National Library
of Canada

Acquisitions and
Bibliographic Services Branch

385 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

385, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Towards Unifying Design and Implementation in OO Systems

by

Khaled Noaman, B.Sc.

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science**

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

January 16, 1995

© Copyright

1995, Khaled Noaman

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Votre file - Votre référence

Votre file - Votre référence

**THE AUTHOR HAS GRANTED AN
IRREVOCABLE NON-EXCLUSIVE
LICENCE ALLOWING THE NATIONAL
LIBRARY OF CANADA TO
REPRODUCE, LOAN, DISTRIBUTE OR
SELL COPIES OF HIS/HER THESIS BY
ANY MEANS AND IN ANY FORM OR
FORMAT, MAKING THIS THESIS
AVAILABLE TO INTERESTED
PERSONS.**

**L'AUTEUR A ACCORDE UNE LICENCE
IRREVOCABLE ET NON EXCLUSIVE
PERMETTANT A LA BIBLIOTHEQUE
NATIONALE DU CANADA DE
REPRODUIRE, PRETER, DISTRIBUER
OU VENDRE DES COPIES DE SA
THESE DE QUELQUE MANIERE ET
SOUS QUELQUE FORME QUE CE SOIT
POUR METTRE DES EXEMPLAIRES DE
CETTE THESE A LA DISPOSITION DES
PERSONNE INTERESSEES.**

**THE AUTHOR RETAINS OWNERSHIP
OF THE COPYRIGHT IN HIS/HER
THESIS. NEITHER THE THESIS NOR
SUBSTANTIAL EXTRACTS FROM IT
MAY BE PRINTED OR OTHERWISE
REPRODUCED WITHOUT HIS/HER
PERMISSION.**

**L'AUTEUR CONSERVE LA PROPRIETE
DU DROIT D'AUTEUR QUI PROTEGE
SA THESE. NI LA THESE NI DES
EXTRAITS SUBSTANTIELS DE CELLE-
CI NE DOIVENT ETRE IMPRIMES OU
AUTREMENT REPRODUITS SANS SON
AUTORISATION.**

ISBN 0-612-03006-7

Canada

Name Khaled Noman

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science

SUBJECT TERM

04987

U·M·I

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern
Literature 0401
General 0294
Classical 0295
Comparative 0297
Medieval 0298
Modern 0316
African 0591
American 0305
Asian 0352
Canadian (English) 0355
Canadian (French) 0593
English 0311
Germanic 0312
Latin American 0315
Middle Eastern 0313
Romance 0314
Slavic and East European

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
General 0318
Biblical Studies 0321
Clergy 0319
History of 0320
Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
Archaeology 0324
Cultural 0326
Physical 0327
Business Administration
General 0310
Accounting 0272
Banking 0770
Management 0454
Marketing 0338
Canadian Studies 0385
Economics
General 0501
Agricultural 0503
Commerce-Business 0505
Finance 0508
History 0509
Labor 0510
Theory 0511
Folklore 0358
Geography 0366
Gerontology 0351
History
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
General 0473
Agronomy 0285
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0777
Wood Technology 0746
Biology
General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0379
Ecology 0329
Entomology 0353
Genetics 0369
Limnology 0793
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oceanography 0416
Physiology 0433
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0992
Dentistry 0567
Education 0350
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Therapy 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
Chemistry
General 0485
Agricultural 0749
Analytical 0486
Biochemistry 0487
Inorganic 0488
Nuclear 0738
Organic 0490
Pharmaceutical 0491
Physical 0494
Polymer 0495
Radiation 0754
Mathematics 0405
Physics
General 0605
Acoustics 0986
Astronomy and Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463
Applied Sciences
Applied Mechanics 0346
Computer Science 0984

Engineering
General 0537
Aerospace 0538
Agricultural 0539
Automotive 0540
Biomedical 0541
Chemical 0542
Civil 0543
Electronics and Electrical 0544
Heat and Thermodynamics 0348
Hydraulic 0545
Industrial 0546
Marine 0547
Materials Science 0794
Mechanical 0548
Metallurgy 0743
Mining 0551
Nuclear 0552
Packaging 0549
Petroleum 0765
Sanitary and Municipal 0554
System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

**The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis
Towards Unifying Design and Implementation in OO Systems
submitted by
Khaled Noaman
in partial fulfillment of the requirements
for the degree of Master of Computer Science**

Director, School of Computer Science

Thesis Supervisor

Carleton University

January 16, 1995

Abstract

The evolution of OOP in the late 80's has led to the development of various OO design and analysis methodologies and tools that support these methodologies. However, many of these tools are based on supporting the top-down style of development associated with the traditional waterfall life cycle. The iterative, incremental object-oriented life cycle more easily accommodates change, encourages reuse, recognizes that the analysis, design and implementation phases are necessarily less distinct, and recognizes that development consists of many analysis, design, and implementation cycles. Moreover, these activities are bi-directional with reanalysis and redesign often being driven by changes at the implementation level. Supporting tools and environments must therefore be bi-directional to ensure synchronization between analysis, design and implementation documentation.

In this thesis, we describe the design and implementation of COIN, a tool for Smalltalk programmers that provides support for both top-down and bottom-up development. COIN aims to move towards the goal of unifying design and implementation by providing facilities for creating interaction and collaboration diagrams, automatically generating Smalltalk code, and by providing automatic diagram update capabilities to cope with modifications in the Smalltalk source code.

Acknowledgments

First of all, I thank God for helping me throughout the course of my studies.

I would like to thank Professor John Pugh for his guidance, encouragement, and constructive criticism throughout this thesis.

The research was supported by COOP (Center of Object-Oriented Programming). I am indebted to them for their assistance.

My good friends Amir Zeid, Abdel-Salam Shanab, Mohamed Elammari, and Jelber Sayyad were particularly supportive during difficult times. Thanks for your moral support and encouragement.

Finally, and most especially, my thanks to my mother and my sister for their love and patience.

Table of Contents

Abstract.....	iii
Acknowledgments.....	iv
List of Figures.....	ix
1 Introduction.....	1
1.1 Object-Oriented Technology.....	1
1.1.1 What is Object-Oriented?.....	1
1.1.2 Benefits	2
1.1.3 Characteristics	3
1.1.4 Software Development	3
1.2 Motivation	4
1.3 COIN.....	8
1.4 Thesis Organization	9
2 Related Research	10
2.1 Background	10
2.1.1 Use Case Model.....	11
2.1.1.1 Actors	12
2.1.1.2 Use Cases.....	12
2.1.1.3 Summary.....	13
2.1.2 Representations of Message Sending Dialogue	13
2.1.2.1 Event Trace Diagrams (Rumbaugh)	13
2.1.2.2 Object Diagrams (Booch).....	14
2.1.2.3 Cunningham Diagrams (Cunningham)	15
2.1.2.4 Interaction Diagrams (Jacobson)	17
2.1.2.4.1 Components	17
2.1.2.4.2 Structure.....	19
2.1.2.4.2.1 Centralized.....	19

2.1.2.4.2.2 Decentralized	20
2.1.2.4.3 Summary	22
2.1.2.5 Summary	22
2.1.3 Collaborations Graph	22
2.1.4 CRC Cards	24
2.1.5 Summary	25
2.2 Related Work	26
2.2.1 Program Visualization	26
2.2.1.1 Cara: An Environment For Interactive Design Of Communications Architectures	27
2.2.1.2 KAESTLE	28
2.2.1.3 Interaction Diagram Generator (IDG)	30
2.2.2 Objectory	31
2.2.3 Rational Rose C++	33
2.2.4 Summary	36
3 Coin Tool	38
3.1 Development Goals	38
3.2 Features	40
3.2.1 Interaction Diagram Editor	40
3.2.2 CRC Card Manager	41
3.2.3 Object Collaborations	41
3.2.4 Code Generation	42
3.2.5 Diagram Modification	42
3.3 Applications	43
3.4 Summary	44
4 An Overview Of Coin	46
4.1 General View	46
4.2 Interaction Diagram Editor View	48
4.3 Object Collaborations View	53
4.4 CRC Card View	56
4.5 Code Generation View	58
4.6 Method Information View	61

4.7 Summary	65
5 The Design of Coin	67
5.1 Design Principles	67
5.2 Design Issues and Choices	69
5.2.1 Diagram Editor	69
5.2.2 Information Extraction and Presentation	71
5.2.3 Code Analysis	73
5.2.4 Automatic Diagram Update	74
5.2.5 Summary	76
5.3 Coin Classes	77
5.3.1 New Classes	78
5.3.1.1 IDComponent	78
5.3.1.1.1 Border	79
5.3.1.1.2 Operation	80
5.3.1.1.3 SystemBorder	80
5.3.1.1.4 Stimuli	80
5.3.1.1.4.1 IDMessage	81
5.3.1.1.4.2 IDSelfMessage	81
5.3.1.1.4.3 Signal	81
5.3.1.2 Tool	81
5.3.1.2.1 BorderTool	83
5.3.1.2.2 OperationTool	83
5.3.1.2.3 StimuliTool	84
5.3.1.2.3.1 MessageTool	84
5.3.1.2.3.2 SignalTool	84
5.3.1.2.4 SelectionTool	85
5.3.1.3. DiagramEditor	85
5.3.1.4 ObjectCollaborations	86
5.3.1.5 ConsistencyChecker	86
5.3.1.6 StaticAnalyzer	88
5.3.1.7 DiagramModifier	88
5.3.1.8 MethodInfo	89
5.3.1.9 ObjectInfo	90
5.3.1.10 IDDictionary	91
5.3.1.11 CodeGenerator	92

5.3.1.12 CRCManager	93
5.3.1.13 CRCCard.....	94
5.3.1.14. ColorsPalette	95
5.3.2 Modified Classes	95
5.3.2.1 ClassHierarchyBrowser	95
5.3.2.2 Stream.....	96
 6. Sample Application: An Airline Reservation System.....	97
6.1 An Airline Reservation System	97
6.2 Analysis and Design	97
6.2.1 Initializing an Airline Reservation System	100
6.2.2 Booking a Flight for a Passenger	114
6.3 Summary	118
 7 Conclusions and Future Work	120
7.1 Conclusions	120
7.2 Future Work	123
 Bibliography	125

List of Figures

Fig 2.1 Use Case Model	11
Fig 2.2 Event Trace Diagram	14
Fig 2.3 Object Diagram	15
Fig 2.4 Cunningham Diagram	16
Fig 2.5 Interaction Diagram.....	18
Fig 2.6 Fork Diagram.....	19
Fig 2.7 Stair Diagram.....	20
Fig 2.8 Collaborations Graph	23
Fig 2.9 CRC Card.....	25
Fig 2.10 KAESTLE.....	29
Fig 2.11 Interaction Diagram Generation using IDG	31
Fig 2.12 Obj tory's interaction diagram editor	33
Fig 2.13 Header file generated by Rose/C++ generator.....	34
Fig 2.14 Implementation file generated by Rose/C++	35
Fig 3.1 Coin Architecture.....	39
Fig 3.2 An interaction diagram.....	42
Fig 3.3 Coin applications in object-oriented life cycle	44
Fig 4.1 General View	47
Fig 4.2 Editor View.....	49
Fig 4.3 Object creation	50
Fig 4.4 Irregularities detected by the overall consistency checker	51
Fig 4.5 Interaction Diagram with duplicate messages	52
Fig 4.6 Renaming a message	53
Fig 4.7 Extracting the users from the helpers.....	54
Fig 4.8 Objects Collaborations	55
Fig 4.9 Card Manager.....	57
Fig 4.10 Code generation view.....	59
Fig 4.11 Object Information	60

Fig 4.12 Method Information View	61
Fig 4.13 Map Window.....	63
Fig 4.14 Mapped Message	64
Fig 4.15 Automatic diagram update.....	66
Fig 5.1 Overview of Coin.....	68
Fig 5.2 Overview of Coin's editor activities.....	70
Fig 5.3 A case of translating diagram components.....	77
Fig 5.4 IDComponent Hierarchy	79
Fig 5.5 Tool Hierarchy	82
Fig 5.6 Tool activation collaborations graph.....	83
Fig 5.7 On-line Consistency Checking.....	87
Fig 5.8 StaticAnalyzer Collaborations Graph.. ..	88
Fig 5.9 MethodInfo Objects Ensemble	90
Fig 5.10 ObjectInfo Objects Ensemble	91
Fig 5.11 CodeGenerator Collaborations Graph	93
Fig 5.12 CRCManger Collaborations Graph	94
Fig 5.13 CRCCard Objects Ensemble.....	94
Fig 6.1 Airline reservation system with interacting actors.....	98
Fig 6.2 Airline reservation system use case model.....	99
Fig 6.3 Initializing an airline reservation system interaction diagram.....	102
Fig 6.4 Printout of interaction diagram.....	103
Fig 6.5 Flight collaboration information	104
Fig 6.6 AirlineReservation CRC card	105
Fig 6.7 Creating Airline class	106
Fig 6.8 Airline class browser	107
Fig 6.9 Inconsistent design and implementation	109
Fig 6.10 Consistent design and implementation.....	110
Fig 6.11 New collaboration information of class Flight	111
Fig 6.12 Code information of addAirline: in AirlineReservation	112
Fig 6.13 Mapping message receivers	113
Fig 6.14 Updating interaction diagram using selected messages	114
Fig 6.15 Partial interaction diagram for booking a passenger on a flight.....	116

Fig 6.16 Implementation information of book:airline:flight:date:	117
Fig 6.17 Interaction diagram for booking a flight	118

Chapter 1

Introduction

In this chapter we define object-oriented technology, discuss the motivation for our work, and give an overview of the organization of the thesis.

1.1 Object-Oriented Technology

Software is inherently complex; the complexity of today's software systems often exceeds the human intellectual capacity. The task of software development is to simplify such complexity. However, human cognition is limited in dealing with complex systems as a whole, so humans cope through the use of decomposition, abstraction, and hierarchy. Object-Oriented technology is considered by many people a promising new approach to controlling software complexity.

1.1.1 What is Object-Oriented?

"The object-oriented approach attempts to manage the complexity inherent in real-world problems by abstracting out knowledge, and encapsulating it within objects" [Wirfs-Brock 90]. In object-oriented technology, a system is modeled as a set of interacting objects; an object is an entity that combines both data (or state) and behavior. Object-Oriented technology has become very popular in recent years due to

the many benefits it promises to provide, and, in particular, its approach to managing complexity and encouraging reuse of software components.

1.1.2 Benefits

Many people have high hopes for the object-oriented paradigm. This trust is based on the benefits that object-oriented technology promises to provide. These benefits include:

- ***Reusability:*** By reducing the interdependency among software components, object-oriented technology permits the development of reusable software components. By building class libraries and using the notion of frameworks, previously built components can be used in the development of new software systems.
- ***Faster development:*** The ability to use reusable components will reduce development time, as we will eliminate the time and effort needed to design, build and test the reused components.
- ***Increased productivity:*** Reusing components produces software that is more reliable and easier to update, since fewer iterations are needed, and revisions are more localized.
- ***Extensibility:*** As the interdependencies among software components are reduced, and through the use of encapsulation, information hiding, and inheritance we can extend our software systems more easily. We can modify existing behavior without affecting the rest of the system.

1.1.3 Characteristics

There are four characteristics that need to be present in an object-oriented system [Rumbaugh 91]:

- **Abstraction:** denotes the essential characteristics of an object that distinguish it from other objects. Abstraction focuses on the view of an object as seen from the outside.
- **Encapsulation:** is the process of hiding the details of an object that do not contribute to its essential characteristics. Abstraction and Encapsulation are complementary concepts. Abstraction focuses on the outside view of an object, while encapsulation prevents clients from seeing the inside view of an object.
- **Polymorphism:** comes from the two Greek words poly (many), and morphos (form)-multiform. In object-oriented systems, polymorphism permits many objects to respond to the 'same message' each in their own way. Polymorphism reduces the name space in large systems and promotes the generation of reusable, generic code.
- **Inheritance:** defines a relationship among classes, where attributes and behavior are shared among classes based on a hierarchical inheritance structure. Such code sharing can often reduce the amount of code by 40% when compared with traditional software systems.

1.1.4 Software Development

Object-Oriented development is a way of thinking about software, based on abstractions that exist in the real world [Rumbaugh 92]. The activities involved in object-oriented development include:

- **Discovering or defining the different objects of the system**
- **Factoring object behavior**
- **Identifying relationships between objects**
- **Specifying message protocols**
- **Implementing objects and their relationships**

Object-Oriented development is most often practiced using an iterative approach. Using iterative development, the software is integrated early and often at each iteration instead of at the end of the project. Frequent integration dramatically reduces risk by exposing problems early in the project life cycle.

Object-Oriented Analysis is a method that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.

Object-Oriented Design is a method encompassing the process of object-oriented decomposition and a notation of depicting both logical and physical as well as static and dynamic models of the system under design.

Object-Oriented Programming is the method of implementation in which programs are organized as cooperative collections of objects, and whose classes are all members of a hierarchy of classes united via inheritance relationship. There are many different object-oriented languages in the market. They are divided into two categories: hybrid (e.g., C++, ObjectiveC), and pure (e.g., Smalltalk, Eiffel).

1.2 Motivation

The evolution of Object-Oriented Programming (OOP) systems in the early 80's created a revolution in software development by emphasizing inheritance, reuse,

and interactive environments. The object-oriented paradigm favors an iterative life cycle where in each iteration a series of deliverables is generated, where each deliverable is a refinement of a previous working one (involving analysis, design, and implementation) and ultimately leading to the final product.

In the object-oriented paradigm, the system is modeled as a set of interacting objects. In order to obtain a picture of how the objects fit into the system, we need to describe the different scenarios in which objects communicate with each others. An **interaction diagram**, which originated from the notion of use cases introduced by Jacobson, is a graphical notation used to describe the message flow between the different objects. There are other notations used to describe the message sending dialogue between objects such as: object diagrams [Booch 94], event trace diagrams [Rumbaugh 91], and Cunningham diagrams [Cunningham 86]. Interaction diagrams are very useful because:

- They depict the message sending dialogue between objects in a system, which gives the user the ability to understand the system under development.
- Useful information can be extracted from them, and represented in other forms such as object collaborations.
- They can be used to determine the "structure" of a system (e.g., centralized, decentralized)
- They can be used for testing the final system.
- They can be used for code generation.

Many new methodologies have emerged to support object-oriented software development (OO analysis, OO design, OO implementation). Some of the more

popular object-oriented analysis and design methodologies are Object Modeling Technique (OMT) [Rumbaugh 91], Responsibility-Driven Design [Wirfs-Brock 90], Object-Oriented Software Engineering (OOSE) [Jacobson 93], Object-Oriented Analysis (OOA) [Coad 91], and Object-Oriented Design (OOD) [Booch 94]. Many tools, such as Objectory, OMT, and Rational Rose, have been developed to support particular methodologies. The majority of these tools support the life cycle in one direction only; i.e., top-down: from analysis to design to implementation.

Unfortunately, there are only few tools (Synchronicity, Together/C++) that provide any support at all for the integration of changes in the bottom-up direction. In OOP systems, it is very difficult to separate design from programming since in the implementation level libraries are accessed and browsed on-line, while in design the programming libraries are taken into account to maximize reuse. In other words, library evolution affects design choices and successful designs get incorporated into the libraries.

In order to integrate analysis, design, and implementation we need to develop an environment that supports bi-directional life cycle development where design is used to drive implementation and implementation is used to drive design. Bottom-up tools are very important since reuse is an important aspect of OOP systems. Our aim was to develop a tool that would provide support for bi-directional development facilities:

Top-Down

- Build interaction diagrams.
- Extract useful information from the interaction diagrams and represent that information to the user in other forms (e.g., collaborations, CRC cards).

- Facilitate the coding process by generating Smalltalk code constructs based on the information embedded in the interaction diagram

Bottom-Up

- Update interaction diagrams automatically to reflect any changes in the code based on a comparison between the messages in the design (interaction diagram) and the implementation (source code).
- Update information in the collaboration diagrams and CRC cards.

When comparing interaction diagrams and source code, we can analyze the code to extract the different messages and their receivers. Such an analysis of the source code can be very useful in software maintenance and reengineering to satisfy goals such as:

- Automated visualization of dynamic behavior to facilitate code sharing and reusability.
- Checking source code for conformance with coding standards.
- Design recovery, a subset of reverse engineering in which domain knowledge, external information and deduction are added to the observation of the subject system to identify useful high level abstractions beyond those obtained directly by examining the system.
- Automated documentation including automated documentation of programs in natural language.
- Extraction of graphical representations, such as structured charts and message flow dialog diagrams, from the analysis of the code.

In this thesis, we describe a tool (named **Coin**) which attempts to make progress towards satisfying the goals above. As our focus is on comparing design with implementation, we considered interaction diagrams and the source code as the heads and tails of a coin, leading us to call our system **Coin**.

1.3 Coin

Coin has been developed to assist designers and programmers using Smalltalk/V for Windows. It provides the user with:

- The ability to manually construct interaction diagrams using an interaction diagram editor.
- Implementation information in the form of source code and analysis information (extracted messages and their designated receivers).
- An automatic diagram update facility based on comparing the message flow in the interaction diagram and the information extracted from the source code.
- A code generation facility.
- A facility for extracting information from the interaction diagram and presenting it to the user in the form of collaborations and CRC cards.

There are two ways to analyze a source code. The first one aims at analyzing the dynamic behavior by debugging the code, while the second one aims at analyzing the static behavior by scanning the code. We decided to use the latter, because our aim is not only to extract the different message calls, but also to obtain additional information such as the different possible execution paths of the code.

1.4 Thesis Organization

The remainder of the thesis is divided into six chapters: related work, the Coin tool, the Coin design, the Coin environment, sample application, and conclusion.

- **Chapter 2 Related Research:** In this chapter we will discuss some background work such as OOSE, use cases and interaction diagrams. We will also define some terminology such as collaboration graphs and CRC cards. Some related research will also be presented.
- **Chapter 3 The Coin Tool:** In this chapter we will discuss in general the different features provided by Coin. Also we will define where Coin fits in the software development life cycle.
- **Chapter 4 An Overview of Coin:** In this chapter we will give an overview of the Coin environment, and we will also discuss some implementation issues.
- **Chapter 5 The Design and Implementation of Coin:** In this chapter we will discuss the different classes used in the development of Coin, their organization and their relationships.
- **Chapter 6 Sample Application: An Airline Reservation System:** In this chapter we will give an example of using Coin in developing a sample application.
- **Chapter 7 Conclusion and Future Work:** This is the last chapter, where we summarize our work, and discuss future enhancements to Coin.

Chapter 2

Related Research

In this chapter we discuss a number of important topics related to the thesis work such as use cases and interaction diagrams. We will also define some terminology such as collaboration graphs and CRC cards. Related research on program understanding and program visualization will also be presented.

2.1 Background

The use case model, a notion introduced by Jacobson in OOSE, is an important model produced during analysis, and forms the basis for the subsequent models developed during design, implementation and testing. An important element of the use case model, namely use cases, can be visualized using interaction diagrams.

Our motivation in presenting some of the graphical notations used to represent the message traffic between objects is to show approaches that have been followed to "visualize" the dynamic behavior of an OO computation. We will discuss several representations including object diagrams [Booch 94], Cunningham diagrams [Cunningham 86], event trace diagrams [Rumbaugh 91], and finally interaction

diagrams [Jacobson 93]. We will see that some of these notations are similar, and also that when the number of objects involved is large, some notations will become very complex and it will be cumbersome to follow the message traffic.

2.1.1 Use Case Model

The use case model specifies the functionality a system has to offer from the user's perspective. It uses actors and use cases as illustrated in fig 2.1. The system is bounded by a box. Each actor is represented by a person outside the box, while the use cases are represented as ellipses inside the box.

The use case model is central, and controls the formation of all other models. This model is further refined in the design model to define the protocols of the blocks (or objects). Ultimately, use cases are implemented in source code and used to test the system.

Fig 2.1 Use Case Model

2.1.1.1 Actors

Actors are models of prospective users of the system, and represent the different roles users can take. The user could be a person, or it could be a system. Thus, actors model anything that needs information exchange with the system under development.

There are two types of actors:

- **Primary Actors:** actors who will use the system directly and most regularly.
- **Secondary Actors:** actors that supervise and maintain the system.

2.1.1.2 Use Cases

A use case describes the sequence of actions performed by the system in response to a request initiated by the user of the system. It is a sequence of related transactions performed by the actor and the system in dialogue. If we consider a recycling system, one of the use cases is *a customer returning an item*. When a customer places the returned items in the recycling machine, the system determines the type of each deposit item (can, bottle, crate), and the day total for each received item is incremented. When the customer finishes depositing all the items, and presses the receipt button, the total amount is computed by the system and printed out to the customer.

Each actor will perform a number of use cases. The complete functionality of the system is achieved by going through all the actors and defining everything they

will be able to do. Thus, the collection of all the use cases describes the functionality required of the system.

2.1.1.3 Summary

The use case model consists of actors and use cases and it describes the different ways to use the system from a user's perspective. It forms a thread running through all development work in OOSE, and is considered the basis for the other models. An interaction diagram which is a visualization of a certain use case, is a graphical notation used to describe the message traffic between objects. We can run a use case to generate a corresponding interaction diagram, or we can use an interaction diagram to extract its corresponding use case. In the next section we will discuss different representations of message sending dialogue, and will emphasize interaction diagrams.

2.1.2 Representations of Message Sending Dialogue

In this section, we discuss different graphical notations used to describe the message traffic between objects, and summarize their common features and their differences.

2.1.2.1 Event Trace Diagrams (Rumbaugh)

Event trace diagrams [Rumbaugh 91] are used to describe the sequence of events that represent a particular execution of a system. An event is a signal that something has happened, and is a one-way transmission of information from one object to another [Rumbaugh 91]. The event trace diagram shows each object as a

vertical line and each event as an horizontal arrow from the sender to the receiver. Time increases downward. Fig 2.2 shows an event trace diagram for a phone call. The diagram describes the scenario of messages exchanged between the three main objects: Caller, PhoneLine and Callee when using a phone line. The caller requests a phone line when lifting the phone receiver, and when the dial tone is received, he/she starts dialing the desired number using the phone line. When finishing dialing the number, a ringing phone is sent to the callee through the phone line, and at the same time a ringing tone is transmitted to the caller

Fig 2.2 Event Trace Diagram

2.1.2.2 Object Diagrams (Booch)

An object diagram [Booch 94] is a "mechanism-based approach" used to describe the inter-object behavior [Fowler 92]. The two elements of an object diagram are:

- **Objects:** they are represented by a labeled amoeba shape

- *Relationships*: they are represented by a line with no arrowheads

The mechanism works by numbering the messages in sequence, and providing a pseudo-code for each message in the diagram. An example of an object diagram that represents a phone call is shown in fig 2.3. The three objects are represented by an amoeba shape, while the messages between the objects are represented by a line with no arrowheads. The messages are numbered in sequence, and each number represent an event taking place and is associated with the corresponding line. Arrows are also associated with the lines to describe the direction of the messages.

Fig 2.3 Object Diagram

2.1.2.3 Cunningham Diagrams (Cunningham)

Cunningham Diagrams are another notation used for visualizing the message sending dialogue. An object is represented as a box. An arc originating from the

sending object and landing in the receiving object represents a message between two objects. The selector of the method invoked is placed at the receiving end of the arc.

In case of inheritance, where the message involved exists in the superclass, the receiver is divided into layers representing the classes involved, and the method is located in the appropriate layer. The top layer will be the receiving class name, while the deeper classes are the superclasses [Cunningham 86].

In fig 2.4, we present a Cunningham diagram representing adding a bank account to a list of bank accounts. Two boxes are used to represent the objects involved. The second box is divided into two layers to represent inheritance and class hierarchy. The top layer represent the object (OrderedCollection) that the message *add:* is sent to, while the second layer represent the parent object (Collection) that the message *at:put:* is sent to. The messages are represented by arcs starting from the sending object and landing at the receiving object, and the order of execution is from left to right (*size* first then *at: put:*).

Fig 2.4 Cunningham Diagram

2.1.2.4 Interaction Diagrams (Jacobson)

Interaction diagrams [Jacobson 93] describe a sequence of stimuli between objects representing a certain action taking place in a system. Thus, interaction diagrams are used to visualize the different use cases of the software system.

2.1.2.4.1 Components

An interaction diagram is composed of:

- A *block* that represents an object of the system and is represented using a vertical line. The system border that represents the outside world is also represented by a column.
- An *operation* is triggered by an event or message sent from one object to another. It is represented by a rectangle that is associated with a block.
- A *stimulus* is an event or message sent from one object to another, and provokes a certain operation to occur. It is represented by a horizontal arrow starting from the sending object and ending at the receiving object.

The time axis in an interaction diagram is viewed as going downwards, but the distance between two events in the diagram has no relation to the real time between these events.

An example of an interaction diagram describing the initialization of an airline reservation system is shown in fig 2.5. The objects involved in the diagram are represented by a horizontal line with labels describing the object names. An object name starting with a capital letter (*Airline*) indicates that we are dealing with a class, while a name starting with a small letter (*anAirline*) indicates that we are dealing with

an instance. When a message (horizontal arrow) is sent from one object to another, an operation is initiated at the receiving object and it is represented in the form of a rectangle. We start the use case by creating an airline and initializing its name. After adding the airline to the airline reservation system, a flight is created with information such as its name, starting and destination location, and departure and arrival time and date. Then, the flight is added to its corresponding airline. The same process can be repeated several times until all the information has been fed to the system.

Fig 2.5 Interaction Diagram

2.1.2.4.2 Structure

Interaction diagrams enable the user, not only to get a picture of the message traffic between the objects participating in a use case, but also to capture the patterns of interactions exhibited in a use case. There are two common structures: *centralized* and *decentralized*.

2.1.2.4.2.1 Centralized

A centralized structure is characterized by having much of the behavior of a system placed in one object. A central object acts like a "spider", and controls all other objects [Jacobson 93]. The diagram representing a centralized structure is called a *fork diagram*, and is illustrated in fig 2.6.

Fig 2.6 Fork Diagram

The interaction diagram, Fig 2.6, represents the use case of moving pallets within a warehouse. When asked to move a pallet, the transporter checks how many times the pallet has been moved. If the pallet has been moved the maximum number of permitted times, a decision has to be made whether to move the pallet to a check station or not. If a new checking station is chosen, the move is done, otherwise the pallet is not moved. In this diagram, we can see that everything is controlled by *Transporter* which is the central object that controls other objects

2.1.2.4.2.2 Decentralized

A decentralized structure is characterized by "delegated responsibility". There is no central object, each object has a separate task, and is only aware of the objects it needs to collaborate with in order to help carry out its task [Jacobson 93]. A diagram representing a decentralized structure is called a *stair diagram*, and is illustrated in fig 2.7.

Fig 2.7 Stair Diagram

In fig 2.7, we represent the same use case of moving pallets within a warehouse, but a different design is being used. Decisions have been delegated, and instead of having *Transporter* controlling all the behavior, *Pallet* checks whether it is permitted to move the load or whether it shall be moved to a checking station. So, decisions are spread, and do not reside in one object.

Discussion

According to Jacobson, many people believe that a stair diagram is more "object-oriented" because of the delegation of the responsibilities. However, both structures have their benefits.

A stair diagram is appropriate when:

- Operations will be performed in the same order
- The goal is to encapsulate or abstract behavior

A fork diagram structure is appropriate when:

- Operations can change order
- New operations could be inserted

To handle changes of how a sequence is actually performed, we should encapsulate the actual sequence and thus obtain a decentralized structure. If we want to change the actual order of the operations, it is better to have a centralized structure, as changes will be local to one object. The two structures may be combined to produce a stable and robust structure, because we cannot only regard behavior or data from a robustness view, but also need to consider the ordering of operations.

2.1.2.4.3 Summary

The interaction diagram describes how each use case is presented by communicating objects. The diagram shows how objects realize the use case through their interaction. The interaction takes place as the blocks (or objects) send stimuli between one another. There are two diagram structures (e.g., centralized, decentralized) that can be identified, and these two structures may be combined to produce a stable and robust structure.

2.1.2.5 Summary

We have presented different notations for describing the message traffic between objects. They differ in the way they represent objects, but an arrow or an arc originating from the sending object and ending at the receiving object is used to represent an event taking place between two objects.

Event trace diagrams and interaction diagrams are very similar in representation, and if we add the notion of operation (time) to an event trace diagram we will obtain an interaction diagram. There is no notion of time in either Cunningham diagrams or object diagrams, however an object diagram can be associated with a timing diagram to represent time.

2.1.3 Collaborations Graph

Collaborations represent requests from a client to a server in fulfillment of a client responsibility. A client is the requesting object, and the server is the providing

Collaborations represent requests from a client to a server in fulfillment of a client responsibility. A client is the requesting object, and the server is the providing object. According to Wirfs-Brock, an object collaborates with another if, to fulfill a responsibility, it needs to send the other object a message.

The pattern of collaborations within an application reveals the flow of control and information during its execution. As collaborations are identified, paths of communication between classes are also identified. Such paths will help in identifying subsystems of collaborating classes, which is important for further behavior and knowledge encapsulation within the design [Wirfs-Brock 90].

A collaboration graph [Wirfs-Brock 90], fig 2.8, displays the collaborations between classes and subsystems in a graphical form. Classes are represented by labeled rectangles. Subclasses are graphically nested within the bounds of their superclasses. Collaborations are drawn as an arrow from the client to the server and defines the behavior being requested.

Fig 2.8 Collaborations graph for initializing an airline reservation system

Collaborations are associated with a responsibility. A CRC card is used to record the collaborations of an individual class.

2.1.4 CRC Cards

A CRC (Class Responsibilities Collaborators) [Wirfs-Brock 90] card is an index card used in the design to determine the responsibilities of objects and the collaborators they need. Responsibilities identify behavior, while collaborators specify other objects needed to help an object carry out its task. If a collaboration occurs with a subclass that is providing a service defined by a superclass, the corresponding responsibility will be recorded on the superclass card.

The index card is 4-inch by 6-inch, and the constraints of such dimensions are a good measure of appropriate detail to be included in the card. A class that is expected to perform more tasks than can fit easily in the card space is probably too complex [Budd 91].

The aim of using index cards is the ease of their manipulation and modification. The compact size of the cards makes it easier to view more classes at the same time, by laying the cards on a table. Also, it simplifies reorganization and reshuffling around. A CRC card representing an airline reservation system is shown in fig 2.9. The left side of the cards describes the class functionality (behavior), while the right-hand side contains the objects needed by *AirlineReservationSystem* to collaborate with in order to accomplish the desired tasks.

Class Name: AirlineReservationSystem	
<u>Responsibilities</u>	<u>Collaborators</u>
* add a new airline	Airline
* add a new flight	Flight
* book a flight	Passenger

Fig 2.9 CRC Card

2.1.5 Summary

Interaction diagrams describe the message flow between objects, and they are a one-to-one mapping of use cases. Use cases are one of two components comprising the use case model which is the backbone of OOSE. The use case model controls a large part of the system development in OOSE, and forms the basis for both the construction and testing processes. Interaction diagrams do not only give the engineer an insight on the system under development, but also contain information that can be represented in other forms such as collaboration graphs. Collaboration graphs represent communication between objects, and are associated with responsibilities. The responsibilities of a particular class are recorded using a CRC card which is an index card utilized to describe the behavior of different objects and the collaborators (or helpers) they need.

2.2 Related Work

In this section we discuss program visualization, the Objectory case tool, and Rational Rose/C++. We are interested in program visualization systems because, like Coin, they use graphic representations to illustrate programs. Objectory SE, is a CASE tool describing Jacobson's methodology OOSE. One of the facilities that Objectory provides is an interaction diagram editor, a component needed by Coin. Rational Rose/C++ provides a C++ code generation facility; one of our goals is to facilitate the coding process by generating Smalltalk code constructs.

2.2.1 Program Visualization

The way a problem is represented strongly affects whether we can understand and solve it. Visual representation takes advantage of the power of the human visual system to provide insight and understanding. A program visualization system aims at visually representing different aspects of a program. Program visualization not only helps the engineers to understand programs, but is also very useful in other areas such as software design. The latter is considered one of the most important applications of program visualization, because the gradual shift from designing software from scratch to creating software through redesign and reuse requires the programmer to first understand the software in its existing form.

2.2.1.1 Cara: An Environment For Interactive Design Of Communications Architectures

Cara [Citrin 90] is a system which allows a designer to enter message-flow diagrams and data operations, and to begin simulating with them. A message flow diagram is a view of the behavior of a set of communicating entities unfolding over time. It defines constraints on the architecture, and it simulates with a graphical representation the response of the system to certain conditions.

The flows are taken as a partial specification of control-flow and data-manipulation operations. Both control and data operations are translated into rules, which can be fine-tuned by the designer and used for simulation. In the Cara system, a flow-to-rule translator and a rule simulator operate concurrently with the designer. As the designer creates and modifies a diagram, the translator incrementally updates the architecture. The simulator uses the updated definition immediately to offer a next action to the designer. In this way, the designer can always see whether the architecture is operating as intended.

The user has the ability to change any part of the diagram at any time. The user's actions can be unusual or meaningless such as trying to send messages backward in time or placing protocol entities on top of event points. The system has to deal with such actions either by preventing them or by dealing with them. The approach used, is to examine the architecture for a defined response to the new object added. If one is found, the system shows it to the user by placing it onto the diagram. If none is found, the architecture is modified to deal with the new object. Making a change in the diagram may invalidate much of the rest of the diagram, so the environment marks those parts of the diagram which are inconsistent with the new

situation, allowing the user to see the effects of the change. This feature inspired us to include a consistency checking facility in Coin to verify the user actions when drawing an interaction diagram.

2.2.1.2 KAESTLE

KAESTLE [Boecker 86] automatically generates graphical representations of list structures and allows the user to edit them directly with a pointing device (fig 2.10).

KAESTLE helps the Lisp beginner to understand certain aspects of the programming language that are difficult to explain. More experienced Lisp programmers use it heavily to display and explore data structures that are difficult to represent symbolically, namely circular and reentrant structures. It can be used to design, debug, document, and understand Lisp programs or data structures.

One of the problems that needed to be solved when designing KAESTLE was the spatial layout of List structures. The approach to solve the problem was to build cooperative systems for the human and the computer. In KAESTLE, the computer uses a simple planning algorithm: it doesn't pay attention to arrows crossing boxes or other arrows, and the display is truncated at the right and lower border of the display area. After the computer generates the representation, the user can move or delete parts of the display, or display additional substructures, until they arrive at a nice output that shows the part of the structures they want to see.

The approach used influenced our decision about diagram modification when automatically updating the diagram. In Coin, when adding new components to the

figure, it may occur that the diagram modification will result in having overlapping components. We had to decide between leaving the user with the task of modifying the diagram, or automatically shuffling the components around to keep the diagram intact. We decided to use the latter approach.

Fig 2.10 KAESTLE

2.2.1.3 Interaction Diagram Generator (IDG)

IDG [Wang 93] is a research tool developed at Carleton University by Yun Wang to support design capturing. IDG uses the notion of use cases and interaction diagrams to display the dynamic behavior of a system.

IDG has the ability to generate interaction diagrams automatically by executing Smalltalk code, using the use cases defined in the system. The generation of interaction diagrams can be done as one shot, or it can be done step by step. In the later case, the tool can be used as a debugger. By running all the use cases and generating their corresponding interaction diagrams, we can get an overview of the overall dynamic behavior of the system. Fig 2.11 shows an example of an interaction diagram generation.

IDG also captures additional design information, through generating collaboration information. Filters can be used to select and display only the objects and messages that the user is interested in.

We were interested in IDG because we wanted to examine the process of capturing dynamic behavior through the use of the Smalltalk/V debugger. When analyzing the Smalltalk code we had to choose between dynamic and static analysis, and the work of Yun Wang was related. Although we used a static analysis approach to analyze the Smalltalk code, we used the approach of displaying the diagram in two panes: one for the object names and the other for the diagram components, as devised in IDG.

Fig 2.11 Interaction Diagram Generation using IDG

2.2.2 Objectory SE

Objectory is a technique for industrial system development based on the object-oriented paradigm using Jacobson's notation. Objectory stands for "Object Factory", and expresses the concept of software factory realized by means of an object-oriented approach. It is "a life cycle-encompassing" technique that fully

supports the tasks of development, from requirements specification, through analysis and design, to a tested, complete system.

Objectory Support Environment (Objectory SE) is a set of integrated workstation- and/or PC-based graphical tools that support system modeling in accordance with the Objectory process. In other words, Objectory SE is a CASE tool supporting the tasks of object-oriented analysis and design in accordance with the Objectory development process.

Objectory SE enable the user to prepare and edit both textual and graphical views of Objectory models, and supports development in the top-down direction; i.e., analysis to design to implementation. Objectory provides the software engineer with a variety of tools that includes: a *browser* for editing an Objectory model, a *document editor* for writing descriptions of other objects or other concepts within the model, a *diagram editor* for creating graphical views of any Objectory model, which can be included in any document, and others. We are interested in the interaction diagram editor, which is one of the available tools. Interaction diagrams are created and changed with the editor (fig 2.12). The editor allows the engineer to create operation paths, stimuli transmissions, and margin texts.

Our goal in investigating Objectory SE was to examine the interaction diagram editor and to identify the features and functions it provides. However, we were not able to get a demo version of the CASE tool as the tool is very expensive.

Fig 2.12 Objectory's interaction diagram editor

2.2.3 Rational Rose/C++

Rational Rose/C++ is a graphical software engineering tool that supports object-oriented analysis, design and implementation by automating the use of Booch notation and by generating C++ code from the design model. It supports software development in the top-down direction.

Rational Rose/C++ provides the user with a C++ generator. The C++ generator is a code generation capability that generates C++ source code files from information in a Rose model (Booch model). These files contain C++ code constructs that correspond to the notation items (classes, relationships) that the software engineer has defined in the model with diagrams and specifications.

The generated files are created in a hierarchy of subdirectories that correspond to the class categories and/or subsystems in the model. The C++ generator creates these directories as necessary, and derives their names from the corresponding category/subsystems names.

```

/**begin module.additionalDeclarations preserve=yes
/**end module.additionalDeclarations

// Class PlanAnalyst
//
// Concurrency: Sequential
// Persistence: Transient
// Cardinality: n
//
class PlanAnalyst
{
public:
    // Constructors
    PlanAnalyst();
    PlanAnalyst(const PlanAnalyst &right);

    // Destructor
    ~PlanAnalyst();

    // Assignment Operation
    const PlanAnalyst & operator=(const PlanAnalyst &right);

    // Equality Operations
    int operator==(const PlanAnalyst &right) const;
    int operator!=(const PlanAnalyst &right) const;

    // Additional Public Declarations
    /**begin PlanAnalyst.public preserve=yes
    /**end PlanAnalyst.public

    protected:
        // Additional Protected Declarations
    /**begin PlanAnalyst.protected preserve=yes
    /**end PlanAnalyst.protected

    private:
        // Additional Private Declarations
    /**begin PlanAnalyst.private preserve=yes
    /**end PlanAnalyst.private

    private: // implementation
        // Additional Implementation Declarations
    /**begin PlanAnalyst.implementation preserve=yes
    /**end PlanAnalyst.implementation
};

#endif

```

Fig 2.13 Header file generated by Rose/C++ generator

A pair of header (.h) (fig 2.13) and implementation (.cpp) (fig 2.14) is generated for each class item selected in the diagram. The name of each file is derived from the name of the corresponding class item. Header files are generated from the implicit module specifications, and implementation files are generated from the implicit module bodies.

```
// Module Body PlanAnalyst (Package)
//
// Subsystem: Planning
// File: planning\plnnlyst.cpp
//

/**begin module.includes preserve=yes
**end module.includes

// PlanAnalyst
#include "planning\plnnlyst.h"

/**begin module.additionalDeclarations preserve=yes
**end module.additionalDeclarations

// Class PlanAnalyst
// Constructors
PlanAnalyst::PlanAnalyst()
/**begin PlanAnalyst::PlanAnalyst%.hasinit preserve=no
**end PlanAnalyst::PlanAnalyst%.hasinit
/**begin PlanAnalyst::PlanAnalyst%.initialization preserve=yes
**end PlanAnalyst::PlanAnalyst%.initialization
{
/**begin PlanAnalyst::PlanAnalyst%.body preserve=yes
**end PlanAnalyst::PlanAnalyst%.body
}

PlanAnalyst::PlanAnalyst(const PlanAnalyst &right)
/**begin PlanAnalyst::PlanAnalyst%copy.hasinit preserve=no
**end PlanAnalyst::PlanAnalyst%copy.hasinit
/**begin PlanAnalyst::PlanAnalyst%copy.initialization preserve=yes
**end PlanAnalyst::PlanAnalyst%copy.initialization
{
/**begin PlanAnalyst::PlanAnalyst%copy.body preserve=yes
**end PlanAnalyst::PlanAnalyst%copy.body
}

// Destructor
PlanAnalyst::~PlanAnalyst()
{
/**begin PlanAnalyst::~PlanAnalyst%.body preserve=yes
**end PlanAnalyst::~PlanAnalyst%.body
}

// Assignment Operation
const PlanAnalyst & PlanAnalyst::operator=(const PlanAnalyst &right)
{
/**begin PlanAnalyst::operator=%.body preserve=yes
**end PlanAnalyst::operator=%.body
}

// Equality Operations
int PlanAnalyst::operator==(const PlanAnalyst &right) const
{
/**begin PlanAnalyst::operator==%.body preserve=yes
```

```
///end PlanAnalyst::operator==%.body
}

int PlanAnalyst::operator!=(const PlanAnalyst &right) const
{
  ///begin PlanAnalyst::operator!=%.body preserve=yes
  ///end PlanAnalyst::operator!=%.body
}

// Additional Declarations
///begin PlanAnalyst.declarations preserve=yes
///end PlanAnalyst.declarations
```

Fig 2.14 Implementation file generated by Rose/C++

When the software engineer is satisfied with the way the code is generated from the model, the engineer can complete the code by filling in implementations for function bodies.

Discussion

Our goal in discussing Rose/C++ is to examine the C++ generator. We are providing a code generation facility in our tool, and we wanted to explore the different activities involved in code generation, and the kind of generated information. We decided to follow the same approach of Rose/C++ by creating the different classes from the objects defined in the interaction diagram, and to create all the methods (class, and instance) that a class can understand, and to add these methods to the class methods list.

2.2.4 Summary

The research done with program visualization, Objectory SE, and Rational Rose/C++ has inspired many ideas for the design of Coin. In particular we adopted the following strategies:

- **Use a graphical editor to construct interaction diagrams**
- **Provide a consistency checker to respond to meaningless users' actions when using the editor.**
- **When updating the diagram automatically, provide automatic diagram translation and modification to prevent any overlapping between diagram components.**
- **Provide a code generation facility.**

Chapter 3

Coin Tool

In this chapter we discuss the goals behind developing the tool, the different features provided by Coin, and where the tool fits in the software development life cycle.

3.1 Development Goals

Our primary goal in developing Coin is to provide a tool that supplies the user with bi-directional software development facilities. In the top-down model we will focus on interaction diagrams as our design notation, and in the bottom-up model we will use Smalltalk code to drive design information. We are using Smalltalk/V for Windows as our development language.

Our interest is not only to provide the user with facilities to build interaction diagrams or utilize Smalltalk code to acquire design information through interaction diagram update and modification, but also to benefit from the richness of the information embedded in interaction diagrams, and to present that information to the user in different forms. In particular, Coin aims to provide the user with:

- A view of the interaction diagram and associated message code.

- The ability to manually construct interaction diagrams.
- The ability to automatically modify the diagram to synchronize with changes found in the code.
- The ability to extract useful information such as object collaborations and make it available to the user in a convenient form.
- A code generation facility.

In fig 3.1, we represent the major components of Coin.

Fig 3.1 Coin Architecture

3.2 Features

We are going to discuss the different features provided by Coin to meet our main developing goals. More details and illustrations will be presented in the following two chapters.

3.2.1 Interaction Diagram Editor

The diagram editor will supply the user with all the tools needed to construct interaction diagrams. It include the following facilities:

- *Drag and Drop* provides the user with the capability to move any diagram component from one place to another, and gives the user more flexibility when building the diagram.
- *Consistency Checking* aims at responding to meaningless actions by the user such as trying an illegal message connection between two objects. Examples of irregularities detected by the checker are described in more detail in chapter 5.
- *Copy, Cut and Paste* allows the user to copy a portion of the diagram and to paste it into another location, eliminating the need for duplicating the drawing of similar segments of the diagrams.
- *Use of different fonts* supplies the user with a set of different fonts that can be used to display object names and message names.

- *Use of different colors* to distinguish between the different components of the diagram gives the user more diagram visibility; e.g. *black* for objects and *red* for messages.

3.2.2 CRC Card Manager

The CRC card manager provides the user with a CRC card for each object in an interaction diagram. The collaborators part will be filled in from the information extracted from the diagram. The user has only to fill in the responsibilities. The CRC card manager is sensitive to diagram changes, as the cards are updated to reflect any modifications in the diagram. The information on the card is saved with the diagram, and is reloaded when the diagram is opened again.

3.2.3 Collaboration Information

For each object in the diagram, we extract the helpers and the users of the object. The information is obtained based on the notion of the message traffic between the objects. In fig 3.2, we have an exchange of information between object *A* and object *B*. We say that:

A uses *B* or

B helps *A*

Fig 3.2 An interaction diagram

3.2.4 Code Generation

The goal of having a code generation facility is to ease the process of implementation. Smalltalk code constructs will be generated upon user's request. The designated objects will be added to the Smalltalk class hierarchy, if they don't exist. All the messages that an object can understand will be considered the methods that a class can respond to, and will be added to the class methods list. A browsing facility is also available for the user to access class information such as attributes (instance and class variables) and behavior (instance and class methods). When an object method is selected, its associated code is provided.

3.2.5 Diagram Modification

This feature is not only concerned with automatic diagram modification, but also with providing the user with another view of the messages defined in the diagram. For every message two views are presented to the user:

- Actual Smalltalk code

- **List of message calls extracted from the code**

The list of message calls extracted from the code will be compared to the messages defined in the diagram. If discrepancies are found, Coin provides an automatic diagram update that will supply the necessary modifications needed to synchronize with the changes found in the implementation.

3.3 Applications

The object-oriented life cycle is an iterative process, and analysis, design and implementation are tightly coupled and it is very difficult for example to separate design and implementation. In OOSE [Jacobson 93], design and implementation are combined together to form a "construction process" that is one of three processes representing the OOSE life cycle. Our aim in developing the tool is to provide a facility for bi-directional software development. We can use Coin in the following ways:

- **Top-Down:** we provide the user with facilities to draw interaction diagrams, to present information extracted in forms such object collaborations and CRC cards. A coding utility is also available where Smalltalk code constructs are generated upon request and based on information embedded in the diagram.
- **Bottom-Up:** we use the implementation information (Smalltalk code) to update the design information. Interaction diagrams are automatically updated when any discrepancies are discovered between implementation and design. At the same time, collaborations and CRC cards are also updated to synchronize with the new changes detected.

In fig 3.3 we summarize the tool's application in the object-oriented life cycle.

Fig 3.3 Coin applications in object-oriented life cycle

3.4 Summary

We have developed Coin with the aim of providing support for bi-directional software development; i.e., top-down and bottom-up development. Coin enables the user to construct interaction diagrams that, in turn, are utilized as a source for extracting different forms of information such as CRC cards. Also, the source code is

used to modify the interaction diagram when any discrepancy is detected between the messages in the code and the messages in the diagram.

Chapter 4

An Overview Of Coin

4.1 General View

Coin is a tool integrated with the Smalltalk/V environment. Our goal in designing the user interface was to find a way to enable the user to view the interaction diagram, and at the same time to obtain the corresponding information (source code and message calls) for any message in the diagram. In this way, the process of matching the design with the actual implementation will become much easier. In addition, there will be no distraction to the user caused by having to toggle between different windows.

In our first design of the tool's interface, we decided to have the interaction diagram editor as our main opening window, and when the user is interested in getting implementation information about any message in the diagram, another window will be opened where the message code, and analysis information will be presented. However, we found that if the user has to compare two pieces of related information in two different windows, it causes a lot of distraction, and it is very hard to keep track of the information presented. Thus, we decided to divide our main

opening window into two views: one to represent the editor, and the other to represent implementation information.

The main view is divided into two portions. The first one represents the interaction diagram view with the corresponding tools to build and modify the diagram. The source code information is represented in the second portion. We will discuss the two views in later sections of the chapter. Fig 4.1 illustrates the general view of Coin.

Fig 4.1 General View

4.2 Interaction Diagram Editor View

Interaction diagrams are created and changed with the interaction diagram editor. The editor provides the tools necessary to construct the diagram, and, in addition, additional facilities for dragging, copying and pasting, and loading and saving. The editor contains two panes: one for displaying the blocks (objects) names and the other for displaying the diagram itself. The reason we have used two panes is very simple: scrolling. If we are scrolling vertically and we are only using one pane, we will lose the names of the objects and it will be difficult for the user to track the flow of messages between the different objects. Thus, two panes are being utilized, so that scrolling will only affect the diagram and not the names of the objects. One problem with having two panes is the need for a synchronous movement in case of an horizontal scrolling, so that the two panes scroll together at the same time.

We are also using different colors to distinguish between the different components of the diagram. We have assigned a default color to each component; e.g., *black* for the objects, *red* for the messages, and *blue* for the operations. When a new component is added to the diagram, the figure representing the desired component is drawn using the default color. The user can alter any of the default colors of any component using the color palette, and the new color will be used in any further drawings. The goal in using different colors is to differentiate between the different elements of the diagram and to enhance diagram visibility. Fig 4.2 shows the different components of the editor.

When creating a diagram object, the corresponding column will be drawn with a question mark (?), representing the name, associated with it (fig 4.3). Our

incentive is to give the user more flexibility when building the diagram, instead of having to pop up a window each time an object is created. Later, the user can rename any of the created objects.

Fig 4.2 Editor View

Fig 4.3 Object creation

Another goal we had in mind when designing the editor, was to incorporate "smart" features. There is an on-line consistency checker (discussed in the following chapter) which detects any irregularities or illegal connections. An overall consistency checker is also available that will identify:

- Objects with no associated operations
- Operations with no associated messages
- Operations with a message from, but no message to.

Fig 4.4 illustrates the irregularities detected by the overall checker.

Fig 4.4 Irregularities detected by the overall consistency checker

It happens often that the same message is used by different objects. Let us consider the situation in fig 4.5, where we have the same message **new:** sent to class **Ordered Collection** from the **System Border** and class **Sorted Collection**. If the user wants to change the name of one of the messages to **myNew:**, the diagram will be smart enough to recognize the existence of the other messages. In such a situation, the user will be given a list of the occurrences of the message in the diagram as shown in fig 4.6. Each message is associated with the sending object. The user has the choice to rename all of the messages, to select some for renaming, or to keep the names unchanged. This feature will provide more reliability and robustness, it will eliminate the need for a manual change of all duplicates, and it will also give the user the choice of changing the name of some of the messages, or keeping them without any modification.

Fig 4.5 Interaction Diagram with duplicate messages

Fig 4.6 Renaming a message

4.3 Object Collaboration View

The object collaboration view shows the cooperation between the objects of an interaction diagram. Two types of objects are involved: *helpers* and *users*. The *helpers* of an object are the set of objects used by that object; i.e., these objects are necessary for the object to collaborate with in order to accomplish its task. The *users* of an object are the opposite of helpers and they represent the objects that need the assistance of and collaboration with that object in order to achieve their goals.

The helpers and users of an object are extracted from the information included in the diagram. The message sending between the different objects is used as our

basis of extraction. The *helper* will be the receiving object, while the *user* will be the sending object. Thus, the process of extracting the helpers and the users is somehow similar. During implementation, we decided to extract only the helpers of each object in the diagram. Then, the helpers information will be used to deduce the users. The example in fig 4.7 illustrates the process of extracting the users of an object using the helpers information.

<u>objects</u>	<u>helpers</u>
A	B
B	A, C
C	A

If we want to get the users of A, we go through the helpers list of the remaining objects. If object A is found in the list, then the users of A will include the object whose helpers include object A. From the above information, the users of A will be B and C. The same procedure will also be followed for B and C.

Fig 4.7 Extracting the users from the helpers

The extraction of collaboration information is very helpful, because it can be used with other documentation notations such as CRC cards, as we will demonstrate later. It is also very useful in capturing design information and in the regeneration of design documents. The collaborations of the objects involved in an airline reservation system (fig 2.5 and fig 4.2) is shown in fig 4.8. Several windows are shown respectively to display the information related to the different objects available in the interaction diagram.

Fig 4.8 Objects Collaborations

4.4 CRC Card View

A **CRC card** is an index card used to describe the activities performed by an object and the collaborators (or helpers) that an object needs in order to achieve its responsibilities. The card is divided into two sides: the left side includes the responsibilities and the right side includes the collaborators.

The **CRC Card Manager** is a tool that provides the user with a CRC card for each object in the diagram. The collaborators part will be automatically obtained from the information extracted from the diagram. We use the helpers that were extracted in the objects collaboration to fill the collaborators part, and the user has the task of filling in the responsibilities information. Any changes in the diagram, such as adding new components, will be reflected in the collaborators part of the CRC card.

Our goal in designing the **CRC Card Manager** is to achieve simplicity and ease of use. The view is divided into two parts:

- A list of the available objects in the diagram
- An empty CRC card

When an object in the list is selected, the CRC card will be filled with information including the object name and the collaborators. If the user has previously filled in some responsibilities, that information will also be provided. The user can edit the responsibilities at any time, and information can be changed, added or deleted. This information will be saved when saving the diagram, so that it can be utilized any time later. We have provided an example of a CRC card view in fig 4.9, based on the airline reservation system setup diagram (fig 2.5, 4.2)

Fig 4.9 Card Manager

4.5 Code Generation View

Forward Engineering is the traditional process of moving from high-level abstractions and logical implementation-independent design to the physical implementation of a system. We have incorporated a code generation facility into our tool to provide the user with the option of using the tool as a forward engineering tool.

The function of the tool is to generate Smalltalk code constructs based on the information included in an interaction diagram. Each object in the diagram will be added to the Smalltalk class hierarchy, if it does not already exist. All the messages that the object can understand will be extracted from the diagram and will be added to the object methods list (class methods and instance methods).

The user will be given a list of the available objects in the diagram, as illustrated in fig 4.10. Two colors are used to distinguish between the objects that already exist in the hierarchy and those that do not exist. When adding an object, the user is asked to provide the superclass so that the object can be placed properly in the hierarchy.

Information about each object is also presented to the user. The user will be supplied with:

- Object attributes (class and instance variables)
- Object behavior (class and instance methods)
- Source code of any selected method.

Information is presented in front of the user classified by attributes, behavior, and source code. We have decided that by presenting all the information, we will eliminate the need to click any buttons or select any menu item.

Fig 4.10 Code generation view

Fig 4.11 Object Information

4.6 Method Information View

The Method Information View is designed to provide the user with a concrete representation of the implementation. The user is provided with three pieces of information (see fig 4.12):

- Method source code
- Source code message calls
- A label describing whether the implementation is consistent with the design or not.

Fig 4.12 Method Information View

The implementation information will be displayed on the screen when a message of the diagram is selected by double clicking on it. The source code will be displayed if the receiving object already exists in the class hierarchy. Otherwise, only the name of the method will appear in the code pane. The source code can be edited by the user and it can also be saved. Any update to the source code will be reflected in the list of corresponding message calls.

The list of message calls is based on the static analysis of the source code. The different messages and their receivers are extracted, and each message call is displayed in the form: 'receiving object <- message'. The aim of the representation is to define which message is sent to which object. To distinguish between class and instance methods, the keyword *class* is used after an object name in case of a class method. The order of the messages in the list is based on the order of execution of messages as defined in Smalltalk. So if we have the following code:

```
|x|  
x := OrderedCollection with:3.  
x add:(x at:1).
```

The order of message execution will be: *with:* followed by *at:* and finally *add:.*, and this will be the order of messages that will appear to the user.

In static analysis, it is very difficult to determine the receiving object of a message. Usually, the receiver will not be known until run-time. In some cases, the receiver can be determined when a keyword like *self* and *super* is being used. The user is supplied with a mapping facility. The function of the mapping facility is to map the receiver name of the message to an existing object in the Smalltalk class

hierarchy. The map window (fig 4.13) contains two list boxes representing respectively: a list of message calls and a list of the available objects that can understand each message. When a message is selected, the list of available objects that can respond to such a message are displayed in the second list box. Any object can be chosen to be mapped as the receiver.

Fig 4.13 Map Window

Once a mapping is done, anytime a message is selected in the map window, the list of objects displayed will only contain the mapped object. We have mapped the receiver of message *at:* (fig 4.13) to *Array*. In fig 4.14, when the message *at:* is selected only *Array* appears in the list of objects. A user can change the mapping of any message receiver at any time. In this case, any previous mapping is deleted, and the process of mapping is repeated once again.

A comparison between the outgoing messages of a selected diagram message and the actual source code message calls is performed. As a result of the comparison, a signal label is displayed showing either a 'Consistent Method' statement or an

'Inconsistent Method' statement. Two different colors are used to distinguish between the two statements and also to attract the attention of the user.

Fig 4.14 Mapped Message

Often a piece of code can contain different execution paths. Let us consider the following code:

```

|x|
x := OrderedCollection new.
x at:1 > 3 ifTrue:[ Transcript show:>3]
               ifFalse:[Transcript nextPutAll:'<=3']

```

We have two possible paths of execution. The first one is to execute *new* followed by *show:* (truth branch). The second one is to execute *new* followed by *nextPutAll:* (false branch). The user has an option to view all the possible paths of the execution at any time, and any path can be used to update the diagram. This function is very helpful because a general diagram can be drawn, then by using different execution

paths we can generate different diagrams for different cases. This way we do not have to redraw the diagram again for each case, and we will reduce time and energy.

Once the user has an overview of the design and implementation information, the interaction diagram can be updated either manually or automatically to maintain consistency between design and implementation. The automatic update of the diagram is done in an incremental way. The user has the two sets of messages (diagram and code), if they are inconsistent an automatic update can be requested. According to the comparison, the interaction diagram will be updated by adding/deleting the necessary components to maintain synchronicity with the implementation.

It often happens that the user is not interested in reflecting certain code messages in an interaction diagram. In the automatic diagram update mode, the default is the whole list of the source code message calls pertaining to a specific execution path. However, the user can select certain messages to be used in modifying the diagram. An example of selecting messages for modifying an interaction diagram is shown in fig 4.15.

4.7 Summary

Coin was designed to provide support for bi-directional software development. We are supplying the user with different views of the design and implementation information. The graphical user interface was designed to ease information access, and the design (interaction diagram) and implementation (source

code and code analysis) information are presented to the user in adjacent panes to minimize distraction and window handling.

Fig 4.15 Automatic diagram update

Chapter 5

The Design and Implementation of Coin

In this chapter we discuss the design principles of Coin, the different classes involved and their relationships.

5.1 Design Principles

Coin is developed using Smalltalk/V under Windows 2.0 and is integrated with the Smalltalk environment. Working with Smalltalk has many benefits, and one of the important benefits is reuse. Smalltalk provides the user with a wide range of classes that can be used when developing an application. Also, new classes can be created and added to the existing class hierarchy. When designing the tool, we tried to maximize the use of the Smalltalk class library as much as possible. We have created new classes, and we have also added new methods to existing classes to satisfy certain requirements that are not previously provided.

The primary goals of Coin are to provide the user with the ability to:

- build interaction diagrams
- view implementation information of any diagram message including source code and message calls

- request an automatic diagram update based on the comparison between the diagram and the actual implementation.

Besides these goals, we wanted also to benefit from the information embedded in the diagram and to present this information to the user in the form of collaborations and CRC cards. In addition, the tool provides the user with the capability to generate Smalltalk code constructs.

In our design, we used a centralized structure when controlling the main functions of the tool. In fig 5.1, we describe a general overview of the tool's functionality. *Coin* is the central element that responds to the user's actions, and directs requests to the responsible elements which in turn know the other relevant elements needed to carry out the desired task. The latter approach constitutes a decentralized structure.

Fig 5.1 Overview of Coin

The rest of the chapter is organized as follows: **Section 2** describes the main technical issues and our approach to each; **Section 3** describes the important classes we added, and some of the classes we modified either by adding new methods or by adding some code to existing methods.

5.2 Design Issues and Choices

This section consists of four parts, discussing in turn the following issues:

- **Diagram editor**
- **Information extraction and presentation**
- **Code Analysis**
- **Automatic Diagram Update**

5.2.1 Diagram Editor

We provide the user with a diagram editor to build and modify interaction diagrams. The necessary blocks for an editor are: the components constituting a diagram and the tools necessary to create the different components. The coordination is maintained by the editor, which also supports features such as dragging and dropping, cutting and pasting, and filing facilities.

The actions of the user may not be those anticipated as being useful, but may also be unusual, or even seemingly meaningless, e.g., trying to send a message between two operations of the same object. We needed to deal with such situations. Coin has a consistency checker which is activated when adding new components to the diagram or moving components to a new position. We will discuss the different irregularities detected by the checker when discussing the different classes

implemented in detail. A general overview of the editor activities is summarized in fig 5.2.

Fig 5.2 Overview of Coin's editor activities

In designing the interaction diagram editor, we had to take care of some design issues such as: keeping the graphical figures up to date and printing the diagram. The former deals with maintaining the picture of the interaction diagram. We do not want to lose the picture of the diagram when the window containing the diagram is resized, hidden, or loses focus. The choice was to use the method *drawRetainPicture*: of the **RecordingPen** object. This method draws the picture and records it in a certain segment. The problem with this solution is that complexity increases with the number of figures drawn. The reason for such complexity is the need to save all the segments of the interaction diagram components to recall them

later when dragging, cutting, or clearing components is desired. We have to remove the segments of the deleted (or removed) components from the list of segments, otherwise the components will remain visible in the diagram. This solution is more severe in our design because we are using two different panes (title, and objects) to show the interaction diagram, and each pane keeps its own list of segments. Our solution to the problem was to use the event *display* that is understood by the two panes. When the event is called, we invoke our own display method that takes care of displaying the diagram to the user.

The problem we faced when printing the interaction diagrams from Coin emerged from our solution to the problem of displaying the diagram (above). When the contents of a graph pane is printed either by sending a bitmap of the contents to the printer, or by supplying the printer with the drawing pen, the list of segments defined in the pane is used to print the contents. As we eliminated the use of the segments, we kept getting a blank page. We decided that when printing is requested, we will redraw the diagram components and record them. When printing is completed, the list of segments for each pane is deleted.

5.2.2 Information Extraction and Presentation

One goal we had in mind when developing the tool, was to extract useful information from the interaction diagrams, and to present that information to the user. The user has access to information in the form of:

- Object Collaborations
- CRC Cards
- Objects information

- In *object collaborations*, the user can obtain information about the helpers and users of a certain object. The information is extracted from the diagram and presented to the user. The information is based on the message flow between objects, when object A sends a message to object B, we consider that A uses B or B helps A. So the process of extracting helpers and users is similar, and we decided to extract helpers first and then use that information to extract the users of an object, and we have discussed our implementation approach in the previous chapter.

In *CRC cards*, we are supplying the user with a CRC card where the collaborators part is automatically filled in with information extracted from the diagram. We use the collaborators part of the collaborations to fill in the collaborators part of the CRC card. The responsibilities part has to be filled in by the user, and we have to keep that information, so that when the user accesses the CRC card, the information will be present. The problem we faced was how to save the responsibilities information. We had to choose either to save the information within the Smalltalk image, or to save the information as a hard copy. We decided that we are going to save the information to a file each time the diagram is saved. So when the diagram is restored from the file, the CRC card information will also be restored.

In *Object information*, we are providing the user with a facility to get implementation information of a certain class in the form of its instance and class variables, its instance and class methods, and also with a facility to view the code of any selected method. The user also has the opportunity to change the code of any method. When the user modifies the code of any method, we must make sure that the new source code is reanalyzed to maintain analysis up to date.

5.2.3 Code Analysis

The aim of analyzing the Smalltalk code is to provide the user with a view of the actual messages defined in implementation, and to compare these messages with those defined in the diagram. The comparison serves as a means of updating the diagram automatically.

Two different strategies can be followed to understand a program: static analysis, and dynamic analysis. In dynamic analysis, we extract information during program execution, while in static analysis we extract information through scanning the code.

Our first approach was to investigate the dynamic behavior, by obtaining information about the different messages and their receivers from the debugger. Dynamic analysis is very useful because it enables us to understand the behavior of the program, and also we can obtain information about the different messages and their receivers. The problems of using a dynamic analysis strategy is :

- The implementation code must be complete.
- Each time a method is changed and an automatic update is requested, we must run the whole use case until the desired message is activated, in order to analyze its code.
- We must supply different execution data in order to get different paths of execution.

Our goal is not only to analyze the code, but also to use the analysis information to update the diagram. However, we wanted to provide more information

such as the possible paths of execution, and also to use Coin as a prototype tool, where the engineer can draw a diagram, then go and write some code, and see how the new code will affect the diagram. These are the reasons that made us follow the static analysis approach.

Smalltalk is a dynamic language, which means that the receivers of the messages will not be decided until run-time. One problem of using a static analysis approach is that we will not be able to determine the receivers of the messages, because static analysis does not provide such information. We solved the problem, by asking for help from the user. We found that it will be more reasonable to provide the user with a point and click facility, instead of asking the user each time to type the name of the objects. So, the software engineer will be given a list of all the objects that can understand and respond to a certain message, and asked to choose which object is going to be the receiver of the message. Once all the messages have been mapped to their receivers, the diagram can be updated.

5.2.4 Automatic Diagram Update

Updating the diagram automatically leads to the addition of new objects, messages and operations, and/or the deletion of existing messages and operations. The problem of adding new components to the diagram is that the structure of the diagram will change and the components may overlap. We had two options to choose from after updating the diagram:

1. Leaving the user with the task of moving overlapping components around until a nicer look is achieved.

- 2. Updating the diagram instantly to cope with new modifications.

We chose the second option because our goal is to ease the process of diagram modification, and to update the diagram without leaving the user with the task of adjusting the diagram for a nicer look.

To solve the problem we needed to make sure that space is available to add the new component and that the added component will not overlap with an existing one. The solution required moving components around to create available space for the new components to be inserted in the diagram. The movements of the different components created other problems that were not apparent the first time.

Interaction diagram components are tightly coupled. A message is sent from a sender to a receiver and it initiates an operation at the receiving object. An operation is always associated with an object. When one of the diagram's components is translated, the relationships between the components need to be maintained. To illustrate the problem, let us consider the following scenario. We want to add an operation to an object, however it overlaps with another operation of the object, and the overlapping occurs between the top portion of the existing operation and the bottom portion of the new operation to be inserted. To solve the overlapping problem, we will translate the existing operation with a certain delta to create space for the new operation.

The problem of translating the operation to a new position, is that the message initiating the operation won't have the translated operation as its receiver. It can have the new added operation as its receiver, or it can have no receiver. The solution is to also translate the message with the same delta to maintain the relationship. If the new

location of the message interferes with other following messages, then the messages are translated and their initiated operations (receivers) are also translated. The process is recursive, because each time you translate a component by a certain delta, we need to check if there is no overlapping. If overlapping occurs, we repeat the process again. When translating messages, it may occur that their new positions will exceed the bottom location of the sending operation, and it may look as if there is no receiver of the message, so we have to enlarge the operation so that it can contain all the messages. In fig 5.3, we illustrate one of the cases that may result when translating diagram components.

5.2.5 Summary

In this section we have discussed some technical issues we faced in our design, and our approach to solve these problems. We have provided a general overview of the tool's functionality. In the next section, we will present the important classes we used.

(1) We are adding a new message, but operation3 overlaps with operation2.

(2) We translated operation2 to create space for operation3. Now m2 has no receiving operation.

(3) We translated m2, to maintain relationship between m2 and operation2

Fig 5.3 A case of translating diagram components

5.3 Coin Classes

We are going to describe the important classes we have used in developing Coin. Some of the classes are abstract ones and their main purpose is to abstract common behavior. Also we will discuss some classes that we modified by adding new methods or by modifying some of the existing methods.

5.3.1 New Classes

5.3.1.1 IDComponent

This is an abstract class that maintains the common characteristics that belong to any general figure representing interaction diagram components. It describes:

- the location where the figure will be drawn
- the pen color used
- the line style (solid, dashed, dotted,...)
- the line width
- any name associated with the figure
- the text color used to display the name
- the text font.

There are no instances created of *IDComponent*. All the elements necessary to construct an interaction diagram are represented as subclasses of *IDComponent*. In fig 5.4, we present the general hierarchy of *IDComponent*. We have chosen the names of the classes based on the naming convention used by Jacobson [Jacobson 93] when describing the different elements of the diagram. The *IDComponent* classes can be described as entity objects and they act as information holders.

Fig 5.4 IDComponent Hierarchy

5.3.1.1.1 Border

The *Border* class represents an object in the diagram. It maintains a list of all the operations associated with the object. It also describes whether the object used in the diagram is a class or an instance of a class. In the case of an instance, information will be required to provide the class of the object. The information about the object type is very important, especially in the case of code generation, because by knowing the object type we will be able to distinguish between the instance and class methods. Inquiries about the existence of a certain operation, or the number of available operations are also provided. The *Border* object also checks if any overlapping exists between a new operation and the existing operations. This capability is very

important, because when creating a new operation, enlarging an operation, or moving an operation, we do not want figures to overlap.

5.3.1.1.2 Operation

An operation represents the execution time of a certain action, and it is associated with an object and initiated by a message. The *Operation* object keeps a list of all the outgoing messages. The class provides information about the existence of a certain message and the number of messages sent. It also keeps information about the border object that the operation is associated with.

5.3.1.1.3 SystemBorder

The class *SystemBorder* represents the system border that depicts the outside world of the system. We have decided to subclass it from *Operation*, because we consider that the system border is a specialized case of an operation. The difference between the two is that the system border is not associated with any border object as in the case of an operation.

5.3.1.1.4 Stimuli

A stimulus represents an interaction taking place between two objects. According to Jacobson, a stimulus is either a message (a normal Smalltalk method) or a signal that represents an outside message, for example an input from a hardware device. A message is modeled by an horizontal closed arrow, while a signal is modeled by an horizontal open arrow. The *Stimuli* object is an abstract class that groups common attributes such as the direction of the message (right or left), and the receiver of the message. It also keeps track of the object sending the message.

5.3.1.1.4.1 IDMessage

The class *IDMessage* contains the methods necessary to draw a horizontal arrow with closed edges ( ) . It inherits behavior from both *Stimuli* and *IDComponent*.

5.3.1.1.4.2 IDSelfMessage

A self-message is a message that is characterized by having the sender and the receiver be the same. It is illustrated by having an horizontal arrow starting and ending at the sending or receiving object ( ) . We decided to create a subclass of *IDMessage* instead of having to include a special method for displaying a self-message in class *IDMessage* . We wanted to provide more generality and flexibility in case of any further additions or modification.

5.3.1.1.4.3 Signal

The class *Signal* contains the methods necessary to draw a horizontal open arrow ( ) . It inherits behavior from both *Stimuli* and *IDComponent*.

5.3.1.2 Tool

The *Tool* class is an abstract class, and all the tools used by the interaction diagram editor will be subclasses from it. All the tools will respond to certain events that will result in creating the designated component. There are certain attributes that are necessary to be recognized by any tool to be able to create the required figure. The desired attributes include the forecolor to display the figure, the line style and

line width, and the font for displaying any labels. In fig 5.5, we present the *Tool* class hierarchy.

Fig 5.5 Tool Hierarchy

When a tool is selected, the attributes are initialized with default settings. However, if the component to be created by the tool already exists in the diagram, the tool will be initialized with the component attributes. In fig 5.6, we describe the collaborations graph of activating any tool. The *DiagramEditor* object checks if the component is present in the diagram, to decide the set of settings needed for the tool initialization. Then, the tool is activated.

Fig 5.6 Tool activation collaborations graph

5.3.1.2.1 BorderTool

The function of the *BorderTool* class is to create a border column with the label '?' associated with it. The height of the column is always constant. The location of the new column is calculated by adding a constant width value to the location of the last border column available in the diagram. So, when the border tool is activated, the *DiagramEditor* object provides the tool not only with the necessary attributes, but also with the next available location.

5.3.1.2.2 OperationTool

The *OperationTool* class allows the user to create an operation that is represented in the form of a rectangle. An operation is associated with a column that is determined by the user. The width of the rectangle is constant, while the length is controlled using the mouse movement. When the tool is activated, it is supplied with the pane where the operation will be drawn. The reason is to provide the facility for scrolling the pane when needed.

2 of/de 2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

5.3.1.2.3 StimuliTool

The activities involved in creating a signal component or a message component are similar, and we have summed up these operations in the *StimuliTool* object. The object maintains the sender and the receiver of the stimulus (event). When the sender and the receiver are decided by the user, the direction of the connection is determined. Now, all the information needed to create the desired stimulus is available, however we need to make sure that we are not interfering with other objects in the diagram, or performing an illegal connection. The *ConsistencyChecker* object is invoked to check for any irregularities, and we will discuss that class later on in the chapter. If no irregularity is found, the user will define the label describing the event taking place.

5.3.1.2.3.1 MessageTool

The *MessageTool* class inherits behavior from the *StimuliTool* class. The function of the tool is to create a message stimulus. If the sender and the receiver are the same, then a self-message component will be created. Otherwise, a message component will be created.

5.3.1.2.3.2 SignalTool

The *SignalTool* class inherits its behavior from the *StimuliTool* class, and its function is to create a signal stimulus.

5.3.1.2.4 SelectionTool

The *SelectionTool* object provides facilities to select diagram objects, and to drag and drop selected objects at a certain location. When objects are selected, a selector in the form of a dotted rectangle will appear surrounding the selected objects. After dropping the chosen objects at the new location, the *DiagramEditor* object checks the new location to ensure legal movement. We do not want objects lying around with no connections between them. For example, we do not want to place the operation anywhere, we need to make sure that it is associated with a column, and afterwards we need to check if all the messages are satisfied, i.e., the receivers of all the messages are known. We do not want to have a diagram that has outgoing message with no end receivers.

5.3.1.3. DiagramEditor

The *DiagramEditor* object controls all the activities performed by the interaction diagram editor. It keeps the following information:

- a list of the created objects
- a list of available tools
- current active tool
- constant values for the column spacing and height
- color palette

The main functionality of the object is to:

- create diagram components
- cut and paste figures
- undo editing operations

5.3.1.4 ObjectCollaborations

The *ObjectCollaborations* class is a subclass of *ViewManager*. It utilizes the information in the diagram to extract the users and helpers of every object defined. We have discussed the process of generating the users and helpers of an object in the previous chapter.

The object provides the following functions:

- extract collaborations from the diagram
- display collaborations
- answer a users dictionary upon request, where the object name is the key and the list of users is the value
- answer a helpers dictionary, where the object name is the key and the value is a list of collaborators

5.3.1.5 ConsistencyChecker

The *Consistency Checker* object provides two main functions:

- An on-line consistency checking
- An overall consistency checking.

An on-line consistency checking is invoked when drawing an operation to insure that operations do not overlap with each other, and when creating a stimulus to avoid any illegal connections between objects. An example of the different irregularities detected by the on-line consistency checker is illustrated in fig 5.7. An overall consistency can be requested by the user to spot any impropriety in the diagram such

as having an outgoing message from an operation, associated with a certain object, that is not initiated by an incoming message.

Fig 5.7 On-line Consistency Checking

5.3.1.6 StaticAnalyzer

The *StaticAnalyzer* object is responsible for the static analysis of a method source code. It needs to know the method name, the class including the method , and the method kind (class or instance). The main functions of the object are to:

- extract the different messages and their receivers
- add the analysis information to IDObjects
- determine the possible execution paths.

The collaboration graph of *StaticAnalyzer* is illustrated in fig 5.8.

Fig 5.8 StaticAnalyzer Collaborations Graph

5.3.1.7 DiagramModifier

The *DiagramModifier* object is responsible for the automatic update of the interaction diagram. The activities involve:

- adding a new stimulus (message) to an existing operation
- adding an operation to an existing object
- adding a new column object
- removing an existing stimulus

During the activities of adding new components to the diagram, it will often occur that an operation needed to be enlarged, messages or operations that needed to be translated. Another function of the class is to shuffle the different components of the diagram to accommodate the new modifications. We will discuss this problem in more detail in the following chapter.

5.3.1.8 MethodInfo

The *MethodInfo* object is an information holder. It is used by the *CodeAnalyzer* object to keep information about the message calls extracted after analyzing a method source code. The object holds information about:

- the name of the message
- the receiver object of the message
- the receiver string of the message as extracted from the code
- whether the receiver of the message is mapped to a Smalltalk class or not

In fig 5.9, we present *MethodInfo* objects ensemble.

Fig 5.9 *MethodInfo* Objects Ensemble

5.3.1.9 ObjectInfo

The *ObjectInfo* class is an object information holder. It keeps a list of the different users and helpers of an object. It also keeps a list of the different methods analyzed by the *CodeAnalyzer* object. Each method analyzed is also associated with the information extracted from the code analysis. By keeping track of the users and helpers of an object, we can have the possibility of maintaining information of more than one diagram at the same time and to use that information to get a view of the whole system. This function is not provided by *Coin*, but in design we considered further enhancements. *ObjectInfo* objects ensemble is shown in fig 5.10.

Fig 5.10 *ObjectInfo* Objects Ensemble

5.3.1.10 IDDictionary

The *IDDictionary* class is a subclass of *Dictionary*, and is used as a repository to keep the information generated from the analysis of the code of the different methods. Our decision to maintain information, instead of generating it each time when requested, is to retain the mapping of the receivers of the different messages. Otherwise, any previous mappings will be lost, and the user has to repeat the mapping process all over again. The key of the dictionary will be the name of the class, while the value will be an instance of an *ObjectInfo*. We are using a global variable *IDObjects* that is an instance of *IDDictionary* to store the analysis information.

The purpose of the class is to provide special functions to retrieve information from the dictionary. Some of these functions include:

-
- users of a certain class
 - helpers of a certain class
 - possible execution paths of a certain method in a class
 - analysis information of a certain method in a class

The class also provides methods for adding information to the dictionary and for removing information from the dictionary.

5.3.1.11 CodeGenerator

The *CodeGenerator* object provides two main functions:

- It acts as a browser by displaying class information
- It generates Smalltalk constructs

In the browser view, the *CodeGenerator* object displays the variables and methods of an object using the Smalltalk system dictionary. A text pane is also provided that will contain the code of any selected method. The code can be edited, and any changes can be saved.

In code generation, the information in an interaction diagram is utilized to create classes that are added to the Smalltalk class library. All the instance and class methods are extracted and added to the new class methods list. The collaborations graph of *CodeGenerator* is shown in fig 5.11.

Fig 5.11 CodeGenerator Collaborations Graph

5.3.1.12 CRCManager

The *CRCManager* object is responsible for providing the user with a CRC card view for every object in an interaction diagram. It keeps a dictionary representing the cards. The key of the dictionary is the object name, and the value is a *CRCCard* object. Each time the CRC manager is activated, it is supplied with a dictionary of collaborations (key: object name, value: list of helpers). The collaboration information will be compared with the collaborator's part of every object CRC card, and any update will be carried out and will be reflected in the CRC card collaborators. The purpose of the comparison is to maintain compatibility between the diagram and the CRC card, and to reflect any modification in the diagram. In fig 5.12 we show the collaborations graph of a CRC manager activation.

Fig 5.12 CRCManager Collaborations Graph

5.3.1.13 CRCCard

The *CRCCard* object is an information holder that keeps information about the object name, the object collaborators and the object responsibilities. Fig 5.13 illustrates *CRCCard* objects ensemble

Fig 5.13 CRCCard Objects Ensemble

5.3.1.14. ColorsPalette

The function of the *ColorsPalette* object is to display a color palette that can be used to alter the colors of the different diagram components. The color is selected using the mouse buttons. The color selected using the right button will be used to modify the color of the different components while the colors selected using the left button will be used to modify the labels associated with the components. The *DiagramEditor* object utilizes the chosen color to alter the display of the components pertinent to the active tool. So for example, if the active tool is the operation tool, then only the operation components will be modified.

5.3.2 Modified Classes

5.3.2.1 ClassHierarchyBrowser

The *ClassHierarchyBrowser* provides a view of the class library and enables the programmer to make additions or modifications to the class library. We have added few lines to three methods:

- **accept:** This method is used to save the contents of the text pane after compiling the contents successfully. The method source code is then updated in the Smalltalk system dictionary. We check the existence of the method in IDObjects, and if it exists, we regenerate the analysis information to reflect the new code changes. Our incentive is to keep the saved analysis information up to date with the source code.
- **removeSelector** The function of the method is to remove a selector (method name) from the class selectors (methods) list. After removing the method from

the list, we check IDObjects to see if we have analysis information for the method so that we delete it.

- **removeSubclass** It removes a class from the Smalltalk class hierarchy. We search the keys of IDObjects for the deleted class, and if it is found, the class key is removed from the dictionary.

5.3.2.2 Stream

We are using the class in the code analysis process. We have added four new methods that we needed to accomplish the task.

- **nextString** It answers the next string in the receiver stream. The end of the string is determined when a separator (space, tab, carriage return, line feed) or end of stream is encountered.
- **previousString** It performs the reverse of the previous method, and it answers the previous string in the receiver stream.
- **previousWord** It answers a string containing the previous word in the receiver stream. A word starts with a letter that is followed by a sequence of letters and digits.
- **previous** It answers the previous object in the receiver stream.

Chapter 6

Sample Application: An Airline Reservation System

In this chapter we will give an example of using Coin in software development. Our goal is to illustrate the usage of the concepts we have discussed earlier.

6.1 An Airline Reservation System

An airline reservation system is a system that provides flight reservations for passengers. In this chapter, we are going to design a simplified version that illustrates some important features; e.g., booking a passenger on a certain flight for a specific airline.

6.2 Analysis and Design

Object-Oriented systems favors an iterative life cycle where in each iteration activities involve all phases of traditional life cycle. So, the distinction between analysis and design is relatively "blurry". In analysis, the focus is more on what is needed; i.e., system requirements, while in design the focus is on how to implement

the requirements. We are going to use Jacobson's notion of actors and use cases to identify the users and activities of an airline reservation system.

Actors portrays roles played by people (or other systems) interacting with the system under development. In an airline reservation system, the primary actor will be the *clerk* responsible for booking flights for passengers. At the same time, an *administrator* (secondary actor) is needed to maintain and supervise the system. Thus, the picture of the airline reservation system with interacting actors will look like that in fig 6.1.

Fig 6.1 Airline reservation system with interacting actors

In order to identify use cases, we need to investigate what sequence of transactions is needed by the system to fulfill actors' request. We are not going to identify all the use cases, however we will mention some of the primary ones. A *clerk* actor is responsible for booking flights, and interesting use cases that come to mind include:

- Booking a flight for a passenger.
- Confirming a reservation.
- Canceling a reservation.

A *system administrator* actor is responsible for maintaining and supervising the system, and is concerned with performing use cases such as:

- Initializing the system.
- Updating databases.

A use case model of an airline reservation system is shown in fig 6.2.

Fig 6.2 Airline reservation system use case model

We will focus on the following use cases:

- Initializing an airline reservation system.
- Booking a flight

6.2.1 Initializing an Airline Reservation System

In the initialization of an airline reservation system, the aim is to add airlines to our system. Each airline is identified by a name, and keeps track of its scheduled flights. So the use case will be as follows:

- (1) Create an airline.
- (2) Add airline to reservation system.
- (3) Create a flight.
- (4) Add flight to an airline.

The use case involves three main classes: *AirlineReservationSystem*, *Airline*, and *Flight*. The *AirlineReservationSystem* object will keep track of the airlines that it services, while the *Airline* object will keep track of the scheduled flights and its name. A *Flight* object must keep track of its name, the passengers on board, in addition to its starting and destination location, and departing and arrival time and date. Thus, a more detailed use case will be as follows:

- (1) Create an airline.
- (2) Initialize airline name, and flights list.
- (3) Add airline to airline reservation system.
- (4) Create a flight.
- (5) Initialize passengers list, starting and destination location, arrival and departure date and time, and flight name.
- (6) Add flight to airline.

Airline keeps track of its scheduled flights, so when the airline reservation system is asked to add a flight to an airline, it will get the desired airline from its list, then the designated airline will add the flight to its scheduled flights. Thus, the use case will be modified as follows:

- (1) Create an airline.
- (2) Initialize airline name, and flights list.
- (3) Add airline to airline reservation system.
- (4) Create a flight.
- (5) Initialize passengers list, starting and destination location, arrival and departure date and time, and flight name.
- (6) Get airline from airline reservation system.
- (7) Add flight to airline.

Interaction diagrams are used to visualize use cases. We are going to use Coin to build the corresponding interaction diagram of "the initialization of an airline reservation system" use case. To create an instance of *Airline* or *Flight*, we are going to use the message `new` which is a class method used for class instantiation. In fig 6.3, we show the interaction diagram of initializing an airline reservation system constructed using Coin. In fig 6.4, we are showing the printout of the whole diagram using Coin print facility.

Fig 6.3 Initializing an airline reservation system interaction diagram

Coin utilizes the information embedded in the diagram to extract other forms of information such as: CRC cards and collaboration information. In collaboration information the user is supplied with a list of the helpers and users of each object in the diagram. From the diagram we can deduce the following collaborations:

	Helpers	Users
AirlineReservation	<i>Airline</i>	
Airline	<i>Flight</i>	<i>AirlineReservation</i>
Flight		<i>Airline</i>

Fig 6.4 Printout of interaction diagram

In fig 6.5, we are presenting collaboration information of class *Flight*. The user get access to that information when **collaborations** is selected from the **utilities** menu. The user will be presented with a list of the available objects in the diagram, and when any object is selected, the list of its helpers and users will be shown. The information is sensitive to any diagram modification, and the list of helpers and users of each object will be updated to accommodate the diagram changes.

Fig 6.5 *Flight* collaboration information

Collaborations are associated with a responsibility, and a CRC card is used to record the collaborations. From the **utilities** menu, when the **CRC Cards** menu item is chosen, the CRC card manager is invoked. The user can record the responsibilities of each object, while the collaborators part of the card is automatically filled in by Coin. For each object in the diagram, Coin provides the user with a corresponding CRC card. The information in the CRC card is saved with the diagram, and can be recalled later when needed. The collaborators part of each card is updated with any diagram change. In fig 6.6, we show the CRC card of *AirlineReservation*.

Fig 6.6 *AirlineReservation* CRC card

Now that the user has created the desired interaction diagram, and obtained collaboration information, a better understanding of the system has been achieved. It is time to create corresponding Smalltalk classes, and the helpers and list of responsibilities will allow the user to decide on instance variables. Coin provide the user with a code generation facility. For each diagram object that does not exist in the Smalltalk class library, the user is asked to provide the superclass name to decide the placement of the object in the hierarchy. Also, the user has the option to input class and instance variables. The tool will automatically create the class for the user, and "set" and "get" methods are automatically generated for each variable defined. In

addition, all the messages that an object can understand will be extracted from the interaction diagram and added to the object's methods list.

In fig 6.7, we are illustrating the creation of class "Airline". The user specifies the super class name, and the variable names. In our case, the user has defined two instance variables: *name* (airline name) and *flights* (list of scheduled flights). When the user is satisfied with his/her selection, the class *Airline* is created and added to the Smalltalk class hierarchy. Coin, also, provides the user with the capability to browse the class information (list of methods and variables), as shown in fig 6.8. A text pane is also available to show the source code of any message, and the user has the capability to modify the source code.

Fig 6.7 Creating *Airline* class

Fig 6.8 Airline class browser

The process of class creation can be repeated with: *AirlineReservation* and *Flight*. An airline reservation system will keep track of the list of airlines that it services (*airlines*) and the list of passengers (for bonus miles accumulated) (*passengers*). A flight will keep track of its name, the passengers on the flight, the airline it belongs to, and starting point and destination along with the departure and

arrival dates and times. Instead of having to include six variables (starting location, departure date, departure time, destination, arrival date, arrival time), we decided to have only two variables: source and destination. A *FlightEntry* object will be used to store location, date and time. So, instead of having to assign six variables to initial values, one flight entry is created for the source and another one is created for the destination, and each one is initialized with the proper information.

The change in the flight design will affect two diagram methods: *departing:date:time:* and *arrival:date:time:*. Instead of having a simple assignment for the flight variables, two flight entries are created and initialized with: source and destination information. In Coin, if we click on the message *departing:date:time:*, the message shape is changed (line style becomes dotted) to indicate that the message has been selected. The corresponding source code will be shown on the right side, accompanied by the list of messages extracted from the source code. A comparison between diagram messages and source messages is performed, and the result of the comparison is displayed in the upper right side, as shown in fig 6.9.

The comparison indicates that there exist a difference between the diagram messages and source code messages, and we want the diagram to reflect the new changes detected in the code. The automatic diagram facility provided by Coin, modifies the diagram by adding the necessary components needed to maintain consistency with the implementation. At the same time, figures translations are automatically carried out to provide space for the newly added diagram components. The new diagram is illustrated in fig 6.10.

Fig 6.9 Inconsistent design and implementation

Fig 6.10 Consistent design and implementation

The automatic diagram modification will not only affect the interaction diagram, but also will be reflected in the collaboration information and CRC cards. The addition of the new class *FlightEntry*, will modify the collaboration information of class *Flight* and its corresponding CRC card. The new added class is considered as one of the collaborators (helpers) of class *Flight*. The new collaboration information of *Flight* will look like that in fig 6.11.

Fig 6.11 New collaboration information of class *Flight*

The automatic diagram update can be also applied to the message *arrival:date:time:* to maintain its synchronicity with the implementation. The user can modify the source code of any message, and utilize the automatic diagram update to monitor the effect of the change on the diagram. The automatic diagram modifier includes a feature that enables the user to select messages of interest to be used to modify the diagram, and as in many cases, the user is interested in certain messages, and he/she does not want the unimportant messages to be reflected in the diagram

Fig 6.12 Code information of *addAirline:* in *AirlineReservation*

Let us consider the source code of the instance message *addAirline:* sent to *AirlineReservation* (fig 6.12). The code contains two messages: *name* and *at:put:*. As we are analyzing the code statically, we were not able to determine the receivers of the two messages. Coin provide the user with a mapping facility where the user is given a list of all the objects that can understand a certain message, and the user will determine the appropriate receiver (object) name. The process is shown in fig 6.13. In our case, the receiver of message *name* will be an *Airline* object, while the receiver of message *at:put:* will be a *Dictionary* object.

Fig 6.13 Mapping message receivers

Once the mapping is completed and the receivers of the messages are determined, the process of updating the diagram can be initiated. Let us consider the following scenario: the user is only interested in the message *name* sent to airline. In order to achieve the desired goal, the user has to selected the message(s) that he/she is interested in, and the automatic diagram modifier will only use the selected list of messages to update the interaction diagram. In fig 6.14, we show the effect of selecting the message *name* to update the diagram.

Fig 6.14 Updating interaction diagram using selected messages

6.2.2 Booking a Flight for a Passenger

The *clerk* actor will be responsible for booking a passenger on a certain flight. When asked to book a flight, the clerk needs to know information needed to accomplish his/her task such as airline name, flight name and date. Thus, the use case will be as follows:

- (1) Create a passenger.
- (2) Get airline according to airline name.
- (3) Get flight according to name and date.
- (4) Book passenger on flight.

In the above use case, we have introduced a new class *Passenger* that keeps information about a passenger name, phone, and address. So, when creating a passenger, we need to initialize the passenger information. When booking a passenger on a certain flight, the flight has to add the booked passenger to the list of passengers on board. Thus, the modified use case will look as follows:

- (1) Create a passenger.
- (2) Initialize passenger name, phone, and address.
- (3) Get airline according to airline name.
- (4) Get flight according to flight name and date.
- (5) Book passenger on flight.
- (6) Add passenger to flight list of passengers.

The same process can be followed as illustrated in the former use case. Our goal in presenting this use case is to show the benefits of the alternative execution paths provided by Coin. In fig 6.15, we are presenting a printout of a partial interaction diagram for booking a flight for a passenger. A message for booking the passenger on a flight (adding passenger to the passengers list on the flight) is needed to complete the use case.

Fig 6.15 Partial interaction diagram for booking a passenger on a flight

We are going to modify our design slightly by adding a waiting list variable (`waitingList`) to class `Flight`. The purpose of the new variable is to keep a list of waiting passengers when the flight is fully booked. The use case will be modified to accommodate the new design. If the flight is fully booked, the passenger will be added to the waiting list, otherwise the passenger will be added to the list of passengers on the flight. We have modified the source code of the message `book:airline:name` to look as shown in fig 6.16.

Fig 6.16 Implementation information of *book:airline:flight:date:*

In fig 6.16, we notice that the list of extracted messages from the source code does not include the message *book:* sent to *flight*. The reason is that *book:* belongs to a different execution path. Coin provides the user with a list of extracted message belonging to a certain execution path, and not just the list of all the available messages. When the **AlternativePath** button is clicked, the user gets access to another available execution path. The exposure to different execution paths is very beneficial because it enables the user to generate interaction diagrams for special (alternative) cases that is accomplished by only modifying the interaction diagram for the general case. In our case, the user is interested in showing the interaction diagram

of successfully booking a flight for a passenger. The execution path for booking a passenger will be selected and used to modify the diagram. The interaction diagram for successfully booking a passenger is shown in fig 6.17.

Fig 6.17 Interaction diagram for booking a flight

6.3 Summary

We have developed a simplified airline reservation system using Coin, and concentrated on two specific use cases: *initializing an airline reservation system* and *booking a passenger on a flight*. We created an interaction diagram for each use case,

and utilized the different facilities provided by Coin such as: code generation, and CRC cards. We also modified the source code of certain messages and examined the effect of the new changes on the diagrams

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The evolution of OOP in the late 80's, and the fact that the object-oriented life cycle is much different from the traditional waterfall model has led to the development of various OO design and analysis methodologies and tools that support these methodologies. Unfortunately, the majority of these tools support software development in the top-down direction only, and are not resilient to changes; i.e. , a change during the implementation phase requires restarting the top-down process somewhere higher in the life cycle. As a result, the implementation is often out of synchronization with the design. The object-oriented life cycle is an iterative process and it is necessary to provide an environment that supports bi-directional software development in order to integrate analysis, design, and implementation. Our goal was to develop a tool that supports both top-down and bottom-up development. To meet our objectives we designed and implemented Coin.

Coin was aimed to provide the following facilities:

-
- **Compare interaction diagrams (design) with the actual source code (implementation), and to facilitate an automatic diagram update if any inconsistency is discovered between design and implementation.**
 - **Utilize the richness of information included in the interaction diagram, and to present that information in different forms to the user.**

We have developed Coin following three main themes:

- **Providing the user with the ability to view and modify interaction diagrams**
- **Representing implementation information in different forms (source code and message calls)**
- **Updating the diagram automatically without altering the relationship between the different components of the diagram**

Coin provides the following facilities:

- **Interaction diagram editor for diagram construction and modification**
- **Useful information in the form of collaborations, CRC cards, object information (class/instance variable, class/instance methods, method code), and analyzed code information.**
- **Automatic diagram update based on comparison between messages in an interaction diagram and the source code**
- **Code generation facility**

Although Coin incorporates several ideas from other tools and systems, Coin is unique in the following ways:

- **The interaction diagram editor is not just an editor for constructing diagrams, but also includes consistency checking facility both on-line and overall, to respond to any meaningless actions.**

- We are providing an automatic diagram update, that will automatically update the diagram if any differences is found between the diagram messages and code messages. The automatic diagram modifier will move the components around if needed to create space for the newly added components to be added without changing the relationships of the existing components. The user also has the option to partially use the list of the source code messages as a basis of comparison instead of the whole list, if in some cases he/she is not interested in certain messages.
- We are providing a static analysis facility that will extract the different messages from the code. The order of the messages is not random, but depends on their order of execution according to the Smalltalk rules.
- We are providing the user with alternative execution paths.

Coin provides useful facilities that can be used in software development and maintenance. The automatic diagram modifier is very beneficial because it allows the user to update an already constructed design without the need to add or remove components manually. Coin can be used as a prototype tool where we can start by drawing interaction diagrams, then get a view about the objects dependencies by looking at the collaborations. Then we can do some coding, see how it will affect the diagrams, and how the diagrams will change when different execution paths are selected.

7.2 Future Work

Coin does not provide an on-line help system. On-line help aid in understanding how the system works, and in getting information about the steps to be followed if a certain function is desired. An example of how help is very important is in the case of Coin's colors pallet. The pallet modifies the color of a diagram component corresponding to the current active diagram tool. In other words, if the message tool is activated, then only the color of the messages will be modified when a new color is selected.

Many enhancements can be incorporated in the interaction diagram editor. One enhancement that will be very important is a zoom facility. The main view of Coin is divided into two portions: one for the editor and one to display the information extracted from the code. When an interaction diagram gets very complex, it is very difficult to get a view of the whole diagram, and a lot of scrolling is needed. It will be nicer to have a zooming facility that enables the user, among other things, to get an overview of the whole diagram.

Another feature that can be used in documentation is a facility that can decide on the structure of an interaction diagram, and whether it is a centralized structure, a decentralized structure, or a combination of both. Also, it will be very helpful to show the part of the diagram that is centralized, and the part that is decentralized. This information can be useful in program documentation, and also it will give some insight on the design structure.

We are using a static analysis approach to get implementation information. As Smalltalk is a dynamic language, if a dynamic analysis facility were added, we will

have more flexibility especially when obtaining the receivers of the messages, and we would not have to involve the user with the task of mapping the receivers to the messages. Thus, if implementation is complete we can use dynamic analysis to get information about messages and their receivers, and use static analysis to get information about the possible paths of execution for each message.

Currently, the user is only able to access one diagram at once. Another useful enhancement would be to have the capability of combining more than one diagram, and to extract information from more than one diagram. By extracting information from more than one diagram, we will be able to recognize subsystems which can be used as a basis for development by different project teams. We can also obtain an overview of the whole system in the form of collaboration information, and that information can be incorporated in CRC cards.

Interaction diagrams are a visualization of use cases. A useful avenue of research would be to devise a tool that uses a certain grammar language where we can define the different use cases of our system. The definition of the different use cases can be later used by Coin to automatically build the corresponding interaction diagrams. At the same time, we would provide a reverse engineering facility, where any changes in the diagram would be reflected in its corresponding use case.

Bibliography

[Boker 91]

Boker, H-D., Fisher, G., and Nieper-Lemke, H. "The Role of Visual Representations in Understanding Software". In: *Artificial Intelligence and Software Engineering*, D. Partridge(ed.). Ablex, Norwood, N.J., pp. 273-290.

[Booch 93]

Booch, G., Fowler, M., Goldberg, A., and Rubin, K. "Object-Oriented Analysis and Design, Finding your Path". *SIGS Publications* 1993.

[Booch 94]

Booch, G. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 1994.

[Brown 85]

Brown, G.P., Carling, R.T., Herot, C.F., Kramlich, D.A., Souza, P. "Program Visualization: Graphical Support for Software Development". *I.E.E.E. Computer* 18(8), August 85, p. 27-35.

[Budd 91]

Budd, Timothy. "An Introduction to Object-Oriented Programming". Addison Wesley Publishing Company, 91.

[Buss 92]

Buss, E. and Henshaw, J. "Experiences in Program Understanding". *Technical Report TR-74.105*, IBM. 35 pages. July 1992.

[Coad 91]

Coad, Peter and Yourdon Edward. *Object-Oriented Analysis*. Prentice Hall, Inc., 1991.

[Chikofsky 90]

Chikofsky, E.J., Cross II, J.H. "Reverse Engineering and Design Recovery: A Taxonomy". *I.E.E.E Software* vol. 7, no.1, p. 13-17, January 1990.

[Citrin 90]

Citrin, W., Cockburn A., Hauser, R.F., and vonKanel, J. "An Environment for Interactive Design of Communications Architectures". *Tenth International Symposium on Protocol Specification, Testing , and Verification*, June 1990.

[Citrin 94]

Citrin, W. and Cockburn A. "Carla: A Rule Language for Specifying Communications Architectures". *Technical Report CU-CS-742-94*. University of Colorado at Boulder. 38 pages, 1994.

[Cunningham 86]

Cunningham, W. and Beck, K. "A Diagram for Object-Oriented Programs". *OOPSLA'86 Conference Proceedings* 21(11), November 1986. pp. 361- 367

[Fowler 92]

Fowler, Martin. "A Comparison of Object-Oriented Analysis and Design Methods". *ObjectWorld*, December 1992.

[Jacobson 92]

Jacobson, Ivar. "Object-Orientation as a Competitive Advantage". *American Programmer*, October 92, vol 5, no.8.

[Jacobson 93]

Jacobson, Ivar. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Publication Company, 1993.

[Lalonde 94]

Lalonde, Wilf. *Discovering Smalltalk*. Benjamin Cummings, 1994.

[Myers 88]

Myers, Brad. "The State of the Art in Visual Programming and Program Visualization". *Technical Report CMU-CS-88*, Carnegie Mellon University. 25 pages, February 1988.

[Objectory 93]

Objectory Support Environment: A Technical Overview. Objectory Corporation, Greenwich, Connecticut, 1993.

[Pugh 94]

Pugh, J., Lalonde, W., White, P., Freeman-Benson, B., and Corriveau, J.P.
"Towards Coupling Analysis, Design, and Implementation". School of
Computer Science, Carleton University, 1994.

[Raeder 85]

Raeder, George. "A Survey of Current Graphical Programming Techniques".
I.E.E.E. Computer 18 (8), August 1985, pp. 11-25.

[Rational Rose 94]

Application Development with Rational Rose C++. A Technical Overview.
Rational Software Corporation, Santa Clara, Ca., 1994.

[Rumbaugh 91]

Rumbaugh, J., Blaha, P., Premerlani, W., Eddy, F., Lorensen, W.
Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, N.J.,
1991

[Shafer 93]

Shafer, Dan. *Smalltalk Programming for Windows.* Prima Publishing,
Rocklin, California, 1993.

[Wang 93]

Wang, Yun. *Interaction Diagram Generator.* Master thesis, School of
Computer Science, Carleton University, 1993.

[Wirfs-Brock 90]

Wirfs-Brock, R., Wilkerson, B., and Wiener, L. *Designing Object-Oriented Software*. Prentice Hall, 1990.

END

1 6 - 1 1 - 9 5

FIN