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

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.

Canada

Interaction Diagram Generator A Tool for Visualizing Use Cases in Smalltalk

by

Yun Wang

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

April 11, 1994

© Copyright

1994, Yun Wang

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

Your title Votre référence

Our title Notre 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 accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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 propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-92934-0

Canada

Name YUN WANG

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

0984

SUBJECT CODE

U·M·I

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

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 0278
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0477
Range Management 0777
Wood Technology 0746
Biology
General 0305
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0379
Ecology 0329
Entomology 0353
Genetics 0369
Limnology 0793
Microbiology 0416
Molecular 0307
Neuroscience 0317
Oceanography 0416
Physiology 0432
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 System Science 0554
Geotechnical 0790
Operations Research 0428
Plastics Technology 0796
Textile Technology 0795
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.

Interaction Diagram Generator
A Tool for Visualizing Use Cases in Smalltalk

submitted by

Yun Wang

in partial fulfilment of the requirements
for the degree of Master of Computer Science

A handwritten signature in cursive script, appearing to read "T. R. P. J.", written over a horizontal line.

Director, School of Computer Science

A handwritten signature in cursive script, appearing to read "T. R. P. J.", written over a horizontal line.

Thesis Supervisor

Carleton University

April 11, 1994

Abstract

Recently, object-oriented technology has emerged and it is being more and more widely accepted. Object-oriented systems work through sending messages among objects. When the number of objects in a system becomes larger, however, it can be difficult to understand how the system works and what is the relationship between objects.

Use cases and Interaction Diagrams are useful tools to help address the above problem. A use case is an example usage of a system. Interaction Diagrams are used to describe how each use case is satisfied by communicating objects. Use cases and Interaction Diagrams can be used to describe the system design before the system is built or to review the dynamic behavior of a running system.

In this thesis we describe an Interaction Diagram Generator - a tool that automatically generates the Interaction Diagrams when use cases are executed. The Interaction Diagram Generator assists in understanding the dynamic behavior of object-oriented systems and can be used to extract the design information from an existing system. It can also be used to validate implementation against design documentation and regenerate design documents. In addition, the tool can be used as a debugging tool since it provides the programmer with an accurate and easy to understand picture of the execution flow.

Acknowledgements

This thesis could not have been completed without the guidance of my supervisor, Prof. John Pugh. In particular, I would like to thank him for all he has taught me and for his patience and perseverance. My thanks also go to Prof. Wilf Lalonde and the other people in the Centre for Object-Oriented Programming (COOP) in School of Computer Science for their assistance.

Financial support from School of Computer Science, and COOP made this work possible.

Finally, I would like to thank my husband, Marvin, for his support and assistance on this thesis, and my little boy, Kevin, who has endured the hardships of being the son of a student. I would also like to thank my parents for taking care of Kevin those days.

Contents

Abstract	iii
Acknowledgements	iv
List Of Figures	ix
1 Introduction	1
1.1 Object-Oriented Technology	1
1.2 Use Cases and Interaction Diagrams	4
1.3 CASE Tools	5
1.4 The NSERC Project	5
1.5 Thesis Overview	6
2 Background	9
2.1 Object-Oriented Systems	9
2.2 Use Cases and Interaction Diagrams	11
2.2.1 Use Case	11
2.2.2 Interaction Diagram	12
2.3 CRC cards and Collaborations Graphs	14
2.3.1 CRC Cards	15
2.3.2 Collaborations Graphs	16
2.4 Object-Oriented System Development Methods	17

2.4.1	Responsibility-Driven-Design	17
2.4.2	Object-Oriented-Software-Engineering	19
2.5	Summary	23
3	Related Work	24
3.1	Reverse Engineering	24
3.2	Understanding Object-oriented Systems	25
3.2.1	Cunningham Diagrams	26
3.2.2	GraphTrace	28
3.2.3	Track	31
3.3	Summary	31
4	Interaction Diagram Generator	34
4.1	The Interaction Diagram Generator Panes	36
4.1.1	Class Pane	38
4.1.2	Method Pane	38
4.1.3	Use Case Pane	38
4.1.4	Diagram Pane	38
4.2	Method Browser and Collaborations View	39
4.2.1	Method Browser	39
4.2.2	Collaborations View	41
4.3	Debugging Use Cases	43
4.4	Other Features	43
4.4.1	Filtering	43
4.4.2	Save, Load, and Print Interaction Diagrams	44
4.4.3	Execution Modes	45
4.4.4	File Menu and View Menu	45
4.4.5	Coloring The Operations	46
4.5	Summary	46

5	The Design and Implementation of IDG	48
5.1	The Design Principles	48
5.1.1	The Creation of <i>IDGDebugger</i>	50
5.1.2	Generating Interaction Diagrams	52
5.2	The Interaction Diagram Generator Classes	53
5.2.1	The IDGDebugger Class	53
5.2.2	The Application Class	56
5.2.3	The UseCase Class	56
5.2.4	The Item Class	58
5.2.5	The Operation Class	58
5.2.6	The Column Class	58
5.2.7	The InteractionDiagram Class	59
5.2.8	The ScalePrompter Class	59
5.2.9	The InteractionDiagramPane Class	59
5.2.10	The IDGProcess Class	60
5.2.11	The InputEvent Class	60
5.2.12	The UseCaseView Class	61
5.2.13	The CollaborationsView Class	61
5.3	Summary	61
6	Sample IDG Application	62
6.1	The Usage of IDG	62
6.1.1	Describing an Application	62
6.1.2	Describing a Use Case	63
6.1.3	Open an IDG	65
6.1.4	Filters	65
6.1.5	Bring Up Collaborations View	69
6.1.6	Opening a Method Browser	69

6.1.7	Saving and Loading an Interaction Diagram	69
6.1.8	Printing an Interaction Diagram	69
6.2	Using IDG to Analyse A Simple Bank Account Program	70
6.3	Summary	72
7	Summary and Future Work	75
7.1	Conclusion	75
7.2	Future Work	76
A	Smalltalk Code for A Simple Bank Account Program	79
B	Interaction Diagrams for Simple Bank Account System	85
	Bibliography	90

List of Figures

1.1	A Typical Waterfall Model	3
1.2	Object-Oriented Software Development Process	4
2.1	Interaction Diagram	13
2.2	Structure of Interaction Diagram	14
2.3	A Class-Responsibility-Collaborator card	15
2.4	Collaborations Graph	16
2.5	A Simple Hierarchy Graph	18
2.6	The models and associated processes	20
2.7	Use Case Model	21
3.1	Cunningham Diagram	27
3.2	Component Graph	29
3.3	Method Invocation Graph	29
3.4	GraphTrace	30
3.5	The Track System	32
4.1	Interaction Diagram Generator	37
4.2	Activating a Method Browser in IDG	40
4.3	Collaborations View	42
4.4	The File Menu	46
4.5	The View Menu	46

5.1	IDG Overview	49
5.2	IDG Class Hierarchy	54
5.3	The Structure of Collaborations	57
6.1	Use Case Model for Simple Bank Account System	63
6.2	Filter on Class and Method (with on-off indicator)	66
6.3	Filter on Class and Method (without on-off indicator)	67
6.4	Hiding Operations On a Interaction Diagram	68
6.5	Collaborations on BankAccount	72
6.6	Collaborations on Transaction	73
6.7	Collaborations on DepositTransaction	73
6.8	Collaborations on CheckingTransaction	73
6.9	Collaborations on WithdrawalTransaction	74
B.1	Open an Account	86
B.2	Make a Deposit	87
B.3	Write a Check	88
B.4	Make a withdraw	89

Chapter 1

Introduction

1.1 Object-Oriented Technology

Developing software systems is a relatively young but fast growing industry. Applications built today are far larger and more complex than they were ten or twenty years ago. They are complex because they model the complexity of the real world. The requirements of a system often change during and after system development. Changes and extensions occur many times during system development and maintenance. One potential problem associated with these changes is that modifications are difficult to isolate. Each bug fixed may introduce numerous other bugs. Each new component might interact with every other component already in the system.

In recent years, object-oriented technology has emerged as a promising new approach to software development. The object-oriented approach attempts to manage the complexity inherent in real-world problems by abstracting out knowledge, and encapsulating it within objects. Object-oriented design encourages a view of the world as a system of cooperating and collaborating agents. Work is accomplished in an object-oriented system by one object sending a request to another to perform one of its operations, reveal some of its information, or both. Object-oriented technology promotes the development of systems that are more resilient to change, and

more likely to produce reusable components - components which may be reused in a different context or project thereby reducing development time.

The primary benefit of object-oriented technology is its ability to manage complexity. It supports abstraction and encapsulation. A system is decomposed into objects. Each object is an abstraction of one part of the system. All the details of an object in the system are hidden. This helps to minimize rework when developing a new system and minimizes traffic between different parts of the system. Other features include inheritance and polymorphism. Inheritance allows a designer to specify common attributes and services once, and specialize and extend them for specific cases. Polymorphism allows the overloading of names. Hence the same name can be used throughout a system to denote a commonly used and well-understood operation.

The potential benefits of object-oriented technology include improved software quality, shorter development time, increased programmer productivity, and better software reusability. It has great potential as an improved way of developing software systems.

Several new object-oriented programming languages have been developed. Some of them are object-oriented extensions of traditional procedural languages like C++ and Objective-C which are extensions of C, and Flavors and Loops which are extensions of Lisp. These languages are hybrid languages that support objects, classes, and class inheritance, but also include intrinsic data types (e.g. integers, strings) that are not objects. Others, like Smalltalk and Eiffel, are not extensions of any languages. These languages are pure object-oriented languages in the sense that everything is an object.

There are now numerous object-oriented analysis and design methods for applying object-oriented technology to software development. Commonly used methods include Object-Oriented Software Engineering (COSE) - A Use Case Driven Approach [Jaco 92], Responsibility-Driven Design [WWW 90], Object-Oriented Design [Booch 91], Object Modeling Technique [RBPEL 91] and Object-Oriented Analysis [CY 91], etc..

Figure 1.1: A Typical Waterfall Model

Object-oriented software development necessitates changes in the software life cycle. The object-oriented software development life cycle does not follow the traditional waterfall model (Figure 1.1) where the developers try hard to do everything right the first time, and the software development is viewed as decomposed into discrete phases. Changes are often only identified in later phases of the life cycle causing untimely delays and requiring major re-analysis, re-design and re-implementation efforts. The waterfall model does not accommodate change, but change is a fundamental part of any large software development.

Object-oriented software development is an iterative, incremental process, see figure 1.2. An object-oriented system evolves over time through a succession of deliverables (or prototypes). Each deliverable builds on the previous one by adding additional functionality. Each deliverable involves several iterations through all the phases in the traditional life cycle.

Figure 1.2: Object-Oriented Software Development Process

1.2 Use Cases and Interaction Diagrams

Use cases are at the heart of Jacobson's OOSE [Jaco 92] methodology. A use case is a specific way of using a system by exercising some part of the required functionality. Use cases provide a concrete representation of software requirements, which allows them to be both formally expressed and systematically tested. Changes in requirements map directly onto changes in the set of use cases. Use cases can provide better traceability of design decisions.

Interaction diagrams in OOSE are used to describe how each use case is satisfied by communicating objects. Each use case has an interaction diagram which shows the relevant message traffic and relevant objects involved in the use case.

We will discuss use cases and interaction diagrams in more detail in chapter 2.

1.3 CASE Tools

When large systems are developed, all documentation must be consistent. This is not easy because there are many documents and many changes are made during system development. To manage this complexity, tools can be used to automate a large number of tasks, especially trivial ones of cumbersome volume. If the tasks in the underlying process are seamlessly connected, a great deal of the work can be automated. Computer Aided Software Engineering (CASE) tools can lead to large productivity improvements, but they cannot ensure that you achieve a good design or a quality product. They are just tools.

A number of CASE tools have been developed to support specific object-oriented system development methods. *OMTool* is a graphic editor for Object Modeling Technique [RBPEL 91]. *OOATool* is the tool for Peter Code's Object-Oriented Analysis [CY 91]. *Rational Rose* is a graphical tool to support Booch notation in Object-Oriented Design [Booch 91], and *OrySE* is for OOSE. All of these are tools with graphics facilities. They support the software life cycle in one direction only - top-down; i.e., from analysis to design to implementation. To a large degree the existing tools are intolerant of change. Consequently, a major change in requirements (analysis) or design while in the midst of the implementation phase generally requires the top-down process to be restarted somewhere higher up in the life-cycle hierarchy. There are no mechanisms for smoothly integrating the changes at that level with existing partial developments lower down. Better tools are needed to support re-analysis, re-design, and re-implementation particularly since reuse is a fundamental aspect of the object-oriented technology.

1.4 The NSERC Project

The Center for Object-Oriented Programming (COOP) group in the School of Computer Science at Carleton University is now working on a project supported by NSERC. The long term goal of the project is to develop a methodology and tools that support re-analysis, re-design, re-implementation, and reuse. More specifically,

to develop a system that supports integrated concurrent activity in all three software life cycle phases (i.e., analysis, design, and implementation) and permits any one phase to lag or lead another. The short term objectives are focusing primarily on bottom-up coupling since it is the least understood and least supported of the life cycle phase directions. More specifically, the goal is to develop tools that support design capturing (i.e., The ability to capture both the static and dynamic properties of working software in textual and pictorial form.) and design regeneration (The ability to re-capture both the static and dynamic properties of working software in textual and pictorial form when changes to the system are made.). [LPWC 93]

1.5 Thesis Overview

This thesis addresses the development of a new CASE tool: IDG (Interaction Diagram Generator) that supports the notion of design capturing. IDG can be used to:

- *Visualize (animate) use cases.*

IDG can generate interaction diagrams step by step as each method is executed. These interaction diagrams give a picture of how the use cases is satisfied by communicating objects.

- *Understand the dynamic behavior of a system.*

We can run through all the use cases of a system, and look at how objects are communicated by sending each other messages thereby get an idea of how the system works.

- *Verify the actual implementation of a system against its design.*

Use cases and their interaction diagrams are usually documented during system design. After the system is implemented, we can run through all the use cases and compare the generated interaction diagrams with documented ones to find the differences.

- *Monitor the evolution of a system.*

We can keep all the versions of interaction diagrams for every deliverable of a system by running through all the use cases at the end of each iteration and save them.

- *Capturing additional design information; e.g., generating collaboration information.*

From the interaction diagrams generated by IDG for the use cases of an application system, we can see the structure of each use case. It helps us to determine the cohesion and coupling of the system. IDG can generate collaboration information based on the interaction diagrams generated for the use cases.

Previously, the tools most often used to understand the dynamic behavior of a system are the debugging tools and trace tools. These tools do not give us enough global information. They do not give us a picture which shows the relationship between all the objects in the system.

Jacobson's use cases and interaction diagrams not only can be used in system development like analysis, design and testing, but also can be used in understanding the behavior of the system. Thus, our tool is centred around these notions. A use case is a typical use of an instance of the application system i.e. the relevant message traffic and relevant objects. It can be described by a corresponding interaction diagram.

In particular, we developed an Interaction Diagram Generator (IDG) which has the ability of generating interaction diagrams automatically from executing Smalltalk code by running the use cases in the system. The changes made during implementation will be reflected in the newly generated interaction diagrams. If the generated interaction diagrams are not consistent with those in the design documents, then either the implementation is not correct or the design has been changed during implementation but this change has not been reflected in the documents yet.

In the first case, implementation should be changed. In the last case, the newly generated interaction diagrams can be saved as new design documents.

IDG can also help to understand the dynamic behavior of software systems by means of showing interaction diagrams. In addition, it helps to understand the system structures by showing collaborations between objects. Furthermore, it can be used as a debugger tool.

Users can open an Interaction Diagram Generator view by specifying an application. Within this view, they can get a interaction diagram for each use case automatically. Filters are available to enable users to select only those objects and messages which they are interested in and which should therefore appear in the generated interaction diagrams. While viewing an interaction diagram, users can hide an operation corresponding to a message send and the operations generated by that operation. users can also get a collaborations browser which shows the collaborations between classes in the application.

In the next chapter, we present terminology and fundamental issues in object-oriented paradigm. Chapter 3 discusses some related works. Chapter 4 describes the features of IDG. Chapter 5 describes the design and implementation of IDG. Chapter 6 describes the usage of IDG and gives a sample IDG application. Finally, we summarize our contributions and make some suggestions for future work.

Chapter 2

Background

In this chapter we will discuss some important concepts and define the terminology that will be used hereafter. We will also briefly review some object-oriented system development methods.

2.1 Object-Oriented Systems

An object-oriented system consists of a number of *objects* that interact. Objects have both *methods* and *local state*¹. The methods are the operations that define the behavior of an object and the local state is the current data values which remember the effect of these operations. The local state is represented by *instance variables*. Object-oriented systems execute through *message passing* among objects. A method in an object executes in response to a message and may propagate further messages to other objects.

Object-oriented programming languages differ from procedural languages in that they support *classes* and *inheritance*. A class is a template that specifies the common structure and the common behavior which a set of objects share. Objects of the same class have the same structure and behavior. Every object is an instance of a class. Inheritance is a relationship among classes. It is the ability of one class to define the behavior and representation of its instances as a superset of the definition

¹To avoid confusion of terminology, terms and concepts are referred to using their commonly used Smalltalk names.

of another class or classes. A class that inherits from one class (*single inheritance*) or more classes (*multiple inheritance*) is a *subclass*. The class from which another class inherits is a *superclass*.

Object-oriented systems have the advantages that they are easy to be reused, refined, tested, maintained, and extended.

Object-oriented design and programming has potential advantages for helping us to achieve the following basic software development principles.

- **Modularity** Modularity expresses the idea that each chunk of code should be autonomous and self-contained. Objects provide the basic modular building blocks containing both data and functionality relating to a particular concept.
- **Encapsulation and Information Hiding** Encapsulation and information hiding have been possible in procedure programming. These relate to the idea of localizing things, protecting them from interactions with the environment. In an object-oriented approach, you bind together data and procedures. You make that capsule as tight as possible so that the interface is as small as possible. You have an interface where the things that are allowed to be seen across the interface are very clearly defined.
- **Reusability and Productivity** One of the most enticing "promises" of object-oriented systems is that of real reusability; reusability of code, reusability of designs. Reusability of code is achieved by building of libraries of classes. Several language environments, such as Smalltalk, currently include an extensive number of prewritten and well-validated classes. The availability of these library classes, once learned, makes building new software systems through specialization significantly easier than in the domain of procedural programming.

Reusability of design occurs in the development of application *frameworks*. A framework is an application specific class library, or a grouping of classes tuned specifically to a single applications-oriented environment, but still general enough to be widely reusable. Reuse of detailed design is also seen at the

program level when classes written in one language are translated into another OO language.

- **extendibility** Because object-oriented programming supports encapsulation and information hiding well, changes and extensions are more easily and safely made to object-oriented systems. Changing the internal behavior of one class will not affect rest of the classes in the system, and new behavior can be added to the system easily through class inheritance.

The object-oriented approach views the world in a more natural way: i.e., it views the world in terms of objects. When we look around at the world we tend to see objects, things that have characteristics. The most immediate characteristic is how the object looks to us (its external interface). Then we notice how the thing behaves. In other words, we see state first and then behavior. This is exactly the object-oriented approach. In object-oriented systems, objects are identified in the real world and translated directly into software objects.

2.2 Use Cases and Interaction Diagrams

Use cases and interaction diagrams are terms used in Jacobson's Object-Oriented Software Engineering (OOSE) [Jaco 92].

2.2.1 Use Case

According to Jacobson, a *use case* is a specific way of using a system by using some part of the functionality. Each use case constitutes a complete course of events initiated by a user; the use case specifies the interaction that takes place between a user and the system. For example in a bank account system, opening an account, making a withdrawal transaction, or making a deposit transaction are all use cases. Use cases can be used during the analysis to specify the requirements of a system.

There are two possible ways of looking at a use case. One is to view the system as an object and the use cases as operations that get invoked on the system. The

other way is to view use cases as classes, and instances. Jacobson chooses the second view which of course maps easily onto the basic notions of object-orientation.

In the second approach, each use case is a specific way of using the system and every execution of the use case may be viewed as an instance of the use case. When a user inputs a stimulus (or sends a message), the use case instance executes and starts a transaction belonging to the use case. This transaction consists of different actions to be performed. The use case instance exists as long as the use case is operating. These use case instances follow a specific class like all instances in an object-oriented system. When a use case is performed, we view this as instantiating an instance from the use case class. A use case class is a description which specifies the transactions of the use case. The set of all use case descriptions specifies the complete functionality of the system [Jaco 92, page 310].

2.2.2 Interaction Diagram

An *Interaction Diagram* describes how each use case is satisfied by communicating objects. Each use case has an interaction diagram. The diagram shows the relevant message traffic and relevant objects involved in the use case. Figure 2.1 shows a typical interaction diagram. The interaction diagram depicts the use case *Open an Account* for a bank account system (Appendix A). First, a *new* message is sent to class *BankAccount* to create a bank account: *aBankAccount*. Then, the messages *owner:*, *account.Number:*, and *balance* are sent to *aBankAccount* to set the owner of the account, the account number, and the initial balance of the account.

In an interaction diagram, each participating object is represented by a column. These columns are drawn as vertical lines. In our example, there are two columns *BankAccount* and *aBankAccount*. If there are several instances of the same class, they can be drawn either as different columns or the same column depending on which is most legible. The initial version of Interaction Diagram Generator draws the instances in the same column. In the case that two instances are used concurrently, the better way is to draw them as different columns. We will discuss how to expand IDG to draw different columns for different instances from same class in chapter 7.

Figure 2.1: Interaction Diagram

The left to right ordering of the columns reflects the order in which objects become involved in a computation, i.e. first involved objects appear at the left.

In many interaction diagrams we have a column called *system border*. This column represents the interface with the outside world, such as external users, and consequently it can correspond to different interfaces outside the system.

There is a time axis in the interaction diagram which runs downwards. The use case thus starts with the behavior described at the top of the interaction diagram. The time axis of the interaction diagram is controlled by message sends. The distance between two messages in the diagram has no relation to the real time between these message sends.

The interaction diagrams are controlled by message sends. A new message gives rise to a new operation. The operation belongs to a certain column which is represented by drawing a rectangle on that column. Messages are sent from one object (sender) to another object (receiver). In our example, a message *initialize* is sent from the class *BankAccount* to an instance *aBankAccount*. The operation occurs

at the receiving object. A message send is drawn in the interaction diagram as a horizontal arrow that starts in the sender column and ends in the receiver column. The message names are shown above the arrow. Most interaction diagrams start with a message send from the outside, that is, it is drawn from the system border.

Interaction diagrams often exhibit two common different structures: the fork structure, and the stair structure. See figure 2.2. Fork diagrams indicate a cen-

Figure 2.2: Structure of Interaction Diagram

tralized structure. Typically the control sequence is placed in one object, often a control object. The other objects are typically used as information carriers or as an interface to a user. Stair diagrams often indicate a decentralized structure. Each object has a separate task and it only knows the surrounding objects it needs in order to help carrying out this task.

2.3 CRC cards and Collaborations Graphs

Class, Responsibility, and Collaboration (CRC) cards and collaborations graphs are two kind of documents used in Wirfs-Brock's Responsibility Driven Design [WWW 90].

2.3.1 CRC Cards

CRC cards characterize objects by class name, responsibilities, and collaborators. The class name of an object creates a vocabulary for discussing a design. Responsibilities identify problems to be solved. Collaborators specify other classes they need to get their jobs done. They are used to document the collaboration of objects. They explicitly represent multiple objects simultaneously. An object is written on a 4" x 6" card. The class name appears underlined in the upper-left hand corner, a bullet list of responsibilities appears under it in the left two-thirds of the card, and the list of collaborators appears in the right third. Figure 2.3 shows a CRC card.

Figure 2.3: A Class-Responsibility-Collaborator card

2.3.2 Collaborations Graphs

Collaboration graphs display the collaborations between classes and subsystems in a graphical form.

According to Wirfs-Brock, *collaborations* represents requests from a *client* to a *server* in fulfillment of a client responsibility. Here a client is the object who makes the request and server is the object provides the service to the request. A collaboration is the embodiment of the *contract* between a client and a server. Here a contract is the list of requests that a client can make of a server. From the client's point of view, each of its collaborations are associated with a particular responsibility implemented by the server. The pattern of collaborations within a system reveals the flow of control and information during its execution.

Collaborations graphs describe the collaborations between classes and subsystems. They represent classes, contracts, collaborations, and superclass-subclass relationships. Classes are shown as labeled rectangles. Subclasses are graphically nested within the bounds of their superclasses. A subclass that has more than one superclass is drawn several times, nested within each of its superclass. Contracts are shown as small semicircles inside the edges of the class to which they belong. Each contract has one semicircle. A number may be assigned to the semicircle to represent the contract number. Collaborations between classes are represented by an arrow from the client to a contract supported by the server.

Figure 2.4: Collaborations Graph

Figure 2.4 shows a collaborations graph. **BankAccount** is a class. **Trans-**

action is another class which has three subclasses: **DepositTransaction**, **WithdrawalTransaction**, and **CheckingTransaction**. **Transaction** collaborates with **BankAccount** through *contract1* in **BankAccount**. This contract may be giving the balance. **BankAccount** collaborates with **Transaction** through *contract1* and *contract2* in **Transaction**. *Contract1* may be printing the transaction. *Contract2* may be applying the transaction to the account which modifies the balance of the bank account.

2.4 Object-Oriented System Development Methods

There are many object-oriented development methods. Jacobson reviewed several publicly available and the most well-known methods in [Jaco 92] chapter 16. In this section, we include two which are related to our work.

2.4.1 Responsibility-Driven-Design

Responsibility-Driven-Design (RDD) by Wirfs-Brock [WWW 90] describes an application in terms of classes, their responsibilities, and their collaborations.

In RDD a system is developed from the requirements specification which should be the basis for actual implementation. The method comprises a number of phases where each phase is described as a number of activities.

Exploratory phase:

1. **Find the classes(objects).** Start with the requirements specification, create classes of objects that will model the domain of your application. Identify abstract classes by looking for common attributes in classes as described by the requirements.
2. **Assign responsibilities to each class.** Responsibilities include the knowledge an object maintains, and the actions an object can perform. The ac-

Figure 2.5: A Simple Hierarchy Graph

tivities include identifying responsibilities from requirements specification and identified classes, assigning responsibilities to classes.

3. **Define collaborations between classes.** Collaborations represent requests from one object (client) to another object (server) in fulfillment of its responsibility. Activities include finding collaborations by analyzing the interactions of each class and examining the responsibilities for dependencies, and recording collaborations by filling out 'CRC' (Class, Responsibilities, and Collaboration) cards.

Refinement phase:

1. **Refine inheritance hierarchies between classes.** Deliver hierarchy graphs, Venn diagrams, and contracts.

A **hierarchy graph** presents a graphical representation of the inheritance relationship between related classes. Classes are represented by rectangles, labeled with the class names. Inheritance is indicated by a line from superclass to subclass. Superclass is positioned above subclasses. An abstract class is indicated by filling in the upper left corner of the rectangle. Figure 2.5 shows a partial hierarchy from the simply Bank Account example. This graph indicates that *DepositTransaction*, *WithdrawalTransaction*, and *CheckingTransaction* are subclass of *Transaction*. *Transaction* is an abstract class.

Venn diagrams are used to show the common responsibilities among classes.

A **contract** defines a set of related requests that a client can make of a server. The server is guaranteed to respond to these requests.

2. **Identify subsystems.** Deliver collaborations graphs and subsystem cards.

Subsystem are group of classes, or groups of classes and other subsystems, that collaborate among themselves to support a set of contracts. Drawing and analysing collaborations graphs (discussed in previous section) can help to find subsystems.

3. **Construct protocols for each class.** Define the specific signatures for the methods that each class will implement. Write a design specification for each class and subsystem. Write a design specification for each contract

Implementation phase:

Choose a programming language. Directly map the design to the implementation.

2.4.2 Object-Oriented-Software-Engineering

Object-Oriented-Software-Engineering (OOSE) by Jacobson is a use case driven approach.

Different from the other methods, Use Case Driven Approach has been in use for many years to design and build real industrial software systems. The ideas and methods are based on practical object-oriented development experiences. The approach centers on an analysis of the ways in which a system is actually used, on the sequences of interactions that comprise the operational realities of the software being engineered [Jaco 92, foreword].

In this method, system development is viewed as a gradual transformation of a sequence of *models*. The first model describes the customer's requirements and the last model is the fully tested program. Between them are a number of other models. In this way the complexity of system development can be handled in an organized way. In particular, Jacobson divides the system development process into

the development of five different models as follows, each of them focusing on a certain aspect of the system.

- The **requirements model** aims at capturing the functional requirements.
- The **analysis model** aims at giving the system a robust and changeable object structure.
- The **design model** aims at adopting and refining the object structure to the current implementation environment.
- The **implementation model** aims at implementing the system.
- The **test model** aims at verifying the system.

Each model tries to capture some part or aspect of the system to be built. Figure 2.6 shows the models and associated processes.

Figure 2.6: The models and associated processes

In this method, the system model is use case driven. A use case model is developed in the requirements model and is used when developing all other models. The use case model uses *actors* and *use cases*. It is represented as follows. The system is bounded by a box. Each actor is represented by a person outside the box. Actors

define roles that users can play. The use cases are represented as ellipses inside the box.

Figure 2.7: Use Case Model

Figure 2.7 shows a use case model.

The use case model will control the formation of all other models. The functionality specified by the use case will be structured into a logical, robust, and implementation environment-independent analysis model. Then the model is adapted to the implementation environment and further refined in the design model using the use cases to describe how the use cases flow over the design objects. The use cases will then be implemented by the source code in the implementation model. Finally the use cases will give us a tool to test the system. The use case model will also give support of writing manuals and other operation instructions.

The requirements model consists of a use case model, interface descriptions, and a problem domain model. It includes actors and use cases supported by an intuitive domain object model and interface descriptions. The actors model something that will interact with the system and a use case specifies a flow that a specific actor invokes in the system. The object model gives a conceptual, easy to understand picture of the system and the interface descriptions describe the system interfaces in detail. Inheritance of use cases, actors and objects are initially introduced in this model.

The analysis model is developed from requirements model. Each use case will be entirely divided into objects (classes) of one of three types. The three object

types are *interface objects*, *entity objects*, and *control objects*. Each of these objects has its own purpose and will model one specific aspect of the system. The entity object models information in the system that should be held for a longer time, and should typically survive a use case. Person is an example of entity object. The interface object models behavior and information that is dependent on the interface to the system. The user interface functionality for requesting information about a person is an example of interface object. The control object models functionality that is not naturally tied to any other object, typically, operating on several different entity objects, doing some computations and then returning the result to an interface object. A typical example of a control object is calculating taxes using several different factors. The advantage of using these three object types, according to Jacobson, is to help to get a stable structure that will be more adaptable to changes. The stability will occur in the sense that all changes will be local, i.e., affect (preferably) only one object in the system. For example, changes to the interface should typically affect only interface objects.

The symbols of the three object (class) types used to draw structure of the system are as follows.

The subsystems and relationships between classes will also be specified in this model.

The design model refines the analysis model further and takes into the consideration of the implementation environment. The classes are represented by rectangles in this model. And communication between objects is described by interaction diagrams. Use case model is used here. Each use case is described as a sequence of stimuli sent between the objects. This model also describes the programming language, operating system, and database management system that will be used to develop the system.

The implementation model consists mainly of the source code written to implement the classes.

The testing model is developed when testing the system. Testing is done on several different levels of granularity. The lower level objects are tested initially when performing unit testing. The use case model is used as the primary tool when doing integration testing.

2.5 Summary

Applying object-oriented technology in system development is a new way of developing software systems. There are many new terminologies, new notations, new developing methods which have emerged. We have found that Jacobson's use cases and interaction diagrams and Wirfs-Brock's collaborations graphs are not only useful during the system analysis, design, implementation, and testing, but also very helpful in understanding the dynamic behavior of the system. They have proved to be very useful methodologies and notations that we can use to describe object-oriented systems.

Chapter 3

Related Work

In this chapter we describe work related to the problems addressed by this thesis. These include reverse engineering and visualization tools for understanding the dynamic behavior of object-oriented systems. Our work is related to these works in the sense that our interaction diagram generator is a powerful tool that can be used in reverse engineering object-oriented systems, and it shares similar features with those tools for understanding object-oriented systems.

3.1 Reverse Engineering

Reverse engineering is the process of analyzing a system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction [CC 90]. The purpose of reverse engineering a system is to increase the overall comprehensibility of the system for both maintenance and reuse. It involves extracting design artifacts and building or synthesizing abstractions that are less implementation-dependent. It is a process of examination, not a process of change or replication. It does not change the system. There are many subareas of reverse engineering. *Redocumentation* and *design recovery* are two widely referenced subareas.

Redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternate views. Redocumentation is the simplest and

oldest form of reverse engineering. Some common tools used to perform redocumentation are pretty printers, which display a code listing in an improved form; diagram generator, which create diagrams directly from code, reflecting control flow or code structure; and cross-reference listing generators. The key goal of these tools is to provide easier ways to visualize relationship among program components [CC 90].

Design recovery recreates design abstractions from a combination of code, existing design documentation, personal experience, and general knowledge about program and application domains. Design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus, it deals with a far wider range of information than found in conventional software-engineering representations or code [Big 89].

3.2 Understanding Object-oriented Systems

Object-oriented application systems contain many complex interrelated objects. One system may have thousands of objects and hundreds of methods defining the operations of those objects. All of the objects are distributed. Usually there is no main program from which you can follow the execution thread. Object-oriented system works through sending messages among objects. When the number of objects becomes large, it can be difficult to understand how the system works and what is the relationship between objects in the system. One main advantage of using an object-oriented language is the reusability of objects. The reusability scenario, however, often involves some understanding of the specific behavior of existing objects. Documentation is one answer to this problem but documentation rarely keeps pace as an object's methods are altered over time. Understanding object-oriented system becomes a general issue. Several object-oriented visualization tools have been developed.

3.2.1 Cunningham Diagrams

Cunningham and Beck introduced a notation for diagramming the message sending dialogue that takes place between objects participating in an object-oriented computation [CB 86]. Here, an object-oriented computation is the same concept as the use case which we described in the previous chapter.

They represent an object as a box and draw a message send as a directed arc landing within the receiving object. When one object invokes a computation in another object through a message send, the message send is shown as an arc originating from the sending object and landing in the receiving object. The selector of methods invoked by messages are placed at the receiving end of a message arc. An object is identified by its class. Class names are shown beside the objects' box. When a method is inherited from another class, the receiver will be divided into layers to represent the classes involved. It will show the class of the receiver, its superclass, superclass' superclass (if applicable) until the class which implements the method is found. The method selector will be placed in the layer which implements the method. Deeper layers will be superclasses. The top layer will be that of the object's own class. A message arc goes from top layer through several layers until it finds its selector in a deeper layer.

Cunningham diagrams were implemented by extending Smalltalk-80's debugger. They added an additional pane on the right as a diagram editor. Information is added to the diagram by execute debugger's commands *step* and *send* which duplicate the ordinary debugger commands and record the message in the diagram.

Figure 3.1 shows a Cunningham diagram. The computation of the diagram is to open a bank account and make a checking transaction in a simple bank account application. The sample Smalltalk code is shown in Appendix A. We can see from figure 3.1 that Cunningham diagrams could become complex and difficult to follow when a system becomes large and complicated. The diagram does not represent the sequence of computation. Although we can see the objects involved in a computation and message sends between these objects, the order of the computation is not easy

Figure 3.1: Cunningham Diagram

to follow. In addition, the diagram's layout may be cumbersome for describing large systems.

Cunningham diagrams are early attempts to help people understand object-oriented systems. The authors have used these diagrams to teach beginning and advanced object-oriented programming to more than one hundred students. It influenced our Interaction Diagram Generator in that we also adopted the approach of reusing Debugger class in Smalltalk.

3.2.2 GraphTrace

GraphTrace [KG 88] is another tool for understanding the dynamic behavior of object-oriented programs. The tool works with both structural and behavioral views of a system. Structure views are simply instantiated and animated. Behavioral views are generated by recording the message activity that occurs as a program is running and then animating those views.

In GraphTrace, graph diagrams are used to display the structure of programs. Graphs consists of nodes and links. Nodes represent components of the system, such as objects, methods and instance variables, and the links represent some relationship or set of relationships that exist among the components, such as inheritance and delegation. Animation is achieved by highlighting and dynamically annotating the nodes and links to show the activity of the program. It provides several views to allow a user to reveal different aspects of an object-oriented program, such as component graph and method invocation graph.

Component graphs are static. A component graph shows the structure of an object by giving the parts information. Figure 3.2 is a component graph which shows A is composed of B, C and D. D is composed of E and F.

A method invocation graph is a condensed view of the sequence of messages that are invoked as a program runs. In this view, the child nodes of a node show the methods that are invoked by the method indicated by that node. The graph as a whole shows the complete nesting of message invocations. Figure 3.3 shows Method1 invokes Method2 and Method3, and Method2 invokes Method4.

Figure 3.2: Component Graph

Figure 3.3: Method Invocation Graph

Figure figure 3.4 shows a snapshot of GraphTrace.

Object invocation graphs and object-slot invocation graphs are similar to method invocation graphs, but show the names of the objects involved or the names of the methods with the names of their objects.

GraphTrace was developed within *Strobe* [SC 86] (an object-oriented language implemented as an augmentation to Common Lisp) and *Impulse-86* [SDB 86] (a substrate for building user interfaces). It was implemented by modifying the low-level Lisp function that implements message passing in *Strobe*.

GraphTrace represents a sequence of message calls by several invocation graphs: method invocation graph, object invocation graph and object-slot invocation graph. These invocation graphs provide different levels of viewing message traffic between objects. Since it shows the sequence of message calls by highlighting the current invoking nodes, we can only see the sequence when we animate the program. It does not give us static graphs which show the sequence of message calls and the relationship between objects after we have run a computation.

Figure 3.4: GraphTrace

3.2.3 Track

Track [BH 90] is a tool kit to interactively construct environments that trace the execution of methods and the flow of messages between Smalltalk-80 objects.

Track uses icons to represent classes and instances. The user sets up trace by placing obstacles between icons. Obstacles describe which messages to be traced. The direction for messages to be traced may be unidirectional or bidirectional. Traced messages or methods are both graphically animated (a little ball moving over the obstacle when traveling from its sender to the receiver) and textually recorded. Thus, the traced program can be analyzed either while its running or after it has finished execution. The user can specify and concurrently run several different traces to have multiple visualizations of programs. Figure 3.5 shows the Track system.

The tracing kernel of Track is implemented by dynamically introducing a new method when a message or method is to be traced. It passes the message to be traced to the original method and, additionally, produces trace information.

Track is a trace tool. It is tightly integrated with the standard tools of the Smalltalk-80 programming environment. It is a helpful tool for programming and debugging in Smalltalk-80. But Track does not graphically show the messages involved in an execution.

3.3 Summary

Several attempts have been made to visualize the interaction between objects. From the study of these works, we believe that to see the dynamic behavior of object-oriented system is very helpful to understand the system. All the tools we discussed above can be used to trace an object-oriented computation. However, after that computation completes, they don't give us a picture which records the message sends between objects during that computation. Interaction diagrams can provide this kind of information.

We have used some ideas from this previous work. One is that to implement our Interaction Diagram Generator efficiently, we can reuse Debugger class in Smalltalk

Figure 3.5: The Track System

as Cunningham and Beck did in Cunningham diagram since the Debugger already has the information from a computation. The other is to add a filter which allows us to view only the messages we interested in. This allows us to view the use case at a desired level of detail and abstraction.

Chapter 4

Interaction Diagram Generator

In this chapter we will give an overview of the Interaction Diagram Generator (IDG). IDG is a tool integrated with the Smalltalk/V environment. The tool provides users with the following capabilities:

1. to generate an interaction diagram from running Smalltalk code.
2. to control the level of information displayed in the diagram using mechanisms both prior to filtering and after diagram is generated.
3. to save a displayed interaction diagram in a compressed description format.
4. to load and display previously saved interaction diagrams.
5. to print interaction diagrams.
6. to display the generation of an interaction diagram in a "single step" mode or in a "go" mode.
7. to analyze the interactions generated from running a set of use cases to provide a view depicting the collaborations taking place between objects.
8. to conveniently view the Smalltalk code for messages displayed in an interaction diagram.

An application system developed in Smalltalk usually consists of some classes. Each class has its own responsibilities, and consists of several attributes and methods. All the classes function together to fulfill the requirements for the application system. We call the application system a *Smalltalk application*. A Smalltalk application sets up the context for the Interaction Diagram Generator. A use case of a Smalltalk application system is a set of Smalltalk code which specifies the way of how to use one functionality of the system. In general, the interaction diagram for a use case should focus on behavior at a useful abstraction level. Not all messages are of interest to a designer. Therefore the interaction diagrams generated by IDG only show the message sent between objects defined within the application. For example, to the designers of a bank account system, the message sends between the objects which are the instances of the classes designed by them are of interests. They want to see *BankAccount*, *CheckingTransaction*, *DepositTransaction*, etc. in the generated interaction diagrams, but not objects predefined in Smalltalk system like *Integer* and *Collection*. However, to abstract at the application level may still be too low. Users may want the generated interaction diagrams to only show the messages sent between certain specified objects within the application. This requirement is fundamental, especially for large complex application systems, because the use cases for large systems may involve many objects and messages, users may only want to look at them at a certain level or they may want to explore only a certain part of a use case. IDG provides filtering features in two ways: (1) it allows a user to select the classes and methods to be shown on the interaction diagram before running a use case, and (2) it allows the user to filter out the displaying of message sends while viewing a generated interaction diagram. The latter facility provides a mechanism for the user to interactively control the level of details shown in a displayed interaction diagram at any time.

An IDG is created for a specific Smalltalk application. It displays all the classes defined in the application, and all the methods for each class. It also shows the use cases for the application. Therefore, IDG users can have a clear view of their application systems.

IDG generates an interaction diagram by running the corresponding Smalltalk code - the use case. The interaction diagrams express use cases in graphical form from which one can trace all the message passing and all the objects involved in the use cases. The IDG can also show collaborations between objects in a collaborations view. When the IDG is invoked, it opens a window titled 'Interaction Diagram Generator' for a given Smalltalk application. The window has four main panes and six buttons. See figure 4.1.

The *Class Pane* lists the classes associated with an application. The *Method Pane* lists the class methods or instance methods of a class. The *Use Case Pane* lists all the use cases of the application. The *Diagram Pane* displays interaction diagrams. A *Method Browser* may pop up in the *Diagram Pane*. Clicking the *collaborations* button will cause a collaborations view be displayed. In this view, you can see the relations among all the classes in the application, i.e., the helpers and users for each class in the application. Clicking the *Print* button will make a hard copy of the currently displayed interaction diagram. Clicking the *Save* button will save the currently displayed interaction diagram to a file. Clicking the *Load* button will load a previously saved interaction diagram from a file. Clicking the *Step* or *Go* button will execute the use case one step, or execute until finishes.

In the first section we discuss these panes. The second section will describe the collaborations views and Method Browser. The third section will discuss use IDG as a debugging tool. The last section will discuss other IDG features.

4.1 The Interaction Diagram Generator Panes

Interaction Diagram Generator has five panes and six buttons in total. We discuss the four main panes which are mentioned above. The rest of the them are straight forward.

Figure 4.1: Interaction Diagram Generator

4.1.1 Class Pane

The *Class Pane* is a list pane which lists the classes in the opened application. Double clicking on a class will turn off the class i.e. the objects from this class will not be shown on the interaction diagrams. Double clicking on the class again will turn on the class so the objects from this class will be shown. This provides a filter at class level. In this pane, users can pop up a menu to add or remove classes, and choose to list all the classes or just the classes that are not turned off.

4.1.2 Method Pane

Similar to the *Class Pane*, the *Method Pane* is a list pane which lists the methods of a selected class in the *Class Pane*. Users can select to list class methods or instance methods by clicking the above button. Similar to the class pane, double clicking on a method will turn off the method i.e. the messages sent to the selected class with this method name will not be shown in the ensuing interaction diagram. Double clicking on the method again will turn on the method. This provides a filter on method level. In this pane, users can pop up a menu and choose to list all the methods or just the methods which are not turned off.

4.1.3 Use Case Pane

The *Use Case Pane* is a list pane which lists all the use cases of the opened application. Selecting a use case will set that use case to be the current use case. Then the interaction diagram for that use case may be generated by clicking the *Go* or the *Step* button.

4.1.4 Diagram Pane

The *Diagram Pane* is used to display the interaction diagram for the selected use case. We designed the *Diagram Pane* to contain two subpanes; the *Diagram Title Pane*, and the *Interaction Diagram Pane*. The *Diagram Title Pane* contains the

date and time when the interaction diagram is generated, the name of the use case, and the names for each of the columns. Each column represents an object involved in the computation of the use case. The *Interaction Diagram Pane* shows all the message sends between columns. The reason for having two panes is that we need to scroll the interaction diagram in the pane but we do not want the title and column headings to be scrolled out of view. We keep the title in a separate pane so it will not disappear when we scroll the diagram vertically. The horizontal scrolling of the *Diagram Title Pane* is controlled by the *Interaction Diagram Pane*, so they will scroll in a synchronized fashion.

The *Diagram Title Pane* is a *GraphPane* without scroll bars. We added a vertical scroll bar which is a *ScrollBar* to allow the scrolling of the titles containing more than two lines. It does not have a horizontal scroll bar.

4.2 Method Browser and Collaborations View

In this section, we describe the Method Browser and Collaborations views.

4.2.1 Method Browser

While looking at an interaction diagram, people may want to know how a receiving object reacts when it receives a message. This can be achieved by opening a method browser and looking at the code for that method.

The Method Browser is a *MethodBrowser* opened by double clicking on the label departing a message send to an object in an interaction diagram displayed in *Diagram Pane*. This browser shows the code of the method of the receiver's class. If the method is inherited from its superclass, the code of its superclass is shown.

Figure 4.2 is a snapshot taken when after a user has double clicked on the message *initialize* sent to *aCheckingTransaction*.

Figure 4.2: Activating a Method Browser in IDG

4.2.2 Collaborations View

In IDG, users can bring up a collaborations view by clicking the *collaboration* button. A Collaboration view shows collaborations between objects by showing an object's helpers and users. Here *helper* is the same concept as *collaborator* defined in CRC cards in chapter 2. The needed information is extracted from the analysis of the interaction diagrams of use cases currently selected by users. Thus, users can compare the final collaborations with the collaborations described in CRC cards or other design documents and verify them. This information is also useful in capturing design information and regenerating design documents. For example, a collaborations view shows the helper classes for every classes in an application. From this we can obtain information on how each class fulfills its responsibilities through collaboration with other classes. See figure 4.3.

Figure 4.3 shows a collaborations view. It contains five panes. The *use case pane* is a list pane which shows all the use cases in the application. The *interaction diagram file pane* displays the interaction file names for all the use cases. If there is no file associated with a use case, that use case will have an empty line in the *interaction diagram file pane*. Users can select a use case and then assign a file name to the use case or remove it through the pop up menus. The bottom three panes are used to display the collaborations according to the use cases selected and interaction diagram files assigned. The left pane *Class* lists the classes in the application. Double clicking on a class in the pane will pop up a class browser for that class. The middle pane *Helper* lists the helpers for the selected class. The right pane *User* lists the users for the selected class.

The information needed for displaying the collaborations is obtained by asking the application its collaborations. We will discuss this more in the section **The Application Class** of the next chapter.

Figure 4.3: Collaborations View

4.3 Debugging Use Cases

Debugging object-oriented system's code can be very challenging because of the the dynamic and distributed behavior of an object-oriented system. Traditional debuggers provide facilities that allows users to inspect the environment at a particular point of execution, but they don't provide users with an easy-to-understand view of the overall flow of execution. For example, with the Smalltalk Debugger, we can know the execution thread as one object sends a certain message to another object, then the second object sends a message to a third object, Finally, the execution returns to the first object. We can inspect those objects and temporary variables at each step, but we don't have a global view of what objects are involved in the execution and how they interact. Using traditional debuggers, users often have to keep track of the execution flow manually which is undesirable. Interaction Diagrams can provide users with an accurate and easy-to-understand global and controlled picture of the execution flow. Therefore, the IDG can also be used as a debugging tool or tracing tool. Users can describe what they want to debug as a use case and let the IDG run that use case. They can trace the execution of the use case and see if the message flow is as anticipated or not. Users may check on the objects involved if they wish, and can control which objects are seen. Users can also modify the code of their application classes and re-run the use case again without exiting from IDG.

4.4 Other Features

4.4.1 Filtering

IDG focuses on the application domain objects to generate interaction diagrams. It only shows the message sends between objects which are specified in the application. If users want to include objects not encapsulated in the application, they can do it by adding those classes to the application in *class pane*.

For complex use cases, the interaction diagrams may become very large. To look at the whole diagram at once, users may feel it is not easy to understand. The

filtering feature allows users to choose the classes and methods they want to be shown in the generated interaction diagrams. They can turn off the parts which they are familiar with or which they are not immediately concerned with, and focus on the objects and messages of current interests. This filtering facility helps users to understand large, complex interaction diagram.

There are two opportunities for filtering. One is prior to the generation of the interaction diagrams, the other is after the generation. In the first case, users can add filters at several different levels. A class level filter allows users to turn off the classes which they are not interested in, so that objects from those classes will not be shown in the generated interaction diagram. User can also add filters at method level. A method level filter allows a user to turn off some methods, so that the messages corresponding to those methods will not be shown.

Users also have the opportunity to manipulate what is shown after an interaction diagram is generated. They can hide or unhide a message and the subsequent messages generated by that message. This can be done by double clicking the message sending arrow. A little black square will be displayed at the place where the message sending arrow was previously located. Double clicking on the little black square will redisplay the hidden messages.

4.4.2 Save, Load, and Print Interaction Diagrams

Object-oriented system development is an iterative incremental process. There will be many iterations involved in developing a system. With the Save, Load, and Print features, we can monitor the evolution of a system. In addition, when users get the hard copy of all the interaction diagrams for all the use cases of a system they may review them together to understand the dynamic behavior of the system. The generated interaction diagrams can be saved as system documentation.

IDG allows users to save and print out interaction diagrams with date and time stamps. Clicking on the *Save* button, IDG will prompt users to input a file name to save the current displayed interaction diagram as a bitmap. Clicking on the *Print* button, IDG will prompt users to input a scaling factor then print out the current

displayed interaction diagram. Clicking on the *Load* button, IDG will prompt users to input a interaction diagram file name then load that file.

4.4.3 Execution Modes

IDG provides two execution modes for executing use cases and generating interaction diagrams. If users want to see how a interaction diagram is generated step by step, they can choose the *step* mode. If users want to get a interaction diagram generated in one step, they can choose the *go* mode.

Clicking on the *Step* button, the selected use case will be run in *step* mode. In this mode, IDG will execute the use case step by step. For each step, one message send will be drawn on the diagram. This helps the user to understand how the system functions by controlling the rate at which the diagram is generated.

Clicking on the *Go* button, the selected use case will be run in *go* mode. In this mode, IDG will execute the use case and generate the whole interaction diagram at once.

4.4.4 File Menu and View Menu

The File Menu is used to save or open an application. An IDG is created on a Smalltalk application. Once the application is created, it is necessary to save it, so that it can be used next time. Also, while working on an application, users may change some settings. For example, they may change the filter information. This information needs to be saved as well to permit users to reload an IDG with settings used to generate it in the first place.

From IDG menu **File**, users can save or open an application. Saving an application saves all the information associated with the application; this includes the classes and use cases in the application, and the filter information for the classes and methods. See figure 4.4.

The View Menu is used to choose whether to show return arrows or not in the generated interaction diagrams. The purpose for the selection is that sometimes

Figure 4.4: The File Menu

Figure 4.5: The View Menu

users may want to see where an operation returns to and what it is returned. Sometimes they may want to focus on the message sends and don't want the diagram to contain too many arrows and text. The **View** menu gives the two choices. See figure 4.5.

4.4.5 Coloring The Operations

The interaction diagrams displayed in *Interaction Diagram Pane* are colored on the their operation boxes. Different colors are used to display nested operations. If we only have one color, the nested operations can't be shown. For example, a method *method1* may contain *self.method2*, *self.method3*..., the operation of *method1* will not finish until the operation of *method2*, *method3* finishes. In this case, we will show the operation of *method1* in blue, and the operation of *method2* and *method3* in red. For messages sent to itself, the message sending arrow will start from and end with the same column.

4.5 Summary

Interaction Diagram Generator can generate interaction diagrams for use cases defined in a running Smalltalk application. It contains the panes to show the classes,

methods and use cases of the application. The interaction diagrams can be generated in two mode: *step* or *go*. There are two ways of filtering in IDG: one is filtering before running the use cases, the other is filtering after the interaction diagram is generated. The generated interaction diagrams can be saved, loaded, and printed. Users can bring up a method browser from displayed interaction diagram, and bring up a collaborations view to view the collaborations between the classes in the application. The application and its settings can be saved and opened later.

Chapter 5

The Design and Implementation of IDG

In this chapter we discuss the design and implementation of Interaction Diagram Generator (IDG). As we said before, the IDG can generate interaction diagrams automatically. It is developed in Smalltalk/V Windows 2.0 and integrated with the Smalltalk environment. The user interface part of IDG was developed using the WindowBuilder/V interface builder.

5.1 The Design Principles

Reuse is an important benefit of developing object-oriented systems. We developed Interaction Diagram Generator by reusing the existing Smalltalk library as much as possible. The key points of implementing interaction diagram generator were how to control the execution of use cases and how to capture the message passing information. We studied the Smalltalk/V *Debugger* class and found that it maintains all the information needed to generate our interaction diagrams. Therefore we decided to develop our interaction diagram generator by subclassing *Debugger* class.

Now we describe how the interaction diagram generator works. An interaction diagram generator is opened on a Smalltalk application. We create a *Debugger* for the selected use case, and change the debugger view to IDG view. Users control the execution of use cases in IDG view, and IDG view passes the execution command to the *Debugger*. The created *Debugger* is a subclass of *Debugger* called *IDGDebugger*. Several *Debugger* methods were overridden in *IDGDebugger*. *IDGDebugger* knows

Figure 5.1: IDG Overview

how to generate interaction diagram incrementally by sending messages to the *InteractionDiagram*. The generated interaction diagram will be displayed in an IDG window.

Figure 5.1 provides an overview of how IDG works. While running a use case, IDG basically consists of an *IDGDebugger*, an *IDGView*, an *InteractionDiagram*, and a *UseCase*. *anIDGView* gets the input commands and lets *anIDGDebugger* execute the commands. *anIDGDebugger* executes use case code according to the command and sends generating message and the message send information to *anInteractionDiagram*. *anInteractionDiagram* adds the message send information to the diagram and displays itself in *anIDGView*.

5.1.1 The Creation of *IDGDebugger*

To implement an Interaction Diagram Generator, we need to create an *IDGDebugger* for a selected use case. We first explored how a *Debugger* is created for a *self halt* message, then we figured out how to create an *IDGDebugger* for a use case.

The Smalltalk/V system provides support for multiple independent processes. A *Process* is an instance of *Process* class. It represents a sequence of actions that can be carried out independently of the actions represented by other *Processes*. The Smalltalk/V virtual machine has only one processor capable of carrying out the sequence of actions a *Process* represents. A single global instance of class *ProcessorScheduler* (*Processor*) keeps track of all of the *Processes*, and schedules which one will be carried out.

In order to create an *IDGDebugger* for a use case, it is needed to have a *Process* that holds the sequence of actions for carrying out the use case.

In Smalltalk/V, the *halt* method in class *Object* is defined as:

Process

```
queueWakeupback: 'halt encountered'  
makeUserIF: CurrentProcess isUserIF  
resumable: true
```

What the code does is to create a process for the currently executing code and add a message to *PendingEvents*.

PendingEvents is a global variable of class *OrderedCollection*. It is a queue which contains the pending events waiting for processing at any time. *Notifier* is another global variable of class *NotificationManager*. It knows all the windows in the host system and controls the execution of events from the host system. Generally, the events coming from user input will be executed first, when there is no this kind of event, *Notifier* will send message *nilEventWin* to *CurrentEvent* (which is an instance of class *InputEvent* representing the current event). To respond to this message, *CurrentEvent* will get the first message from *PendingEvents* and send a *perform: withArguments:* to itself. The arguments to this message is the first message from

PendingEvents and its arguments. This *performArguments* method causes the first event in *PendingEvents* be processed.

Since the receiver for executing the messages in the *PendingEvents* is an *InputEvent*, the class *InputEvent* must contain a corresponding method. The message added to *PendingEvents* by the above "self halt" code is

```
errorIn: aProcess label: aString.
```

and the class *InputEvent* contains the method.

Here *aProcess* is the created process for the current executing code, and *aString* is "halt encountered". What this method does is to initiate a walkback with "halt encountered" message for debugging. A *Debugger* for *aProcess* with label *aString* can then be created by clicking the *Debug* button on the walkback window.

To create an *IDGDebugger* on a process which contains the use case code, we needed to add a different message to *PendingEvents* which will create an *IDGDebugger* instead of *Debugger*. Therefore, we defined a class *IDGProcess* which is a subclass of *Process*. It contains a new class method *queueWalkback: makeUserIF: resumable: useCaseView* which adds the different message to *PendingEvents*. We have mentioned before that the message will be performed by an *InputEvent*. Therefore we must add the corresponding method to *InputEvent* which creates an *IDGDebugger* instead of a *Debugger*.

Now, we can create an *IDGDebugger* on a use case by asking Compiler to evaluate the following code.

temp variables in the use case

```
IDGProcess queueWalkback: 'IDGDebugger'  
    makeUserIF: CurrentProcess isUserIF  
    resumable: true  
    useCaseView: aUseCaseView.
```

rest use case code

When Compiler evaluates the code, the currently executing code will include the use case code. Therefore, when the method *IDGProcess queueWalkback: makeUserIF: resumanle: useCaseView:* gets executed, the created *Process* will contain the use case code.

The message added to the *PendingEvents* by the method *IDGProcess queueWalkback: makeUserIF: resumanle: useCaseView:* is

```
anIDGIn: aProcess label: aString useCaseView: aUseCaseView.
```

Here *aProcess* is a process which contains the sequence of actions for carrying out the use case code. We added a corresponding method to the class *InputEvent*. This method will create an *IDGDebugger* for that process with *aUseCaseView* as its view.

5.1.2 Generating Interaction Diagrams

In this section, we will briefly describe how an interaction diagram is generated for a selected use case.

methods is an instance variable in class *Debugger*. It keeps track of the last executed method which includes the class name and the method name. This is the information needed to generate our interaction diagrams. Besides the class name and method name, we also need to know the object who sends the message which cause the execution of that method. We call this object the "sender". The object which executed the last method may become the sender of the next method. Therefore, we keep track of the sequence of object and its executed methods and maintain them as a stack of senders. The top object in the stack is the sender of last executed method. We will discuss this more in 'The IDGDebugger Class' section.

An interaction diagram is generated incrementally while running a use case. *InteractionDiagram* has an instance variable *time*. It contains the number of messages added to the *InteractionDiagram*. Each time a message send is added to the *InteractionDiagram*, the *time* will be increased by 1. Initially an *InteractionDiagram* is created with *time* value equal to 0. It only contains a system border. There are

neither involved objects nor involved messages. Once the *IDGDebugger* executes the code in the *UseCase* and finishes a message send, the *IDGDebugger* checks if the receiver of the message is within the application and both the class and method are required to be included in the interaction diagram. If the answer is yes, then it will get the sender from the sender stack and modify the sender stack. Then it will pass the sender as well as the class name and method name of that message send to *InteractionDiagram* and ask it to include this message send. The *InteractionDiagram* will then search its column list and create a new column for the receiver if there is no column created for that object in the diagram, create a new operation for this message on that column, and makes the sender of the operation be the sender. The *InteractionDiagram* will increase the time value. Finally, the *IDGDebugger* will update the *InteractionDiagram* to redisplay.

5.2 The Interaction Diagram Generator Classes

In this section, we describe the main classes developed in IDG. The design of these classes is based on the criteria that they should support changes and should be easy to extend. We believe that our classes are reasonably designed and implemented since we were able to add more and more functionality incrementally without having to rewrite the existing classes. The class hierarchies are shown in figure 5.2

We describe the responsibilities of each class in the following sections. Implementation related technical issues are also discussed.

5.2.1 The IDGDebugger Class

The *IDGDebugger* is a subclass of *Debugger*. It is the heart of our IDG. *IDGDebugger* inherits the information needed to generate an interaction diagram from *Debugger* and changes the original debugger view to IDG view. It knows how to generate an interaction diagram incrementally by sending messages to an object of *InteractionDiagram*.

Object**Application** (name classList useCase)**UseCase** (name code interactionDiagramFile)**Item** (name displayMode)**ClassItem****MethodItem****Operation** (message sender receiver start end displayMode displayRectangle)**Column** (title operations displayMode)**InteractionDiagram**(title columns time pane date ...)**ViewManager****UseCaseView** (application itemTable currentClass currentUseCase
debugger interactionDiagram ...)**CollaborationView** (application selectedClass collaborations useCase)**Debugger****IDGDebugger** (useCaseView senderStack ...)**Process****IDGProcess****GraphPane****InteractionDiagramPane** (titlePane vScrollBar hScrollBar vTitleScrollBar)

Figure 5.2: IDG Class Hierarchy

IDGDebugger refines the methods *hop* and *singleStep* so that it can run the process in both *step* mode or *go* mode and it knows the finishing point of the executing use case. It also refines the method *walkback*: so that it can send message to an *InteractionDiagram* to add the current message send to the diagram and show the interaction diagram in the diagram view. *IDGDebugger* also has additional behavior. It knows its IDG view (a *UseCaseView*) and changes the cursor to indicate executing mode.

Our *step* mode is different from the *step* mode in *Debugger*. In *Debugger*, it stops at every message. In *IDGDebugger*, it stops at the message which is within the scope of the application and which is required to be included in the generated interaction diagram.

Next, we will describe how *IDGDebugger* generates an interaction diagram while running a process which represents a use case.

The *Debugger* shows all the message sends as strings containing the class name and method name. If method *method1* of class *class1* sends a message *method2* to an object in class *Class2*, and *method2* sends a message *method3* to an object in class *Class3*, the message sending shows as

```
Class1 >> method1
Class2 >> method2
Class3 >> method3
Class2 >> method2
Class1 >> method1
```

It shows *Class2 >> method2* again when *method3* returns to it. From this, we know when a message starts to execute and when it finishes. We call the first *Class2 >> method2* a 'sending message' and the second one a 'return message'.

IDGDebugger maintains a sender stack. This stack records the class names and method names for all sending messages. We add the names into the stack if it is a sending message, and remove all the names above its name if it is a return message. When we add a sending message to stack, we tell the interaction diagram

to add a new operation. And when we reach a return message, we tell the interaction diagram to end that operation. In this way, the interaction diagram can be generated incrementally.

5.2.2 The Application Class

The *Application* class describes objects which the IDG can open on. An instance of this class represents a Smalltalk application. The instance variable includes a *name* which is the name of the application, a *classList* which is a collection of all the classes in the application, and a *useCases* which contains all the use cases of the application. An Application maintains all the classes and use cases in the application. It knows the collaborations according to the information of its use cases. It obtains the collaboration information by going through the use cases and analyzes their interaction diagrams recorded in *interactionDiagramfile*. The structure of collaborations is represented in figure 5.3

It is a *Dictionary*. The keys are the class names, and the values are pairs (two element arrays). The first element of the pair is a set of helpers to the class saved in the key. The second element of the pair is a set of users of the class saved in the key.

UseCase is a helper of the *Application* class.

5.2.3 The UseCase Class

An instance of this class describes a use case. It has a name, a piece of Smalltalk code to describe the use case, and it records the interaction diagram file generated by this use case.

To create a *UseCase*, a user can describe the piece of Smalltalk code either in a class method, or describe it directly as a string of Smalltalk code. For the former, running the use case is just to execute that class method. In this case you will see on the interaction diagram that the system border only sends one message (with the name of that method name) to the class which contains the class method. Other

classList

Figure 5.3: The Structure of Collaborations

message sends are from that class to other objects or between other objects in the application system. For the latter, running the use case is running the Smalltalk code described. The system border may send several messages to the objects in the application system. We recommend the use of the first approach, since the use case is captured in a class method, hence it can be reused, and it is easy to file out, file in and print out.

5.2.4 The Item Class

This class captures the common behavior of `ClassItem` and `MessageItem` used in `UseCaseView`. It maintains a name and the display mode. `ClassItem` and `MessageItem` are subclasses of `Item`.

5.2.5 The Operation Class

An instance of this class describes an operation. An operation maintains a sender which sends a message, a receiver which receives the message, a message name, the start and end time of the message, a display mode, and the location on the interaction diagram to display the message name of the operation. Double clicking on the location will pop up a method browser.

5.2.6 The Column Class

An instance of this class represents a column of an interaction diagram. The instance variable *title* describes the title of the column. Usually the name of a title is formed by prefixing the class name with an 'a' or 'an' if it begins with a vowel. The instance variable *operations* maintains the operations on this column. An operation on a column indicates a message send to the object described by the column. We added a flag, *interested*, for adding a filter on the columns for later extension of our system.

The helper of this class is *Operation*.

5.2.7 The InteractionDiagram Class

An instance of this class represents an interaction diagram. It maintains a title, a time, a date, and all the columns on the interaction diagram. The columns are represented as an *OrderedCollection*. The first column of the interaction diagram is always 'SystemBorder'. An *InteractionDiagram* knows how to add an *Operation* to a *Column* and sets the start and end time for that operation. It also knows how to display itself on a *GraphPane* and how to hide some operations while displaying. It may or may not include return arrows depending on the value of *returnArrow* flag.

An *InteractionDiagram* is generated incrementally through executing methods sent by an *IDGDebugger*. Each executing step of the *IDGDebugger* adds an *Operation* to the *InteractionDiagram* and a *Column* if needed. We described this in more detail in the section **The IDGDebugger Class**.

The helpers of this class include *Column* and *InteractionDiagramPane*.

5.2.8 The ScalePrompter Class

This is a subclass of *Prompter*. It is used to prompt the input of a scaling factor when printing an interaction diagram. The reason for subclassing is that the input field of *Prompter* is too long, but we only need a pop up window with a small input field which allows users to input a scaling factor.

5.2.9 The InteractionDiagramPane Class

The *InteractionDiagramPane* is a subclass of *GraphPane*. It maintains its own vertical and horizontal scroll bars. The horizontal scroll bar controls the scroll of both the *InteractionDiagramPane* and *DiagramTitlePane*. The *InteractionDiagramPane* refines some method in *GraphPane*, so it knows its title pane and two scroll bars, and controls the horizontal scroll for its title pane.

The *InteractionDiagramPane* contains instance variables *titlePane*, *vTitleScrollBar*, *vScrollBar*, and *hScrollBar*. It refines the method *scrollingRectangle* of *GraphPane* to answer the rectangle that contains the whole interaction diagram. And it

includes several new methods to maintain horizontal scroll bar, vertical scroll bar, and its title pane.

The reasons for subclassing *GraphPane* are as follows. First, *GraphPane* can only view a range of 9 times the size of the initially opened pane. We need to modify the view range according to the size of the interaction diagram. Second, when we resize the window, the picture will be displayed in the middle of the pane in *GraphPane*. However, we want our interaction diagram to be displayed at the upper left corner of the pane. Third, *GraphPane* initially put its scroll bars in the middle of its range. We want to put them at the upper left corner of the pane. Fifth, our graph pane needs to have the knowledge of the title pane.

When we vertically scroll up the diagram pane, everything will go up. If we don't have a separate pane for the title, the title of the interaction diagram may disappear, so we cannot view the object names. We don't want this to happen, therefore we keep the title in a separate pane. When we scroll the diagram, the title will remain in the title pane.

5.2.10 The IDGProcess Class

The *IDGProcess* is a subclass of *Process*. It inherits all the behavior from *Process*. The new behavior is that it has a new class method *queueWalkback:makeUserIF:resumable:useCaseView:*. This method will cause *InputEvent* to create an *IDGDebugger*.

5.2.11 The InputEvent Class

We added a new instance method *anIDGIn:label:useCaseView:* to class *InputEvent*. This method is similar to method *errorIn:label:*, but it creates an *IDGDebugger* instead of a *D-bugger*.

5.2.12 The UseCaseView Class

An instance of this class is a view of IDG. A *UseCaseView* is opened on a given Application. It controls all the panes and buttons in IDG. It knows how to create an *IDGDebugger* for the selected use case by sending message *queueWalkback: makeUserIF: resumable: useCaseView:* to *IDGProcess*. The behavior of its panes and the actions of clicking each button were described in the last chapter.

5.2.13 The CollaborationsView Class

An instance of this class represents a collaborations view. The behavior has been described in the last chapter.

5.3 Summary

In IDG we developed 14 classes. The main idea of the design is to modify the *Debugger* class in Smalltalk. The information needed to generate a interaction diagram is inherited from *Debugger*. The view of IDG is designed in the way that users can easily know how to use it and they can control how to show interaction diagrams easily. They can have different level of filters which allow them to understand the dynamic behavior of an application more quickly.

Chapter 6

Sample IDG Application

In this chapter we describe the usage of IDG with a sample IDG application. The example we use is a simple bank account program. The Smalltalk code corresponding to the application is shown in Appendix A.

In this example, we have five classes and four use cases. The classes are: *BankAccount*, *Transaction*, *CheckingTransaction*, *WithdrawalTransaction*, and *DepositTransaction*. The use cases are: *open an account*, *make a deposit*, *make a withdrawal*, and *write a check*. These use cases cover all the existing ways of using the system. The use case model is shown in figure 6.1.

6.1 The Usage of IDG

We have mentioned some uses of IDG in the earlier chapters. In this section we review the main uses once again.

6.1.1 Describing an Application

An application is an instance of *Application* class. It is created by *Application new: aName*. Here *aName* is the name of the application. Once we have an application, we can add classes and use cases to it using *addClass: aClassName*, and *useCases: aUseCase*. Or we can remove them using *removeClass: aClassName* and *removeUseCase: aUseCaseName*. A more convenient way to add classes to or remove classes

Figure 6.1: Use Case Model for Simple Bank Account System

from an application is to do it within IDG. We can pop up a menu in *Class Pane* and select *add* or *remove*.

In our simple bank account example, we create an application by saying:

```
|app|  
app := Application new: 'Simple Bank Account'.
```

We can then add class *BankAccount* and use case *aUseCase* by saying:

```
app addClass: #BankAccount.  
app useCase: aUseCase.
```

6.1.2 Describing a Use Case

We describe a use case by creating an instance of *UseCase*. We can either write the use case we want to define as a class method in some class, or write it directly

as a string. The class method we should use is *UseCase* name: *aName* code: *useCaseCode*. For the former, *useCaseCode* should be `'aClassName aMethodName'`. For example,

```
aUseCase := UseCase name: 'Open an Account'
              code: 'BankAccount egCreateAccount'.
```

Here, the class method *egOpenAnAccount* of class *BankAccount* contains the code to open an bank account. For the latter, *useCaseCode* should be a string. Don't forget to make one quote to two quotes if original Smalltalk/V code contains quotes. For example,

```
aUseCase := UseCase name: 'Make a deposit'
              code: ' |jimsAccount|
                    jimsAccount := BankAccount new.
                    jimsAccount owner: ''jim'';
                    accountNumber: 1234.
                    jimsAccount addTransaction:
                        (DepositTransaction new amount: 1000.00).
                    ^jimsAccount'.
```

After we get an instance of *UseCase*, we can add it to our application by sending the message *useCase: aUseCase* to the application.

Use cases are very important during a system's life cycle. We define them during the analysis and design, and use them during implementation, testing, and maintenance. When we extend our system later on, we can add new use cases to the system. They are also useful for regression testing. We only describe each use case once, so take the time to define them well. We can reuse the code for use cases if their code is defined as class methods in other classes by giving that class method as the *useCaseCode*.

6.1.3 Open an IDG

An IDG is an instance of *UseCaseView*. When we have an application, we can open an IDG by executing *UseCaseView new openOn: anApplication*. We will see an Interaction Diagram Generator window in the middle of our screen. We can close it just like we close other window applications. In our example, we open an IDG by saying:

```
UseCaseView new openOn: app.
```

6.1.4 Filters

Filters can be added at the class and method level. We can turn on or turn off a class by double clicking on the classes in the *class pane* or on the methods in the *method pane*. We will see the change of *On-Off* indicator. We can choose only to list the classes or methods which are turned on by selecting *On classes* or *On methods* in the pop up menus. In this case, the *On-Off* indicator will not be shown. See figure 6.2 and figure 6.3

These figures show that the class *CheckingTransaction* and instance method *initialize* of class *BankAccount* are turned off. The objects from the class and *initialize* message sent to the object of *BankAccount* will not be shown in the interaction diagrams.

When an interaction diagram is displayed, we can hide an operation by double clicking on the message sending arrow to the operation. This will also hide the subsequent operations generated by that operation. A little black square will be displayed at the place where the message sending arrow used to be. Figure 6.4 shows the interaction diagram for use case *make a deposit*. The snapshot was taken after we had double clicked on the *new* message sending arrow to *DepositTransaction* and the *applyTo:* message sending arrow to *aDepositTransaction*. By double clicking on the little black square, we can bring those operations back. We can hide as many operations as we want.

Figure 6.2: Filter on Class and Method (with on-off indicator)

Figure 6.3: Filter on Class and Method (without on-off indicator:)

Figure 6.4: Hiding Operations On An Interaction Diagram

6.1.5 Bring Up Collaborations View

A collaborations view will pop up if the *Collaborations* button is clicked. We can view the collaborations between classes based on any combinations of use cases. If we want the collaborations view covers one use case, we should assign an interaction diagram file to that use case. If that file has not been created, then we should run that use case in IDG, and save it to a file. After that we can assign that interaction diagram file to the use case.

6.1.6 Opening a Method Browser

A Method Browser can be opened by double clicking on a message name shown on an interaction diagram. This browser shows us the implementation of that method.

6.1.7 Saving and Loading an Interaction Diagram

Saving and loading an interaction diagram is achieved by clicking the *Save* or *Load* button. If a use case is selected in the *use case pane* and we load an interaction diagram file, then that file will be assigned to the selected use case. We should have an interaction diagram in *interaction diagram pane* while we save it. The default file extension of an interaction diagram file is *.ind*.

6.1.8 Printing an Interaction Diagram

We can print out an interaction diagram currently displayed in interaction diagram pane in bitmap form. A prompter will pop up and ask us for a scaling factor. The scaling factor can be any number. For example, input 1, if we want a printout with the same size of what we see on the screen; input 2, if we want to get a printout which doubles the size; input 1.5, if we want to get a printout which is one and a half of the size on the screen.

6.2 Using IDG to Analyse A Simple Bank Account Program

In this section we analyse the result of applying IDG to our simple bank Account example.

We open an IDG by executing the following codes.

```
|app aUseCase|
app := Application new: 'Simple Bank Account'.
app addClass: #BankAccount.
app addClass: #Transaction.
app addClass: #CheckingTransaction.
app addClass: #DepositTransaction.
app addClass: #WithdrawalTransaction.

aUseCase := UseCase name: 'Open an Account'
codes: 'BankAccount egOpenAnAccount'.
app useCase: aUseCase

aUseCase := UseCase name: 'Make a deposit'
codes: '    | jimsAccount |
jimsAccount := BankAccount new.
jimsAccount owner: ''jim'';
    accountNumber: 1234.
jimsAccount addTransaction:
    (DepositTransaction new amount: 1000.00).
    ^jimsAccount'.
app useCase: aUseCase.

aUseCase := UseCase name: 'Make a withdrawal'
codes: 'BankAccount egMakeWithdrawal'.
```

```
app useCase: aUseCase.
```

```
aUseCase := UseCase name: 'Write a check' codes: 'BankAccount egWriteCheck'.  
app useCase: aUseCase.
```

```
UseCaseView new  
  openOn: app.
```

Since this is just a simple example, we didn't save any data to a database or a file. We always open an account for each transaction.

Appendix B shows the interaction diagrams generated by IDG for the four use cases.

From these diagrams we can clearly see the dynamic behavior of this bank account system. For example, when we look at the diagram for the use case *make a deposit*. We can see that we first send a message *new* to *BankAccount* class to create *aBankAccount*. *BankAccount* class creates and initializes *aBankAccount*. Then we tell *aBankAccount* the owner, and the account number¹. After that we ask *DepositTransaction* class to create *aDepositTransaction*, and pass the amount that we want to deposit to *aDepositTransaction*. We can see that *DepositTransaction* class asks *aDepositTransaction* to initialize itself. Then we ask the *BankAccount* to add the *DepositTransaction*. For adding a transaction, *aBankAccount* sends message *applyTo:* to *aDepositTransaction*. We can get the argument and the code for *applyTo:* by double clicking on them to get a method browser in IDG. Now *aBankAccount* asks *aDepositTransaction* to apply the deposit amount to its balance. *aDepositTransaction* then asks the balance in *aBankAccount*, and get the deposit amount from itself. Finally, *aDepositTransaction* sends message *balance:* with the new amount to *aBankAccount*.

We can also view the collaborations between classes generated by running through all these use cases. Figure 6.5 to figure 6.9 shows the collaborations between all the five classes.

¹In our example, customer can choose their own account number

Classes	Helpers	Users
BankAccount CheckingTransaction DepositTransaction SystemBorder Transaction WithdrawalTransaction	CheckingTransaction DepositTransaction WithdrawalTransaction	CheckingTransaction DepositTransaction SystemBorder WithdrawalTransaction

Figure 6.5: Collaborations on BankAccount

These collaborations viewed together with the interaction diagrams generated for the use cases clearly describes the dynamic behavior of the system. They can be used to understand the application system or capture the design information of the system. We can compare them with our design documents, and find the difference between the design and final implementation to make necessary changes to the design documents.

IDG can be used as a debugging tool during the implementation of the bank account program. For example, we know that when we make a deposit transaction, the balance should be increased by the deposit amount. That must be shown on the interaction diagram of the use case *make a deposit*. If it is not shown, then we know that there must be some error in the operation *addTransaction*: or its subsequent operations. We can make the necessary changes to some methods and rerun the use case from beginning without leaving the IDG.

6.3 Summary

Interaction Diagram Generator can generate interaction diagrams for use cases of a running Smalltalk/V application. We can add filters on class level or method level. IDG also provides a collaborations view from which we can view the collaborations between classes. IDG is integrated with Smalltalk/V environment. Changes to the Smalltalk/V code will be reflected on the subsequent generated interaction diagrams.

Classes	Helpers	Users
BankAccount CheckingTransaction DepositTransaction SystemBorder Transaction WithdrawalTransaction		

Figure 6.6: Collaborations on Transaction

Classes	Helpers	Users
BankAccount CheckingTransaction DepositTransaction SystemBorder Transaction WithdrawalTransaction	BankAccount	BankAccount SystemBorder

Figure 6.7: Collaborations on DepositTransaction

Classes	Helpers	Users
BankAccount CheckingTransaction DepositTransaction SystemBorder Transaction WithdrawalTransaction	BankAccount	BankAccount

Figure 6.8: Collaborations on CheckingTransaction

Classes	Helpers	Users
BankAccount CheckingTransaction DepositTransaction SystemBorder Transaction WithdrawalTransaction	BankAccount	BankAccount

Figure 6.9: Collaborations on WithdrawalTransaction

We used IDG to analyse a simple bank account program. Although the application system is very simple, it shows that IDG provides us a tool which can show the dynamic behavior of a system. It records and shows all the message sends while running a use case. We can also see the collaborations between classes in a system. In addition, we can use IDG as an alternative debugger.

Chapter 7

Summary and Future Work

7.1 Conclusion

The problem addressed in this thesis is to develop a tool which can be used to capture the design information from a running system and help to understand the dynamic behavior of object-oriented systems. From the study of some object-oriented system development methods, we believe that use cases and interaction diagrams are powerful tools to be used. Use cases and their interaction diagrams describe the behavior of a system, and can be used in system analysis, system design, through system testing and maintenance. Therefore, we decided to develop a tool which can generate interaction diagrams for use cases of a running system. From the interaction diagrams, we can see the dynamic behavior of a system and thus understand the system. As result we built an IDG system successfully.

The idea of implementing the IDG is to get needed information from debugger. When running a use case, IDG will create a process on the use case, open a debugger on that process, and generate the interaction diagram incrementally. The interaction diagram is shown in a graph pane and can be saved or printed out. IDG also provides a collaborations view. From that view, users can see the collaborations between classes.

IDG records all the message sends between objects while running a use case and show them as an interaction diagram with a time thread. We believe that IDG can

be used as a reengineering tool to capture the design information of object-oriented systems, and may help people to understand the dynamic behavior of object-oriented systems. It can be used as a teaching aid since it can show message sends from one object to another object step by step. From this, students can learn how an object-oriented system works. It can also be used as a verification tool to verify implementation against design documentation. It can be used in debugging use cases and regression testing. Integrating IDG with other tools now being developed by C'OOOP will provide a powerful environment for developing object-oriented systems in Smalltalk.

7.2 Future Work

Our IDG is a prototype. We have implemented the basic functions and showed that it is a useful tool. However, there are many improvements and extensions which can be made. For example, we can add some graphical features like zooming, resizing, and coloring. The following is a proposed list of improvements and extensions.

- The current implementation of IDG shows objects from same class in the same column. It may get confusing if two objects from same class are communicating. In this case, it is better to show these two objects as different columns. This can be done by extending the *Column* class to include information about object. We can get the information from the *Debugger* class. It is saved in variable *object* (inherited from *Inspector*). Objects are unique in the system so that we can distinguish one object from another even if they are from same class.
- Currently, a generated interaction diagram does not have label on the return arrow which shows what is returned. It would be better to label what is returned. The difficulty of labeling return arrow with result is that now we only have the information at method level which means that we know what method of what class is called by another method of another class, but we

don't know the returning information. This may need more exploration of how the *Debugger* works.

- Now the use cases can only be added to an application by code. It is not difficult to add a pop up menu to *Use Case Pane* which allows users to add or delete use cases interactively through menu.
- Currently, we use the existing Smalltalk/V method to print an interaction diagram, it would be better to print an interaction diagram in a postscript format instead of bitmap format so that they may be embedded more easily in system documentation.
- Currently, our IDG can save or load an application. It would be better to make applications persistent and provide a browser that can browse all the applications. Then we can open an IDG through system menu.
- As we mentioned earlier, collaborations graphs show collaborations between classes in graphical form. They summarize the information shown in our collaborations view in graphs. It would be nice to generate this kind of collaborations graphs. Since we already have the collaboration information, the only thing left is to draw them as graphs, although laying out such graphs remaining a problem. One way to solve the problem is to draw the graph in a possible way, and let user to modify it. Users have the ability to move the elements and resize the elements.
- A powerful system development tool should support system development in both forward and backward directions. We can add an Interaction Diagram Editor which can be used to generate interaction diagrams by users, or edit the existing interaction diagrams generated by either IDG or users.
- Another thing we should do is to support applications with user interfaces. Therefore, we may see the application running as well as the interaction diagram generating. Currently our IDG can't control this kind of application

well. Sometimes we can't get the control back from the application to our IDG.

Appendix A

Smalltalk Code for A Simple Bank Account Program

```
Object subclass: #BankAccount
```

```
    instanceVariableNames:
```

```
        'owner accountNumber balance transactions '
```

```
    classVariableNames:
```

```
    poolDictionaries:
```

```
BankAccount class methods
```

```
new
```

```
    ^super new initialize
```

```
BankAccount methods
```

```
accountNumber
```

```
    ^accountNumber
```

```
accountNumber: aNumber
```

```
    accountNumber := aNumber
```

Appendix A. Smalltalk Code for A Simple Bank Account Program 80

```
addTransaction: aTransaction
    transactions add: aTransaction.
    aTransaction applyTo: self.

balance
    ^balance

balance: anAmount
    balance := anAmount

initialize
    accountNumber := 999.
    balance := 0.
    owner := 'Unknown'.
    transactions := SortedCollection new
        sortBlock: [:transaction1 :transaction2 |
            transaction1 date < transaction2 date]

owner
    ^owner

owner: aName
    owner := aName

printOn: aStream

    aStream nextPutAll: 'a BankAccount with name: '.
    self owner printOn: aStream.
    aStream nextPutAll: ' number: '.
```

Appendix A. Smalltalk Code for A Simple Bank Account Program 81

```
self accountNumber printOn: aStream.  
aStream nextPutAll: ' balance: '.  
self balance printOn: aStream
```

transactions

```
^transactions
```

Object subclass: #Transaction

```
instanceVariableNames:
```

```
'amount date '
```

```
classVariableNames:
```

```
poolDictionaries:
```

Transaction class methods

new

```
^super new initialize
```

Transaction methods

amount

```
^amount
```

amount: anAmount

```
amount := anAmount
```

Appendix A. Smalltalk Code for A Simple Bank Account Program 82

applyTo: aBankAccount

 "For check and withdrawal only"

 aBankAccount balance:

 (aBankAccount balance - self amount).

date

 ^date

date: aDate

 date := aDate

initialize

 amount := 0.

 date := Date today.

printOn: aStream

 aStream nextPutAll: 'Date: '.

 self date printOn: aStream.

 aStream nextPutAll: ' ', (self class name), ' \$'.

 self amount printOn: aStream

Transaction subclass: #CheckingTransaction

 instanceVariableNames:

 'checkNumber '

 classVariableNames:

 poolDictionaries:

CheckingTransaction class methods

CheckingTransaction methods

checkNumber

^checkNumber

checkNumber: anInteger

checkNumber := anInteger

initialize

super initialize.

checkNumber := 0

printOn: aStream

aStream nextPutAll: 'Date: '.

self date printOn: aStream.

aStream nextPutAll: ' Check # '.

self checkNumber printOn: aStream.

aStream nextPutAll: ' \$'.

self amount printOn: aStream

Appendix A. Smalltalk Code for A Simple Bank Account Program 84

Transaction subclass: #DepositTransaction

instanceVariableNames:

classVariableNames:

poolDictionaries:

DepositTransaction class methods

DepositTransaction methods

applyTo: aBankAccount

aBankAccount balance:

(aBankAccount balance + self amount)

2 OF/DE 2

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

PRECISIONSM RESOLUTION TARGETS

Appendix B

Interaction Diagrams for Simple Bank Account System

Appendix B. Interaction Diagrams for Simple Bank Account System 86

<< Generated on Nov 14 1993 21:54:15 >>

Open an Account

SystemBorder

BankAccount

aBankAccount

Appendix B. Interaction Diagrams for Simple Bank Account System 87

<< Generated on Nov 14, 1993 21:50:16 >>

Make a deposit

SystemBorder

BankAccount

aBankAccount

DepositTransaction

aDepositTransaction

Appendix B. Interaction Diagrams for Simple Bank Account System 88

<< Generated on Nov 14, 1993 21:48:30 >>

Write a check

SystemBorder

BankAccount

aBankAccount

CheckingTransaction

aCheckingTransaction

Appendix B. Interaction Diagrams for Simple Bank Account System 89

<< Generated on Nov 14, 1993 21:51:29 >>

Make a withdrawal

SystemBorder

BankAccount

aBankAccount

WithdrawalTransaction

aWithdrawalTransaction

Bibliography

- [BH 90] Heinz-Dieter Bocker and Jurgen Herczeg. TRACK - A Trace Construction Kit. *CHI'90 Conference Proceedings*, Seattle, Washington. April 1990. Pages 415-422.
- [Big 89] T. J. Biggerstaff. Design Recovery for Maintenance and Reuse. *Computer*. July 1989. Pages 36-49.
- [Booch 91] Grady Booch. *Object Oriented Design with Applications*. RedWood City, California: Benjamin/Cummings. 1990.
- [CB 86] Ward Cunningham and Kent Beck. A Diagram for Object-Oriented Programs. *OOPSLA'86 Conference Proceedings, September 1986*. Pages 361-367.
- [CC 90] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Softwar*. January 1990. Pages 13-18.
- [CY 91] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Englewood Cliffs, New Jersey: Yourdon Press, Prentice Hall. 1991.
- [CY 91] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Englewood Cliffs, New Jersey: Yourdon Press, Prentice Hall. 1991.
- [GB 89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison Wesley. 1989.

- [Hend 1992] Brian. Henderson-Sellers. *A Book OF Object-Oriented Knowledge*. Prentice Hall, 1992.
- [Jaco 92] Ivar Jacobson. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Wokingham, England: Addison-Wesley, 1992.
- [KG 88] Michael F. Kleyn and Paul C. Gingrich. GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views. *OOP-SLA '88 Conference Proceedings, September 1988* Pages 191-205.
- [Loomi 93] Mary E.S. Loomis. Object and relational technologies. *Object Magazine*, January-February 1993. Pages 35-40.
- [LP 90] Wilf R. LaLonde and John Pugh. *Inside Smalltalk Volume I*. Englewood Cliffs, New Jersey: Prentice Hall, 1990.
- [LPWC 93] Wilf LaLonde, John Pugh, Paul White, and Jean-Pierre Corriveau. Towards Unifying Analysis, Design, and Implementation in Object-Oriented Environments. *Proceeding of East EWOOPPE'93, Bratislava, Slovakia, Nov. 1993*.
- [RBPEL 91] James Rumbaugh, Michael Blaha, William Premerian, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.
- [SC 86] R. G. Smith and P. J. Carando. *Structured Object Programming In Strobe*. Research Note SYS-86-26, Schlumberger-Doll Research, October 1986.
- [SDB 86] R. G. Smith, R. Dinitz, and P. Barth. Impulse-86: A Substrate for Object-Oriented Interface Design. *OOPSLA '86 Conference Proceedings*, September 1986. Pages 167-176.
- [WWW 90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Englewood Cliffs, New Jersey: Prentice Hall, 1990.

END

0 8-1 1-94

FIN