1

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

PRECISIONSM RESOLUTION TARGETS

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

On-Line Dynamic Views of Applications Written in Smalltalk

by

Jeffrey C. Alguire, B.C.S.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment 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

December 12, 1994

© copyright

1994, Jeffrey C. Alguire

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

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

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

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

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

ISBN 0-315-98566-6

Canada

Name JEFFREY C. ALGUIRE

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 U·M·I

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

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

EDUCATION

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

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

LANGUAGE, LITERATURE AND LINGUISTICS

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

PHILOSOPHY, RELIGION AND THEOLOGY

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

SOCIAL SCIENCES

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

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 0615
General 0616
International Law and Relations 0617
Public Administration 0814
Recreation 0452
Social Work 0626
Sociology 0627
General 0938
Criminology and Penology 0631
Demography 0628
Ethnic and Racial Studies 0629
Individual and Family Studies 0630
Industrial and Labor Relations 0700
Public and Social Welfare 0344
Social Structure and Development 0709
Theory and Methods 0999
Transportation 0453
Urban and Regional Planning 0453
Women's Studies

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and Nutrition 0476
Animal Pathology 0359
Food Science and Technology 0478
Forestry and Wildlife 0479
Plant Culture 0480
Plant Pathology 0817
Plant Physiology 0777
Range Management 0746
Wood Technology 0306
Biology 0287
General 0308
Anatomy 0309
Biostatistics 0379
Botany 0329
Cell 0353
Ecology 0369
Entomology 0793
Genetics 0410
Limnology 0307
Microbiology 0317
Molecular 0416
Neuroscience 0433
Oceanography 0821
Physiology 0778
Radiation 0472
Veterinary Science 0786
Zoology 0760
Biophysics 0425
General 0996
Medical

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

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 0566
General 0300
Audiology 0992
Chemotherapy 0567
Dentistry 0350
Education 0769
Hospital Management 0758
Human Development 0982
Immunology 0564
Medicine and Surgery 0347
Mental Health 0369
Nursing 0570
Nutrition 0380
Obstetrics and Gynecology 0354
Occupational Health and Therapy 0381
Ophthalmology 0571
Pathology 0419
Pharmacology 0572
Pharmacy 0382
Physical Therapy 0573
Public Health 0574
Radiology 0575
Recreation

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

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

Applied Sciences

Applied Mechanics 0346
Computer Science 0984

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

PSYCHOLOGY

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

**The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of this thesis,**

**On-Line Dynamic Views of Applications
Written in Smalltalk**

submitted by

Jeffrey C. Alguire, BCS

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

Thesis Supervisor

Director, School of Computer Science

**Carleton University
January 23, 1995**

Abstract

The application of the object-oriented paradigm entails the distribution of code among collaborating objects. When used well, it can facilitate real-world modelling and code reuse. However, code distribution tends to complicate the task of understanding how an application works. An interaction diagram [Jaco 92] is a visual representation of the objects collaborating to accomplish a well-defined task (called a use case) relating to an application, and the messages sent in order to carry out the use case. Wang has written an "interaction diagram generator," running under Smalltalk V/Windows 2.0, which displays such diagrams based on the messages produced as the use case runs [Wang 94].

As application development proceeds, though, evaluation of collaboration patterns and comparisons with old patterns become helpful in the identification of programming deficiencies and the monitoring of development progress. In recognition of the importance of evaluation and comparison to development, this thesis, along with direct extensions to Wang's work which include the addition of the ability to visualize use cases involving graphical user interfaces, introduces comparison and evaluation tools for interaction diagrams.

Acknowledgements

This thesis would not have been possible without the support, guidance and patience of my supervisor, Professor John Pugh. All of his help is greatly appreciated. In addition, I would like to thank all of the faculty and students who have been associated with Carleton University's Centre for Object-Oriented Programming during my period of membership, which began in May, 1993. Their suggestions were quite helpful.

The financial support for this work, provided by the Natural Sciences and Engineering Research Council (NSERC) and the School of Computer Science, is gratefully acknowledged.

As well, I would like to thank my manager at my place of employment (Bell Northern Research), Ms. Lynda Timmerman, who has been supportive of my completion of this thesis on a part-time basis since I joined the company in September, 1994. Her flexibility with respect to the scheduling of my working hours, and my use of vacation time, have been very important in enabling me to finish the thesis as quickly as I have.

Also, I would like to thank my parents whose support, encouragement and assistance have never failed, even under the pressures of my academic pursuits. Their faith in me means so much.

Finally, I would like to thank God for empowering me to do good things, and for being with me always. To Him, His Son, and the Holy Spirit be the glory forever, Amen.

Trademarks

80086, 80286, 80386 and 80486 are registered trademarks of Intel Corporation.

ENVY is a registered trademark of Object Technology International Inc.

MacIntosh is a registered trademark of Apple Computer Inc.

Smalltalk/V is a registered trademark of Digitalk Inc.

Subpanes/V is a trademark of Cooper and Peters, Inc.

Windows is a registered trademark of Microsoft Corporation.

WindowBuilder is a trademark of Cooper and Peters, Inc.

XRunner and WinRunner are registered trademarks of Mercury Interactive Corporation.

Other products may be trademarks of their respective companies.

Table of Contents

1. Introduction.....	1
1.1 Object-Oriented Technology.....	1
1.2 Design Methodologies and Visual Representations.....	3
1.3 OOSE, Interaction Diagrams and Use Cases.....	3
1.4 Automatic Generation - The Original Generator.....	5
1.5 Object-Oriented Development, Change and Evaluation.....	6
1.6 The NSERC Project.....	8
1.7 Thesis Outline.....	8
 2. Background.....	 10
2.1 Object-Oriented Software Engineering.....	10
2.2 The Original Interaction Diagram Generator.....	14
2.2.1 Functionality and User Interface.....	14
2.2.1.1 Functionalities and the Justifications.....	14
2.2.1.2 User Interface.....	18
2.2.2 Selected Implementation Details.....	23
2.2.2.1 Using System-Maintained Information Through Inheritance: IDGDebugger and IDGProcess.....	23
2.2.2.2 Application and UseCase.....	26
2.2.2.3 InteractionDiagram, Column and Operation.....	26

2.2.2.4 Implementation Summary.....	28
2.3 Summary.....	28
3. Regeneration.....	30
3.1 What is Regeneration?.....	30
3.2 The Basic Regeneration Functionality and Interface.....	31
3.2.1 Initiating a Regeneration.....	31
3.2.2 Comparison Results and Their Diagrammatical Presentation.....	35
3.2.3 The Textual Log.....	37
3.3 Interpretations and Comparisons of Interaction Diagrams.....	38
3.4 Obsolete Filters and Changes in Class Structure.....	42
3.4.1 Definitions and Tools for Adjustment.....	42
3.4.2 Specifying Mappings.....	44
3.5 Summary.....	47
4. Generating Diagrams for Applications with Graphical User Interfaces (GUIs).....	50
4.1 Rationale.....	50
4.2 Applications Using GUIs but No Dialog-Type Boxes.....	50
4.3 Applications Using Dialog-Type Boxes.....	55
4.4 Summary.....	57
5. Identifying, Differentiating and Viewing Application-Related Objects.....	58
5.1 What Objects should be Identified?.....	58
5.2 Distinguishing Among Objects of the Same Class.....	59

5.3 Arguments and Returned Objects.....	60
5.3.1 The Process Stack.....	60
5.3.2 Arguments.....	61
5.3.3 Returned Objects.....	62
5.3.3.1 Isolating Executing Code Segments.....	63
5.3.3.2 The Central Part of Returned Object Extraction.....	66
5.3.3.3 Handling the "hasBlock" Condition.....	67
5.3.4 Pointers Versus Copies.....	68
5.3.5 Interface.....	70
5.4 Statement of Intent: Viewing Only, Not Debugging.....	73
5.5 Summary.....	74
 6. Other Improvements.....	 76
6.1 The Applications Editor.....	76
6.2 Zooming.....	80
6.3 Performance Improvements.....	82
6.4 Redirected Printing.....	83
6.5 Summary.....	83
 7. Sample IDG Applications.....	 85
7.1 The Applications.....	85
7.1.1 The "Simple Bank Account" Application.....	86
7.1.2 The "CPLauncher" Application.....	87

7.2 Basic Usage.....	88
7.2.1 Building an Application.....	88
7.2.2 Generating and Saving an Original Diagram.....	89
7.2.3 Diagram Evaluation, Regeneration and Comparison.....	92
7.3 Usage with Applications Involving GUIs.....	92
7.4 Summary.....	93
8. Conclusions and Future Work.....	95
8.1 Conclusions.....	95
8.2 Future Work.....	97
 Appendix A Smalltalk Code for a Simple Bank Account Management System (Application Code and Use Case Definitions).....	 99
A.1 Application Code.....	99
A.1.1 The BankAccount Class.....	99
A.1.2 The Transaction Class and its Subclasses.....	102
A.2 Use Case Definitions.....	104
A.2.1 "Make a deposit".....	104
A.2.2 "Make a withdrawal".....	105
A.2.3 "Open an Account".....	105
A.2.4 "Simulated transfer".....	105
A.2.5 "Write a check".....	105

Appendix B Interaction Diagrams for a Simple Bank Account Management System	106
B.1 "Make a deposit".....	106
B.2 "Make a withdrawal"... ..	107
B.3 "Open an Account".... ..	108
B.4 "Simulated transfer".....	109
B.5 "Write a check".....	110
Bibliography.....	111

List of Figures

1.1 A Simple Interaction Diagram Example.....	4
1.2 A Standard Smailtalk Debugger Window.....	5
2.1 A Use Case Model.....	11
2.2 Main Interaction Diagram Generator Window (IDGView).....	19
2.3 A MethodBrowser.....	21
2.4 A CollaborationsView Focussed on the BankAccount Class.....	23
3.1 New IDGView.....	32
3.2 An Interaction Diagram File Dialog.....	34
3.3 An Interaction Diagram Delta View.....	36
3.4 A Delta View focussed on the First Difference.....	36
3.5 A Text (log) View of Interaction Diagram Differences.....	38
3.6 A "Fork" Type Interaction Diagram with SystemBorder as the Controller.....	40
3.7 A "Stair" Type Interaction Diagram.....	40
3.8 An OutOfDateFilterDialog.....	46
3.9 An ObjectLoadDialog (from Digitalk).....	47

4.1 The UseCaseView>>perform Method - (a) original (b) new.....	52
4.2 Code for a Use Case Involving a Graphical Interface.....	53
4.3 Code for a Use Case Involving a Prompter.....	56
5.1 Code for Extracting the Process Stack from the System.....	60
5.2 Code for Extracting Arguments to the Current Method from the Process Stack and Placing them (in Order of Occurrence) into argumentsArray.....	61
5.3 The IDGDebugger>>walkback: Method.....	64
5.4 The Last Part of the drawDiagramMethodPositionStart:end: Method.....	66
5.5 The Code which Extracts Returned Objects from Blocks when the Method Returning "hasBlock".....	68
5.6 New IDGView (Repeated from Fig. 3.1).....	69
5.7 Main Interaction: Diagram Generator Window (Maximized and then Reduced in Size as a Bitmap) with Return Arrows Showing, the owner: Operation Selected and the Popup Menu for the Diagram Pane Displayed.....	71
5.8 An OperationArgumentsBrowser on BankAccount>>owner:.....	72
6.1 An ApplicationsEditor.....	77
6.2 A Zoomed IDGView.....	81
7.1 Use Case Model for the Simple Bank Account Management System.....	86
7.2 A CPLauncher Window.....	87
7.3 A CPCClassAnalyzerWindow.....	88
7.4 An ApplicationsEditor from which the User is about to Initiate an IDGView on the "Simple Bank Account" Application.....	89

7.5 An IDGView After the Generation of an Interaction Diagram for the "Make a withdrawal" Use Case of the "Simple Bank Account" Application.....	90
7.6 An InteractionDiagramFileDialog Prepared to Save a Diagram Associated with the "Make a deposit" Use Case of the "Simple Bank Account" Application.....	91
7.7 An IDG-Generated Diagram for the "CPLauncher to Class Analyzer" Use Case of the "CPLauncher" Appliction.....	94

Chapter 1

Introduction

1.1 Object-Oriented Technology

In object-oriented programming, an application is modelled as a group of objects which collaborate, through message-passing, to present the desired functionality. For example, a simple bank account application may use objects such as bank accounts and transactions. A bank account can be expected to have an owner, a balance and a collection of transactions. Transactions may have a date and an amount. Transactions and bank accounts may collaborate to produce the effect of a deposit (adding the amount of a transaction, which is extracted by sending a message to a transaction asking for its amount, to a bank account balance). The most common messages are those which either request information or request a modification to information contained in an object. Notice the use of the word "request." This means that the object receiving the request has the right to refuse. The idea that objects control their own (state) information is referred to as the principle of **encapsulation**.

The increasing prevalence of object-oriented programming in both the industrial and academic communities can be attributed to certain advantages which it holds over the procedural model. The two most important advantages are:

● *Reuse through inheritance and the "parts relationship"*

Object-Oriented languages use some form of inheritance. In the majority of cases, including Smalltalk V/Windows (the language in which the software implemented for the purpose of this research is written), the inheritance is class-based. This means that objects belong to classes, and instance (state) variables and methods (behaviour) for objects of a given class consist of that which is defined by the class and its ancestors in the class hierarchy (eg. a class Car may inherit from class Vehicle which may inherit from a general class, Object). In addition, the "parts relationship" denotes one object being part of another (eg. a tire may be part of a car or a bicycle). Both inheritance and parts relationships can *potentially* lead to greater code reuse, when hierarchies are appropriately built and objects are designed to readily become parts of other objects.

● *Improved real-world modelling*

Modeling is made easier and more accurate when code is associated with semantic objects that have real-world equivalents.

Also, some languages, like Smalltalk, include an extensive library of pre-defined and pre-validated classes, which, once learned, greatly facilitates the writing of application software.

One problem with this approach, however, is that since code is widely distributed among many (hopefully largely reusable) objects, it is often difficult to gain a thorough understanding of the interactions and collaborations among objects that yield a working application.

1.2 Design Methodologies and Visual Representations

Several object-oriented design methodologies have been put forward. These include Responsibility-Driven Design (RDD) [WWW 90], Object-Oriented Design (OOD) [Booch 91], Object Modeling Technique (OMT) [RBPEL 91] and the combination of Object-Oriented Analysis (OOA) [CY 91a] and its related design methodology [CY 91b]. In recognition of the magnitude of the difficulties involved in understanding the interactions among objects that come together to form a working application, visual representations of object-oriented systems have also been devised. Visualization is a tool that has been used in many different realms to reduce complexity and improve comprehension. Accordingly each of the above-mentioned methodologies includes some form of visual representation. Jacobson provides a summary of these methodologies, among others, in [Jaco 92, pp. 465-493]. Primarily, though, Jacobson's book presents his own methodology, Object-Oriented Software Engineering (OOSE). This methodology forms the basis for the software tools described in this thesis.

1.3 OOSE, Interaction Diagrams and Use Cases

At the heart of OOSE is the concept of the 'use case.' A use case consists of a set of inputs to a system where the expected response is well-defined. The messages generated by a use case can be depicted in an 'interaction diagram,' which uses vertical lines to represent objects participating in a use case, and horizontal arrows to represent messages that flow amongst them. Such diagrams provide a relatively convenient mechanism through which to view several message interactions at once, as opposed to the message-by-message view yielded by the Smalltalk V/Windows text-based debugger, for example. A simple interaction diagram, depicting a deposit transaction in a 'Simple

Bank Account' application, is shown in Fig. 1.1. A Smalltalk V/Windows standard text-based debugger window is shown in Fig. 1.2.

Despite the utility of the representation, a problem remained. Like many other graphical views before it (regardless of the domain being modelled), the production of the diagrams was time-consuming and error-prone. A way to improve the reliability and efficiency of diagram production was needed.

FIGURE 1.1: A Simple Interaction Diagram Example

FIGURE 1.2: A Standard Smalltalk Debugger Window - One can look at only one message at a time (by clicking on it in the upper left hand list pane); there are no tools in standard Smalltalk for system assisted versioning and comparison of message-sending patterns.

1.4 Automatic Generation -The Original Generator

It seemed that, in systems where code is actually run in an object-oriented manner using message-passing, as opposed to systems that do preprocessing on object-oriented code in order to run a functionally equivalent procedural version, it should be possible to extract the message-related information that the system produces as the application runs. Having that information, it would be possible to construct the interaction diagram related to the use case being run.

Proving this point for the Smalltalk V/Windows environment, Wang built an 'interaction diagram generator,' which extracts information from the Smalltalk debugger and uses it to produce and display the interaction diagram associated with a use case, while it is running [Wang 94]. It also permits the loading, saving and printing of

diagrams, and provides the ability to operate in a step-wise manner when desired - that is, the ability to proceed one message at a time, waiting for the user's signal before continuing.

However, Wang's generator does have its limitations. For example, it cannot distinguish among objects of the same class; all objects of a given class are represented by a single vertical line. Although this is not something that Jacobson appears to have concerned himself with in his definition of "interaction diagram" [Jaco 92], it can confuse and/or mislead users when dealing with use cases that employ multiple instances of a given class. Another problem is that it cannot generate diagrams for use cases which involve graphical user interfaces (GUIs). Since GUIs are common and becoming increasingly so, this is a serious restriction on the utility of the original generator. These issues are addressed in this thesis, but results obtained in these areas are not its most important contributions.

1.5 Object-Oriented Development, Change and Evaluation

As noted in [Wang 94, p. 3]:

The object-oriented software development life cycle does not follow the traditional waterfall model 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 identified in later phases causing ultimately 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.

Instead of following a waterfall model that moves in a sequence like [requirements analysis, general design, detailed design, implementation, testing], with rigid boundaries between phases, object-oriented development is an incremental process where all phases

progress concurrently. A developer works on small parts of a system at a given time, continually performing activities which would have been previously ascribed to each of the phases of the waterfall model, making changes as appropriate.

While this incremental approach is better because it is more resilient to change, one must account for the quicker pace of change to which it gives rise. Aside from the direct limitations on diagram generation discussed in the previous section, the original generator is not sufficient for the purpose of monitoring change and progress. To do this, one still needs visualizations to facilitate understanding, but one needs more than diagrams showing the messages generated in running use cases at the time when they are executed. In the same way as some source code managers, such as ENVY/Developer, allow one to compare different versions of the code associated with methods and classes (perhaps even focussing on methods originated in the building of an application), one will want to compare different versions of an interaction diagram, to isolate differences and ensure that they correspond with design intent. One may also wish to evaluate (characterize) a diagram, or, more specifically, to extract commentary from the system which characterizes the design of the application software underlying it; such information may assist the developer in identifying design flaws.

In addition to the monitoring function, the generation and comparison of interaction diagrams can also be used as a technique to assist in debugging, since it tends to be easier and faster to isolate problems visually than in the textual debugger provided by Smalltalk (see Fig. 1.2), where one must consider only one message at a time, without any system-supported way of comparing message patterns encountered in previous executions of the given use case.

1.6 The NSERC Project

The Centre for Object-Oriented Programming (COOP) in the School of Computer Science at Carleton University is working on a major project which is being funded by the Natural Sciences and Engineering Research Council (NSERC). Its aim is to develop a methodology and tools (environment) suitable to support re-analysis, re-design, re-implementation and reuse in object-oriented systems. Naturally, this entails managing concurrent work in analysis, design and implementation, given the discussion in section 1.5. More specifically, the focus is currently on design capturing (ie. the ability to capture both the static and dynamic characteristics of working software in both textual and pictorial form) and design regeneration (ie. the ability to capture both the static and dynamic characteristics of working software in both textual and pictorial form after changes have been made) [LPWC 93]. This thesis, by dealing with design regeneration and comparison of interaction diagrams, clarifies some of the issues surrounding the latter focus, as well as the project's overall aim.

1.7 Thesis Outline

The primary contribution of this thesis is to provide a mechanism for the evaluation, regeneration and comparison of interaction diagrams. In addition, it encompasses extensions to Wang's work which permit the generation of diagrams depicting the execution of use cases associated with applications that have GUIs, and introduces the ability to differentiate among objects of the same class. In fact, some of the modifications originally made for the purpose of regeneration also facilitated the addition of GUI-handling capability.

The remainder of this thesis is organized as follows. Chapter 2 consists of a summary of relevant background material, including a closer look at Wang's work. Chapter 3 explores diagram regeneration, comparison and evaluation, while complexities posed by applications having graphical user interfaces, together with some solutions, will be considered in chapter 4. Chapter 5 concerns the method of differentiating objects of the same class in a diagram, as well as the isolation of arguments and returned objects for each method. Chapter 6 details some smaller enhancements, while two sample applications are used in chapter 7 to exhibit the utility of the tool and to clarify the procedures involved in its use. Conclusions and suggestions for future work are presented in chapter 8. It is hoped that the tool discussed here will serve as a prototype for future tools which will use graphical means to assist software developers in evaluating the design and functionality of their object-oriented applications.

Chapter 2

Background

In this chapter, Object-Oriented Software Engineering (OOSE) [Jaco 92] will be briefly discussed, as it, unlike other object-oriented analysis and design methodologies, has been used for many years in building real industrial applications [Jaco 92, foreword]. More specifically, it is the one that uses the concepts of "use case" and "interaction diagram" on which [Wang 94] and this work are based. Following that discussion, Wang's work [Wang 94] will be looked at in more detail. In the course of this review, terminology will be defined for use later in this thesis.

2.1 Object-Oriented Software Engineering

OOSE is a use case driven approach, in which system development is viewed as the gradual refinement of a sequence of models. The first model, known as the **requirements model**, attempts to capture the functional requirements of the system. It is here that the use cases of the system are defined. These use cases drive the creation of the rest of the models, which are:

- the **analysis model**, which aims at giving the system a robust and changeable object structure,

- the **design model**, which aims at adopting and refining the object structure to the current implementation environment,
- the **implementation model**, which aims at implementing the system, and
- the **test model**, which aims at verifying the system.

The requirements model consists of a use case model, interface descriptions and a problem domain (object) model. The problem domain model provides an easy-to-understand description of the system and its functionality, while the interface descriptions are detailed explanations of how it interfaces with the outside world. The most important part of the requirements model, though, is the **use case model**, which is composed of actors and use cases. An **actor** represents something that interacts with the system. Actors are differentiated from users, in that users are actual people, whereas actors play functional roles. For example, a single user may at different times act as a pilot or a ticket agent for a small airline. These roles are represented by two separate actors, even though both roles may be played by a single person. An actor has the capability of initiating one or more use cases. Fig. 2.1 depicts a use case model.

FIGURE 2.1: A Use Case Model. The system is represented as a box containing use cases. Each actor is situated outside of the system and can initiate a subset of the system's use cases (those to which the actor is connected by arrows).

The use case model defines the functionality of the system. This functionality is then structured to produce a robust, logical, implementation environment-independent analysis model. Further refinement occurs in the design model, together with adjustments for the implementation environment. The design model shows how the use cases are to be implemented using design objects (ie. it shows how the objects are to interact to realize the use cases). Source code is then used to implement the use cases. Finally, the use cases give us a tool with which to verify the system, since they define the functionality that needs to be verified. These models are all developed concurrently and incrementally, with the use cases driving the process.

More specifically, in constructing the **analysis model**, the objects involved are divided into three different types: **entity objects**, **interface objects** and **control objects**. Entity objects are held for a long time, and typically survive a use case execution. Examples are Person objects and Building objects. Interface objects model behaviour that is dependent on the interface of the system. Functionality that belongs in an interface object includes the user interface functionality associated with fetching data from the user. Control objects manage computation. They typically interact with entity objects, do calculations and then send the results to interface objects. For example, a control object may be used to calculate taxes based on numerous factors. The concepts of control and control objects will be discussed further in section 3.3. The purpose of using these three types of objects is to create a structure which is stable and resilient to change. The aim of such a structure is to localize change. That is, interface changes should be localized in the interface objects, a computational change should be localized in the control object responsible for the computation, and changes pertaining to the behaviour of an entity should be localized in its related entity object. Subsystems and the relationships among

objects are also defined in the analysis model. Notice that all these objects are built to satisfy the requirements defined by the use cases.

The **design model** is a refinement of the analysis model, where the implementation environment is considered, and classes are specified. Each use case is described as a sequence of stimuli among the objects, and depicted in an **interaction diagram**. The model also describes the programming language and database management system (if any) to be employed in the implementation.

The **implementation model** simply consists of the annotated (ie. commented) source code, which is based on the design model. It is very helpful here to be able to use **components**, which are fully implemented parts which allow the use of higher-level abstractions than the programming language provides. For example, a data structure, such as a list or a queue, might be implemented as a component, as might a set of statistical calculation tools or a group of special panes that a company might reuse frequently in an attempt, perhaps, to give its applications user interfaces which are in some way consistent with one another. The key characteristic of a component should be its reusability across many applications.

Finally, the **test model** describes the result of testing. First, lower level objects and object modules are tested. Then, integration testing is done at various levels, as the subsystems are put together, until at last the entire system is tested. The use cases are the main tool in completing the integration testing, as they define the functionality which is to exist and hence that which is to be tested.

From this brief summary of OOSE, it is clear that use cases are central to the methodology, and that interaction diagrams are used to depict the interactions among

objects that should occur during the running of a use case (the diagrams form part of the design model). It seems reasonable then to employ use cases and their interaction diagrams as mechanisms for monitoring implementation, finding obvious errors, etc., as outlined in Sections 1.4 and 1.5. The information thus extracted can be easily fed back into the next iteration of the design model. Having seen the utility of generated interaction diagrams, consider now the original Interaction Diagram Generator (IDG).

2.2 The Original Interaction Diagram Generator (IDG)

In this section, two aspects of IDG will be considered. First, IDG will be explained and justified strictly with regard to its functionality. Then, selected implementation details relevant to the current work will be presented.

2.2.1 Functionality and User Interface

2.2.1.1 Functionalities and Their Justifications

In standard Smalltalk V/Windows, there is no explicit concept of "application." That is, there are no application identification or management facilities built in. Some enhanced Smalltalk development environments, such as ENVY/Developer, include such facilities, and it is expected that as part of the NSERC project (see Section 1.6), an environment which supports application management as well as graphical views will be produced. However, in standard Smalltalk, an application is simply made up of a group of classes. The grouping is a conceptual one, in the mind of the developer.

Since use cases are associated with applications, and use cases are to be depicted in interaction diagrams (according to OOSE), the concept of "application" was needed in the building of IDG. In fact, an application is the context over which an IDG is opened.

Unfortunately, it is impossible to specify new applications from within an IDG. One can only operate within the application currently in the context of the IDG, or load a previously saved application. New applications must be specified directly within the **Application** class, which has been added to the Smalltalk hierarchy. This point will be addressed again later, when the new approach to application management, defined as part of the current work, is explained.

Use cases need to be specified such that it is clear what application they belong to and what functionality they encompass. Given that IDGs operate within the Smalltalk environment, it seemed that the simplest way to specify a use case was to specify a piece of Smalltalk code which embodied its functionality, since any other approach would have required automated translation into Smalltalk in order to run the use case. Again though, new use cases for use by an IDG must be defined together with their applications, within the **Application** class; the "new approach to application management" will improve on this.

Another concept which is essential to the usability of IDG is that of **abstraction**. Often, when a use case is run, it produces many messages, but a large proportion of them are not likely to be relevant to the application developer. For example, any use case that employs collections will produce messages that surround the implementation details of adding and deleting from those collections. Similarly, any use case that employs mathematical calculations will produce messages relating to the implementation details of numbers and calculations. These messages are not typically of interest to the application developer. They can cause a diagram to become large, unwieldy and difficult to understand. Therefore, the developer must be able to abstract out uninteresting messages, a process commonly called **filtering** or **applying filters**. Filters are rules

which limit the amount of information (messages) displayed. Skillful use of them in IDG can make important information much easier to comprehend.

IDG supports several ways of filtering. Firstly, when establishing an application, one specifies, among other things, the classes whose interactions are typically of interest when considering a use case of that application. Secondly, these classes can be modified from within an IDG, and the effects will hold for all subsequent diagrams generated in that IDG. This may be advisable if, for example, a particular developer on a team is not concerned with a particular class in an application and does not wish to lose focus due to information which concerns only other team members. As well, a developer may want to add to the default classes if, for example, the information thereby obtained is expected to help in isolating a programming deficiency. A developer may also filter out particular methods within an IDG, to focus attention on the rest, where appropriate. All of these filtering possibilities take place before diagram generation. One may also wish to induce a filtered view after a diagram has been generated. This is useful, for example, in concentrating on different areas of a large diagram, each in their turn. Accordingly, one can filter out messages after diagram generation and later bring them back.

In the course of development, one may need to compare diagrams to track progress or identify a programming deficiency. Although, as stated earlier, there is no direct automated support for this in IDG, it at least allows one to accomplish this manually. This is done by enabling one to load and save diagrams. A saved diagram can thus be recalled. To get a view of two diagrams at once (typically an old one and a current one), one can run two IDGs, one displaying each. As this could get rather cumbersome, an ability to print diagrams becomes desirable. Not only can one then look at the hard copies together relatively easily, but it becomes easier to share information

with members of a development team (by annotating diagrams, and making copies) for consultation purposes. Accordingly, such a facility exists in IDG.

Another facility of use in speeding the development process would be one which gives the developer quick access to the code found in a particular method, perhaps one that is, for example, generating unexpected messages. It would be convenient not to have to continually switch back and forth between the interaction diagram generator and Smalltalk browsers in order to establish connections in one's mind between method names and the underlying source code. Conveniently, Smalltalk provides a method browser which makes this fairly easy to implement with a straightforward user interface.

Especially in the case of large applications, that is, those with a large set of relevant classes, it can be difficult to identify subsets of classes that work together to accomplish parts of use cases. If one wants to focus on the carrying out of the responsibilities of a particular class, for example, one should focus on messages sent by that class and its **collaborators** (sometimes called **helpers**). Collaborators are classes that help a given class carry out its responsibilities. They can be easily identified as classes whose instances receive messages from instances of the given class. Another aspect of understanding a given class' responsibilities is identifying its **users**, that is, the classes whose instances send messages to instances of the given class. If one puts together lists of the collaborators and users of each relevant class (considering the union of all use cases pertaining to the application), this information is termed the **collaborations** of the application. IDGs provide this information.

As well, a developer may wish to have messages displayed one at a time during diagram generation, and to be able to control the timing of each message display. This is particularly useful when one wants to track the order of message sends, but the diagram

does not completely fit in the visible area of the diagram display pane, since the pane can be scrolled between message displays to view the current area of activity. This way of handling generation and displays is called "stepping", whereas a "go" command signals an IDG to complete the current diagram immediately.

Finally, sometimes interaction diagrams are drawn with "return arrows" from the receiver of a message back to its sender, to signify the completion of the associated method. This clarifies the scope of a message send (ie. the segment of a diagram that details further messages precipitated by the one in question) and allows the developer to recognize when objects are returned. Although the colour scheme used in diagram display is intended to facilitate scope identification, it was decided to add this mechanism which is more common than colouring in hand drawings, and may be reassuring to the developer concerned about verifying that objects are returned. The current work introduces the identification and inspection of returned objects (see Section 5.3).

2.2.1.2 User Interface

To invoke a new IDG, one issues the command:

IDGView new openOn: <anApplication>

Executing this results in the opening of an IDG main window (also called an IDGView or a UseCaseView) like that shown in Fig. 2.2 (pane identifiers have been added for easy reference). It was designed to conveniently present the functionalities outlined and justified in Section 2.2.1.1. The design bears some similarity to that of a Smalltalk class hierarchy browser in that the panes at the top of the window and the

FIGURE 2.2: Main Interaction Diagram Generator Window (IDGView)

menus allow the developer to specify what information (diagram) is to be displayed (generated) in the lower area (with the exception of post-generation filtering which is done using mouse operations in the diagram pane).

The "File" menu contains the options "Save Application" and "Open Application" ("Open" is a synonym for "Load" in this context). This parallels the "Save" and "Open..." options found in the file menus of many word processing software packages, except that the file dialog that is brought up for file identification (applications are stored in files) suggests the default file extension ".app." The "View" menu contains the options "Return Arrow On" and "Return Arrow Off" to control the viewing of arrows indicating the

return of objects. In Fig. 2.2, the option is off, which is the default value since most of the time the arrows are not needed. Perhaps unfortunate is the fact that return arrows can only be triggered and viewed after a diagram has been completed, not during generation. The return arrow option can also be set from the popup menu in the diagram pane. The introduction of the popup menu mirrors the use of popup menus in Smalltalk V/Windows browsers, for the convenience of the user.

Aside from the initial specification of an application, the pre-generation filtering of diagram information is handled in the class and method panes. The class pane lists the classes associated with the application in question. Methods appearing in the method pane are either the class or instance methods of the selected class, depending on the setting of the class/instance radio buttons. Double-clicking the left mouse button with the mouse pointer situated on top of a class in the class pane toggles the class on or off, meaning that messages sent by that class will or will not be included in subsequently generated diagrams, respectively. Similarly, double-clicking the left mouse button with the mouse pointer situated on top of a method in the method pane toggles the method on or off, meaning that the particular message will or will not be included in subsequently generated diagrams, respectively. Thus, toggling a class off has the same effect as toggling off all its methods. Finally, if one needs to explore classes other than those usually of interest when dealing with the current application, classes can be added and/or removed (within the scope of a single IDG) by triggering suitably named options in the class pane's popup menu.

After setting the filters and choosing a use case from the list in the use case pane, one may press either the "Step" button or the "Go" button to begin diagram generation. As explained in Section 2.2.1.1, "Step" mode means moving forward one message at a

time, waiting for the next command after completing that task, whereas "Go" mode causes the diagram to be completed immediately.

The "Save" and "Load" buttons initiate the saving and loading of interaction diagrams (not applications!). Interaction diagrams are stored in files. Therefore, file dialogs are used for diagram file identification. The default extension for interaction diagram file names is ".ind."

Up to this point in this section, discussions have considered the user interfaces for those functionalities that form part of the process of generating, loading or saving an interaction diagram. Once a diagram is in the diagram pane though, one can, as prescribed in the previous section, do more than just examine it in the pane. Double-clicking on a method name in the pane brings up a method browser (see Fig. 2.3) on the

FIGURE 2.3: A MethodBrowser on a method from the interaction diagram in the diagram pane of Fig. 2.2, namely the accountNumber: instance method of class BankAccount.

method. This is usually done during debugging. Double-clicking on a message arrow collapses the message and those generated by it to a small black square. Double-clicking on the square later makes the messages reappear. This process is referred to in the previous section as post-generation filtering. Also, any diagram can be printed by pressing the "Print" button, provided, of course, that a working printer is on-line

The final part of the interface is a straightforward way to present the **collaborations** involved in an application. Collaborations are not based on only one use case. Rather, in theory, all use cases should be considered. However, in order to use all use cases, they would all have to be run, a potentially time-prohibitive process. As a compromise, an IDG considers the use cases whose diagrams have been generated and saved using it. A record is kept of the names of the files in which the diagrams are saved, and the data from those files is used to compute the collaborations. If a use case has been run and the diagram saved more than once using the IDG, the newest version is used.

Then there is the issue of presentation. There is not enough room to do this in an IDG main window. Therefore, pressing the "Collaborations" button brings up another window, called a **CollaborationsView**. An example of such a view is shown in Fig. 2.4. The top two panes in a **CollaborationsView** list the use cases and their corresponding diagram files, so that the developer is reminded of which files are being used, which is justifiable given that if there are many use cases, one could forget some of the file names by the time all diagrams have been generated. If no generation has been done for a use case, the use case still appears, but the file name is left blank, alerting the developer. This may be done intentionally, in order to limit the scope of the collaborations data to a subset of the existing use cases, or it may constitute an error, but it is clearly something

FIGURE 2.4: A CollaborationsView Focussed on the BankAccount Class

the developer should be told about. When a class is selected in the classes pane, the users and helpers of that class are displayed in their respective panes, based on the information in the files listed in the interaction diagram file pane.

This interface seems to be a good one. The current work builds on it to accommodate the new functionality.

2.2.2 Selected Implementation Details

2.2.2.1 Using System-Maintained Information Through Inheritance: IDGDebugger and IDGProcess

One of the main pillars of object-oriented programming is inheritance, that is, the ability to create a subclass of a (parent) class which automatically responds to the messages and recognizes the class and instance variables of the parent. In order to implement IDG, information about messages sent and their receiver objects had to be

obtained. Such information would have been exceedingly difficult to procure, were it not for the fact that Smalltalk automatically maintains this information in a usable form when a process is running under the Smalltalk debugger. This is not surprising because it is the same kind of information needed when using the standard debugger. The standard debugger is implemented in the class Debugger. Specifically, the Debugger shows every message send and return as a string of the form 'class name >> method name.' Thus, one possible sequence could be:

```
class1 >> method1    "message send invoking method1"  
class2 >> method2    "message send in method1 invoking method2"  
class3 >> method3    "message send in method2 invoking method3"  
class2 >> method2    "back to method2; usually means method3 completed"  
class1 >> method1    "back to method1; usually means method2 completed"
```

The term "usually" is used because there is a valid exception involving recursion, namely that, for example, method2 could invoke method3 which could subsequently invoke method2 with different parameters. Fortunately, problems caused by recursion can be addressed and form part of the current work. They are discussed in chapter 5. In any case, it made sense, to use the existing Debugger class, but a few modifications had to be made for the purpose of this application. Logically, this led to the use of inheritance. A subclass of Debugger, called IDGDebugger was created. The information maintained by standard debuggers could thus be obtained, with behaviour and state variables added to or overridden as appropriate.

There are three major differences between Debuggers and IDGDebuggers. One is that IDGDebuggers do not present walkback windows like the one seen in Fig. 1.2. This is because IDGDebuggers are meant for drawing interaction diagrams without halting

execution, not for halting execution to support text-based debugging. Another is that IDGDebuggers know how to cooperate with InteractionDiagrams by providing them with the necessary information to construct themselves and telling them to update their displays (in their IDGView diagram panes) when that is necessary. (Note that InteractionDiagrams keep track of their respective diagram panes and know how to display themselves in those panes. The form of the display is controlled within the InteractionDiagram class). The third difference is that every IDGDebugger maintains a "sender stack." Essentially, when a message is sent, it is placed on the stack; when it returns, it is recognized as a returning message by matching it with it's corresponding message send in the stack. For a message send, a new "Operation" (the IDG parallel of "message") is created; when a return occurs, this is recorded within the corresponding operation by noting the diagram "time count," information which facilitates the display of return arrows when they are requested.

As part of enabling the use of IDGDebugger, inheritance had to be used once more. Normally, when a Debugger is initiated for a Process, it is done by running the following:

Process

queueWalkback: 'Window Label String'

makeUserIF: CurrentProcess isUserIF

resumable: <boolean - Can the process be resumed after this?>.

This method starts up a standard text-based Debugger. Since an IDGDebugger is needed, a subclass of Process, called IDGProcess, was created. Using this class, an IDGDebugger is started as follows:

IDGProcess

queueWalkback: 'IDGDebugger'

makeUserIF: CurrentProcess isUserIF

resumable: true useCaseView: <aUseCaseView>

The class method `#queueWalkback:makeUserIF:resumable:useCaseView:` is the only behaviour in the class `IDGProcess`. All state information and all other behaviour is inherited. The method starts up an `IDGDebugger`. Notice the additional `useCaseView:` argument; it is there because `IDGDebuggers` take filtering information, among other things, from the view. The label '`IDGDebugger`' does not actually appear anywhere during execution, but is given as an argument for implementational reasons.

2.2.2.2 Application and UseCase

The classes `Application` and `UseCase` represent their namesake concepts from OOSE. Therefore, they are easy to understand conceptually. An instance of the class `Application` consists of a name, a default collection of 'interesting' classes for developers working on the application, and a collection of use cases. An instance of the class `UseCase` primarily consists of a name and a string containing Smalltalk code which is to be run when the use case is executed. In addition, it holds a string denoting the path name of the file to which an `InteractionDiagram` relating to the use case was last saved; this is used for the purpose of identifying the files to be used in determining the collaborations present in an `Application` (Section 2.2.1.2 includes a discussion about collaborations). Notice that both the `Application` name and the related `UseCase` names appear in `IDGViews` opened on a given `Application` (Fig. 2.2 shows an `IDGView`).

2.2.2.3 InteractionDiagram, Column and Operation

The classes `InteractionDiagram` and `Column` also represent their namesake concepts from OOSE (the vertical lines in an `InteractionDiagram` that represent objects are often called columns). The class `Operation` represents the concept more frequently referred to as "message" (Smalltalk already has a `Message` class which is used for

something different. "Operation" is another term for "message" in some contexts including in relation to interaction diagrams). InteractionDiagrams keep track of their columns, which in turn each keep a record of the operations that denote messages sent to the object which it represents. Aside from being a realistic representation, grouping operations by their columns is helpful in identifying the users of objects and classes (recall that users are one aspect of the specification of collaborations), since operations keep track of their senders, which are in fact users of the class of the object that the column containing the given operation represents. There are times, as shall be seen in Chapter 3, when for the purposes of the current work, the operations appearing in an InteractionDiagram must be considered in chronological order regardless of their associated columns, but this is fairly easily handled.

InteractionDiagrams keep track of their operations. Each diagram also knows about the pane which is to be the destination of its display operations (a display pane of an IDGView; the pane is an instance of IDGGraphPane, a specialized subclass of GraphPane), and how to display itself in that pane. The form of the display is specified in class InteractionDiagram.

Each operation mainly keeps track of its related method's name, its sender and receiver classes, and its starting and ending time (measured in internal "time" units). It also keeps track of some information, like the rectangle in which it is to be displayed in a pane when the InteractionDiagram of which it is a part is displayed, which, although not conceptually relevant to an Operation, is needed for the purpose of display.

2.2.2.4 Implementation Summary

The implementation was done with OOSE in mind. It clearly distinguishes the concepts of "application," "use case," "interaction diagram" and "operation," thus making this part of it easy to understand and relate to the ideas of OOSE.

The most difficult part of the implementation to understand is the extraction of the information needed to build an interaction diagram from the Smalltalk system while the code to which the diagram pertains is running. The main ideas here are that the standard text-based Smalltalk debugger maintains the necessary information, and that by subclassing the Debugger class, the information can be obtained. A few changes are then made in the subclass (IDGDebugger) so that no walkback window appears and so that instances of this class know how to communicate the information to an InteractionDiagram.

At a high level of abstraction, the IDG operates as follows. A user provides input to an IDGView. Once the filters are set and a diagram generation demanded for a given UseCase, an IDGDebugger is set up to run the code by which the use case is described. While running the code, the IDGDebugger provides information about the messages involved to an InteractionDiagram, which uses this information to build itself up. As it is building, the InteractionDiagram continually updates its display on the IDGView's display pane, until the diagram and the display are completed.

2.3 Summary

OOSE is a use case driven approach to software engineering. It is based on Jacobson's years of real-world development experience, and aims at developing software

that is resilient to change. It introduces five models, namely the requirements model, the design model, the analysis model, the implementation model and the test model. Use cases are developed as part of the requirements model, and then influence all other models. Although these model names may be reminiscent of structured (procedural) design's "Software Development Life Cycle" (SDLC), the process is quite different, in that, rather than trying to complete one model before moving to the next as in SDLC, OOSE proposes the concurrent incremental development of all the models.

In OOSE, use cases are visualized as interaction diagrams. These diagrams are helpful in understanding the inner workings of an application, because they present overall pictures of the interactions among objects, instead of the one message at a time that one sees with the standard Smalltalk V/Windows debugger. Producing the diagrams on-line based on actual messages sent, using IDG, eliminates the problem of human error from the drawing process. It also eliminates a good deal of tedious manual work usually needed to complete the drawings. Also, on-line diagrams can often be helpful to the process of debugging an application.

The interface of IDG was designed to be easy to use, with some limited degree of similarity to that of Smalltalk's ClassHierarchyBrowser. The implementation makes use of the Smalltalk Debugger class by subclassing it, so that message-related information maintained in the standard debugger can be extracted while still permitting behaviour specific to the generation of interaction diagrams.

Chapter 3

Regeneration

3.1 What is Regeneration?

During the course of both the initial development and the maintenance of an application, changes are continually being made. In the context of object-oriented applications, this implies class hierarchy changes, new classes, new methods, and the removal of old classes and methods. In short, the interactions and relationships among objects are altered.

It would be useful to have some automated means of identifying changes in the way in which given use cases are accomplished, for the purpose of monitoring progress and/or debugging application software. Looking for changes manually is time-consuming and error-prone; it is especially time-consuming when checking volumes of source code, but is also somewhat so when looking at larger diagrammatic representations. OOSE presents a diagrammatic representation that shows an overall view of how a use case is carried out (that is, a view of the interactions among objects that occur in attempting to fulfil the functional requirement to which the use case pertains), namely the interaction diagram. Therefore, in order to retain the overall perspective of interaction diagrams while supporting automated change identification, it seems natural to consider the

comparison of interaction diagrams. This means reproducing - ie. 'regenerating' - an interaction diagram, running the same use case as the 'original' version with the same filters, insofar as it still makes sense to do so. (Admittedly, the code to be run to initiate a use case may change, and filters may, over time, become out-of date, referring to classes or methods which no longer exist, for example. Such exceptions will be dealt with in section 3.4.) Once discrepancies in interactions have been identified (by comparison), they must be presented in an easy-to-understand visual manner. Then, developers can isolate unwanted interactions for debugging purposes, and accept desired ones as either partial implementation of or modifications to design intent.

As well, though this will not be expanded upon here, one would likely want to expand the regeneration idea to include automated update of external entities (eg. documentation) based on new interaction information. Currently, the technologies required to establish the necessary communication links between Smalltalk applications and such external entities are unclear, and are being investigated by others, including people working under the auspices of the Centre for Object-Oriented Programming's NSERC project, which was discussed in Section 1.6.

3.2 The Basic Regeneration Functionality and Interface

The question remains: How are such interaction diagram comparisons initiated and how are the results presented? These issues will now be looked at, each in their turn.

3.2.1 Initiating a Regeneration

Clearly, the original IDGView exhibited in Fig. 2.2 is insufficient to support regeneration, because it does not include any reference to regeneration, and more

specifically, it provides no mechanism which could be logically used to trigger that functionality. As the original IDGView used buttons as the mechanism through which to initiate major operations (eg. 'Step', 'Go', 'Load'...), it seemed sensible to continue to adhere to this straightforward idea by having regeneration triggered in the same way, given that buttons are easy to use. Hence, one of the noticeable differences between the old IDGView and the new one depicted in Fig. 3.1 is the addition of a 'regenerate' button through which the user can access the regeneration functionality. There have also been some changes to the menus; these will be clarified later.

FIGURE 3.1: New IDGView.

Now, consider the process of identifying the diagram which is to be regenerated. There are two possible sources for the diagram. The first is the current generator settings. This may appear illogical at first; after all, why would one want to regenerate a diagram that has just been generated within the same IDG session? The answer is that one might put an IDG session aside (perhaps by minimizing an IDGView), make some changes to application code, and then wish to compare the diagram previously generated with that obtained by rerunning the use case after the changes. In cases like this, the simplest approach is to go back to the IDGView and regenerate the diagram from the current settings.

The second possible source is a previously saved diagram. This is the only option that permits comparisons of diagrams generated in different IDG sessions. If that choice is made, the name of the file in which the diagram to be regenerated resides is required. It was decided that the developer using IDG should not have to worry too much about where exactly the diagrams are stored, and also that there should be some cue, other than a filename that is too short to support a satisfactory degree of descriptiveness, whereby a developer can identify a particular diagram before selecting it for regeneration. Hence, the "Interaction Diagram File Dialog" has been introduced. An example of such a dialog is given in Fig. 3.2. The idea is that to select a diagram one need only select the desired application and use case, and then choose from a list of files that pertain to that use case. Also, when a file is selected, a summary of its contents (application name, use case name, diagram title, date and time of generation and a list of the classes involved and the filtered items) is provided in the text pane below the lists; developers use this information to identify the diagram in the file. Pressing the 'Regenerate' button or double-clicking on the file name finalizes the choice.

FIGURE 3.2: An Interaction Diagram File Dialog.

In fact, the idea of filtering file lists and providing the information necessary for the developer to identify a given diagram can also be applied to the loading and saving of diagrams. Thus, interaction diagram file dialogs are now used for diagram specification in carrying out those functionalities as well. In the case of saving, the application and use case involved is already known when the operation is triggered - the use case is the one just run before triggering the operation. Therefore, there is no point in providing lists of applications and use cases; such lists would just be a useless distraction to the developer. Accordingly, in this case the dialog comes up with only the relevant (pre-selected) item appearing in each of these two lists. Also, the button labelled

'Regenerate' in Fig. 3.2 takes on the labels 'Load' and 'Save' where appropriate, so as to avoid confusion.

3.2.2 Comparison Results and their Diagrammatical Presentation

Primarily, the differences of concern when comparing interaction diagrams are the addition and deletion of messages. Diagram characteristics such as those discussed in section 3.3 can also be compared, but the data on these are typically short and digestible for the user. Therefore, although some identification of characteristics is provided, the focus is on the message-level comparison.

When a developer is satisfied that progress has been made, regeneration can be used simply to update old diagrams. In this case, a visual diagrammatic comparison is not needed. For this reason, once a source diagram file is specified, the developer is asked whether the destination is the 'Screen' (meaning that diagrammatic comparison is needed) or a file (meaning that IDG is to run the use case indicated in the source diagram file using the filters specified there, and save the result to a destination file subsequently identified by the developer/user). There is also a 'both' option, which renders a diagrammatic comparison in addition to saving the new diagram to a file. This can be used if, for example, a developer knows that there has been sufficient progress to warrant saving, but wants to view the message changes again as a prelude to continuing with the work because, perhaps, a problem has been noted but not clearly remembered.

Once the necessary information has been provided, the use case is run to get the current result, which is displayed in the diagram pane. Then, if visual comparison is required, a delta view (see Fig. 3.3) is opened. 'Delta' is a term meaning 'difference'

FIGURE 3.3: An Interaction Diagram Delta View.

FIGURE 3.4: A Delta View focussed on the First Difference (ServiceChargeTransaction>>new).

which is often used in, for example, mathematics and other sciences. Thus, the name `InteractionDiagramDeltaView` ('delta view' for short) was given to the view exhibited in Fig. 3.3 since it is built for the purpose of clearly showing the differences between a source (or, 'old') diagram and a current (or, 'new') version. Initially, the view is 'focussed' on the first difference (see Fig. 3.4). From there, users can choose 'Go' (complete display) or 'Next Difference' (display operations one by one in both the old and new diagrams in parallel until a difference is found, then focus on that difference); these options are analogous to the 'Go' and 'Step' functionalities of the `IDGView`, respectively. Differences are shown in red for emphasis, and the focussed difference is displayed with a thicker arrow than usual for identification.

3.2.3 The Textual Log

It also seems advisable to have a mechanism for keeping a textual log of changes. Accordingly, a log is created. Depending on the setting of the 'Log To' option of the options menu in the main window, it can appear on screen, be saved as text in a file, or both. If information is being logged to a file, the file, which can be appended to in successive IDG sessions, becomes a long-term log, so that changes are tracked without having to repeatedly search out old diagrams and rerun comparisons. If it is only being logged to a window, the long-term recording functionality is lost. In log windows, though, individual differences are displayed in a list box (near the top) as this makes it easy for one to select a change, which is then focussed on by the delta view automatically. This makes a pictorial review of changes easy to do. There are four kinds of differences: an added operation, a deleted operation, changed arguments to the method invoked in an operation, and the returning of a different class of object being returned by the method invoked in an operation. Arguments and returned objects (new considerations

for the Interaction Diagram Generator) will be discussed in detail in chapter 5. The text box at the bottom of the window contains quantitative and qualitative interpretations of the comparison. The next section gives a detailed description of the interpretive information. Fig. 3.5 provides an example of such a log view (also called a "differences text view").

FIGURE 3.5: A Text (log) View of Interaction Diagram Differences.

3.3 Interpretations and Comparisons of Interaction Diagrams

There are a few ways in which individual interaction diagrams can be characterized. For example, in [Jaco 92, pp. 220-221] Jacobson defines two broad types of interaction diagrams. Fork diagrams exhibit centralized control. Here, "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." Fig. 3.6 depicts a fork diagram. Stair diagrams, on the other hand, exhibit decentralized control. "This type is characterized by delegated responsibility. The included objects only know a few of the other objects and know which objects can help with a specific behaviour. Here, we have no 'central' objects." Fig. 3.7 depicts a stair diagram.

Why would anyone be interested in such a characterization? What kind of diagram is best? As Jacobson states: "Often, it is claimed that the stair structure is more object-oriented and thus better." This idea is derived from the understanding that in object-oriented programming, each object should have its own narrow set of responsibilities, such that in order to carry them out, it only needs to communicate with a small set of other objects. Under this principle, processing is widely distributed among objects which work together to achieve the final goal. However, Jacobson notes that the real concern is not the degree of distribution of processing, but rather, ease of change as well as object reusability. He states that different structures support different kinds of change. Where the way in which some parts of the execution sequence is performed is the focus of change, it makes sense to split the processing so that when there is a change in a certain part of the process, it is likely to be localized in one object, or a very small number of objects. On the other hand, when the order of execution of the various parts of a process is likely to be the focus of change, or new operations could be added, the operation sequence should be localized in one object, a control object. Therefore, this characterization must be evaluated in the context of the kinds of changes that are expected, but within that limitation it provides interesting information. Usually, a mix of the two control mechanisms is the best approach. Therefore, one may see things like "local control objects" for segments of a calculation. Hence, when identifying control

objects, one must keep in mind that there may be none (ie. stair diagram extreme), one (ie. fork diagram extreme), or a few (which corresponds roughly to a mix of control mechanisms).

FIGURE 3.6: A "Fork" Type Interaction Diagram with SystemBorder as the Controller.

FIGURE 3.7: A "Stair" Type Interaction Diagram.

The bottom half of the differences text view gives individual evaluations of both the 'old' and 'new' diagrams, as well as relevant comparisons between the two. The individual evaluations consist of a statement of the percentage of messages sent by each class (or instances thereof), as well as the identification of any evident control classes.

The term "control class" is not to be confused with Jacobson's concept of "control object." The two terms are loosely related, in that control objects are often instances of control classes, but this is not necessarily the case. For the purposes of this work, the term "control class" aims at identifying those classes which, together with their instances, are responsible for sending a high proportion of the "interesting" messages involved in executing a given use case. Control classes might, perhaps more appropriately, be referred to as "central classes." They are identified relative to use cases rather than entire applications. Also, in recognition of the existence of structures that yield neither a purely stair-type diagram nor a purely fork-type diagram, and of the usefulness of such structures in many situations, the definition of "control class" used here was designed so as to detect classes responsible for a high proportion, though not necessarily all, of the interesting messages. Such detection was thought to be more generally useful than simply looking for the case of a pure fork-type structure. Specifically, the definition chosen was:

"A control class is one which either

- [1] is responsible for at least 50% of the message sends found in a use case, or**
- [2] is responsible for at least twice the average number of message sends for classes in the use case (or, more formally,**

$$\text{numberOfMessagesSent}(\text{control class}) \geq 2.0 * (\text{totalMessageCount}(\text{use case}) / \text{totalClassCount}(\text{use case}))"$$

These individual evaluations, which can be obtained without doing a comparison by triggering the 'Evaluate Diagram' item from the options menu of the main window of the interaction diagram generator after running a use case, in themselves provide data which can be used for comparison. In addition, the bottom half of the differences text

view also identifies classes which are new to the diagram, and formerly used classes which have been removed.

3.4 Obsolete Filters and Changes in Class Structure

3.4.1 Definitions and Tools for Adjustment

An **obsolete filter**, found in an out-of-date interaction diagram file, is one which either (1) refers to a class or method that no longer exists in the Smalltalk image, or (2) refers to a class which still exists, but which is no longer pertinent to the application in question. The most important type of obsolete filter to address is the nonexistent class. The reason for this is that unlike Wang's generator [Wang 94], for the purpose of implementing important enhancements (see chapter 5), the current one stores application-related objects, rather than strings representing them which simply denote their classes. Consider that each time a user tries to load an instance of a class which does not exist, the **ObjectFiler** (the object in Smalltalk which is responsible for transferring objects to and from files), as part of its standard loading sequence, asks questions in order to resolve the situation; the questions are repeated for each instance of the class, which is very irritating for the user! The same thing occurs when the structure of instances of a class (that is, their instance variables, both local ones and those which are inherited) changes. Upon trying to load an instance with an obsolete structure, the **ObjectFiler** asks the user to verify its default mapping from old instance to new; again, this is repeated as many times as there are instances of the class in the file! Finding a more convenient way of dealing with these problems is doubly important, because aside from the 'load' functionality itself, a file is usually read and loaded as part of diagram regeneration.

In order to address this and other issues, the table of filters (commonly referred to as the item table) and a dictionary which associates 'interesting' classes with their instance variables, are now stored near the beginning of each interaction diagram file. They can be read not only to identify the diagram in the file, but also to get the names of classes, instance variables and methods of interest to the application concerned. With this information, any obsolete items (including nonexistent and structurally out-of-date classes) can be isolated. As well, once information is obtained from the user such that a map is defined from obsolete to existent items (classes, variables or methods), the **ObjectFiler** provides the underlying functionality required to automatically reuse this information as often as necessary while loading the rest of the objects in the file. The method used is **ObjectFiler class>>loadAllFromPathName: <aPathNameString> loadMaps: <aLoadMapsDictionary>**, and the "load maps dictionary" contains associations of the form:

```
key =      name of filed (potentially obsolete) class

value =    anAssociation key: name of new class
           value: aCollection of associations of the form
                 key =  new instance variable name
                 value = old (filed) instance variable name
```

Notice that aside from specifying how to load instances of obsolete classes, this also allows for the mapping of instance variables, regardless of whether or not it is necessary to account for an obsolete class. That is, one can account for changes in class structure (ie. instance variables) by setting the filed class name equal to the new class name and specifying the appropriate instance variable mappings. This is clearly quite powerful in the context of managing class and class structure changes.

Method name changes are not critical, as they need only show up as differences, highlighted in the basic comparison operation described earlier in this chapter. Nonetheless, functionality has been added which allows the user to specify method-to-method mappings.

Admittedly, there is a problem with IDG which causes it to fail when trying to regenerate a diagram where the use case involves a nonexistent class or method. This occurs because the code that runs when the use case is triggered has not been adjusted to reflect the changes in the class hierarchy. The easiest way to address this would seem to be to allow the user to change the code that defines the use case as part of the process of specifying the required mappings. This should probably be implemented as part of some future work.

3.4.2 Specifying Mappings

In order to make the tool usable, it had to have an interface which allowed the user to conveniently specify mappings and control interaction diagram file updates. The importance of controlling file updates is due for the most part to the fact that an obsolete filter detected while loading one diagram should also be found in all diagram files relating to the same application that have not been updated since the filter became obsolete. As there may be many of these diagram files, updating each of them one-by-one may very well be too time consuming for the user. There needs to be some support whereby the user has the option of updating only the current file or all diagram files relating to the application in question. As well, there needs to be an option that allows for file updates not to be undertaken at all. This may be used, for example, when a change in class structure is expected to be temporary (ie. experimental).

When obsolete filter(s) are detected in a filed diagram during a load or regeneration operation, an `OutOfDateFilterDialog` appears (see Fig. 3.8). Without loss of generality, in the rest of this section, focus will be given to the load operation; one should note that since regeneration operations based on filed diagrams use load operations behind the scenes, the method specified here applies equally well to regeneration operations. Since filters refer to an entire application and are checked before looking at any operations, obsolete filters must be corrected regardless of whether or not they are pertinent to the particular use case depicted in the filed diagram. The example in Fig. 3.8 depicts such a dialog, where the user has attempted to load a diagram pertaining to the 'Simple Bank Account' application, where the `CheckingTransaction` class has been removed and where the `BankAccount` class has had an instance variable 'x' temporarily added for the purpose of demonstration. `BankAccount` has also lost one of its former methods (`printOn:`). As an illustration of the above point, note that even if the only change had been the removal of class `CheckingTransaction`, the dialog would have appeared even if the use case associated with the filed diagram consisted of making a deposit, a function that never uses class `CheckingTransaction`.

Double-clicking on a class in the list which no longer exists, and thus has two asterisks beside its name, produces a prompter, as in Fig. 3.8, requesting the name of the class to which the old one is to be mapped. The user is then asked to verify and/or modify the instance variable mapping, using `Digitalk's ObjectLoadDialog`, a supporting view for the `ObjectFiler` of `Smalltalk V/WIN 2.0`. The default mapping simply leaves unchanged those variables which exist in instances of both classes, and sets those pertaining only to the new class to the value `nil`. Double-clicking on a class whose instance variables have changed, a characteristic denoted by the appearance of one

FIGURE 3.8: An `OutOfDateFilterDialog`. Classes whose names are followed by two asterisks no longer exist in the Smalltalk image. Those whose names are followed by one asterisk have different instance variables from those existing when the diagram was saved. Those whose names are not followed by an asterisk have had old methods removed.

asterisk beside its name, simply brings up an `ObjectLoadDialog` to allow the user to establish the mapping between the old instance variables and the new ones. Fig. 3.9 shows an `ObjectLoadDialog`. As `ObjectLoadDialogs` fulfil the requirements for establishing instance variable mappings, and are already used for that purpose by the `ObjectFile`, there was no need to create a new dialog. The point here was only to ensure that once mappings are specified, they weren't requested again while loading the same file. This has been accomplished.

Double-clicking on a class whose form has changed or which has lost some of its previously existing methods results in the problems being displayed in the right-hand list pane, with the user being asked to specify corrections in a similar manner to that shown above for the class list.

FIGURE 3.9: An ObjectLoadDialog (from Digitalk). Users modify a link in the map by clicking on the 'handle' next to the variable name, moving the mouse to redirect the connection, and clicking on the handle on the opposite side to establish a new connection.

As explained above, a mechanism for controlling file updates is required. Accordingly, upon completion of the required changes, the user presses the 'OK' button in the OutOfDateFiltersDialog, completion is verified, and the user is given three choices with respect to updating saved diagrams for the stated changes: (1) no updates (changes were made for loading purposes only; no permanent record need be kept), (2) update current diagram only, or (3) update all saved interaction diagrams related to the current application. The updating of multiple diagrams is done by silently (that is, without diagram display) loading each in turn under the established load maps, and then saving it.

3.5 Summary

The term "regeneration" refers to the reproduction of an interaction diagram by running the same use case as the "original" version with the same filters, insofar as it still makes sense to do so. The disappearance of formerly relevant classes, or changes in the structures (ie. instance variables) of the instances of relevant classes, dictate that the user specify a mapping from the old classes and instance

structures to currently existing ones, so that old diagrams may be read and understood. The process of establishing this mapping has been facilitated.

Regeneration is not only a mechanism that can be used to update saved interaction diagrams for class hierarchy changes without having to memorize filter settings, but also a required ingredient for reliable diagram comparison functionality. The reliability comes from the fact that filter settings do not have to be memorized. The user is assured that comparisons will take place only between diagrams pertaining to the same use case and having the same filters.

Automated comparison allows the user to more quickly become aware of changes in the way use cases are carried out. The user can then decide whether or not a given change was intended. If not, the comparison becomes a tool in identifying and eliminating the programming deficiency which caused the change. Comparisons are presented in two forms: graphical and textual. The graphical comparison is embodied in the `InteractionDiagramDeltaView`, which presents both the old and new diagrams, with operations that have been deleted highlighted in the old diagram and operations that have been added highlighted in the new diagram. "Highlighting" here means that the given operation is drawn in red. Highlighted operations are the only ones drawn in red, a colour commonly used to bring attention to things, just as differences are to be brought to the attention of the user. The textual information, when saved to a file, provides a long term record of changes. The diagram evaluations that are part of the textual view provide an idea of how the execution of the use case is controlled, the appropriateness of which can then be evaluated by the user based on the kinds of functional and implementational changes most likely to occur in the application concerned. The list of changes that forms the rest of the textual information co-operates with the `InteractionDiagramDeltaView` in

that selecting a change from the textual list causes the delta view to focus on that change, so that it can be located quickly in the diagrams for investigation in conjunction with surrounding messages by the user.

It is expected that development teams will find this functionality useful, once built into a production system, both for monitoring progress and for identifying programming deficiencies.

Chapter 4

Generating Diagrams for Applications with Graphical User Interfaces (GUIs)

4.1 Rationale

The ease of use associated with graphical user interfaces (GUIs) makes them an important feature of much of today's "user-friendly" software. Hence, many developers work on them, or on applications that use them. Therefore, for the sake of usability, support for GUIs is an important thing to add to IDG, since the original version does not have it. In the rest of this chapter, the reasons for the absence of GUI support in the original IDG will be explored, and the way in which the problems were circumvented in implementing GUI support will be explained.

4.2 Applications Using GUIs but No Dialog-Type Boxes

There are two reasons for the absence of GUI support in the original IDG. First, control is not being shifted properly between the generator and application windows (which correspond to other processes). Second, the volume of messages involved in executing graphical applications (typically hundreds or even thousands, many of which relate to window layout) makes it impossible to have an IDGDebugger run such

applications by hopping through each message regardless of whether or not it is to be displayed. When running use cases involving GUIs, the original IDG simply works for a long time and then come up with an 'Application Error.'

The first problem, which originally showed itself in a different context, namely that of transferring control from the main generator window to the delta view, was dealt with in the `UseCaseView>>perform` method which initiates use case execution under an `IDGDebugger`. The required modification involved the introduction of a new process. Both versions of the method are exhibited in Fig. 4.1.

The second problem was solved by changing the technique used to move from operation to operation - instead of 'hopping' through every message, 'jumping' is now used (the terms 'hopping' and 'jumping' take on the same meanings here as in the standard Smalltalk debugger described in [Digi 92a, pp. 116-117, 293-297]). This means that the generator sets breakpoints at the methods which are considered 'interesting' according to the initial filter settings, and then 'jumps' from one breakpoint to the next as the use case is executing. Hence, very little time (comparatively speaking) is spent dealing with the common window layout operations when they are not considered interesting.

Admittedly, even now, a user cannot successfully interact directly with a window that is running under the Interaction Diagram Generator and thus interacting with an `IDGDebugger`. However, that problem is not serious, since direct interactions are prone to error. That is, the user may make an error in attempting to repeat a sequence of events (button clicks, menu choices, etc) that was executed in a previous session, resulting in invalid comparisons of diagrams that depict different use cases. The technique chosen

```

(a)
perform
    "execute use case"
    [codes]
    interactionDiagram := InteractionDiagram withTitle: currentUseCase name.
    interactionDiagram pane: (self paneNamed: #interactionDiagram).
    self clear.
    codes :=
        (currentUseCase tempVariable,
         'IDGProcess
          queueWalkback: "IDGDebugger"
          makeUserIF: CurrentProcess isUserIF
          resumable: true useCaseView: self.',
         currentUseCase restCodes, '.').
    Compiler evaluate: codes in: self class to: self notifying: nil ifFail: [].

```

```

(b)
perform
    "execute use case"
    [codes mainProcess answer]
    interactionDiagram := InteractionDiagram withTitle: currentUseCase name.
    interactionDiagram pane: (self paneNamed: #interactionDiagram).
    self clear.
    codes :=
        (currentUseCase tempVariable,
         'IDGProcess
          queueWalkback: "IDG Debugger"
          makeUserIF: CurrentProcess isUserIF
          resumable: true useCaseView: self.',
         currentUseCase restCodes, '.').
    mainProcess := CurrentProcess.
    (step) ifFalse: [self startProgressReport].

    Processor fork: [CurrentProcess makeUserIF.
                    answer := Compiler evaluate: codes
                        in: self class
                        to: self
                        notifying: nil
                        ifFail: [Transcript cr; show: 'Fail';cr].
                    mainProcess makeUserIF].
    [self finished] whileFalse: [Processor yield].
    (step) ifFalse: [self inProgressDialog close].

```

FIGURE 4.1: The UseCaseView>>perform Method - (a) original (b) new.

```

[launcher classAnalyzer maxKey numberOfInstancesButton]

launcher := CPLauncher new open.
(launcher paneNamed: 'CPClassAnalyzer') event: #clicked. "click the CPClassAnalyzer"
                                     "button."

maxKey := 0.
Notifier windows associationsDo: [:anAssoc]
    (((anAssoc value isKindOf: TopPane) and: [anAssoc value owner class ==
CPClassAnalyzer]) & (anAssoc key > maxKey)) ifTrue: [
    maxKey := anAssoc key.
    ].
].

classAnalyzer := (Notifier windows at: maxKey) owner.

"select InteractionDiagram"
(classAnalyzer paneNamed: 'classList') selection: InteractionDiagram; event: #select.

numberOfInstancesButton := (classAnalyzer mainView children values select: [:aPane]
aPane label = 'Count Instances')) asArray at: 1.
numberOfInstancesButton event: #clicked. "click the numberOfInstances button."
classAnalyzer close.
launcher close.

```

FIGURE 4.2: Code for a Use Case Involving a Graphical Interface.

instead was to have use cases defined in code which triggers the related events using the event: method. This code can be stored, like that of any other use case, and re-executed for comparison purposes. Fig. 4.2 shows the code for such a use case. It concerns opening a CPLauncher and from there requesting the opening of a CPClassAnalyzer to extract the number of instances of InteractionDiagram that exist in the system².

The use of "Notifier" here may be a source of confusion. "Notifier, the single instance of NotificationManager, dispatches system event messages to windows. It keeps

²CPLauncher and CPClassAnalyzer are classes defined in the SubPanels/V package,
©Cooper and Peters, 1992.

a collection of all open windows, translates event messages (instances of class WinMessage) to Smalltalk messages, and sends the appropriate selector to the window object that matches the WindowHandle in the message." [Digi 92b, p. 147] More specifically, the collection of windows maintained by Notifier is a dictionary, and the later a window is opened, the greater is its key. Thus, when an action performed in one window causes another to be opened (as when clicking a button in a CPLauncher causes a CPCClassAnalyzer to be opened), one can search the windows dictionary and identify the newly-created window as the one with the highest key. Identifying the new window allows events relating to it to be generated. The given code also uses the class of the newly-created view as an identifying characteristic. This is an extra safety precaution, in case multiple windows are created at close to the same time.

One significant down-side of this is that the code to generate the required events must be manually written by the user. It would be preferable if the code could be generated based on actions taken by the user while using the application to which the use case is to relate. This would involve trapping the events generated and using this information to construct the use case. It is highly likely that transparent modifications to system methods would have to be made. Mercury claims that its XRunner tool, which was built to support the automatic testing of GUI-based applications, can record the series of events generated as a result of user interactions with a window of a given structure, and store the sequence in a form that it understands, so that the sequence could be replayed by XRunner at any time on a window of the same form [Shep 94]. Mercury presents this as a tool for automatic regression testing (that is, testing to ensure that previously working features have not been compromised as an undesired side effect of a change to application code or the environment in which it operates). But, whether the point is regression testing or rerunning a use case in order to regenerate an interaction

diagram, the requirement for event **replay** appears to be unchanging. This would seem to imply that XRunner's technology (and more specifically, the technology associated with WinRunner, the version of the tool which is built for use under Microsoft Windows) would be useful in attempting to improve on the mechanism used for specifying the use cases of GUI-based applications. Further investigations are needed to resolve this issue. Perhaps this should be considered as part of some future work.

Responding to (modal) dialogs is also a potential problem. Modality presents its own challenges to the Interaction Diagram Generator, but in the next section, the problem will be discussed further and a solution presented for a large group of dialogs, namely all subclasses of `WindowDialog`.

4.3 Applications Using Dialog-Type Boxes

The modality of most dialog-type boxes means that once one is opened, the process which initiated it waits for it to be answered by the user and discharged. In general, this concept is simple, straightforward and useful. When trying to run a use case by generating events within code, however, a difficulty arises. Once a dialog appears, the execution sequence which initiated it is stopped, meaning that commands used to discharge it cannot be issued from within that sequence.

In the case of dialogs which are subclasses of `WindowDialog`, though, a solution has been implemented. The solution includes small modifications to class `WindowDialog` which are designed to be transparent to the general Smalltalk programmer. The idea is to set a global variable `InitialDialogCode` to refer to an array of source code strings before initiating a use case. The string referenced as `InitialDialogCode at: n` is to contain the


```

|aGreedGameWin|

InitialDialogCode := Array with: '(self paneAt: #text) contents: "2".
                                ((self mainView children values select: [:aPane] (aPane label = "OK"))
                                asArray at: 1) event: #clicked.'.

aGreedGameWin := GreedGameWin new openOn: (GreedGame new).
(aGreedGameWin paneNamed: 'rollButton') event: #clicked.

```

FIGURE 4.3: Code for a Use Case Involving a Prompter (a kind of WindowDialog), which is used to ascertain the number of players taking part in a game of GREED. The code string placed in the first position of the array assigned to InitialDialogCode causes the prompter to receive the response '2' and the OK button to be clicked; the prompter is thus discharged without user intervention when the use case is run.

code necessary to discharge the *n*th dialog generated in running the use case. This implies that the builder of a use case involving such dialogs must know a priori the expected sequence of dialogs, as well as the response to be given to each. This should not be a problem though, since use cases are supposed to follow only one well-defined execution path; that is, there should be no conditionals, as each conditional splits an execution path into two potential paths. Without conditionals, the execution path, including the sequence of dialogs, is clearly defined. This technique is demonstrated in Fig. 4.3.

There is another major group of dialog-type boxes, namely all subclasses of DialogBox. These include the system classes FileDialog, FindReplaceDialog, FontDialog, MessageBox, NewSubclassDialog and PrintDialog. They are built using Dynamic Link Libraries (DLLs). Those wanting to build new windows in Smalltalk will typically build them as descendants of the WindowDialog class, but those familiar with DLLs may choose to use them and subclass DialogBox. Unfortunately, it appears upon preliminary investigation that, although a conceptually similar approach to that described for WindowDialogs would likely solve the modality-related problems here as well,

implementation would require individual class modifications and in some cases, changes to DLLs. It was decided that, having provided a prototype solution for WindowDialogs, the extension to DialogBoxes would be left as future work.

4.4 Summary

Graphical user interfaces (GUIs) are common and becoming more so, both in business applications and in applications aimed at the private individual user. They are proliferating largely because their presence makes applications easy to use, thereby increasing the size of the potential market for the applications beyond the bounds otherwise imposed by the limited technical literacy of large segments of the population.

Given the importance of GUIs, more and more software developers are either working on them or working on applications that use them. Therefore, for the sake of usability, GUI-handling capability has been added to IDG.

There is one case where the GUI-handling capability fails. That is where the interface includes instances of subclasses of class DialogBox. The problem here is, at least in part, related to the fact that important parts of these classes are implemented in Dynamic Link Libraries (DLLs). Further research may reveal a convenient way to eliminate this limitation.

Chapter 5

Identifying, Differentiating and Viewing Application-Related Objects

5.1 What Objects should be Identified?

Consideration was given to the question of what types of information (other than those discussed in the preceding chapters) might be useful as well as manageable from an implementation stand-point. It was decided that, in accordance with what one sometimes sees in manually-drawn diagrams (eg. see [Pugh 93]), objects of a given class should be differentiated, each having its own column and distinguished through the use of numerical indices. This provides a more complete picture of the workings of a use case. The downside of this is the common pitfall of information management and use in general - when the volume of information (in this case, the size of the diagram) gets too large, it often becomes difficult to extract that which fulfils the user's immediate needs. Fortunately, [Wang 94] provided for post-filtering, that is, filtering after the diagram is completed.

In addition, it was decided to make it possible to inspect both the arguments to each method and the objects returned from those methods. Identifying returned objects is something one sometimes sees (and which has been intermittently done in academic

course work by the author of this thesis), while identifying arguments appears to be a new idea in the context of interaction diagrams. The class name of each returned object, truncated to fit a limited space, is placed on a dashed arrow from the column representing the receiver back to the column representing the sender of the returning message. The arguments, on the other hand, are not placed on the diagram itself as they, being variable in number, could clutter the display, making it difficult or impossible to read. Both returned objects and arguments can be inspected at the command of the user, just as the original generator presented method code on demand, in a separate window. This again provides a more complete picture of the interactions. Also, unlike the identification of returned objects, the identification of arguments was easy to implement (see section 5.3).

5.2 Distinguishing Among Objects of the Same Class

The concept underlying the distinction of individual objects is simple: store the actual objects associated with columns and operations, rather than only the classes of those objects, and then use the identity operator (`= =`), rather than equality (`=`), when comparing objects to see, for example, which one sent a particular message. (Here, and in the rest of this thesis, the phrase "objects associated with an operation" denotes both the sender and receiver of the message represented by the operation. Pointers to both are stored as part of the operation.) The actual objects must also be used instead of their classes in the sender stack, a data structure used locally to keep track of senders (the sender of one message is the receiver of the previous one), since only receivers are accessible directly using standard Debugger protocols (see [Wang 94]). As mentioned in section 5.1, each instance should have an index associated with it. This index was originally intended solely as a visual cue to appear in the column titles, making it easy for the user to identify and refer to specific instances of a given class. Later, it was

discovered that Digitalk's ObjectFiler, on which the diagram loading and saving functions depend, always provides a copy of an object it is loading for each reference to it. Thus, the index in the column title proved indispensable in removing duplicate objects and re-establishing links after reading in a diagram; all references to duplicate objects were redirected to the 'copy' held as the 'associated object' of the object's column, the column being distinguished using the column title which is made unique by the index.

5.3 Arguments and Returned Objects

5.3.1 The Process Stack

The arguments and returned objects of methods are maintained within the process stack, a copy of which is extracted as needed from the system by the debugger and assigned to the debugger's 'process' instance variable. Fig. 5.1 shows how this extraction is done. The format of the process stack is not easy to track down. This is understandable because, in general, users should not use such information. Nonetheless, it made it quite difficult to find the returned objects, though obtaining the arguments was easy.

```
|saveInterruptsEnabled|  
  
saveEnableInterrupts:=Process enableInterrupts."Boolean-are interrupts enabled?"  
Process enableInterrupts: false. "must extract stack with interrupts disabled"  
process := Process copyStack. "get a copy of the process stack - Smalltalk uses a"  
                                "primitive to do this in the copyStack method"  
Process enableInterrupts: saveEnableInterrupts. "put the old value back"
```

FIGURE 5.1: Code for Extracting the Process Stack from the System.

5.3.2 Arguments

It was relatively easy to find the arguments because methods to support this are included within Smalltalk, making it unnecessary to specify a particular index into the process stack within generator code. First, the instance methods 'argumentCount' and 'tempCount' (both private) of class CompiledMethod return the number of arguments and temporary variables, respectively, found in the receiver. Second, the instance method 'tempAt: <frameIndex> number: <tempNumber>' (also private) returns the object referenced by the temporary variable or argument number <tempNumber> within stack frame <frameIndex>. At the point when the process stack is extracted, the message being considered by the interaction diagram generator is described in stack frame number 1. If there are 'M' arguments and 'N' temporary variables (temps), then "tempAt: 1 number: <tempNumber>" returns (tempN, temp(N-1),...,temp1, argM, arg(M-1),..., arg1) respectively, where <tempNumber> = (1, 2, 3,...), that is, temporary variables first, followed by the arguments, each in reverse order. The reverse ordering is something one should expect when the data structure is referred to as a "stack".

Using these tools, an array containing the arguments to "method" in their correct order, say 'argumentsArray', can be constructed with the reasonably straightforward piece of code presented in Fig 5.2.

```
[argumentCount tempVarCount argumentsArray]
argumentCount := method argumentCount.
tempVarCount := method tempCount.
argumentsArray := Array new: argumentCount.
(1 to: argumentCount) do: [:i]
    argumentsArray at: i put: (process tempAt: 1 number: (tempVarCount +
        argumentCount - i + 1)).
].
```

FIGURE 5.2: Code for Extracting Arguments to the Current Method from the Process Stack and Placing them (in Order of Occurrence) into argumentsArray.

5.3.3 Returned Objects

In the apparent absence of similar support in the standard Smalltalk environment for the easy extraction of returned objects, and not having access to the code for the class methods of the Compiler class (which might very well have, were it available, revealed the internal mechanism for managing the return of objects), returned objects had to be located by inspecting the process stack directly, and checking code in the Process class for characteristics of messages, compiled methods and process frames that seemed important to process management, as such characteristics seemed likely to also be important in determining the position in the process stack where one would find the relevant returned objects (Here, it seems useful to clear up a small question of terminology - in Smalltalk, a **message** invokes a **compiled method** and is represented in the process stack by a **process frame**; an **operation** is the interaction diagram generator's internal representation of a message.)

The investigation yielded two major findings. Firstly, immediately following the explicit return of an object other than "self," the returned object is usually found at position seven in the process stack. The only known exception to this is that, when the method which is returning satisfies the "hasBlock" condition (discussed further below), the returned object is found within an object of class HomeContext (commonly called a "block") found at position seven in the stack. The second finding was that there does not appear to be a reliable way of distinguishing implicit returns, or explicit "^self"s, from other explicit returns using only the process stack. Thus another technique is used which is described in more detail in this section - examination of the executing Smalltalk code segments.

5.3.3.1 Isolating Executing Code Segments

In a standard Smalltalk debugger window (see Fig. 1.2), the message that is about to be executed and its receiver are always highlighted, including returns ("^"). This means that Smalltalk has to be able to find the position in a method of the code segment representing that message. Once the code segment in question is identified, Smalltalk syntax allows one to, in a straightforward manner, answer the question: "If this were the last statement in this method to be executed before returning a result, which of the following cases apply:

- the statement, after removal of parentheses and spaces, is "^self," in which case the receiver would be returned,
- the statement starts with "^" but is not "^self," in which case, after the code segment has executed, the returned object would be found at position 7 of the process stack, or in a block found there if the method "hasBlock,"
- the statement does not start with "^," in which case "^self" would be implied subsequent to the completion of the statement, and thus the receiver would be returned.

It is because of the latter possibility of implicit returns that one cannot necessarily identify the last statement before the method actually returns - when these returns occur, there is no cue (namely "^") by which to identify it from the code. Therefore, potential returned objects are recorded every time a code segment in a method of interest is executed (see section 5.3.3.2).

The key point here is that in order to reliably identify returned objects and thus give the user a better understanding of use case execution, the Smalltalk mechanism for isolating the currently running code segment had to be exploited. The author of this work


```

walkback: anInteger
"Private - Select the walkback line
at index position anInteger."
"Modified by J.A. Remove duplicate oldMethod local variable (now instance variable) Also make sure the positions dictionary is
updated so that drawDiagramMethodPositionStart:end: gets the right start and end positions of the code segment being dealt with
within the method frame which is the debugger's focus. Note: In the original generator, the final call was simply to a method called
drawDiagram'; the parameters are now necessary to the isolation of returned objects, a feature not found in the original generator."

| ic node tree start end headerSize |

headerSize := 12.
walkbackIndex := anInteger.
oldMethod := method.
browseWalkback
  ifTrue: [
    method := process methodAt: walkbackIndex]
  ifFalse: [
    oldMethod := nil.
    method := breakpoints at: walkbackIndex.
    method := method classField
    compiledMethodAt: method selector.
    method isNil "if doesn't exist"
    ifTrue: ["self removeBreak"].

(useCaseView isInterestedInClass: (method classField name) method: (method selector asString)) ifTrue: {
  (sourceDict includesKey: method) ifFalse: {sourceDict at: method put: (method sourceString)}.
  methodSource := sourceDict at: method. "added by J.A. Positions used to extract the current code segment executing (the"
  self updatePositions "==(1)" "highlighted segment in a standard Debugger window) and check if it involves a return"
  "^(^)" and whether or not such a return is a ^self. We need to distinguish ^self from other
  "returns in order o identify returned objects. sourceDict used to speed repeat lookups."
}.
oldMethod == method ifFalse: [
  self changed: #method:].
self
  changed: #tempList;
  changed: #instance;.
start := 1.
end := 1.
browseWalkback
  ifTrue: [
    object := process receiverAt: walkbackIndex.
    ic := process icAt: walkbackIndex - 1.
    walkbackIndex = 1
    ifTrue: {ic := ic + 1}.
    (positions includesKey: method)
    ifTrue: [
      tree := positions at: method.
      node := tree value whoContains: ic + 1 "==(2)"
      - headerSize.
      start := node position.
      end := node endPosition]].
  start = end ifTrue: {end := end + 1}.
  "methodPane showSelectionFrom: start to: end." "==(3)"
  self drawDiagramMethodPositionStart: start end: end. "YW/JA modified" "==(4)"

```

FIGURE 5.3: The IDGDebugger>>walkback: Method

does not claim to fully understand this mechanism. It uses private methods, including at least one (whoContains:) which comes from deep within the system and is not shown in

the visible class hierarchy. Complete understanding, though, is fortunately not necessary for use. This is one of the ideas related to code reuse.

The first key to the isolation of running code segments is found in the IDGDebugger>>walkback: method shown in Fig. 5.3. It is based on the standard Debugger's walkback: method. Certain lines have been numerically marked (eg. ****(<number>)**) for easy reference here. Line ****(2)** exhibits the use of the whoContains: system method noted earlier. Line ****(3)** refers to showing (ie. highlighting) the selection (ie. the message about to execute). The line is commented out because the ability to show a selection implies a text pane display which does not exist in an IDGDebugger. It is found in the original version of the method - in class Debugger which supports text pane displays - and appears here simply as a reminder of the change and an explanatory aid. Its existence implies that the segment of code to be highlighted, that is the segment denoting the message about to be sent, begins at position "start" and terminates at position "end" when the statement is executed in the Debugger method.

It turns out that the only extra other step needed to ensure that the start and end values are correct for IDGDebuggers is to appropriately maintain the positions dictionary. For standard Debuggers, this is done in the method: method, executed when the user selects a method from the upper left list pane of the debugger window (see Fig. 1.2). Since there is no such window for IDGDebuggers, the updating mechanism for the positions dictionary was extracted and placed in the IDGDebugger>>updatePositions method, with references to text pane displays removed. Therefore the call to this new method on line ****(1)**, in conjunction with the calculations involving "start" and "end" taken from the original walkback: method, ensure that the parameters passed to drawDiagramMethodPositionStart:end: on line ****(4)** correctly denote the starting and

ending positions of the active code segment within the focussed method's source code string. Given those positions, `drawDiagramMethodPositionStart:end:` plays the central role in extracting returned objects.

5.3.3.2 The Central Part of Returned Object Extraction

As mentioned previously, because of the existence of implicit returns - if a method is completed without encountering an explicit return statement (beginning with `"^"`), `"^self"` is implied - it is difficult to identify the final return of a method (recall from section 2.2.2.1 that debuggers return to a given method every time another method it calls finishes, but this does not necessarily imply the completion of the original method). Recursive message sends also pose a complication in isolating method returns - does a given reference to a method by the debugger indicate a return or a recursive invocation? The last part of the `drawDiagramMethodPositionStart:end:` (see Fig. 5.4) handles these issues. Note that, on the line marked **** (1)**, the code invokes a method which identifies the current operation. The method uses a process frame identifier, stored when the operation is initiated. It considers only active operations for the sake of uniqueness, since

```
"added by J.A. to isolate returned objects. See: singleStep method for additional processing
required to extract the returned objects for operations with methods satisfying #hasBlock."

currentOperation := self interactionDiagram activeOperationWithProcessFrameID: (process frameAt: 1 offset: 0) ** (1)
(currentOperation isNil) ifFalse: [
    activeCodeSegment := (methodSource copyFrom: start to: (end min: (methodSource size))) trimBlanks.
    ( ( (self copyRemoveWhiteSpaceAndParenthesesFrom: activeCodeSegment) = '^self') |
      ((activeCodeSegment isEmpt.y) or:
        [(activeCodeSegment at: 1) ~= '$']) ) &
      (method hasBlock not) ) ifTrue: [
        currentOperation returnedObject: (currentOperation toObject). "self"
    ] ifFalse: [
        currentOperation returnedObject: (process at: 7).
    ].
].
```

FIGURE 5.4: The Last Part of the `drawDiagramMethodPositionStart:end:` Method, which is the central part of the code which sets the returned object of an operation.

identifiers can be reused multiple times during the execution of a use case, but only one of the currently active operations can have a given frame identifier.

The code of Fig. 5.4 also checks if the current operation's method satisfies the "hasBlock" condition, in which case, when the method terminates, (process at: 7) is a block which contains, at a certain position within it, the object returned by the method invocation.

5.3.3.3 Handling the "hasBlock" Condition

For some methods, rather than make the returned object accessible directly by indexing into the process stack, the object is placed in a HomeContext (commonly called a 'block') which is found at the position in the stack where the returned object normally is. It has not been determined why Smalltalk does this, nor is it known exactly what criteria are used in deciding whether or not to do this for any particular method. It is suspected that efficiency concerns constitute at least part of the rationale, but this is not clear. One thing that is clear, however, is that CompiledMethods respond to the message 'hasBlock', which returns true if a HomeContext is used in this way, or false otherwise.

When a returned object is found in a block, the above code (Fig. 5.4) assigns the block to the instance variable returnedObject. Just before the diagram building completes, the actual returned objects are extracted from the blocks that have been stored. The code in Fig. 5.5 accomplishes this. Recall that before this code executes, the 'returnedObject' instance variable, for those operations whose related compiledMethods store their returned objects in blocks, refers to the block. The position in the block which contains the actual returned object is calculated from the block size and number of

```

useCaseView interactionDiagram sortedOperations do: [:anOperation|
((anOperation compiledMethod hasBlock) &
(anOperation returnedObject notNil) &
((anOperation returnedObject) ~= (anOperation toObject))) ifTrue: [
    blockIndex := (anOperation returnedObject size) - (anOperation compiledMethod tempCount).
    (blockIndex <= 0) ifTrue: [
        blockIndex := blockIndex + (anOperation compiledMethod tempCount).
    ].
    anOperation returnedObject. (anOperation returnedObject at: blockIndex).
].
].

```

FIGURE 5.5: The Code which Extracts Returned Objects from Blocks when the Method Returning "hasBlock."

temporary variables in the method. The fact that all extractions of returned objects from blocks are done together at the end of diagram generation has two advantages over incorporating it in the main code of Fig. 5.4. Firstly, it makes the main code much easier to follow. Secondly, extraction is only done when it has to be, namely on the final block in its final state, not every time `drawDiagramMethodPositionStart:end:` is invoked while tracking a given message send.

5.3.4 Pointers Versus Copies

The user will notice, on inspection of arguments and returned objects, that they are stored as pointers. That is, no matter how often a use case changes an object, or at what place in the diagram it is inspected, inspection will always show it in its last known form. For example, given the diagram in the display pane of Fig. 5.6, inspecting the returned object of the second operation (`BankAccount>>initialize`) after the whole diagram is completed will give one a view of the `BankAccount` as it stands at the end of the use case's execution (with the `DepositTransaction` completed, rather than simply initialized with a balance of zero). If one needs to look the object at different stages, one can run the use case one message at a time, using the 'Step' button, and view the object after each step which is relevant to current information requirements.

FIGURE 5.6: New IDGView (Repeated here from Fig. 3.1 for Convenient Reference)

The reason for this is that, rather than copy the objects that may have been modified after each operation, the objects presented in the diagrams are in fact the original ones, which may have been modified several times during use case execution. It was decided to use the original objects because the default 'copy' and 'deepCopy' methods found in class Object may not function appropriately for all objects, though the builder of an application may not be aware of this and may not think that application-related objects need to respond to such messages. Therefore, 'copy' and 'deepCopy' may not be overridden sufficiently to allow the interaction diagram generator to confidently use them

for any object. The user should not need to know whether or not the generator uses 'copy.'

One example of a situation in which the default copy methods don't perform the way they would have to in order to permit unrestricted transparent use (that is, use by programmers where the user is not aware of or inconvenienced by said use) is where the creation of a new object requires input from the user. The problem was detected while attempting to run a use case of the GreedGame application under an old version of the generator which tried to generically copy objects. A GreedGame is a video game which in its current implementation, asks the user from within its initialization protocol (invoked whenever a new GreedGame) to supply, via a Prompter, the number of players that are to take part in the game. Since the default copy methods start by requesting a new object of the receiver's class, an extraneous Prompter would pop up every time a copy was requested. This was definitely not transparent; in fact it was quite disconcerting until the reason for it was discovered.

There may be other potential problem areas for generic copying as well, but this has not been fully investigated. One obvious background consideration though is that copying large objects several times (if they were returned frequently by methods invoked in executing a use case) could result in shortages of memory.

5.3.5 Interface

Having looked at some of the technical issues underlying the ability to view arguments and returned objects, consider now the actual view provided, that is, the part of the user interface of the generator that supports this part of the generator's

FIGURE 5.7: Main Interaction Diagram Generator Window (Maximized and then Reduced in Size as a Bitmap) with Return Arrows Showing, the owner: Operation Selected and the Popup Menu for the Diagram Pane Displayed.

functionality. Fig. 5.7 shows a generator window after running the use case of the 'Simple Bank Account' Application called 'Open an Account', with the 'owner:' operation selected. An operation is selected by clicking the left mouse button with the mouse cursor situated on the associated method name.

The first two items in the popup menu, which are also found in the menu bar's 'File' menu, are disabled (and thus greyed-out) if no operation is selected. Otherwise, the 'Browse Method' item brings up a MethodBrowser on the method associated with the operation together with methods of the same name found along the inheritance hierarchy of the class of the receiver object. This is the same thing that happens when the user

double-clicks the left mouse button with the cursor positioned on top of a method name, as in Wang's original work [Wang 94]; the extra interface has been added. The 'Browse Arguments and Return Object' item brings up an `OperationArgumentsBrowser` on the selected operation. Alternatively, clicking the left mouse button with the mouse cursor placed on top of a method name in the diagram pane and the "shift" key held down brings up such a browser on the related operation. Fig. 5.8 shows an `OperationArgumentsBrowser`, generated by selecting the 'Browse Arguments and Return Object' item from the popup menu, after generating the diagram

FIGURE 5.8: An `OperationArgumentsBrowser` on `BankAccount>>owner:`.

shown in Fig. 5.7 and selecting the `owner:` operation. Notice from Fig. 5.8 that the browser has an 'inspect' menu. Following the example of the standard Smalltalk V/Windows 2.0 debugger window, this menu contains a single 'inspect' item which, when triggered, causes the opening of an inspector window on the object selected in the list on the left hand side of the browser. Arguments are represented in the list by the names given to them in the method (eg. the header of the `owner:` method considered in

Fig. 5.8 is 'owner: aName,' hence the list item 'aName'); they are listed in the order in which they are specified in the method, and followed by the returned object.

The third item in the diagram pane's popup menu, allows the user to change the view of a diagram by specifying whether or not to include return arrows. This choice can also be made using the menubar's 'View' menu. If return arrows are displayed, they appear only for nonreflexive messages (that is, messages not sent to 'self'), and they carry with them indicators of the kind of objects returned, based on class names, as in Fig. 5.7. (Trying to implement this for reflexive messages would result in a cluttered picture.) Double-clicking on one of these indicators brings up an inspector on the particular object that the indicator represents. This effect is the same as that produced by triggering the inspection of a returned object from an `OperationArgumentsBrowser` (see above), but the use of the browser is more general because it can work with reflexive messages - one can select an operation which depicts a reflexive message in the same way as one selects an operation which depicts a nonreflexive message.

Although there is clearly duplication within this interface, it is hoped that it will promote convenience rather than confusion for users, regardless of their degrees of familiarity with the generator. Those who are new to the system can use the menus, while others may choose to use mouse double-clicking and shift-clicking for improved efficiency of use.

5.4 Statement of Intent: Viewing Only, Not Debugging

The ability to view arguments and returned objects has been established. However, it is important to distinguish this functionality from that of a debugger. The generator, although it supports a limited subset of the capabilities generally associated

with debuggers, is not intended to provide full debugging capability. It is simply a viewing and comparison tool. If it was a debugger, it would have to permit users to view temporary variables, and probably to change values before continuing execution as well. Because items are extracted from the process stack after computations associated with them have completed, and since the internal stack, instead of the copy that `Process` class `>>copyStack` gives, is used for actual application-related calculations, the idea of rolling calculations backward and forward and changing values is likely quite complex and is beyond the scope of this thesis. Perhaps this may provide impetus for future work.

5.5 Summary

Differentiating and viewing the objects involved in a use case may in some cases be important to the understanding of the inner workings of its execution. Being able to view (inspect) arguments and returned objects helps the user understand the information flow involved. Tracking these objects can also clearly be useful for debugging purposes; the standard debugger window supplies this functionality.

Thus, unlike the original generator, the new one provides for the differentiation of objects of a given class, and for the identification and inspection of arguments and returned objects. Differentiating objects of a given class involves using identity tests and comparing the actual objects that are receiving messages, rather than simply comparing the classes of those objects (using either equality or identity tests). Isolating the arguments in a message send is easy because there are underlying (admittedly mostly if not entirely private) methods in the `Process` and `CompiledMethod` classes which strongly support this activity. The isolation of returned objects, however, is not well supported, which is not surprising since users should not need such a capability. Overcoming this problem required the examination of the process stack and the source code. When returns

are explicit other than ^self, the returned object is found at position 7 in the process stack, or within a block found there for messages whose methods that satisfy the "hasBlock" condition. Differentiating between explicit and implicit returns is done by using the Smalltalk mechanism for identifying the currently running code segment and checking it for a leading "^," the symbol denoting explicit returns.

There is one important limitation of the generator that relates to the identification and monitoring of objects involved in a use case. It is that due to some logistical problems surrounding the copying of objects (see section 5.3.4), pointers to objects, rather than copies are stored, so that an inspected object will always be presented in its last known state regardless of the point in the diagram at which it is inspected.

Despite this limitation, it is expected that the ability to monitor key objects which is provided by the generator will help users to understand the flow of information involved in use case execution, and thus to better comprehend use cases and application implementation in general.

Chapter 6

Other Improvements

6.1 The Applications Editor

In the course of one's work, one will want to add or modify applications and use cases. One may even wish to delete obsolete applications or use cases. Use case modification is a routine part of the development process. Changes in the list of classes involved in an application are also common in each cycle of incremental development. The addition and deletion of applications is less common, but is still important, especially in the initial stages of a project. Prior to the current work, in order to build new applications or modify existing ones by adding to, deleting from or changing their use cases, one had to add or change code directly, within the Smalltalk class hierarchy, according to a code template showing how to set up an application, including its collection of use cases and its default collection of "interesting classes." This was unnecessarily complicated for the user, requiring the comprehension of some implementation details. Even though the details in question are fairly straightforward, users should not have to concern themselves with such things.

Accordingly, the ApplicationsEditor was created, an example of which is presented in Fig. 6.1. An ApplicationsEditor contains three main panes: an applications

FIGURE 6.1: An ApplicationsEditor

list pane, a use cases list pane, and a text pane where detailed information is displayed and can be updated. Applications present within a Smalltalk image will appear in an ApplicationsEditor opened with the command

```
(ApplicationsEditor new) openOn: (Application all)
```

Applications can be added by clicking the application 'Add' button and typing in the name of the new application when prompted to do so. Selecting an application brings up a list of its default classes of interest in the text pane (if such a class no longer exists in the image, it does not appear in the list). As well, a list of the names of the use cases associated with the application appears in the use cases list pane. Changes to the class list

can be made and then saved using the "Save" item on the text pane's popup menu. A selected application may be deleted (removed from the Smalltalk image) by pressing the application 'Delete' button, and then confirming the deletion when asked to do so. Finally, the popup menu in the applications list pane and the 'Applications' menu on the menubar, aside from application addition and deletion items, also provide for updating the Editor, and for loading and saving applications to files. For reliable functionality, the Editor should be updated any time a class involved in one or more of the applications it presents is either removed from or reintroduced into the image. The application filing capabilities existed in the original generator, and have been retained despite the fact that applications can conveniently be created and reside within images. There are two reasons for the retention of application file saving and loading. First, it maintains compatibility with the original generator, in that applications saved using it can be loaded into an image using an ApplicationsEditor. Second, it generally supports the portability of applications between Smalltalk images. Notice, though, that the application filing functions have been moved from the generator itself to the ApplicationsEditor, so that all changes to applications existing in an image are accomplished through that tool only.

Use cases are managed in a similar manner, except that they cannot be loaded or saved independently of the application to which they relate. This is a reasonable limitation, given that use cases only have meanings in the context of their associated applications. They can be added to the selected application by pressing the use case 'Add' button and typing the name of the new use case when prompted. A use case can also be deleted by selecting it, pressing the use case "Delete" button and confirming the deletion. As with use cases, a popup menu and a menubar menu provide alternate ways to access the stated functionalities. The only remaining functionality essential to comprehensive

editability is that of modifying the code which is to run when the use case is triggered. This is handled in much the same way as the editing of an application's default list of interesting classes. That is, when a use case is selected, its code is displayed in the text pane; code in the text pane can be edited and then saved into the image by triggering the "Save" item in the text pane's popup menu.

The target of the editor was to enable convenient and comprehensive application and use case creation and editing. It is believed that the functionality just described embodies that objective. Note as well that the retention of applications and use cases within an image until explicitly deleted facilitates the continuity of work from session to session. Admittedly, though, the Smalltalk change.log expands each time a change is made (just as when regular methods are saved), so one may need to monitor one's disk space, compressing the log as appropriate, using the "Smalltalk compressChanges" command (see [Digi 92a, pp. 276-277]). Also, again just as when regular methods are saved, changes made in an ApplicationsEditor only become permanent (that is, persistent between Smalltalk sessions) if the image is saved after the changes have been made.

Having explained the utility of the ApplicationsEditor, the link must now be made between it and the IDG. The user must be able to select an application and open an IDG on that application, otherwise all the applications built in the editor would be useless except as a record of their logical existence in the image. Fortunately, the "File" menu of the editor contains an "Interaction Diagram Generator" item which, when triggered, opens an IDG on the selected application. If no application is selected, the menu item is disabled (greyed out). If both an application and a use case are selected, an IDG is opened on the application, with the use case selected in the use cases list pane of the IDG. Given this link, the easiest way to open an IDG (without having to know the

underlying mechanism for storing applications) is to open an ApplicationsEditor, using the command given above for that purpose, select an application and then trigger the "Interaction Diagram Generator" menu item. That way, the system fetches the application into the IDG, freeing the user from the need to understand the attendant details.

The result of this is that the user can now build and edit, in a straightforward manner, applications which are understood by IDGs. This is a significant addition to the functionality of the original generator, in that the usability of IDG is improved by its presence.

6.2 Zooming

Interaction diagrams can get quite large. Therefore, it was decided to mimic the "zooming" functionality of the ClassHierarchyBrowser, where pressing "Alt-Z" expands the code text pane to occupy the whole browser window, allowing more of a (long) method to be seen at once. That is, in the current IDG, unlike the original, pressing "Alt-Z" expands the interaction diagram and diagram title panes to occupy the whole generator window, so that more of a (large) diagram can be seen at once, making comprehension of the diagram potentially easier in some cases. An example of such a zoomed IDGView is given in Fig. 6.2. As in ClassHierarchyBrowser zooming, the expanded panes revert to their regular positions when "Alt-Z" is pressed a second time, to permit alternation between the zoomed and unzoomed views.

As well, it seemed logical that if zooming is useful for IDGViews, it would also be so for the individual ("old" and "new") diagrams in a delta view (see Fig. 3.3 for an example of a delta view). Accordingly "Alt-Z" zooms the diagram in an active delta view.

which has last been clicked on with the mouse. This provides the flexibility of being able to get a closer look at either diagram.

FIGURE 6.2: A Zoomed IDGView

Unfortunately, there exists a programming deficiency with respect to the zooming capability. The window layouts are in error. The distances involved are small but additive, such that after multiple rotations between the zoomed and unzoomed views, the scrollbars become noticeably disproportioned. Nonetheless, zooming is a helpful idea, and is at least usable in the current implementation.

6.3 Performance Improvements

There are two major sources of performance improvement over the original generator. One of which, namely the use of debugger "jumping" rather than "hopping," has already been discussed in section 4.2. Not only does this change enable the production of interaction diagrams for applications using GUIs, but also, by vastly reducing the number of messages that are checked by the generator to see if they should be drawn, it greatly improves the performance of the generator where the use cases involved in the application under consideration produce a high proportion of uninteresting messages. Notice that GUI applications generally fit this criteria, since users typically are not interested in the huge quantity of messages (hundreds or even thousands) sent during the opening and layout of any reasonably complex window.

The second source of performance improvement pertains to use cases where many interesting messages are displayed. Previously, each time a message was added to a diagram, the whole diagram (except the headings) would be redrawn. This was unnecessary duplication of work which was time consuming since graphical processing tends to be slow relative to simple data processing, in general. Now, messages are added incrementally, removing most of the duplication and thus enhancing performance. Diagrams are redrawn upon completion though, as part of the process of ensuring that they are retained in the display pane even if the IDGView becomes covered by another window or is moved partly off the screen. The system implementation of GraphPane (the superclass of IDGGraphPane) requires special actions to trigger the retention of pictures in these cases. The author suspects that part of the reason for the great degree of duplication used in displaying diagrams in the original generator may have been the desire to provide diagram retention at all stages of its production. However, as the case

where an IDG is covered during diagram production and then the user wishes to see the diagram again before it is completed is expected to be rare, it was decided to implement the reduction in duplication and capitalize on the associated performance improvement, as this is a higher priority than diagram retention during production, most of the time.

Admittedly, there is also at least one noticeable source of performance degradation, namely the reading of method source code from the change.log (ie. disk) in order to use it to differentiate "`^self`" from other returns when isolating returned objects (see section 5.3.3). Overall, however, there appears to be a performance improvement. The first source mentioned (debugger "jumping") provides the most dramatic improvement in the context of the kind of use case for which it was intended.

6.4 Redirected Printing

There are times (as in the production of this thesis), where one may want to take the image produced by "printing" an interaction diagram and feed it into another application (eg. a word processor). In order to allow this to be done conveniently, when the user requests the printing of a diagram, the system, in addition to asking about scaling, will also ask whether the diagram should actually be printed or dumped to a file (in bitmap format). The system then obeys the user's response, directing the output either to the current printer or to a user-specified file. Filed pictures can then be fed from the file into applications that recognize the bitmap format.

6.5 Summary

All the enhancements discussed here impact on the usability of the generator. The ApplicationsEditor makes it significantly easier to build new applications (a function

critical to the generalization of the tool) than was previously the case. Zooming is an idea borrowed from Smalltalk's ClassHierarchyBrowser which facilitates the viewing of an interaction diagram, by permitting more of it to be seen at once, thus potentially enhancing the user's ability to comprehend the workings of a use case quickly. Performance improvements ameliorate the overall efficiency of working with the generator, and efficiency is often a critical factor in assessing usability. Finally, the ability to direct diagram bitmaps to files allows the use of the images in other applications. For example, they might appear in reports generated using word processors.

There are other enhancements and completely new functionalities which would add to the completeness and/or usability of the generator. Other tools may be built to complement it. Some of these potential future developments are discussed in chapter 8. First, however, in order to practically bring together the existing functionalities, the use of the generator with two sample applications is exhibited in the next chapter.

Chapter 7

Sample IDG Applications

7.1 The Applications

In this chapter, the use of IDG will be demonstrated. Two sample applications are used for this purpose:

- a modified version of the bank account management example used in [Wang, 94], called "Simple Bank Account"; the new version includes service charges, which facilitate the demonstration of diagram comparisons (eg. one diagram including service charges versus one without them), and
- a simple GUI-based application, constructed around Cooper and Peters' CPLauncher, which did not require the implementation of any new classes; all of its "interesting" classes are found in any Smalltalk image which has WindowBuilder 2.0 and Subpanes/V integrated into it. This is used to exhibit the capability of IDG to handle applications having GUIs.

The Smalltalk code corresponding to the "Simple Bank Account" application is found in Appendix A. This appendix also includes the code that defines the use cases associated with that application. The code defining the use case associated with the "CPLauncher" application was given in Fig. 4.2. The source code associated with the "CPLauncher" is copyrighted by Cooper and Peters, Inc., and is of no particular interest here. Appendix B contains interaction diagrams for all of the use cases pertaining to the "Simple Bank Account" application. The diagram for the use case pertaining to the "CPLauncher"

FIGURE 7.1: Use Case Model for the Simple Bank Account Management System

application is given in Fig. 7.7. Both appendices present the version of the "Simple Bank Account" application where checking transactions spawn service charges, but withdrawal transactions do not.

7.1.1 The "Simple Bank Account" Application

The modified "Simple Bank Account" application will serve as the main demonstration vehicle for IDG. It involves five main classes, namely *Bank Account*, *CheckingTransaction*, *DepositTransaction*, *ServiceChargeTransaction* and *WithdrawalTransaction*. All the "<type>Transaction" classes are subclasses of the *Transaction* class. This makes sense because all of the subclasses of the *Transaction* class represent kinds of transactions. As well, some of the subclasses take advantage of the opportunity to use the inheritance which this hierarchical arrangement affords. The use case model for the "Simple Bank Account" application is given in Fig. 7.1. It shows the four use cases given in the original version of the application, plus a new one called

"Simulated transfer." It is called "simulated" because no specialized *TransferTransaction* class has been implemented. Transfers are simply enacted as a withdrawal from one account followed by a deposit to another. This use case has been added for completeness. As well, checking transactions and withdrawal transactions may or may not carry service charges. The ability to change this quickly in the application code is useful in testing and demonstrating diagram comparisons in the presence of changes in the operation and result of a use case. The only feature of IDG that cannot be shown using the "Simple Bank Account" application is its support for GUI-based applications.

7.1.2 The "CPLauncher" Application

The CPLauncher is a sample application written by Cooper and Peters (C&P). As its name suggests, it is a window from which other Cooper and Peters applications can be launched. A CPLauncher window is shown in Fig. 7.2. The user of CPLauncher presses one of the nine buttons in the window to start another C&P example application. Clearly, this means that the CPLauncher application should have at least nine use cases associated with it - one for each of the nine buttons. However, for the purposes of this discussion, only one use case has been associated with the CPLauncher application, namely "CP

FIGURE 7.2: A CPLauncher Window

FIGURE 7.3: A CPClassAnalyzer Window

Launcher to Class Analyzer." In this use case, the "Class Analyzer" button is clicked, bringing up a CPClassAnalyzer window like the one shown in Fig. 7.3. The class InteractionDiagram (known to exist in any Smalltalk image containing IDG) is selected and the "Count Instances" button is clicked. Once the number of instances appears, both the CPClassAnalyzer and CPLauncher windows are closed. The use case is straightforward, meant simply to demonstrate the utility of IDG with respect to applications involving GUIs.

7.2 Basic Usage

Most of that which appears here has been described in some form in previous sections, but here, the main usage procedures are given together in their expected order of occurrence. The emphasis will be on the main features added by the current work.

7.2.1 Building an Application

Before one can generate interaction diagrams, one must build the application to which they are to pertain (ie. write the application code), and then declare the existence

of the application using the ApplicationsEditor. Section 6.1 details the usage of the ApplicationsEditor for this purpose, as well as other manipulations of applications and use cases.

7.2.2 Generating and Saving an Original Diagram

The next step in using IDG after an application has been created is to generate and (optionally) save interaction diagrams pertaining to the application. Doing this is straightforward. If one wished to do it for the "Simple Bank Account" application, one would select the application, and then invoke the generator from the ApplicationsEditor as shown in Fig. 7.4. This brings up an IDGView on the application. Each use case can

FIGURE 7.4: An ApplicationsEditor from which the User is about to Initiate an IDGView on the "Simple Bank Account" Application.

FIGURE 7.5: An IDGView After the Generation of an Interaction Diagram for the "Make a withdrawal" Use Case of the "Simple Bank Account" Application.

then be selected and dealt with in its turn. Each one can be selected, and a diagram generated, by pressing the "Go" button in the IDGView. Return arrows can then be triggered using the popup menu in the display pane. In the case of the "Make a withdrawal" use case, this would lead to an IDG looking like the one found in Fig. 7.5.

In order to be able to compare the generated diagram to future versions of the interaction diagram for the same use case using the same filters, one should save the current one by pressing the "Save" button on the IDGView. This action will cause an `InteractionDiagramFileDialog`, like the one in Fig. 7.6, to appear. The dialog restricts the

FIGURE 7.6: An InteractionDiagramFileDialog Prepared to Save a Diagram Associated with the "Make a deposit" Use Case of the "Simple Bank Account" Application.

user's view to the use case to which the diagram that is to be saved pertains, for the sake of clarity as well as ease of ensuring that the diagram gets stored in a location which simplifies the generator's task of determining later which diagram files pertain to a given use case. Typing the name of the file in the "Directory Path" field and pressing the "Save" button completes the operation. For convenience, double-clicking on an existing file name (listed in the top right pane of the dialog) denotes an intention to overwrite that file. Whenever an overwrite is requested, the system requests confirmation before the action is performed, in order to safeguard against accidental erasure of the old file.

7.2.3 Diagram Evaluation, Regeneration and Comparison

At any time after initial generation, diagrams can be evaluated, regenerated and/or compared as described in chapter 3. Evaluation can be requested for the diagram currently displayed in an IDGView. Regeneration is the rerunning of a use case with the same set of filters as used originally. Typically, the diagram to be regenerated is taken from a file, but the currently displayed diagram can also be used as the source. When a diagram is regenerated, the new version is compared with the old, and new or missing messages are highlighted. Evaluations of both diagrams are also provided. The new diagram can be saved at the option of the user.

Regeneration is a key point in this thesis. It is useful in monitoring the progress of a development project, in that differences in the message flows produced during the execution of a given use case at two points in time are isolated. It might also be useful to have an (animated) display of the progression of change in the message flows for a given use case over an extended period of time. This information may provide insight into the kinds of mistakes made during development and the causes of those mistakes. One would expect that such insight would help developers avoid repeating their own errors, as well as those made by others on their development team. This could be considered for future research.

7.3 Usage with Applications Involving GUIs

As discussed in chapter 4, the function of IDG when it is used with applications involving GUIs is almost identical to its function when dealing with applications that do not have GUIs. The only difference is that the coding of use cases is somewhat more

complicated due to the need to identify and isolate windows, and to generate events relating to windows and their panes.

Fig. 7.7 provides proof of the generator's ability to generate simple diagrams involving GUIs. It shows the interaction diagram for the only use case implemented for the CPLauncher application described in section 7.1.2.

7.4 Summary

The tools discussed in this work allow the user to create applications and use cases, and then to generate and regenerate interaction diagrams for those use cases. The comparison of the old and new versions of a regenerated diagram, provided by the new IDG, is useful in tracking development progress, though each comparison is limited to the consideration of only two points in time. An animated view that considers multiple points in time could be considered in future work.

Throughout this thesis, the "Simple Bank Account" example has been used to demonstrate the functionality of IDG. In this chapter, the CPLauncher example has also been used, for the purpose of exhibiting the new IDG's ability to generate diagrams for applications having GUIs. The introduction of this ability constitutes a major improvement over the old IDG, since GUIs are commonly used in today's "real" applications.

FIGURE 7.7: An IDG-Generated Diagram for the "CPLauncher to Class Analyzer" Use Case of the "CPLauncher" Application.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The main problem with which this thesis concerns itself is the development of a tool which supports the regeneration, comparison and evaluation of interaction diagrams, with a view towards facilitating the understanding of changes in the message flows associated with given use cases, which in turn facilitates debugging and the monitoring of the development process.

The new IDG (an improved version of the one described in [Wang, 94]) enables the regeneration of interaction diagrams, using the same filter settings as were present in the original version. It permits the user to specify a mapping from each class in the old diagram that no longer exists (if any) to a class which does exist. This is not very flexible, in that all occurrences of a given non-existent class are always mapped to the same existent class, but it does get around the problem of non-existent classes appearing in old diagrams and it is easier to use than it would be if the user was forced to specify a mapping for each object.

Regenerated diagrams are compared with their original versions, and new or missing messages are shown in both graphical and textual forms. The textual form of the comparison can be stored in a file, which then serves as a long term record of the result. Comparisons include "evaluations" of each diagram. Individual diagrams can also be evaluated. The evaluations specify what percentage of message sends that each class is responsible for; in addition, they identify control classes based on this information. This is useful, given that [Jaco 92] provides some guide-lines as to the kind of control structure that is good for a given type of application. That is, using the given information in combination with the diagram itself, the user may be able to identify structural deficiencies.

In addition to accomplishing the main objective, several of the suggestions for future work given in [Wang, 94] have been implemented as part of the current work, including:

- support for applications having GUIs (other than IDG itself - ie. IDG can't generate interaction diagrams to describe its own operation. The reason for this is unknown.)
- an easy way to add and delete applications and use cases and support for persistent applications (applications that stay in the Smalltalk image, provided the image is saved before exiting) through the ApplicationsEditor
- the ability to distinguish among objects of the same class - each instance now has its own column in the generated diagrams
- the labelling of return arrows and the inspection of returned objects

As well, zooming (the expansion of the diagram pane to cover the whole IDGView) is now available, so that the user can see more of the diagram at once. Also,

diagram bitmaps are now exportable to a file, which can then be read by other applications. Finally, performance has been improved, making the generator more usable.

The generator has clearly been significantly enhanced over the original, which itself was a notable accomplishment for Wang. However, there is still potential for related future work.

8.2 Future Work

Several sources of future work have already been mentioned in various parts of this thesis. They are:

- **the need for a "cleaner" and more general solution to the problem of generating use-cases for GUI-based Applications** - The user should not have to specify events in code. Use case code should be generated automatically, based on actions performed by the user from within the interface (eg. button clicks, list item selects, menu item selects,...). Mercury's XRunner may be of assistance in implementing this.
- **the need to fix an intermittent deficiency** - Sometimes, the return from a GUI event handler is not properly recognized. For example, sometimes the CPCClassAnalyzer>>close" message in Fig 7.7 appears as part of the countInstances: message's execution in the diagram, instead of as a message coming from SystemBorder. This in turn causes the wrong returned object to be associated with the countInstances: operation. More research into this is needed.
- **the need to enable the generator to generate diagrams relating to its own functionality** - Not only would this be useful in explaining to others how IDG works, but it would also serve to increase the user's faith in the generator's reliability.
- **an exploration of the possibility of adding full debugging capability.**

- the provision of an animated view of the progression of change - This would be intended as an easy-to-understand way of looking at comparisons over more than two points in time.

Also, three of the suggestions for future work given in [Wang 94] have not been addressed here. They are:

- a more general diagramming tool - [Wang 94] suggested support for collaborations graphs (a pictorial representation of the information provided by the CollaborationsView). If one could store application and use case information in a general way, and add whatever other information is necessary, it should also be possible to support the generation of representations used by other object-oriented development methodologies, which would be a nice idea in that it would give more flexibility to the user.
- the creation of an interaction diagram editor so that the system could be used to support bottom-up as well as top-down development - Users should be able to create their own interaction diagrams, and to modify existing diagrams, whether they were created by users or generated by IDG. Another graduate student is working on an interaction diagram editor, but it is not yet linked with IDG. Clearly, the same idea applies to any other graphical representation which may be added later. A useful extension to this would be to allow for markup, that is, to allow managers and/or other developers to embed comments in an electronic copy of a diagram, which is then sent back to the originator who can take appropriate action (eg. design corrections, debugging).
- the use of postscript rather than bitmap format for printing diagrams - Postscript is a more standard format, and postscript drawings are more easily embedded in documentation. However, implementing this change would not be easy. First, an understanding of the postscript language would have to be acquired, and then this information would have to be coded in Smalltalk. Without an understanding of the inner workings of postscript, it is difficult to predict how long the coding task would take.

Appendix A

Smalltalk Code for a Simple Bank Account Management System (Application Code and Use Case Definitions)

A.1 Application Code

A.1.1 The BankAccount Class

```
Object subclass: #BankAccount
instanceVariableNames:
  'owner accountNumber balance transactions '
classVariableNames: "
poolDictionaries: " !

!BankAccount class methods !

egCreateAccount
  "BankAccount example1"
  | jimsAccount |
  jimsAccount := BankAccount new.
  jimsAccount owner: 'jim'.
  jimsAccount accountNumber: 1234.
  jimsAccount balance: 100.00.
  ^jimsAccount!

egMakeDeposit
  "BankAccount example2"
  | jimsAccount |
  jimsAccount := BankAccount new.
  jimsAccount owner: 'jim';
    accountNumber: 1234.
  jimsAccount addTransaction:
    (DepositTransaction new amount: 1000.00).
  ^jimsAccount!
```

egMakeWithdrawal

```
"BankAccount example3"
| jimsAccount |
jimsAccount := BankAccount new.
jimsAccount owner: 'jim';
    accountNumber: 1234;
    balance: 1000.00.
jimsAccount addTransaction:
    (WithdrawalTransaction new amount: 580.00).
^jimsAccount!
```

egWriteCheck

```
"BankAccount example4"
| jimsAccount |
jimsAccount := BankAccount new.
jimsAccount owner: 'jim';
    accountNumber: 1234;
    balance: 1000.00.
jimsAccount addTransaction:
    (CheckingTransaction new amount: 450.00;
        checkNumber: 1;
        date: (Date today )).
^jimsAccount!
```

new

```
^super new initialize!
```

!BankAccount methods !

= anotherBankAccount

```
"for testing purposes"
```

```
^((anotherBankAccount isKindOf: BankAccount) and:
    [(self accountNumber) = (anotherBankAccount accountNumber)]).!
```

accountNumber

```
^accountNumber!
```

accountNumber: aNumber

```
accountNumber := aNumber.!
```

addTransaction: aTransaction

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

balance

```
^balance!
```

balance: anAmount

```
balance := anAmount!
```

fileInActivate: anObjectFiler

"Added by J.A."

"SortedCollection of transactions must be converted to OrderedCollection (since ObjectFiler can't handle blocks, and specifically sortBlocks) for storage by ObjectFiler. Conversion done in local fileOutSurrogate: method (rather than, say, generally in SortedCollection) so that the proper specific sortBlock can be re-established here:"

```
self transactions: (self transactions asSortedCollection:
    [:transaction1 :transaction2 | transaction1 date < transaction2 date]).
```

^self.!

fileOutSurrogate: anObjectFiler

"Added by J.A."

"SortedCollection of transactions must be converted to OrderedCollection (since ObjectFiler can't handle blocks, and specifically sortBlocks). Conversion done here (rather than, say, generally in SortedCollection) so that the proper specific sortBlock can be re-established in fileInActivate:"

```
| aBankAccountSurrogate |
```

```
aBankAccountSurrogate := self.
```

```
aBankAccountSurrogate transactions: (aBankAccountSurrogate transactions asOrderedCollection).
```

```
^aBankAccountSurrogate.!
```

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: '.
```

```
self accountNumber printOn: aStream.
```

```
aStream nextPutAll: ' balance: '.
```

```
self balance printOn: aStream!
```

```

transactions
  ^transactions!

transactions: aCollection
  "Added by J.A."

  transactions := aCollection.! !

```

A.1.2 The Transaction Class and its Subclasses

```

Object subclass: #Transaction
  instanceVariableNames:
    'amount date '
  classVariableNames: "
  poolDictionaries: " !

!Transaction class methods !

new
  ^super new initialize! !

!Transaction methods !

= anotherTransaction
  "Added by Jeff Alguire for testing and investigation."

  ((self class) = (anotherTransaction class)) ifFalse: [^false].
  "now we know that anotherTransaction is a transaction and thus has the expected
  instance variables."

  ^((self amount = (anotherTransaction amount)) & (self date = (anotherTransaction date)))!

amount
  ^amount!

amount: anAmount
  amount := anAmount.!

applyTo: aBankAccount
  "For check and withdrawal and service charge only"
  aBankAccount balance:
    (aBankAccount balance - self amount).

(self isKindOf: CheckingTransaction)
  ifTrue: [aBankAccount addTransaction: (ServiceChargeTransaction new amount: 1.00)].!

```



```
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! !
```

```
Transaction subclass: #DepositTransaction
instanceVariableNames: "
classVariableNames: "
poolDictionaries: " !
```

```
!DepositTransaction class methods ! !
```

```
!DepositTransaction methods !
```

```
applyTo: aBankAccount
    aBankAccount balance:
        (aBankAccount balance + self amount)! !
```

```
Transaction subclass: #ServiceChargeTransaction
instanceVariableNames: "
classVariableNames: "
poolDictionaries: " !
```

```
!ServiceChargeTransaction class methods ! !
```

```
!ServiceChargeTransaction methods ! !
```

```
Transaction subclass: #WithdrawalTransaction
instanceVariableNames: "
classVariableNames: "
poolDictionaries: " !
```

```
!WithdrawalTransaction class methods ! !
```

```
!WithdrawalTransaction methods ! !
```

A.2 Use Case Definitions

A.2.1 "Make a deposit"

```
|jimsAccount|
jimsAccount := BankAccount new.
jimsAccount owner: 'jim';
    accountNumber: 1234.
jimsAccount addTransaction:
    (DepositTransaction new amount: 1000.00).
^jimsAccount
```

A.2.2 "Make a withdrawal"

BankAccount egMakeWithdrawal

A.2.3 "Open an Account"

BankAccount egCreateAccount

A.2.4 "Simulated transfer"

| jimsCheckingAccount jimsSavingsAccount |

**jimsSavingsAccount := BankAccount new.
jimsSavingsAccount owner: 'jim';
accountNumber: 1234.**

**jimsSavingsAccount addTransaction:
(DepositTransaction new amount: 1000.00).**

**jimsCheckingAccount := BankAccount new.
jimsCheckingAccount owner: 'jim';
accountNumber: 1235.**

**jimsSavingsAccount addTransaction:
(DepositTransaction new amount: 1000.00).**

**jimsCheckingAccount addTransaction:
(DepositTransaction new amount: 500.00).**

**jimsSavingsAccount addTransaction:
(WithdrawalTransaction new amount: 500.00).**

A.2.5 "Write a check"

BankAccount egWriteCheck

Appendix B

Interaction Diagrams for a Simple Bank Account Management System

B.1 "Make a deposit"

<< Generated on Dec 1, 1994 12:34:08 >>

Make a deposit
SystemBorder BankAccount aBankAccount1 DepositTransaction aDepositTransaction1

B.3 "Open an Account"

<< Generated on Dec 1, 1994 15:19:53 >>

Open an Account

SystemBorder BankAccount aBankAccount1

B.4 "Simulated transfer"

all Generated on Dec 1, 1994 09:09:29
for use by the user only

Abstract

[illegible]

B.5 "Write a check"

<< Generated on Dec 1, 1994 15:28:51 >>

Write a check

SystemOrder	BankAccount	aBankAccount1	CheckingTransaction	aCheckingTransaction1	ServiceChargeTransaction	aServiceChargeTransaction1
1	1	1	1	1	1	1
2	2	2	2	2	2	2
3	3	3	3	3	3	3
4	4	4	4	4	4	4
5	5	5	5	5	5	5
6	6	6	6	6	6	6
7	7	7	7	7	7	7
8	8	8	8	8	8	8
9	9	9	9	9	9	9
10	10	10	10	10	10	10
11	11	11	11	11	11	11
12	12	12	12	12	12	12
13	13	13	13	13	13	13
14	14	14	14	14	14	14
15	15	15	15	15	15	15
16	16	16	16	16	16	16
17	17	17	17	17	17	17
18	18	18	18	18	18	18
19	19	19	19	19	19	19
20	20	20	20	20	20	20
21	21	21	21	21	21	21
22	22	22	22	22	22	22
23	23	23	23	23	23	23
24	24	24	24	24	24	24
25	25	25	25	25	25	25
26	26	26	26	26	26	26
27	27	27	27	27	27	27
28	28	28	28	28	28	28
29	29	29	29	29	29	29
30	30	30	30	30	30	30
31	31	31	31	31	31	31
32	32	32	32	32	32	32
33	33	33	33	33	33	33
34	34	34	34	34	34	34
35	35	35	35	35	35	35
36	36	36	36	36	36	36
37	37	37	37	37	37	37
38	38	38	38	38	38	38
39	39	39	39	39	39	39
40	40	40	40	40	40	40
41	41	41	41	41	41	41
42	42	42	42	42	42	42
43	43	43	43	43	43	43
44	44	44	44	44	44	44
45	45	45	45	45	45	45
46	46	46	46	46	46	46
47	47	47	47	47	47	47
48	48	48	48	48	48	48
49	49	49	49	49	49	49
50	50	50	50	50	50	50
51	51	51	51	51	51	51
52	52	52	52	52	52	52
53	53	53	53	53	53	53
54	54	54	54	54	54	54
55	55	55	55	55	55	55
56	56	56	56	56	56	56
57	57	57	57	57	57	57
58	58	58	58	58	58	58
59	59	59	59	59	59	59
60	60	60	60	60	60	60
61	61	61	61	61	61	61
62	62	62	62	62	62	62
63	63	63	63	63	63	63
64	64	64	64	64	64	64
65	65	65	65	65

Bibliography

- [Booch 91] Grady Booch. *Object-Oriented Design with Applications*. RedWood City, California: Benjamin/Cummings. 1991.
- [CY 91a] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Englewood Cliffs, New Jersey: Prentice Hall. 1991.
- [CY 91b] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Englewood Cliffs, New Jersey: Prentice Hall. 1991.
- [Digi 92a] Digitalk, Inc. *Smalltalk/V For Windows Tutorial and Programming Handbook*. Los Angeles, California. Digitalk, Inc. 1992
- [Digi 92b] Digitalk, Inc. *Smalltalk/V For Windows Encyclopedia of Classes*. Los Angeles, California. Digitalk, Inc. 1992
- [Jaco 92] Ivar Jacobson. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Wokingham, England: Addison-Wesley. 1992.
- [LPWC 93] Wilf Lalonde, John Pugh, Paul White and Jean-Pierre Corriveau. Towards Unifying Analysis and Design, and Implementation in Object-Oriented Environments. *Proceedings of East EWOOPE'93, Bratislava, Slovakia, Nov. 1993*.
- [Pugh 93] John Pugh. Introduction to Object-Oriented Analysis and Design (course notes). Ottawa, Ontario, Canada: Carleton University. 1993.
- [RBPEL 91] James Rumbaugh, Michael Blaha, William Premerlan, Frederick Eddy and William Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall. 1991.
- [Shep 94] David W. Shepherd, Bell Northern Research, Private Communication.

- [Wang 94] Yun Wang. *Interaction Diagram Generator - A Tool for Visualizing Use Cases in Smalltalk*. M.C.S. Thesis, Carleton University, Ottawa, Ontario, Canada. 1994.
- [WWW 90] Rebecca Wirfs-Brock, Brian Wilkerson and Lauren Wiener. *Designing Object-Oriented Software*. Englewood Cliffs, New Jersey: Prentice Hall. 1990.

END

3 1-05-95

FIN