1

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

PRECISIONSM RESOLUTION TARGETS

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services Branch

Direction des acquisitions et
des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

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 Supporting Need-Driven Design and Programming

by

Jelber Sayyad Shirabad, B.Sc.

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

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

October 11, 1994

© copyright

1994, Jelber Sayyad Shirabad

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 0-315-98568-2

Canada

Name JELFER SAYYAD SHIRAZI

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

0434

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
Administrator 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0277
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 Science 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern 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 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology 0323

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 0358
Geography 0351
Gerontology 0578
History 0578
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0337
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
Ethnic and Racial Studies 0628
Individual and Family 0629
Studies 0630
Industrial and Labor 0700
Relations 0344
Public and Social Welfare 0709
Social Structure and Development 0999
Theory and Methods 0453
Transportation 0453
Urban and Regional Planning 0453
Women's Studies 0453

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 0306
General 0287
Anatomy 0308
Biostatistics 0309
Botany 0379
Cell 0329
Ecology 0353
Entomology 0369
Genetics 0793
Limnology 0410
Microbiology 0307
Molecular 0317
Neuroscience 0416
Oceanography 0433
Physiology 0821
Radiation 0778
Veterinary Science 0472
Zoology 0786
Biophysics 0760
General 0425
Medical 0996

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

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

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 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 0569
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 0575

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
Mathematics 0605
Physics 0986
General 0606
Acoustics 0608
Astronomy and Astrophysics 0748
Atomic 0607
Atmospheric Science 0798
Electronics and Electricity 0759
Elementary Particles and High Energy 0609
Fluid and Plasma 0610
Molecular 0752
Nuclear 0756
Optics 0611
Radiation 0463
Solid State 0346
Statistics 0984
Applied Sciences 0346
Applied Mechanics 0984
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 0994

PSYCHOLOGY

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

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis:

On Supporting Need-Driven Design and Programming

submitted by
Jelber Sayyad Shirabad. B.Sc.
in partial fulfillment of the requirements
for the degree of Master of Computer Science

A handwritten signature in black ink, appearing to read 'J. L. ...', is written over a horizontal line.

Director, School of Computer Science

A handwritten signature in black ink, appearing to read 'W. Leford', is written over a horizontal line.

Thesis Supervisor

Carleton University

October 11, 1994

Abstract

In this thesis we present an implementation driven approach for designing and programming object-oriented (OO) systems along with a prototype tool to support this method. The method is inspired from three main observations:

- (1) object-oriented software is organized around objects' responsibilities and collaborators,
- (2) these responsibilities are captured naturally by use-cases, and
- (3) programming can be viewed as the act of implementing these use-cases in an evolving iterative manner.

The support tools are intended to combine low-level design, implementation, and debugging facilities into one unified framework. The goal is to provide an environment which is designed to facilitate the application of the method. The tool assumes users develop their use-cases first and provides facilities to speed-up the process of generating code that makes the use-case work. We called this approach need-driven design and programming (NDDP) because the code is added based on the actual need imposed by other objects in the system, not by the designer's prediction of what might be needed. Furthermore the method strives to reduce the possibility of creating extraneous software during programming. The approach in conjunction with the proper support tools can help the programmer develop software in a shorter time, making it suitable for rapid prototyping, exploratory programming, and small project development.

Acknowledgments

I would like to thank my supervisor Wilf Lalonde for everything that he taught me and for giving me so much to think about. His thorough proof reading of the thesis had a definite impact on its quality. I would like to express my gratitude to him for his financial support during the course of this research. I also thank him for always suggesting challenging ideas, which have shaped my understanding of object oriented systems and Smalltalk. I would also like to thank Professors John Pugh and Louis Birta for their useful comments and suggestions during the thesis defense.

Special thanks go to our graduate secretary, Marlene Wilson, for being a very nice, cooperative and caring person and doing her best for me and my other colleagues. I would also like to thank Prof. Frank Dehne for his efforts in the Graduate Supervisor position. Many thanks to our software and hardware support staff, Murali Thiyagarajah, Marlene Elliott, Brian Fultz and Peter Choynowski for their cooperation during this work. The 600 dpi printout was made possible with Murali's assistance. Thanks to Rosemary Carter and Brian for arranging the demo computer for the defense.

I would particularly like to thank Dorin Sandu for lending me some of the reference books that I needed and his useful hints about software packages that I used to produce the thesis text. Thanks also to Prof. Jean-Pierre Corriveau for allowing me to keep his copy of "Object oriented design" by Grady Booch for a very long time. I also thank Raghunath for proof reading the conclusion chapter and providing helpful suggestions. I would like to thank Mohsen Rais-Ghasem, Michael Valenta, Khaled Noaman and Ravi Palepu for attending my thesis rehearsal session and providing me with useful comments. I also extend my thanks to all people who have been of assistance during my study at Carleton.

My hearty thanks to my father, mother, and sister for always having faith in me and their continuing encouragement and support specially throughout last two years, for which I am dedicating this thesis to them.

Table of Contents

Acceptance Sheet.....	ii
Abstract	iii
Acknowledgments.....	iv
Table of Contents	v
List of Figures.....	vii
Chapter 1 Introduction	1
1.1 Object Oriented Technology.....	1
1.1.1 The Object Oriented Software Development Life Cycle	3
1.2 Programming Environments	5
1.3 Motivation	6
1.4 Thesis Overview.....	9
Chapter 2 Need Driven Design and Programming	10
2.1 General Concepts.....	10
2.2 An Example	17
2.3 Summary.....	30
Chapter 3 Support for Need Driven Design and Programming	33
3.1 Need Driven Declaration of Variables.....	33
3.2 Unifying Browsers and Debuggers	40
3.3 Need Driven Method and Method Header Construction	48
3.4 Background Coloring	50
3.5 Summary.....	52
Chapter 4 An Example of Need Driven Design and Programming With Support .	53
4.1 The Bank Account Example.....	53
4.2 Summary.....	70

Chapter 5. On the Design and Implementation of the NDDP Browser.....	72
5.1 The Overall Design	72
5.2 The Detailed Design.....	75
5.2.1 Compilation for Execution Initiation	75
5.2.1.1 Undefined Local Variables.....	76
5.2.1.2 Undefined Global Variables	79
5.2.2 Compilation for Execution Continuation	82
5.2.2.1 Default behavior when an object does not.....	82
understand a message	
5.2.2.2 New behavior when an object does not understand a	84
message	
5.2.2.3 Starting a Method	87
5.2.2.4 Hop, Skip, Go and Debug	89
5.2.2.5 The Walkback Buttons	91
5.3 Summary.....	91
Chapter 6 Conclusion.....	92
6.1 Summary.....	92
6.2 A Discussion of NDDP and Its Support Environment.....	93
6.3 Future Work	95
Appendix A The Smalltalk/V Frame and Process Structure	97
References.....	102

List of Figures

Figure 1.1	The Waterfall Software Development Life Cycle	3
Figure 1.2	Object Oriented Software Development Life Cycle	4
Figure 2.1	The collection of use-cases describes the system behavior.....	14
Figure 2.2	Adding one method may result in the need to add other methods	16
Figure 2.3	Each use-case may cover part of the behavior of one or more classes.....	17
Figure 2.4	Method owner: is missing from the BankAccount class.....	19
Figure 2.5	The Class Hierarchy Diagrams for the Bank Account Example	29
Figure 2.6	The Parts Diagrams for the sample Bank Account Example	30
Figure 3.1	(a) Defining a temporary variable. (b) Defining an instance variable	36
Figure 3.2	Defining a Global Variable.....	37
Figure 3.3	An NDDP Browser can be opened from the File menu	40
Figure 3.4	The NDDP Browser	41
Figure 3.5	Starting a class method with HOP	43
Figure 3.6	Hierarchy Navigation Keys.....	47
Figure 3.7	Result of Executing the egCreateAccount method	49
Figure 4.1	Code for creating an account.....	53
Figure 4.2	The system asks the programmer to define the BankAccount class.....	54
Figure 4.3	The NDDP browser automatically manufactures the owner: method.....	55
	for the BankAccount class	
Figure 4.4	The NDDP browser automatically manufactures the header for the.....	57
	balance: method	
Figure 4.5	The NDDP browser dialog for defining variable balance	58
Figure 4.6	Inspecting the result of the code we wrote in a workspace	59
Figure 4.7	The NDDP dialog for defining global DepositTransaction.....	60
Figure 4.8	The NDDP dialog for defining global Transaction.....	61
Figure 4.9	The NDDP browser manufactures the amount: method for the	62
	DepositTransaction class	
Figure 4.10	Defining the amount: method in the Transaction class.....	63

Figure 4.11	Clicking on the Out button will show the previous method in the sender chain	64
Figure 4.12	Restarting the egMakeDeposit use case	65
Figure 4.13	The balance method manufactured by the browser	66
Figure 4.14	The browser prompting for a definition of davesAccount.....	69
Figure 4.15	The browser prompts for a definition of CheckingTransaction after defining temporary variable davesAccount	70
Figure 5.1	The Class Hierarchy Diagram for (a) NDDPBrowser..... (b) NDDPDebugger	73
Figure 5.2	The collaboration Graph for an NDDPBrowser and its associated..... NDDPDebugger	74
Figure A.1	A process and its frames	98
Figure A.2	A sample frame for a method with no blocks	99
Figure A.3	A sample frame for a method with blocks	100

Chapter 1

Introduction

1.1 Object Oriented Technology

As the cost of computer hardware decreases, the use of computers is increasing in business and industry in general. But barely half a century has passed since the appearance of the first generation of modern computers. Although software technology has lagged behind in hardware developments, there have been some remarkable improvements in the software development techniques. If we compare the software systems created today with those that were created a decade ago, we can see that software systems have grown in size and complexity. Brooks suggests that the complexity of software is not an accidental property but an essential one [Brooks 87, pp. 11]. One of the reasons for this increased complexity is that software is increasingly modeling the complexity of the real world [Booch 91, pp. 4][Wirfs-Brock et. al. 90, pp. 2].

Over the past several years, object-oriented technology has gained a foothold in many different areas of computer science as a technique for managing the complexity of software systems [Booch 91]. Abstraction is one of the fundamental tools that human beings use to cope with complexity. When designing object-oriented software, systems are decomposed into objects that are abstractions of the real world objects they represent.

Such abstractions are often further abstracted into more general abstractions in an inheritance hierarchy [Wirfs-Brock et. al. 90]. Objects can support modularity because the behavior of an object can be made independent of the internal details of its parts. This minimizes the interdependence between system components and results in the reduction of the complexity of the system [Goldberg et. al. 83, pp. 9].

Booch suggests that the object-oriented paradigm provides us with the following benefits:

- It allows us to exploit the expressive power of the object-based (supports objects but not inheritance) and object-oriented (supports objects and inheritance) programming languages.
- It encourages reuse of both software and design.
- It results in systems that are more resilient to change.
- It reduces the risk of developing complex systems, because integration is spread out across the life cycle.
- It appeals to our natural human tendency to identify with the software counterpart of tangible real-world objects.

An empirical study on reuse and the object-oriented paradigm indicates that object-oriented paradigm substantially improves productivity, mostly because of reuse [Lewis et. al. 91]. A more recent report from AT&T states that object-oriented technology has resulted in significant reuse of code, design and analysis; reduced development cycles; simplified system integration; and provided a more efficient method of generating and using requirements [Kamath et. al. 93].

Object-oriented languages can be divided into two groups. The first group consists of object-oriented extensions of existing languages, such as C++ and Objective C (extensions to C), LOOPS and CLOS (extensions to Lisp), and Object Pascal (an extension to Pascal). These languages are usually known as **Hybrid** languages. The second class of languages are known as **Pure** object-oriented languages. Eiffel, Self, and Smalltalk fall into this category. A survey of different object-oriented programming languages can be found in [Saunders 88].

1.1.1 The Object Oriented Software Development Life Cycle

Figure 1.1 The Waterfall Software Development Life Cycle.

An early model for the software development life cycle, known as waterfall model, is shown in Figure 1.1. This model [Sommerville 92] was proposed in reaction to the software crisis of the late 1960s, and helped promote the notion that software development was an engineering process. Although it is useful for managers to consider the phases of the software life cycle as distinct, in practice the real process of software development does not follow the waterfall model. Boehm discusses some of the issues that the waterfall model fails to address [Boehm 86].

Figure 1.2 Object Oriented Software Development Life Cycle.

The object oriented software development life cycle is illustrated in Figure 1.2 [Booch 91, pp. 200] Booch defines object oriented design and analysis as follows:

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

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

He also suggests that, "Object oriented design embodies an incremental, iterative process" [Booch 91, pp. 199]. As this diagram illustrates, steps in the process may feed back to each other. The evolution phase combines the traditional aspects of coding, testing, and integration. The result is the production of a series of prototypes which finally evolves into the final implementation.

1.2 Programming Environments

The first software development environments were environments to support the programming process [Sommerville 92, pp. 352]. They typically include programming language compilation/interpretation facilities, program debugging, and program testing tools. Programming environments can be divided into two groups.

(1) *General purpose environments* which provide a set of general-purpose software tools for the development of programs in different programming languages. UNIX is the best known of these environments.

(2) *Language-oriented environments* which support the development of programs in a single programming language.

Because language-oriented environments are intended to support development of systems in a specific programming language, knowledge about the language can be embedded into the environment. As a result, powerful language specific support facilities can be made

available. A very well known example of these environments is the Smalltalk environment. Smalltalk is often described as a combination of three different aspects: the language, a class library, and the programming environment. After more than a decade since the first official release of Smalltalk¹, it is still acknowledged to be one of the best language-oriented programming environments commercially available. As a matter of fact, some of today's C++ environments support facilities that were borrowed from Smalltalk; e.g., the Class Browser View in Symantec C++ pro [Symantec 93], the C++ Class Browser in COHESIONWorX C++ [Digital 94], or ClassExpert in Borland C++ [Borland 93]. A look at other state of the art programming systems shows that they support integrated edit-compile-debug features and at their best may also include a user interface builder as one of the available tools. In other words, the best that we can get in terms of the programming facilities is an integrated set of general purpose (language specific) tools² [Borland 93][Symantec 93][Turbo 91][Visual Basic 92].

1.3 Motivation

As we mentioned earlier, the object-oriented software development is best supported by an iterative process model. We believe that a methodology can be better exploited if there is a suitable set of tools to support it. Few methodologies cover the whole life cycle; some may focus entirely on the analysis phase, others on the design phase, for example. A relatively comprehensive comparison of different object-oriented methodologies (methods) can be found in [Monarchi et. al. 92][Fowler 92]. Generally, these methods provide

¹ Smalltalk-80 from ParcPlace Systems was first released in 1983. Smalltalk/V 286 was first released in 1988.

² There are also environments such as ENVY from OTI that include important features such as versioning.

techniques for finding relevant application classes and their external interface, but they usually provide few details about the implementation process.

Most of the methodologies are typified by the Wirfs-Brock [Wirfs-Brock et. al. 90] method which generally aims to define the interfaces of the objects by determining their responsibilities and collaborators. Another well known methodology proposed by Jacobson [Jacobson et. al. 93] focuses on the notion of use-cases (execution scenarios) to uncover the underlying objects, their interfaces, and their interaction patterns.

On the other hand, current language-oriented environments focus on the implementation phase. Consequently, they tend to provide general facilities such as editing, browsing, inspecting and debugging. This allows the programmer to use the facilities in any way he likes to develop the software. In our experience, an environment such as Smalltalk satisfies many of the needs that an object-oriented design and implementation demands. We also believe that many Smalltalk developers share common approaches to developing their systems — approaches that have not yet been articulated very well. We believe there is a pattern of development during design and implementation which is natural from the object-oriented programming perspective. Yet, the tools do not adequately support this pattern of development. We also believe that to better exploit any programming and implementation technique, the environment should provide tools that are directly designed to facilitate the application of the technique.

In this thesis, we will describe an approach to detailed design and programming, which we believe is suitable for object-oriented systems development. It is a refinement of techniques that we have been using for some time but for which suitable tools were

unavailable. It combines the notion that objects must have interfaces dictated by their usage patterns and their behaviors as in the Wirfs-Brock method [Wirfs-Brock et. al. 90], that usage patterns are best described through use-cases (or usage scenarios) as in Jacobson [Jacobson et. al. 93], and that use-cases are best represented in the formal notation of a programming language (in our case, Smalltalk). We will call this method which dictates the steps that must be followed to combine these three components into a successful pattern of software development, **need driven design and programming (NDDP)**. Thus, very loosely, $NDDP = \text{Wirfs-Brock (responsibilities)} + \text{Jacobson (use-cases)} + \text{a programming language as a formal use-case notation (in our case, Smalltalk)}$.

Although we believe the Smalltalk environment itself encourages an approach to programming such as NDDP, there are features that can be added to the current Smalltalk environments to extend its functionality to facilitate the NDDP methodology. This thesis will also introduce these new features to the Smalltalk/V environment to support this style of software development. We believe that combining NDDP with an environment that supports it will facilitate and speed up the design and programming process for many programmers.

It should be made clear that we do not claim that our approach is necessarily the best or the only successful strategy for developing object-oriented software. However, we do believe that it is a natural approach that, in the hands of experienced developers with the appropriate tools, leads to better designed software that can be developed faster than with the standard tools. It is applicable for small and perhaps even medium sized projects. It was not meant for large scale projects, because these projects require preliminary in-depth design. Also, large scale projects are carried out by teams of programmers, so that, issues

such as project management and team coordination should be addressed. Usually these projects require a well defined software development process that covers all the stages of the software life cycle. Software quality assurance, versioning, and configuration management are just some of very many topics that are involved in large scale projects.

1.4 Thesis Overview

In Chapter 2 we will describe the *Need Driven Design and Programming* approach. This chapter provides an overview of the process. To elucidate the process, we will be applying it to the design of a simple bank account.

Chapter 3 describes the features that we have added to Smalltalk/V to support Need Driven Design and Programming. This chapter acts as a user guide, and mainly describes the functionality of the system as seen from the user's point of view.

In Chapter 4, we show how the same process can be applied more conveniently and more efficiently — with the tools of Chapter 3. We illustrate this by re-implementing the bank account example discussed in Chapter 2, using our NDDP support environment.

Chapter 5 describes the implementation details of the NDDP support environment from both an architectural perspective and from the perspective of the extensions that had to be added above and beyond what is already in the base Smalltalk environment.

Finally, in Chapter 6, we review the important aspects of our work and suggest further extensions to the current work.

Chapter 2

Need Driven Design and Programming

In this chapter, the notion of *Need Driven Design and Programming* (NDDP) will be introduced. Experience has shown that the best way to explain any programming or design technique is to illustrate its use by applying the technique to sample problems. We will use a simple bank account example to clarify the ideas behind NDDP. The chapter will end with our conclusions about the strength and weaknesses of the technique.

2.1 General Concepts

Today, there are many object-oriented methodologies for analysis, design and implementation of software systems. Some of them are general, and some are geared toward a specific application area such as ROOM for real-time systems[Selic et. al. 94]. There have been reviews and comparisons of different object-oriented analysis and design techniques in recent years [Fowler 92][Monarchi et. al. 92][Jacobson et. al. 93][Martin et. al. 95]. What will be introduced in this chapter is an alternative method for designing and programming. It is our belief that the method itself has been practiced by experienced programmers and designers before, but as far as we know, it has never been documented

as a different method³. For example the following is a quote from Objectworks\Smalltalk User's Guide manual [ParcPlace 92, pp. 178]:

"This is one aspect of iterative development. You add a method to the model only as the need for such a method becomes apparent during coding of the user interface"

To our knowledge there is also no established term for this approach to designing and programming. We called it *Need-Driven Design and Programming* (NDDP) because it explains the very nature of the process clearly, although the term *Need-Driven* has already been used in literature.

Regardless of what analysis and design methodology is used, the followings are the key steps in developing object oriented software [Lalonde 94, pp. 373]:

- Finding the classes
- Determining the relationship between classes
- Determining the relationship between instances
- Determining the behavior of the classes and instances

After all, an object-oriented system consists of classes and objects, and these objects communicate with each other. Some objects are part of other objects, and every class and instance of a class has a unique behavior. Although the order of these steps may be different from one method to another and they may use different terminology, all of them must handle these steps in one way or the other. Wirf-Brok's methodology for example provides a host of techniques for finding the classes; focusing on the names in a

³ Wilf Lalonde in his book *Discovering Smalltalk*, uses this technique to illustrate evolutionary software development and design [Lalonde 94]. Although the chapter describes general design concepts, it employs the technique to clarify the process. The term need-driven appears in this book as a synonym for requirements driven and as such as been used elsewhere. However, it has never been used in the formal sense that we are using it.

requirements document, imagining typical responsibilities and then postulating what happens in typical situations.

The classes involved in an application along with the helper classes can generally be determined by imagining typical usage patterns for the application. Such execution scenarios or use-cases can also be written down for others to understand. Use-cases are sometimes written in English prose (as is done by analysts using Objectory [Jacobson et. al. 93]), but they can also be written directly in a programming language as example code that should ultimately be executable once the application is complete. Once a designer has documented an illustrative number of use-cases, a programmer can use them to infer what classes are needed and also the required behavior of the classes and their instances. By the behavior of an object, we mean the set of messages that it responds to, and the precise manner that it responds to these messages. An alternative to use-cases is to have the designer provide the programmer with the set of classes needed along with a general description of the requirement for each class. The programmer can then fill in the details. Alternatively, the designer can attempt to provide a full specification for the classes and leave little design decision to the programmer. In any event, there are problems with this approach. It is very easy for the designer to request methods that are not really necessary, i.e. methods that have not been used in other methods, or methods that are not quite what is needed. There can also be methods that are too specific, methods with the correct intent but the wrong types or number of parameters, or simply poorly named methods. In any case, introducing a method because we "believe" it might ultimately prove useful is quite different from introducing a method because it is needed in the implementation of an existing method. Of course, the more experienced the designer is, the better the chances that the predicted methods will prove to be useful. The later has much more chance of

surviving the implementation stage and becoming part of an object's behavior than the former.

If we only add methods to an application because they are used in the implementation of other methods, we will end up in the end with fewer extraneous methods than if we added them based on possible future utility. The standard Smalltalk image has very few unreferenced methods. When we can find one, it generally turns out to be a method that is not useful — a method that the original designer forgot to delete.

Although NDDP is independent of the language of implementation, it can be better exploited in a language that has a powerful programming environment. In this thesis, we will be illustrating the approach in the context of Smalltalk and so will be using standard Smalltalk terms such as *classes*, *methods* and *messages*. But the approach can equally well be applied in other languages; e.g., Ada or C++.

The concept behind **need-driven** design and programming is very simple, and it's the simplicity that perhaps will make it even more popular in the coming years. Basically, the idea is that the behavior of an object is determined by the actual needs imposed by other objects, not by the designer's anticipation for possible need. Of course, the behavior of these other objects is determined in turn by the requirements of yet more objects. Where does the process bottom out? — at use-cases. To properly design our objects, then, we need to start with the use-cases and work in the opposite direction. Each use-case will dictate what objects are needed and what behaviors those objects will have to provide. So we need to start designing by writing typical execution scenarios — use-cases for our application.

Jacobson defines a *use-case* as "*a complete course of events in the system, seen from a user's perspective*". The other definition that we are interested in is "*a specific way of using the system by performing some part of the functionality*" [Jacobson et. al. 93, pp. 159].

Generally, the NDDP process can be divided into two steps:

1. Collecting the use-cases.
2. Making the use-cases work.

As we said earlier, a use-case can be described in a natural language such as English. Such a description can be very useful in the requirement analysis phase of the software life cycle and it could very well be a base for design and implementation phase use-cases as it is in Objectory.

Figure 2.1 The collection of use-cases describes the system behavior.

According to Jacobson, we can obtain a picture of how the object fits into the system by describing different scenarios or use-cases in which the object takes part and communicates with other objects. It is not the individual use-case but the totality of all the use-cases that fully describe an object's context and consequently the requirements imposed by the other objects. The object's interface can be deduced from these scenarios

[Jacobson et. al. 93]. The union of all the use-cases for all the objects thus defines the required functionality of the system (see Figure 2.1). In general, each use-case cuts across the behavior of many different kinds of objects at once and it is this property that is exploited in designing object oriented software.

To be maximally precise and maximally effective, we believe that a use-case should be written in the implementation language — in our case Smalltalk. The use-cases or needs are specified as part of the analysis phase. NDDP, then is concerned with realizing these needs. Of course, it is always possible for design or implementation issues to provide feedback that effects decisions made in the analysis phase. In those cases, new use-cases may need to be introduced and older ones made obsolete. So analysis and NDDP are deeply intertwined. To quote Fowler [Fowler 92, pp. 3], " It is worth pointing out that most would agree that analysis is the design of a conceptual (logical) model; and there is arguably more 'design' in analysis than there is in design!". NDDP is not concerned with changing the relationship between analysis and design. But it is deeply committed to lessening the gap between design and implementation. This will result in earlier feedback from implementation decisions to design decisions and as a consequence provide useful feedback about the use-cases themselves.

The next step in the need-driven design and programming process is to define the missing methods and the missing classes used in the use-case. One way to do this is to mentally simulate the use-case. As we follow the flow of control, we can add methods (and classes) that are missing. Adding a method may result in the need to add other methods in one or more classes (and also to add other classes) as shown in Figure 2.2.

The process continues until we have defined all the methods and classes required to make our use-case work. The same scenario is repeated for every use-case until we have defined all the classes and methods required for the application, as illustrated in Figure 2.3 .

Figure 2.2 Adding one method may result in the need to add other methods.

In Figure 2.3, Use-case 1 covers part of the behavior of class A and C. Use-case 2 covers classes A, B, and D, and Use-case N covers B and D.

Figure 2.3 Each use-case may cover part of the behavior of one or more classes.

2.2 An Example

In this section, we will try to illustrate *Need Driven Design and Programming* (NDDP) by applying it to the design and implementation of a simple Bank Account⁴. The example is relatively simplistic — its sole purpose is to clarify the ideas behind NDDP. The issues

⁴ This is a modified version of a sample application that appeared in the M.CS. thesis by Yun Wang [Wang 94]

involved in designing a realistic bank account, say as part of a bank system, are definitely much more complex than what can be considered in this limited space, but it is our belief that the process applied in this example can be scaled up to bigger systems. This will be discussed later in this chapter. The example will be implemented in Smalltalk.

As the requirements for this example we would like to be able to:

- Create a bank account for an individual,
- Deposit money into the bank account,
- Withdraw money from the bank account,
- Write checks to withdraw money from the bank account,
- Record all the transactions related to a specific account.

As we mentioned earlier, our implementation starting point will be a use-case that illustrates how to get and initialize a bank account. Consider the following:

egCreateAccount

```
"BankAccount egCreateAccount"  
| davesAccount |  
davesAccount := BankAccount new.  
davesAccount owner: 'Dave'.  
davesAccount accountNumber: 1234.  
davesAccount balance: 100.00.  
^davesAccount
```

We first create an instance of a bank account from a class with an appropriate name such as *BankAccount*. An account belongs to a person, so we might specify the owner of the account using a method such as *owner:*. In this use-case, we decided that an account should also have an account number which we can specify using a method such as *accountNumber:*. Since a new account can have an initial balance, we might also specify

this opening balance with a method such as **balance:**. In the last step, we return the newly created account. This simple use-case captures the following design decisions.

- A class named *BankAccount* must be created.
- The *BankAccount* class most likely will have at least three instance variables for holding information about owner, the account number, and the balance.
- The behavior of a *BankAccount* object will definitely include methods **owner:**, **balance:**, and **accountNumber:** for setting the corresponding information.

For the sake of simplicity, we decided to represent the owner as a string. In a more realistic context, we might have chosen to create a new *Person* class to keep track of the owner. But we won't pursue that direction here. Additionally, if we had not decided on the responsibilities of a *BankAccount* object before writing the use-case, then writing the use-case will clearly direct us toward finding them. The use-case itself is part of analysis but the detailed syntax and the specific names for the messages is part of design.

Figure 2.4 Method **owner:** is missing from the *BankAccount* class.

As we mentioned earlier, the next step gives us the opportunity to define the missing methods used in the use-case. The very first missing method in the **egCreateAccount** use-case is the **owner:** method. As indicated in Figure 2.4, we would get a Smalltalk error if we were to attempt to execute the use-case at this stage.

We should define this method in the *BankAccount* class. It is an instance method because we have sent the **owner:** message to *davesAccount* which is an instance of *BankAccount*. Setting the owner of an account does not seem to require any more than an assignment to a variable that can hold this information. It seems natural to use an instance variable to keep track of the owner of the bank account.

We clearly need to choose a name for this variable — *owner* seems like an obvious choice. So in this case we will end up with the following **set**⁵ method.

```
owner: name  
    "Sets the owner of the bank account to name."  
    owner := name
```

The next missing method is **accountNumber:**. This is again an instance method of the *BankAccount* class. Its sole purpose is to keep track of the account number. It too can be kept in an instance variable. Following the standard conventions, we will name the instance variable **accountNumber**. So the resulting method will appear as follows:

```
accountNumber: aNumber  
    "Sets the account number of the bank account to aNumber."  
    accountNumber := aNumber
```

⁵ It is common among Smalltalk programmers to refer to a method that is meant to assign its argument to an instance variable as a *set* method. The method designed to return an instance variable is usually called a *get* method. If the variable is called "owner", the get method is usually called **owner** too and the set method is called **owner:**.

The next statement to be executed if we continue the simulating of the use-case will be

davesAccount balance: 100.00.

Applying the same logic as above, we need to introduce a **balance:** method as follows:

balance: aNumber

"Sets the balance of the bank account to aNumber."

balance := aNumber

This introduces the need for an instance variable called *balance* in the *BankAccount* class.

Since all methods, instance variables, and classes required by the use-case have been provided, we can now actually execute the use-case in its entirety.

Before continuing with the next use-case, let's have a second look at the methods that we just added to *BankAccount*. All of them set the value of an instance variable. An experienced designer will know at this point that he will need the corresponding get methods, i.e. **owner**, **accountNumber**, and **balance**. This is of course a side effect of applying the **need-driven** technique which guided us to creating the set methods in the first place. We can add these methods to the system immediately, or if we have doubts about their utility, we can delay this decision and see if we can find a need for them while writing other use-cases.

For our next use-case, let's consider creating a bank account and immediately depositing \$1000.00 dollars.

egMakeDeposit

1 "BankAccount egMakeDeposit"

2 | davesAccount |

3 davesAccount := *BankAccount* new.

4 davesAccount owner: 'Dave'; accountNumber: 1234.

5 davesAccount addTransaction: (*DepositTransaction* new amount: 1000.00).

6 ^davesAccount

As we proceed with simulating the use-case, lines 3 and 4 of this use-case create a *BankAccount* with account number 1234 and owner *Dave*. There is nothing new here. Since we have already defined **owner:** and **accountNumber:** methods, these methods too will work. So, we can proceed with the next method which is **addTransaction:**. Since there are many kinds of transactions that can be applied to a bank account, we decided to create a class for each type of transaction. Again this decision can be the result of a previous analysis stage or we could be performing analysis in parallel. Although it can be argued that having a separate class for each kind of transaction (instead of just one general transaction class) is a design decision, deciding to keep a record of all transactions is an analysis decision. In any case, to be able to execute this use-case we have to create a class called *DepositTransaction*.

Now suppose that we created the *DepositTransaction* class. The next method to be executed will be the **amount:** method. Every transaction, whether it be a deposit transaction or a withdrawal transaction, has an amount. So it seems natural to have an instance variable called *amount* and to set its value using an **amount:** method. We will define this method as follows:

```
amount: aNumber  
    "Sets the amount of the deposit transaction to aNumber"  
    amount := aNumber
```

If we continue simulating our use-case, we will find out that we have to define method **addTransaction:** which we have used but have not defined in the *BankAccount* class. If we want to keep a history of all the transactions applied to an account, perhaps we should keep them in a collection. The type of the collection required may not be very obvious at this time (should we use an *OrderedCollection* or a *SortedCollection*). For the moment let's use an *OrderedCollection*. We can always come back and make a change if it is

necessary. We also want to adjust the balance to reflect the amount of the transaction. One approach is to have the bank account determine what to do based on the type of transaction. Another is to make use of polymorphism⁶ and allow the transactions to determine what to do. We chose the second option. The following is the code for `addTransaction:`.

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

As it can be seen, we decided to call the *OrderedCollection* `transactions`. Writing this method ought to remind us of a very important point that we ignored so far. For this method to work correctly, the *transactions* instance variable should already be bound to an *OrderedCollection*. The fact is that it is a widely accepted practice to set all the instance variables of an object to a reasonable default upon creation of the object. In Smalltalk, it is common to have an instance method called `initialize` for this purpose, and then override the class method `new` with a method such as:

```
new
  ^super new initialize.
```

The `initialize` method for *BankAccount* might be written as follows:

```
initialize
  owner := 'Unknown'.
  accountNumber := 999.
  balance := 0.
  transactions := OrderedCollection new.
```

The next method *needed* to make this use-case work is the `applyTo:` method. The method should be added to the *DepositTransaction* class (it is not a bank account method because the receiver is *aTransaction*). Adding a deposit transaction to an account should not change the owner or the account number but it should increase the balance of the account

⁶ We say that a message is polymorphic if it is understood by many different types of objects (usually with different behavior).

by the transaction amount. Consequently, **applyTo:** must add the amount deposited to the balance as shown in the following method.

applyTo: *aBankAccount*

"Applies self to *aBankAccount*. "

aBankAccount **balance:** (*aBankAccount* **balance** + self **amount**).

We already mentioned that we might need get methods for some of our instance variables. This **applyTo:** method reveals the need for two get methods, namely **balance** for *BankAccount* and **amount** for *DepositTransaction*. These methods will return the *balance* and *amount* instance variables in turn as shown below.

instance method in BankAccount

balance

"Returns the balance of a bank account."

^balance

instance method in DepositTransaction

amount

"Returns the amount of a transaction."

^amount

Adding these two methods will allow us to execute our second use-case, **egMakeDeposit**, to completion.

Another possible complementary scenario is withdrawing an amount from an account. The use-case illustrated below:

egMakeWithdrawal

```
1 "BankAccount egMakeWithdrawal"
2 | davesAccount |
3 davesAccount := BankAccount new.
4 davesAccount owner: 'Dave'; accountNumber: 1234; balance: 1000.00.
5 davesAccount addTransaction: (WithdrawalTransaction new amount: 580.00).
6 ^davesAccount
```

The first four lines are familiar. In this use-case we create account 1234 with an initial balance of \$1000.00 for Dave and proceed to withdraw \$580.00 from Dave's account. Since we already decided to create a different class for each type of transaction, we create an instance of *WithdrawalTransaction* for the amount \$580.00. When we first defined *DepositTransaction* class, we assumed it to be a subclass of *Object*. However, a better choice is to create an abstract class called *Transaction* and have different types of transactions as subclasses. We should notice that methods **amount:** and **amount** can be shared by all types of transactions and can therefore be placed in the *Transaction* class. This means that we should move the **amount** instance variable from the *DepositTransaction* to the *Transaction* class and write the appropriate **new** and **initialize** methods. Now, we can create the *WithdrawalTransaction* class under *Transaction* and make use of inheritance to access the **amount:** and **amount** methods.

new

"Creates a new instance of *Transaction*."

^super new initialize

initialize

"Initializes an instance of *Transaction*."

amount := 0.

Now we have to implement the **applyTo:** method for the *WithdrawalTransaction* class to update the balance of the account. Withdrawing money from an account will cause the balance to be reduced by the amount of transaction. The following is the code for the **applyTo:** method in the *WithdrawalTransaction* class.

applyTo: *aBankAccount*

"Applies self to *aBankAccount* "

aBankAccount **balance:** (*aBankAccount* **balance** - self **amount**)

By adding this method, we have created all the methods needed to execute the **egMakeWithdrawal** use-case.

Our next use-case will be about writing a check. When we write a check, the money comes out of our own account. So it is similar to a withdrawal transaction. However, there is a small difference. A check usually has a check number associated with it. Normal withdrawal or deposit transaction using bank slips do not carry such a number. The **egWriteCheck** use-case is shown below:

egWriteCheck

```

1  "BankAccount egWriteCheck"
2  |davesAccount|
3  davesAccount := BankAccount new.
4  davesAccount owner: 'Dave'; accountNumber: 1234; balance: 1000.00.
5  davesAccount addTransaction: (CheckingTransaction new
    amount: 450.00; checkNumber: 1;
    date: (Date today subtractDays: 1)).

6  ^davesAccount

```

For our purposes, *CheckingTransaction* could be defined as a subclass of *Transaction* but it could equally well be defined as a subclass of *WithdrawalTransaction*. Let's assume the latter. Since *CheckingTransaction* is below *Transaction* in the hierarchy, the **amount:** method can be inherited. The **checkNumber:** method used in this use-case, however, has not been defined yet. Its purpose is to record the check number associated with check. An all-too-obvious choice for the name of the instance variable is *checkNumber* which we need to add to the class definition. We also need to add the get method **checkNumber** to the *CheckingTransaction* class. These methods are shown below:

checkNumber: anInteger

```

    "Sets the check number of the check."
    checkNumber := anInteger.

```

checkNumber

```

    "Returns the check number of the check."
    ^checkNumber

```

Now that we have added an instance variable to the *CheckingTransaction* class, we also need to add an initialization method.

initialize

```
"Initializes a checking transaction."  
super initialize.  
checkNumber := 0.
```

The next method to be written is the **date:** method which needs to be added to the *CheckingTransaction* class. Having a date for a check seems very natural but it is also true that every transaction could have a date as well. We could make this extension without effecting any of the existing use-cases as follows:

- Add instance variable **date** to the *Transaction* class.
- Add the **date** and **date:** get and set methods.
- Add the appropriate statement in the **initialize** method of the *Transaction* class.

It is also common for accounts to have their transactions sorted by transaction date. We can accommodate this change by replacing the *OrderedCollection* by a *SortedCollection* in the **initialize** instance method of the *BankAccount* class. The new additions have been shown below.

instance methods in Transaction

date: aDate

```
"Sets the date of the transaction."  
date := aDate.
```

date

```
"Returns the date of the transaction."  
^date
```

initialize

```
"Initializes the transaction."  
amount := 0.  
date := Date today.
```

*instance method in BankAccount***initialize**

```

"Initializes the bank account."
owner := 'Unknown'.
accountNumber := 999.
balance := 0.
transactions := SortedCollection new
    sortBlock: [:transaction1 :transaction2| transaction1 date <= transaction2 date].

```

So far, we have captured most of the important functionality of our simple bank account. We might also add **printOn:** methods to complete the behavior of our classes. In Smalltalk, **printOn:** methods are used to provide printable representations of objects: i.e., **print strings** [Lalonde 94, pp. 422].

*instance method in BankAccount***printOn: aStream**

```

"Prints as Dave's $50 BankAccount."
self owner printOn: aStream.
aStream nextPutAll: "'s $".
self balance printOn: aStream.
aStream nextPutAll: ' BankAccount'.

```

*instance method in Transaction***printOn: aStream**

```

"Prints as a $50 DepositTransaction dated August 1, 1994."
aStream nextPutAll: 'a $'.
self amount printOn: aStream.
aStream nextPutAll: ' '. self class printOn: aStream. aStream nextPutAll: ' dated '.
self date printOn: aStream.

```

*instance method in CheckingTransaction***printOn: aStream**

```

"Prints as a $50 CheckingTransaction dated August 1, 1994 numbered 4193"
super printOn: aStream.
aStream nextPutAll: ' numbered '.
self checkNumber printOn: aStream.

```


All of the above methods were written using methods that already exist. Consequently, we have finished the design and implementation of a preliminary version that satisfies our requirements, as captured by the use-cases provided so far. The resulting class hierarchy and parts diagrams are shown in the Figures 2.5 and 2.6 respectively. A more realistic example would require many more use-cases and perhaps more complex use-cases. There would be a need for a bank object and a whole host of additional transaction types. But in spite of the additional complexity, the process would remain the same.

Figure 2.5 The Class Hierarchy Diagrams for the Bank Account Example.

Figure 2.6 The Parts Diagrams for the sample Bank Account Example.

2.3 Summary

In this chapter, we tried to present the **need-driven** approach to design and programming. Although the method has been practiced individually by some expert object-oriented programmers, particularly in the Smalltalk community, it has received little attention in the software engineering literature.

We divided the NDDP process into two steps:

- Writing scenarios (use-cases) that capture the requirements of the application.
- Incrementally adding the required methods to make these use-cases work.

By applying this need-driven technique, we are seeking a closer match between the design and implementation. Additionally, the technique strives to generate fewer extraneous methods, because all methods are added by need.

The following observations about the NDDP process can be made:

- NDDP focuses on the behavior of an application as specified by the use-cases and uses the use-cases to generate the required decomposition. This is similar to, but quite different from, non-object-oriented techniques such as structured design.
- In NDDP, the use-case plays a very fundamental role. A use-case written in the implementation language bridges the gap between analysis, design and implementation in one place.
- NDDP by its very nature is highly interactive process. This does not mean that it can not be used without computer assistance, but it can be better exploited if it is supported by the programming environment.
- In an object oriented setting, evolution and incremental development play an essential role. NDDP in essence is an incremental approach. It provides the framework for making extensions — writing use-cases and making every use-case work is an incremental and disciplined approach to adding new functionality to the system. We incrementally add functionality to the system by adding the components dictated by a use-case; e.g. in an object oriented language, components include classes and methods.

- In NDDP, the designer's view of the world is captured by the use-cases. The better he designs the use-cases, the closer the resulting implementations are to the requirements of the application. As a result, a more experienced designer will come up with a better design and implementation.
- NDDP does not claim to create perfect implementations, but it does allow the developers to generate the first version very fast (if the appropriate support tools exist) and this version must match the requirements of the use-case (or the use-case itself must be changed). We believe the approach will be useful for rapid prototyping, exploratory programming, and small project development.
- We chose Smalltalk to describe the use-cases at the implementation level. But any other language would have worked as well, as long as corresponding incremental support tools are provided.

In the next chapter we will describe the extensions to the Smalltalk/V environment to support the NDDP process.

Chapter 3

Support for Need Driven Design and Programming

In order to make it easier for programmer/designers to develop their software using the **need-driven** methodology, we need to develop facilities and/or tools that directly support this process. In this chapter, we focus on four major extensions:

1. better support for automatically declaring variables,
2. eliminating the distinction between browsers and debuggers,
3. having the system generate methods and context dependent method headers automatically, and
4. distinguishing contexts for methods using color (e.g., are we executing a method or simply browsing it).

3.1 Need Driven Declaration of Variables

One way of supporting the **need-driven** programming process is to make it easier for the programmer to concentrate on the logical aspects of a piece of code that he is creating by eliminating the requirement to predefine all the variables that he needs. More specifically, we need to make the system take on more responsibility for managing the declarative aspects by initiating requests for declarative information when and if it is needed. This

eliminates the need for the programmer to go off on a side-track as a prelude to carrying on with the details of the code. As a rule, the system should be responsible for all declarative aspects, from the declaration for a local temporary variable or an instance variable, to a global reference like a class including error correction. Some Smalltalk systems notably VisualWorks, already provide some support of this nature but we need to go beyond that provided.

In the standard Smalltalk⁷ environment, all the identifiers used in a piece of code must already be defined before the programmer can compile the code. Since saving the text of the code by default forces it to be compiled, the programmer has no choice other than defining all the references before saving the code. This quite frequently requires opening another browser and adding the missing elements. In some cases, an entire class along with variables and supporting methods have to be defined. Switching between one browser and another can be annoying. It can also cause the programmer to lose track of the working context, particularly if additional side executions are needed in the midst of the existing side track⁸.

Generally speaking, we provide two new⁹ services to support **need-driven** declarations:

1. Semi-automatic definition of *local* variables.
2. Semi-automatic definition of *global* variables.

We used the term "semi-automatic" to indicate that it is system initiated but user controlled. A local variable in the context of this thesis could be a temporary variable or an

⁷ All the references to Smalltalk or Smalltalk/V in this chapter are to Smalltalk/V 2.0 for Windows, unless stated otherwise.

⁸ Although the above criticism is valid for Smalltalk environments, Smalltalk still has one of the most powerful program development environments.

⁹ Both services are new to Smalltalk/V. Semi-automatic definition of classes is new to VisualWorks.

instance variable. By a global variable, we mean any variable that is defined in the *Smalltalk System Dictionary* along with class variables. The programmer has the choice of choosing among a set of different possibilities by selecting appropriate options and filling-in the required information if necessary. The system is sophisticated enough to provide only those options that are valid. For instance, during the evaluation of an expression in a work space, instance variables are not one of the allowed definition types. Of course, they are allowed when saving a method in a Class Hierarchy Browser.

A sample local variable definition dialog (as provided in our environment) is shown in Figure 3.1. As mentioned above, a *local variable* could be either a *temporary variable* or an *instance variable*. Figure 3.1 (a) illustrates the situation where the system has provided **Temporary** as the default choice and also disabled the **Instance Variable** choice because it cannot apply in the given situation. In those cases where defining an instance variable is valid, the **Instance Variable** option is enabled. In the case of Figure 3.1 (a), if the user clicks on the **OK** button, the source code will be updated by adding *owner* to the existing list of temporary variables. The system correctly handles the case where *owner* is the first temporary variable to be defined as illustrated below:

method1

```
"The system must add 'I' delimiters."  
| owner |  
...  

```

method2

```
"No-delimiters needed."  
lexistingVariable owner |  
...  

```

Figure 3.1 (b) illustrates the situation where the user has elected to define *owner* as an **Instance Variable**. This is only permitted in a context where an instance method is being defined. In that case, the class for the method being defined is known. Since the instance variable can be defined in any of the superclasses of this class and still be accessible via

inheritance, all such candidate classes are listed in the drop down list of an associated combo box. By default, the current class is selected. Figure 3.1. (b) illustrates the situation where the user is viewing the list of candidates: *BankAccount* and *Object*. If he decides to choose *BankAccount*, he has simply to click on the OK button. Otherwise he will have to first select *Object*. Defining an instance variable will not result in any change to the method source code but will change the definition of the selected class. This class will be recompiled automatically without any further assistance from the programmer.

The third alternative is a typographical error. Although it is possible to provide suggestion lists as in Apple Smalltalk and VisualWorks\Smalltalk, we have not gone to that extreme. To correct a typographical mistake, the programmer overtypes *owner* by the correctly spelled variable name (e.g. *newOwner*) and clicks on the OK button. The system updates the source code and recompiles the new source. If the user decides to abort compilation, he can click on the Cancel button. This will cause the 'undefined' error message to be inserted in front of the undefined variable (as is done in the standard system).

Figure 3.1 (a) Defining a temporary variable. (b) Defining an instance variable.

A sample **Global Variable Definition** dialog (also provided in our environment) is shown in Figure 3.2. This dialog will appear if the compiler detects an undefined identifier that is capitalized.

The dialog box is titled "Define Global". It contains a "Define:" label followed by a text field containing "BankAccount". Below this, there are four radio button options: "Class", "Class Variable", "Global", and "Constant in Pool". The "Class" option is selected. To the right of the "Class" option, there are two groups of radio buttons: "Indexed" with "Yes" and "No" (where "No" is selected), and "Contains" with "Pointers" and "Bytes" (where "Pointers" is selected). Below the "Class" option, there are three text fields labeled "Super class", "Instance Variable Names:", and "Class Variable Names:". Below the "Class Variable" option, there is a text field labeled "In Class:" and a small square button. Below the "Global" option, there is a text field labeled "Value:". Below the "Constant in Pool" option, there are two text fields labeled "Pool" and "Value:". On the right side of the dialog, there are three small rectangular buttons stacked vertically.

Figure 3.2 Defining a Global Variable.

The default behavior in the standard system is to prompt the user and ask about defining the variable in the system dictionary. If he replies **Yes**, the variable is defined but not initialized. The programmer generally has to proceed on a side track to initialize the variable; e.g., by executing code in a work space. If he replies **No**, the standard '**undefined**' error message is generated.

The reality is that a capitalized variable can be either a *Class name*, a *Class Variable*, a *Global Variable*¹⁰, or a *Constant In a Pool*. These options are now available through our **Global Variable Definition** dialog. As is the case for defining local variables, not all options are possible at any particular time. Specifically, the class variable option is only available when the programmer compiles a method; for example, when he saves a method in a Class Hierarchy Browser. For each of the options, validity tests are performed to make sure that the information entered by the user is acceptable.

After entering the necessary information, the programmer should click on the **Apply** button. If the entered data is valid, the variable will be automatically defined by the system. The dialog remains open until the user clicks on the **OK** button. This allows the programmer to redefine the variable or define other variables in the same dialog session¹¹.

When defining an undefined variable as a class, the programmer should provide the name of the superclass. If he wishes, he can also enter the names of the instance and class variables along with the names of any pool dictionaries that class may use but none of these are mandatory. A special case occurs when the superclass itself is undefined. In this

¹⁰ The term **Global Variable** in this context has been used to refer to any variable that has been defined in the *Smalltalk System Dictionary* except those corresponding to class names.

¹¹ An alternative might be to eliminate the **Apply** button and have **OK** not close if anything invalid is encountered.

case another **Global Variable Definition** dialog will be opened which allows the user to define the superclass.

When the programmer chooses the **Class Variable** option, the name of candidate classes that can contain the class variable are shown in the combo box. Any one of the target classes can be selected as the recipient for the class variable.

The third option in the **Global Variable Definition** dialog allows the programmer to define a variable in the *Smalltalk System Dictionary* and also assign an initial value if it is necessary. As mentioned before, the standard system only permits the variable to be defined but not initialized.

The last option in the dialog shown in Figure 3.2 allows the programmer to add a variable to a pool dictionary. It also permits him to initialize the new variable.

Finally, if the variable is a typographical mistake, the programmer can simply correct the name. The new name replaces the old one in the source which is then recompiled.

This concludes our discussion of the **need-driven** declaration facilities introduced in our thesis. In the next section we will explain the run time execution features which are essential to support an automated Need Driven Design and Programming environment.

3.2 Unifying Browsers and Debuggers

In the standard Smalltalk environment, the programmer must continually switch contexts between browsing and debugging. He will typically write code for a while, then run a test which will often bring up a debugger. He will then investigate the source of the problem in the new debugger context until he has determined what changes need to be made. Very simple changes can sometimes be made directly in the debugger but the more typical situation requires a switch back to the browser to add more code or make complex changes to existing code. To speed up the implementation process, we need to eliminate the context switch between the browsing and debugging environments.

Figure 3.3 An NDDP Browser can be opened from the File menu.

The *Need Driven Design and Programming (NDDP) Browser* was designed to merge a standard browser and a standard debugger into one unified tool to facilitate **need-driven** programming. It also includes special features that have no counterparts in standard browsers and debuggers. An NDDP Browser can be opened by selecting the corresponding menu item from the **File** menu. Any application that uses the system **File**

menu will also have this option available. Figure 3.3 shows the File menu in the Transcript window. The **Default Browser** menu shown in this figure allows the programmer to switch between the standard system behavior and the NDDP support behavior, by checking the *NDDP Browser* menu item.

When NDDP Browser is chosen as the default, an NDDP Browser is automatically opened by the system when an object receives a message that it does not understand. In this case, in addition to opening the browser, the correct class is also automatically selected by the system and a header for the missing method created automatically. This will be discussed in more details later in this chapter.

Figure 3.4 The NDDP Browser.

A typical NDDP Browser is shown in Figure 3.4. Except for the buttons in the middle (to be referred to as the *Control Panel* buttons), it looks very much like a standard browser. The Control Panel buttons include (1) the **Hop**, **Skip**, **Go**, **Start Method**, **Home**, **In**, **Out** and **Debug** buttons and (2) the *Hierarchy Navigation Buttons*. These buttons can be selectively enabled/disabled depending on the context. The rest of this section is devoted to the Control Panel buttons along with the services provided by these buttons.

Since an NDDP Browser can be activated either by direct user request or automatically by the system, there are times that the browser is keeping track of a process¹². The two cases result in different behaviors for most of the buttons in the Control Panel.

The **Hop**, **Skip** and **Start Method** buttons can all start execution of a class method, provided that the method has no parameters; e.g., method *egCreateAccount* in Figure 3.4. The NDDP Browser simply sends the selector of the method to the class it belongs to. Without such a facility, programmers tend to use comments inside the method containing the appropriate code to execute. The programmer must still take the time to select the appropriate code and execute it using **Show It**, **Do It**, or **Inspect It**. This is better than typing the expression, but not as convenient as clicking on a nearby button.

When holding a process, the **Hop** and **Skip** buttons have the same functionality that corresponding Debugger's buttons have. In a Debugger window:

- **Hop** - Executes the next expression in the process being debugged.
- **Skip** - Executes the next expression in the selected method, and it also may execute several expressions in lower-level methods.

¹² This is a loose statement. Actually the NDDP Browser is keeping track of the process indirectly through a debugger. More details about this will be presented in the implementation chapter .

In the NDDP Browser, starting a class method with any of these buttons conceptually has the same effect as :

- Adding a **self halt** statement to the beginning of the method.
- Executing the method and opening a Debugger Window.
- Clicking on the appropriate **Hop** or **Skip** button.

Figure 3.5 Starting a class method with HOP.

Of course, all of this occurs without leaving the NDDP Browser. Subsequently, the programmer can continue stepping through the method using these buttons. Figure 3.5

shows the effect of clicking on the **Hop** button of the NDDP Browser in Figure 3.4. Note that method `egCreateAccount` had already been selected.

As we can see, the text pane containing the method code has shrunk to allow the main window to accommodate two new panes labeled 1 and 2. Also the background color of the method text pane has changed to gray to indicate that it is part of an executing process; it is an active method.

Text pane background colors and their meaning will be explained in a separate section of this chapter.

The list box labeled 1 in Figure 3.5 contains all the information that standard debuggers provide for an active method; i.e., self (the message receiver) along with all of the active method's parameter and temporary variables. It behaves just like the corresponding list box in a standard debugger; i.e., you can click on an entry to see its value or double-click on it to get an inspector.

The **Start Method** button offers a wider range of functionality. The programmer can start a method using this button in three different scenarios:

- The starting method is a unary¹³ class method that is not active.
- The starting method is an active method.
- The starting method is the missing method that activated the browser.

Conceptually the **Start Method** button does the same thing in all three cases; i.e. start executing a method. However, the corresponding implementation details are quite

¹³ A method with no parameters.

different. More discussion on this topic will be presented in the chapter on implementation.

The first scenario starts executing the selected class method and attempts to execute it to completion (unlike **Hop** or **Skip** which both stop the execution on the first statement). There are two possible results: (1) some message is not understood in which case the missing method is manufactured and displayed in the browser (the programmer can change it (or not) and force its integration into the environment by further clicking on **Start Method**; we discuss this further in the next section of this chapter), or (2) the original method runs to completion in which case it is displayed as an ordinary method.

The second scenario allows the programmer to start any of the methods in the active process. As in the standard Debugger, the programmer is able to make changes to any of the methods in the active process. Starting the method restarts the new version.

The third and last scenario permits a method manufactured by the system to be executed. If successfully compiled, execution will then proceed as in the other two scenarios.

When stepping through a method using **Hop** and **Skip**, there are times when the programmer may wish to continue the execution of a method without interruption. In a standard Debugger, this can be done with the **Go** button. In an NDDP Browser, the **Go** button serves the same purpose. One should note that this is different from starting a method. **Go** will continue the execution from the last point a method was interrupted, whereas **Start Method** will execute the method from the beginning.

When debugging using a standard Debugger, all the messages sent but unanswered (the sender chain) can be seen in the Debugger's Walkback pane. To see a method in the sender chain using the NDDP Browser, the programmer can travel deeper into the chain, or come back, using the **In** and **Out** buttons respectively.

Since the NDDP Browser offers the functionality of both a Class Hierarchy Browser and a Debugger, a programmer should be able to temporarily browse some arbitrary method in the system without leaving the active process. Subsequently clicking on the **Home** button causes the Browser to go back to the last active method selected.

We added those features of the standard debugger to our NDDP Browser that we found most useful for our purposes. But to provide more flexibility to our environment, we allow the programmer to open a standard debugger on the process. To do so, programmer can click on the **Debug** button. In practice, however, we found little need for this feature.

Because Smalltalk supports inheritance, a class does not have to implement all the methods that it responds to; some are inherited from the superclasses. For example, we can introduce a class called *Transaction* and define a method **amount:** in the class to set the amount of a transaction. As a result, subclasses *DepositTransaction* and *WithdrawalTransaction* can inherit this method without actually implementing it. Now consider the scenario where a message **amount:** is sent to a *DepositTransaction* prior to its definition anywhere in the hierarchy. Then clearly **amount:** will not be understood by the deposit transaction. This will cause the NDDP browser to manufacture a method with that name in the class of the receiver; in the *DepositTransaction* class. The knowledgeable programmer will realize at this point that it should be higher up in the hierarchy. A set of

Hierarchy Navigation buttons were introduced for this purpose. They are illustrated in Figure 3.5.

The left arrow button or *Superclass* button will select the superclass of the currently selected class as the target for the newly manufactured method. Note that the superclass button points left to correspond to the superclass location in the "class list" pane; i.e., the superclass of *CheckingTransaction* is extended further to the left. The right arrow or *Subclass* button selects the first subclass of currently selected class. The other subclasses can be targeted with the *Next* and the *Previous Sibling* buttons. The middle button labeled *Find Class* will bring up the Find Class dialog permitting the programmer to select an arbitrary class.

Figure 3.6 Hierarchy Navigation Keys.

Note: clicking on a Hierarchy Navigation button does not change the manufactured method that is displayed in the text pane. However, it does change the selected class in the NDDP browser's class list pane, thus providing direct feedback to the programmer. Saving the method directly by clicking on **Save** or indirectly by clicking on **Start Method** will result in compiling the method in the selected class. If there is no compilation error, the method will be added to the system.

3.3 Need Driven Method and Method Header Construction

The gist of a system that supports **need-driven** design and programming is a facility that can generate method headers (and sometimes, entire methods) by need. When a message is not understood, substantial information is available to the NDDP browser:

- (1) the receiver (and consequently the receiver's class) - we can use this information to determine where to add the new method (as discussed in the previous section)
- (2) the arguments (and consequently, the class of each of the individual arguments) - we can use this information to manufacture appropriate argument names.

On this latter point, if three arguments (for example) are members of classes `String`, `BankAccount`, and `String` respectively, we can generate names `string1`, `aBankAccount`, and `string2` respectively. Argument classes unique to the list are preceded by "a" or "an"; non-unique ones have the class name decapitalized and suffixed by a unique integer starting at 1. Since this is the convention used deliberately by many programmers¹⁴, only the exceptional situations will require modifications by the programmer.

Of course, there is also a third piece of useful information that is available to the NDDP Browser:

- (3) the selector - we can use this to generate the correct method name.

But we can do more in certain cases. In particular, if the name happens to match the name of an instance variable, we can manufacture the corresponding `get` or `set` method¹⁵.

¹⁴ The great majority of the methods in the existing class library also follows this convention.

¹⁵ A *get* method returns the value of a variable. A *set* method assigns a value to a variable.

For example, assume that we have just created a *BankAccount* class with instance variable *owner* and no instance methods. If we execute the *egCreateAccount* class method, as shown in Figure 3.5, by clicking on the **Start Method** button, the NDDP Browser will suddenly change to the state shown in Figure 3.7. To inform the programmer that the method seen in the text pane is a suggested method (and still not part of the system), the background color is set to yellow. We provide a summary of coloring interpretations in the next section.

Figure 3.7 Result of Executing the *egCreateAccount* method.

As can be seen in Figure 3.7, the *NDDP Browser* has taken the initiative to write a complete set method for the *owner* instance variable. This is also based on the common convention followed by Smalltalk programmers. A get method for an owner instance variable is generally called *owner* (the same name) and a set method is called *owner:*.

As can be expected, if variable *owner* was not already defined, the NDDP Browser would only create the method header. The programmer would have to add the required code.

To summarize, once the NDDP Browser introduces a manufactured method (or method header), the programmer can accept the method or change it as appropriate. Also, by using Hierarchy Navigation buttons, the programmer can move the method to the class that he or she believes is a more appropriate container class. Saving the source code will result in the addition of the method to the selected class. So will clicking on the **Start Method** button which also has the side-effect of carrying on with the execution. As explained earlier, the **In** and **Out** buttons are also available for studying the context of the new method before adding new code.

3.4 Background Coloring

In the standard environment, browsers and debuggers have independent windows. Consequently, changes to a method in a debugger are not always reflected in the browsers¹⁶. A browser, for example, could be showing an old method that was just modified in a debugger. The converse is a scenario in which a method is changed in a browser but the old version is part of the sender chain in an active debugger.

¹⁶ Although this could be fixed if all browsers were forced to update after a method was saved anywhere in the system.

Consequently, there are times that the programmer is looking at two different versions of a method, and what he debugs may be different from the last saved version of the method. We can use color to inform the programmer if a method is the latest version.

In general, there are a number of useful time dependent properties of methods that can be discriminated with color. So far, we make use of four different background colors to highlight a method in a text pane:

- *Yellow* - The method was automatically created by the NDDP Browser, but not yet saved by the programmer.
- *Pale Gray* - The method is an active method (it's on the sender chain).
- *Red* - Same as above but it either does not match the latest version or it exists only in the process. (Both of these cases can occur by saving or deleting a method in another browser, for example).
- *White* - All other cases.

Once a programmer is familiar with the color interpretations, he can distinguish potentially ambiguous situations. For example, consider the case where an incorrectly designed recursive factorial was executed. For "10 factorial", we might have 10 active copies on the sender chain if it failed on the base case ("0 factorial"). All of the methods will be pale gray since they are active. If we change this method in another browser and try to see one of the methods in sender chain; e.g. by clicking on **Home** or **Out**, the method will be shown in red (it does not match with the last version of factorial). If we suddenly start to browse outside the sender chain: e.g., by clicking on class *Integer* and then clicking on factorial in the class, this version will be white.

3.5 Summary

In this chapter, we covered the most important aspects of an environment designed to support **need-driven** design and programming. In particular, we reviewed how the process could be supported by improving facilities for automatically declaring variables, by merging the notion of browsers and debuggers, and by providing facilities for automatically generating methods (and method headers). The result is a special designed browser and supporting framework that we called the NDDP Browser. We ended by describing how color could help distinguish different context in the unified browsers.

Chapter 4

An Example of Need Driven Design and Programming With Support

In this chapter, we will once again apply the **need-driven** methodology to developing the **BankAccount** class introduced in Chapter 2. However, we will do so with the aid of the NDDP browser introduced in Chapter 3. The aim is to provide the reader with an understanding of the utility of the tool. We will start at the beginning, using a base image that has none of the Banking classes.

4.1 The Bank Account Example

Figure 4.1 Code for creating an account.

In our extended environment, we often start the design and implementation process by collecting interesting use-cases in an appropriate class. In particular, we might begin by creating the most important classes needed (in this case *BankAccount*) and then add class methods that illustrate the use-cases. Sometimes, we are not even that well organized!. We might just begin by writing a simple use-case in a workspace. Let imagine such a scenario where we first wrote a use-case for creating an account. Figure 4.1 shows the results at a point where we are about to execute the code. Our goal is to inspect the result.

Define Global

Define : **BankAccount**

☒ **Class**

Superclass **Object**

Instance Variable Names :
owner accountNumber

Class Variable Names :

Pool Dictionaries :

Indexed
☐ Yes
☒ No

Contains
☒ Pointers
☐ Bytes

☐ **Class Variable**
In Class :

☐ **Global**
Value :

☐ **Constant in Pool**
Pool :
Value :

Figure 4.2 The system asks the programmer to define the *BankAccount* class.

Since the *BankAccount* class does not exist in the system, the system will respond with the **Global Variable Definition** dialog. This is illustrated in Figure 4.2. We decided to set the superclass of *BankAccount* to *Object* and also defined *owner* and *accountNumber* instance variables for the class. The *balance* instance variable was intentionally left to be defined later.

After defining the *BankAccount* class (which is partially defined at this moment), we can force execution to carry on by clicking on the **OK** button.

Figure 4.3 The NDDP browser automatically manufactures the *owner:* method for the *BankAccount* class.

The first missing method encountered is **owner:** which is sent to the *BankAccount* instance. As a result, the system will bring up an *NDDP Browser*, select the *BankAccount* class and manufacture the **owner:** method as shown in Figure 4.3.

Several observations can be made:

- Since the *BankAccount* class already has an instance variable called *owner*, the NDDP browser can generate the complete code for the **owner:** set method.
- Since the argument for the **owner:** method is a string, the browser can name the parameter of the manufactured method *aString*. If we wish to change the name, we can simply edit the source code and then continue the process.
- To inform us that the displayed method has been newly manufactured, the background color is set to yellow.

We can continue execution by clicking on the **Start Method** button. The NDDP browser will compile the **owner:** method and continue execution from the point that it was interrupted.

The next missing method is **accountNumber:**. The NDDP browser manufactures the method in an identical manner to the previous case. Once again we simply continue execution by clicking on the **Start Method** button.

The third missing method is **balance:**. So execution stops with the browser in the state shown in Figure 4.4. As we can see, the browser lacks the necessary information to complete the method. So, it only manufactures the method header. There is no instance variable called "balance" so it can't interpret the missing method as a set method. However

we know that this should be a set method and so we simply add the missing code "*balance* := *amount*."

Figure 4.4 The NDDP browser automatically manufactures the header for the *balance:* method.

Once more, we continue execution by clicking on the **Start Method** button. Since variable *balance* is not defined, the browser will respond by bringing up a dialog for defining local variables as shown in Figure 4.5. Because we want to declare *balance* as an instance variable, we simply select the **Instance Variable** button and then click on **OK** to continue execution.

Figure 4.5 The NDDP browser dialog for defining variable *balance*.

With the addition of the `balance:` method, the rest of our code can run to completion. Figure 4.6 shows the result of inspecting the `davesAccount` variable which is returned as the result of completing the execution.

Figure 4.6 Inspecting the result of the code we wrote in a workspace.

Now that we have created the *BankAccount* class, we can develop subsequent use-cases in class methods for this class. We can also add the code we just wrote as method `egCreateAccount`.

As our next use-case, we added the `egMakeDeposit` class method to the *BankAccount* class (see Figure 4.7). We attempt to compile and execute the method by clicking on the **Start Method** button. Since the *DepositTransaction* class is undefined at this moment the browser responds by bringing up a dialog for defining globals as shown in Figure 4.7.

Figure 4.7 The NDDP dialog for defining global *DepositTransaction*.

We can simply define the *DepositTransaction* class as a subclass of *Transaction* even though the *Transaction* class itself is not yet defined. As a result, the browser will prompt us with another **Global Variable Definition** dialog asking us to define *Transaction*. This is illustrated in Figure 4.8.

We decided to define the *Transaction* class as a subclass of *Object* with an *amount* instance variable. This will allow us to proceed further with the `egMakeDeposit` use-case. The first method manufactured by the browser in this case will be the `amount:` method shown in Figure 4.9

Figure 4.8 The NDDP dialog for defining global *Transaction*.

As we discussed in Chapter 2, a better place for the `amount:` method is the *Transaction* class (it's likely that all transactions will need an amount, not just deposit transactions). Consequently, two changes can be made to the manufactured method:

- We can move it to the *Transaction* class by clicking on the superclass button in the *Hierarchy Navigation Buttons* as shown in Figure 4.9.
- We can make the method code more general by changing parameter name *aFloat* to *aNumber*.

Figure 4.9 The NDDP browser manufactures the **amount:** method for the *DepositTransaction* class.

The resulting method is shown in Figure 4.10. We subsequently continue execution of our use-case by clicking on the **Start Method** button, which results in the compilation and addition of the **amount:** method to the *Transaction* class.

The next manufactured method is the **addTransaction:** method in the *BankAccount* class. The browser will manufacture the header for the method which we can complete as follows:

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

Figure 4.10 Defining the `amount:` method in the *Transaction* class.

Because *transactions* is not defined in the *BankAccount* class, the browser will require us to define it. We can simply define this variable as an instance variable of the *BankAccount* class. When we continue executing, the browser next manufactures the method `add:` and places it in the *UndefinedObject* class. To determine the reason for this, we can go one step back and find out where we sent message `add:` to an undefined object. We can do this by clicking on the Out button of the Control Panel.

Figure 4.11 Clicking on the *Out* button will show the previous method in the sender chain.

As shown in Figure 4.11, the last method executed before the `add:` message is the `addTransaction:` method in the *BankAccount* class. The last statement executed is highlighted in the text pane. As it can be seen, the `add:` method is sent to a *transactions* instance variable. To check that *transactions* contains the `nil` object, we can double-click on *self* (the current bank account) and observe the value of the instance variable *transactions*. This has also been illustrated in Figure 4.11.

Figure 4.12 Restarting the egMakeDeposit use-case.

As we mentioned in Chapter 2, the reason for this error is that we have not properly initialized the new *BankAccount* instance. We can remedy this by adding a new class method and an *initialize* instance method to both *BankAccount* and *Transaction* classes as described in Chapter 2. In this case, we have no choice but to reexecute the whole use-case again because the receiver; i.e. *davesAccount*, needs to be initialized. In general, each of the methods in the sender chain can be restarted. In this case, we can click on the **Out** button until we reach to the *egMakeDeposit* method. Then to restart this active method, we can click on the **Start Method** button. This is illustrated in Figure 4.12.

Figure 4.13 The balance method manufactured by the browser.

The next missing method manufactured by the browser is the **applyTo:** method in the *DepositTransaction* class. We can complete the code for this method as follows:

applyTo: *aBankAccount*

aBankAccount **balance:** (*aBankAccount* **balance** + self **amount**)

and continue execution. As can be seen, we used **balance** and **amount** methods which have not yet been defined.

Figure 4.13 illustrates the **balance** get method manufactured by the browser. Since the manufactured method does exactly what we want, we simply continue execution which will result in the manufacture of the **amount** get method in the *DepositTransaction* class.

As we did in the **amount:** set method, we decided to move the manufactured method to the *Transaction* class by using the *Hierarchy Navigation Buttons*. This is the last method that is needed to make our second use-case work. After finishing the use-case, the browser redisplay the **egMakeDeposit** class method in the *BankAccount* class to indicate that execution ran to completion.

We are now ready to execute our third use-case which contains the following code:

```
egMakeWithdrawal
  "BankAccount egMakeWithdrawal"
  | davesAccount |
  davesAccount := BankAccount new.
  davesAccount owner: 'Dave',
    accountNumber: 1234;
    balance: 1000.00.
  davesAccount addTransaction:
    (WithdrawalTransaction new amount: 580.00).
  ^davesAccount
```

When we do, the following interactions result:

1. The browser prompts us to define the *WithdrawalTransaction* class. We respond by defining it as a subclass of *Transaction*.
2. The browser creates a header for the **applyTo:** method in the *WithdrawalTransaction* class. We complete the method as follows:

```
applyTo: aBankAccount
  "Applies self to aBankAccount."
  aBankAccount balance: (aBankAccount balance - self amount)
```

3. We continue execution by clicking on the **Start Method** button.

Because there are no more missing methods, execution proceeds to the end and the browser re-displays the **egMakeWithdrawal** method of the *BankAccount* class.

For our last use-case, we decided to write a method **egWriteCheck** as shown below:

egWriteCheck

```
davesAccount := BankAccount new.  
davesAccount owner: 'Dave';  
    accountNumber: 1234;  
    balance: 1000.00.  
davesAccount addTransaction:  
    (CheckingTransaction new amount: 450.00;  
        checkNumber: 1;  
        date: (Date today subtractDays: 1)).  
^davesAccount
```

As soon as we click on the **Start Method** button, the browser prompts us to define variable *davesAccount* as shown in Figure 4.14. By selecting **Temporary** and clicking on **OK**, the browser modifies the source code of the **egWriteCheck** method before compiling and executing it. As a result, the following sequence of events occurs:

1. The browser prompts us to define the *CheckingTransaction* class as shown in Figure 4.15. We simply respond by defining it under the *WithdrawalTransaction* class with the instance variable *checkNumber*.
2. The system manufactures the **checkNumber:** set method. Since the code does what we need, we simply continue execution.
3. The system creates the header for the **date:** method. Because we want all our transactions to have a date (transactions could be sorted by date), we will move the manufactured method to the superclass, i.e., *Transaction*, using the *superclass* button of the *Hierarchy Navigation Buttons* and complete the code as follows:

```
date: aDate  
    "Sets the date of a transaction."  
    date := aDate.
```

Figure 4.14 The browser prompting for a definition of *davesAccount*.

4. As soon as we click on the Start Method button, the browser prompts us to define variable *date*. By selecting Instance Variable and clicking on OK, the browser recompiles the new definition of the *Transaction* class and continues execution of the use-case.
5. Execution subsequently continues all the way to the end because all required methods now exist.

As we mentioned in Chapter 2, adding a `printOn:` methods at this point does not requires any new methods. Presumably, we can continue in this manner to further evolve these application classes.

Figure 4.15 The browser prompts for a definition of *CheckingTransaction* after defining temporary variable *davesAccount*.

4.2 Summary

In this chapter, we tried to show how our new environment supports the need-driven methodology. We did not intend to show each and every step of the development process. But we did want to provide a flavor for the speed that object-oriented software could be

developed with the new browser. In the next chapter, we will look at the implementation issues.

Chapter 5

On the Design and Implementation of the NDDP Browser

This chapter will explain the design and implementation aspects of the *NDDP Browser*, whose features were described in chapter 3. As we discussed earlier, it is integrated with Smalltalk environment to support need driven design and programming. The system is developed in Smalltalk/V Windows 2.0.

5.1 The Overall Design

Reuse has been considered by many as one of the most important benefits of object-oriented systems. Programming environments such as the Smalltalk programming environment are well known for their highly reusable class library. With this in mind, we have tried to develop our system by reusing as many available classes as possible. The crucial point in implementing a system that supports need-driven programming is the ability to execute a program in a supervised manner so that the system (or programming environment) is able to intercept the execution of the program, allow modifications (e.g., adding new methods, changing the source code of methods), and continue the execution from the point that it was interrupted. The Smalltalk/V *Debugger* class already supports many of the features that we would like to incorporate in our system. As a result, we

decided to create a specialized NDDP debugger by subclassing the existing Debugger class.

By contrast, the Smalltalk/V Class Hierarchy Browser is the main application in the system that allows programmers to organize and develop their programs. Any new software development facility should at least provide the features that are already available in the Class Hierarchy Browser. So , we decided to create a specialized NDDP browser by subclassing the existing *ClassHierarchyBrowser* class.

Figure 5.1 The Class Hierarchy Diagram for (a) NDDPBrowser (b) NDDPDebugger.

So far, we have created two subclasses as shown in Figure 5.1. To connect the two, an NDDP Browser keeps an instance of an NDDP Debugger as one of its parts (an instance

variable) as shown in Figure 5.2. Correspondingly, the NDDP Debugger also knows about the NDDP Browser. In this way, we have an NDDP browser that:

1. Unifies the existing browsers and debuggers i.e. blurs the distinctions.
2. Makes extensive reuse of code from both the *ClassHierarchyBrowser* class and the *Debugger* class.
3. Minimizes user training time because the user interface has many similarities with the known browser and debugger facilities.

Figure 5.2 The collaboration Graph for an NDDPBrowser and its associated NDDPDebugger.

Although the NDDP Debugger inherits many of the duties that we need, we cannot hope to make changes to its behavior without knowing about one of the more important entities that it manipulates; namely, a process, an object that represents an executing program's thread of control. To be able to control browsing and debugging facilities, the NDDP browser must manipulate processes. As a consequence, it is very important to know the structure of a process and to know what services are provided by the *Process* class. In Appendix A, we summarize our view of process objects and associated stack frames whose representations we had to re-engineer on our own. Documentation was never provided for these objects since they were never intended for direct public manipulation. The Appendix is designed to answer some of the technical questions that might be raised by the reader but it is not necessary for understanding the implementation details presented

in this chapter. For the moment, it is sufficient to know that a process is an object that represent a sequence of computational steps in Smalltalk. To perform a computational step, objects send messages to other objects and wait for the result. A process is logically divided into *frames*. Each frame corresponds to a message in a stack of sent but unanswered messages. Each frame stores information such as the receiver object, instruction counter, compiled method corresponding to the message, arguments, and temporary variables in message.

5.2 The Detailed Design

This section describes the implementation details of the NDDP Browser. Although it is possible to save a method without intending to immediately execute it, this capability is not interesting from the perspective of this thesis because it is simply inherited. What is interesting is what we have to do when a method is created in a context where it is specifically expected to be executed. We will break down the description into two independent sections: compilation for execution initiation and compilation for execution continuation. Also, because of the extensive reuse made of the system, in each of the following subsections, the original behavior of the system will be described first, then the necessary changes and additions made to provide the new functionality will be discussed.

5.2.1 Compilation for Execution Initiation

Generally, when a programmer saves the changes made to a text pane, method `accept` of the *TextPane* class is executed. As part of the execution of this method a *save* event is generated. The owner of the text pane (in our case the NDDP browser), can perform

special "accept" processing by defining an event handler method for this event. In Smalltalk/V, when a text pane is used to display source code for a method, (e.g., in a class hierarchy browser or a method browser), the event handler for the *save* event as part of its job compiles the source code. Compilation of source code also occurs when a user evaluates a fragment of Smalltalk code; for instance, when the programmer uses the **Do It**, **Show It**, or **Inspect It** menu items. However, in these cases, compilation is done as part of the evaluation process. Since Smalltalk/V's compiler treats undefined local and global variables differently, we will discuss each case in a separate subsection.

In essence, we handle both cases by rewriting a method that gets invoked as a result of the error. Depending on the circumstances, there may (or may not) be changes to the source code. In any case, we could subsequently restart the compilation process from the beginning. However, we don't do that. Instead, we resume the execution of the active stack frame at such a point that recompilation results. This approach unifies what we do in this section with the approach taken for execution continuation discussed later.

5.2.1.1 Undefined Local Variables

If compilation is not successful, the general approach used in this version of Smalltalk/V is to notify the text pane of the error. More specifically, it sends the following message

compilerError: *aString* at: *anInteger* in: *codeString* for: *class*

to the pane that should be notified. The first argument of this message, *aString*, is the error message. The default behavior is to insert the error message at position *anInteger* in source code *codeString* and then to highlight it using reverse video. As mentioned before,

the case for undefined variables that begin with an uppercase letter is handled differently. We will discuss this in the next part.

Of special interest to us is the '**undefined**' error message. This is the error message generated when an identifier beginning with a lowercase letter is used as a variable but is not defined as a local variable. In this thesis, both temporary variables and instance variables have been referred to as local variables to differentiate them from global variables.

Generally there are two reasons for an identifier to be considered an undefined local variable. Either it is a typographical error, or the variable has truly not been defined yet. In the first case, correction is matter of replacing the identifier with a corrected identifier and recompiling. On the other hand, the situation is more complex when the identifier is correctly used in a situation where it has not yet been defined¹⁷. In this case, the local identifier can be either a temporary variable, or when compiling a method, it can be an instance variable of the receiver and consequently might be defined in the receiver's class or any of its superclasses.

If the programmer decides to define a local identifier as a temporary variable, this will require changes to the source code by creating an appropriate temporary variable definition, and recompiling the source. But, if programmer wants to define the local variable as an instance variable, then the source code for the method will not change, but the definition of the class containing this instance variable must be updated and

¹⁷ This may sound contradictory. When we say correctly used, we mean that the programmer intends to use the variable as a deliberate part of his design.

recompiled. After this has been done, recompilation of the method will result in the correct binding of the reference to the variable.

Even with Smalltalk's powerful programming environment, doing the above manually can be very cumbersome for the programmer. There are at least two benefits in introducing semi-automatic application of the necessary modifications. First, it is faster and obviously much more convenient for the programmer because he will be dealing with a specialized user interface that requires less interaction with system. Second, letting the programmer apply these decisions easily and as late as possible is essential for need driven programming.

To achieve these goals, method **compilerError:at:in:for:** has been modified to pass control to the **Define Local Variable** dialog, along with the process that caused the initiation of the error¹⁸. After the programmer decides on the most appropriate definition, as described in Chapter 3, he clicks on the OK button. Depending on the programmer's request, the handler for OK will apply the necessary changes mentioned above to the original text, or in the case of defining an instance variable, to the desired class.

The next step is recompiling or reevaluating the source code. To do this, the appropriate frame in the process that initiated the error must be located, and the process restarted at that frame. A suitable frame for the case of compiling a method is the frame that belongs to method **accept** in class *TextPane*. When the compilation error is initiated during evaluation of an expression, the correct restart frame belongs to method **doIt:** in class *TextPane*. After locating the frame, all the extra frames that come after it must be dropped

¹⁸ If the identifier to be defined begins with an upper case letter the Global Variable Definition dialog is used instead.

from the process stack and all temporary variables in frame must be set to **nil**. Since the value of the instruction counter has been incremented during the execution of the method, it must be reset to the appropriate initial value. For the current Smalltalk/V implementation, this value is equal to 20¹⁹.

At this stage, the process is ready to resume. After closing the dialog, the cleaned up process can be scheduled by **Processor** which is the single instance of the *ProcessScheduler* class which is responsible for scheduling ready processes. As a result another attempt will be made to compile or evaluate the source. If again there is another undefined local variable, the same sequence will be repeated. As mentioned earlier, if the programmer decides to abort the compilation process he can click on the **Cancel** button of **Define Local Variable** dialog. This will result in the appearance of an 'undefined' error message in front of the variable that caused the error.

3.2.1.2 Undefined Global Variables

If an undefined identifier begins with an uppercase letter, Smalltalk's default behavior is to give the programmer the chance to define it as a **global** variable. What is referred to as a global variable in this case is the special case of a variable stored in the *Smalltalk System Dictionary*²⁰. If the user responds yes to the dialog box that results, the compiler adds the undefined identifier to the system dictionary. The dialog supports no other possibility. Additionally, since the added variable is by default initialized to **nil** it most probably cannot

¹⁹ Smalltalk/V 2.0 for Windows.

²⁰ In Smalltalk, global variable *Smalltalk* is the single instance of class *SystemDictionary*. A System Dictionary contains all of the names of the classes, global variables and pool dictionaries in the system [Digitalk 92a, pp. 271].

be used in the evaluation of an expression, and the programmer will have to initialize it directly through an assignment statement.

The reality is that an identifier that begins with an uppercase letter can also be a class, a class variable, a global variable, or a constant in a pool. The additional possibilities added as part of this thesis allows the programmer to define the identifier as any of these different possibilities, with a choice of initializing value if applicable. In chapter 3, the **Global Variable Definition** dialog box was described from the user's point of view. This section deals with the implementation aspects of this feature.

In Smalltalk/V, when the compiler encounters an undefined identifier that begins with an uppercase letter, message **defineGlobal:** is sent by the compiler to the Smalltalk System Dictionary. This method is defined as:

defineGlobal: *aSymbol*

"Private - The compiler performs this method when it finds an undefined global. Answer true if name should be a new global. Answer false if name should remain undefined."

^MessageBox confirm: 'Define ', *aSymbol* **printString,**
' as global variable?'

If this method returns **true**, the compiler adds *aSymbol* to the System Dictionary and continues compilation. If it returns **false**, the compiler displays the '**undefined**' error message in front of the identifier and stops compilation. To allow the programmer to selectively define the undefined identifier in a more controlled manner, we changed this method to always return **false**. This forces the compiler to send **compilerError:at:in:for:** message to the pane in which compilation was initiated. At this point, we can determine that the undefined variable is a global and invoke the **Define Global Variable** dialog.

As in the case of defining a local variable, the copy of the Process that is currently executing will be passed to the dialog. The dialog itself will be working under a new running process. When the user clicks on the **Apply** button, depending on the choice made, appropriate validation tests will be performed. If the programmer's request is valid, then it will be performed. In the case of defining a class or class variable, this will require the generation or modification of a class definition and compilation of the new definition. If the request is to define a global variable, or a variable in a pool, appropriate changes must be made to the Smalltalk System Dictionary. If the programmer changes the name of the global variable in the dialog, the source must be updated accordingly.

Defining the variable will allow compilation to proceed. Instead of forcing the programmer to recompile or reevaluate the source, the interrupted process that was passed to the dialog can be restarted. This is done when the programmer closes the dialog. At this time, the appropriate process frame to restart will be located and initialized for resumption. As is the case for defining local variables, if the attempt to define a global variable was made during the evaluation of an expression, the relevant frame belongs to message **doIt:**. Otherwise, it belongs to message **accept**. As mentioned earlier both methods belong to the *TextPane* class.

The preparation of the interrupted process to resume will be performed in the same way that it was done for local variables. Again, the instruction counter will be set to 20. Now that the process is ready to resume, the Process Scheduler can schedule it and as a result, the process will start from one of the above messages. If the compiler finds another capitalized undefined variable, the above scenario is repeated. If the programmer decides

not to define the global variable, he can stop the process by clicking on the **Cancel** button in the **Define Global** dialog.

5.2.2 Compilation for Execution Continuation

The power of the NDDP browser lies in the ability to allow the programmer to define missing methods as they are encountered²¹ during execution while letting him continue from that point once the method is completed. In fact, he can restart, anywhere along the senders chain but that's just a minor variation of the basic capability. What makes it easy to use is the fact that the system will automatically determine the appropriate target class for the missing method, whether or not it is an instance or class method, and create a suitable header for the method. Of course, the programmer is not forced to accept the suggestion.

5.2.2.1 Default behavior when an object does not understand a message

Before describing the details of Compilation for Execution Continuation, it would be useful to present a short overview of the standard behavior of the system when a message is sent to an object that does not have a corresponding method.

When a Smalltalk message is sent to an object, Smalltalk's Virtual Machine (VM) tries to locate the appropriate method in the receiver and create a frame for it²². If found, this

²¹ Of course, we can't encounter a missing method but we can detect the fact that it is missing and substitute at least the header in the appropriate context.

²² All the messages that an object can respond to is defined in the object's class. In Smalltalk/V every class is an instance of a metaclass of the same name. The class contains the instance methods while metaclasses contains the class methods [Digitalk 92b, pp. 28]. *Behavior* which is the super class of *Class* includes an instance variable named **dictionaryArray**. This is an array of method dictionaries associated with the class and its super classes. The system searches this dictionary to locate the method corresponding to a message.

frame becomes frame 0 of the current process, and execution continues by executing the associated code for that method. If not found, the VM creates a frame for a **doesNotUnderstand:** message directed at the same receiver. Additionally, it provides one parameter, an instance of class *Message* which contains the receiver object, selector, and the arguments of the message that was not understood. There is one default **doesNotUnderstand:** method which is defined in class *Object*²³. Since almost every class in hierarchy is a direct or indirect subclass of *Object*²⁴, this is the default behavior for all these classes. The code in method **doesNotUnderstand:** defined in class *Object* sends message **error:** to the receiver object with a string as the parameter which explains the error. Currently, method **error:** is defined only in class *Object*. The following is the code for this method:

error: aString

"Create a walkback window describing an error condition
with the error message aString in the window label."

Cursor reset.

Process

queueWalkback: aString

makeUserIF: CurrentProcess isUserIF

resumable: false

As we can see, this method in turn sends **queueWalkback:makeUserIF:resumable:** to the *Process* class with the error string from **error:** along with two other boolean arguments that indicate if the current process is a User Interface process²⁵ and if the currently

²³ Classes *SelfInitializingObject*, *SelfDefinedStructure*, and *FloatCoprocesor* have their own version of **doesNotUnderstand:**.

²⁴ *SelfInitializingObject* is an exception. It is a subclass of *nil*.

²⁵ The Smalltalk/V user interface is driven by a single process which responds to all keyboard and mouse events for all windows.

executing process can be resumed, which in this case have been set to **false** because it is assumed that when an error occurs the process should not be resumed.

The **queueWalkback:makeUserIF:resumable:** method (code not shown) in turn creates a new process, assigns it to global variable *CurrentProcess*, and places a special message in the *PendingEvents* queue. This message has selector **errorIn:label:** and two arguments: the copy of currently running process and the error string that was passed as argument.

The user interface process polls the *PendingEvents* queue when there is no input activity. This is done in the **nullEventWin** method of the *InputEvent* class where a message is removed from the *PendingEvents* queue. This message is executed using **perform:withArguments:** as follows:

```
self perform: message selector withArguments: message arguments
```

As we might expect, there is an **errorIn:label:** method in the *InputEvent* class. It creates a new debugger and initiates a walkback window. If the programmer wants to debug the process, he can proceed by choosing the Debug menu item or by clicking on the Debug button.

5.2.2.2 New behavior when an object does not understand a message

To be able to activate the appropriate NDDP Browser when an object does not respond to a message, the code for **doesNotUnderstand:** must be altered. If the programmer has decided to activate the NDDP facility, method **doesNotUnderstand:** executes **controlledError:forMessage:** instead of executing **error:**. This results in the execution of the following code:

ControlledProcess

```
queueWalkback: aString  
makeUserIF: (CurrentProcess isUserIF)  
resumable: false  
message: aMessage
```

As a rule we have avoided adding methods to original classes as long as it was possible to provide the same functionality by subclassing the original classes. *ControlledProcess* is a subclass of *Process* created specifically for the additional behavior required to implement the capability of continuing execution after adding methods.

The first three arguments of `queueWalkback:makeUserIF:resumable:message:` have the same meaning and usage as the original `queueWalkback:makeUserIF:resumable:` method. To be able to locate the target class and create a correct method header, the message that triggered `doesNotUnderstand:` in the first place is passed as the fourth argument. Let's refer to the message that was not understood as the faulty message for the rest of this chapter.

Most of the code in instance method `queueWalkback:makeUserIF:resumable:message:` is the same as its counterpart in the *Process* class but since we do not want to activate the default system Debugger, it is necessary to introduce our own version of `errorIn:Label:`. This new method has been called `controlledError:label:forMessage:` (an instance method of the *InputEvent* class), and is placed in the *PendingEvents* queue instead of the `errorIn:Label:` message. The arguments of this message will be the copy of the currently executing process, the error string, and the faulty message. This is the method that will create and initiate an NDDP debugger.

The *NDDPDebugger* has all the functionality of a standard debugger, and in addition, can operate in a silent mode which does not involve any user interface. In addition, it is associated with a corresponding NDDP browser.

The **controlledError:label:forMessage:** method sends the following message to the *NDDPDebugger* that it first creates:

walkbackFor: aProcess label: aString message: aMessage

Besides setting up internal debugger variables, this method performs two important tasks. First, it drops all the extra frames that were created after the **doesNotUnderstand:** message. Second, it establishes the link between the NDDP Debugger and the corresponding NDDP Browser (a new browser will be created if necessary). In the next step, it must cause the NDDP Browser to select the correct class, invisibly select the appropriate instance or class switch, and create the appropriate method header for the faulty message. This behavior is provided in the *NDDPBrowser* class.

At this stage, there is an *NDDPDebugger* holding the interrupted process, and a corresponding *NDDPBrowser* selecting a suitable class and method for the faulty message. What happens next, depends on the user's interaction with the NDDP Browser. The following sections are devoted to the implementation details of the responses to user requests.

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

UTER SCIENCE

SUBJECT TERM

0484

SUBJECT CODE

U·M·I

Categories

HUMANITIES AND SOCIAL SCIENCES

HUMANITIES AND THE ARTS

0729
0377
0900
0378
0357
0723
0391
0399
0708
0413
0459
0465

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

0515
0514
0516
0517
0273
0282
0688
0275
0727
0518
0524
0277
0519
0680
0745
0520
0278
0521
0279
0280
0522
0998
0523

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

PHILOSOPHY, RELIGION AND THEOLOGY

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

SOCIAL SCIENCES

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

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science 0615
 General 0616
 International Law and 0617
 Relations 0616
 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 0630
 Studies 0700
 Industrial and Labor 0629
 Relations 0630
 Public and Social Welfare 0700
 Social Structure and 0705
 Development 0999
 Theory and Methods 0453
Transportation 0453
Urban and Regional Planning 0453
Women's Studies 0453

SCIENCES AND ENGINEERING

SCIENCES

0473
0285
0475
0476
0359
0478
0479
0480
0817
0777
0746
0306
0287
0308
0309
0379
0329
0353
0369
0793
0410
0307
0317
0416
0433
0821
0778
0472
0786
0760
0425
0996

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

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 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 0569
 Nursing 0570
 Nutrition 0380
 Obstetrics and Gynecology 0354
 Occupational Health and 0381
 Therapy 0571
 Ophthalmology 0419
 Pathology 0572
 Pharmacology 0382
 Pharmacy 0573
 Physical Therapy 0574
 Public Health 0574
 Radiology 0575
 Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences 0485
Chemistry 0749
 General 0496
 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 0607
 Astrophysics 0798
 Atmospheric Science 0759
 Atomic 0609
 Electronics and Electricity 0610
 Elementary Particles and 0752
 High Energy 0756
 Fluid and Plasma 0611
 Molecular 0463
 Nuclear 0346
 Optics 0984
 Radiation 0984
 Solid State 0984
Statistics 0984
Applied Sciences 0346
Applied Mechanics 0984
Computer Science 0984

Engineering 0537
 General 0538
 Aerospace 0539
 Agricultural 0540
 Automotive 0541
 Biomedical 0542
 Chemical 0543
 Civil 0544
 Electronics and Electrical 0545
 Heat and Thermodynamics 0546
 Hydraulic 0547
 Industrial 0548
 Marine 0549
 Materials Science 0799
 Mechanical 0541
 Metallurgy 0747
 Mining 0555
 Nuclear 0555
 Packaging 0547
 Petroleum 0766
 Sanitary and Municipal 0555
 System Science 0799
Geotechnology 0422
Operations Research 0799
Plastics Technology 0799
Textile Technology 0999

PSYCHOLOGY

General 062
Behavioral 038
Clinical 062
Developmental 062
Experimental 062
Industrial 062
Personality 062
Physiological 098
Psychobiology 034
Psychometrics 063
Social 045

2 of /de 2

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

PRECISIONSM RESOLUTION TARGETS

5.2.2.3 Starting a Method

At the implementation level, this is probably the most complex request that a user can make because the response depends on the context of the selected method. From the user's perspective, however, only one thing can happen. A method will be executed.

The reason for the complexity from the implementation's point of view is that four different contexts are possible:

1. The Starting method is a class method and it is not part of the process sender chain.
2. The starting method is a method that was missing and now it has been added to the system.
3. The starting method is one of the methods in the process sender chain (we are restarting a previously executed method) and such a method exists (i.e., has not been deleted after execution was halted).
4. The starting method only exists in the process sender chain.

Recall that if the method text in a text pane has been changed, it will be first compiled and then depending on which one of the above condition holds, the appropriate action will be taken.

For case 1, the method should have no parameters. This allows the NDDP browser to execute it by simply sending the selector of the method to the class as shown below:

```
SelectedClass selectorMethod.
```

```
self unwindProcess.
```

Message `unwindProcess` is needed to finalize code that executes to completion. It will result in the reselection of the starting method.

For case 2, the programmer must have finished modifying the template provided by the NDDP browser and he wants to continue with the new method. Since we already dropped the extra frames after `doesNotUnderstand:`, the current process consists of a sequence of frames leading to this message. The frame immediately prior to this frame belongs to the sender of the faulty message. If the method had not been missing, the VM would have created a frame for it instead of a `doesNotUnderstand:` frame. There are several different approaches that can be used to create the illusion of starting a method that is not part of the stack. We found that the best and cleanest way of implementing it is to replace the whole `doesNotUnderstand:` frame with the frame that the VM would create if the method existed. Of course this could not be done without knowing the frame structure which is not documented. By studying the code for manipulating processes in different classes such as the *Debugger* and *Process* class, and also by inspecting real processes, we re-engineered the frame structure which is documented in Appendix A. By knowing the frame structure for different kinds of methods (there are a few variants), it is possible to construct a frame for the new method after compiling it. After replacing the `doesNotUnderstand:` frame with the newly constructed frame, we simply restart the process from this point as if it was already part of the process sender chain.

For case 3, the corresponding frame in the process stack is located and all the extra frames after it are dropped. The temporary variables of the method in the frame must be set to `nil` and program counter reset to the correct value for restarting the method.

Case 4, is identical, from the implementation perspective, to case 3. But there is one important consequence that is different. Since the method exists only in the process stack, there is no guarantee that the programmer will be able to access it again after starting it. If the method ends, the system will automatically remove it from the process. That is why the programmer is warned about this special case using a red background color.

5.2.2.4 Hop, Skip, Go and Debug

Two of the most useful features of the standard Smalltalk Debugger are its capabilities to perform **Hops** and **Skips**. **Hop** executes the next expression in the process being debugged and may follow the thread of execution to other methods. **Skip** executes the next expression in the selected method, but returns to the current method. As a consequence, it may execute an arbitrary number of expressions in lower-level methods. We found it useful to incorporate these features in the NDDP Browser. So a programmer is able to start a class method either in the **Hop** mode or the **Skip** mode.

To do the same thing in the standard environment, the programmer usually adds a '**self halt.**' statement at the desired start point and executes code that leads to the **halt**. After opening a Debugger window that results, he needs to click on **Hop** or **Skip** button to jump over the extra **halt** method and start walking through the desired method.

To create an initial process for the **Hop/Skip** mode, code like the following needs to be evaluated:

```
ControlledProcess
    queueWalkbackHopping: "
    makeUserIF: (CurrentProcess isUserIF)
    resumable: true.
SelectedClass selectedMethod.
self unwindProcess.
```

The only difference between method **queueWalkbackHopping:makeUserIF:resumable:** and method **queueWalkback:makeUserIF:resumable:** in the *Process* class which was discussed earlier is that the former queues **controlledHop:label:** instead of **error:Label:**. Method **controlledHop:label:** is defined as an instance method of *InputEvent*. It creates a new debugger and updates the process by silently skipping expressions until the next **Hop/Skip** stop point is reached. Later **Hop/Skip** requests by the programmer will cause the NDDP Browser to send the corresponding messages to its NDDP debugger. The *NDDPDebugger* will perform the same task as the standard *Debugger*, but it knows that in its silent operation mode, instead of updating its own pane, it should inform its browser to enable it to update its own text pane.

The **Go** button is equivalent to **Jump** in the standard Debugger. It allows execution to the end. The real task of setting the process state and resuming it is performed by an NDDP debugger upon receipt of request from the NDDP browser.

As we mentioned earlier, in most cases, our debugger is operating in the silent mode. To provide more user flexibility, we allow the programmer to open a standard debugger on the process by clicking on the **Debug** button. If the programmer later closes the debugger window, the debugger will switch to silent mode.

5.2.2.5 The Walkback Buttons

Quite frequently during the execution of code that leads to the activation of an NDDP Browser, the programmer may wish to see an arbitrary method in the process's sender chain. In many situations, fixing a bug requires a change in these methods.

The **In** and **Out** buttons allow the programmer to browse through the sender chain. To implement this feature, the NDDP browser maintains an index to the active process' frames. When the programmer clicks on the **In** and **Out** buttons, the NDDP browser updates the index and queries the NDDP debugger for the information needed to show the method for that frame. The special case of clicking **In** on the last frame of the active process (to go forward along the sender chain) is handled by the NDDP browser by creating the header for the faulty method or by showing the finalized version if the programmer has already saved it.

5.3 Summary

This chapter provided a detailed description of the design and implementation aspects of the NDDP Browser and its complementary NDDP debugger. We emphasized both those implementation aspects associated with execution initiation and execution continuation. The former was concerned primarily with facilities for semi-automatic declaration of variables and the latter with the incremental addition of missing methods. Both discussed techniques for setting up stack frames and resuming execution.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we introduced an approach to designing and programming that we called *Need Driven Design and Programming* (NDDP). Although the method has been practiced individually by some expert object oriented programmers, particularly in the Smalltalk community, it has received little attention in the software engineering literature.

We divided the NDDP process into two basic steps:

- Writing scenarios (use-cases) that capture the requirements of the application.
- Incrementally adding the required methods to make these use-cases work.

We argued that this approach is natural for object oriented software development, because it provides a disciplined approach to incremental software development. By applying the need-driven technique, we are seeking a closer match between the design and implementation. Additionally, the technique strives to generate fewer extraneous methods, because all methods are added as and when needed. Because of the interactive nature of the process, which is only partially supported by existing software development environments, we created a prototype of an environment to support NDDP. It is our belief

that better tools can be provided only if they are specifically designed to facilitate the application of planned techniques or a specific methodology. Our prototype was implemented as an extension to Smalltalk/V.

In Chapter 1, we presented the background information and the motivation for this thesis. In Chapter 2, we described the details of the NDDP process, and then applied it to the design of a bank account example. Chapter 3 presented the features and the user interface of a prototype that we developed to support NDDP in Smalltalk/V. In Chapter 4, the above prototype environment was used to redevelop the bank account example. We illustrated most of the important features of the new environment. Finally in Chapter 5, we presented the design of our new programming environment, and our approach to integrating the NDDP support features with the Smalltalk/V environment.

6.2 A Discussion of NDDP and Its Support Environment

We introduced the NDDP approach to designing and programming in Chapter 2. Here we present some of the conclusions that we reached as a result of our experience using the NDDP methodology and the supporting tools.

- NDDP by its very nature is an incremental approach. It provides a disciplined framework for making use-cases work.
- To make use of NDDP, the designer's view of the world must be captured by use-cases. The better the use-cases, the better the resulting applications. As a result a more experienced designer will come up with a better design and implementation.

- NDDP does not claim to create perfect implementations, but it does allow the developers to generate a first version very fast if the appropriate support tool exist — a version that satisfies the requirement of the use-case.
- The NDDP process is highly interactive. This does not mean that it cannot be used without computer assistance, but it can be better exploited if it is supported by the programming environment.
- The NDDP process is particularly useful for rapid prototyping, exploratory programming, and small project development. We are not making any claim for its utility in developing large and complex systems although, it may be used in the implementation of components of the system, once a relatively clear understanding of the system has been achieved and a higher level design is available.
- Since NDDP favors incremental development, it is best used in an environment that supports incremental compilation; more explicitly, one that can be extended to support the facilities for our own NDDP browser (the language itself is secondary).
- Unifying the Class Hierarchy Browser and the Debugger in our new browser has increased development speed. One reason is that it eliminates tedious context switches. Another reason is that it helps keep the programmer focused on the development process.

Finally, as Meyer states, the introduction of a new methodological principle inevitably meets a well known three-step sequence: (1) "it's trivial"; (2) "besides, it won't work"; (3)

"anyway, that's how I did it all along" [Meyer 88]. Of course we do not see our work as a completely new methodological principle, but if the reader has labored through all the previous pages, we hope he or she has reached step 3 by this time.

6.3 Future Work

A very important idea behind this thesis is that we can provide designers and programmers with a method and an environment that fits the development process being used. The effect of the programming environment on the implementation process cannot be ignored. While there is no doubt that research should be conducted to find better analysis and design methods, and perhaps newer paradigms, we must also pay more attention to the implementation phase. As long as there is an activity called programming (in its current connotation) which is part of the software development process, the need for sophisticated programming environments will exist.

Integrated environments with general purpose facilities such as browsers, editors, inspectors and debugger are definitely useful and should be considered a minimum requirement for today's software development environments. But we believe there is more that a programming environment can provide for programmers. This calls for better understanding of common scenarios of programming activity. We believe once one understands these patterns, one can propose methods and build environments that support a more productive software development process. NDDP and its support environment is just a step in this direction.

We should emphasize that such support, which can be seen as specialization of an environment, should not completely replace the more general purpose facilities. We do not think that environments should totally restrict the programmer to just one specific style of interaction. Our new environment helps the programmer adopt the need-driven approach but doesn't force him to. Of course, the existence of these new facilities will tend to make the programmers become more need-driven. How much influence does an environment have on its users? Clearly, there must be an important influence. Research is needed to further quantify that influence.

So far, we have not yet exposed the NDDP browser to large communities of programmers. We might expect valuable feedback and perhaps suggestions for even more useful extensions.

As a last point, much of the code in our NDDP browser dealing with need-driven variable declarations and automatic generation of methods could (and should) be integrated by the Smalltalk vendors with the *Compiler* class and the virtual machine. We, however, could not do this because the source code for these components is not publicly available.

Appendix A

The Smalltalk/V Frame and Process Structure¹

This Appendix briefly describes the Smalltalk/V frame and processes structure. The details have been re-engineered from the existing code and behavior of the process and debugger classes. A *Process* is defined as :

" ... an object representing a sequence of Smalltalk computations. The computations are performed by objects sending messages to other objects and waiting for the results. The process describes the current execution state, including the stack of unanswered messages." [Digitalk 92b pp. 180]

Physically, class *Process* is a subclass of *OrderedCollection*, and instance variable *contents* contains the frame data. There is a frame for every unanswered message in the process and a corresponding compiled method for every frame.

Frames are numbered beginning from 0 where frame 0 represents the last partially executed method in the call stack (the top of the stack), frame 1 represents the second last method, etc. A process can be viewed as an ordered collection of pointers which have been divided into unequal sized segments called frames (see Figure A.1). Thus a frame is simply a subsequence within the larger sequence; i.e. they are not logically distinct objects.

²⁶ All the references to Smalltalk or Smalltalk/V in this Appendix are to Smalltalk/V 2.0 for Windows, unless stated otherwise.

Figure A.1 A process and its frames.

Frame size varies from method to method and depends on:

- . The number of method parameters
- . The number of temporary variables in the method
- . Whether or not the code in the method has a home context²⁷.

Temporary objects created during execution of a method are placed above frame 0. When a new frame is created, it is placed on top of the stack to become the new frame 0. The simplest form of a frame has neither a home context, parameters, or temporary variables. Figure A.2 shows the structure of the frame when the method have parameters and/or temporary variables.

²⁷ A block executed by sending it a **value** message must execute in the context it was defined in; i.e., its home context.

Offset	Content
0	Next Frame Entry
1	Compiled Method Byte Code Array
2	Compiled Method
3	IC
4	Receiver
5 . . .	Arguments followed by Temporaries in Reverse order of definition

Figure A.2 A sample frame for a method with no blocks.

In Figure A.2 (and also A.3 to be considered next), offsets are numbered to match the terminology used in the *Process* class. The First entry of a frame begins at offset 0 of the frame, the next at offset 1, and so on. The next frame is one below the current frame, the previous frame is one above (in the context of frame 10, the next frame is 11 and the previous frame is 9).

Process frames are logically connected to each other as a linked list. The first entry or *Next Frame Entry*, acts as a pointer to the next frame down the chain (the caller). *Compiled Method Byte Code Array* is byte code array for the compiled method associated with the frame²⁸. The third entry in the frame is the compiled method itself. The instruction counter used internally by the Smalltalk virtual machine is stored in the fourth entry of the frame. Every frame stores the instruction counter for the frame below. It can be viewed as the return point for the caller. The instruction counter for the currently

²⁸ Every *CompiledMethod* has an instance variable called *byteCodeArray* which contains a *ByteArray* of interpreter code.

executing method is stored as the first element of the process stack. Finally the arguments and temporaries are stored independently in the reverse order of definition.

For the following method:

method: *parameter1* **with:** *parameter2*

| temporary1 temporary2 temporary3 |

"The rest of the code for the method. It does not include blocks."

...

The associated stack frame arguments and temporaries are stored in the order:

parameter2, *parameter1*, *temporary3*, *temporary2*, *temporary1*

Offset	Content
0	Next Frame Entry
1	Compiled Method Byte Code Array
2	Compiled Method
3	IC
4	Receiver
5	Home Context
6 . .	Arguments, begining from the rightmost one

Figure A.3 A sample frame for a method with blocks.

The structure of the frame will be different if the method has a block. In this case (see Figure A.3), what comes immediately after the receiver is an instance of class

HomeContext. Having a home context in the frame, temporary variables in the method are not stored in the frame. Instead, they are stored in the *HomeContext* along with block variables, which can be viewed as compiler generated temporary variables. Method arguments are stored after the *HomeContext* in reverse order²⁹.

²⁹ A copy of the arguments is also kept in the home context.

References

- [Boehm 86] Boehm, B. *A Spiral Model of Software Development and Enhancement*, Software Engineering Notes, Vol. 11(4), pp. 23.
- [Booch 91] Booch, Grady *Object-Oriented Design with Applications*, 1991 The Benjamin/Cummings publishing company Inc.
- [Brooks 87] Brooks, F. *No Silver Bullet: Essence and Accidents of Software Engineering*. IEEE Computer, Vol. 20(4)
- [Borland 93] *Borland C++ Ver. 4.0, Users Guide*, 1993 Borland Inc.
- [Digital 94] *COHESIONworX Solution Version 2.0 for Sun*, SPD 50.64.01. February 1994, Digital Inc.
- [Digitalk 92a] *Smalltalk/V for Windows. Tutorial and Programming HandBook*. 1992 Digitalk Inc.
- [Digitalk 92b] *Smalltalk/V for Windows. Encyclopedia of Classes*. 1992 Digitalk Inc.
- [Fowler 92] Fowler, Martin *A comparison of Object-Oriented Analysis and Design*, 1992 Object World.
- [Goldberg et. al. 83] Goldberg A.; Robson D. *Smalltalk-80. The Language and its Implementation*, 1983 Reading MA: Addison Wesley.
- [Jacobson 93] Jacobson, Ivar *Object-Oriented Software Engineering, A Use case Driven Approach*. 1993 Addison-Wesley
- [Kamath et. al. 93] Kamath, Y.H.; Smilan, R.E.; Smith, J.G. *Reaping Benefits with Object-Oriented Technology*, AT&T Technical Journal, Vol. 72(5) pp. 14-24, Sept.-Oct. 1993.

- [Lewis et. al. 91] Lewis J.; Henry S.; Kafura D.; Schulman R. *An Empirical Study of the Object-Oriented Paradigm and Software Reuse*. OOPSLA'91, pp. 184-196.
- [Lalonde 94] Lalonde, Wilf *Discovering Smalltalk*. 1994 Benjamin Cummings
- [Martin et. al. 95] Martin, James; Odell James J. *Object-Oriented Methods: A Foundation*. 1995 Prentice-Hall Inc.
- [Meyer 88] Meyer, Bertrand *Object Oriented Software Construction*. 1988 Prentice-Hall Inc.
- [Monarchi et. al. 92] Monarchi, David E.; Puh, Gretchen I. *A Research Typology for Object-Oriented Analysis and Design*, CACM September 1992/Vol. 35, No. 9.
- [ParcPlace 92] *Objectworks\Smalltalk Release 4.1, Users Guide*. 1992 ParcPlace Systems, Inc.
- [Saunders 88] Saunders, J. *A Survey of Object-Oriented Programming Languages*, Journal of Object-Oriented Programming, Vol. 1(6).
- [Selic et. al. 94] Selic, Bran; Gullekson, Garth ; Ward, Paul T. *Real-Time Object-Oriented Modeling*, 1994 Wiley
- [Sommerville 92] Sommerville, Ian *Software Engineering*, 4th ed. 1992 Addison-Wesley.
- [Symantec 93] *Symantec C++ Professional User's Guide and Reference. Version 6*, 1993 Symantec Corporation.
- [Turbo 91] *Turbo Pascal for Windows, Windows Programming Guide*, 1991 Borland Inc.
- [Visual Basic 92] *Visual Basic 2.0 for Windows*, 1992 Microsoft Inc.
- [Wirfs-Brock et. al. 90] Wirfs-Brock Rebecca; Wilkerson Brian; Wiener Lauren *Designing Object-Oriented Software*, 1990 Prentice-Hall Inc.
- [Wang 94] Wang, Yun *Interaction Diagram Generator. A Tool for Visualizing Use Cases in Smalltalk*. 1994 M.CS. thesis, School of Computer Science, Carleton University.

END

3 1-05-95

FIN

Chapter 6

Conclusion

6.1 Summary

In this thesis, we introduced an approach to designing and programming that we called *Need Driven Design and Programming* (NDDP). Although the method has been practiced individually by some expert object oriented programmers, particularly in the Smalltalk community, it has received little attention in the software engineering literature.

We divided the NDDP process into two basic steps:

- Writing scenarios (use-cases) that capture the requirements of the application.
- Incrementally adding the required methods to make these use-cases work.

We argued that this approach is natural for object oriented software development, because it provides a disciplined approach to incremental software development. By applying the need-driven technique, we are seeking a closer match between the design and implementation. Additionally, the technique strives to generate fewer extraneous methods, because all methods are added as and when needed. Because of the interactive nature of the process, which is only partially supported by existing software development environments, we created a prototype of an environment to support NDDP. It is our belief