ***level*** to the ***User Account*** entity class.
4. The verbs: ***Enter***, ***Manage*** and ***Access*** are actions and relate to data entry operations on entity classes; therefore, these operations will be given common and technical operations names such as ***create***, ***modify***, ***delete***, ***assign and unassign*** operations.

The Domain model is visually represented with class diagram(s). Class diagrams shows the static structure of classes and their relationships rather than its behavior. They are constructed using classes (classes have attributes and operations) of the domain model, and associations which represent relationships that relate two or more other classes.

In Figure 3.2, the class diagram shows the structure of the system containing the entity classes of the domain model identified earlier from our problem statement:

An Activity Sheet consists of both activities and employees and has date and hours. The "0..n" string between the ***Activity Sheet*** entity class and the ***Activity*** entity class denotes that zero or more activity sheet records are involved in the relationship. The "1..1" string attached to the ***Activity*** entity class of the previous association indicates that only one activity is involved in the relationship. A similar relationship holds between the ***Employee*** entity class and the ***Activity Sheet*** entity class. The relationship between the ***User Account*** entity class and the ***Employee*** entity class indicates that every user of the system should have an employee record, but not

necessarily, every employee has a user account. In this case, the administrator will enter data on behalf of the employees that do not have a login account.

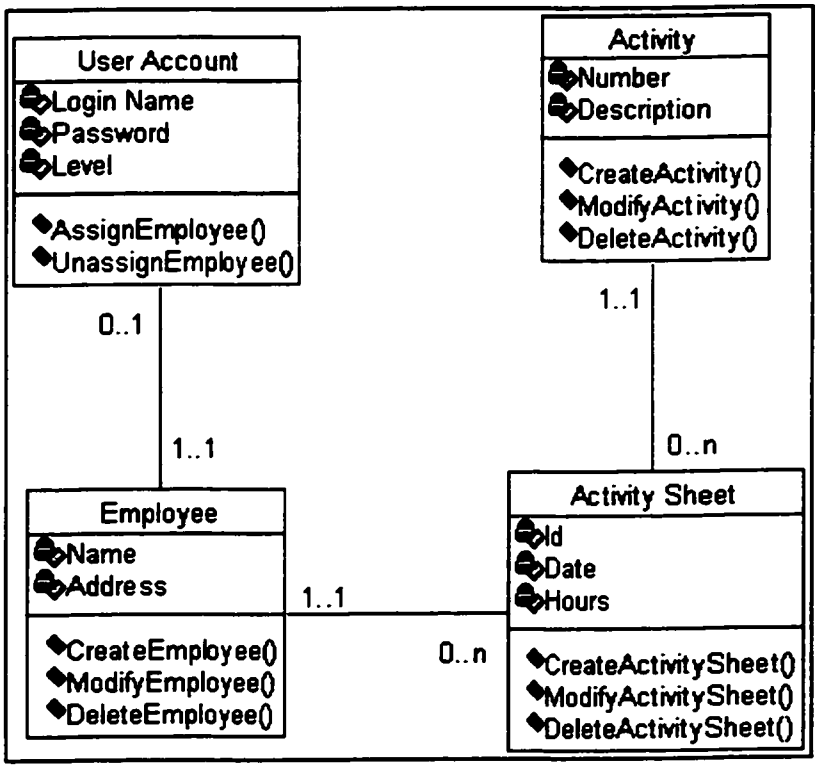


Figure 3.2: Class Diagram –Example

3.1.2 USE-CASE MODEL

The second prerequisite is to construct a use-case model from requirements. Use cases describe the things actors (person or external system) want the system to do [SCHNEIDER 01]. Use cases have to identify actors who perform the use case, and scenarios that describe the purpose or usage of a use case [TEXEL 97]. The use-case model is visually represented with use case diagram(s) as illustrated in Figure 3.3 for our previous example.

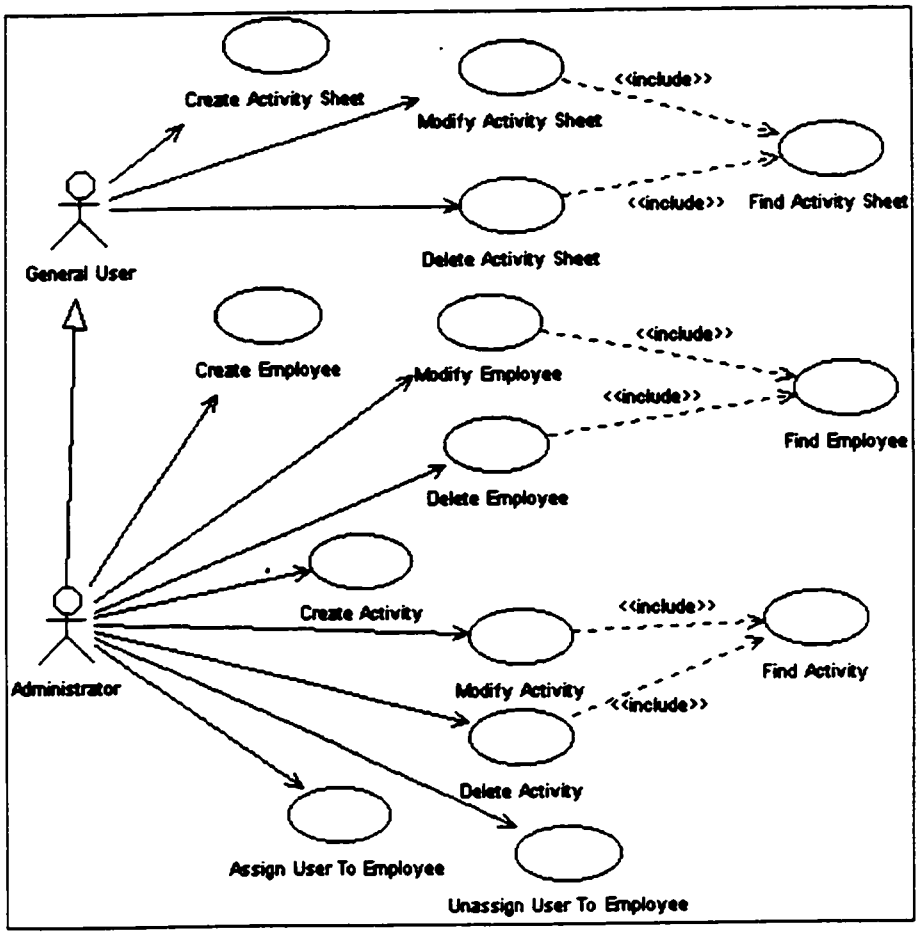


Figure 3.3: Use Case Diagram – Example

The generalization relationship between the ***Administrator*** actor and the ***General User*** actor denotes that the ***Administrator*** actor is a ***General User*** actor with more capabilities to manage employees, activities and users accounts.

The communication of an actor with a use case is shown with a unidirectional association line from the actor symbol to the use case symbol (for instance, from the ***General User*** actor to the ***Create Activity Sheet*** use case). The ***General User*** and the ***Administrator*** actors interact with the system through the ***Create Activity Sheet***, ***Modify Activity Sheet*** and ***Delete Activity Sheet*** use cases. In addition, the ***Administrator*** actor interacts with the system through the ***Create Employee***, ***Modify Employee***, and etc. use cases.

The ***<<Include>>*** relationship between use cases is shown with a *dependency* or *instantiates* arrow from the main use case to the use case being included. For example, The ***<<Include>>*** relationship from the ***Modify Activity*** use case to the ***Find Activity*** use case indicates that the former includes an instance of the latter. In this case, the use case ***Modify Activity*** is no longer complete by itself. It must have the use case ***Find Activity*** included to be completed. On the other hand, the use case ***Find Activity*** does not know when or if it is being included. The same applies to ***Find Employee*** and ***Find Activity Sheet*** use cases.

3.2 ROBUSTNESS ANALYSIS

In this section, we give the definition of RA, and we discuss the three stereotypes classes and the five construction rules required to construct RD.

3.2.1 DEFINITION

RA is a technique that involves scanning and analyzing the requirements in use-case form so as to capture the interactions of objects by constructing a robustness diagram.

RA is a useful technique to better understand and maintain the requirements, and to view the structure of the whole system. RA is a more detailed form of requirements capture. It examines in depth the requirements found in the narrative text of each use case by structuring and refining them into the RD. RA, which occurs during the elaboration phase, serves as a crucial link between the requirements and the design by modeling the requirements into a RD to be the first-cut of the design.

With RA, we are trying to make sure that we have captured correctly user requirements, and we are trying to visualize the overall structure of the system. Our main goal is to capture the most

important decisions about how the classes are going to interact for each use case. For this reason, RA should not be bogged down to too detailed a level [ROSENBERG 99].

Why "***robustness***" analysis? The RA technique improves the ***robustness*** of high-level design which is achieved through the refinement of use cases, validation and verification of user requirements, and detection of errors [Chapter 5]. The result is a ***robust*** object structure and design, i.e. one able to survive unexpected user behaviors.

3.2.2 STARTING ROBUSTNESS ANALYSIS

We start RA by analyzing the narrative text of each use case and identifying required classes to execute the use case. Classes required by each use case are classified into three stereotypes: *boundary*, *entity* and *control* classes. Usually the domain model will provide most of the entity classes required by the system and use cases will provide boundary, control and entity classes (entity not yet identified). Then each use case will be scanned and visually represented with an RD.

Before constructing the RD, we must understand the role of each of the stereotypes and the constraints imposed on them. In the next section, a definition of the three RA stereotypes and their constraints is presented followed by a use case example and its associated RD.

3.2.3 ROBUSTNESS ANALYSIS STEREOTYPES

An RD is constructed using three stereotypes: *boundary*, *control*, and *entity* classes. These stereotypes are standardized in the UML and are used to help developers distinguish concerns among different classes. They conform to a *model-view-controller (MVC)* pattern (discussed in chapter 6) and allow us to partition the system by separating the view (*boundary* classes) from the domain (*entity* classes) from the control (*control* classes) needed by the system. Each stereotype has its own symbol, as shown in Figure 3.4.

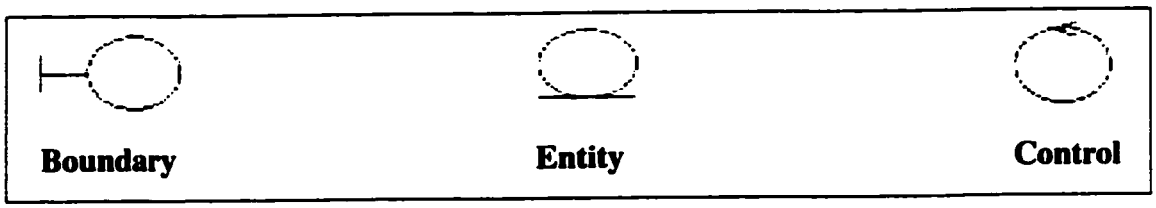


Figure 3.4: Stereotypes Classes

3.2.3.1 BOUNDARY CLASSES

Boundary classes handle the communication between the system surroundings (actors i.e., users and external systems) and the inside of the system. The interaction often involves receiving/displaying information and requests from/to users and external systems. Boundary classes often represent abstractions such as windows, forms, panes, communication interfaces, printer interfaces, sensors, terminals and (possibly non-object-oriented) APIs. [JACOBSON 99A]. Boundary classes should stay at a high and conceptual level and should not address too many details.

3.2.3.2 ENTITY CLASSES

Entity classes handle information and associated behavior that is generally long lived. They are often derived from the domain model. They often map to the database tables and files that hold the information that needs to "outlive" use case execution. They are typically independent of their surroundings; that is, they are not sensitive to how the surroundings communicate with the system.

3.2.3.3 CONTROL CLASSES

Control classes model sequencing behavior specific to one or more use cases. They coordinate the events needed to realize the behavior specified in the use case i.e. they represent the dynamics of the use case i.e. we can think of a control class as executing the use case. They

serve as the connecting tissue between the users and the stored data. Control classes typically are application-dependent classes.

3.2.4 ROBUSTNESS DIAGRAM CONSTRUCTION RULES

In this section, we give the definition of RA, and we discuss the three stereotypes classes and the five construction rules required to construct an RD.

3.2.4.1 CONSTRUCTION RULE 1: DIAGRAM ELEMENTS

The basic elements of an RD are the main use case actor, boundary, control and entity classes as discussed before. Any RD must consist of at least an actor, one boundary class and one control class to be valid. [ROSENBERG 99] recommends the use of two to five control classes per RD. Five to ten control classes are still workable, however, with more than ten controls, we should consider splitting the use case.

The other elements that can be used in the construction of an RD are use cases and notes. Use cases can be shown on an RD to provide completeness of the use case. In addition, we can also show Notes on an RD to provide more information.

3.2.4.2 CONSTRUCTION RULE 2: CONSTRAINTS

The allowable directed associations that connect the elements of an RD are shown in Table 3.1 and discussed below [ROSENBERG 99]:

From / To	Actor	Boundary	Entity	Control
Actor		√		
Boundary	√			√
Entity				√
Control		√	√	√

Table 3.1: Stereotypes Constraints

1. *Actors can talk only to boundary classes.*
2. *Boundary classes can talk only to control classes and actors.*
3. *Entity classes can talk only to control classes.*
4. *Controllers can talk to boundary, control, and entity classes, but not to actors.*

In addition to the above constraints, boundary classes can also be associated to use cases. The association is a one-way navigation from a boundary class to the use case. This allows us to defer the execution of *extended* use cases until later which will eventually simplify the RD and make it easier to read.

3.2.4.3 CONSTRUCTION RULE 3: ASSOCIATIONS

Classes in the RD are connected with an arrow. The arrow is either one-way or two-way navigation. The arrow direction does not represent software messages; rather they indicate

logical associations. An arrow pointing from class A to class B implies that class B is to perform an action triggered by A. The allowable directed associations on a RD can be described as:

1. *An arrow pointing from an actor to a boundary class implies that the actor is interacting and requesting a function from the system.*

2. *An arrow pointing from a boundary class to an actor implies that the boundary is displaying a message to the user, which might require a response from the actor.*

3. *An arrow pointing from a boundary class to a control class implies that the boundary class is executing a code function as a response to an event raised by the actor on the boundary class.*

4. *An arrow pointing from a control class to a boundary class implies that the control class is returning information (or data) to be displayed in the boundary class.*

5. *An arrow pointing from a control class to another control class implies that the former is calling a code function of the latter.*

6. *An arrow pointing from a control class to an entity class implies that the former is writing to the latter.*

7. *An arrow pointing from an entity class to a control class implies that the latter is reading from the former.*

3.2.4.4 CONSTRUCTION RULE 4: AGGREGATIONS

Aggregation, *a stronger form of an association where the relationship is between a whole and its part(s)*, can be placed on an RD to associate two boundary classes of the same type. An aggregation between boundary classes shows the whole (for example a screen object) and the part (for example a push button object) relationship. For simplicity, we do not recommend the extensive use of the aggregation association because it does not contribute much to the clarity or understanding of the diagram.

3.2.4.5 CONSTRUCTION RULE 5: SIMPLIFYING RD

The analysis of each use case leads to the discovery and classification of classes, however, a decision must be made on what classes to include and what classes to exclude. A good decision will eventually simplify the RD and make it easy to read. Hence, we need a construction rule to guide our decision as follows:

1) *All entity classes must be included because entity classes are analysis classes that come from our static domain model.*
2) *A Boundary class can be excluded if (a) and (b) and (c) are true:*
   a) *It is not linked to an entity class (note that the link is indirectly made through control classes by construction rule 2). For example, a List Box that is populated from an entity class must be included whereas a Text Box could be excluded.*
   b) *It is not an essential boundary class, i.e. if we remove the boundary class, the RD does not lessen its meaning and purpose (the use case path can be clearly traceable). For example, the main window of the use case is an essential boundary class, which must be included whereas a Text Box is not an essential boundary class, which could be excluded.*
   c) *A boundary is not linked to a use case.*

3) A control class can be excluded if (a) or (b) or (c) is true:

a) Its associated boundary class has been excluded (condition 2).

b) It can be merged with other controls (in order to hide details), however; the use-case basic path should remain recognizable.

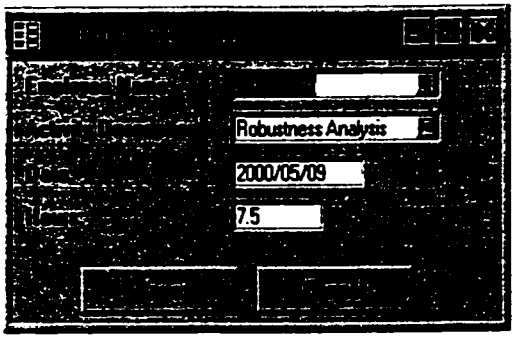
c) It does not link a boundary class to an entity class as described above in 2a, it does not perform complex functions within the use case such as computations, sorting records locally, etc., and it is not part of the use-case basic path, or secondary scenarios.

3.3 CONSTRUCTING ROBUSTNESS DIAGRAMS

In this section, we show how to construct RDs. First, we present a use case from our previous example. Next we identify and classify classes into the three stereotypes as discussed above before constructing the RD.

3.3.1 USE CASE – EXAMPLE

The New Activity Sheet Use Case starts when the Administrator actor opens the Activity Sheet Window from the main application menu as shown below [Figure 3.5].



Single underlined words are all boundary classes

Wave underlined words are all control classes.

Double underlined words are all entity classes

Actors

- Administrator

Precondition

- The administrator has logged into the system.

Flow of Events

Basic Path

1. The use case starts when the administrator selects Activity Sheet Window from the menu of the main application screen.
2. The system displays the Activity Sheet Window.
3. The administrator selects an employee from the employee list.
4. The administrator selects an activity from the activity list.
5. The administrator enters a date.
6. The administrator enters the hours.
7. The administrator clicks on the Save button.
8. The system validates the data then saves the record.
9. The use case ends.

Alternative Paths:

1. The customer can click on the Cancel button any time before selecting the Save button.

The Activity Sheet is not saved, the Activity Sheet Window is closed and the use case ends.

Postcondition

- The activity sheet data is saved to the database.

Secondary Scenarios

- Employee and Activity are not supplied.
- Date and Hours are not supplied.
- The Activity already exists for the specified employee and date.

An error message is displayed to inform the user about missing information or wrong data entered.

Special Requirements

- None.

Figure 3.5: Activity Sheet Use Case

3.3.2 ROBUSTNESS DIAGRAM – EXAMPLE

While reading and analyzing the use case text, we identify and classify all required classes into the three stereotype classes as well as the actors performing the use case [Table 3.2]. Next, we construct the RD for the ***New Activity Sheet*** use case [Figure 3.6] in two steps:

- All identified use case classes will be placed into the RD.
- All RD elements (actors and Classes) will be connected by following the use case basic path, alternative paths, and secondary scenarios, and using the previous constraints rules.

Classes Type	Classes Description
Boundary	1. The ***Main Screen*** boundary class is the main application screen that contains a menu from which the user can access all application functions. This relates to step 1 of the use case basic path. 2. The ***Activity Sheet Window*** boundary class contains all visual classes the user can access to perform the use case. This relates to step 1 of the use case basic path. 3. The ***Employee List*** boundary class is a list box the user can select an employee from. It is populated with data using the entity class ***Employee***. This relates to step 3 of the use case basic path. 4. The ***Activity List*** boundary class is a list box the user can select an employee from. It is populated with data using the entity class ***Activity***. This relates to step 4 of the use case basic path. 5. The ***Date*** boundary class is a text box the user will use to enter the activity's date (step 5 of the use case basic path).

Classes Type	Classes Description
	6. The ***Hours*** boundary class is a text box the user will use to enter the activity's hours (step 6 of the use case basic path). 7. The ***Save Button*** boundary class is a push button the user will use to save the data to the database. This relates to step 7 of the use case basic path. 8. The ***Cancel Button*** boundary class is a push button the user will use to close the screen without saving data to the database. This relates to step 1 of the use case alternative paths.
Entity	1. The ***Employee*** entity class will be taken from our domain model identified previously. It provides data to the ***Employee List*** boundary class. This relates to step 3 of the use-case basic path. 2. The ***Activity*** entity class will be taken from our domain model identified previously. It provides data to the ***Activity List*** boundary class. This relates to step 4 of the use-case basic path. 3. The ***Activity Sheet Table*** entity class will be taken from our domain model identified previously. It serves to store the employees' activities. This relates to point 1 of the use case Postcondition.
Control	1. In step 1 of the use-case basic path, we need a control to open the ***Activity Sheet Window*** boundary class when the Administrator selects it from the ***main screen*** application menu. Therefore, we create the ***Menu Handler*** control class. 2. In step 3 of the use-case basic path, we need a control to link the ***Employee*** entity class to the ***Employee List*** boundary class. Therefore, we create the ***Pick Employee*** control class.

Classes Type	Classes Description
	3. In step 4 of the use-case basic path, we need a control to link the ***Activity*** entity class to the ***Activity List*** boundary class. Therefore, we create the ***Pick Activity*** control class. 4. The ***Validate Data*** control class serves to validate the data entered (step 8 of the use case basic path). It covers the first four items of the secondary scenarios. 5. The ***Check Duplicates*** control class serves to extend the ***Validate Data*** control class (step 8 of the use case basic path). It covers the last item of the secondary scenarios. 6. The ***Save Data*** control class saves the data to the ***Activity Sheet Table*** entity class (step 8 of the use case basic path). 7. The ***Display Error*** control class returns an error message to the user if any of the secondary scenarios return is exercised (secondary scenario). 8. The ***Close Screen*** control class closes the activity sheet window (alternative paths).

Table 3.2: Stereotype Classes for New Activity Sheet Use Case

How much detail should an RD contain? If it is too detailed, the RD becomes less readable, difficult to maintain, and time consuming. The RD we have constructed in Figure 3.6 contains many details for a use case with few boundaries and entities.

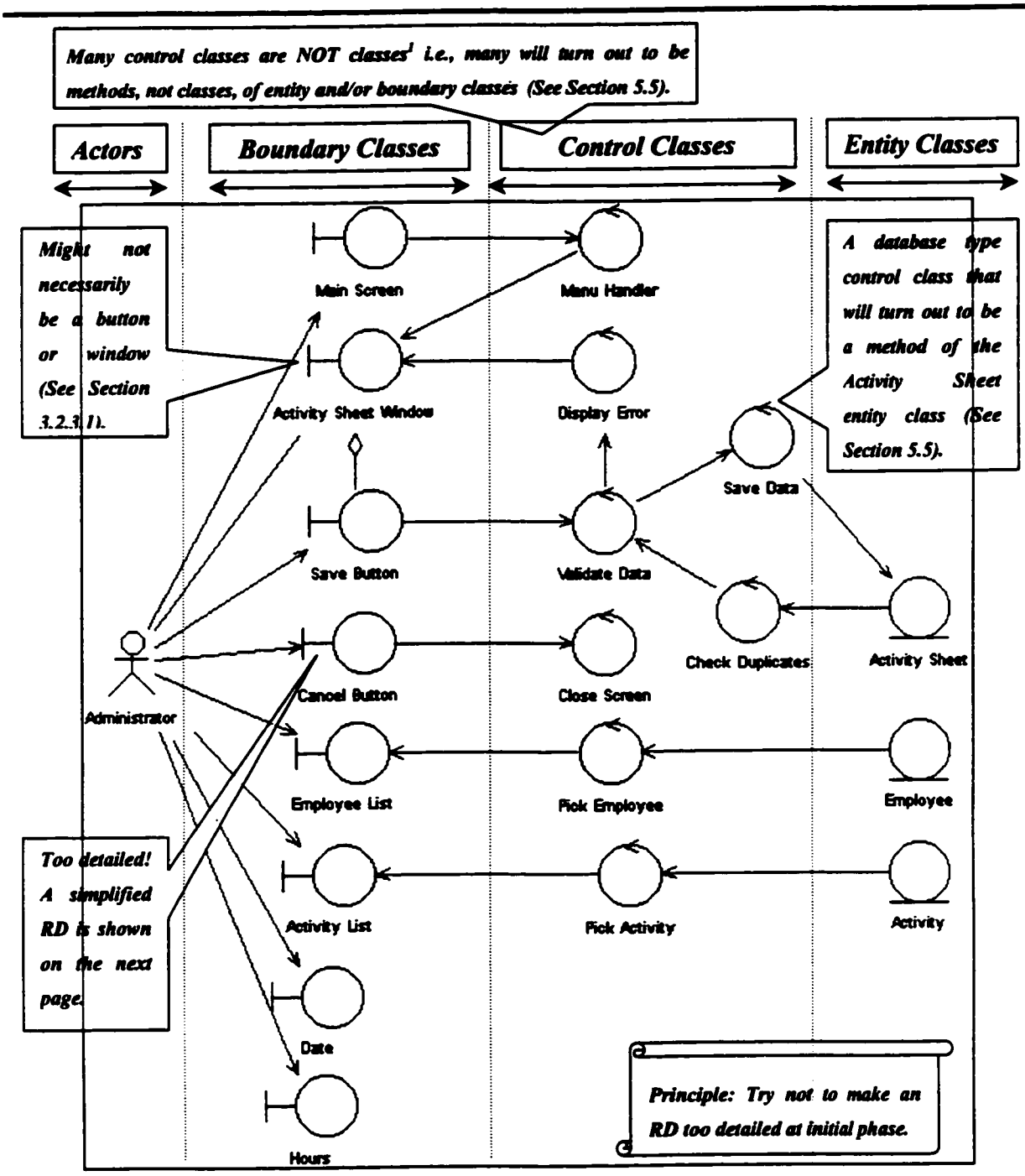


Figure 3.6: Robustness Diagram – Example

[1] We would have liked to rename control classes to "control concerns", however, we have decided to refer to it as classes in order to remain consistent with the literature [JACOBSON 92].

A simplified RD can be achieved using construction rule #5 as shown in Figure 3.7.

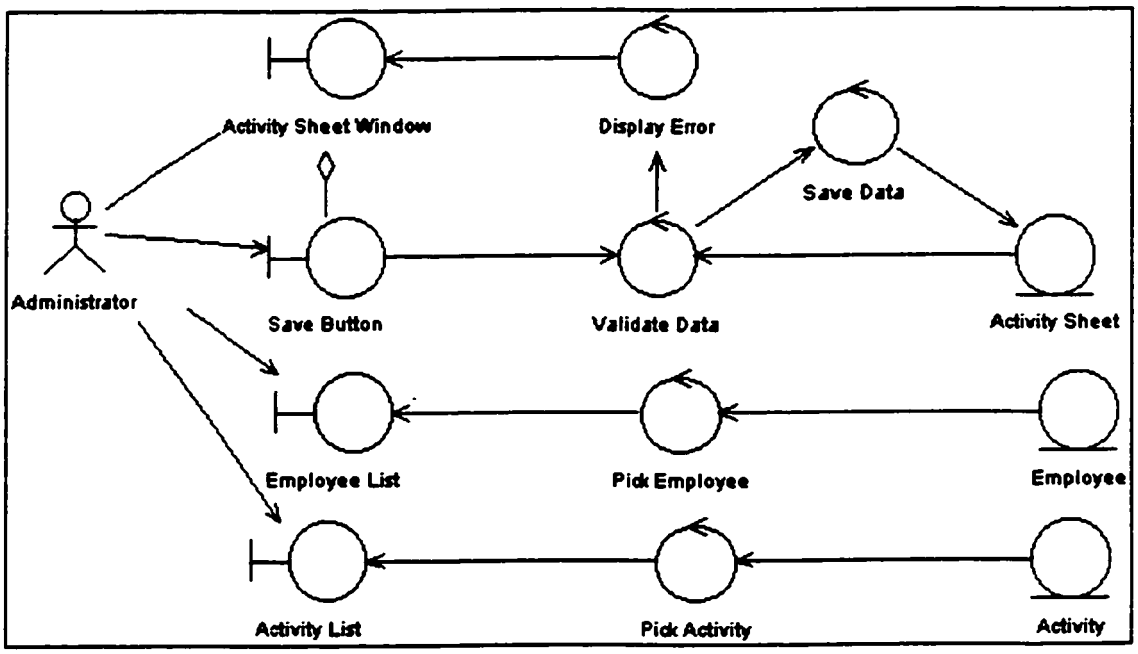


Figure 3.7: Robustness Diagram Simplified – Example

All entity classes must be included (using construction rule #5, 1st condition). Therefore, we will include ***Activity Sheet***, ***Employee***, and ***Activity*** entity classes.

For boundary classes, we will include ***Employee List***, and ***Activity List*** boundary classes (using construction rule #5, 2nd condition, Part a is false), and ***Save Button***, ***Activity Sheet Window*** boundary classes (using construction rule #5, 2nd condition, Part b is false). On the other hand, we will exclude ***Main Screen***, ***Date***, ***Time***, ***Cancel*** boundary classes (using construction rule #5, 2nd condition, Part a, b, and c are true).

For control classes, we will exclude ***Menu Handler***, and ***Close Screen*** control classes (using construction rule #5, 3rd condition, Part a is true). In addition, we will exclude ***Check Duplicate*** control class by merging it with ***Validate Data*** control class (construction rule #5, 3rd condition, Part b is true). On the other hand, we will include, ***Save Data***, ***Pick Employee***, ***Pick Activity***,

Validate Data, and ***Display Error*** control classes (using construction rule #5, 3rd condition, Part a, b, and c are false).

In total, we have excluded seven classes out of nineteen. Hence, the simplified RD now contains twelve classes.

Note that the RD in Figure 3.6 and Figure 3.7 fits well the N-tier architecture. A detailed discussion of this subject is described in Chapter 6.

3.4 ROBUSTNESS ANALYSIS TRANSITION TO DESIGN

The successor to the RA activity is the design model represented with Interaction Diagrams. Interaction diagrams are models that describe how groups of objects collaborate in some behavior [FOWLER 97A]. There are two kinds of interaction diagrams: sequence diagrams and collaboration diagrams.

3.4.1 SEQUENCE DIAGRAM

Sequence diagrams describe interactions among classes. These interactions are modelled as messages. After constructing an RD for a particular use case, sequence diagram is much easier to draw (Section 3.3.3). In fact, boundary and entity classes of the RD will be placed on the Sequence Diagram while control classes convert to messages (***Check Duplicates***), remain control classes (***Menu Handler***), or are not considered (***Pick Employee***) as shown in Figure 3.8.

Note that there is a small difference between the RD and the related sequence diagram that should be clarified:

In the RD [Figure 3.7], we created ***Pick Employee*** and ***Pick Activity*** control classes to link the ***Employee List*** boundary class to the ***Employee*** entity class, and the ***Activity List*** boundary class to the ***Activity*** entity class respectively.

In the sequence diagram [Figure 3.8], when the Administrator opens the ***Activity Sheet Window*** from the ***Main Screen*** application menu, the ***Get Employee List*** and ***Get Activity List*** messages (Operations/Methods) loads data from the ***Employee*** and ***Activity*** entity classes into their respective boundaries (***Employee List*** and ***Activity List***). In addition, the ***Employee List*** and ***Activity List*** boundary classes do not exchange messages with ***Employee*** and ***Activity*** entity classes respectively.

This difference illustrates the difference between analysis (*what*) and design (*how*). In RD, we are looking at the system from a user viewpoint and we are linking boundary classes to entity classes logically using control classes. For instance, we have created the ***Pick Employee*** control class to logically link the ***Employee List*** boundary class to the ***Employee*** entity class, but we did not say *how* this is done. Consequently, we have shown *what* must *be* done (RD) not *how* must be done (sequence diagram).

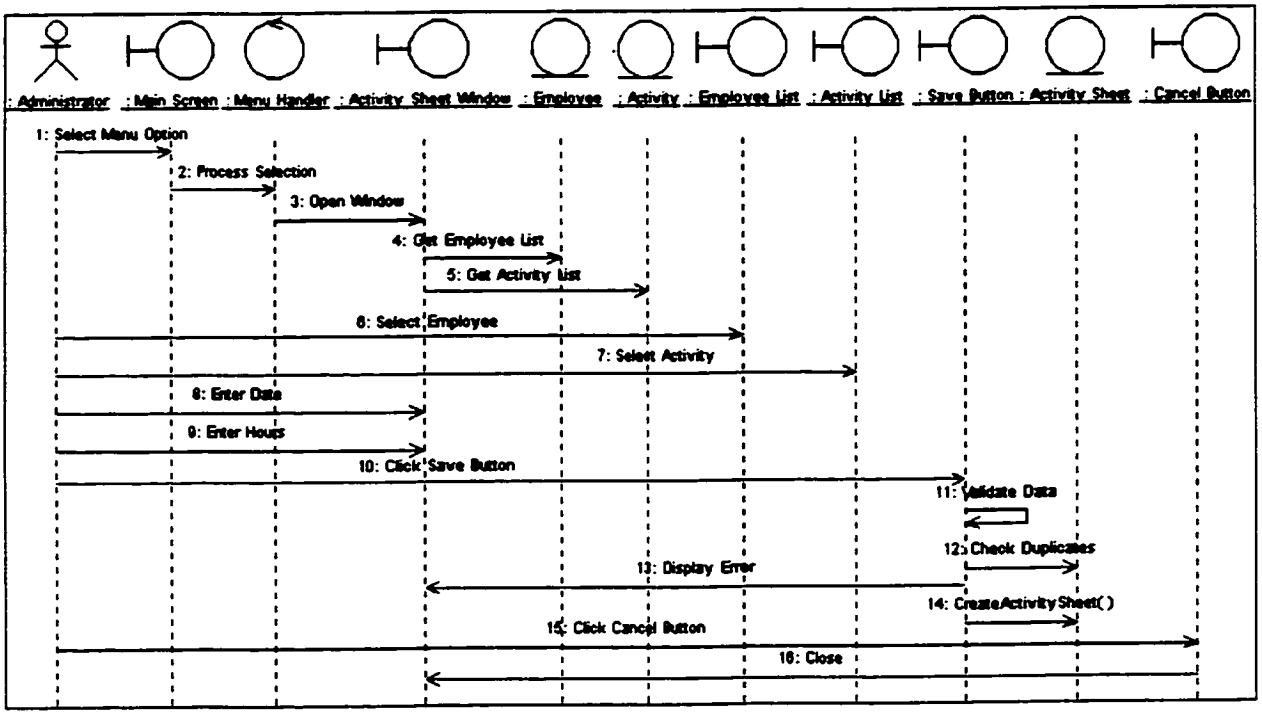


Figure 3.8: Sequence Diagram – Example

3.4.2 COLLABORATION DIAGRAM

Collaboration diagrams describe interactions among classes and associations. These interactions are modeled as exchanges of messages between classes. Collaboration diagrams are essentially the same as sequence diagrams; however, each type of diagram expresses the information with a different view. While sequence diagrams focus on the time dimension, collaboration diagrams focus on object instances. If the Rational Rose visual modeling tool is used, the conversion of sequence and collaboration diagrams is automated as shown in Figure 3.9 [QUATRANI 98]. In addition, the tool automatically translates the changes between the two diagrams.

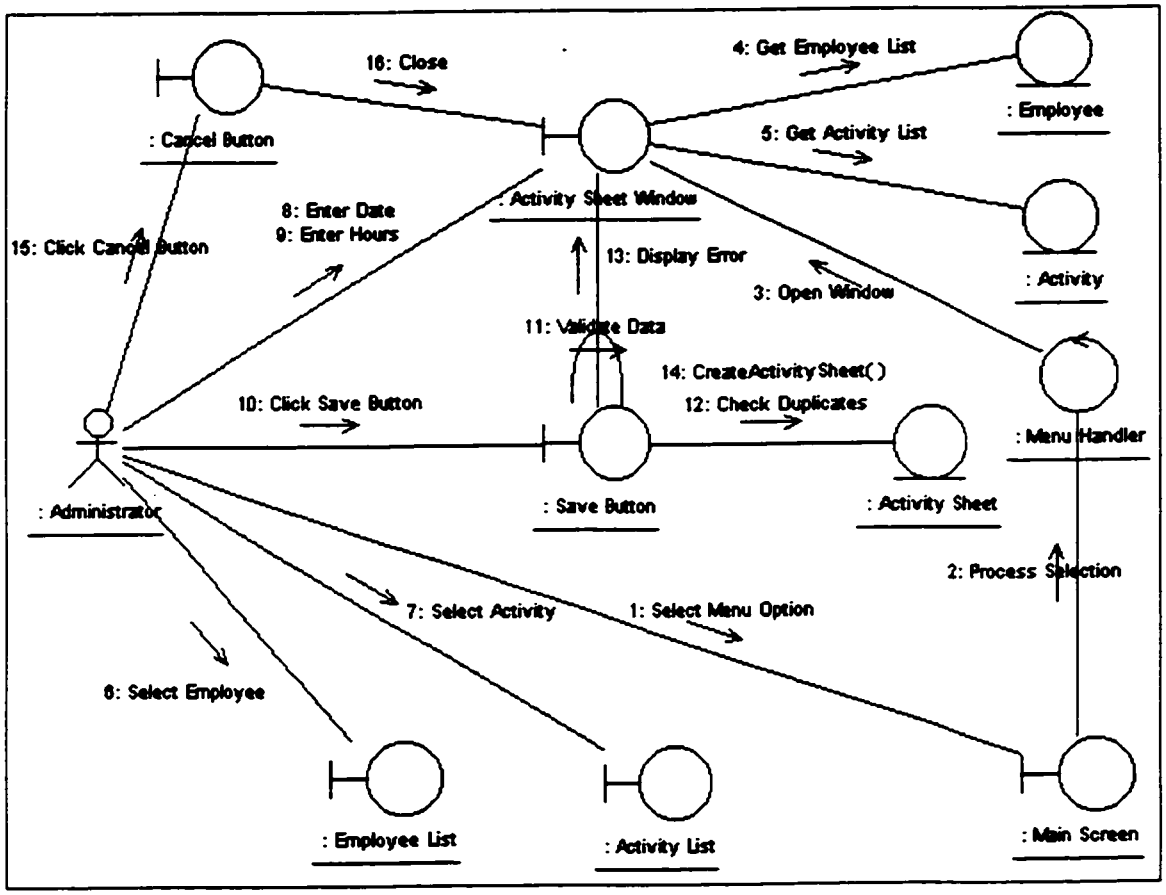


Figure 3.9: Collaboration Diagram – Example

3.5 COMPLETE STEPS TO PERFORMING RA

In this section, we summarize and describe the complete steps for performing RA and constructing RDs for each use case. Not all steps have been exercised in this chapter; however, we will see how they are applied later in Chapter 5.

For each use case,

1. *Read carefully the whole use case text to accustom yourself with its content and purpose, and to identify the use case actor.*

2. *Identify and extract candidate classes required to build the RD by analyzing verbs and nouns of each sentence.*
3. *Classify these candidate classes into the three stereotypes: boundary, control and entity classes.*
4. *If a new entity class is discovered, then add this entity to the domain model.*
5. *Create a new RD and add the identified actor and classes to it.*
6. *Go through the use-case basic path and create the links between the use case actor and boundary classes and among the three stereotypes classes.*
7. *Repeat step 6 for each use-case alternative path. If an alternative path refers to another use case, add this use case to the RD and link it to the related boundary class.*
8. *Go though each of the secondary scenario and make sure that it has been considered and covered.*
9. *Perform a quality analysis check of the RD and make sure that the use case is realizable and correct and that all RD elements are used and connected.*

End use case.

The completion criteria for the RA activity can be summarized as follows:

1. *For each use case, an RD has been constructed.*
2. *Quality analysis check has been performed on each RD.*
3. *The use case text has been refined and corrected (use-case basic path, use-case alternative path, and use-case secondary scenarios) if errors were found in criteria 2.*
4. *Entity classes discovered during the RA process were added to the domain model and class diagrams if errors were found in criteria 2.*

3.6 SUMMARY

In this chapter, we defined RA and we showed the required steps to perform RA, and the necessary construction rules to construct RD. In the next chapter, we will present a case study and show the benefits of performing RA during the analysis phase.

CHAPTER 4

INTRODUCTION TO CASE STUDY OF AN ON-LINE STOCK BROKER

*I*n this chapter, we present an Internet brokerage case study based on the Schneider's approach to requirements capture at high-level design [SCHNEIDER 01], the problem statement and stock trading class diagrams [RICHTER 99], and the TD Waterhouse WebBroker application [TDWATERHOUSE]. The main purpose of this case study is to show how to effectively perform RA to uncover errors in the requirements. First we define e-commerce, then we introduce and motivate our case study followed by the requirements analysis phase with the preliminary requirements prior to performing RA: case study problem statement, domain object modeling (class diagrams) and use case modeling (use case diagrams).

4.1 OVERVIEW OF ELECTRONIC COMMERCE

In this section, we give an overview of Electronic Commerce and Internet Commerce and we highlight its importance as more users are using the Internet and more companies are building web applications.

4.1.1 E-COMMERCE AND INTERNET COMMERCE DEFINITIONS

Electronic Commerce involves the combination of networked applications and commercial transaction handling. It will provide the foundation of the emerging information society as well as the information technology infrastructure to support future business processes and the exchange of goods and services. Internet Commerce is one type of Electronic Commerce.

By *Internet Commerce*, we mean the use of the global Internet for purchase and sale of goods and services, including service and support after the sale [TREESE 98]. Nowadays Internet Commerce is expanding to include more of Electronic Commerce which includes the use of computing, telephone and communication technologies in financial businesses, on-line airline reservation systems, order processing, inventory management, stock trading, and so on.

The importance of the Internet has increased as more users are using the web [Figure 4.1]. As of November 2000, the total number of users using the Internet reached 407.1 million [NUA 00].

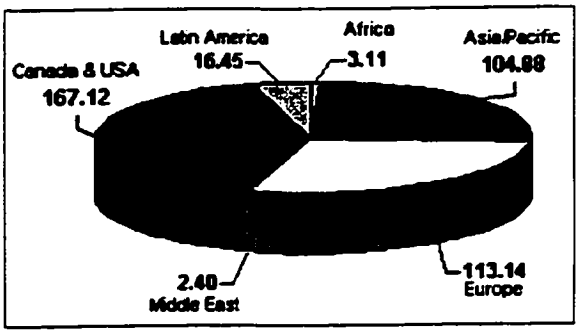


Figure 4.1: Internet Users

4.1.2 INTERNET COMMERCE APPLICATIONS

There are big differences between building a web site and building a web application [CONALLEN 00]. Web sites are relatively static and easy to build as they involve mainly

publishing information. Web applications are more dynamic and difficult to built as they deal with changing the state of the business and are rich in content.

In general, web applications do not require different analysis than conventional applications. In fact, traditional analysis techniques and models apply equally well to many web applications. However, web applications raise new issues such as browser rendering, and higher robustness requirements because they are potentially accessible to anyone on the Internet. These new issues need to be addressed in the development of web applications.

4.2 ON-LINE STOCK BROKERAGE

On-line brokerage has actually been around for more than 10 years in one form or another with companies providing their customers proprietary software that was utilized through a modem connection. Customers dialed in directly into private networks, and were able to place trades through these direct connections. However, these systems were very elementary compared to the much more dynamic, robust systems that are offered today for on-line trading. It was not until the development and widespread acceptance of the Internet that on-line trading became more widely used.

Nowadays, the fever of online investing is taking a toll on financial institutions around the world as over 250 banks and brokerages have already invested in establishing Internet-based brokerage units [ASPIN 99]. This expansion is based upon the promise of speed, convenience, and cost-effective access to capital markets. While the impact of the on-line market is clearly being felt by full service and discount investment firms alike. Offensive and defensive strategies are being quickly mounted with the hope of capturing share and building presence in this growing market.

4.3 INTERNET STOCK BROKER CASE STUDY

In this section, we introduce and give motives to study the on-line stock broker system. We present a high-level system requirements of the system, we analyze risks and market factors that are associated with the development of the system and we give assumptions to building the system.

4.3.1 MOTIVATION AND INTRODUCTION

We have selected to design an on-line stock broker as our case study to validate the new RA methodology because:

- It is a new and important aspect of Internet Commerce and of great interest to the financial sector.

- It is representative of a real life system where things can go wrong and where robustness is an important design constraint.

We now proceed to describe the scope and context of our case study system.

North American Broker inc. (NAB) is a brokerage company that offers its customers various types of investments. Its customers can place orders using an automated telephone system or by calling a customer representative. In order to compete with other on-line brokerage firms and due to the appropriateness of the emerging Internet technology, NAB has decided to offer its customers a new on-line service through the web. The new system will be known as "**NetBroker**".

4.3.2 CASE STUDY SYSTEM REQUIREMENTS

As shown in Figure 4.2, the NetBroker trading system will be a web application accessible to all NetBroker's customers through the Internet. The system will be linked to two other existing NAB systems: the Accounting System and the Security Exchange System.

The Accounting System maintains customers profile and accounts, calculates interest, post activities, transfer money from banks through a banking interface, etc.

The Security Exchange System maintains security information such as converting, adding, removing securities, performing stock split, etc. The system also routes orders and retrieves real time quotes through a stock exchange interface such as the New York Stock Exchange (NYSE) or the Toronto Stock Exchange (TSE), etc.

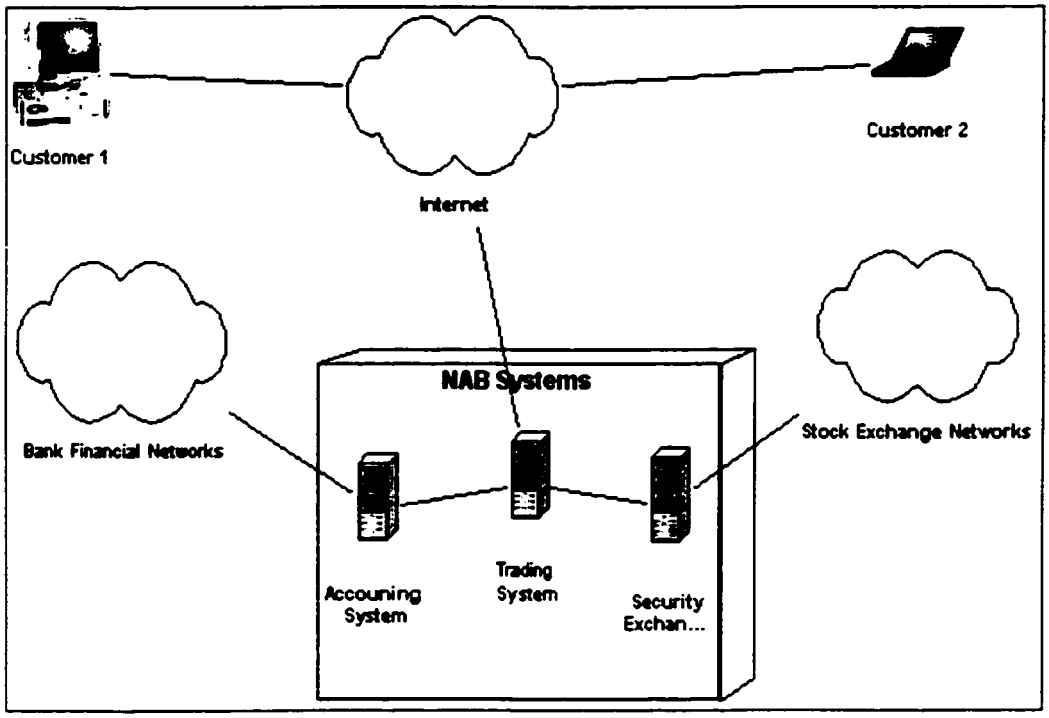


Figure 4.2: On Line Stock Broker: Physical View

The separation of these three systems is mainly because we want the trading system to be concerned only with order processing with interfaces to the two other existing systems. This architecture optimizes system performance and minimizes security risks.

4.3.3 CASE STUDY RISK AND MARKET FACTORS ANALYSIS

There is always a risk associated with the development of a new system. Before starting any development activity including RA, a risk and market analysis must be done to cover all risk factors that influence the success of the project and to identify all market factors that are the rationale to developing a web application. These analyses form the business case for a new Internet application development project.

For the NetBroker web application, the major *risk factors* that contribute to the success of the project are as follows:

- **Availability:** The NetBroker web site should be up and running 24 hours, seven days a week making it available for clients to check their accounts, orders, etc. Any glitch in the software could cause the company to lose money and clients or result in damage to its reputation.

- **Useability:** The NetBroker web site should be easy to use. Web sites that are hard to use frustrate customers, forfeit revenue, and erode brands [MANNING 98], [BECKER 01].

- **Timeliness:** In a *surge downward* market (the same applies for a surge upward market), the company should execute orders in less than a minute. Surge downward means that sellers largely outweigh buyers resulting in a sharp decline in the market. This time frame includes the time the order arrives at the system to the time it is executed and routed to the appropriate stock exchange. Failure to due so would result in customers selling their shares for less, buying their shares for more, or simply missing the market surge entirely.

- **Robustness:** The system should be able to recover (or attempt to recover) from any system failure. In case of an order placement or execution, when a system failure occurs, the order must be deemed cancelled and the user must be informed and advised to contact a customer representative. While a zero-fault system seems to be unrealistic, the maximum acceptable down time should be minimized.

- Other issues that must be addressed require an underlying strategic plan to handle database or application crashes, server capacity and recoverability, and maximum system utilization.

The major incentive *market factors* with respect to developing the web application are as follows:

- **Market reach:** The web offers the fastest, easiest and, broadest access to market capital for customers to trade securities on-line on main stock exchanges. Moreover, the web allows access to the latest market news for customers to conduct financial research and subsequently to act appropriately.

- **Market economics:** Web brokers are popular and competitive in the market. The automation of these trading services reduces the company's cost of doing business (less staff are required) and hence reduces the transaction fee. This attracts more customers.

- **Customer convenience:** The web allows customers to check their accounts and orders status, and place their orders anytime (even after market close) and anywhere (outside North America).

4.3.4 ASSUMPTIONS

Before proceeding with the analysis of the NetBroker system, we make the following assumptions for our case study:

- In order to access the system, the customer should have both an account with NAB and have access to the web. To achieve this, the customer has to submit an application to the accounting department. The management of the customer profile, accounts and web access are done through an external application (accounting system), and hence are not considered in this case study.

- The transfer of securities from other brokerage firms or financial institutions into the customer account is done through the external application (security exchange system), and hence is not considered in this case study.

- NAB offers various types of investments; however, in this case study we will only address buying and selling of equities for customers having cash or margin accounts. Specifically, options and mutual funds are not considered here.

- The target architecture for our case study involves the use of scripted pages such as Java Server Pages (JSP) and Active Server Pages (ASP). We made this assumption because it is typical and widely used in the development of web applications.

4.4 PRELIMINARIES TO ROBUSTNESS ANALYSIS

In this section, we analyze user requirements before performing RA. It includes analysis and discovery of candidate classes extracted from the case study problem statement, and construction of the class diagram, and use case diagram.

4.4.1 DOMAIN ANALYSIS

The first source of information to discover candidate classes is the case study problem statement [RICHTER 99]:

Statement of Problem:

We are to develop an on-line trading system for the NAB Company to allow its customers to buy and sell securities from their web browser.

The system is open i.e. all orders are matched outside the system. The orders received by the company are routed to the appropriate stock exchange where the orders are matched and executed. The stock exchange interface will notify the system whenever the status of an order has changed. At any time, an order can have only one out of six possible states: waiting, opened, filled, partially filled, cancelled or expired.

Each order consists of a stock name, number of shares, a price, and a "good till" date and should be associated with an account number.

At any time, customers can check the status of their orders. They can also cancel, or modify an order that has not yet been executed.

Customers can get real time quotes on a specific stock name. The retrieved quote should reflect the most current market price.

Each security has an allowable margin percentage varying from 0 to 70% depending on the security type. NAB reserves the right to modify the percentage without any notice.

After the market close, the Account Manager validates all customer accounts for any margin call. If a margin call is identified, the system will place a sell order on the stock that has either the highest margin call or the highest market value[1].

[1] SOP is not an input to RA, so the ambiguity in the last sentence (***either ... or***) will not be addressed by RA. It should properly be addressed in use cases.

The main domain analysis classes (including their attributes and operations) that have been discovered so far come from the above problem statement and the use case text (in the next section) as shown in Figure 4.3. For example, a ***customer*** can have several ***accounts***. An account is made up of several ***activities,*** associated with multiple ***orders,*** and has holdings (***portfolio***). A ***stock*** has ***fundamentals*** and represents a ***company.*** A company can have more than one stock symbol traded on one or more capital ***markets*** and each market contains several stocks.

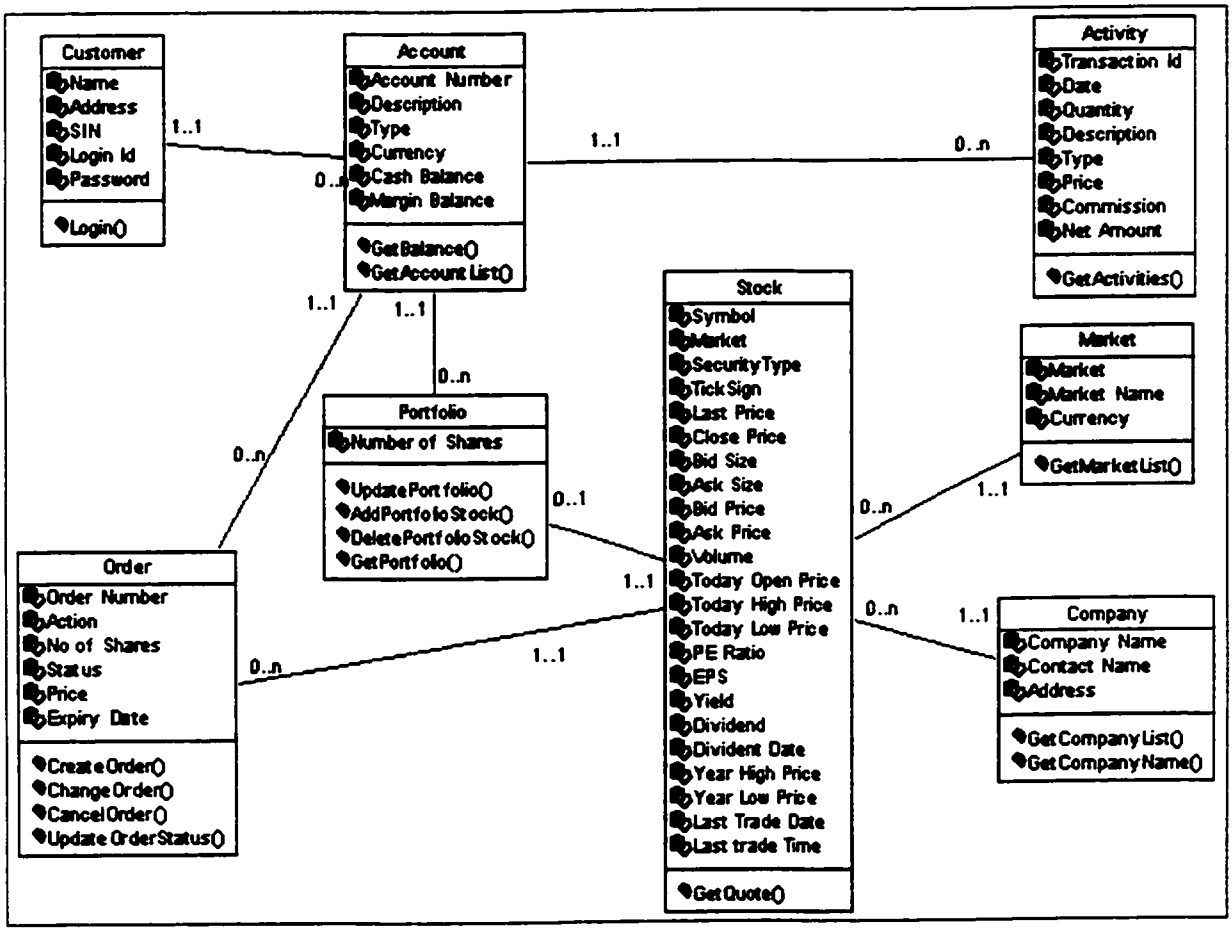


Figure 4.3: On-line Trading Class Diagram (Incomplete)

Note that this domain analysis is done in parallel with derivation of the use-case model (discussed in the next section).

We have high confidence in the domain analysis, but this will be verified during RA.

4.4.2 DERIVING THE HIGH-LEVEL USE CASE DIAGRAM

Five actors were identified in this case study: ***customer***, ***customer representative***, ***security exchange system***, ***accounting system***, and ***trading system***.

The ***customer*** actor is a user of the system.

The ***customer representative*** actor, a generalization of the ***customer*** actor, has access to the same system functions as the user plus other functions that have yet to be identified.

The ***security exchange system*** actor is an external application whose interface allows the access to stock exchanges such as processing of orders and getting real time quotes.

The ***accounting system*** actor is an external application whose interface allows access to customer accounts.

The ***trading system*** is the back office system that supports the NetBroker on-line trading system and processes batches such as initiating and monitoring customers' margin calls and processing orders back and forth into the system.

The high-level use case diagram in Figure 4.4 shows the role of each actor, and shows how each actor interacts with the system. After the customer logs (***Login***) into the system, he can retrieve a real-time quote (***Get Quote***) before proceeding to place a buy (***Buy Order***) or sell (***Sell Order***) order on securities. At any time, the customer can check the status of his **orders** (***Get Order Status***). When checking his order statuses, the customer can select to view (***View Order Details***) order details, cancel (***Cancel Order***) an existing order, or modify (***Modify Order***) an opened

order. Moreover, the customer can check his account balance (***Get Account Balance***), account activities (***Get Account Activity***), and view his holdings (***View Portfolio***). In addition, the customer can select to change his password (***Change Password***).

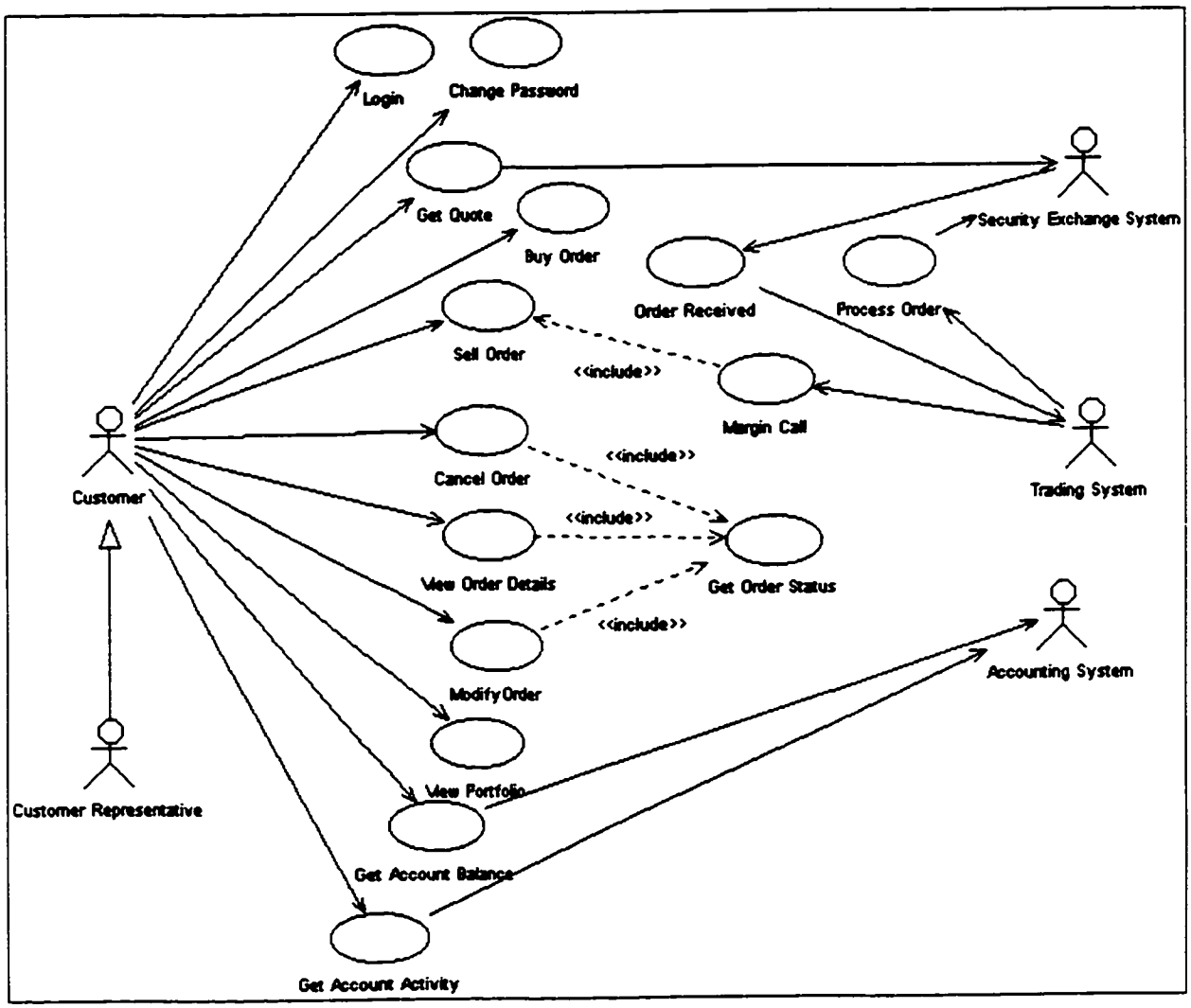


Figure 4.4: On Line Stock Broker: Use Case Diagram

The Accounting System enables the customer to get his account balance (***Get Account Balance***) and account activities (***Get Account Activity***).

The Security Exchange System provides the customer with real-time quotes (***Get Quote***), executes orders (***Process Order***) and notifies (***Order Received***) the Trading System whenever an order has been filled or its status has changed.

On the other hand, the Trading System processes batches such as processing (***Process Order***) and receiving (***Order Received***) orders, and monitoring customers' accounts margin (***Margin Call***).

4.4.3 DERIVING WEB SITE NAVIGATION MAP

At this point, we have a better understanding of the system requirements. An overview of the primary web site navigation map of the system is shown in Figure 4.5. Except for the Login and Main NetBroker web pages, the other pages can be either web pages or frames of the *Main NetBroker* web page. Either case can be equally well modeled with RA because web pages and frames are considered boundaries of the system. The same equally applies for Java applets, ActiveX controls, etc.

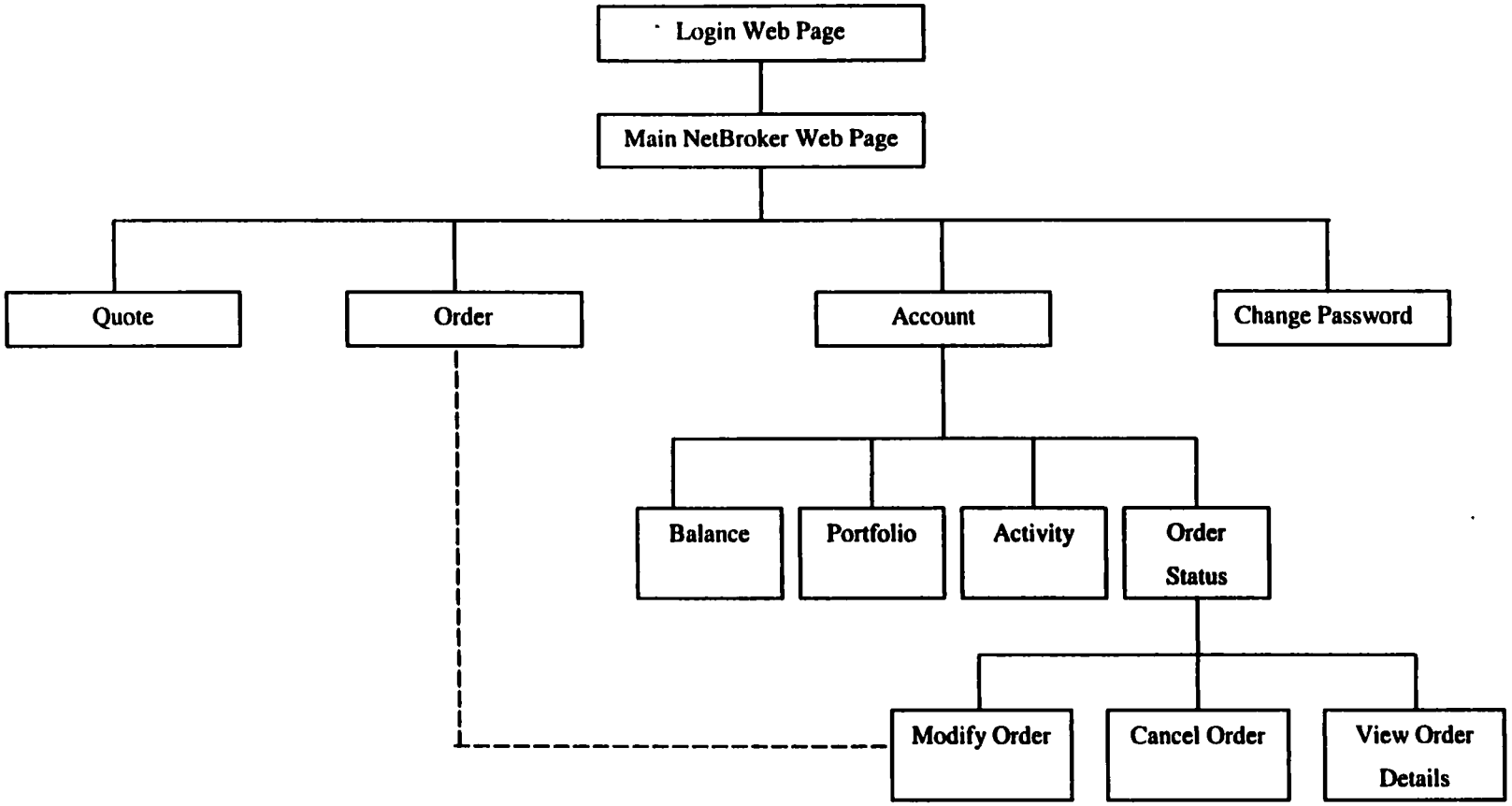


Figure 4.5: Web Site Navigation Map

Note: The Order Status web page allows the user to check the status of his orders whereas the Order web page allows the user to place a buy or sell order. Both Order and Modify Order web pages have the same functionality, however, the Modify Order web page is preloaded with data of an existing order.

4.5 SUMMARY

In this chapter, we studied the preliminary steps in the requirements analysis phase of the building of an Internet brokerage application. First, we reviewed the importance of Internet Commerce and hence its relevance for a case study. Next, we presented the domain analysis including case study problem statement, and application class diagrams. In parallel, we constructed the high-level use-case model. In the next chapter, we will continue and complete the analysis phase by performing RA and constructing RD for selected use cases. Finally, we will show how RA validates and verifies user requirements.

CHAPTER 5

ROBUSTNESS ANALYSIS OF THE ON-LINE STOCK BROKER

*I*n this chapter, we conduct RA and we show how it verifies and validates user requirements for three NAB use case examples: ***Get Quote***, ***Buy Order***, and ***Fill Buy Order Received***. We will walk through each example and demonstrate how to perform the RA and how to construct the RD and most importantly how to uncover errors early in the analysis and design process. Then for each use case we will construct the corresponding interaction diagrams. First, we consider the types of requirements errors and design errors which may be caught by RA, then we will apply RA in turn to each of the three NAB use case examples and detect errors of three distinct types. In addition, we will demonstrate how RA fits well with N-tier architecture.

5.1 ROBUSTNESS ANALYSIS ERROR TARGETS

RA is suggested as an initial validation of use cases [JACOBSON 92], [ROSENBERG 99]. According to [BINDER 99], the analysis required to produce extended use cases is very effective at finding omissions, inconsistencies, and other requirements errors. The main advantage of using RA is its ability to find such errors early in the process when they are less costly to fix. The types of errors that RA uncovers fall into four major categories:

1. **Infeasible Specifications:** RA detects specifications that can not be realistically realized given the type of application we are building, the available technology and system infrastructure, our experience in the domain, or perhaps project time frame or available resources.

   One major impact of this error type can be on the application performance contributed to mainly by sophisticated design or data manipulation requirements. In order to detect this type of error, an analysis of every boundary and entity class of RD is made. The analysis of boundary classes detects complexity in the GUI while the analysis of entity classes allows us to detect the amount of data transferred (approximate number of records is usually sufficient) when reading from or writing to an entity. The amount of data that the application can handle at one time, and the complexity of the application GUI (network connection is another story not discussed here) are major factors for degrading performance.

2. **Class Omissions:** At the analysis phase, domain analysis class omission is a common mistake. This is mainly due to the fact that the analysis of text based requirements does not allow the visualization and analysis of classes as do models and diagrams. With RA, this can be achieved through the analysis of entity classes. We mainly investigate the correct usage of each class in RD (are we writing to or reading from the correct entity and do we need a new one?), the existence of all entity classes that support primary and secondary scenarios, and the need to split, merge, or even removing entities from the domain model.

3. **Design errors:** Design errors are the most devastating errors because they could affect the whole system and are costly to fix as we move into the detailed design and code implementation. Design errors can happen due to weak design strategy, missing, wrongful or misunderstanding requirements.

4. **Inconsistency:** Inconsistency can exist among use cases (for example, same entity or boundary class can be referred to differently in text), and among the use case text, its associated RD and interaction diagrams. RA allows the correct and consistent writing of use cases (use cases are written with correct voice [ROSENBERG 99]]), and to remove inconsistencies between requirements and design when reading and analyzing use case text to construct the RD.

We now give examples from our case study of the first three types of errors. The fourth error type is straightforward and not discussed or illustrated here.

5.2 INFEASIBLE SPECIFICATION EXAMPLE: GET QUOTE

In this section, we will see that RA can detect infeasible specification in the ***Get Quote*** use case example. The ***Get Quote*** example consists of providing the customer with real time quotes for securities listed on North American stock exchanges. We start the example with the use case text. Next, we construct its RD followed by interaction diagrams (sequence and collaboration diagrams).

5.2.1 GET QUOTE USE CASE

Actors

- Customer

Precondition

- The customer has logged into the system.

Flow of Events

Basic Path

1. The use case starts when the customer selects Quote from the Main NetBroker web page.
2. The system displays the Get Quote web page.
3. The customer selects a company name from the company list.
4. The customer selects a market from the market list.
5. The customer clicks on the Get Quote button.
6. The system validates the data.
7. USE Display Quote use case.
8. The use case ends.

Alternative Paths:

1. The customer can select any other options at any time before selecting Get Quote.

The quote page is not displayed and the use case ends

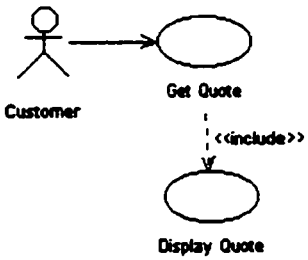


Figure 5.1: Get Quote Context Diagram

Postcondition

- The Display Quote web page is displayed.

Secondary Scenarios

- Company is not supplied.
- Market is not supplied.

An error message is displayed to inform the user about missing information or wrong data entered.

Special Requirements

- The system must return a quote within 10 seconds.

5.2.2 GET QUOTE ROBUSTNESS DIAGRAM

We apply the RD construction rules and noun analysis given in section 3.3 to classify the elements by RA type. The identifiable RD elements for the Get Quote use case are shown in Table 5.1 below:

Element Type	Element Description
Actor	Customer
Boundary	Get Quote web page, Company List, Market List, and Get Quote button.
Control	Display Error, Validate Data, Pick Company, and Pick Market.
Entities	Company, and Market
Use Cases	Display Quote

Table 5.1: RD Elements for Use Case Get Quote

We have identified and included all entity classes (***Company*** and ***Market*** entity classes) by construction rule #5 (1st condition). We have also identified the ***Market List*** boundary class by construction rule #5 (2nd condition, Part a is false). In addition, we have identified the ***Pick Market*** control class by construction rule #5 (3rd condition, Part a, b, and c are false).

After constructing the RD of the Get Quote use case, we want to analyze if its specification is realistic by examining boundary and entity classes [Figure 5.2]. We observe that all boundary classes are standard GUI classes that can be realized with no performance effect. For entity classes, the ***Market*** entity class is relatively small with at most 10 records. On the other hand, the ***Company*** entity class could return a large number of records (above 1000 on major stock exchanges). Given the application type (we are building a web application and customers are using modems to connect), this could have a negative impact on the application performance.

A solution, i.e. using the company stock symbol instead of the company name, can be derived from our expertise and knowledge of the application domain. In fact, investors are familiar with the company stock symbol; hence, we can safely replace the ***Company List*** boundary class with a standard text box to enter the company stock symbol. Thus, no data is required to download, and performance is optimized.

Are there not faster and cheaper techniques that might catch the same thing such as prototyping? In fact, there are some dangers with using only prototyping. For example, the fact that there is an entity in RA raises a serious flag that might be overlooked – an error that might not be caught with prototyping.

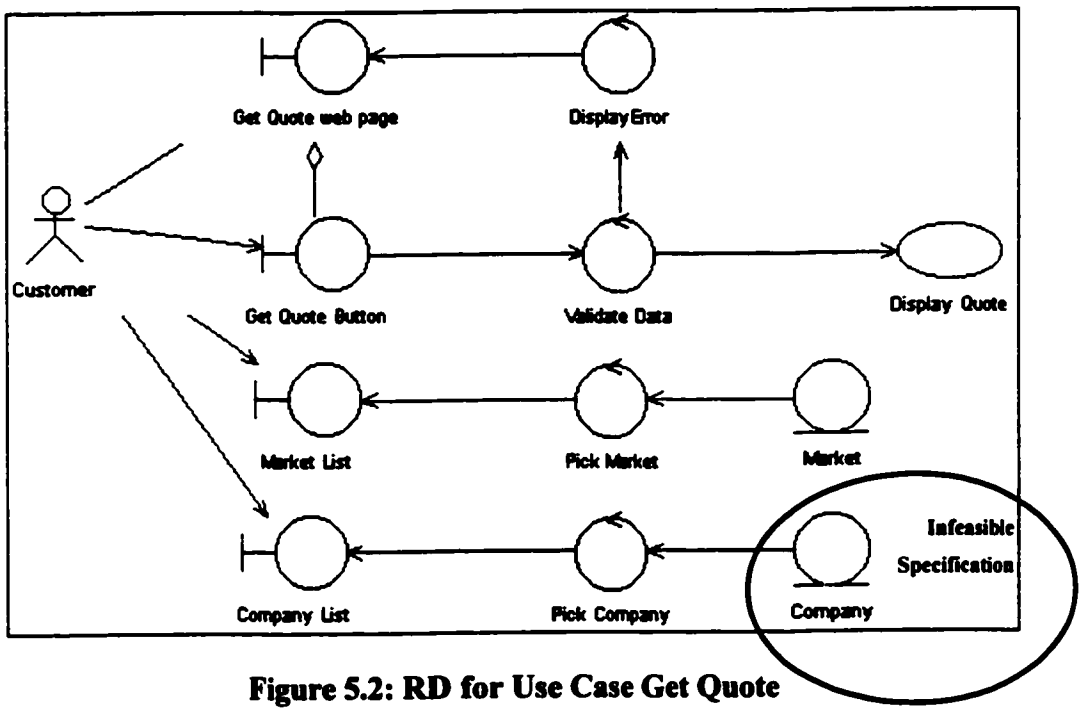


Figure 5.2: RD for Use Case Get Quote

The required steps and changes needed to fix this requirement error are:

1. Remove the ***Company list*** boundary class, ***Pick Company*** control class and ***Company*** entity class from the RD. Note that the ***Company Symbol*** boundary class (text box) does not need

to appear in the RD (because it satisfies construction rule #5, 2nd condition, Part a, b, and c are true), as we do not model every simple boundary. In addition, the use case in step 3 of the use-case basic path should read: ***"The customer enters the company symbol"***.

2. Include a ***Find Symbol*** button boundary class to allow the user to search for company symbols using the company name, which will run a new identified use case, ***Find Company Symbol*** use case. Therefore, this modification should be reflected in the alternative path of the use case text and the ***Find Company Symbol*** use case must be added to our use-case model. In addition, two operations should be added to the stock entity class: ***CheckSymbol*** and ***FindCompanySymbol*** operations.

3. Because the user can enter an invalid company symbol, we need to perform a validation check on the symbol entered and we need to show it on our RD. Therefore, we add the ***Stock*** entity class to RD and we draw an arrow from the ***Stock*** entity class to the ***Validate Data*** control class. This validation should also be reflected in the use-case secondary scenarios.

The resulting enhanced version of the Get Quote use case and its RD [Figure 5.3] are given below:

Enhanced GET Quote Use Case:

Actors

- Customer

Precondition

- The customer has logged into the system.

Flow of Events

Basic Path

1. The use case starts when the customer selects Quote from the Main NetBroker web page.
2. The system displays the Get Quote web page.
3. The customer enters the company symbol.
4. The customer selects a market from the market list.
5. The customer clicks on the Get Quote button.

6. The system validates the data.
7. USE Display Quote use case.
8. The use case ends.

Alternative Paths:
1. The customer can click on the Find Symbol button. The Find Symbol use case is executed.
2. The customer can select any other options at any time before selecting Get Quote.

The quote page is not displayed and the use case ends

Postcondition

- The Display Quote web page is displayed.

Secondary Scenarios

- Symbol is not supplied.
- Market is not supplied.
- Symbol does not exist for the specified market.

An error message is displayed to inform the user about missing information or wrong data entered.

Special Requirements

- The system must return a real time quote within 10 seconds.

Enhanced Get Quote RD:

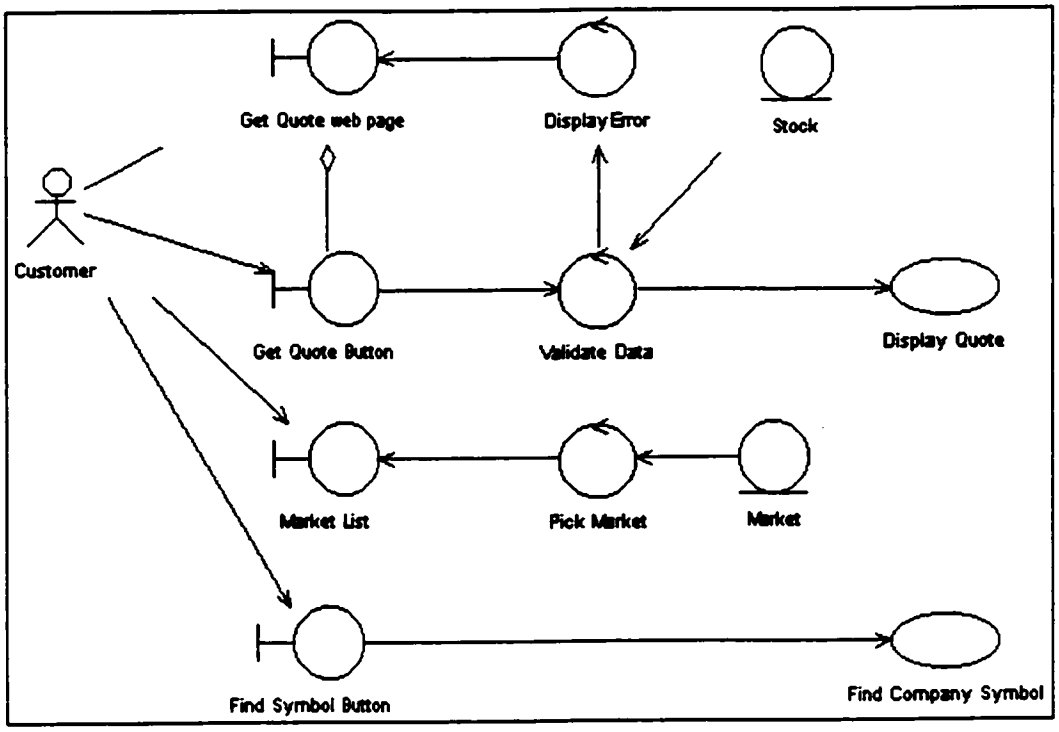


Figure 5.3: RD for Enhanced Use Case Get Quote

This correction will lead to feasible interaction diagrams as shown next.

5.2.3 GET QUOTE INTERACTION DIAGRAMS

The sequence and collaboration diagram for the use case ***Get Quote*** are shown in Figure 5.4 and Figure 5.5 respectively.

The first step to create interaction diagrams is to copy the actor, boundary and entity classes from the RD to the sequence diagram. Next, we show the interactions (modeled as messages) among the classes. Most of these messages will emanate from the RD control classes such as ***Display Error,*** and ***Validate Data***. Other messages (or methods) that are required to complete use case scenarios will be discovered at design.

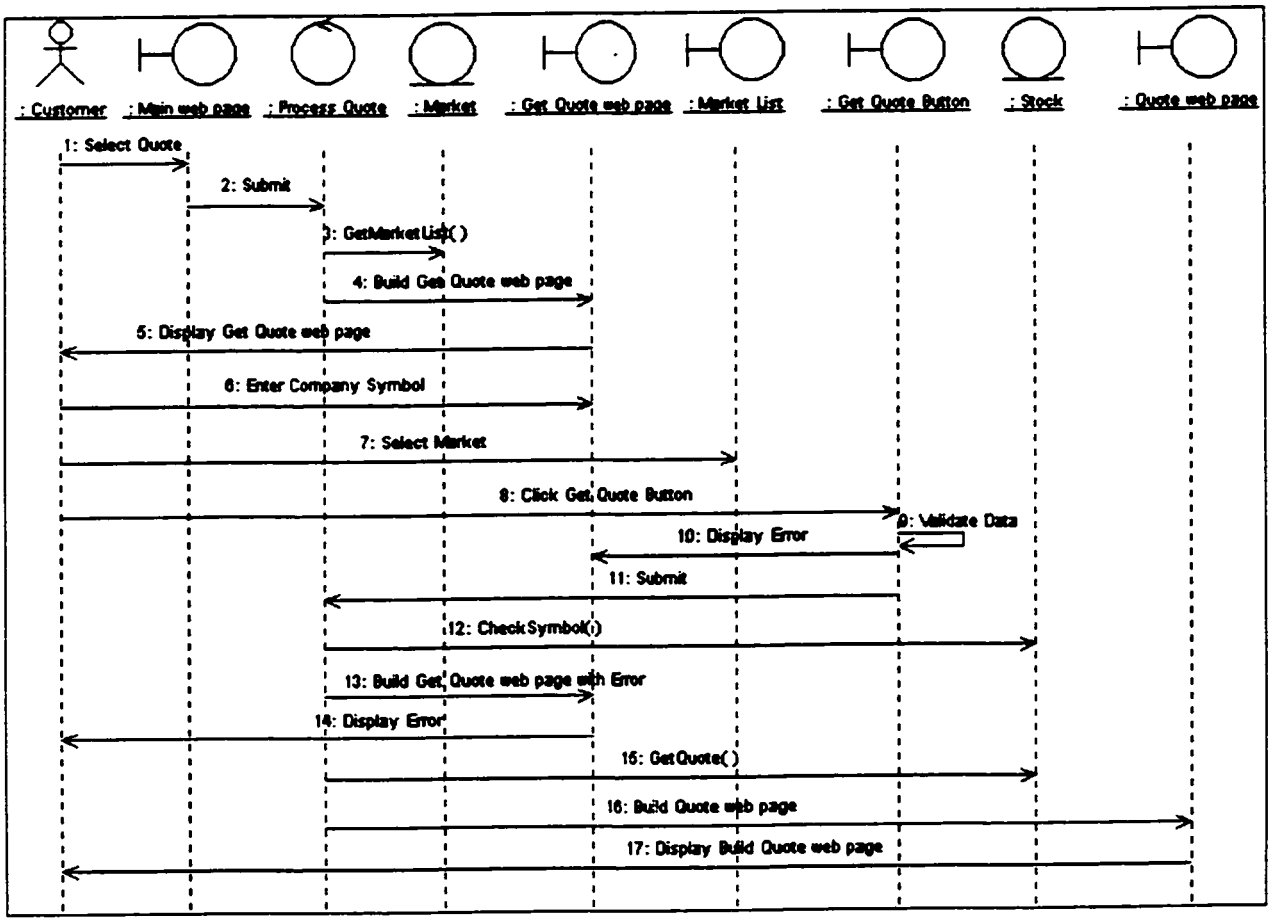


Figure 5.4: Sequence Diagram for Use Case Get Quote

As we move into detailed design, new classes and messages are required to model the use case, as it is the case in our sequence diagram:

- We have created a new boundary class ***Main web page*** to be the start point (or entry point) for executing the use case on the sequence diagram. This class was omitted in RD because we wanted to simplify the RD, however we must show the details on our sequence diagram.

- We have created a new boundary class ***Quote web page*** to represent the actual web page, which contains the resulting quote when the use case ***Display Quote*** is executed (see Figure

5.3). The ***Quote web page*** boundary class contains real-time quote details of the securities specified by the customer.

- We have created a new control class ***Process Quote*** to process and validate customer's requests on the Web server. When the customer selects Quote from the Main NetBroker web page, a request is submitted to the web server. Given the case study assumption stated in the previous chapter (the target architecture for our case study involves the use of scripted pages), we need to model server-side object activity in the interaction diagram. Therefore, we need to create a new control class ***Process Quote*** to act as scripted page of server-side object activity [CONALLEN 00].

- When the customer request to view the ***Get Quote web page***, the ***Process Quote*** control class builds it. Since the ***Get Quote web page*** must contain a list of available markets, the ***Process Quote*** control class sends a message (***GetMarketList***) to the ***Market*** entity to retrieve data. Once the data is received by the ***Process Quote*** control class, it builds the Get Quote web page and returns it to the customer.

- At design, we are now making decisions that were not appropriate during the analysis phase. The ***Validate Data*** control class validates *passive data* (data that can be validated by the class within the scope of its instance such as verifying if the stock symbol has been entered) represented with a message-to-self on the ***Get Quote*** boundary class. If data is not supplied (stock symbol or market) an error message will be displayed, otherwise the ***Get Quote web page*** is submitted. When the ***Process Quote*** control class receives the submitted ***Get Quote web page***, it sends the message ***CheckSymbol*** to the ***Stock*** entity class to check if the symbol-market combination exists. This will result in either building the ***Get Quote web page*** with an error message attached to it or retrieving the quote (using the ***GetQuote*** message) before building the ***Quote web page***.

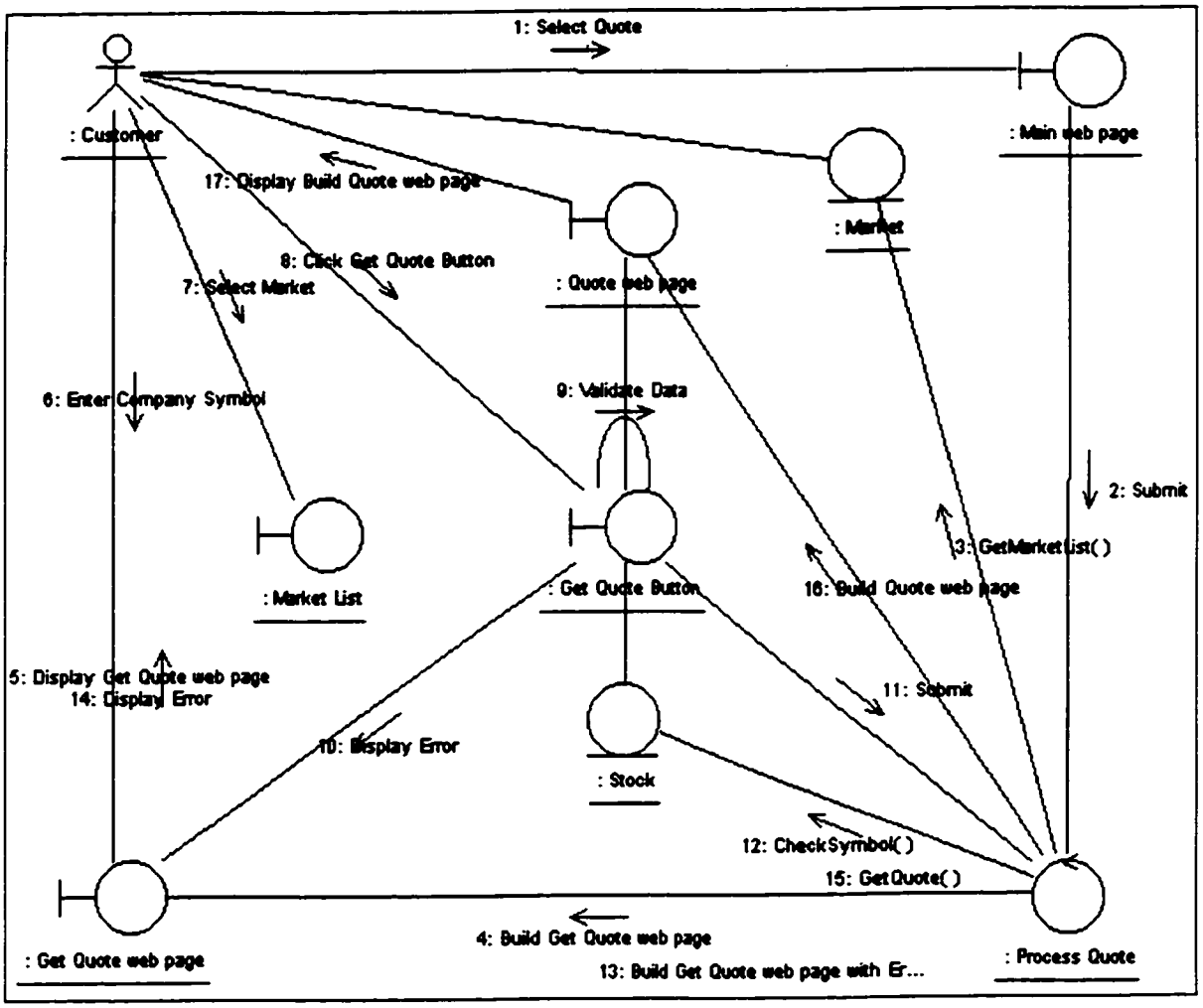


Figure 5.5: Collaboration Diagram for Use Case Get Quote

5.3 CLASS OMMISSIONS EXAMPLE: BUY ORDER

In this section, we will see that RA can discover new classes that we omit during requirement analysis in the Buy Order use case example. The ***Enter Buy Order*** example allows the customer to enter an order to buy securities. We start the example with the use case text. Next, we construct its RD followed by interaction diagrams (sequence and collaboration diagrams).

5.3.1 BUY ORDER USE CASE

Actors

- Customer

Precondition

- The customer has logged into the system.

Flow of Events

Basic Path

1. The use case starts when the customer selects Order from the web page menu option.
2. The system displays the Order web page.
3. The customer selects an account from the account list.
4. The customer enters a symbol.
5. The customer selects a market from the market list.
6. The customer selects buy from the Action list.
7. The customer enters number of shares.
8. The customer enters a price.
9. The customer enters a 'good till' date.
10. The customer clicks on the Submit Order button.
11. The system validates the data, saves the order.
12. USE Display Order Confirmation use case.
13. The use case ends.

Alternative Paths:

1. The customer can click on the Reset button.
2. The customer can select any other options at any time before selecting Submit Order button.

The order is not submitted and the use case ends.

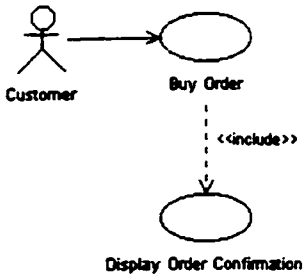


Figure 5.6: Buy Order Context Diagram

Postcondition

- The order is saved to the database.

- The Display Order Confirmation web page is displayed.

Secondary Scenarios

- Account is not supplied.
- Symbol is not supplied.
- Market is not supplied.
- Action is not supplied.
- Number of shares is not supplied.
- Price is not supplied.
- Good till date is not supplied.
- Symbol does not exist for the specified market.
- The specified price is far off the current market price.
- Insufficient cash or margin for the specified account.

An error message is displayed to inform the user about missing information or wrong data entered.

Special Requirements

- The system must return an order confirmation within 10 seconds.

5.3.2 BUY ORDER ROBUSTNESS DIAGRAM

We apply the RD construction rules and noun analysis given in section 3.3 to classify the elements by RA type. The identifiable RD elements for the Buy Order use case are shown in Table 5.2 below:

Element Type	Element Description
Actor	Customer
Boundary	Order web page, Market List, Account List, and Submit button.
Control	Display Error, Validate Data, Save Data, Pick Account, and Pick Market.
Entities	Stock, Order, Market, and Account.
Use Cases	Display Order Confirmation.

Table 5.2: RD Elements for Use Case Buy Order

We have identified and included all entity classes (***Stock, Order, Market*** and ***Account*** entity classes) by construction rule #5 (1st condition). We have also identified the ***Account List*** boundary class by construction rule #5 (2nd condition, Part a is false). In addition, we have

identified the ***Validate Data*** control class by construction rule #5 (3rd condition, Part a, b, and c are false).

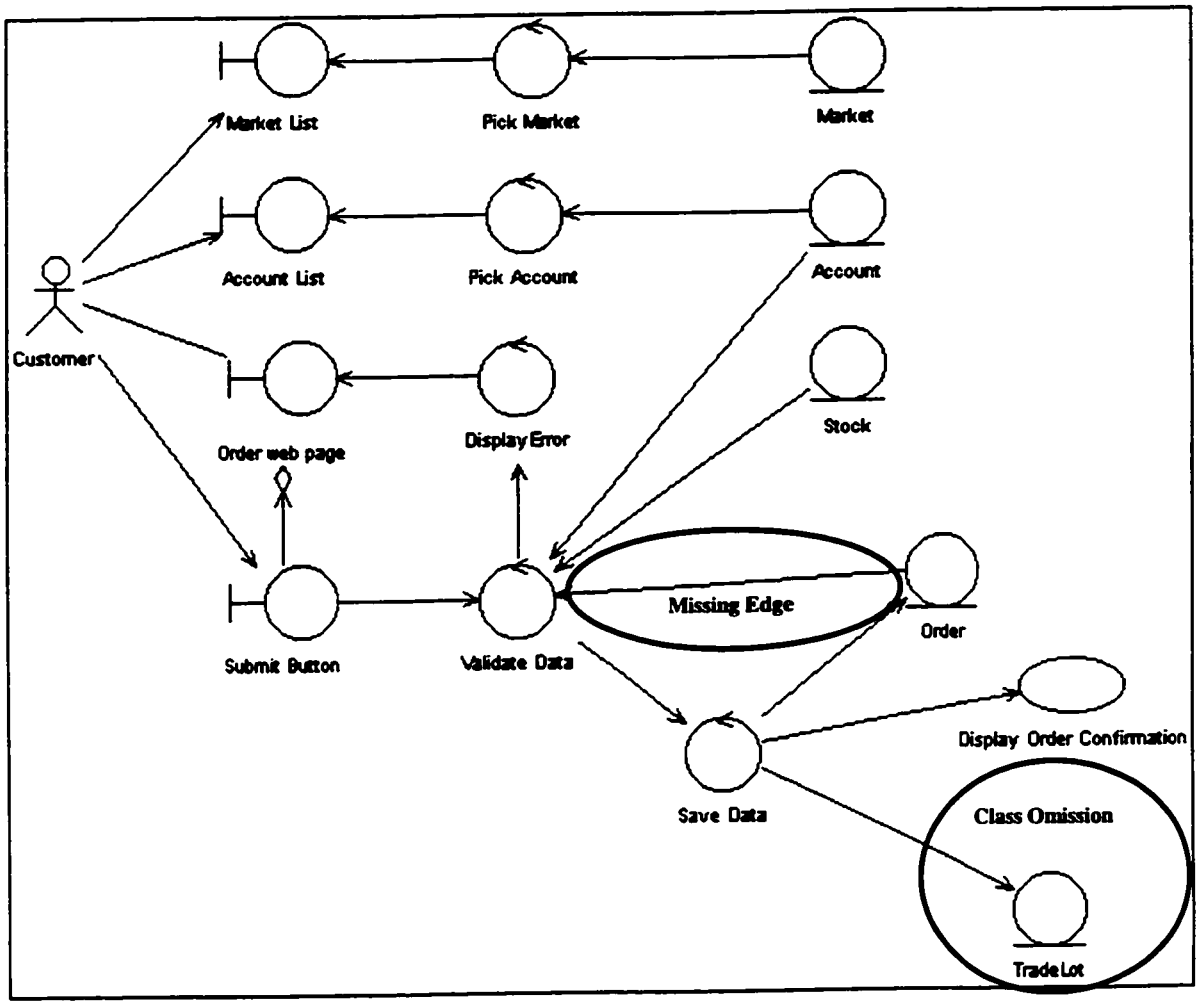


Figure 5.7: RD for Use Case Buy Order

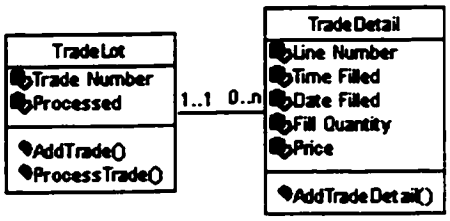
A close analysis of the use-case basic path in RD [Figure 5.7], we realize that we need a ***TradeLot*** entity class for the ***Trading System*** to process later the new order with the ***Security Exchange System***. The new entity class will contain the ***order number*** provided by the ***Order*** entity, the ***trade number*** provided (later when order is executed) by the ***Security Exchange***

system and a flag to filter orders that has (or has not) been processed. Moreover, we do need a ***TradeDetail*** entity class to show the details of the order when it is filled or partially filled.

Another analytical bug that we uncover is that we need an arrow between the ***Validate Data*** control class and the ***Order*** entity to ensure that the customer cash or margin balance can cover the value of the current order in addition to all opened orders (not yet filled).

The required steps and changes needed to fix these requirements errors are:

1. Add the ***TradeLot*** and ***TradeDetail*** entity classes to the domain model with required attributes and operations.



2. Adjust the use case text (basic path) to reflect the requirements of saving data to the ***TradeLot*** entity class.

3. Add to the use case secondary scenario a validation check to reject the order if the value of the new order plus the value of all opened orders exceeds the customer's account cash or margin balance.

In this case, we see that RA enabled us to detect an omitted class.

5.3.3 BUY ORDER INTERACTION DIAGRAMS

The sequence and collaboration diagrams for the use case ***Buy Order*** are shown in Figure 5.8 and Figure 5.9 respectively. The construction of the interaction diagrams for this use case is similar to the use case ***Get Quote***.

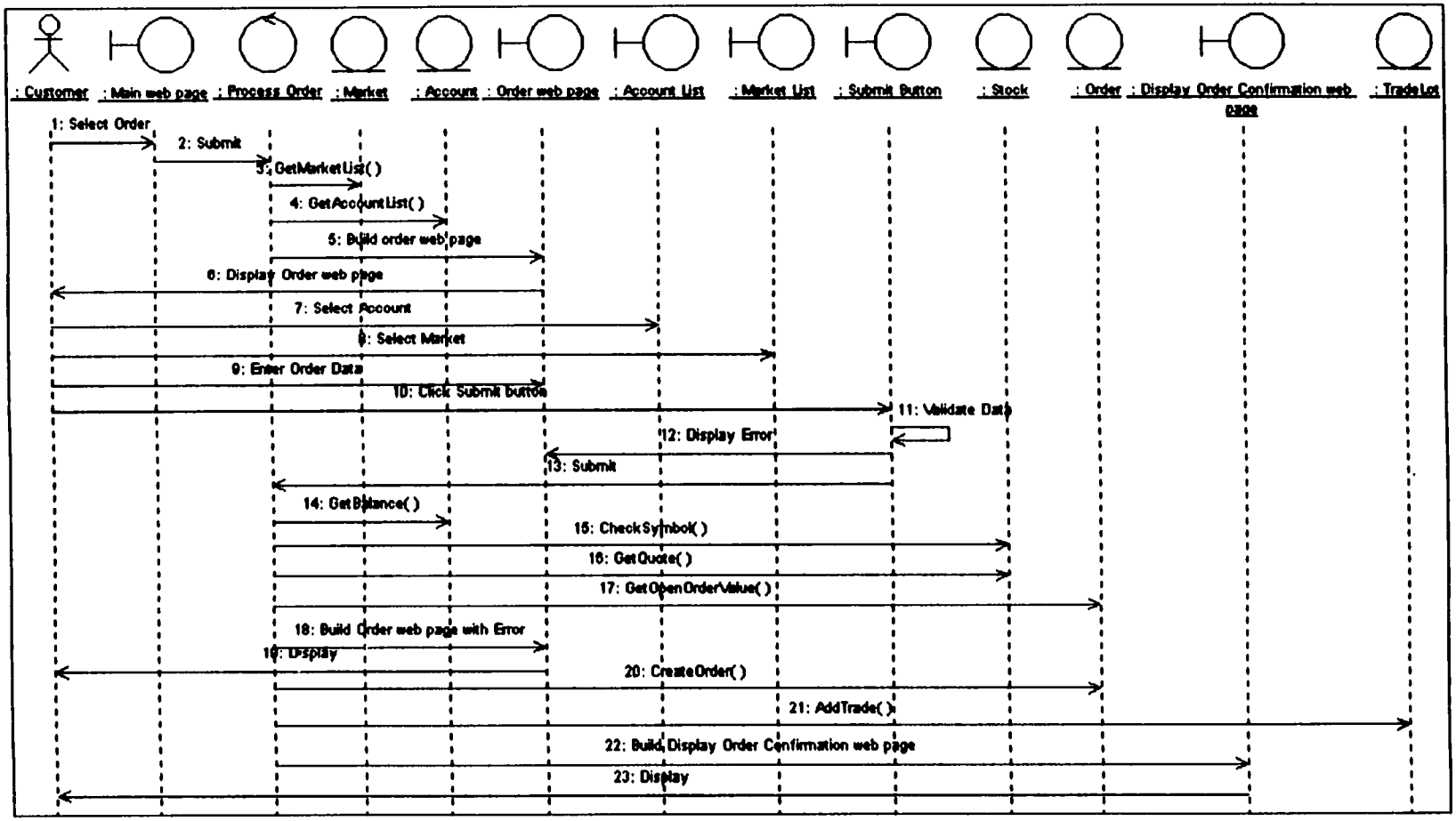


Figure 5.8: Sequence Diagram for Use Case Buy Order

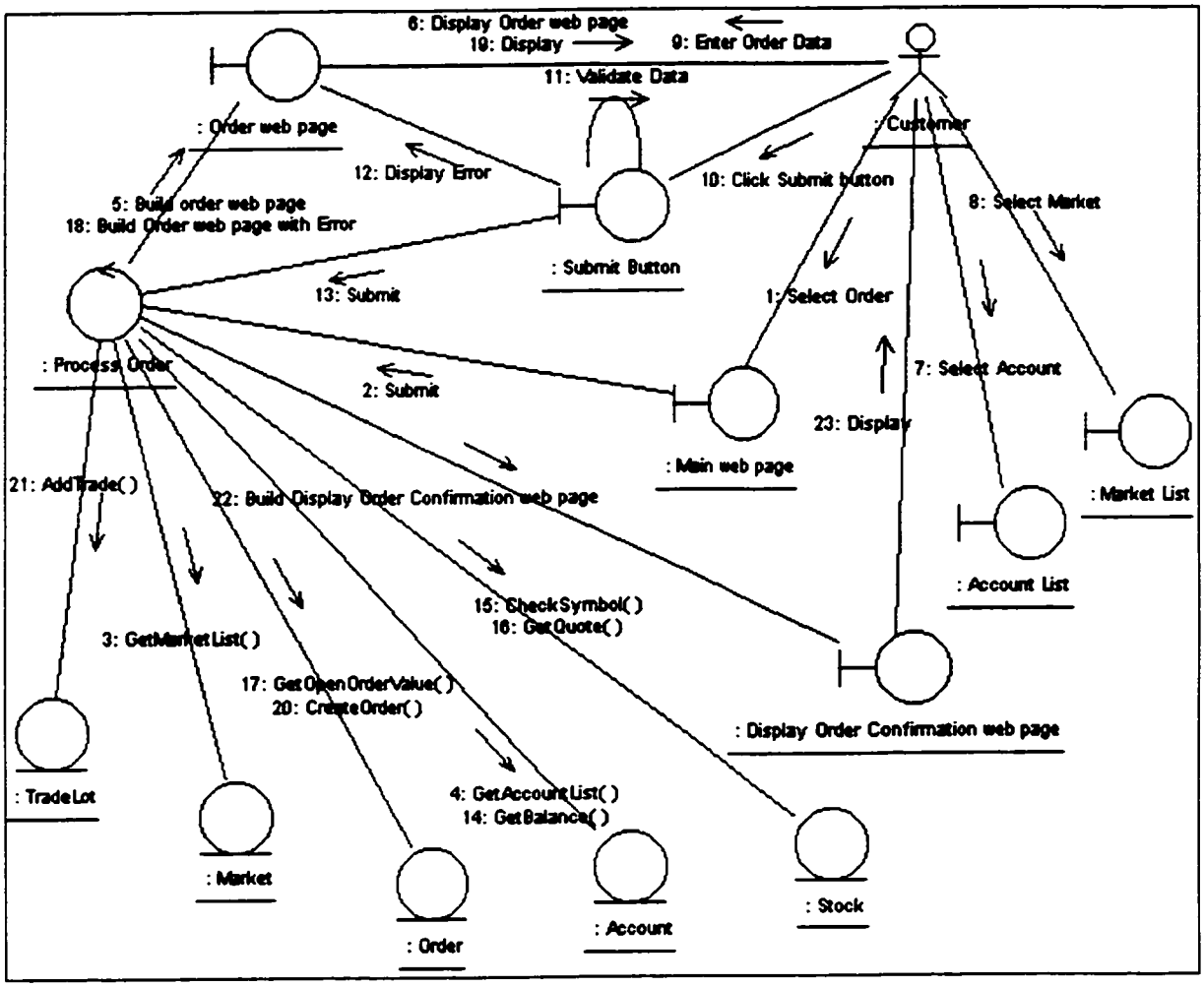


Figure 5.9: Collaboration Diagram for Use Case Buy Order

5.4 DESIGN ERRORS: FILL BUY ORDER RECEIVED

In this section, we will see that RA can detect design errors in the ***Fill Buy Order Received*** use case example. The ***Fill Buy Order Received*** use case is an extended use case of the ***Order Received*** use case. It occurs when the ***Trading System*** is notified that an order has been received and that the order is a buy order that has been filled or partially filled at the

appropriate stock exchange. We start the example with the use case text. Next, we construct its RD followed by interaction diagrams (sequence and collaboration diagrams).

5.4.1 FILL BUY ORDER RECEIVED USE CASE

Actors

- System

Precondition

- The system is in wait mode.

Flow of Events

Basic Path

1. The use case starts when the system is notified that a buy order has been filled.
2. The system starts the Order Receipt process.
3. The system updates the order status to fill.
4. The system saves the trade detail.
5. The system ends the process.
6. The use case ends.

Alternative Paths:
None.

Postcondition

- The order is updated and the trade detail data is saved.

Secondary Scenarios

- None

Special Requirements

- None.

5.4.2 FILL BUY ORDER RECEIVED ROBUSTNESS DIAGRAM

We apply the RD construction rules and noun analysis given in section 3.3 to classify the elements by RA type. The identifiable RD elements for the Buy Order Received use case are shown in Table 5.3 below:

Element Type	Element Description
Actor	Trading System.
Boundary	Order Receipt Process.
Control	Save Data.
Entities	Order, and Trade Detail.
Use Cases	None.

Table 5.3: RD Elements for Use Case Fill Buy Order Received

We have identified and included all entity classes (***Order,*** and ***Trade Detail*** entity classes) by construction rule #5 (1st condition). We have also identified the ***Order Receipt Process*** boundary class by construction rule #5 (2nd condition, Part a is false). In addition, we have identified the ***Save Data*** control class by construction rule #5 (3rd condition, Part a, b, and c are false).

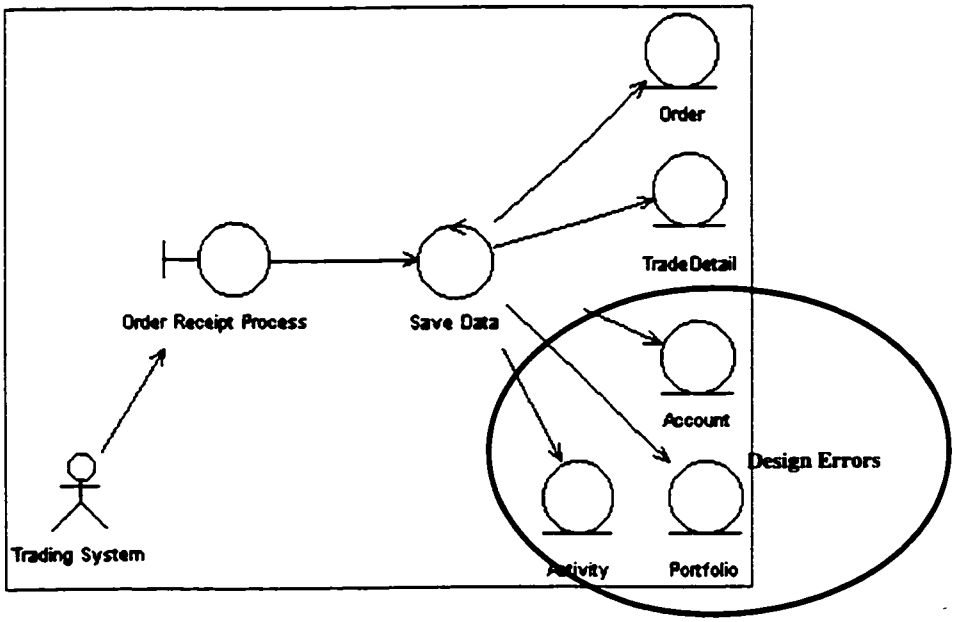


Figure 5.10: RD for Use Case Fill Buy Order Received

The use case states that we need to update the order status to fill, and to save the trade detail data. However, we also need to update both the ***Portfolio*** and ***Account*** and ***Activity*** entity classes as shown in [Figure 5.10].

If the ***Portfolio*** and ***Account*** entity classes are not updated, we will be faced with a major design error, an error that we have observed at one of the leading discount brokerages TD Waterhouse [TDWATERHOUSE]. The impact of this design error on the system is spread to main application components as described below:

1. When a buy order is filled, the ***Portfolio*** entity class should be updated. If the security exists in the portfolio, then (1) the number of shares should be augmented; otherwise, (2) a new entry should be created. Failing to do so, the system would wrongly inform the customer who is selling the security that (1) the number of shares exceeds his holdings or (2) the security does not exist!

2. When a buy order is filled, the ***Account*** entity class should be updated. The value of the transaction should be extracted from the account cash balance, and the account margin call balance should be recalculated and lowered by (1 – margin percent)*(transaction value). Failing to do so, the system would wrongly allow the customer to over-buy securities despite insufficient cash or margin to cover!

3. When a sell order is filled, the ***Portfolio*** entity class should be updated. The number of shares sold should reduce securities' holding in the portfolio. Otherwise, the system would wrongly allow the customer to sell securities more than once!

4. When a sell order is filled, the ***Account*** entity class should be updated. The value of the transaction should be added to the account cash balance and the account margin call balance should be recalculated and augmented by (margin percent)*(transaction value). Failing to do

so, the system would wrongly prohibit the customer to buy other securities because of insufficient cash or margin to cover!

The reason behind not updating both the ***Portfolio*** and ***Account*** entity classes is TD Waterhouse proper strategy. Any security bought or sold is not reflected on the same day, it is rather postponed until the next day when overnight maintenance takes place. From a customer viewpoint, this is frustrating and unacceptable. Given the market volatility or the customer type (the customer could be an active trader also known as day trader), the customer might buy and sell the security more than once during the day. Unfortunately, the only remedy to this design error is to call a TD Waterhouse customer representative who will perform the transaction on behalf of the customer.

This correction (detecting and resolving design errors) will lead to a better design of interaction diagrams as shown next.

5.4.3 FILL BUY ORDER RECEIVED INTERACTION DIAGRAMS

The sequence and collaboration diagrams for the use case Fill Buy Order Received are shown in Figure 5.11 and Figure 5.12 respectively. The construction of the interaction diagrams for this use case can be easily made from the previous RD.

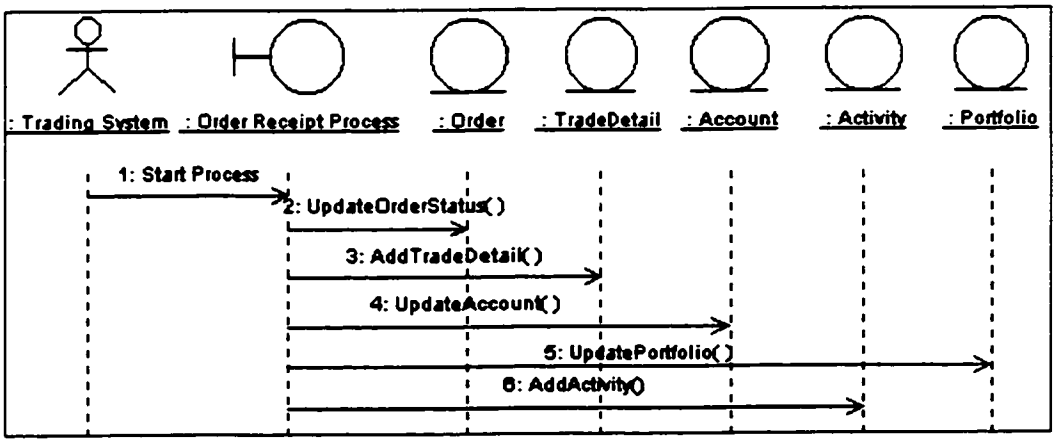


Figure 5.11: Sequence Diagram for Use Case Fill Buy Order Received

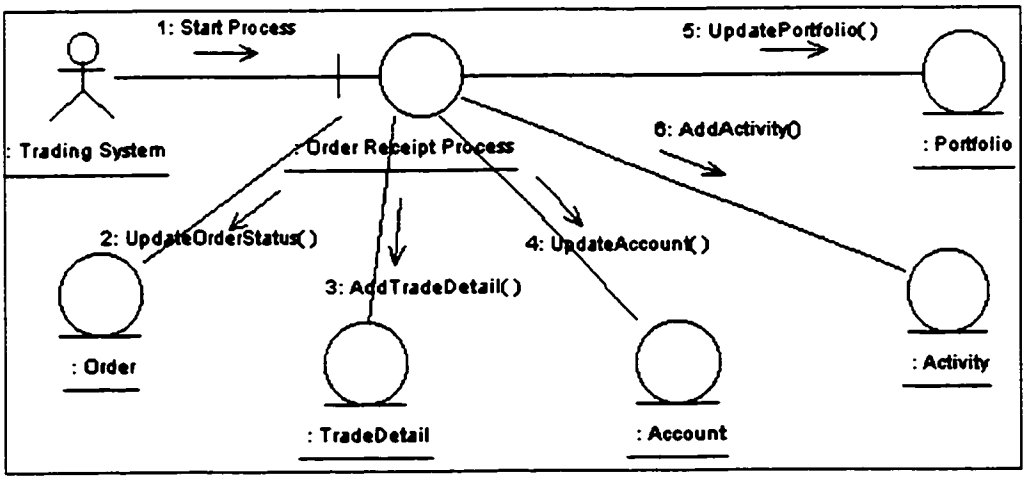


Figure 5.12: Collaboration Diagram for Use Case Fill Buy Order Received

5.5 ARCHITECTURAL PATTERN

A *pattern* addresses and presents a solution to a recurring design problem that arises in specific design situations [KRUTCHEN 98]. Examples of architectural patterns are model-view-controller (MVC) and Object Request Broker (ORB) [OMG 95].

The MVC pattern divides an interactive application into three components. The model contains the core functionality and data (entity classes), views display information to the user (boundary classes) and controllers handle user input (control classes). Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

With RA, classes are partitioned into the three stereotype classes within a MVC pattern, an important foundation for building an N-Tier architecture application.

The N-Tier architecture model involves designing the application in several logical layers or tiers [MICROSOFT 01], [FOWLER 97B]. A multi-tier application, in general, consists of several layers [Figure 5.13]:

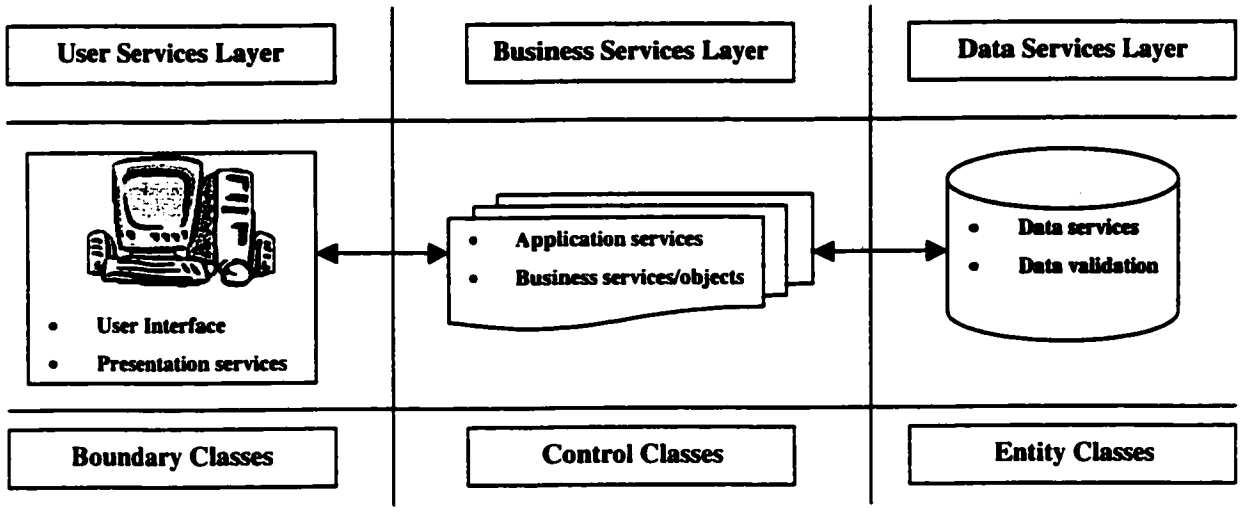


Figure 5.13: N-Tier Architecture Mapping

- A user services layer gives a user access to the application. This layer presents data to the user and optionally permits user data manipulation and data entry (boundary classes).
- A business services layer consists of business and data rules (control classes).
- A data services layer interacts with persistent data stored usually in a database or in permanent storage (entity classes).

Boundary classes are part of the GUI interface; hence, they belong to the user services layer. Entity classes often map to the database tables and files that hold the information; hence, they belong to the data services layer. However, control classes are different. Control classes do not necessarily belong to the business services layer as they could end up being operations on either boundary, entity or broaden control classes, could merge (or split) with other control classes, or simply not considered. For example;

- The ***Display Error*** control class of our previous RD examples will be most likely a method of a more general control class called ***Error Handler*** as it makes more sense to bundle all error messages in one class. This is specifically important in a multi-language application.

- The ***Validate Data*** control class of our previous RD examples could end up belonging to one or both boundary and control classes depending on the application architecture and performance required. For instance, in the ***Get Quote*** use case example, the ***Validate Data*** control class contains both ***passive*** data validation (validation that does not require data communication) and ***active*** data validation (validation that requires data communication). Since we are building a web application, we do not want to validate passive data on the server (such as the company symbol is not blank). Therefore, the data validation could happen either within the ***Get Quote Button*** click event or as a new operation of the ***Get Quote web page*** boundary class. On the other

hand, the active data validation must be validated on the server (such as the company stock is valid for the specified market). In this case, we could call the ***CheckSymbol*** operation of the Stock entity class from a server business class.

While the user interface (boundary classes) and presentation services (control classes as messages) run on the client machine, application services and business services objects (business control classes) could either run on a client machine (ActiveX controls and Java applets for a *thick* web client [CONALLEN 00]), or completely on the application server (*thin* web client). However, data validation and data services (entity classes) run only on the server. In addition, the data services layer could split into multiple data services layers to allow multiple accesses to data repository (Oracle database or Microsoft SQL Server Database).

Thus, RA fits well with the N-tier architecture.

5.6 SUMMARY

In this chapter, we demonstrated how RA can detect specific types of errors early in the analysis and design process by validating and verifying user requirements for three use case examples. In the next chapter, we will assess and evaluate the RA technique from a cost-benefit analysis perspective.

CHAPTER 6

EVALUATION OF ROBUSTNESS ANALYSIS

*I*n this chapter, we present a cost-benefit analysis of RA based validation. We assess and evaluate the benefits of RA technique, and we discuss its disadvantages and associated cost.

6.1 BENEFITS OF RA

In this section, we evaluate the benefits of using RA that includes improvements in Software Quality factors, and facilitating, preliminary and iterative design.

6.1.1 IMPROVEMENTS IN SPECIFIC SOFTWARE QUALITY FACTORS

Software quality is a complex mix of factors that will vary across different applications and the customers who request them [PRESSMAN 97]. According to [MCCALL 77], there are two levels of quality attributes that affect software quality: Quality factors that can not be directly measured and quality criteria that can be measured subjectively or objectively. In this section, we discuss the contribution of the RA technique to software quality factors mainly traceability, completeness, consistency, correctness and robustness.

6.1.1.1 TRACEABILITY

Traceability is the ability to link software components to requirements [MCCALL 77]. The Models Traceability Links (MTL) as defined by [BEN HAJLA 97] occur among all artifacts produced during an iteration of the development process. It links together the user requirements, the analysis models, the design models, code and test cases.

This traceability facet is well preserved under the RA activity by definition. RA provides traceability between what the system does (user requirements) and how the system works (design models) using a conceptual object view.

Traceability with RA is bi-directional: forward and backward navigation mechanisms exist. A forward traceability is realized (1) between use cases and RA as each use case is analyzed to produce RD and (2) between RD and Interaction Diagrams as each RD is transited into detailed design. On the other hand, a backward traceability is realized by tracking each object from design into RA to requirements.

All visual modeling software tools provides support to traceability at the object level. For every class, a list of all locations can be produced showing its usage across all diagrams. However, these tools do not provide a version control mechanism to enable the traceability of changes in one iteration or among iterations.

A traceability document can be maintained throughout the project to ensure the traceability across all artifacts. Table 6.1 identifies the usage and reference of each entity class in use cases, RD and interaction diagrams:

The first column contains the name of each entity class from the domain model. Each subsequent column contains the name of the use case that is divided into three columns: Use case id, RD id, and interaction diagram id. If a class participates in one of the three columns, a check box will be shown.

Entity Classes	Use Case Name 1			Use Case Name 2			...	Use Case Name M		
	UC_1	RD_1	ID_1	UC_2	RD_2	ID_2	...	UC_M	RD_M	ID_M
Class A	√	√	√					√	√	√
Class B	√			√		√		√	√	
...										
Class N										

Table 6.1: Traceability Document

The table allows us to trace each class by showing its usage and to verify that it has been used correctly using the following rules:

- Each entity class must be used at least in one use case; otherwise, the class is questionable and subject to removal from the domain model. For instance, Class N is not referenced in any of the available use cases, which is not the case with Class A and Class B.

- Each entity class should be referenced in either all three columns (Use case, RD and Interaction Diagram) or none of them. For example, Class A is referenced in all artifacts of Use Case Name 1 and Use Case Name M and not in Use Case Name 2. On the other hand, Class B is partially referenced in Use Case Name 1, Use Case Name 2 and Use Case Name 3.

6.1.1.2 COMPLETNESS

Completeness is the degree to which a full implementation of the required functionality has been achieved [MCCALL 77]. In requirements analysis, completeness means that missing requirements are not present. RA helps make sure that each use case has addressed all alternate courses of action and secondary scenarios. RA also provides an ongoing discovery of classes that eventually completes the domain model.

6.1.1.3 CONSISTENCY

Consistency is the use of uniform design and implementation techniques and notations throughout a project [MCCALL 77]. With use case analysis, consistency means that all use cases are written using a consistent and technical style. RA forces us to write our use cases using the noun-verb-noun consistent style so that statements can be easily translated into RD elements.

6.1.1.4 CORRECTNESS

Correctness is the extent to which a program satisfies its specification and fulfills the customer's mission objectives [MCCALL 77]. The relationship between quality factors and quality criteria shows that the factor correctness is considered a function of the criteria traceability, consistency and completeness. RA ensures that use case text is correct and that we haven't specified system behavior that is unrealizable or unreasonable given the set of classes we have to work with, the current available technology, and probably project timeframe or available resources. This is because RA allows a reuse pass across all use cases before starting the design phase.

6.1.1.5 ROBUSTNESS

The RA technique improves the robustness of high-level design. This is achieved through the refinement of use cases, validation and verification of user requirements, and detection of errors (in particular infeasible specifications, class omissions and design errors). The result is a robust object structure and a strong design that is able to survive unexpected user behaviors.

6.1.2 FACILITATING, PRELIMINARY & ITERATIVE DESIGN

RA provides a first view of the system at a high-level design. The design is preliminary and easy to draw given that we have to work with only three class types. The result is a RD for each use case which is quicker to draw and simpler to read than interaction diagrams.

The analysis and design process is iterative in which the list of classes change as time moves on. Several iterations occur between developing the domain model and analyzing the use cases while other iterations occur involving the domain model, RD and interaction diagrams. Thus, RA by mean of RDs yields a bridge between iterations.

6.2 DISADVANTAGES OF RA

The main disadvantages of RA are as follows:

- RA can be only applied to a use-case-driven process such as RUP and ICONIX. Fortunately, these are popular processes.

- Each software engineer or analyst needs to learn RA. Fortunately, the rules are simple and only three class types are used.

- RA is new in the industry with few documented case studies. Moreover, tutorials and documents on the method are limited in availability.

- The visual modeling tool to be used must support RA by allowing the construction of RD and the representation of its associated stereotype classes.

- A sufficient time must be allocated to perform RA and to construct RD. We estimate in section 6.3.3 about one to two hours per RD.

- RA becomes harder to maintain during the implementation phase as design changes are introduced.

- RA is best suited for new software development. It is less recommended for enhancement or conversion projects where RA was not previously used.

6.3 COSTS OF RA

6.3.1 LEARNING UML & RA

To learn UML, several sources are available. Some of these sources include books that vary from beginners to professionals [ALHIR 98], [FOWLER 97A], [LARMAN 97], [OESTEREICH 99], [SCHACH 98]. White papers and documentation on UML are also available and can be downloaded freely or for a relatively low cost from the Internet [RATIONAL 01B], [AMBLER 98]. Other effective sources for learning are UML training courses that vary from 2 to 5 days. Nevertheless, we find that the best way to master UML is to practice and apply it on a real application because the challenge that the real world offers can not be captured in literature.

To learn RA, unfortunately the only available sources are the book by [JACOBSON 92], [ROSENBERG 99] and this thesis. The in-depth discussion of RA in this thesis complements the original work done by [ROSENBERG 99] by alleviating difficulties and removing confusion we ourselves found when first using this technique.

6.3.2 AVAILABLE TOOLS

Today, several visual modeling tools that support UML are available and the number is growing as more enterprises are adopting the UML notation. The enterprises need modeling tools to create their analysis and design models. Among the tools that support Robustness Diagram are Rational Rose [RATIONAL 01A] and GDPRO [GDPRO 01]. The main difference,

with respect to RD, between Rational Rose and GDPRO is that RD is recognized by GDPRO as a separate model or artifact whereas, with Rational Rose, RD is created using class diagrams. In the next section, we show how to use Rose to support RA and the construction of RDs.

An evaluation copy of these tools, ranging from one week to one month, can be downloaded from the related company web site. The price can range from $1500 to $5000^{+} US depending on the number of licenses and the product version.

6.3.3 TOOLS FOR CONSTRUCTING RDs

RDs are fast to draw with the use of the three stereotype classes. They are also easier to read than interaction diagrams because they do not deal with detailed design. The main purpose of RDs are to construct a high-level design of the system under construction.

The cost of constructing RDs is relatively low and the time spent in drawing RDs pays off because analysis and design issues are dealt with earlier in the development cycle. Based on our experience, drawing RD may take one to two hours per use case, and thus may not involve excessive time overhead.

If the Rational Rose Enterprise Edition software is your choice for modeling, then creating your sequence diagrams from your RD can be easily automated using the *Rational Rose Scripting* language. The Rational Rose Scripting language is an extended version of the *Summit BasicScriptlanguage* that allows to automate Rational Rose specific functions, and in some cases perform functions that are not available through the Rational Rose user interface.

For example, the following script creates a new sequence diagram from an RD (The RD must be the active model and classes must be selected in the order we want them to appear before running the script) [ROSENBERG 99]:

```
Sub Main
    Dim theModel As New model
    Dim thePackage As New category
    Dim myDiagram As New ScenarioDiagram
    Dim theClasses As New classcollection
    Dim theInstance As New ObjectInstance
    Dim aClass As New class

    Set theModel = RoseApp.CurrentModel
    Set thePackage = theModel.RootUseCaseCategory
    Set theClasses = theModel.GetSelectedClasses
    Set myDiagram = thePackage.AddScenarioDiagram("newSequenceDiagram",1)
    Print "Creating sequence diagram with the following classes:"
    Set theClasses = theModel.GetSelectedClasses

    For i= 1 To theClasses.count
        Print theClasses.GetAt(i).name
        Set aClass = theClasses.GetAt(i)
        Set theInstance = myDiagram.AddInstance("", aClass.name)
    Next i
    mydiagram.Layout
End Sub
```

This script can be further enhanced to allow the copying of only boundary and entity classes (without control classes) if you know in advance that most of your control classes will end up being methods on your boundary and entity classes.

Thus, the cost of providing tools support for RA is mainly the cost of licensing a specific tool, such as GDPRO or Rose

6.4 SUMMARY

In this chapter, we evaluated the usage of RA by discussing the pros and cons of this technique. We highly recommend the use of this method due to its many benefits that outweigh disadvantages and costs. We do find that the following conditions favor this RA technique:

- UML is your company choice of modeling, you possess UML expertise, and a UML visual modeling tool is available and affordable.

- An appropriate period has been allocated during the analysis phase to perform RA and construct RD.

- The project to be developed is a new system and the most important use cases have been completed.

CHAPTER 7

ROBUSTNESS ANALYSIS POINTS

*Y*ou can not control what you do not measure [DEMARCO 82]. Many projects fail due to lack of management control. Predicting the size of the software correctly is the key to reliable project estimates and effective project planning. In this chapter, we introduce a new metric for estimating person effort called *Robustness Analysis Points* (RAP). Then, we compare our metric to two well-known metrics: *Function Points* (FP) and *Use Case Points* UCP).

7.1 PROJECT ESTIMATION

In this section, we give the definition of project effort and cost estimation, and we discuss the benefits and risks involved.

7.1.1 DEFINITION

Project effort and cost estimation is one of the most challenging and important activities in software development. Proper project planning and control is difficult without a sound and reliable estimation metric. *Software metrics* attempts to capture various attributes of a software product such as *Cost*, *Quality*, and *Maintainability*. Metrics provides a means for management to estimate the complexity of a project, the cost involved in developing the product, the manpower required, and the tools required. In this chapter, we propose RAP as a metric for estimating the size of a project, and compare it in our case study with other estimation metrics.

7.1.2 BENEFITS OF ESTIMATION

When the software estimation process is performed correctly, the benefits realized outweigh the cost of doing the estimation [SCHACH 98]. The major benefits include:

- Controlling the cost of doing business.
- Increasing the probability of winning new contracts.
- Increasing and broadening the skills level of key staff members.
- Acquiring a deeper knowledge of the proposed project
- Understanding, refining and applying the proper software life cycle.

Software effort estimation and the estimation method itself should be continuously improved resulting in better estimates, budgeting, and promoting a reputation for timely delivery of products.

7.1.3 RISKS OF ESTIMATION

Under-estimating a project effort leads to under-staffing it (possibly more overtime), under-scoping the quality assurance effort (low quality deliverables), and setting too short a schedule (missing deadlines) [QSM 01].

On the other hand, over-estimating a project can also be bad for the organization. The project is assigned more resources than it really needs, is likely to cost more than it should, may take longer to deliver than necessary, and may delay the use of key personnel the resources on the next project.

7.2 SOFTWARE COMPLEXITY METRICS

Successful organizations adapt estimation techniques and measures for their own use [DEMARCO 90]. In this section, we present and discuss three known software complexity measurements that are used in industry: Lines of Code (LOC), Function Points (FP), and Use Case Points (UCP). Then, in the next section, we propose a new metric called RAP, a variation of UCP.

7.2.1 LINES OF CODE (LOC)

A commonly used approach for measuring program complexity is counting lines of code. Unfortunately, there are no standards for counting LOC and the implied functionality of LOC differs depending on the implementation language. A commonly used definition is given from [CONTE 86]:

> *"A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements."*

But LOC as a complexity metric has a number of drawbacks [LORENZ 94], [ASHLEY 95]:

- LOC is language dependent i.e. a line of code in assembly language provides far less end user function than a line of code in Smalltalk or Cobol.
- LOC is extremely difficult to estimate early on during development. If a complexity metric is used for estimation then it needs to be available as early as possible, certainly at the end of high-level design.
- Code complexity is not reflected in LOC. The code in extensive graphical algorithm counts the same as in a procedure code to set an array to zero.
- The LOC metric encourages larger code volume and hence rewards programmers to write more code rather than more functions (reuse).

- LOC is not a good predictor of quality or progress. When we see an LOC measurement, we have not discovered anything about reliability, performance, maintainability or completeness of what we measured.
- LOC is not meaningful to the user and does not facilitate negotiations or user communication.

In spite of these various disadvantages, LOC has some advantages. First, LOC can be used to estimate project size and effort that will have similar implementation to previous projects using LOC historical data. Secondly, LOC is advantageous to use after a project has been started where LOC can be computed for the piece of work accomplished and used as an estimator for the completion of the new project.

7.2.2 FUNCTION POINTS

In this section, we introduce FP, we give the definition of FP measures and we calculate FP for our case study on-line stockbroker.

7.2.2.1 INTRODUCTION

In 1970, Albrecht and his colleagues at IBM developed the Function Points metric in an attempt to overcome difficulties associated with lines of code as a measure of software size, and to assist in developing a mechanism to predict effort associated with software development [ALBRECHT 79].

Since the method was published in 1979, many variations of the FP method have been introduced such as:

- IFPUG (International Function Point Users Group) Function Point Analysis version 4.1.1 [IFPUG 01].

- Mark II Function Point Analysis version 1.3.1 [UKSMA].
- Full Function Points approach version 1.0 [ST-PIERRE 97].
- COSMIC Full Function Points version 2.0 [COSMIC 99].

The major benefits of using FP are to measure the functionality of the software product in standard units independent of the coding language, to assist organizations to derive unit costs that are critical to understanding overall metrics and project performance, and to provide the correct level of analysis to understand and communicate project functionality [SHEPPERD 93], [SOMMERVILLE 96], [VLIET 96]

However, FP has also some disadvantages. FP is difficult to compute, contains a large degree of subjectivity, raises doubt it actually measures functionality [KAN 95], and can only be applied to data processing applications (for instance, FP can not measure heavily computational software).

In this thesis, we consider only the recent version of IFPUG Function Point Analysis, namely release 4.1.1.

7.2.2.2 DEFINITION OF MEASURES

FP is the product of two factors: Unadjusted Function Point (UFP) and Value Adjustment Factor (VAF) is calculated in six steps, using the following formulas:

$$FP = UFP*VAF \quad (1)$$
$$VAF = (0.01*TDI) + 0.65 \quad (2)$$

Step 1: Determine the Type of Function Point count

The first step in the function point counting procedure is to determine the type of function point count: development project function point count, enhancement project function point count, or application function point count.

For example, the type of function point count for our online stock broker is a development project function point count.

Step 2: Identify the Counting Scope and Application Boundary

The counting scope defines the functionality that will be included in a particular function point count and the application boundary indicates the border between the software being measured and the user.

For example, the boundaries of our online stock broker come from the entities identified in our class diagram in Figure 4.3. However, only ***Portfolio***, ***Account***, ***Activity***, ***Customer***, ***Order***, ***TradeLot*** and ***TradeDetail*** are maintained within the application boundary.

Step 3: Count the Data Functions to Determine their Contribution to the UFP Count

Data functions represent the functionality provided to the user to meet internal and external data requirements. Data functions are either internal logical files or external interface files. An internal logical file (ILF) is a user identifiable group of logically related data or control information maintained within the boundary of the application. An external interface file (EIF) is a user identifiable group of logically related data or control information referenced by the application, but maintained within the boundary of another application.

For both ILF and EIF, the number of *Record Element Types* (RET) and the number of *Data Elements Types* (DET) are used to determine a ranking of low, average or high [Table 7.1], [Table 7.7]. A RET is a user recognizable subgroup of data elements within an ILF or EIF. A DET is a unique user recognizable, non-recursive field on an ILF or EIF.

	1 to 19 DET	20 to 50 DET	51 or more DET
1 RET	Low	Low	Average
2 to 5 RET	Low	Average	High
6 or more RET	Average	High	High

Table 7.1: RET and DET Weights

For example, the data functions and the number of DETs and RETs of our online stock broker are identified in [Table 7.2].

	Type	DET	RET	Rating
Portfolio	ILF	4	1	Low
Order	ILF	9	1	Low
TradeLot	Not a user identifiable group.			
TradeDetail	ILF	6	1	Low
Account	ILF	7	1	Low
Activity	ILF	9	1	Low
Customer	ILF	6	1	Low
Fundamentals	ELF	13	1	Low
Stock	ELF	13	1	Low
Market	ELF	3	1	Low
Company	ELF	4	1	Low

Table 7.2: On-line Stock Broker Data Functions Count

Step 4: Count the Transactional Functions to Determine their Contribution to the UFP Count

Transactional functions represent the functionality provided to the user to process data. Transactional functions are either External Inputs (EI), External Outputs (EO), or External Inquiries (EQ). An EI is an elementary process that processes data or control information that

comes from outside the application's boundary. An EO is an elementary process that sends data or control information outside the application's boundary. An EQ is an elementary process that sends data or control information outside the application boundary.

A File Type Referenced (FTR) is an internal logical file read or maintained by a transactional function or an external interface file read by a transaction function.

Assign to each identified EIs, EOs and EQs a functional complexity based on the number of FTRs and DETs [Table 7.3], [Table 7.4], and [Table 7.7].

	1 to 4 DET	**5 to 15 DET**	**16 or more DET**
0 to 1 FTR	Low	Low	Average
2 FTRs	Low	Average	High
3 or more FTRs	Average	High	High

Table 7.3: External Inputs

	1 to 5 DET	**6 to 19 DET**	**20 or more DET**
0 to 1 FTR	Low	Low	Average
2 to 3 FTRs	Low	Average	High
4 or more FTRs	Average	High	High

Table 7.4: External Outputs and External Inquiries

For example, the count of transactional functions and the number of DETs and RETs of our online stock broker are identified in [Table 7.5].

	Type	DET	FTR	Rating
Change Password	EI	4	1	Low
Buy Order	EI	10	5	High
Sell Order	EI	10	5	High
Cancel Order	EI	3	1	Low
Modify Order	EI	6	5	High
Order Received	EI	25	5	High
Process Order	EI	3	2	Low
Margin Call	EI	9	4	High
Get Quote	EO	24	2	High
View Portfolio	EO	10	3	Average
Get Account Balance	EO	7	3	Average
Find Company Symbol	EO	4	2	Low
View Order Details	EQ	6	1	Low
Get Account Activity	EQ	8	2	Average
Get Order Status	EQ	10	2	Average
Login	EQ	4	1	Low

Table 7.5: On-line Stock Broker Transactional Functions Count

Step 5: Determine VAF

The value adjustment factor (VAF) indicates the general functionality provided to the user of the application. The VAF is comprised of 14 general system characteristics that assess the general functionality of the application. [Table 7.6].

Each characteristic has associated descriptions that help determine the degree of influence of the characteristic. The degree of influence range on a scale of zero to five, from no influence to strong influence.

General System Characteristics	
1. Data Communications	2. Distributed Data Processing
3. Performance	4. Heavily Used Configuration
5. Transaction Rate	6. On-line Data Entry
7. End-User Efficiency	8. On-line Update
9. Complex Processing	10. Reusability
11. Installation Ease	12. Operational Ease
13. Multiple Sites	14. Facilitate Change

Table 7.6: General System Characteristics

For example, the Total Degree of Influence (TDI) and VAF of our online stock broker is (formula 2):

TDI = 5 + 5 + 5 + 0 + 5 + 5 + 5 + 5 + 4 + 0 +5 + 1 + 0 + 3 = 48

VAF = (48 *0.01) + 0.65 = 1.13

Step 6: Calculate the Adjusted FP count

Having computed UFP and VAF, FP is easily calculated for our online stock broker using formula 1 and [Table 7.7]. Thus, FP ≅154

Type of Component	Complexity of Components			
	Low	Average	High	Total
External Inputs	3 x 3 = 9	0 x 4 = 0	5 x 6 = 30	39
External Outputs	1 x 4 = 4	2 x 5 = 10	1 x 7 = 7	21
External Inquiries	2 x 3 = 6	2 x 4 = 8	0 x 6 = 0	14
Internal Logical Files	6 x 7 = 42	0 x 10 = 0	0 x 15 = 0	42
External Interface Files	4 x 5 = 20	0 x 7 = 0	0 x 10 = 0	20
	Total Number of Unadjusted Function Points			136
	Multiplied Value Adjustment Factor			1.13
	Total Adjusted Function Points			153.68

Table 7.7: FP Complexity

7.2.3 USE CASE POINTS

In this section, we introduce UCP, we give the definition of UCP measures and we calculate UCP for our case study on-line stockbroker.

7.2.3.1 INTRODUCTION

In 1993, Gustav Karner of Objectory AB did an estimating work for projects based on use cases [KARNER 93]. UCP is based on the work of Albrecht, and it shares most of the benefits and disadvantages of FP. In addition, UCP is preliminary and must be used with other estimating methods such as FP or COCOMO [COCOMOII 01].

Little work has been done on deriving estimation metrics from use cases. However, the interested reader can consult [ARMOUR 96], [FETCKE 97], [SMITH 00], [THOMSON 94].

7.2.3.2 DEFINITION OF MEASURES

UCP is the product of three factors: Unadjusted Use Case Points (UUCP), Technical Complexity Factor (TCF) and Environmental Factor (EF) is calculated in five steps, using the following formulas:

$$UCP = UUCP*TCF*EF \quad (3)$$

Where:

$$UUCP = UAW + UUCW \quad (4)$$

$$TCF = 0.6 + (0.01*TFactor) \quad (5)$$

$$TFactor = \sum TLevel * WeightingFactor \quad (6)$$

$$EF = 1.4 + (-0.03*EFactor) \quad (7)$$

$$EFactor = \sum FLevel * WeightingFactor \quad (8)$$

Step 1: Calculate the Unadjusted Actor Weights (UAW)

The UAW is calculated by counting how many actors of each kind (simple, average, and complex) we have and multiplying each by its weighting factor [Table 7.8]:

Actor Type	Description	Factor
Simple	Program Interface	1
Average	Interactive, or protocol-driven interface	2
Complex	GUI	3

Table 7.8: Actor Weights

For example, in our on-line stock broker case study, we have identified six actors. The ***Customer*** and ***Customer Representative*** actors are complex. The ***Trading System*** actor is average. The ***Security Exchange System*** and ***Accounting System*** actors are simple.

Therefore, UAW = 2*3 + 1*2 + 2*1 = 10.

Step 2: Calculate the Unadjusted Use Case Weights (UUCW)

UUCW is calculated by counting how many use cases (excluding included or extended use cases) of each kind (simple, average, and complex) we have and multiplying each by its weighting factor [Table 7.9]:

Use Case Type	Description	Factor
Simple	**Either** 3 or fewer transactions **Or** Fewer than 5 analysis classes	5
Average	**Either** 4 to 7 transactions **Or** 5 to 10 Analysis Classes	10
Complex	**Either** More than 7 transactions **Or** More than 10 Analysis Classes	15

Table 7.9: Use Case Weights

At this point, the UUCP (formula 4) is calculated.

For example, in our on-line stock broker case study, we identify the following use cases (after excluding included or extended use cases – see note below):

The ***Modify Order, Buy Order, Sell Order, Order Received*** and ***Margin Call*** use cases are all average. The ***Find Company Symbol, View Order Details, Change Password, Get Quote, View Portfolio, Get Account Balance, Get Account Activities, Cancel Order*** and ***Process Order*** use cases are all simple.

Therefore, UUCW = 5*10 + 9*5 = 95 and UUCP = 95 + 10 = 105.

Note that in the high-level use case diagram in Figure 4.4, we have not shown the ***Login*** use case as being included by other use cases (in fact, all of them) for reasons of clarity and simplicity. This is why we excluded it from our count as well as the ***Get Order Status*** use case.

Step 3: Calculate the Technical Complexity Factor (TCF)

TCF (formula 5) is calculated by computing the Technical Factor (TFactor). The TFactor is calculated by assigning a value (TLevel) from 0 to 5 to each technical factor then multiplying it by its weight and then summing all weights to obtain formula 6 [Table 7.10]. A rating of 0 means the factor is irrelevant for the project while 5 means it is essential.

Factor Number	Factor Description	Weight
T1	Distributed System	2
T2	Response or throughput performance objectives	1
T3	End-user efficiency (on-line)	1
T4	Complex internal processing	1
T5	Code must be reusable	1
T6	Easy to install	0.5

T7	Easy to use	0.5
T8	Portable	2
T9	Easy to change	1
T10	Concurrent	1
T11	Includes special security features	1
T12	Provides direct access for third parties	1
T13	Special user-training facilities are required.	1

Table 7.10: Technical Complexity Factor

For example, the technical complexity factor for our on-line stock broker case study is:

TFactor = 5*2 + 5*1 + 5*1 + 4*1 + 0*1 + 5*0.5 + 5*0.5 + 5*2 + 3*1 + 5*1 + 5*1 + 5*1 + 0*1

= 57

TCF = 0.6 + (0.01*57) = 1.17

Step 4: Calculate the Environmental Factor (EF)

EF (formula 7) is calculated by computing the Environmental Factor (Efactor). The EFactor is calculated by assigning a value (FLevel) from 0 to 5 to each environmental factor then multiplying it by its weight and then summing all weights to obtain formula 8 [Table 7.11]. a value of 0 means being the lowest and 5 the highest.

Factor Number	Factor Description	Weight
T1	Familiar with Rational Unified Process	1.5
T2	Application Experience	0.5
T3	Object-oriented experience	1
T4	Lead analyst capability	0.5
T5	Motivation	1
T6	Stable requirements	2
T7	Part-time workers	-1
T8	Difficult programming language	-1

Table 7.11: Environmental Factor

[SCHNEIDER 01] suggests that the estimation probably gives the best results when the team is experienced and everyone is full time on the project. Hence, we will consider that for our on-line stock broker case study when calculating the EF:

EFactor = 4*1.5 + 5*0.5 + 5*1 + 4*0.5 + 4*1 + 3*2 + (-1*0) + (-1*3) = 22.5
EF = 1.4 + (-0.03*22.5) = 0.725

Step 5: Calculate UCP

Having computed UUCP, TCF and EF, UCP is easily calculated using formula 3.

For example, the UCP for our on-line stock broker case study is:
UCP = 105 * 1.17 * 0.725 = 89.07 ≅ 89

7.2.4 ROBUSTNESS ANALYSIS POINTS

In this section, we propose a new metric method, RAP, to estimate the size of a project. We introduce and give the definition of RAP measures and we calculate RAP for our case study on-line stockbroker.

7.2.4.1 INTRODUCTION

Complexity can be defined as 'the degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, the types of data structures, and other system characteristics [IEEE 83].

While the FP complexity metric is based on transactional functions and files to determine functional size, and UCP complexity metric is based on actors and use cases, RAP complexity metric is based on the complexity of the three class types identified during the RA activity:

boundary, control and entity classes. Hence, RAP partitions complexity weights equally among the GUI interface (boundary classes), application code (control classes), and data (entity classes).

RAP is the successor of UCP. It provides more accurate results than UCP because the RA technique refines use cases and produces a first-cut of high-level design. The main difference between RAP and UCP is in the calculation of UUCP, however we use the same TCF and EF tables of UCP to adjust our points because these factors apply to a use-case driven methodology.

RAP uses a small complexity range number (from 0 to 3) when assigning complexity to classes. This resolves some issues related to the calculation of actors and use cases weights in UCP as discussed below:

- Actors' weights could easily mask the outcome of estimation results. An application with many use cases and few actors could be easily under-estimated. On the other hand, an application with few use cases and several GUI actors could be easily over-estimated.

- Use cases' weights are based on either transactions or classes. The issue is the gap in points assigned to the use-case and in counting range. For example, a use case with five analysis classes gets (10 UCP) as much as a use case with ten analysis classes. Hence, there is a wide range in counting and complexity numbers assigned. This could easily over-estimate a project.

7.2.4.2 DEFINITION OF MEASURES

RAP is the product of three factors: Unadjusted Robustness Analysis Points (URAP), Technical Complexity Factor (TCF) and Environmental Factor (EF) is calculated in five steps, using the following formulas:

$$RAP = URAP*TCF*EF \quad (9)$$

Where:

$$URAP = \sum UBW + UCW + UEW \quad (10)$$
$$UBW = \Sigma (BoundaryClass * Factor) \quad (11)$$
$$UCW = \Sigma (ControlClass * Factor) \quad (12)$$
$$UEW = \Sigma (EntityClass * Factor) \quad (13)$$
$$TCF = 0.6 + (0.01*TFactor) \quad (14)$$
$$TFactor = \sum TLevel * WeightingFactor \quad (15)$$
$$EF = 1.4 + (-0.03*EFactor) \quad (16)$$
$$EFactor = \sum FLevel * WeightingFactor \quad (17)$$

Step 1: Calculate the Unadjusted Robustness Analysis Points (URAP)

URAP is calculated by summing the Unadjusted Boundary Weights (UBW), the Unadjusted Control Weights (UCW) and the Unadjusted Entity Weights (UEW) for each RD using formulas 11,12 and 13 and Table 7.12 below:

Class Type	Complexity Type	Description	Factor[1]
Boundary	Neutral	Not a window. We do not count window widgets.	0
	Simple	The user interface (window) is simple and contains standard boundary classes such as textboxes, buttons, lists, etc.	1

[1] Factor weights are based on the definition of UCPs, but fine tunes the UCPs.

Class Type	Complexity Type	Description	Factor[1]
	Average	In addition, the user interface contains advanced boundary classes such as ActiveX controls (TreeView control, grid control, etc.) or Java applets or the user interface is a process.	2
	Complex	In addition, the user interface is highly interactive.	3
Control	Neutral	No influence. The control does not perform any complex computations, data manipulations (validation, saving, updating, deletion, sorting, etc.), or data communications. In addition, the control is not counted if it populates or links boundary classes to entity classes because its entity will be counted.	0
	Simple	The control performs simple computations, or linked to less than three entities but does not perform data communications.	1
	Average	The control performs significant computations, or linked to less than five entities but does not perform data communications.	2
	Complex	The control performs complex computations, or linked to more than four entities, or handles data communications.	3
Entity	Simple	The following three conditions must be true: 1. Not a derived data. 2. Read twice or none. 3. Write once or none.	1
	Average	The following three conditions must be true: 1. Derived data. 2. Multiple reads. 3. Write twice or less.	2
	Complex	The following three conditions must be true: 1. Derived data. 2. Multiple read-write.	3

Table 7.12: Boundary, Control and Entity Weights

Once all RDs have their boundary and entity weight calculated, URAP is calculated using formula 10.

For example, in our on-line stock broker case study, the URAP for the ***Buy Order*** RD [Figure 5.7] is calculated as follows [Table 7.13]:

Class Type	Class Name	Complexity Factor	Justification
Boundary	Market List	0	Not a window.
	Account List	0	Not a window.
	Order web page	1	Simple window.
	Submit Button	0	Not a window.
Control	Pick Market	0	No influence. Its entity has been counted.
	Pick Account	0	No influence. Its entity has been counted.
	Display Error	0	No influence. Messaging display
	Validate Data	2	Validates data from three entities.
	Save Data	1	Saves data to two entities.
Entity	Market	1	Not derived, read once.
	Account	1	Not derived, read twice.
	Stock	1	Not derived, read once.
	Order	1	Not derived, read once, write once.
	TradeLot	1	Not derived, write once.

Table 7.13: URAP for Buy Order RD

Hence, URAP for the Buy Order RD is = 9.

We have calculated the URAP for the on-line stock broker as shown in Table 7.14 below:

RD	UBW	UCW	UEW	Total
Change Password	1	2	1	4
Buy Order	1	3	5	9
Sell Order	1	3	5	9
Cancel Order	1	1	1	3
Modify Order	1	3	5	9
Order Received	2	3	5	10
Process Order	2	1	2	5
Margin Call	2	2	4	8
Get Quote	1	1	2	4
View Portfolio	1	2	3	6
Get Account Balance	1	2	3	6
Find Company Symbol	1	1	2	4
View Order Details	1	1	1	3
Get Account Activity	2	1	2	5
Get Order Status	2	1	2	5
Login	1	1	1	3

Table 7.14: On-line Stock Broker Transactional Functions Count

Hence, URAP for the on-line stock broker case study is = 93

Step 2: Calculate the Technical Complexity Factor (TCF)

TCF (formula 14) is calculated by computing the TFactor (formula 15) as in UCP [Table 7.10]. Therefore, the TCF for our on-line stock broker is = 1.17.

Step 3: Calculate the Environmental Factor (EF)

EF (formula 16) is calculated by computing the Efactor (formula 17) as In UCP [Table 7.11]. Therefore, the EF for our on-line stock broker is = 0.725.

Step 4: Calculate RAP

Having computed URAP, TCF and EF, RAP is calculated using formula 9.

For example, the RAP for our on-line stock broker case study is:

RAP = 93 * 1.17 * 0.725 = 78.89 ≅ 79.

7.3 APPLICATION & ASSESSMENT OF RAP METHODS

In this section, we demonstrate how to convert FP, UCP and RAP into person-month effort. Then, we calculate and compare the size of three applications using the three discussed estimation metrics: FP, UCP, and RAP.

7.3.1 EFFORT PREDICTION

In order to compare the FP, UCP and RAP methods, we have to convert the points into effort. The simplest effort prediction model takes the following form [SHEPPERD 95]:

$$Effort = p*S$$

Where p is a productivity coefficient (1/productivity rate) and S is the size of the software system to be developed. Note that both terms must have the same units, in our case both terms should be either FP or UCP. For example, if we know that the productivity rate is 20 FP per month and the size is 100 then the effort is (1/20)*100 = 5 person months.

To estimate effort for FP, UCP, and RAP, we need to calculate the productivity rate of each metric:

In FP, [CAPERS 96] relates languages to productivity. Table 7.15 shows productivity per person-month for some languages of our interest:

Programming Language	Level	Level Range	Avg. Source Statements per FP	Productivity Average per Person Month
C++	6.00	4 - 8	53	10 to 20
JAVA	6.00	4 – 8	53	10 to 20
ORACLE	8.00	4 – 8	40	10 to 20
Visual Basic 5.0	11.00	9 -15	29	16 to 23
Visual C++	9.50	9 -15	34	16 to 23

Table 7.15: Programming Languages & Productivity

In UCP, [KARNER 93] suggested a factor of 20 person hours per UCP. However, a close examination of his data and based on field experience [SPARKS 99], [SCHNEIDER 01], the suggested effort can range from 15 to 30 person hours per UCP.

In RAP, a close examination of our data suggests the use of the same person hours as per UCP.

7.3.2 CASE STUDIES

Three case studies were considered and studied each having a different level of complexity and criticality. The first case study is the Warehouse Software Portfolio (WSP), a simple application by [FETCKE 99]. The second case study is the National Widget (NW), an average application by [SCHNEIDER 01]. The third case study is our on-line stock broker (OSB), a complex application used in this thesis.

The results are summarized in Table 7.16:

						*Month = 21 days * 8.0 hours = 168 hours*
Application	**Method**	**Unadjusted Points**	**Adjusted Factor**	**Adjusted Points**	**Effort Range**	**Effort Size Range Adjusted (month)**
WSP	FP	77	0.92	71	16-23 FP per month	3.09-4.44
	UCP	92	0.57	52	15-30h per UCP	4.64-9.29
	RAP	83	0.57	47	15-30h per RAP	4.20-8.39
NW	FP	79	0.96	76	16-23 FP per month	3.30-4.75
	UCP	80	0.60	48	15-30h per UCP	4.28-8.57
	RAP	64	0.60	38	15-30h per RAP	3.39-6.78
OSB	FP	136	1.13	154	10-20 FP per month	7.70-15.40
	UCP	105	0.85	89	15-30h per UCP	7.95-15.90
	RAP	93	0.85	79	15-30h per RAP	7.05-14.11

Table 7.16: Case Study Results

Case Study	FP/UCP %	FP/RAP %	RAP/UCP %
WSP	54	60	90
NW	63	79	79
OSB	97	110	89

Table 7.17: Average Effort Ratio

The calculation of the average effort ratio for the three case studies is shown in Table 7.17.

After a close examination of the results, we conclude:

1. Our results are accurate. However, due to the large subjectivity in the calculation, we admit a 15% marginal error mainly contributed to the adjustment factors and to the application type as discussed below.

2. WSP is a small case study. It contains three entities and sixteen transactions. The wide difference between FP and the other two methods, UCP and RAP, is mainly due to the relatively few entities and the ***"LOW"*** complexity rating of the sixteen transactions when computing FP.

3. NW is an average case study. It contains six entities and ten transactions. The less wide difference between FP and the other two methods UCP and RAP is mainly due to the ***"LOW"*** complexity rating of the ten transactions when computing FP.

4. OSB is a case study with complex functionality. It contains six entities and sixteen transactions. The difference between FP and the other two methods UCP and RAP is tight due to an average number of entities and a variety of the complexity rating (***"LOW", "AVERAGE", and HIGH")*** of the sixteen transactions when computing FP.

5. RAP is much closer to UCP than FP because RAP uses UCP adjustment factors, UCP effort size range, and both RAP and UCP are used in a use-case driven methodology. In addition, RAP is often less than UCP and probably more accurate because the RA technique refines use cases and produces a first-cut of a high-level design.

6. Our method is neither better nor worse than the two other methods. Our results show that use case points are over-estimating and that RAP is approximately 90% of UCP. RAP seems to be useful, but needs to be validated for several real projects varying from small, medium to

large size projects. In addition, we suggest using the three estimation methods in the following order: UCP, RAP then FP.

7. Based on our previous conclusions, we also note that FP is more suitable than UCP and RAP when estimating small and average applications.

7.4 SUMMARY

In this chapter, we proposed a new metric to estimate project size based on RA. Although the method is promising, more study should be done to test the suitability of RAP as a software effort-estimation metric. Meanwhile, RAP should be used in conjunction with other estimating methods such as FP or COCOMO.

CHAPTER 8

CONCLUSION

This thesis provides evidence that RA is a useful technique, and evidence to support our proposed RAP metric, however further quantitative research is needed. In this chapter, we review the work reported in the thesis, and we give guidance and directions for future work and research.

8.1 SUMMARY

In this thesis, we gave a complete methodology description of RA, including complete guidelines for performing RA and constructing RDs. We made explicit the construction rules required to build RDs, and we added a new rule to simplify the construction of RDs. Our result can be used as a foundation for explaining and teaching the RA technique.

We assessed and showed the quality of the RA approach by a realistic case study on-line stock broker. We illustrated this Quality Assessment approach by demonstrating how RA can detect three serious errors in requirements: infeasible specifications, class omissions, and design errors. The results of our case study support the use of the methodology for improving robustness of high-level design.

We evaluated the RA technique from a cost-benefit analysis viewpoint. The results are the relevant contributions of this technique to Software Quality and the architecture pattern that results.

We proposed and illustrated a new software effort-estimation technique, RAP that was calculated for three case studies. The results were compared to FP and UCP. However, more study should be done particularly to test the suitability of RAP as a software effort-estimation metric.

We see the primary beneficiaries of our research as including:

- Software Engineers and Analysts who would like to learn how to perform RA.
- Quality Managers who are seeking high quality and robust designs of their software by detecting and resolving design issues at early stages of the development life cycle.
- Project Managers who need to get an early estimate for projects using the RUP or IUP approach.
- Researchers who are interested in pursuing more research like those described in the next section.

8.2 FUTURE DIRECTIONS

Little research has been done on RA although it dates back to 1992 (older than UML) in the form of analysis models. Therefore, many research opportunities are still available.

Further industrial case studies are needed to validate the approach, and to verify that it scales up and is accessible to software designers and engineers. In particular, further studies should assess:

1. ***Ability to estimate and ensure system robustness early in development.*** We believe, for example, it is possible to define a robustness measure based on the complexity and number of classes.

2. ***Ability of RAP to approach the cost of real projects.*** What adjustments should be made to the RAP complexity factors or method? This can be achieved only by calculating the RAP for several real projects varying from small, medium to large size projects.

3. ***Ease and automability of constructing RDs and performing RA.*** We have discussed that sequence diagrams can be partially constructed from RDs using Rose script. How about automating RDs from use cases? Could we write use cases in a technical and programmatic way that enables us to automate RDs?

4. ***Cost and time required to performing RA.*** We certainly would like to know how much time is involved in performing RA and constructing RDs. Was the time spent justified given the benefits of RA?

Most of these questions were raised during our research. We were not able to answer these questions because of necessary constraints on the scope of the work. However, we believe we have established a foundation for such future research.

GLOSSARY OF TERMS

This appendix contains a list of financial glossary of terms for our case study and UML glossary of terms used in this thesis.

A. *Financial Glossary of Terms*

The following list contains financial glossary of terms used in this thesis for our case study [CAMPBELL 01], [CNBC 01].

Term	Description
BROKER	An individual who is paid a commission for executing customer orders. Either a floor broker who executes orders on the floor of the stock exchange, or an upstairs broker who handles retail customers and their orders. In addition, person who acts as an intermediary between a buyer and seller, usually charging a commission.
BUY ORDER	An order to a broker to purchase a specific quantity of a security.
CASH ACCOUNT	A brokerage account that settles transactions on a cash-rather than credit-basis.
DIVIDEND	A portion of a company's profit paid to common and preferred shareholders. A stock selling for $20 a share with an annual dividend of $1 a share yields the investor 5%.
INVESTOR	The owner of a financial asset.
MARGIN	In the stock market, the amount of cash that must be put up in a purchase of securities. If the margin requirement is 50%, the buyer must put up 50% of the purchase price; the buyer must borrow the rest.
MARGIN ACCOUNT	A brokerage account allowing customers to buy securities with money

Term	Description
	borrowed from the brokerage.
MARGIN CALL	A demand upon an investor to put up more collateral for securities bought on credit. The lender, usually the brokerage firm, makes the call when the equity in the investor's account falls below the level set by the brokerage.
MUTUAL FUND	A fund that pools the money of its investors to buy a variety of securities. Open-end mutual funds sell as many shares as investors want. Closed-end mutual funds offer only a fixed number of shares and usually trade on an exchange.
OPTION	An agreement allowing an investor to buy or sell something, such as shares of stock, within a stipulated time and for a certain price. Also, it is a method of employee compensation that gives workers the right to buy the company's stock during a specified period of time at a stipulated exercise price.
PORTFOLIO	A collection of securities held by an investor.
QUOTE	A bid to buy a security or an offer to sell a security in a given market at a given time.
SECURITY	A financial instrument that indicates the holder owns a share or shares of a company (stock) or has loaned money to a company or government organization (bond).
SELL ORDER	An order to a broker to sell a specific quantity of a security.
SHARE	An investment that represents part ownership of a company or a mutual fund.
STOCK	An investment that represents part ownership of a company. There are two different types of stock: common and preferred. Common stocks provide voting rights but no guarantee of dividend payments. Preferred stocks provide no voting rights but have a set, guaranteed dividend payment. Also called shares.

Term	Description
STOCK EXCHANGES	Formal organizations, approved and regulated by the Securities and Exchange Commission (SEC), that are made up of members who use the facilities to exchange certain common stocks.
STOCK MARKET	Also called the equity market, the market for trading equities.
STOCK SYMBOL	An abbreviation assigned to a security for trading purposes.
TRADERS	Individuals who take positions in securities and their derivatives with the objective of making profits.
TRADING	Buying and Selling securities.
TRADING PRICE	The price at which a security is currently selling.

B. UML Glossary of Terms

The following list contains UML Glossary terms used in this thesis. A full list of glossary terms can be found in the UML documents.

Term	Description
ACTIVITY	Behavior that occurs while in a state. An activity can be interrupted by a transition event.
ACTOR	Someone or something external to the system that must interact with the system under development.
AGGREGATION	A stronger form of an association where the relationship is between a whole and its part(s).
ASSOCIATION	A bi-directional, semantic connection between two classes.
ASSOCIATION CLASS	A class that holds information belonging to a link between two objects, not with one object by itself.
ATTRIBUTE	A data definition help by objects of a class. The structure of the class.
CLASS	A description of a group of objects with common properties (attributes), common behavior (operations), common relationships to other objects (associations and aggregations), and common semantics.
CLASS DIAGRAM	A view or picture of some or all of the classes in a model.
COLLABORATION DIAGRAM	A diagram that shows object interactions organized around the objects and their links to each other.
COMPONENT DIAGRAM	A diagram that shows the organization and dependencies among software components, including source code components, run-time components, and executable components.
DESIGN	How the system will be realized in the implementation phase.
ELABORATION	Planning the necessary activities and required resources; specifying the features and designing the architecture.
GENERALIZATION	The process used to create superclasses that encapsulate structure and behavior common to several classes.
IMPLEMENTATION	The production of the code that will result in an executable system.
INCEPTION	The specification of the project vision.
INHERITANCE	A relationship among classes where one class shares the structure and/or behavior defined in one or more other classes.
MODEL	An abstraction that portrays the essentials of a complex problem or structure, making it easier to manipulate.
OBJECT	A concept, abstraction, or thing with sharp boundaries and meaning for an application.
SCENARIO	An instance of a use case-it is one path through the flow of events for

Term	Description
	the use case.
SEQUENCE DIAGRAM	A diagram that depicts object interactions arranged in time sequence.
STEREOTYPE	A new type of modeling element that extends the metamodel. Stereotypes must be based on elements that are part of the UML metamodel.
SUBCLASS	A class that inherits from one or more classes.
SUPERCLASS	The class from which another class inherits.
TEST	The verification of the entire system.
UNIFIED MODELING LANGUAGE	A language used to specify, visualize, and document the artifacts of an object-oriented system under development.
USE CASE	Representation of the business processes of the system. The model of a dialogue between an actor and the system.
USE CASE DIAGRAM	A graphical representation of some or all of the actors, use cases, and their interactions.
USE-CASE MODEL	The collection of actors, use cases, and use case diagrams for a system.

REFERENCES

1. [ALBRECHT 79] A. J. Albrecht, Measuring Application Development Productivity. IBM Applications Development Symposium, 1979.

2. [ALHIR 98] Sinan Si Alhir, UML in a Nutshell. O' Reilly & Associates Inc., 1998.

3. [AMBLER 98] Scott Ambler. The UML and Beyond. www.ambysoft.com/umlAndBeyond.html, 1998.

4. [ARMOUR 96] F. Armour, B. Catherwood, et. Al, Experiences Measuring Object Oriented System Size with Use Case, Proc. ESCOM, Wilmslow, UK, 1996.

5. [ASHLEY 95] Nicholas Ashley. Measurement as a Powerful Software Management Tool, McGraw-Hill Book Company, 1995.

6. [ASPIN 99] Jim Aspin, StockRouter NewsLetter, Do Local Online Brokers Have a Chance?, Volume 1, September 1999.

7. [BECKER 01] Sirley A. Becker and Florence E. Mottay, A Global Perspective on Web Site Usability, Usability Engineering, IEEE Software, January 2001.

8. [BEN HAJLA 97] Halim Ben Hajla, Traceability in Object-Oriented Quality Engineering, Master of Computer Science, University of Ottawa, October 1997.

9. [BINDER 99] Robert V. Binder. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison Wesley, 1999.

10. [BLAHA 98] Michael Blaha and William Premerlani. Object-Oriented Modeling and Design for Database Applications. Prentice Hall, 1998.

11. [BOOCH 94] Grady Booch, Object-Oriented Analysis and Design with Applications, Second Edition. The Benjamin/Cummings Publishing Company, 1994.

12. [CAMPBELL 01] Campbell R. Harvey's Hypertextual Finance Glossary, January 2001, http://www.duke.edu/~charvey/Classes/wpg/glossary.htm.

13. [CANTOR 98] Murray Cantor, Object-Oriented Project Management With UML, John Wiley & Sons, Inc., 1998.

14. [CAPERS 96] Jones Capers, Software Productivity Research Inc., Programming Languages Table, March 1996.

15. [CNBC 01] CNBC.com Finance Glossary, 2001, http://www.cnbc.com/datatools/education/glossaryterms/a.html.

16. [COCOMOII 01], Center for Software Engineering, http://sunset.usc.edu/research/COCOMOII.

17. [CONALLEN 00] Jim Conallen. Building Web Applications With UML, Addison-Wesley, 2000.

18. [CONTE 86] S. D. Conte et al, Software Engineering Metrics and Models, Benjamin / Cummings Publ. Co., 1986.

19. [COSMIC 99] COSMIC-FFP Measurement Manual. Common Software Measurement International Consortium. Version 2.0, 1999.

20. [DBMS 96] DBMS Interview - October 1996, Rational Software's Grady Booch, Ivar Jacobson, and Jim Rumbaugh, www.dbmsmag.com/9610d11.html.

21. [DEMARCO 82] Tom DeMarco. Controlling Software Projects: Management Measurement & Estimation, Yourdon Press, 1982.

22. [DEMARCO 90] Tom DeMarco and Tomothy Lister. Editor'sNote: Software State-of-the-Art - Selected Papers, Dorset House Publishing, 1990.

23. [FETCKE 97] Thomas Fetcke, A. Abran, et al., Mapping the OO-Jacobson Approach into Function Point Analysis, Proc. TOOLS USA 97, Santa Barbara, California, 1997.

24. [FETCKE 99] Thomas Fetcke, The Warehouse Software Portfolio: A Case Study in Functional Size, 1999.

25. [FOWLER 97A] Martin Fowler and Kendall Scott, UML Distilled. Addison-Wesley, 1997.

26. [FOWLER 97B] Martin Fowler, Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997.

27. [GDPRO 01] GDPRO Home Page, www.gdpro.com.

28. [ICONIX 01] ICONIX Home Page, www.iconixsw.com.

29. [IEEE 83] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std729, 1983.

30. [IFPUG 01] Function Point Counting Practices Manual. International Function Point Users Group, Westerville, Ohio. Release 4.1.1, 2001.

31. [JACOBSON 92] Ivar Jacobson, et al. Object-Oriented Software Engineering: A Use Case Driven Approach. Revised Fourth Printing. Addison-Wesley, 1992-1994.

32. [JACOBSON 95] Ivar Jacobson, Maria Ericsson, Agneta Jacobson. The Object Advantage: Business Process Reengineering with Object Technology. ACM Press, 1995.

33. [JACOBSON 99A] Ivar Jacobson, Grady Booch, and Jim Rumbaugh. The Unified Software Development Process, Addison-Wesley, 1999.

34. [JACOBSON 99B] Ivar Jacobson, Grady Booch, and Jim Rumbaugh. The Unified Modeling Language User Guide, Addison-Wesley, 1999.

35. [JACOBSON 99C] Ivar Jacobson, Grady Booch, and Jim Rumbaugh. The Unified Modeling Language Reference Manual, Addison-Wesley, 1999.

36. [KAN 95] Stephen H. Kan, Metrics and Models in Software Quality Engineering, Addison Wesley, 1995

37. [KARNER 93] Gustav Karner, Use Case Points - Resource Estimation for Objectory Projects, Objective Systems SF AB (copyright owned by Rational Software), 1993

38. [KRUCHTEN 98] Philippe Kruchten, The Rational Unified Process: An Introduction, Addison Wesley, 1998

39. [LARMAN 97] Craig Larman. Applying UML and Patterns: an Introduction to Object Oriented Analysis and Design, Prentice Hall, 1997.

40. [LORENZ 94] Mark Lorenz, Jeff Kidd. Object-Oriented Software Metrics, Prentice Hall, 1994.

41. [MANNING 98] H. Manning, "Why Most Web Sites Fail", Forrester Research, www.forrester.com, Sept. 1998.

42. [MCCALL 77] A. McCall, P. K. Richards and G.F. Walters, Factors in Software Quality, RADC-TR—77-369, US Department of Commerce, 1977.

43. [MEYER 97] Bertrand Meyer, OOSC-2: How to Find the Classes, www.eiffel.com/doc/manuals/technology/oosc/finding/page.html

44. [MICROSOFT 01] Microsoft Corporation, MSDN library. http://msdn.microsoft.com.

45. [NUA 00] Nua Internet Surveys, www.nua.ie/surveys/how_many_online/index.html.

46. [OESTEREICH 99] Bernd Oestereigh. Developing Software With UML: Object-Oriented Analysis and Design in Practice, Addison-Wesley, 1999.

47. [OMG] Object Management Group Home Page, www.omg.com.

48. [OMG 95] Object Management Group, The common Object Request Broker: Architecture and Specification; Revision 2.0. Framingham, Massachussetts, U.S.A, 1995.

49. [POOLEY 98] Rob Pooley, and Perdita Stevens. Using UML, Software Engineering With Object And Components, Addison-Wesley, 1999.

50. [PRESSMAN 97] Roger S. Pressman. Software Engineering: A practitioner's Approach, 4th Edition, McGraw-Hill, 1997.

51. [QSM 01], Quantitative Software Measurement, www.qsm.com.

52. [QUATRANI 98] Terry Quatrani. Visual Modeling With Rational Rose and UML, Addison-Wesley, 4th Printing 1998.

53. [RATIONAL 01A] Rational Software Home Page, www.rational.com.

54. [RATIONAL 00] Rational Software–Whitepapers, Rational Unified Process: Best Practices for Software Development Teams, www.rational.com/products/whitepapers/100420.jsp

55. [RATIONAL 01B] Rational Software, Unified Modeling Language Resource Center, UML Documentation version 1.3, www.rational.com/uml/index.jtmpl.

56. [RICHTER 99] Charles Richter. Building Flexible Object-Oriented Systems With UML. Macmillan Technical Publishing, 1999.

57. [ROSENBERG 99] Doug Rosenberg and Kendall Scott. Use Case Driven Object Modeling With UML, Addison-Wesley, 1999.

58. [ROSENBERG 00] Doug Rosenberg and Kendall Scott Software Development Magazine: Driving Design with Use Cases, December 2000, www.sdmagazine.com/articles/2000/0012/0012c/0012c.htm.

59. [ROYCE 98] Walker Royce. Software Project Management: A Unified Framework, Addison-Wesley, 1998.

60. [RUMBAUGH 91] James Rumbaugh, Michaeal Blaha, William Premerlani, Frederick Eddy, William Lorensen. Object-Oriented Modeling and Design. Prentice Hall, 1991.

61. [SCHACH 98] Stephen R. Schach. Classical and Object-Oriented Software Engineering WITH UML and C++, McGraw-Hill, 1998.

62. [SCHNEIDER 01] Geri Schneider, Jason Winters. Applying Use Cases, Second Edition Addison-Wesley, 2001.

63. [SHEPPERD 93] Martin Shepperd. Software Engineering Metrics Volume I: Measures and Validations, McGraw-Hill Book Company, 1993.

64. [SHEPPERD 95] Martin Shepperd. Foundations of Software Measurement, Prentice Hall, 1995.

65. [SMITH 00] John Smith, The Estimation of Effort Based on Use Cases, Rational Software White Paper, 2000.

66. [SOMMERVILLE 96] Ian Sommerville. Software Engineering, Addison-Wesley, 1996.

67. [SPARKS 99] Steve Sparks and Kara Kapczynski, The Art of Sizing Projects, SunWorld, December 1999. www.sunworld.com/sunworldonline/swol-12-1999/swol-12-itarchitect-p.html

68. [ST-PIERRE 97] Denis St-Pierre et al, Full Function Points: Counting practices manual, Software Engineering Management Research Laboratory, Université du Québec á Montréal, and Software Engineering Laboratory in Applied Metrics, 1997.

69. [TDWATERHOUSE], TD Waterhouse Investor Services (Canada) inc., a subsidiary of TD Bank, www.tdwaterhouse.ca.

70. [TEXEL 97] Putnam P. Texel and Charles B. Williams. Use Cases Combined With Booch OMT UML, Prentice Hall, 1997.

71. [THOMSON 94] N. Thomson, R. Johnson et al., Project Estimation Using an Adaptation of Function Points and Use Cases for OO Projects, Proc. Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA '95, 1994

72. [TREESE 98] G. Winfield Treese, Lawrence C. Stewart. Designing Systems for Internet Commerce, Addison-Wesley, 1998.

73. [UKSMA 98] Mk II Function Point Analysis Counting Practices Manual. United Kingdom Software Metrics Association. Version 1.3.1, 1998.

74. [VLIET 96] Hans Van Vliet, Software Engineering: Principles and Practice. John Wiley & Sons, 1996.