1

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

PRECISIONSM RESOLUTION TARGE

Daedalus: Reducing the Complexity of Smalltalk

by

Julie A. Beeston

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

August 14, 1995

copyright, 1995.

Julie A. Beeston

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

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

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

ISBN 0-612-13828-3

Canada

Name Julie Beeson

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

Computer Science

SUBJECT TERM

0984

U·M·I

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

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

EDUCATION

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

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0286
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

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

PHILOSOPHY, RELIGION AND THEOLOGY

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

SOCIAL SCIENCES

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

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

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0777
Wood Technology 0746

Biology

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

Biophysics

General 0786
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

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

HEALTH AND ENVIRONMENTAL SCIENCES

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

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences

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

Applied Sciences

Applied Mechanics 0346
Computer Science 0984

Engineering 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 System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

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

The undersigned hereby recommend to
The faculty of Graduate Studies and Research

acceptance of this thesis
Daedalus: Reducing the Complexity of Smalltalk

submitted by
Julie A. Beeston. 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 "E. Demetris", is written over a horizontal line.

Director, School of Computer Science

A handwritten signature in black ink, appearing to read "Bin N. F. B.", is written over a horizontal line.

Thesis Supervisor

Carleton University
August 23, 1995.

Abstract

Daedalus is the prototype of an extension to Smalltalk which reduces the complexity of Smalltalk by automatically formatting code and by adding a new layer of abstraction to the current Smalltalk hierarchy.

This thesis confirms that automatic formatting reduces the complexity of Smalltalk by making complex code easier to read and keeps programmers from losing the logical flow of their own code even when it becomes large and complex.

This thesis also demonstrates the benefits of SELECTION BOXES. Selection boxes provide a new layer of abstraction to the Smalltalk hierarchy by grouping similar classes together into a single abstraction. The benefits of selection boxes are two fold. First they reduce the number of new concepts that a programmer needs to understand. Second they give the programmer fast, easy access to the most relevant information about a class - its behavior and user interface - from within the user's own code.

The Legend

In Greek Mythology, Daedalus was an Athenian inventor and artist.

Daedalus designed a labyrinth to hold the powerful, mythical monster Minotaur. Like many of man's creations, the labyrinth was used for things that it was not originally designed for. Inevitably the Athenian hero Theseus was imprisoned in the labyrinth.

Daedalus devised a scheme to guide Theseus out of the labyrinth, but by freeing Theseus, Daedalus incurred the wrath of King Minos. In a fit of rage, King Minos imprisoned both Daedalus and his son Icarus in the labyrinth.

Daedalus then came face to face with a dilemma that has confronted every problem solver since the beginning of time: a problem is much harder to solve when you are in the middle of it rather than being an outsider with a more global view of the situation.

Daedalus finally realized the solution. Working in two dimensions the problem was far too complex to solve, but when he added the third dimension (up) the problem became trivial. He gathered some feathers and wax then he crafted some wings for himself and his son. Soon they were both flying out of the maze to freedom. Icarus, in his delight, flew so close to the sun his wings melted and he fell into the water, but Daedalus went on to safety.

[Russo]

Acknowledgments

I would first like to thank my advisor Bjorn Freeman-Benson. His knowledge, insight and tireless pursuit of perfection gave me a glimpse of the traits of a good researcher. I would like to thank the Carleton University School of Computer Science for allowing me the opportunity to do this thesis. Finally I would like to thank my husband for never letting me give up on my dreams.

Table of Contents

Abstract.....	iii
The Legend.....	iv
Acknowledgments.....	v
<hr/>	
Introduction	1
Motivation.....	2
Shrinking The Hierarchy.....	6
Formatting.....	11
Summary.....	14
Related Work.....	15
Reducing Program Complexity.....	16
The Training Wheels Approach.....	17
The Reduced Language Complexity Approach.....	19
Automatic formatting.....	21
Summary.....	23
<hr/>	
Daedalus: The Programmer's perspective	24
Daedalus: The Programmer's Perspective.....	25
Daedalus Auto Formatting.....	25
Underline ClassNames and Globals.....	27
Unary Messages in Bold Font.....	29
Keywords in Bold font.....	31
Receiver and keywords on separate lines.....	32
Comments in grey background.....	33
Syntax Errors and Misspellings in Italic.....	35
Stretchy Brackets Spanning Several Lines.....	38
Automatic Indentation.....	41
Daedalus Selection Boxes.....	44
<hr/>	
Daedalus: The Computer's perspective	48
Daedalus: The computer's perspective.....	49
The Parser.....	50
The Rule Based Tree Walker.....	50
The Display Manager.....	53

Summary.....	54
<hr/>	
User Testing	55
Subjects.....	56
Apparatus	57
Experimental Design	58
Part A:	58
Part B:	58
Part C:	58
Procedure.....	59
Rating	60
Results.....	61
The Assignment	61
Part A	61
Part B	64
Part C	67
The Questionnaire	68
Interpretation of Results	71
The Assignment	71
Part A	71
Part B	72
Part C	75
The Questionnaire	75
Bias.....	76
Gender Bias.....	76
Tester Familiarity Bias.....	77
Tester Bias	77
Subject Bias	78
Sample Size Bias.....	78
Alternate Interpretation of Results.....	79
Better Programmers In Test Groups Than In The Control Group	79
Summary	80
<hr/>	
Future Work	81
Future Work.....	82
<hr/>	
Summary	86
Summary	87
<hr/>	
References	88

List Of Figures

1: ParcPlace Smalltalk-80 Collection Hierarchy	7
2: Daedalus Selection Box	8
3: Information Launched by Selection box	10
4: Daedalus Stretchy Brackets	12
5: ODE Life Cycle of Programing	18
6: Formatting of C code	22
7: Example of Class Name in Vanilla Smalltalk	28
8: Example of Class Name in Daedalus Smalltalk	28
9: Example of Syntax Error Vanilla Smalltalk - Class Name	28
10: Example of Syntax Error in Daedalus Smalltalk - Class Name	28
11: Vanilla Smalltalk - Unary Message	29
12: Daedalus Smalltalk - Unary Message	29
13: Vanilla Smalltalk - new Unary message	30
14: Daedalus Smalltalk - new Unary message	30
15: Vanilla Smalltalk - keyword formatting	32
16: Daedalus Smalltalk - keyword formatting	32
17: Daedalus Smalltalk - Blank comment	33
18: Daedalus Smalltalk - Comments	34
19: Smalltalk - No Syntax Error	37
20: Smalltalk - Syntax Error	37
21: Daedalus - No Syntax Error	37
22: Daedalus - Syntax Error*	37
23: Daedalus - Cascaded message	38
24: Smalltalk - Brackets	40
25: Daedalus - Brackets	40
26: Smalltalk - Indentation	42
27: Daedalus - Indentation	42
28: Smalltalk - Temporary Variables	43
29: Daedalus - Temporary Variables*	43
30: Daedalus - Selection Boxes	44
31: Daedalus - Selection Box List of Methods for OrderedCollection	45
32: Daedalus - Selection Box Comment for add: Method of OrderedCollection	45
33: Daedalus - Selection Box List of Methods for New Class GroceryStore	46
34: Daedalus - Selection Box Comment for amountAvailable: brand:	47
35: Flow of Source Code to Screen	49
36: Source Code	50
37: Parse tree	51
38: Daedalus Code	52
39: Profile of subjects	57
40: Coding times for Part A of the Assignment	62

41: Vanilla Smalltalk Comments in Part A	62
42: Formatted Smalltalk Comments for Part A	63
43: Daedalus Smalltalk Comments for Part A	63
44: Programming Times for Part B.....	65
45: Programming Times for Part B (continued)	66
46: Coding Times for Part C.....	67
47: Questionnaire Results	68
48: Programming Results.....	74

Introduction

Motivation

Object-oriented programming has evolved from a sideshow curiosity into a full fledged paradigm with academic and industrial support. One reason for this success is the inherent usefulness of object oriented techniques for describing complex problems. Inheritance and encapsulation allow the programmer to build manageable abstractions that hide the implementation so they can be reused or replaced with little or no effort. In other words, classes and objects reduce the cognitive complexity of large systems to the point where programmers can grasp both the underlying system and the new features being implemented. This is one of the benefits of switching to object oriented programming.

However, as Peer's law suggests, the solution to a problem changes the problem. Object oriented programming is no exception. One of the powerful features of object-oriented programming -- building abstractions (classes) -- is also one of its drawbacks. The problem is that the number of classes added to the modern system grows continually. In the last few years it has reached a point where the cognitive complexity of the system is as great as it was pre-objects. Thus, to make the make the next leap in productivity, we must develop a mechanism for reducing the complexity of object-oriented programs akin to the way that objects reduced the complexity a decade ago.

We can see this growth of complexity in the evolution of Smalltalk. In 1987, Alan Borning observed that the 200 classes and 500 methods of Smalltalk-80 were already too much for students to master, yet today such systems as ENVY/Smalltalk [Object Technology International Inc.] contain more than 700 individual classes with over 17000 methods. Programmers, be they first year students or experienced designers, can easily lose the simple logic of their design within the immensity of choices available.

If the growth rate of the class hierarchy continues unabated Smalltalk (and other object oriented languages) will continue to expand until they exceed the capacity of the programmer to understand them. If this growth is going to be effectively sustained, then some mechanism must control the cognitive complexity of the system. Possible solutions include (1) restricting the functionality of programming languages (e.g. Portia [Gold and Rosson 91]); (2) developing a new paradigm (e.g. Multiparadigm [Meyers and Reiss 91]); Or (3) producing tools that reduce the apparent complexity of the programming language. This thesis proposes the latter.

There are many techniques one could use to reduce the cognitive complexity of object-oriented programs. I implemented two that I believe have the largest reduction-per-unit-effort payoff: abstracting the class hierarchy and designing consistent code layout formats.

Any abstraction that reduces the amount of information necessarily also reduces the cognitive complexity [Terry 95]. Grouping similar classes together in a single abstraction saves the user from the burden of learning several similar ideas. This also complements the statement attributed to Alan Kay in [Borning and O'Shea 87], "in programming language design similar constructs should either be made identical or they should be made clearly dissimilar", since the properties of the classes within the abstraction can be made clearly discernible.

This thesis demonstrates through student testing that using selection boxes as an abstracting device not only allows students to learn fewer concepts, it also motivates them to explore concepts that they would not otherwise use. A side benefit of selection boxes is that they give fast, easy access to method comments and this thesis demonstrates that this not only encourages students to make better use of the comments that are already written, it also encourages them to make more meaningful comments in their own code.

Reusable code allows the user draw on code written by several different programmers, but every programmer has his or her own programming style. Deciphering another programmer's style can slow the user down so that he or she loses the flow of the program. The consistent formatting in Daedalus enables programmers to see at a glance the logical flow of a piece of code. The fonts, colors, and spatial positions of the text maximize the readability of the code while minimizing the effort required to follow the logic.

The benefits of automatic formatting were unmistakable during student testing. Those students with automatic formatting were more than forty percent faster at writing code than those who did not have automatic formatting.

This thesis demonstrates that both automatic formatting and the use of selection boxes have a measurable impact on programmers effectiveness and satisfaction.

Shrinking The Hierarchy

A more detailed look at the cognitive complexity caused by the size of Smalltalk will show some of the benefits of Daedalus.

The immense size of Smalltalk can confound even the most experienced programmers and can stonewall new users who feel compelled to try to understand the entire system before they start [Skublics 91]. Many people use the analogy of spaghetti code [Borning and O'Shea 87] using a similar analogy one could call the effect ravioli code. Smalltalk treats each class as an independent unit, and puts each method in its own window or pane. The net result of this plethora of windows is a somewhat disconnected view of the code that does not necessarily follow any linear, logical path. The mass of available classes seem like disconnected pieces of ravioli. If the full language is to be used effectively

(1) the language will have to be restricted; or

(2) tools reducing the complexity will have to be developed.

Restricting the number of classes included in the system would limit the size of the class library, but it would also diminish the functionality and flexibility of the language. The goal then is to minimize the size of the hierarchy without losing any of Smalltalk's abilities.

Daedalus achieves a measurable decrease in the complexity of the Smalltalk hierarchy by grouping similar classes together and encapsulating them within a single abstraction. For example the "Collection" hierarchy consists of classes that store multiple items. Each subclass of Collection has a slightly specialized function, but they are easily thought of as a single abstraction: Collections.

1: ParcPlace Smalltalk-80 Collection Hierarchy

Figure 1: "ParcPlace Smalltalk-80 Collection Hierarchy" shows an example of the collection hierarchy. Since different versions of Smalltalk have different classes and structures, this is not the only possible collection hierarchy, but it does illustrate the problem. This view of the hierarchy is confusing. Not only are there too many different classes to consider, it is also not immediately clear what the difference between an Array and a ByteArray is. In Daedalus Collections are grouped together and represented by a single abstraction like the one in Figure 2: "Daedalus Selection Box".

2: Daedalus Selection Box

Collection	
☒	Ordered
☐	Sorted
☐	Linked
☐	Unordered
☐	No Duplicates
☐	Indexed
☐	By Integer
☐	By Object

With this new abstraction, the user is presented with a single object rather than thirteen disjoint ones. This produces fewer concepts for the user to grasp, and it is also immediately obvious what the properties of the collection are.

The class Interval is conspicuously absent from the selection box in Figure 2: "Daedalus Selection Box". Since Intervals are not normally considered Collections by new programmers, they were left out of this abstraction. This demonstrates one of the most powerful features of selection boxes. They do not necessarily have to follow the rules of hierarchies and class inheritance. Instead they are free to be completely logical from the type point of view rather than having to be logical from the implementation point of view.

The information box is also a good launch point for giving the user relevant information. "Information Launched by Selection box" on page 10 shows the browser window which appears when the Collection selection box is clicked on. This browser displays the messages the Collection responds to with the current properties. The programmer can select a method and view just the comment of that method in a workspace akin to the one at the bottom of the diagram. This reduces the programmer's compulsion to read through the source code to discover how a method works [Skublics 91].

3: Information Launched by Selection box

Formatting

Although reading through source code is not desirable, it is often necessary when code needs to be updated and maintained or the programmer wishes to see how another programmer solved a similar problem. Smalltalk has no enforced formatting rules, but if the programming style of the code under inspection is not elegant, or is even just different than the style the user usually uses, this process can be unnecessarily difficult. Excessively flexible syntax is not only unnecessary but a problem [Golberg and Kay 77].

One paper [Miara et al. 83] revealed that a simple change like automatically indenting block statements had a noticeable effect on the readability of code, but much more could be done. Comprehension could increase and the debugging process could be more effective by using different font sizes, different emphasis, different color of characters and different spatial locations [Baecker and Marcus 86].

Daedalus uses the formatting style of [Baecker and Marcus 86] as a template for its own user interface. Daedalus uses different font sizes to make parenthesis more descriptive. The parenthesis in Daedalus stretch to encompass the entire block even if that block spans several lines. This makes it simpler to match each open parenthesis with its closing pair. When there are nested parenthesis it is clear which code is encompassed within the parenthesis. It is also immediately obvious if a parenthesis is missing.

4: Daedalus Stretchy Brackets

```

badBrackets: aNumber
  "This method tests Daedalus stretchy brackets"
  ||
  aNumber > 4
    ifTrue: [1
              to: 6
              do: [:index |
                    index
                    to: 7
                    do: [:sub |
                          Transcript show: sub]
                  ]
            ]
    ifFalse: [aNumber
               to: 1
               by: -1
               do: [:i |
                     Transcript show: 'Not four']
             ]

```

Daedalus uses different emphasis to draw attention to important words. The keywords of a message are in bold print to distinguish them from the arguments. Daedalus also puts the receiver and each key word on different lines to intensify the impact of bold key words.

The introduction of different colors greatly enhances the visual perception of the structure. Comments are displayed in grey so they are easier to locate. The grey background of each comment indicates to the user if a comment is used correctly, or if a piece of code has been commented out.

These changes make browsing code unambiguous. Programmers who can not read each other's code can not talk about it [Golgberg and Kay 77]. On the other hand, code that is nicely formatted is straight forward so programmers can draw on other programmers experience to solve problems and find creative new ideas.

Summary

Programming is essentially a creative act, yet the it is easy to lose the creative flow of a program in the complexity of a system like Smalltalk. If the complexity of Smalltalk was reduced without losing its flexibility, students could focus their attention on object oriented programming techniques rather than the class browser.

Daedalus helps alleviate program complexity by making the hierarchy appear smaller and by doing elegant formatting automatically. Making the hierarchy appear smaller inhibits the student from experiencing information overload. Adding auto formatting enhances student understanding and facilitates discussions on other students' code so students can focus on the dynamics of programs without getting bogged down trying to understand and debug simple syntax.

Daedalus helps the programmer focus his or her efforts on the fundamental problem solving aspects of programming without being overwhelmed by the programming language itself. Focusing the user's attention on problem solving makes Smalltalk nicer to use as it continues to evolve to handle more complex problems.

Related Work

Learning a new programming language can be as complicated and frustrating as learning a new human language or it can be as natural as a child learning to talk. Researchers devote considerable resources to studying this phenomena in the hope that it will lead to a teaching technique that will make learning more enjoyable and productive. [Kumar 95] sums up the bulk of this research by quoting twenty-one guidelines for effective instructional design. Four of the twenty-one guidelines are directly relevant to Daedalus:

- 04. Motivate the learner by introducing the subject vis-a-vis its future relevance and importance
- 08. Organize complex information in easy - to - remember structures.
- 12. Allow time for cognitive processing in order to internalize the concepts.
- 19. Provide immediate feedback of the learner's responses

Guidelines 4 and 12 flow automatically from Daedalus since it is a final programming language rather than an introduction to a programming language. From the start students program in the context of the complete language.

Guideline 8 relates to reducing the complexity of the programming language. Daedalus achieves this through the use of selection boxes which take large complex concepts such as the collection hierarchy and reduces them to one simple, envisagable item.

Guideline 19 is provided in part by Daedalus's automatic formatting which allows the programmer to know immediately if there are any syntax errors in the code and if the logic of the code is correct.

Reducing Program Complexity

There have been many approaches to reduce the cognitive complexity of programming. The traditional approach of having structured classes followed by exercises to confirm and enhance understanding is still widely used, but the classroom approach to teaching programming is slowly being replaced with a more hands on approach. The high tech teaching motto of the nineties has been adopted from a young persons agricultural club that has been around for decades. The 4-H Motto: Learn To Do By Doing.

Many papers emphasizes the importance of giving the student meaningful tasks from the beginning in order to establish the process and practice of identifying and analyzing objects in a problem domain. [Lilly 94] expressed it best by saying "Teaching object technology is useless... unless we see our role as facilitators of technology transition, not just as classroom teachers." These papers all concur that the classroom is at most only part of the experience of learning object oriented programing and that on the job training, mentoring and continuing education are all essential parts of the process.

There are many schools of thought about what the best approach is to making make programming simple enough so that students can immediately become productive programmers. Two of the schools of thought gaining popularity are: the Training Wheels approach, and the Reduced Language Complexity approach (the approach that Daedalus uses). The Training Wheels approach breaks a language down into domains and gives

the new user access to only the simplest domain. As the new user becomes more experienced, the more complex domains are introduced until the entire language is mastered. The Reduced Language Complexity approach identifies the areas that new users (or experienced users) have difficulty with, and tries to illuminate them or reduce their impact.

The Training Wheels Approach

A group at the IBM T.J Watson Research Center is researching how people begin to program with Smalltalk. The group has proposed several variations on the training wheels approach in order to help students learn and understand Smalltalk easily. [Carroll et al. 90] reduces the complexity of the Smalltalk Hierarchy by only displaying pertinent classes.

[Campbell 90] and [Campbell et al. 90] propose breaking the problem domain down into several domains and allowing the new user to master one domain at a time. To this end, they suggest seven levels of Smalltalk programming proficiency.

1. Interacting with the visual interface
2. Syntax rules and order of precedence
3. Locating classes and methods in the hierarchy
4. Class versus instance distinction
5. Model-View-Controller
6. Object-Oriented Design
7. Grandmaster level

Another such paper [Robertson et.al. 94] involving the language ODE promotes teaching students to work on identifying objects in a problem domain while de-emphasizing purely syntactic and notational details. Using ODE, the analyst or designer first studies the program domain. Through the study of the problem domain, the analyst or designer comes up with different scenarios (narrative descriptions of what activities can take place within the Problem Domain). These scenarios are then used to generate an object model. The problem domain itself is designed to make the analysis and design phase of this cycle to give more information about the problem domain, which in turn produces more scenarios and the whole cycle can be repeated indefinitely.

5: ODE Life Cycle of Programing

The ODE study found that students who were exposed from the onset to realistic tasks were more proficient at learning, and since they were programing from the beginning in the same context which they would eventually be working, the students found it easier to transfer the knowledge they gained in the teaching environment to real work and work situations.

The Training Wheels Approach is a good approach to teaching, but it has very little usefulness once the student has mastered the grandmaster level. From this point on, the user has a through understanding of the programming language, but many of the tools he or she used to learn the language are no longer useful to him or her. The search for tools which students can use both during and after training leads us to the Reduced Language Complexity approach.

The Reduced Language Complexity Approach

Reducing Language Complexity can mean one of two things: reducing the number of features in a language, thus reducing the functionality of the language itself, or reducing the cognitive complexity of the constructs in the language. The first approach is used by Umbriel [Terry 95] which is designed to bring out the intrinsic simplicity of Modula-2 by only allowing a subset of features and eliminates those features which seem to cause the most bewilderment to beginners. This approach is simply a special case of the Training Wheels approach where there is only one domain and the user is never guided to the grand master level.

The more interesting approach - reducing the complexity without losing any functionality - is provided in part in Deltatalk [Borning and O'Shea 87]. Deltatalk was developed to reduce the complexity of Smalltalk 80.

The producers of Deltatalk noticed that no one problem dominated the difficulties new users had learning Smalltalk, but they also found that a small subset of problems impact nearly every student. Both metaclasses and inheritance were found to be problems to students and so Deltatalk recommended that they either be illuminated or eliminated. This paper favors illumination over elimination since Daedalus contends that nothing should be lost in the pursuit of simplicity.

Another problem addressed by both Deltatalk and Daedalus is system size. Deltatalk introduces what it calls lasagne programming, where a form of layering of the modules in the hierarchy is introduced. This complements Daedalus' idea of having the selection boxes which add a new layer of abstraction to the Smalltalk hierarchy in order to reduce the number of key concept the user has to master.

Building off their lasagne metaphor, I propose the ravioli metaphor which describes the problem. There are so many distinct, self contained classes many of which are indistinguishable to the new user, that the user can get lost in a sea of modules and have no idea how all of the classes fit together. Daedalus alleviates this problem through selection boxes which give easy access to the most relevant information about a class - its interface to the user - and by reducing (conceptually) the number of classes the user has to master.

Automatic formatting

The benefits of consistent formatting on both new and experienced users' ability to write and understand code is a widely accepted concept and has been studied in great detail on many different languages. [Hary et al. 88] determined that code formatting which highlights the logical flow of the code has a positive impact on both novice and experienced users. The paper [Miara et al. 83] found that something as simple as indentation makes a marked improvement on user satisfaction and comprehension. [Baecker and Marcus 86] went beyond this to develop five design principles:

1. **Typographical Vocabulary:** Choose a small number of appropriate type styles of suitable legibility, clarity and distinguishability, applying them thoughtfully to encode and distinguish various kinds of program tokens. Within each typeface, choose or design a set of enhanced letter forms and symbols with which to represent the text effectively.
2. **Typesetting parameters:** Adjust to enhance readability the typesetting parameters of text point size, headline size and usage, word spacing, paragraph indentation, and line spacing.
3. **Page Composition:** Bring out the program structure by carefully structuring the page composition through the use of grids, the application of explicit and implicit lines of varying thickness (typographic "rules") and the inclusion of appropriate white space.
4. **Symbols and Diagrammatic Elements:** Integrate appropriate symbols and diagrammatic and graph elements to elucidate essential program structure. In this way we achieve non-textual augmentation of the source code proper.
5. **Metatext:** Augment the source text with mechanically generated supplementary text, additional data and commentaries that enhance the comprehensibility of the source text.

Using these design principles they produced the C code demonstrated in "Formatting of C code" on page 22.

Summary

Daedalus is part of a large body of research dedicated to increasing new programmers' enjoyment and satisfaction by reducing program complexity and increasing readability.

Some programming languages prefer to reduce the functionality of a language or only give the students the complete functionality after they have mastered the basic concepts.

Daedalus aligns with the other school of thought which promote making languages easier to understand by increasing their readability while decreasing the number of constructs that must be learned without reducing the overall functionality of the language.

Daedalus

The Programmer's perspective

Daedalus: The Programmer's Perspective

Within this section I will attempt to show the benefits of Daedalus from the perspective of an end user. I will present views of non-Daedalus Smalltalk and contrast them with Daedalus in order to demonstrate the real benefits of the Daedalus enhancements to Smalltalk. This should promote a clearer understanding of the results obtained in the student trials.

Daedalus Auto Formatting

First of all I will examine the benefits obtained from the addition of auto-formatting. The benefits of auto-formatting can be broken down into the following areas:

- (1) Underlined ClassNames and Globals
- (2) Unary messages in bold font
- (3) Keywords in bold font
- (4) Receiver and keywords on separate lines.
- (5) Comments in grey background
- (6) Syntax errors and misspellings in italic
- (7) Unparsable code in italic
- (8) Stretchy brackets spanning several lines
- (9) Automatic indentation
- (10) Temporary variables in italic

These areas were chosen to emulate [Baecker and Marcus 86] as closely as possible. In the areas where the two programming languages were not compatible, I chose a formatting style that appeared to highlight the logical flow of the code.

The formatting style was chosen as much by my own bias and personal preference as it was by empirical study. As such it is not the only possible formatting style. In “Future Work” on page 82 I will discuss the possibility of making the formatting style configurable so the programmer can decide what styles work best for him or her, but it is important to keep in mind that the more freedom the programmer has in choosing his or her own style, the harder it will be for programmers to read and discuss each other’s code. Configurable Daedalus would automatically convert the code to the user’s favorite format if it was loaded into the user’s image, but quite often new users work together at the same terminal on a project, and they may have a communication barrier if they are both accustomed to different programming styles. Therefore I recommend that Configurable Daedalus be confined to the realm of study, and independent or experienced programmers, while the domains of students and new users be restricted to a consistent style of formatting.

Underline ClassNames and Globals

Within the Smalltalk language class names and globals retain special status. They are the landmarks the designer uses to map the Smalltalk jungle. It is important that they be distinguishable from the code wrapped around them. To accomplish this I chose to underline them. With this knowledge a designer can easily distinguish the classnames and globals within their code, or for that matter locate the classnames and globals within other designer's code. It also enables the designer to avoid simple spelling errors, or other finger problems. This can save a designer vast amounts of time searching for coding problems.

Figure 7: "Example of Class Name in Vanilla Smalltalk" shows an example of vanilla Smalltalk with a new (non-Smalltalk standard) class name. Figure 8: "Example of Class Name in Daedalus Smalltalk" shows the same code but with Daedalus auto-formatting. As you can clearly see there is no doubt within the Daedalus example exactly what the class being referred to is. This becomes even more apparent when syntax errors creep into the new code.

With Figure 9: "Example of Syntax Error Vanilla Smalltalk - Class Name" the S in GroceryStore is not capitalized. This syntax error is not apparent, nor is there any indication of any problem at all.

7: Example of Class Name in Vanilla Smalltalk

```
test
    "Create a new instance of class GroceryStore"
    | groceryStore |
    groceryStore := GroceryStore new
```

8: Example of Class Name in Daedalus Smalltalk

```
test
    "Create a new instance of class GroceryStore"
    | groceryStore |
    groceryStore := GroceryStore new
```

9: Example of Syntax Error Vanilla Smalltalk - Class Name

```
test
    "Create a new instance of class GroceryStore"
    | groceryStore |
    groceryStore := Grocerystore new
```

10: Example of Syntax Error in Daedalus Smalltalk - Class Name

```
test
    "Create a new instance of class GroceryStore"
    | groceryStore |
    groceryStore := Grocerystore new
```


Once Daedalus has performed its auto-formatting, there is a clear indication that some error has occurred. As we can see in Figure 10: "Example of Syntax Error in Daedalus Smalltalk - Class Name" the class GroceryStore is not underlined. It appears as a temporary variable or a misspelling. A designer would have immediate information that there is something wrong with the class reference.

Unary Messages in Bold Font

Unary messages are the simplest type of messages within Smalltalk. Because of their simplicity, it is often difficult to distinguish a unary message from a temporary or instance variable.

11: Vanilla Smalltalk - Unary Message

multiUnaryMessage

"This method tests writing out of multiple unary messages."

| a |

a isNil printString.

12: Daedalus Smalltalk - Unary Message

multiUnaryMessage

"This method tests writing out of multiple unary messages."

| a |

a isNil printString

Figure 11: "Vanilla Smalltalk - Unary Message" shows a common unary message. Both `isNil` and `printString` are unary messages. This code is shown with Daedalus formatting in Figure 12. "Daedalus Smalltalk - Unary Message".

Although the formatting is very evident, the benefits may not be as obvious. As most Smalltalk designers are familiar with both `isNil` and `printString` as unary messages. However when designing new code, or examining someone else's code, the unary messages may be a bit more difficult to identify.

13: Vanilla Smalltalk - new Unary message

shop: aGroceryStore

"This method demonstrates unary messages in bold"

| products |

products := aGroceryStore products.

14: Daedalus Smalltalk - new Unary message

shop: aGroceryStore

"This method demonstrates unary messages in bold"

| *products* |

products := aGroceryStore **products**

Figure 13: "Vanilla Smalltalk - new Unary message" shows a more complex but very common example of overloading words. The word `products` is used as both the unary message and a temporary variable. In this example the unary messages are not so obvious. But in Figure 14: "Daedalus Smalltalk - new Unary message", once again Daedalus has points out exactly which bits of the code are unary messages. This example is still quite simple, but the evidence is apparent. In a more complex piece of code, the benefits would be even more apparent.

Keywords in Bold font

With Smalltalk the keyword messages are the most complex method by which Smalltalk objects may communicate with one another. They may contain any number of selectors or keywords. Each selector has its own argument and the message itself has a receiver. With keyword messages it is important to be able to determine which part of the message is the keyword, which part is the receiver and which part is the argument. This can become very difficult in some cases. For instance when there are nested keyword messages. In Figure 15: "Vanilla Smalltalk - keyword formatting" is an example of one of the simplest nested keyword messages. Even in its simplest form it is quite complex, especially to an inexperienced designer.

15: Vanilla Smalltalk - keyword formatting

buySquared: aNumber

"To demonstrate keyword message formatting in
Daedalus"

GroceryStore new buy: aNumber * aNumber of: 'Apple'
brand: 'Spartan'.

16: Daedalus Smalltalk - keyword formatting

buySquared: aNumber

"To demonstrate keyword message formatting in Daedalus"

||

GroceryStore new

buy: aNumber * aNumber

of: 'Apple'

brand: 'Spartan'

Figure 16: "Daedalus Smalltalk - keyword formatting" displays the same code after Daedalus has auto formatted it. In the Daedalized code it is more apparent which parts of the message are keywords and which are arguments or receivers.

Receiver and keywords on separate lines

The purpose of this addition to Formatting in Daedalus is fairly apparent. Compare Figure 16: "Daedalus Smalltalk - keyword formatting" with Figure 15: "Vanilla Smalltalk - keyword formatting". In the Daedalus the code is much clearer. The same Formatting style is possible in all versions of Smalltalk. Some versions of Smalltalk, including ParcPlace, offer an automatic formatter for code which puts the receiver and

keywords on separate lines and indents the keywords. Daedalus takes this a step further by formatting automatically while the code is being generated and it goes beyond the simple text formatting that ParcPlace does by using different fonts and new characters such as stretchy brackets. It also makes a consistent view of the code for designers to re-visit, or borrow from other designers.

Comments in grey background

One of the most important parts of coding and also one of the least attended is comments. This is even more prevalent in object oriented programming languages such as Smalltalk due to the intended re-use of code. With proper comments designers can quickly become familiarized with inherited code. This enables designers to quickly determine if the code does what they need and if it will be useful to them.

Daedalus has even more reasons for designers to make proper use of comments. When a designer accesses a selection box, he/she has immediate access to the comments of the Methods of a Class. This allows the designer to quickly browse the available methods within a class and determine if he/she has selected the appropriate Class.

17: Daedalus Smalltalk - Blank comment

shop

"Comment"

| a |

^self

By putting comments with a solid grey box, Daedalus entices students to properly comment their code. If the user forgets to put in a comment, Daedalus will put in a default comment like the one shown in Figure 17: "Daedalus Smalltalk - Blank comment"

18: Daedalus Smalltalk - Comments

```

brands: aProduct
"Returns a collection of brands for the specified product"
"groceryStore"
"groceryStore := GroceryStore new"
"groceryStore brands: 'Apples' "
||
^ (inventory at: aProduct) keys

```

Figure 18: "Daedalus Smalltalk - Comments" demonstrates that the comments can hold quite a bit of useful information if the designer chooses to take full advantage of them. This example demonstrates the designer using the comments to both describe the purpose of the code, and to provide an example of the proper syntactical use of the code. The benefits of Selection Boxes will be explored further in the section "Daedalus Selection Boxes" on page 44.

Syntax Errors and Misspellings in Italic

One of the most frustrating parts of designing code is troubleshooting errors which turn out to be simple misspellings or syntax. These errors are sometimes extremely difficult to locate. The most frustrating part of them however is the ease at which they are corrected upon identification. This is a problem which faces both the novice designer and his or her more experienced colleagues. By bringing the error to the immediate attention of the designer, Daedalus eliminates most of these errors before they have a chance to cause any problems.

In vanilla Smalltalk, syntax errors can be missed quite easily. They may be misspellings, or simple typing errors. However they are not always easy to locate. Figure 19:

"Smalltalk - No Syntax Error" is probably the most commonly written algorithm in Computer Science. It does not (as the title claims) contain any syntax errors. Contrast this with Figure 20: "Smalltalk - Syntax Error". This piece of code is identical in all but one misspelled word (Transcript). In this instance the result is minor, but it is not always so trivial.

With Daedalus's italicization of syntax errors and misspelled words it is unlikely this type of error would get past first reading, if that far. By changing the font style so dramatically it brings to the designer's attention that there is some type of finger problem. With this immediate knowledge the designer can easily locate and fix the error before it becomes a de-bugging issue.

Comparing the two examples shown in Figure 21: and Figure 22: we can see how the italicization of the misspelling is immediately visible.

19: Smalltalk - No Syntax Error

```
printt
  "Prints Hello World to the Transcript"
Transcript show: 'Hello World'
```

20: Smalltalk - Syntax Error

```
printt
  "Prints Hello World to the Transcript"
Transcrt show: 'Hello World'
```

21: Daedalus - No Syntax Error

```
printt
  "Prints Hello World to the Transcript"
  |
  Transcript show: 'Hello World'
```

22: Daedalus - Syntax Error*

```
printt
  "Prints Hello World to the Transcript"
  |
  Transcrt show: 'Hello World'
```

*Note: Italics is more noticeable on the computer screen than on paper

Unparsable code in Italic

Since Daedalus is was built as a prototype, not all coding features were implemented.

For example, cascaded messages are not understood by Daedalus. When Daedalus runs into a piece of code that it can not parse, it considers the code to be a syntax error and write the code in italics without any formatting. Figure 23: "Daedalus - Cascaded message" shows a cascaded message in Daedalus.

23: Daedalus - Cascaded message

cascade

"This method demonstrates a limitation in Daedalus."

"It does not understand cascaded messages"

||

Transcript show: 'Hi world'; cr

Stretchy Brackets Spanning Several Lines

One of the most difficult skills for a student programmer to master is the idea of nested loops. This is made even more difficult if there is complex functionality performed within the brackets. The idea of enforced formatting is probably most important in the area of brackets. Keeping track of brackets is precarious at the best of time (just ask any LISP coder), but within nested loops the problem compounds itself. The tedious job of organizing the brackets to make the code easier to follow by keeping track of matching brackets is a job Daedalus performs.

The example used to describe this feature is intentionally complex to demonstrate that even with a complex set of nested loops, and a total lack of Formatting by the designer, Daedalus will make the code quite readable and understandable. Figure 24: "Smalltalk - Brackets" shows a piece of code which is very difficult to follow, and would be even worse to modify or debug. The same code put through Daedalus, Figure 25: "Daedalus - Brackets", is quite easily understood, and would be very simple to modify and debug.

24: Smalltalk - Brackets

```
badBrackets: aNumber
  "This method tests Daedalus stretchy brackets"
  aNumber > 4 ifTrue: [ 1 to: 6 do: [:index| index to: 7
do: [:sub| Transcript show: sub]]] ifFalse: [aNumber to: 1
by: -1 do: [:il| Transcript show: 'Not four']]
```

25: Daedalus - Brackets

```

badBrackets: aNumber
  "This method tests Daedalus stretchy brackets"
  ||
  aNumber > 4
    ifTrue: [
      to: 6
      do: [ :index |
          index
            to: 7
            do: [ :sub |
                Transcript show: '
                    '
            ]
          ]
    ]
    ifFalse: [ aNumber
      to: 1
      by: -1
      do: [ :i |
          Transcript show: 'Not found
    ]
  ]

```

Automatic Indentation

The idea of automatic indentation is not a new one, but it is a good one. Indentation makes the code much easier to group together, and in large pieces of code it is very useful in aiding the readability of the code. Most designers use some form of indentation, but Daedalus makes the indentation consistent throughout all of the code. This assists the designer in both writing original code, and reading/modifying inherited code. By giving all code a consistent "look" the designer feels partial, familiar with code the first time he/she examines it.

Figure 26: "Smalltalk - Indentation" shows an example of coding with no use of indentations. The code is very cryptic and difficult. Daedalus takes this and applies the indentation producing code which is more readable as shown in Figure 27: "Daedalus - Indentation".

Although few designers are this negligent in the use of indentations, there are striking differences between individual preferences in this area. Daedalus removes this inconsistency and presents a consistent indentation formatting style.

26: Smalltalk - Indentation

indentation

"Demonstrates the automatic indentataion in Daedalus"

| groceryStore |

groceryStore buy: (2 squared raisedToInteger: 4) of: 'A',
'pples' brand: ('Spook',
'an' changeFrom: 3 to: 6 with: 'art').

27: Daedalus - Indentation

indentation

"Demonstrates the automatic indentataion in Daedalus"

| groceryStore |

groceryStore

buy: (2 squared raisedToInteger: 4)

of: 'A', 'pples'

brand: { 'Spook', 'an'
changeFrom: 3
to: 6
with: 'art' }

Temporary Variables in Italic

A temporary variable is exactly what its name implies. Making it easily distinguishable from Classes and Methods serves twofold. First, it enables easy identification for the Designer (both the writer and any future inheritor) and second, it makes it very simple to eliminate syntax errors and typos from any incorrect Class or Method declarations.

28: Smalltalk - Temporary Variables

```
shop
  "This method demonstrates temporary variables in
  italics"
  | groceryStore |
  groceryStore := GroceryStore new.
```

29: Daedalus - Temporary Variables*

```
shop
  "This method demonstrates temporary variables in italics"
  | groceryStore |
  groceryStore := GroceryStore new
```

*Note: Italics is more noticeable on the computer screen than on paper

As we can see by comparing Figure 28: "Smalltalk - Temporary Variables" with Figure 29: "Daedalus - Temporary Variables*" the italic Formatting of temporary variables in Daedalus is a subtle reminder to the designer.

Daedalus Selection Boxes

Within Daedalus, selection boxes are a graphical representation of the new level of abstraction added to the Smalltalk Class Hierarchy. They combine similar Classes into one integrated checklist from which the designer may examine and explore the Class most suitable for his or her programming needs.

As shown in Figure 30: "Daedalus - Selection Boxes" a subset of the sub-Classes of Collection are displayed with Ordered being chosen. By selecting Ordered, the programmer determined the code will use the Class OrderedCollection. After selecting a type of Collection, the designer can examine his or her chosen Collection further by clicking the mouse inside the Selection Box anywhere where the click will not be used to select a different Collection.

31: Daedalus - Selection Box List of Methods for OrderedCollection

When the selection box is selected it opens the browser as shown in Figure 31:

"Daedalus - Selection Box List of Methods for OrderedCollection". This enables the designer to examine the methods available for the selected Class to determine if it is appropriate for his/her needs.

32: Daedalus - Selection Box Comment for add: Method of OrderedCollection

If the designer wishes further detailed information upon selected methods he/she may select one of them and accept it. This will bring forth a work space with the comments of the method within it as shown in Figure 32: "Daedalus - Selection Box Comment for add: Method of OrderedCollection".

33: Daedalus - Selection Box List of Methods for New Class GroceryStore

If the user creates a new Class it would need to have a selection box defined for it. By default Daedalus puts new Classes into a selection box of one Class. Figure 30:

"Daedalus - Selection Boxes" shows this with the new Class GroceryStore. The functionality of this new selection box is identical to that of the selection boxes for existing code such as Collection. As we can see in Figure 34: "Daedalus - Selection Box Comment for amountAvailable: brand:" the easy access to the comments of new and existing methods can be used for more than examination of code comments. It promotes the designer to include more detailed comments with their new code as they can quickly see the benefits of this discipline.

34: Daedalus - Selection Box Comment for amountAvailable: brand:

Industrious designers may take the comments a step further by including coding examples within the comments. This will enable them to quickly clone these pieces of code into new methods, thus taking code re-usability a step further.

Daedalus

The Computer's perspective

Daedalus: The computer's perspective

Within this section I will discuss the technical side of Daedalus. That is I will discuss how Daedalus produces the effect that the user sees. All of the information in this section is done behind the scenes and is totally transparent to the user.

In order to promote portability across platforms, Daedalus code is stored in an identical fashion to normal Smalltalk code. All of the formatting done by Daedalus is done during the display of the source code. In normal Smalltalk, there is a one-to-one connection between the source code and the code that appears on the screen. In Daedalus, the source code must go through a parser, rule based tree walker and a display manager before it is displayed. (See "Flow of Source Code to Screen")

35: Flow of Source Code to Screen

The Parser

A parser converts a flat piece of source code into a parse tree like the one in "Parse tree" on page 51. In vanilla Smalltalk the interpreter traverses the resulting parse tree in order to execute the code. In Daedalus the parse tree is used instead by the rule based tree walker as a tree with a one-to-one correspondence to the Daedalus Objects tree. Because both parsers perform essentially the same function Daedalus uses the normal Smalltalk parser for parsing code.

The Rule Based Tree Walker

The parse tree that the parser returns is then processed by the rule based tree walker which goes through the tree and translates each node directly into a Daedalus object.

Daedalus objects can be simple objects (For example, Daedalus Literal which manages a literal such as a string) or compound objects (Such as Daedalus Binary Message which contains a receiver a selector and an argument). Each Daedalus object knows how to display itself and all sub-objects once an origin has been provided. Each Daedalus Object also knows how much space it is going to take to display.

For example, the code in "Source Code" on page 50 has the parse tree displayed in "Parse tree" on page 51.

36: Source Code

```
divide: aNumber
  "Divide self by aNumber if aNumber is not equal to zero."
  aNumber = 0
    ifTrue:[^self error: 'Divide by zero']
  self // aNumber.
```

37: Parse tree

When the rule based tree walker and the display manager have done their tasks, the code is displayed in a Daedalus window using the format displayed in "Daedalus Code".

38: Daedalus Code

```


divide: aNumber



"Divide self by aNumber if aNumber is not equal to zero."
      aNumber = 0
      ifTrue:
        [ ^ self error: 'Divide by zero' ]
      ^ self // aNumber
    ]


```

The selection boxes that are used when the rule based tree walker encounters a Class name are a special case of the Daedalus objects. Each class is responsible for providing a method that returns the correct selection box. The rule based tree walker calls this method to produce the selection boxes. Since the code hierarchy does not necessarily correspond to the Daedalus selection box hierarchy, many subclasses of the same class need to have different selection boxes defined. For example, the Collections Interval and OrderedCollection although siblings in the inheritance tree, are not logically related, so their selection boxes are quite different, therefore both collections need their own unique selection box generator.

Once the Daedalus objects are created, they are placed in an ordered collection (the display collection) for the display manager.

The Display Manager

On the first pass through the display collection, the display manager tells each Daedalus object in turn to display itself on the screen and then asks it for its display height so it can calculate the origin for the next Daedalus Object in the display collection.

Once the first pass of the display collection is done, the source code is displayed on the screen and the user can edit it. As the user edits the code displayed by a Daedalus object, the display manager continually sends it back to the parser for re-parsing, and then to the rule-based-tree-walker for re-translation into Daedalus objects. The new Daedalus objects then replace the old Daedalus Object in the display collection and if the combined height of the resulting Daedalus objects is different from the old Daedalus object then the origins of all the surrounding Daedalus objects are recalculated, and the surrounding daedalus objects are re-displayed.

If at any time the code that the user is typing in is not parsable, it is put in a nonparsable Daedalus object which is displayed without any formatting and all of its text in italics.

Summary

The Parser, the Rule-Based-Tree-Walker and the Display Manager massage the source code on its journey to the screen rather than allowing the source code to be displayed directly the way Vanilla Smalltalk does. All three steps add extra overhead, so they must be done efficiently in order to make the transition totally transparent to the user. Since Daedalus was only a prototype many of the efficiency issues were not fully addressed, but they are discussed in the section "Future Work" on page 82.

User Testing

The viability of the Daedalus programming environment was demonstrated by construction. I designed and coded Daedalus as an extension to Smalltalk which provides a conceptually smaller set of classes and formatting which is both automated and elegant. I then proceeded to verify that Daedalus reduces conceptual complexity by conducting tests with first year computer science students and monitoring their reactions to both Daedalus and Smalltalk.

Subjects

The test was composed of three groups with three subjects in each group. The control group was exposed to vanilla Smalltalk. The next group was exposed to Smalltalk with the automatic formatting Daedalus provides. The final group was exposed to full blown Daedalus encompassing both automatic formatting and class simplification.

Each of the subjects was a student at Carleton University taking Computer Science 95.105 (a first year course in computer programming). Within this class there was a wide range of education and experience. "Profile of subjects" on page 57 show the education level and gender of each of the nine subjects.

None of the students had worked with this style of Smalltalk previously and they were not made aware of where Smalltalk ended and Daedalus began until after the test was complete.

39: Profile of subjects

Subject	Gender	Year	Major
A1	Female	1	Law
A2	Male	11	Civil
A3	Male	Cont. Ed.	N/A
B1	Male	1	Electrical
B2	Male	1	Computer Science
B3	Male	4	Geology
C1	Male	2	Mathematics
C2	Male	1	None
C3	Female	Special	N/A

Apparatus

The tests were conducted on an IBM clone 486 66 MegaHertz computer with 32 Megabytes of RAM. The computer was equipped with a 15 inch color monitor.

Experimental Design

Assignment

In Smalltalk, there is currently a class called GroceryStore that keeps track of the inventory of a corner grocery store. It responds to such methods as:

- products
- brands: 'Apples'
- priceOf: 'Apples' brand: 'Spartan'
- amountAvailable: 'Apples' brand: 'Spartan'
- brandNamesAndPrices: 'Apples'
- buy: 20 of: 'Apples' brand: 'Spartan'

Part A:

Create a class called GroceryItem that has product, brand and amount as instance variables. Write get and set methods for each.

1. Write a printOn: method that will print out the grocery item in the form: 20 Apples (Spartan)

Part B:

Create a class called GroceryList that has groceryStore (among others) as an instance variable.

2. Create a graphical user interface (GUI) which allows the user to add grocery items to the GroceryList.
3. Create a method shop which takes the shopping list and returns a collection with the items the user was able to buy. If the item was not available in the preferred brand name, then it should be replaced with the cheapest available brand name of the same product. If there is not enough of the item (in all the brand names of the item) then buy whatever is available and inform the shopper with a dialog box that the order was not completely filled.
4. Write a print method that prints the collection obtained in (2) on the Transcript.

Part C:

(Do only when part B is finished)

Modify your shop method to prompt the user for a new brand name if the brand name of the item they want is not available.

Procedure

Each subject was given a copy of the assignment and a brief introduction to the Smalltalk environment immediately before the test.

They were then asked to do part A of the assignment while I observed and took notes.

They were then given a more in-depth lesson into the Smalltalk Graphical User Interface (GUI) builder. This enabled them to build an interface for the class they produced in part A. In the final group of students, the lesson also included an introduction to the selection boxes and their functionality. They were then asked to complete Parts B and C of the assignment while I observed and took notes. While the students were programming, communication between the student and myself was kept to a minimum. The communication was restricted to occasional reminders to the student to think out loud so I could follow their thought process, questions about the programming environment (for example one student who was not familiar with class categories asked how to find what category a class was in) which I answered and questions about the correct coding algorithm or Daedalus specific questions which I declined to answer until after the test was complete.

Finally they were given the questionnaire in the following section to complete.

Rating

Name: _____ Student

Number: _____

Year of Program: _____ Major: _____

Signature: _____

1. Rate the level of difficulty of this assignment compared to other assignments you have done.

Too Easy	Easy	Average	Hard	Too Hard
-------------	------	---------	------	-------------

2. Rate the amount of time needed to complete this assignment compared to other assignments you have done

Very Little	Little	Average	A Lot	Too Much
----------------	--------	---------	----------	-------------

3. Rate your confidence level that the code does what the assignment specifies

Not Sure At All	It Might	It Is Probably Correct	Fairly Confident	It Is Definitely Correct
-----------------------	----------	------------------------------	---------------------	--------------------------------

4. For your shopping list, what type of Collection did you use? _____
Why did you choose that type of collection?

5. Please state any comments you think are appropriate.

Results

The Assignment

Part A

All six of the subjects using Daedalus commented all of their methods without being prompted. Two of the three without Daedalus did not comment their methods at all and the third person asked if it was really necessary. He decided to go back and comment the code later if there proved to be enough time.

Without exception, all six of the subjects not using selection boxes had trouble remembering the syntax of such simple methods as `printOn:`, `nextPutAll:` and `printString`. For example, "Vanilla Smalltalk Comments in Part A" on page 62 identifies that subject A1 at time 6 minutes could not remember the syntax of the `printOn:` method. His problem was caused because he could not remember that the argument to a `printOn:` method was generally a stream, and could not recall the messages a Stream responds to. Subject B1 in "Formatted Smalltalk Comments for Part A" on page 63 had a similar problem. He knew that he was writing to a stream, but could not remember the message name `nextPutAll:` for writing strings to the stream. The three subjects using selection boxes had no such problem.

40: Coding times for Part A of the Assignment.

In the preceding figure, A1-A3 represent the three subjects using Vanilla Smalltalk. B1-B3 represent the subjects using Formatted Smalltalk. C1-C3 represent the subjects using Daedalus.

41: Vanilla Smalltalk Comments in Part A

Subject	Time* (Minutes)	Action
A3	2	Comment: "Do I have to put a comment in? I guess it depends on how much time we have"
A2	5	Syntax error: Put instance variable names in uppercase.
A1	6	Could not remember the syntax of the printOn: method
A2	6	Could not remember syntax of nextPutAll:
A1	8	Syntax error: did not put printString when printing an integer
A2	10	Could not remember the syntax of printString.
A3	10	Could not remember how to write a printOn: method

Note: Time here refers to the actual time that the subject made the comment. Use "Coding times for Part A of the Assignment." on page 62 to correlate his or her other activities at that time.

42: Formatted Smalltalk Comments for Part A

Subject	Time (Minutes)	Action
B3	4	Comment: "How do you write a method?"
B3	4	Comment: "Do I need to write comments?"
B1	1-4	Trying to remember how to make a new class
B2	5	Comment: "What the heck is going on? Is there some reason it is changing font on me?" (Noticed Daedalus fonts for the first time)
B3	10	Comment: "I find this a bit slow. I have to guess how many back spaces I need."
B1	5-10	Trying to remember the syntax of accessing method for instance variables
B1	12	Can not remember syntax of nextPutAll:
B3	16	Could not remember the syntax of nextPutAll:

43: Daedalus Smalltalk Comments for Part A

Subject	Time (Minutes)	Action
C2	1	Comment: "Do you care about comments?"
C2	7	Comment: "That's cool that unary and keyword items are highlighted in bold"
C3	8	Comment: "[formatting] It's kinda cool"
C4	10	Comment: "That [formatting/stretching brackets] is kinda neat"

Part B

The average time for finishing part B without Daedalus was 91 minutes, but only two of the three subjects without Daedalus managed to finish. The third became very confused while trying to sort out bracketing problems and was unable to following the logic of her code. She finally gave up on the part B after 80 minutes.

When automatic formatting was introduced, the average time to complete part B went from 91 minutes to 50 minutes.

When full blown Daedalus was added, the average time for part B dropped to 38 minutes.

45: Programming Times for Part B (continued)

In the preceding figure, A1-A3 represent the three subjects using Vanilla Smalltalk. B1-B3 represent the subjects using Formatted Smalltalk. Note, subjects using Daedalus did not take this long

Part C

“Coding Times for Part C” on page 67 shows that the time to write the code in part C was between two and six minutes for both Formatted Smalltalk and Daedalus, whereas the subjects using Vanilla Smalltalk took between ten and eleven minutes. However the most dramatic increase in speed is in the length of time the subjects need to find the spot in the code that needed changing (the hot spot). The subjects without Daedalus took an average of six minutes to find the hot spot while those with Daedalus took less than a minute.

46: Coding Times for Part C

In the preceding figure, A1-A represent the three subjects using Vanilla Smalltalk. B1-B3 represent the subjects using Formatted Smalltalk. C1 -C3 represent the subjects using Daedalus. Note that A1 is missing since she got so confused in part B that she gave up on the assignment.

The Questionnaire

47: Questionnaire Results

Group	Subject	Assignment Difficulty	Time required	Confidence in code	Type of Collection	Reason for Collection	Comments
Vanilla Small-talk	A1	Hard	Average	It might	Didn't get that far		
	A2	Hard	Average	Fairly Confident	Ordered	95-105 influence. Only one I really know about	
	A3	Easy/Average	Average	Fairly Confident	Ordered Collection	Save memory compared to Dictionary	

47: Questionnaire Results

Group	Subject	Assignment Difficulty	Time required	Confidence in code	Type of Collection	Reason for Collection	Comments
Small-talk with formatting	B1	Average	Average	Fairly Confident	OrderedCollection	It keeps every item in one place, and allows an easy way to access these items	I liked the comments are highlighted in gray and can be seen easily and bold characters keeps it interesting and easy to correct.
	B2	Easy	Average	Fairly Confident	OrderedCollection	Fairly Obvious and simple choice for the task at hand (no need to sort, discard doubles, etc.)	<ul style="list-style-type: none"> -Formatting can show syntax errors -Good way at showing bracket definitions/excesses -I like Daedalus format of drawing the code (bracket sizes, different fonts; would recommend different colors for some of the code's highlights)
	B3	Easy	Average	Fairly Confident	OrderedCollection	Access and data retrieval is easy	<ul style="list-style-type: none"> -I love the stretchy brackets -Syntax errors were italicized -very easy to comment (they were in grey) -The different fonts were hard to get used to, but once [they] became easier to understand, [they became] useful with practice.

47: Questionnaire Results

Group	Subject	Assignment Difficulty	Time required	Confidence in code	Type of Collection	Reason for Collection	Comments
Daedalus	C1	Average	Little	It is Definitely correct	Ordered Collection	It seemed appropriate didn't require an more specific structure	The ability to search the methods of each class was helpful.
	C2	Easy	Average	It is definitely Correct	Unordered	Because I don't shop in any order! (Minimum collection was an unordered collection	-Visual aid in access appropriate methods was very useful. -Interface creation (Dialog boxes) was very easy. Daedalus Browser formatting may be slower than acceptable on slower PCs when accessing complex/large methods.
	C3	Average	Average	Fairly Confident	Ordered Collection	To maintain some semblance of logic and consistency with the grocery list	Liked visual format of program; found font changes a little disconcerting. Especially liked the boxes and big brackets for ifTrues etc. (thought the Control-F and T was cool, also).

Interpretation of Results

The assignment was designed to test Daedalus' ability in two areas. First to see if Daedalus improved students ability to write code and second to see if Daedalus improved students understanding of the code they had written.

The Assignment

The assignment was designed to quantify the advantages of Daedalus in three main programming situations. Part A was designed investigate how much Daedalus improves new users ability to write simple code. Part B was designed to test how well Daedalus performs when new users write complex methods that require the user to manipulate other people's code. Part C was designed to see if Daedalus improved the subject's ability to modify their own code.

Part A

The greatest benefit of Daedalus in part A was its tendency to encourage students to comment their code. Comments are extremely important when writing code to make it easier to maintain and reuse later. In Daedalus comments are even more critical due to the selection boxes giving users easy access to the comments of methods. This is to encourage the more effective and consistent use comments. The subjects using Daedalus commented all of their methods without being prompted. Two of the three without Daedalus did not comment their methods at all and the third person asked if it was really necessary. He decided to go back and comment the code later if there proved to be enough time.

Even on the simple methods of part A, Daedalus' selection boxes provided a marked improvement on the subjects ability to remember method names. Without exception, all the subjects not using selection boxes had trouble with such simple methods as `printOn:`, `nextPutAll:` and `printString`, whereas the subjects using selection boxes had no such problem.

On simple methods like those of part A with only one or two lines of code, students found Daedalus to be slow, but this was mainly due to the fact that the tests were performed on a prototype. A production version of Daedalus could - presumably - be enhanced to run more efficiently.

Part B

In part B of the assignment the benefits of Daedalus were unmistakable. Without Daedalus part B proved to be long and frustrating. This was expressed best by one subject. She became very confused while trying to sort out bracketing problems. The size of her *shop* method compounded her problem by causing her to be unable to follow the logic of her code. She finally gave up on the part B after 80 minutes.

Those who did managed to complete the assignment without the help of Daedalus were the ones who enforced a strict formatting style upon themselves, and spent a good proportion of their time looking over their code to ensure that it was properly formatted.

When automatic formatting was introduced, the average time to complete part B went from 91 minutes to 50 minutes. This decrease in time roughly corresponds to the amount of time those in the first group spent formatting their code.

The improvements caused by the auto formatting were incredible, but the subjects with only auto formatting and no selection boxes shared a problem with those who had only vanilla Smalltalk. They spent a good deal of their time looking through the hierarchy to find which methods they could use. Searching through the hierarchy not only decreased their coding speed, it also broke their train of thought and forced them to re-examine their code prior to resuming code writing.

When full blown Daedalus was added, the average time for part B dropped to 38 minutes. The subjects were no longer forced to search through the hierarchy to find which methods their classes responded to. Instead they simply wrote the name of the class and used the selection box to get the list of methods. Once they became proficient at that, they discovered that many of the methods had examples of code usage included within the comments. They simply cut and pasted these examples into their own code. This encouraged code reuse and reduced simple syntax errors caused when typing.

48: Programming Results

Part C

In part C the automatic formatting again made a marked improvement. Those without Daedalus took an average of six minutes to find the spot in their code where the change was to be made while those with Daedalus took less than a minute.

The Questionnaire

The responses to the questionnaire indicated that Daedalus made a marked improvement for the subjects in two areas. It caused the difficulty rating of the assignment to be reduced and it increased general understanding.

Daedalus caused a marked increase in the ease with which the students found the assignment. Those in the control group found the assignment to be hard, while those in the other two groups found the assignment easy or average.

Daedalus also had an impact on the subjects understanding of collections. All but one of the subjects used an Ordered Collection. Those without the selection boxes gave reasons for using them such as 'OrderedCollections are the only collections I really know about', or compared them to Sets or Dictionaries which were the other two sets they had touched on in their course. Those with Daedalus gave answers such as 'It understands add:, and

do:'. The only person to not use an Ordered Collection was one of the subjects using Daedalus with selection boxes. He used an Unordered Collection (Bag). On the questionnaire he stated that even though he had not been exposed to Unordered Collections, he decided that an unordered collection was the minimum collection necessary to do the assignment.

In the comments sections of the questionnaire there was a definite trend of things the subjects liked and did not like. All liked the stretching brackets, the comments in grey and the way the code became italic when syntax errors were present. Many found the different fonts confusing at first, but found them helpful once they got used to them. Finally, the subjects using the selection boxes did not find the selection boxes themselves that useful, but they did appreciate the easy access to the method names and comments that the selection boxes gave them.

Bias

The possibility for bias was eliminated as much as possible but there is some bias inherent in the experiment itself which inevitably affects the results.

Gender Bias

The effect of gender on programming has been studied with great enthusiasm. The male mind is organized differently than the female mind, and both sexes are specialized to perform different tasks more effectively.

Two of the nine subjects were female, so it was impossible to have the gender mix be identical. To reduce the error caused by this discrepancy, the two female subjects were tested as A1 (the first Vanilla Smalltalk subject) and C3 (the last Daedalus subject).

There were two reasons for this choice. First, making the formatted Smalltalk group be the odd group had the least impact on the experiment since the Daedalus group was also exposed to the automatic formatting and therefor retested the formatting. Second, it put the two females as far apart chronologically so that gender bias was not compounded by tester familiarity bias.

Tester Familiarity Bias

The tests were done in the order Vanilla Smalltalk, Formatted Smalltalk, Daedalus.

Undoubtedly my understanding of the assignment had improved by the time the Daedalus subjects were tested since I had already explained it to the other two groups.

To keep this effect at a minimum, the notes for the lesson were written on a white board once. With the exception of the addition of an explanation of how to use selection boxes that was added solely for the Daedalus subjects, the notes were identical for each of the groups.

Tester Bias

A tester is naturally more apt to be excited about his or her own research than about other people's research. Plus the subject is likely to want to help to the tester get the desired results if the subject knows what those results are.

In order to reduce the effect of this the subjects were tested in a version of Smalltalk that they had never used before, so even the control group thought that they were a test group. The subjects were not told until they completed the assignment which aspects were added by Daedalus and which were vanilla Smalltalk. They were encouraged to give comments on the whole programming environment, and their comments were recorded whether they were related to Daedalus or not.

Subject Bias

All of the subjects were students in 95.105 (a first year course in computer science), but very few of them were computer science students, and they were all at different levels of education. "Profile of subjects" on page 57 shows that the students education ranged from 1 to 11 years of university, and they had a wide variety of majors. In order to reduce the impact of this, the subject for each test group were chosen randomly in order to achieve a fair distribution. The only non random aspects of the choice of subjects for each test group were the constraints caused by subject availability and the constraint of having at least one female in both the vanilla Smalltalk group and the Daedalus group.

Sample Size Bias

Since there were only three subjects in each of the three groups, the sample size has the most dramatic impact on experimental error. Three subjects are not enough to have any statistical significance. They can only be used to reveal trends towards increasing productivity and give evidence that this subject is worth studying further.

Alternate Interpretation of Results

The test results indicate that Daedalus has a positive impact on programming efficiency and enjoyment, however there are alternate interpretations to the results obtained.

Better Programmers In Test Groups Than In The Control Group

The results obtained through user testing could be explained as “the subjects in the test groups were simply better programmers than those in the control group”. Adding authority to this claim is the fact that the three subjects with a Computer Science related major (B1, B2 and C1) were all in the test groups.

However, there are three arguments to this claim. First, in part A of this assignment where the benefits of Daedalus were not as apparent, all the subjects finished in about the same time. Second, in part B of the assignment, B1 and B2 did no better than B3 (the Geology major) while C1 did worse than both C2 and C3. Third, the subject with the most university experience (A2) was in the control group.

Summary

Daedalus appears to have a positive influence on both the user's speed of writing and understanding of code.

The dramatic impact of Daedalus is caused when it is used to write large methods or reading and understanding complex code. The length of time needed to write a large method is cut by more than half.

The selection boxes have a dramatic impact on the subject's writing and understanding of code, while the automatic formatting keeps the user from losing the logic of the code in the complexity of a large method.

The results of the testing show a definite trend towards helping the new user in both speed and understanding of programming. Although the small sample size makes the results statistically insignificant, the trend is still evident.

Future Work

Future Work

Three streams of research naturally flow out of the results of this thesis. The first stream of research involves doing more in-depth user testing experiments with Daedalus. The second stream of research involves enhancing Daedalus' programming environment. The third stream of research involves studying the long and short term benefits of programming by example.

The methods used to test Daedalus on students could be enhanced to produce more convincing results. The main areas where the testing could be enhanced would be in increasing the sample size, decreasing the Hawthorn effect and increasing user flexibility.

The small sample size used to test Daedalus had a dramatic impact on the confidence level of the results. Daedalus could be tested on a large number of programmers who were at different skill levels in order to confirm Daedalus' usefulness to a wide range of programmers.

The Hawthorn effect is the effect that the tester has on the subjects being tested by simply conducting tests. The testing in this thesis tried to reduce this impact by allowing all three groups to believe that they were in the test group, but this did not eliminate the effect completely. The fact that the tester was in the room while the tests were being conducted also contributes implicitly to the Hawthorn effect. To reduce the Hawthorn effect either the subject could be video taped so the tester would not have to be in the same room, or a key recording device could be added to Daedalus. With the key

recording device, all of the programmers key strokes and mouse movements could be recorded so that the subject could do the test at his leisure and the tester could reproduce the subjects actions after the test is complete. Both of these option suffer from the ability to solicit comments from the subjects while they program so they should be combined with other testing procedures.

The tests done in this thesis all involved seeing how well students performed while using Daedalus. None of the tests in this thesis studied if students would chose to use Daedalus if they were given the freedom to choose¹. In order to test which features of Daedalus students would choose to use, they could be given a version of Daedalus in which each of the features could be turned on and off at will.

These enhancements to the testing process would provide results that are more statistically significant and more complete which will give more confidence to Daedalus' ability as a programming language.

Another stream of research that goes beyond the scope of Daedalus involves enhancing Daedalus' programming environment. I received several suggestions on how to make Daedalus more effective that went beyond the scope of this thesis. The first (and perhaps most appropriate) is to enhance the programming environment so that the automatic formatting style could be configurable. This would give the user the opportunity to

1. As a post thesis field test of Daedalus I have loaned Daedalus to some real world developers who have found it quite useful, and they have received requests from their co-workers for copies of Daedalus

decide how far blocks should be indented, and whether keywords should be in bold. Since Daedalus only changes the programmer's view of the code, not the underlying structure of the code itself, making the formatting style configurable would have the added benefit of allowing several users to work on the same code and each of them have a different view of the code with their own desired formatting. This is only one step away from having programmers on different continents viewing and modifying the same code in their own native language. Comments, of course would not be as easily transferable, but that would be an obstacle for those researchers who pick up this project to explore.

A second enhancement to Daedalus would be to have a GUI to automatically generate complex selection boxes without the user having to know how to generate the source code for the selection box manually.

A third enhancement to Daedalus would be to do the engineering work necessary to make it run more efficiently so that the speed of Daedalus would not affect the programmer. The two real bottle necks to Daedalus are the parser and the mouse controller. Daedalus currently uses the Smalltalk parser which is fine tuned to parse code for execution. Daedalus could have its own parser that was optimized to parse code for formatting. This could include (1) recognizing unparseable code more quickly since while a line of code is being written it is not - generally - parsable, or (2) parsing a line of code without knowing the context of what method it is in. To parse code without knowing

which method it is in, instance variables would have to be treated as a special case. Either the instance variable would have to be included in the parsing, or all spelling errors would have to be treated as temporary variables. The preceding enhancements would be relatively simple to make, but optimizing the mouse controller would require a much more thorough investigation.

A third stream of research which goes beyond the scope of Daedalus is the short and long term benefits of programming by example. The test subjects that used selection boxes found that comments that contained examples of how to use the code were very useful. This mirrors the accidental finding of both Data I/O and Apple Computer, Inc. Data I/O produced ABEL with some examples of simple code. Like Daedalus, they soon found that creations of man are almost never used entirely for the purpose they were designed. These examples of simple code were being used as templates, and over 90% of the code written in ABEL contained at least one of these examples in one form or another. MacApp from Apple had a similar problem. Although the numbers were much lower for MacApp they were still significant. In fact, McApp's sample code had bugs and these bugs showed up in many early Macintosh applications. [Freeman-Benson 94]

Both streams of research, enhancements to Daedalus' programming environment and the long and short term benefits of programming by example are research fields in their own right and worth of further study.

Summary

2 of/de 2

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

PRECISIONSM RESOLUTION TARGETS

Summary

The purpose of this thesis was to maximize programmer efficiency and enjoyment by reducing the complexity of Smalltalk. This purpose was realized through developing, prototyping, and verifying Daedalus, an extension to Smalltalk. The net result is a programing environment with both reduced complexity and automatic formatting.

Daedalus draws on the inspiration of other designers who have sought ways to reduce the complexity of programing and takes the research a step further by displaying and testing a new approach.

The significance of Daedalus is three fold. First it demonstrates what every programmer knows intuitively, but has considerable difficulty trying to prove: automatic formatting that highlights the logical flow of the code and makes simple errors easy to detect increases user's productivity and enjoyment.

The second significant aspect of Daedalus is its new approach to reducing the complexity of the Smalltalk hierarchy by adding a new layer of abstraction.

The final significance of Daedalus is that it demonstrates the value of having relevant information about classes - their method and their comments - within easy reach of the programmer. This not only helped the programmer to not lose the logical flow of his program when he or she was writing it, it also encouraged the programmer to write meaningful comments.

Daedalus gives the programmer fresh ammunition for the continuing struggle against program complexity.

References

Baecker and Marcus 86

Ronald Baecker and Aaron Marcus. "Design Principles for the Enhanced Presentation of Computer Program Source Text". CHI '86 Conference Proceedings. April 1986. pp. 51-58.

Berry 92

Robert Berry. "Computer Benchmark Evaluation and Design of Experiments, A Case Study". 1992

Borning and O'Shea 87

Alan Borning and Tim O'Shea. "Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language". ECOOP Conference Proceedings, 1987.

Campbell 90

Robert L. Campbell. "Developing Scenario Analysis of Smalltalk Programming" SIGCHI '90 Conference Proceedings, Seattle, Washington. April 1990. pp 269-276

Campbell et al. 90

Robert L. Campbell, Norman R. Brown and Lia A. DiBello. "The Programmer's Burden: Developing Expertise in Programming. IBM Research Report RC15382. 1990. IBM T.J Watson Research Center, Yorktown Heights, NY. 8 pages.

Carroll et al. 90

John M. Carroll, Janice A. Singer, Rachel K.E. Bellamy, and Sherman E. Alpert. "A View Matcher for Learning Smalltalk". SIGCHI Conference Proceedings, Seattle, Washington. April 1990. pp 431-437.

Carroll 90

John M Carroll. The Numberg Funnel: Designing Minimalist Instruction For Practical Computer Skill. Cambridge, MA: MIT Press. 1990

Casais 90

Eduardo Casais, "Managing Class Evolution in Object-Oriented Systems". Object Management, Center Universitaire d'Informatique, University of Geneva, July 1990. pp 133-195

Chiou 95

Guey-Fa Chiou. "Beliefs and Computer - Based Learning". Educational Technology. 35 (3) May June 1995. pp 48-52

Cunniff et al. 86

Nancy Cunniff, Robert P. Taylor and John B. Black. "Does Programming Language Affect the Type of Conceptual Bugs in Beginners' Programs? A comparison of FPL and Pascal". CHI '86 Conference Proceedings. April 1986. pp. 175-182.

Dix et al. 93.

Alan Dix, Janet Finlay, Gregory Abowd and Russell Beale. Human Computer Interaction. Prentice Hall International Limited. Hemel Hempstead, Hertfordshire. 1993.

Eberts 83

Ray E. Eberts. User Interface Design. Prentice Hall, Englewood Cliffs, New Jersey. 1983

Freeman-Benson 94

Bjorn Freeman-Benson. Private communication. 1994

Gasen 95

Jean Gasen. "Questioning Minds Need To Know...". SIGCHI Bulletin 27 (2). April 1995. pp 3

Genter and Grudin 90

Donald R. Genter and Jonathan Grudin. "Why Good Engineers (Sometimes) Create Bad Interfaces". CHI '90 Conference Proceedings. April 1990. pp. 277-282.

Golberg and Kay 77

Adele Goldberg and Alan Kay. Teaching Smalltalk. Technical Report SSL 77-2, Xerox Palo Alto Research Center, 1977.

Goldberg and Ross 81

Adele Goldberg and John Ross. "Is the Smalltalk-80 System for Children?". BYTE Publications Inc. August 1981. pp 348-368

Gold and Rosson 91

Eric Gold and Mary Beth Rosson. "Portia: An Instance Centered Environment for Smalltalk". OOPSLA '91 Conference Proceedings 26 (11). Phoenix, Arizona. October 1991. pp 62-74

Hary et al. 88

Joseph M. Harry, Lori A. Cohan and Michael J. Darnell. "Users' Preferences Among Different Techniques For Displaying the Evaluation Of Lisp Functions In an Interactive Debugger". CHI 88 Conference Proceedings. Washington, D.C. May 15-19 1988. pp 45-50

Hirschman and Cuomo 95

Lynetter Hirschman and Donna Cuomo. "Evaluation of Human Computer Interfaces". SIGCHI Bulletin 27 (2). April 1995. pp 4-10

Kempf and Stelzner 87

Renate Kempf and Marilyn Stelzner. "Teaching Object-Oriented Programming with the KEE System". OOPSLA '87 Conference Proceedings. December 1987.

Knudsen and Madsen 88

Jurgen Lindskov Knudsen and Ole Lehrmann Madsen. "Teaching Object-Oriented Programming Is More Than Teaching Object-Oriented Programming Languages". ECOOP '88 Conference Proceedings. Oslo, Norway. August 1988. pp 21-40

Konstan and Rowe 91

J seph A. Konstan and Lawrence A. Rowe. "Developing a GUIDE Using Object-Oriented Programming". COPSLA '91 Conference Proceedings. Volume 26, Number 11. November 1991. pp. 75-88.

Kumar 95

Madhumita K.L. Kumar. "Twenty-one Guidelines for Effective Instructional Design". Educational Technology. 35 (3) May June 1995. pp 58-61

Leventhal and Teasley 94

Laura Marie Leventhal and Barbee Eve Teasley. "Analysis of Factors Relating to Positive Test Bias in Software Testing". International Journal of Human - Computer Studies. 41 (5). November 1994. pp 717-749

Lieberherr and Riel 89

Karl J. Lieberherr and Arthur Riel. "Contributions to Teaching Object Oriented Design and Programming". OOPSLA '89 Conference Proceedings. 24 (10). New Orleans, Louisiana. October 1989. pp 11-22

Lilly 94

Suzan Lilly. "Panel: Training Professionals in Object Technology". OOPSLA '94 Conference Proceedings 29 (10), October 1994. pp 47.

Mangennis 95

Jo P. Mangennis. "Training Quality: Before and After Winning the Deming Prize". Educational Technology. 35 (3) May June 1995. pp 26-31

Meyers and Reiss 91

Scott Meyers and Steven P. Reiss. "A System for Multiparadigm Development of Software Systems. Sixth International Workshop On Software Specification and Design Proceedings. Como, Italy. October 1991. pp 202-209.

Miara et al. 83

R.J. Miara, J.A. Musselman, J.A. Navarro and B. Schneiderman (1983). Program indentation and comprehensibility. Comm. of the ACM, 26 (11), 861-867.

Pintado 93

Xavier Pintado, "Advances in Information Visualization Techniques". working paper, Centre Universitaire d'Informatique, University of Geneva, 1993. in preparation.

Reenskaug and Skaar 89

Trygve Reenskaug and Anne Lise Skaar. "An Environment for Literate Smalltalk Programming. OOPLSA '89 Conference Proceedings. 24 (10). New Orleans, Louisiana. October '89. pp 337-345

Robertson et.al. 94

Scotte P. Robertson, John M. Carroll, Robert L. Mack, Mary Beth Rosson, Sherman R. Alpert, Jurgen Koenemann-Belliveau. "ODE: A Self Guided, Scenario-Based Learning Environment For Object-Oriented Design Principles". OOPSLA '94 Conference Proceedings Volume 2, Number 10, October 1994. pp 51-64

Rosson and Carroll 90

Mary Beth Rosson and John M. Carroll. "Climbing the Smalltalk Mountain". SIGCHI Bulletin 21 (3). January 1990. pp 76-79

Rosson and Carroll 93

Mary Beth Rosson and John M. Carroll, "Active Programming Strategies in Reuse". ECOOP '93 Conference Proceedings. Kaiserslautern, Germany. July 1993. pp 4-20

Rosson et al. 90

Mary Beth Rosson, John M. Carroll, and Rachael K.E. Bellamy. "Smalltalk Scaffolding: A Case Study of minimalist Instruction". SIGCHI '90 Conference Proceedings, Seattle, Washington. April 1990. pp 423-429

Rosson and Gold 89

Mary Beth Rosson and Eric Gold. "Problem-Solution Mapping in Object-Oriented Design". OOPLSA '89 Conference Proceedings. 24 (10). New Orleans, Louisiana. October 1989. pp 7-10

Rubin 86

R.V. Rubin. "Language Constructs for Programming by Example". Third ACM-SIGOIS Conference on Office Information Systems.

Russo 81

Robert G. Russo. Queens College, New York. Daedalus. The Encyclopedia Americana. International Edition. Volume 8. Grolier Incorporated. Danbury Connecticut. 1981. pp 417.

Schildt 92

Herbert Schildt. Teach Yourself C++. McGraw-Hill, Inc. 1992

Skublics 91

Suzanne Carol Skublics. "Sherpa: Guiding Students Up the Smalltalk Mountain" Thesis (M.S.C) Carleton University. 1991

Schuster 95

James M. Schuster. "Learning to Improve Learning: A Case Study". Educational Technology. 35 (3) May June 1995. pp 23-25

Strickland 95

Ted J. Strickland Jr. "An Information Systems Perspective of SIGCHI Curricula". SIGCHI Bulletin 27 (2). April 1995. pp 4-10

Terry 95

P. D. Terry. "Umbriel - another minimal programming language". SIGPLAN Notices 30 (5). May 1995

Turner 76

John H. Turner. The Myth Of Icarus in Spanish Renaissance Poetry. Tamesis Books Limited, London. 1976

Winograd 73

Terry Winograd. "Breaking the Complexity Barrier Again". SIGPLAN - SIGIR Interface Meeting: Programming Languages - Information Retrieval. Gaithersburg, Maryland. November 1973. pp. 13-30.

END

2 1 - 1 1 - 9 6

FIN