

**FROM SPECIFICATION THROUGH REFINEMENT TO
IMPLEMENTATION: A COMPARATIVE STUDY**

by

INGRID H. M. VAN COPPENHAGEN

Submitted in part fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in the subject

INFORMATION SYSTEMS

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISORS: PROF PAULA KOTZÉ

PROF JOHN A. VAN DER POLL

JUNE 2002



This dissertation is dedicated to my mother Herna de Villiers, and my husband Frikkie, and children Louis, Linda and Johan.

The author wishes to express her sincere appreciation to Prof Paula Kotzé for her patience, motivation, guidance, and support in formulating, executing, and completing this research project.

The author also wishes to express her sincere appreciation to Prof John A. van der Poll for his thorough and educated scrutiny of this dissertation and providing excellent guidance and advise towards the completion of this dissertation/research project.

The image features a large, semi-transparent watermark of the University of South Africa (UNISA) logo. The logo is circular with a scalloped edge. The outer ring contains the text 'UNIVERSITY OF SOUTH AFRICA' at the top and 'UNIBESITH VAN SUID-AFRIKA' at the bottom. Inside the ring, the Latin motto 'SPES IN ARDUIS' is written. The central emblem depicts an open book, a quill pen, and a scale of justice, all set against a red and yellow background.

UNISA

The author wishes to express his sincere appreciation to Prof John A. van der Pijl for his thorough and valuable scrutiny of the dissertation and providing excellent guidance and advice towards the completion of the dissertation project.

The author wishes to express his sincere appreciation to Prof John A. van der Pijl for his thorough and valuable scrutiny of the dissertation and providing excellent guidance and advice towards the completion of the dissertation project.

The author wishes to express his sincere appreciation to Prof John A. van der Pijl for his thorough and valuable scrutiny of the dissertation and providing excellent guidance and advice towards the completion of the dissertation project.

UNISA LIBRARY
2007 -01- 2 6
Class



0001943726

005.12 VANC

Abstract

This dissertation investigates the role of specification, refinement and implementation in the software development cycle. Both the structured and object-oriented paradigms are looked at. Particular emphasis is placed on the role of the refinement process.

The requirements for the product (system) are determined, the specifications are drawn up, the product is designed, specified, implemented and tested. The stage between the (formal) specification of the system and the implementation of the system is the refinement stage.

The refinement process consists out of data refinement, operation refinement, and operation decomposition. In this dissertation, Z, Object-Z and UML (Unified Modelling Language) are used as specification languages and C, C++ , Cobol and Object-Oriented Cobol are used as implementation languages.

As an illustration a small system, The ITEM System, is specified in Z and UML and implemented in Object-Oriented Cobol.

Key terms:

Specification, Refinement, Implementation, Data refinement, Operation refinement, Operation decomposition, UML, Z, Object-Z, C, C++, Cobol, Object-Oriented Cobol, Software development life cycle, Structured, Object-oriented, Class, Object, Schema, Inheritance, Verification, Validation, Axiomatic description.

Contents

Chapter	Page
List of Figures	IX
List of Tables	XI
1 Introduction	1-1
1.1 Introduction	1-1
1.2 Areas of investigation and research for this study	1-2
1.3 Questions posed in this study	1-3
1.4 Method of study	1-3
1.5 Layout of the dissertation	1-4
1.6 Summary of main findings and results	1-7
2 Refinement and the software systems development life cycle	2-1
2.1 Introduction	2-1
2.2 The Structured paradigm	2-2
2.2.1 Waterfall life cycle model	2-3
2.2.2 Boehm's spiral model	2-5
2.3 The Object-oriented paradigm	2-7
2.3.1 Fountain model	2-7
2.3.2 Outline development process	2-8
2.4 Overview of phases	2-9
2.4.1 Requirements definition phase	2-9
2.4.2 Requirements specification phase	2-9
2.4.3 Design phase	2-10
2.4.4 Implementation phase	2-11
2.4.5 Integration phase	2-11
2.4.6 Maintenance phase	2-11
2.5 Refinement process	2-12
2.6 Comparison of traditional life cycle models	2-12
2.7 Summary and Conclusion	2-15
3 Refinement: Non-object-oriented systems using Z	3-1
3.1 Introduction	3-1
3.2 Z Specifications	3-1
3.2.1 Introduction	3-1
3.2.2 Discussion	3-3
3.2.2.1 Types	3-3
3.2.2.2 Axiomatic descriptions	3-4
3.2.2.3 Schemas	3-5
3.2.2.3.1 State schemas	3-5
3.2.2.3.2 Operation schemas	3-6
3.3 Specification validation and verification	3-8
3.3.1 Verification	3-8
3.3.1.1 Terminology	3-8
3.3.1.2 Methods of verification	3-8

3.3.1.3	Verifying consistency of global definitions	3-9
3.3.1.4	Verifying consistency of state models	3-10
3.3.1.5	Verifying consistency of operations	3-11
3.3.2	Validation	3-12
3.4	Refinement	3-13
3.4.1	Global view of refinement	3-14
3.4.2	Data refinement	3-15
3.4.2.1	Concrete state adequacy	3-18
3.4.2.1.1	The Concrete state must be consistent	3-18
3.4.2.1.2	It must be determined whether every abstract state has at least one concrete representative	3-19
3.4.2.1.3	Verifying the correctness of the concrete initial state	3-20
3.4.2.2	Concrete operation applicability	3-21
3.4.2.3	Concrete operation correctness	3-22
3.4.3	Operation refinement	3-23
3.5	Operation decomposition	3-27
3.5.1	Sets	3-28
3.5.1.1	Smaller sets	3-28
3.5.1.1.1	Implementation into C	3-29
3.5.1.1.2	Implementation into Cobol	3-29
3.5.1.2	Larger sets	3-29
3.5.1.2.1	Implementation into C	3-30
3.5.1.2.2	Implementation into Cobol	3-30
3.5.1.3	Set membership	3-31
3.5.1.3.1	Implementation into C	3-31
3.5.1.3.2	Implementation into Cobol	3-31
3.5.2	Relations	3-32
3.5.2.1	Implementation into C	3-32
3.5.2.2	Implementation into Cobol	3-32
3.5.3	Functions	3-33
3.5.3.1	Implementation into C	3-33
3.5.3.2	Implementation into Cobol	3-33
3.5.4	Function application	3-33
3.5.4.1	Implementation into C	3-34
3.5.4.2	Implementation into Cobol	3-34
3.5.5	Functions and relations	3-34
3.5.5.1	Implementation into C	3-35
3.5.5.2	Implementation into Cobol	3-36
3.5.6	Bindings	3-36
3.5.6.1	Implementation into C	3-37
3.5.6.2	Implementation into Cobol	3-37
3.5.7	State Schemas	3-38
3.5.7.1	A system with a single instance (binding) of a schema type	3-38
3.5.7.1.1	Implementation into C	3-38
3.5.7.1.2	Implementation into Cobol	3-38
3.5.7.2	State schema implemented by the declaration of a C structure	3-38
3.5.7.2.1	Implementation into C	3-39
3.5.7.2.2	Implementation into Cobol	3-39
3.5.8	Operation Schemas	3-40

3.5.8.1	Implementation into C	3-40
3.5.8.2	Implementation into Cobol	3-40
3.5.9	Schema Expressions	3-40
3.5.9.1	Implementation into C	3-41
3.5.9.2	Implementation into Cobol	3-41
3.5.10	Assignment	3-41
3.5.10.1	Assignment where only one variable changes value	3-41
3.5.10.1.1	Implementation into C	3-42
3.5.10.1.2	Implementation into Cobol	3-42
3.5.10.2	Assignment where several variables can change	3-42
3.5.10.2.1	Implementation into C	3-43
3.5.10.2.2	Implementation into Cobol	3-43
3.5.11	Guarded Command	3-43
3.5.11.1	Implementation into C	3-44
3.5.11.2	Implementation into Cobol	3-44
3.5.12	Disjunction	3-44
3.5.12.1	Implementation into C	3-45
3.5.12.2	Implementation into Cobol	3-45
3.5.13	Conjunction	3-45
3.5.14	New Variables	3-46
3.5.14.1	Implementation into C	3-46
3.5.14.2	Implementation into Cobol	3-47
3.5.15	Quantifiers	3-48
3.5.15.1	Implementation into C	3-48
3.5.15.2	Implementation into Cobol	3-48
3.5.16	Partial Operations	3-49
3.5.16.1	Implementation into C	3-49
3.5.16.2	Implementation into Cobol	3-50
3.5.17	Modules and programs	3-50
3.5.17.1	Implementation of the <i>Case Analysis</i> refinement law into C	3-50
3.5.17.2	Implementation into Cobol	3-51
3.6	Summary	3-52
4	Refinement: From UML to non-object-oriented implementation languages	4-1
4.1	Introduction	4-1
4.2	UML in general	4-2
4.2.1	Background	4-2
4.3	Modelling with UML	4-3
4.3.1	Class diagrams	4-3
4.3.2	Relationships	4-3
4.3.2.1	Association	4-3
4.3.2.2	Aggregation	4-4
4.3.2.3	Inheritance	4-4
4.3.2.4	Generalisation	4-6
4.3.2.5	Dependencies	4-6
4.3.2.6	Refinement	4-7
4.3.3	Packages	4-7
4.3.4	Instantiates	4-8

4.3.5	Use Case Diagram	4-8
4.3.6	Interaction Diagram	4-10
4.3.6.1	Sequence Diagram	4-10
4.3.6.2	Collaboration Diagram	4-11
4.3.7	State Diagram	4-11
4.3.8	Component Diagram	4-13
4.3.9	Deployment Diagram	4-14
4.3.10	CRC Cards	4-14
4.3.11	Object Message Diagram	4-16
4.4	Refinement: Translating a design into an implementation	4-16
4.4.1	Translating classes into data structures	4-19
4.4.1.1	Implementation into C	4-19
4.4.1.2	Implementation into Cobol	4-20
4.4.2	Passing arguments to methods	4-20
4.4.2.1	Implementation into C	4-20
4.4.2.2	Implementation into Cobol	4-21
4.4.3	Allocating objects	4-22
4.4.3.1	Implementation into C	4-22
4.4.3.2	Implementation into Cobol	4-23
4.4.4	Implementing inheritance	4-24
4.4.4.1	Implementation into C	4-24
4.4.4.2	Implementation into Cobol	4-25
4.4.5	Implementing method resolution	4-26
4.4.5.1	Implementation into C	4-26
4.4.5.2	Implementation into Cobol	4-28
4.4.6	Implementing associations	4-29
4.4.6.1	Mapping associations to pointers	4-29
4.4.6.2	Implementing association objects	4-29
4.4.6.3	Implementation into C	4-30
4.4.6.4	Implementation into Cobol	4-30
4.4.7	Dealing with concurrency	4-31
4.4.8	Encapsulation	4-31
4.4.8.1	Implementation into C	4-31
4.4.8.2	Implementation into Cobol	4-31
4.5	Comparison between C and Cobol	4-31
4.6	Summary	4-33
5	From Specification through refinement to implementation: Z and UML compared	5-1
5.1	Introduction	5-1
5.2	A general comparison: Z and UML	5-2
5.3	A comparison of Z and UML in terms of specification, refinement and implementation	5-3
5.3.1	Specification, refinement and implementation in general	5-4
5.3.1.1	UML applications	5-4
5.3.1.2	Z applications	5-4
5.3.1.3	Comparison	5-5
5.3.2	Implementing Z states and UML classes	5-5
5.3.2.1	UML applications	5-5
5.3.2.2	Z applications	5-5

5.3.2.3	Comparison	5-5
5.3.3	Implementing a Z state invariant and a UML class invariant	5-6
5.3.3.1	UML applications	5-6
5.3.3.2	Z applications	5-6
5.3.3.3	Comparison	5-6
5.3.4	Implementing associations (relationships)	5-6
5.3.4.1	UML applications	5-6
5.3.4.2	Z applications	5-6
5.3.4.3	Comparison	5-6
5.4	Classes and subclasses: Z and UML compared	5-7
5.4.1	Specifications in UML	5-7
5.4.2	Specifications in Z	5-7
5.4.3	Verification and Refinement	5-9
5.4.3.1	Verification and Refinement (Z)	5-9
5.4.3.2	Verification and Refinement (UML)	5-10
5.4.4	Implementation into C	5-10
5.4.4	Implementation into Cobol	5-13
5.4.6	Comparison	5-14
5.4.6.1	Specification	5-14
5.4.6.2	Verification and Refinement	5-14
5.4.6.3	Implementation	5-14
5.5	Operations: Z and UML compared	5-15
5.5.1	Specifications in UML	5-15
5.5.2	Specifications in Z	5-16
5.5.3	Verification and Refinement	5-17
5.5.4	Implementation into C	5-26
5.5.5	Implementation into Cobol	5-31
5.5.6	Comparison	5-33
5.5.6.1	Specification	5-33
5.5.6.2	Verification and Refinement	5-33
5.5.6.3	Implementation	5-33
5.5.6.4	Conclusion	5-33
5.6	Relationships: Z and UML compared	5-34
5.6.1	One-to-one relationships	5-35
5.6.1.1	Specifications in UML	5-35
5.6.1.2	Specifications in Z	5-35
5.6.1.3	Verification and Refinement	5-36
5.6.1.4	Implementation into C	5-37
5.6.1.5	Implementation into Cobol	5-39
5.6.1.6	Comparison	5-40
5.6.1.6.1	Specification	5-40
5.6.1.6.2	Verification and Refinement	5-41
5.6.1.6.3	Implementation	5-41
5.6.1.6.4	Conclusion	5-42
5.6.2	One-to-many and many-to-one relationships	5-42
5.6.2.1	Specifications in UML	5-42
5.6.2.2	Specifications in Z	5-43
5.6.2.3	Verification and Refinement	5-43
5.6.2.4	Implementation into C	5-44
5.6.2.5	Implementation into Cobol	5-47

5.6.2.6	Comparison	5-48
5.6.2.6.1	Specification	5-48
5.6.2.6.2	Verification and Refinement	5-49
5.6.2.6.3	Implementation	5-49
5.6.2.6.4	Conclusion	5-50
5.6.3	Many-to-many relationships	5-50
5.6.3.1	Specifications in UML	5-50
5.6.3.2	Specifications in Z	5-51
5.6.3.3	Verification and Refinement	5-51
5.6.3.4	Implementation into C	5-52
5.6.3.5	Implementation into Cobol	5-55
5.6.3.6	Comparison	5-56
5.6.3.6.1	Specification	5-56
5.6.3.6.2	Verification and Refinement	5-56
5.6.3.6.4	Conclusion	5-57
5.6.4	Resolved many-to-many relationships	5-57
5.6.4.1	Specifications in UML	5-57
5.6.4.2	Specifications in Z	5-57
5.6.4.3	Verification and Refinement	5-58
5.6.4.4	Implementation into C	5-61
5.6.4.5	Implementation into Cobol	5-61
5.6.4.6	Comparison	5-63
5.6.4.6.1	Specification	5-63
5.6.4.6.2	Verification and Refinement	5-64
5.6.4.6.3	Implementation	5-64
5.6.4.6.4	Conclusion	5-64
5.7	Summary and conclusions	5-65
6	From specification through refinement to implementation for object-oriented systems: Z and UML compared	6-1
6.1	Introduction	6-1
6.2	An object-oriented approach to modeling systems	6-1
6.2.1	Terminology	6-1
6.2.2	Object-oriented methods, analysis and design	6-2
6.2.3	The object-oriented versus structured paradigm	6-5
6.3	Object-oriented Refinement	6-6
6.3.1	Refining classes	6-6
6.3.2	Refining modules	6-7
6.3.3	Functional refinement	6-7
6.3.4	Object refinement	6-8
6.4	Modelling with object-Z	6-8
6.4.1	Overview of Object-Z	6-9
6.4.2	Moving from conventional Z to Object-Z	6-11
6.5	Overview of C++	6-13
6.6	Classes: Object-Z and UML compared	6-15
6.6.1	Specifications in UML	6-15
6.6.2	Specifications in Object-Z	6-16
6.6.3	Verification and Refinement	6-16
6.6.3.1	Verification (Object-Z)	6-16
6.6.3.2	Refinement (Object-Z)	6-17

6.6.3.3	Verification and Refinement (UML)	6-17
6.6.4	Implementation into C++	6-17
6.6.5	Comparison	6-19
6.6.5.1	Specification	6-19
6.6.5.2	Verification and Refinement	6-20
6.6.5.3	Implementation	6-20
6.6.5.4	Conclusion	6-20
6.7	Operations: Object-Z and UML compared	6-21
6.7.1	Specifications in UML	6-21
6.7.2	Specifications in Object-Z	6-21
6.7.3	Verification and Refinement	6-22
6.7.3.1	Refinement (UML)	6-22
6.7.3.2	Verification and Refinement (Object-Z)	6-22
6.7.4	Implementation into C++	6-35
6.7.5	Comparison	6-37
6.7.5.1	Specification	6-37
6.7.5.2	Verification and Refinement	6-37
6.7.5.3	Implementation	6-38
6.7.5.4	Conclusion	6-38
6.8	Using Inheritance	6-38
6.8.1	Specifications in UML	6-38
6.8.2	Specifications in Object-Z	6-40
6.8.3	Verification and Refinement	6-40
6.8.3.1	Verification and Refinement (Object-Z)	6-40
6.8.3.2	Verification and Refinement (UML)	6-40
6.8.4	Implementation into C++	6-41
6.8.5	Comparison	6-44
6.8.5.1	Specification	6-44
6.8.5.2	Verification and Refinement	6-44
6.8.5.3	Implementation	6-45
6.9	Associations	6-45
6.9.1	Specifications in Object-Z and UML	6-45
6.9.2	Implementation into C++	6-45
6.9.3	Comparison	6-49
6.10	Summary and conclusions	6-50
7	Case Study: From specifications in UML and Object-Z to implementation in Object-Oriented Cobol: The ITEM system	7-1
7.1	Introduction	7-1
7.2	The next generation of Cobol	7-2
7.3	The Micro Focus Cobol compiler	7-3
7.4	The ITEM system	7-4
7.4.1	Introduction	7-4
7.4.2	Phase 1: Analysis and Modelling	7-5
7.4.2.1	A brief description of the ITEM system	7-5
7.4.2.2	Next Step – Examination of Candidate Classes	7-5
7.4.2.3	Next Step – Use-Case Scenarios	7-6
7.4.2.4	Next Step – Class Diagrams	7-6
7.4.3	Phase 2: High-Level Design	7-6
7.4.3.1	Adding a Control Class	7-6

7.4.3.3	CRC Cards	7-7
7.4.4	Phase 3: Low-Level Design	7-7
7.4.4.1	Phase 3 Deliverables	7-7
7.4.4.2	Responsibilities of the Driver	7-7
7.5	UML and Object-Z specifications for the ITEM system	7-9
7.5.1	Class diagram for ITEM system	7-9
7.5.1.1	Specifications in UML	7-9
7.5.1.2	Specifications in Object-Z	7-10
7.5.1.2.1	Basic types, sets, constants, choices and messages	7-10
7.5.1.2.2	Object-Z ITEM system	7-11
7.5.1.3	Operation decomposition: implementation into Object-Oriented Cobol	7-11
7.5.1.4	Comparison	7-11
7.5.2	CRC cards for the Driver	7-12
7.5.2.1	Specifications in UML	7-12
7.5.2.2	Specifications in Object-Z	7-12
7.5.2.3	Operation decomposition: implementation into Object-Oriented Cobol	7-12
7.5.2.4	Comparison	7-14
7.5.3	CRC cards for ITEM	7-14
7.5.3.1	Specifications in UML	7-14
7.5.3.2	Specifications in Object-Z	7-15
7.5.3.3	Operation decomposition: implementation into Object-Oriented Cobol	7-16
7.5.3.4	Comparison	7-21
7.5.4	Object-message Diagrams	7-21
7.5.4.1	Specifications in UML	7-21
7.5.4.1.1	Object-message diagram: normal request processing (UML)	7-21
7.5.4.1.2	Object-message diagram: invalid ITEM number (UML)	7-21
7.5.4.2	Specifications in Object-Z	7-22
7.5.4.3	Operation decomposition: implementation into Object-Oriented Cobol	7-23
7.5.4.4	Comparison	7-26
7.5.5	Object creation by Driver	7-26
7.5.5.1	Specifications in UML	7-26
7.5.5.2	Specifications in Object-Z	7-27
7.5.5.3	Operation decomposition: implementation into Object-Oriented Cobol	7-27
7.5.5.4	Comparison	7-27
7.5.6	A message trace diagram (sequence diagram)	
7.5.6.1	Specifications in UML	7-28
7.5.6.1.1	A message trace diagram: ITEM exists (UML)	7-28
7.5.6.1.2	A message trace diagram: ITEM does not exist (UML)	7-28
7.5.6.2	Specifications in Object-Z	7-29
7.5.6.3	Operation decomposition: implementation into Object-Oriented Cobol	7-30
7.5.6.4	Comparison	7-30

7.6	Validation and verification of the Object-Z diagrams	7-30
7.6.1	Verifying consistency of global definitions	7-30
7.6.2	Verifying consistency of state models	7-32
7.6.3	Verifying consistency of operations	7-34
7.7	Summary and conclusion	7-34
8	Summary and Conclusion	8-1
8.1	Introduction	8-1
8.2	Brief summary	8-1
8.3	What has been achieved?	8-2
8.4	Conclusions	8-3
8.5	Future research	8-4
8.6	Conclusion	8-6
Appendix A		A-1
Z notation		A-1
Appendix B		B-1
UML – Past, Present and Future		B-1
Appendix C		C-1
Object-Oriented Cobol programs of the ITEM system		C-1
References		R-1

List of Figures

Figure	Page
1.1 The main areas of investigation of this study	1-3
1.2 Structure of the study	1-6
2.1 Standard waterfall life cycle model	2-3
2.2 The waterfall model	2-4
2.3 Boehm's spiral model	2-5
2.4 Simplistic version of the spiral model	2-6
2.5 Fountain model	2-7
2.6 Outline Development Process	2-8
3.1 A global view of the refinement task	3-14
4.1 Class diagram	4-3
4.2 An author uses a computer	4-4
4.3 An aggregation structure	4-4
4.4 Two similar classes	4-5
4.5 Class diagram illustrating the inheritance relationship	4-5
4.6 Vehicle is a general class derived to specific classes	4-6
4.7 A dependency relationship between classes	4-7
4.8 Refinement relationship	4-7
4.9 Packages	4-8
4.10 Instantiates	4-9
4.11 Use-case diagram for an insurance business	4-9
4.12 A sequence diagram for a print server	4-10
4.13 A collaboration diagram for a printer server	4-11
4.14 A state diagram for an elevator	4-12
4.15 A component diagram showing dependencies between code components	4-13
4.16 A deployment diagram showing the physical architecture of a system	4-14
4.17 CRC cards for the driver and for ITEM	4-15
4.18 Object-message diagrams: normal request processing	4-16
4.19 Object model for a graphics editor	4-18
5.1 Subtyping in UML	5-7
5.2 Subtyping in Z	5-9
5.3 Specifications in UML	5-15
5.4.1 Registration class with its corresponding state diagram	5-27
5.4.2 Sequence diagram	5-27
5.4.3 Collaboration diagram	5-28
5.5 One-to-one relationship mandatory on SCHOOL side in UML	5-35
5.6 One-to-one relationship mandatory on SCHOOL side in Z	5-36
5.7 One-to-many relationship and its inverse (mandatory on Teacher side) in UML	5-42
5.8 One-to-many relationship and its inverse (mandatory on Teacher side) in Z	5-43
5.9 Many-to-many relationships in UML	5-50
5.10 Many-to-many relationships in Z	5-51
5.11 A resolved many-to-many relationship in UML	5-57
5.12 A resolved many-to-many relationship in Z	5-58

6.1	The operator Specify Class	6-6
6.2	Object-Z specification	6-10
6.3	The drawing system	6-14
6.4	Medicine_Catalogue class	6-16
6.5	Medicine_Catalogue class in Object-Z	6-18
6.6	UML Class	6-21
6.7.1	Class diagram with the corresponding State diagram	6-22
6.7.2	Collaboration diagram	6-23
6.7.3	Sequence diagram	6-24
6.8	Inheritance in UML	6-39
6.9	Inheritance in Object-Z	6-39
7.1	A set of class responsibilities and collaborators for the ITEM system	7-8
7.2	A simplified class diagram for the ITEM system	7-9
7.3	A more detailed class diagram for the ITEM system	7-9
8.1	The heterogeneous basis with UML concepts	8-5
C.1	Programs of the ITEM project	C-2
C.2	Compiled program messages	C-3
C.3	Program run completed	C-4

List of Tables

Table		Page
2.1	Comparison of some life cycle models	2-14
3.1	Mathematical and logical operators used by Z and C	3-28
6.1	Object examples	6-4

Chapter 1

Introduction

The problem statement and aims of this study are introduced. This chapter discusses the method of study and the specific questions (problems) investigated in this study. Towards the end an overview of each of the chapters is presented.

1.1 Introduction

This dissertation investigates the processes involved in moving from a system specified in any of UML (for both the structured and object-oriented paradigms), the formal specification language Z, or object Z to a high-level language like Cobol and C for the structured approach and object Cobol and C++ for the object oriented approach. A comparison of the advantages and disadvantages of taking any of the above routes is done and some conclusions resulting from these comparisons are drawn towards the end of this work.

The process of moving from a specification to an implementation is generally known as *refinement* and this refinement mechanism is investigated given the above starting points (i.e. UML, Z, and object Z) and final deliverables (i.e. Cobol, C, and C++).

The *Concise Oxford Dictionary* (4th edition, p.1021) presents the following definition of refinement:

Refinement *n.* Refining or being refined; fineness of feeling or taste, polished manners etc; subtle or ingenious manifestation of, piece of elaborate arrangement, (*all the refinements of reasoning, torture; a countermine was a refinement beyond their skill*); instance of improvement (*up*)on; piece of subtle reasoning, fine distinction.

The relevant phrase in the above definition is ‘instance of improvement upon’. Refinement is all about *improving* specifications.

Woodcock et al. [Woo96] describe the process of improvement as the removal of *non-determinism*, or uncertainty. Non-determinism is an approach to deliberately leave a decision open and to abandon the exact predictability of future states. Non-determinism, or under-

specification, leaves various possibilities open [Pot96]. A non-deterministic specification leaves more choice for the implementation [Sek97].

At each step, the design is refined by making decisions that transform the design into an implementation in an implementation language [Ran94].

1.2 Areas of investigation and research for this study

The main area of investigation for this study is the refinement process of the software systems development life cycle.

Sommerville [Som92] claims that 'a software specification (design specification) is an abstract description of the software which is a basis for its design and implementation'. During implementation the software design is realised as a set of programs or program units in an implementation language. (Chapter 2 presents a more detailed description of the software systems life cycle stages.)

The refinement process is investigated for specifications in Z, Object-Z (for object-oriented designs) and UML (Unified Modelling Language) for both the object-oriented and non-object-oriented (structured) paradigms. This is followed by an assessment and comparison of the differences and similarities between Z, Object-Z and UML in the specification and refinement of the systems development life cycle stages.

A case study (called the ITEM system) is presented in Chapter 7, where a system is developed from the analysis stage, specified in both Object-Z and UML and implemented in Object-Oriented Cobol.

Four main programming languages, namely C, C++, Cobol and Object-Oriented Cobol are used in this study. The reason is to illustrate the flexibility, or lack thereof, of Z, Object-Z and UML to be implemented into more than one major third generation programming language.

The primary areas of investigation and research for this study are illustrated in Figure 1.1.

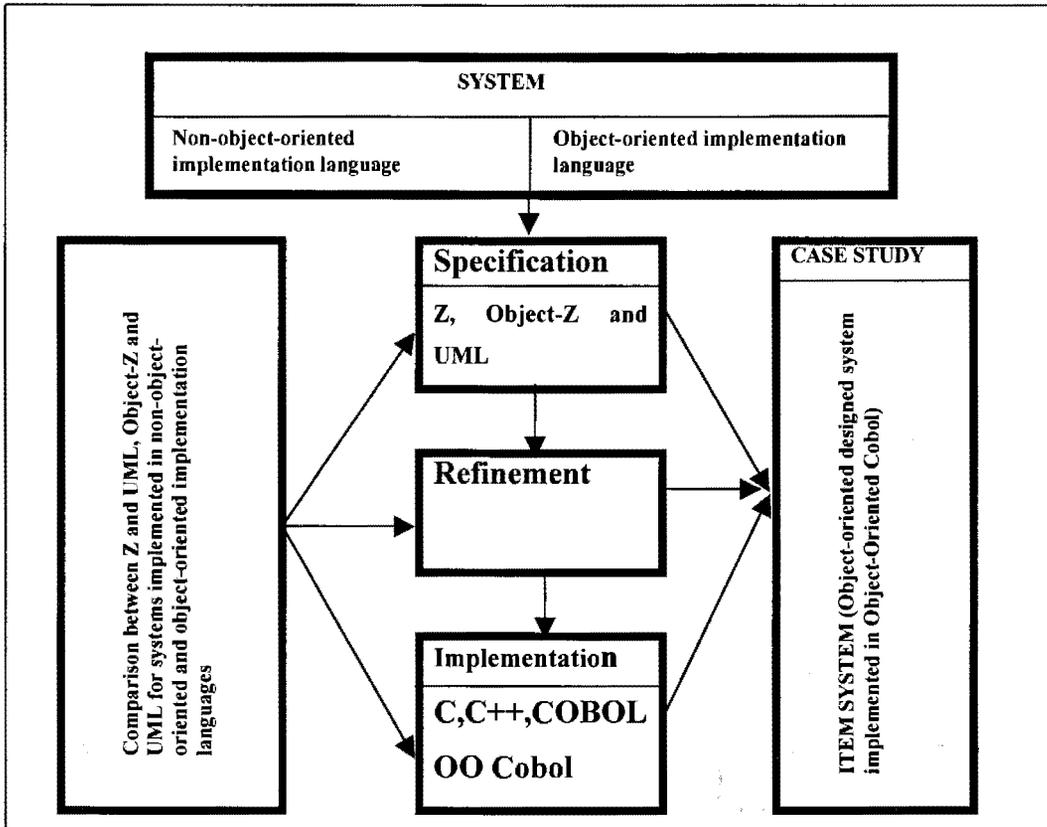


Figure 1.1 The main areas of investigation of this study

1.3 Questions posed in this study

The following questions encompass the problems investigated in this study:

1. How are systems transformed (refined) from specification methods in Z, Object-Z and UML to an implementation in languages such as C, C++, and Object-Oriented Cobol?
2. What are the differences (strengths and weaknesses) between Z, Object-Z, and UML for the specification and refinement of systems that are implemented into non-object-oriented and object-oriented implementation languages?

The first question above is addressed in chapters 3 and 4. The second question is addressed in chapters 5 and 6.

1.4 Method of study

In addressing the problems (questions) investigated in this study, we embark on the following methods:

- Exploration of the software systems development life cycles for the structured and object-oriented development approaches. This sets the scene for the more detailed exploration of the specification phase, the refinement process and the implementation phase that follow in subsequent chapters (chapters 3 to 6). The entire software development life cycle is revisited again with the ITEM system (Chapter 7), where it is taken through all the software development life cycle phases.
- Examining the use of Z (Object-Z for object-oriented systems) and UML in the specification phase, refinement process and implementation phase. The focus will be on both the non-object-oriented (structured) as well as the object-oriented paradigms.
- Comparing the strengths and weaknesses of the specification languages Z (Object-Z for object-oriented systems) and UML for the specification, refinement and implementation of systems.
- Making use of the implementation language C, C++, Cobol and Object-Oriented Cobol in this study to illustrate the implementation phase from the specification phase and refinement process.
- Consolidating the investigations of Chapters 2 to 6 in the form of a case study in Chapter 7 to illustrate the transition of a system from specification to implementation. A small system is taken from the requirements phase through to the implementation phase. The system is specified in both Object-Z and UML and thereafter implemented in Object-Oriented Cobol.

1.5 Layout of the dissertation

This study is presented in eight chapters and four appendices.

- Chapter 1 introduces the problem statement and aims of this study. It also includes the method of study and the specific questions (problems) to be investigated. An overview of the rest of the dissertation is given.
 - In Chapter 2 the software systems development life cycle for both the structured and object-oriented paradigms is presented. For the structured systems development, the waterfall life cycle model as illustrated by Royce [Roy70], Davis [Dav90], and Schach [Sch99], as well as Boehm's spiral model [Som92, Sch99], are examined. For the object-oriented systems development the fountain model [Hen90], and the outline development process [Fow97], are described and investigated. The life cycle models are compared. An overview of the life cycle development phases is given. Each life cycle model has its own particular strengths
-

and weaknesses as highlighted in Chapter 2. The position of refinement within the software systems development life cycle is indicated.

- In Chapter 3 the process of refinement from specification to implementation is demonstrated using Z as the specification language for non-object-oriented designs. A brief introduction to the Z specification language is given. The validation and verification of the specifications prior to refinement and implementation are discussed. Thereafter, the refinement of Z specifications comprising of data refinement, operation refinement and operation decomposition is investigated. The Z specifications are refined into Cobol and non-object-oriented C.
 - Chapter 4 starts off with a broad background on the UML modelling technique, followed by the illustration of the implementation of UML specifications into non-object-oriented C and Cobol.
 - In Chapter 5 a comparative study between Z and UML applications from specification through refinement to implementation for non-object-oriented implementation languages (in this case C and Cobol) is made.
 - Chapter 6 introduces Object-Z and C++. A background study on Object-Z and C++ describes some of the main features of these two languages. Object-Z and UML are compared for the specification, refinement and implementation into an object-oriented language (C++) for various aspects of design.
 - In Chapter 7 the software systems development life cycle of Chapter 2 is revisited in the form of a system, called the ITEM system, where this system is developed from the requirements phase through to the implementation phase. Specification languages used are Object-Z and UML, and the implementation language is Object-Oriented Cobol.
 - Chapter 8 concludes the study by referring back to Chapter 1 and stating the major findings and conclusions that can be drawn as a result of this study. The answers to the two specific questions investigated in this study are examined and the research objectives re-appraised. An overview of the conclusions reached for each of the chapters is presented and suggestions for future research offered.
-

Three appendices are included. Appendix A summarises some of the features of the Z notation, Appendix B focuses on UML, and Appendix C contains the programs of Chapter 7.

Figure 1.2 sets out the structure of the study and shows the relationships between chapters.

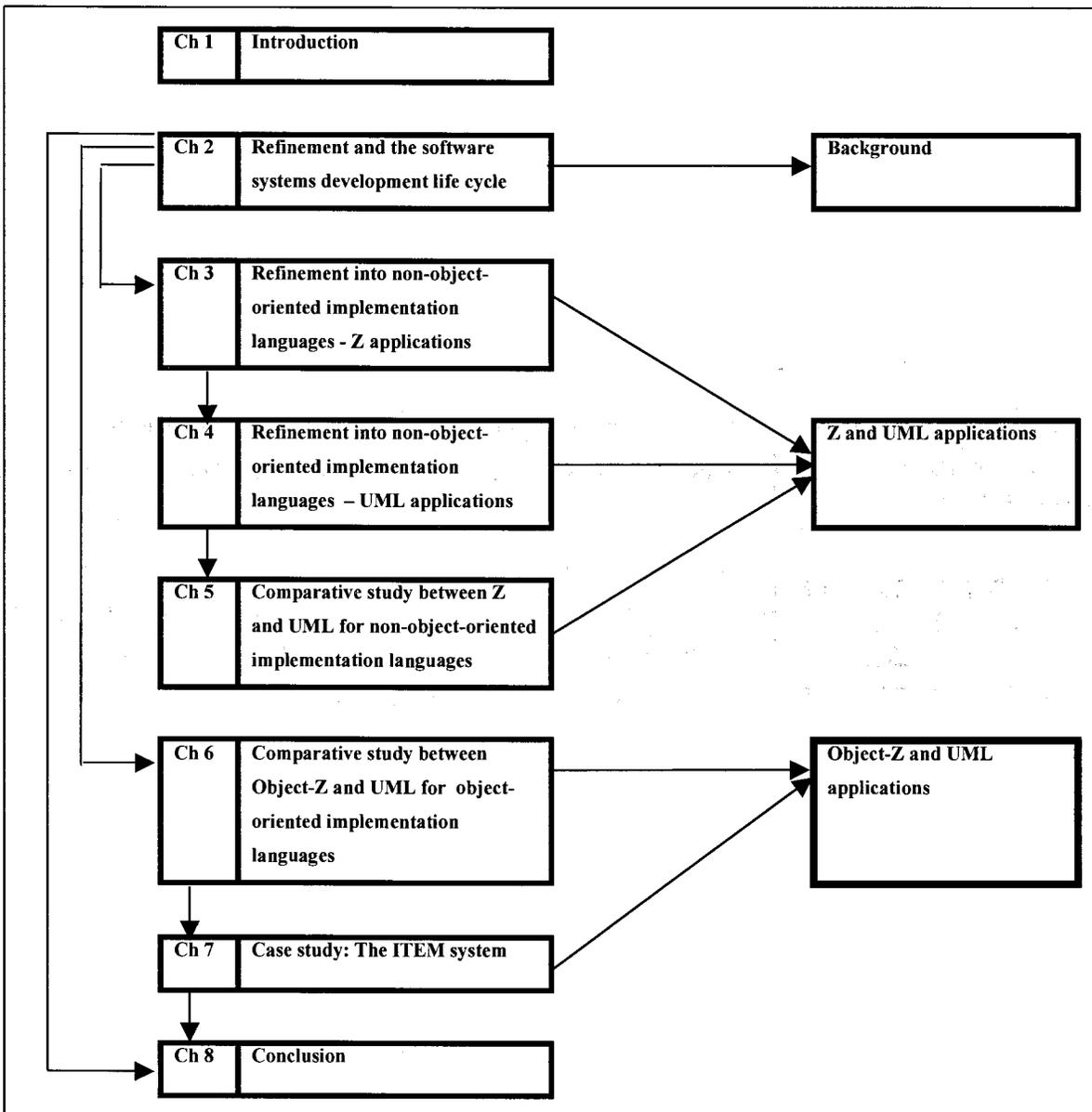


Figure 1.2 Structure of the study

1.6 Summary of main findings and results

In this study the importance of the refinement stage in the software systems development cycle is highlighted. This is because both the specification languages and the implementation languages are becoming more and more sophisticated. Equally rapid is the development of systems development tools that aid in the design, specification and implementation of software systems. This is a dynamic field of study that demands constant re-evaluation and updating.

An assessment and comparison of the transformation from specification through refinement to implementation for systems specified in specification languages Z, Object-Z and UML are made. This includes both the non-object-oriented (structured) and object-oriented paradigms. The implementation languages include C, C++, Cobol and Object-Oriented Cobol.

In the illustration and comparison of the abovementioned specification languages, the strengths, weaknesses and applicability of the languages are highlighted with regard to the appropriateness and applicability for different specifications in both non-object-oriented and object-oriented application examples.

A practical illustration of how an object-oriented system (the ITEM system) is implemented in Object-Oriented Cobol from specifications in Object-Z and UML is provided.

From our study the main conclusions are:

- For the software systems development cycle, the structured and object-oriented paradigms are each applicable and preferred for the development of structured and object-oriented systems respectively.
 - Z, Object-Z and UML each have their own strengths and weaknesses. For example, UML is more graphical in nature while Z and Object-Z are more precise mathematical specification languages.
 - UML and Object-Z are used for object-oriented systems. UML can also be used for the specifications of non-object-oriented systems, while Z is mainly used for specifying non-object-oriented systems. Although Z can also be used for the specification of object-oriented systems, Object-Z is preferred in this regard.
 - The refinement steps involved for UML and Z (Object-Z) are different. The refinement of Z and Object-Z involves data refinement, operation refinement and operation
-

decomposition. Those refinement steps are not used for the refinement of UML, unless UML is combined with a program design calculus specification language, for example OCL (object constraint language) or Z, in which case the refinement will also involve the data refinement, operation refinement and operation decomposition stages.

- In a single project, the programming problems that arise usually present a range of difficulty. A large percentage of the project may be routine, therefore no formal description other than the code itself is required. Only a portion may require specification in a formal notation such as Z. Within this portion, only a fraction might be refined to a detailed design. In this fraction only a page or two of code might be derived and verified. The rest, because it is so obvious, can be translated to code by intuition and then verified by inspection [Jak97]. In a number of examples provided in this study the specifications (Z, Object-Z and UML) are directly implemented into an implementation language without the intermediate steps of validation, verification, data refinement, operation refinement and operation decomposition.
 - C and Cobol are two non-object-oriented languages each with their own strengths and weaknesses. The ease of implementation into C and Cobol are different for different specification languages. C++ and Object-Oriented Cobol are two object-oriented languages, again with their own strengths and weaknesses. The ease of implementation from a specification language into these two object-oriented languages also differs.
 - Finally, we conclude that the tendency in specification modelling languages is towards an integration of a program design calculus and UML, that is, integrating, for example, Z (or Object-Z) and UML. This results in the combination of the best features of the more graphic (UML) and detailed (e.g. Z) specification languages. This appears to be the direction future specification modelling languages [Pol92, Ach97, Fra97, Shr97, Hai98, Pai99b] are taking.
-

Chapter 2

Refinement, specification, and implementation in the software systems development life cycle

In this chapter the software systems development life cycle for the structured and object-oriented systems are examined.

2.1 Introduction

Fowler [Fow97] motivates the need to study modelling techniques by looking at their role in the context of a life cycle model.

Modelling techniques normally use specification languages (formal or informal) to describe the properties of the proposed system. Examples of formal languages are Z [Spi92] and VDM [Jon90]. Non-mathematical languages or techniques include ERD (entity relationship diagrams) and UML (unified modelling language). Z, Object-Z and UML will be discussed in the rest of this dissertation.

In this study, we pay particular attention to Z, Object-Z and the UML modelling language. Z, Object-Z, and UML are called modelling languages and not methods (e.g. object-oriented analysis and design method (OOA&D)). Most methods consist of both a modelling language and a process. The *modelling language* is the notation that methods use to express designs, while the *process* describes the steps to take in doing a design [Fow97].

Of course, in designing a software system, the process chosen should be appropriate for the project. The modelling language is selected accordingly to record the resulting analysis and design decisions.

Therefore, we present a brief overview of some of the main software life-cycle models and processes below, before we embark on the discussions of the modelling languages. Also, the refinement of a specification cannot be studied in isolation, since the software systems life

cycle must be reviewed in its entirety to put the role of the refinement process into perspective.

Once the need for a product or system has been established, the product goes through a series of developmental phases (see e.g. Schach [Sch99]). Typically, the product is specified, designed and thereafter implemented. If the product is what the client wants, the product is installed, and while it is operational it is maintained. Once the product has reached the end of its useful life, the product is retired or decommissioned. The sequence of steps through which the product progresses is called the *life-cycle model*.

Schach [Sch99] describes the evolution of the various software life cycles:

- First the waterfall model utilising explicit feedback loops.
- The next major development was the rapid prototype model; one of its major aims was to reduce the need for iteration.
- Rapid prototyping was followed by the spiral model, which explicitly reflects an iterative approach to software development and maintenance.

Other non-object-oriented life cycle models include the build-and-fix model, incremental model and the synchronise-and-stabilise model [Dav90, Hen90, Jac92, Ber93, Boo94, Fow97].

The development of the various object-oriented models, included the fountain model [Hen90], the objectory model [Jac92], the recursive/parallel model [Ber93], the round-trip gestalt design model [Boo94] and the Outline Development Process [Fow97].

The waterfall model, spiral model, the fountain model, and the outline development process model are looked at in more detail in the remainder of this chapter.

2.2 The structured paradigm

Various life cycle models exist for the structured approach to systems development. We will briefly describe the waterfall life cycle model, followed by the spiral model and thereafter compare these two models.

2.2.1 Waterfall life cycle model

The waterfall model [Sch99] illustrates the system life cycle model. It was first put forward by Royce [Roy70]. Figure 2.1 below illustrates Royce's original model as presented by Davis [Dav90]. Figure 2.2 shows the current form of the waterfall model explicitly involving iterative and re-design processes [Sch99].

The requirements for the product (system) are determined first, followed by the drawing up of the specification, the product design, implementation and testing. Both the client and members of the software quality assurance (SQA) group check, after each stage, the documentation. Once the client agrees that the product satisfies its specification, the product is installed, commissioned and handed over to the client [Sch99].

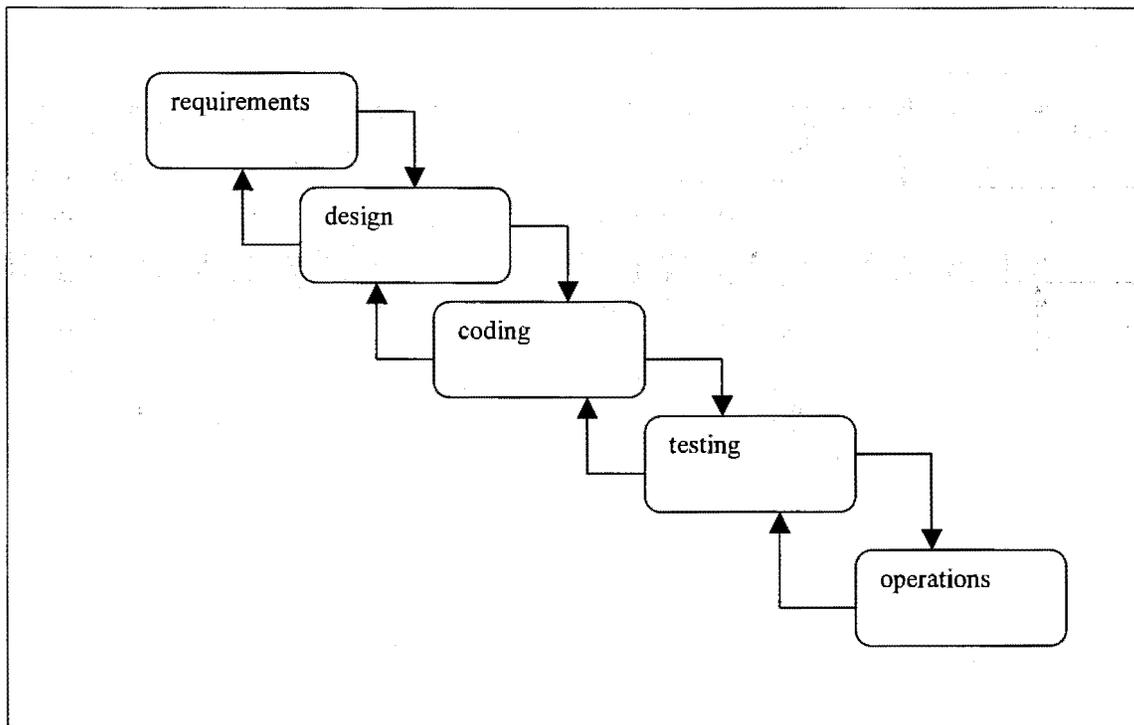


Figure 2.1 Standard waterfall life cycle model [Dav90, p.8]

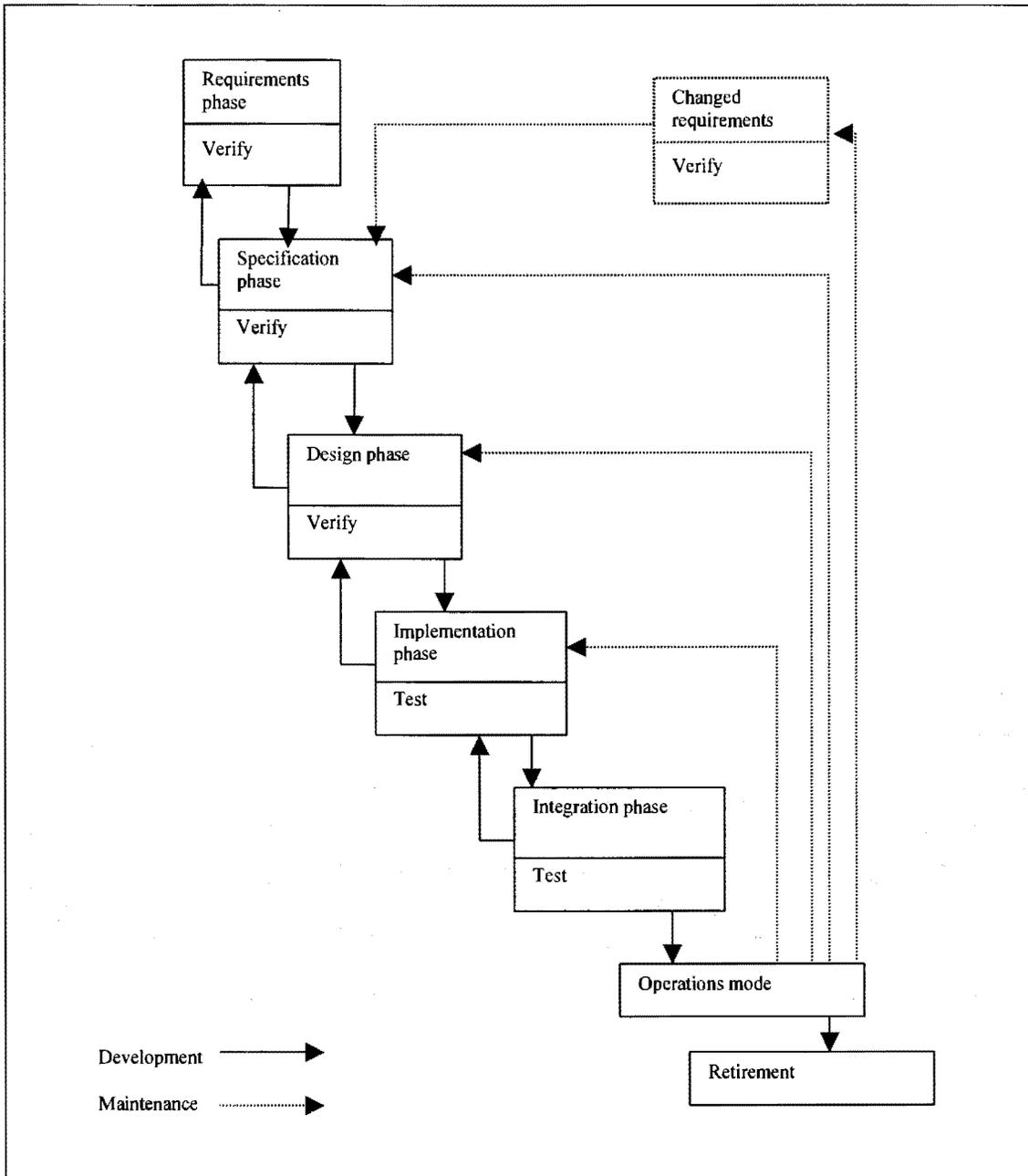


Figure 2.2 The waterfall model [Sch99, p.66]

The waterfall life cycle model is accepted as the standard reference life cycle model for the purposes of this study because it covers all the phases that are dealt with in this study. Note that the refinement process takes place after the specification phase and before the implementation phase.

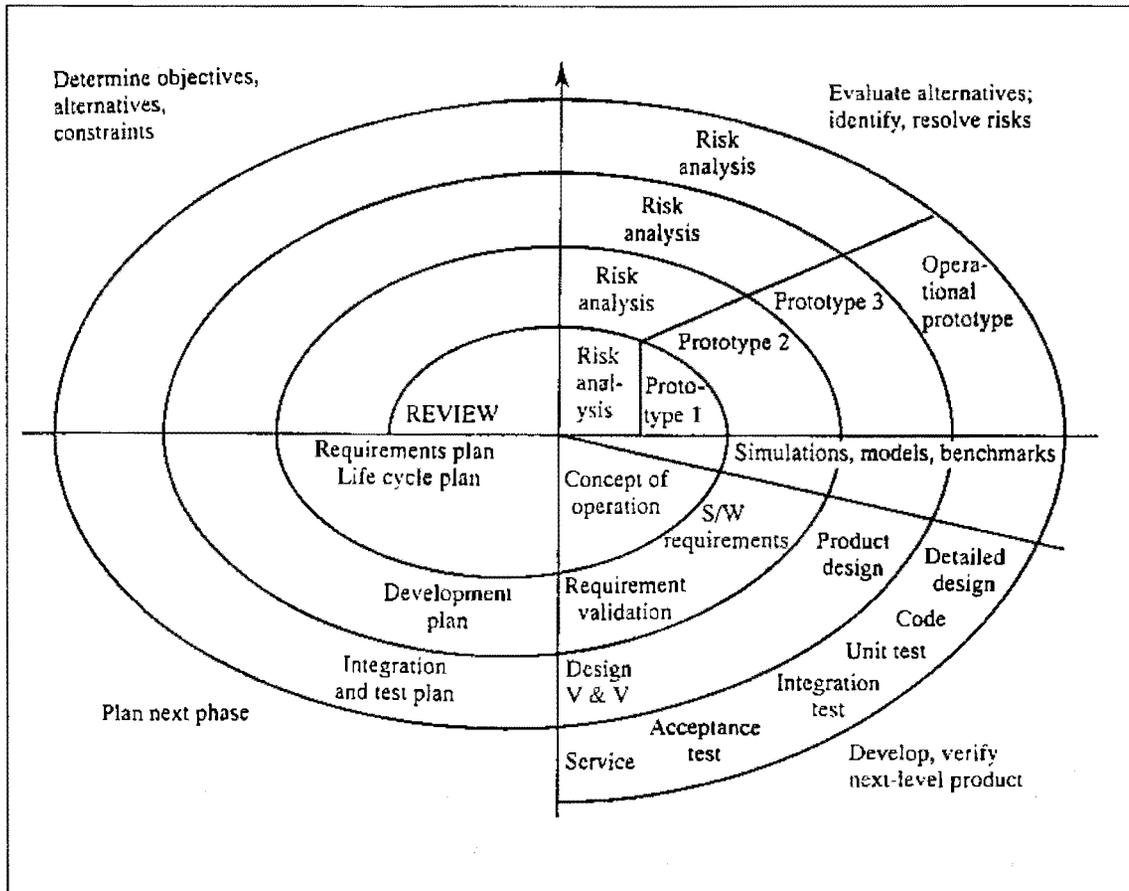


Figure 2.3 Boehm's spiral model [Som92, p.15]

2.2.2 Boehm's spiral model

The spiral model [Boe88] is an improved process that subsumes a number of generic models (e.g. waterfall, rapid prototyping, etc.).

Management risk items are assessed at regular stages in the project and the initiation of actions to counteract these risks. A risk can roughly be defined as something that can go wrong. A risk analysis is initiated before each cycle. A review procedure assesses whether to move on to the next cycle of the spiral at the end of each cycle [Som92, Sch99].

A development model for the system is chosen after risk evaluation. This risk evaluation determines whether the delivered product will satisfy the client's needs or not. Prototyping may be used in one spiral to resolve requirements risk and this may be followed by a conventional waterfall process model development [Som92, Sch99].

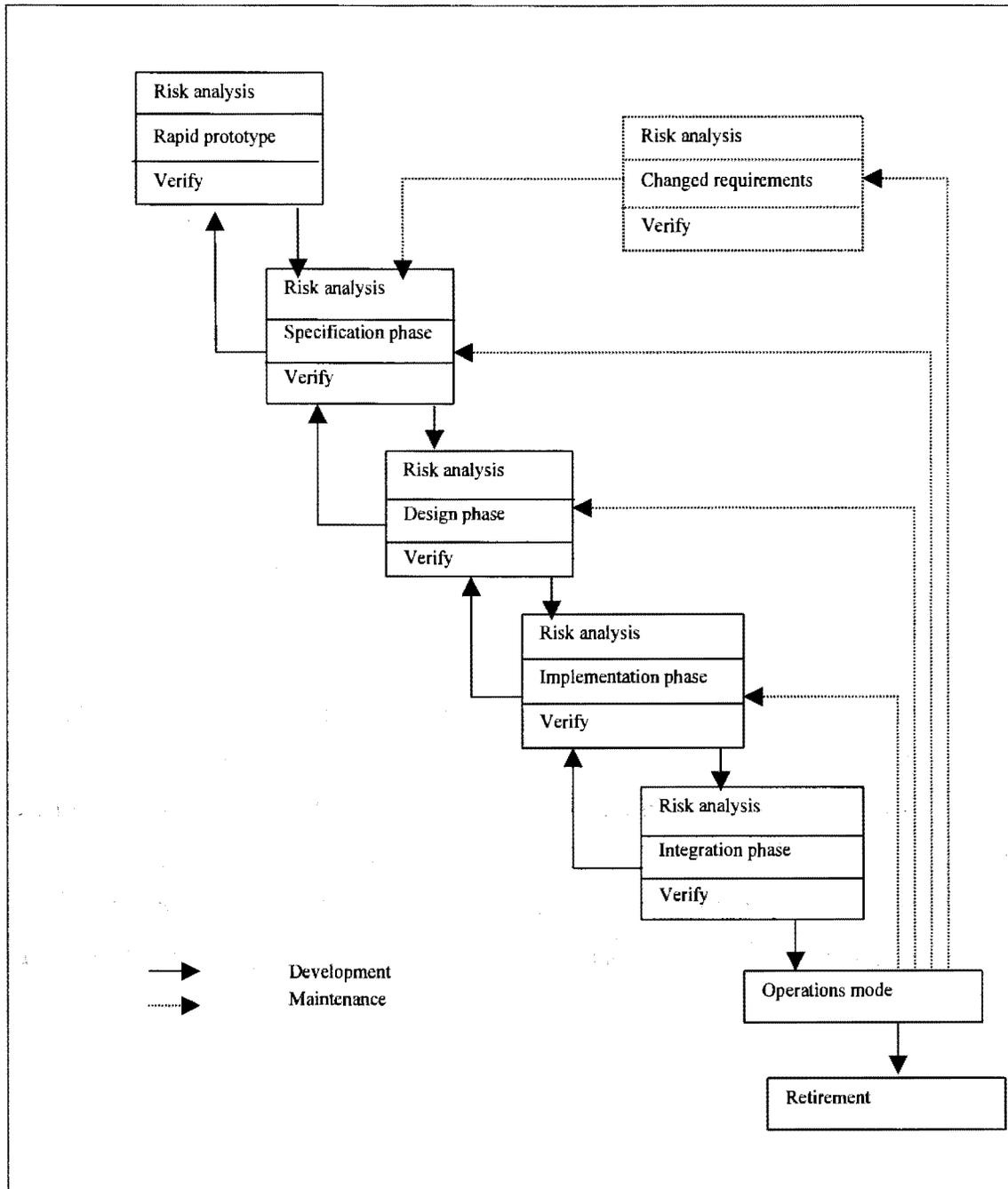


Figure 2.4 Simplistic version of the spiral model [Sch99, p.79]

Figure 2.4 shows that a simplistic way of looking at the spiral model is to structure it in a similar way as the waterfall model with each phase preceded by risk analysis [Sch99].

For the spiral model, as for the waterfall model, the refinement process is a life cycle process embarked on before the implementation phase.

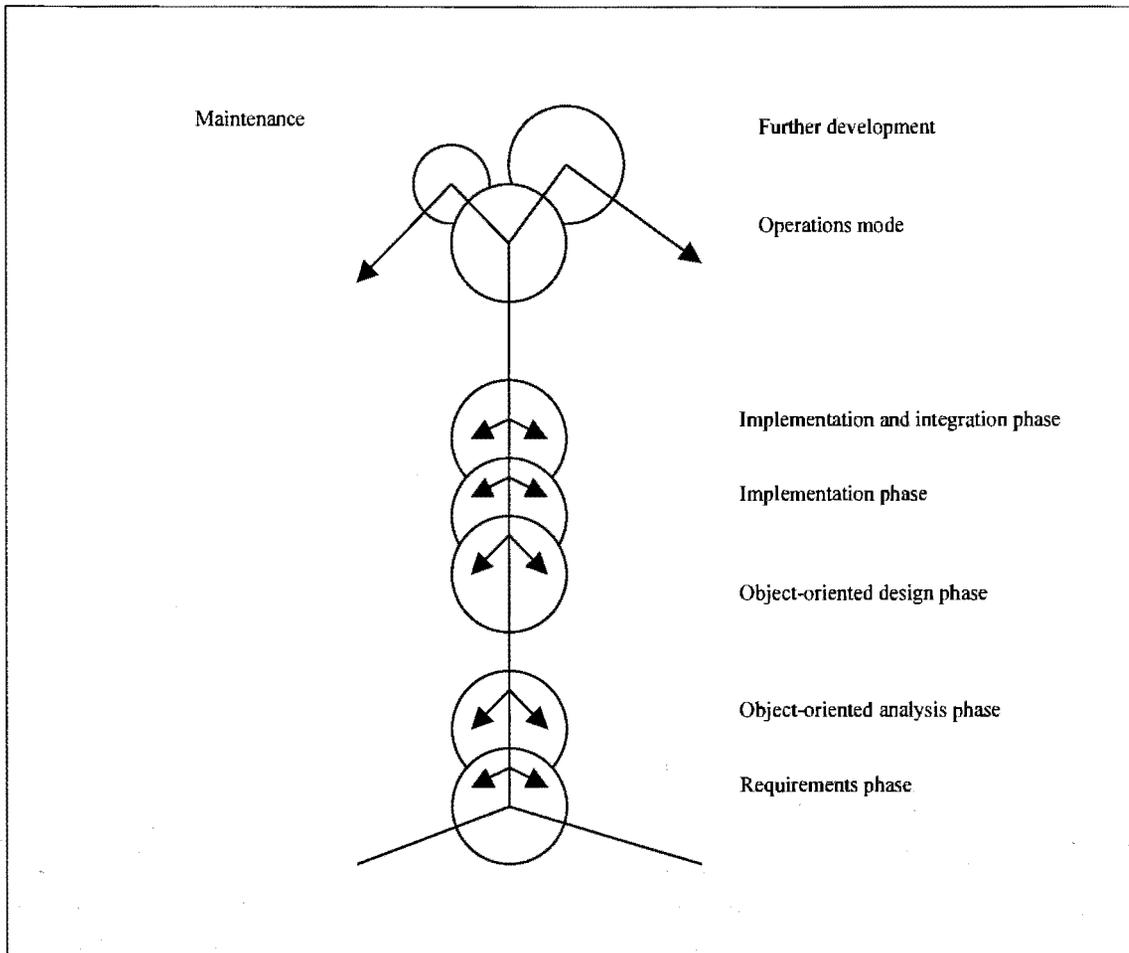


Figure 2.5 Fountain model [Hen90]

2.3 The Object-oriented paradigm

A number of life cycle models exist for the object-oriented approach to systems development. We briefly describe the fountain model, followed by the outline development process.

2.3.1 Fountain model

Schach [Sch99] states that the need for iteration between phases or portions of phases of the process appears to be more common with the object-oriented paradigm than with the structured paradigm. Various object-oriented life cycle models that reflect the need for iteration have been proposed. An example is the fountain model (Figure 2.5) by Henderson-Sellers and Edwards [Hen90].

The circles that represent the various phases overlap and therefore explicitly reflect overlapping between activities. The arrows within a phase represent iteration

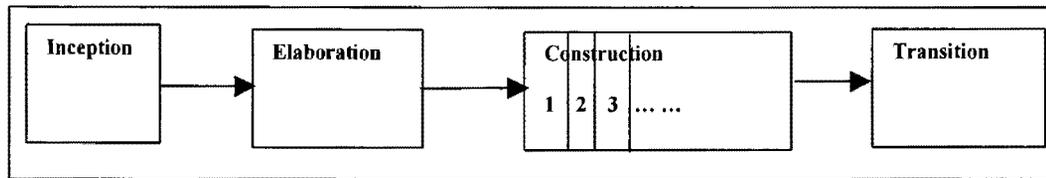


Figure 2.6 Outline Development Process

The maintenance circle is smaller to symbolise a reduced maintenance effort when using the object-oriented paradigm.

As with the structured model, the refinement process for the object-oriented model is located between the design phase and the implementation phase. Strictly speaking, just prior to the refinement phase is the specification phase, as with the waterfall model. Note that the refinement process is not indicated in the fountain model.

Other object-oriented life cycle models are: the objectory life cycle (Jacobson et al. [Jac92]), the recursive/parallel life cycle (Berhard [Ber93]), the round-trip gestalt design (Booch [Boo94]), and the outline development process (Fowler [Fow97]).

Booch [Boo94] says that a round-trip gestalt design is a style of design that emphasises the incremental and iterative development of a system, through the refinement of different yet consistent logical and physical views of the system as a whole, the process of object-oriented design is guided by the concepts of round-trip gestalt design; round-trip gestalt design is a recognition of the fact that the big picture of a design affects its details, and that the details often affect the big picture [Boo94, p.518].

2.3.2 Outline development process

The outline development process (Figure 2.6) is iterative and incremental in nature, since the software is developed and released in pieces.

The first two phases of the Outline Development Process are inception and elaboration. During *inception*, the business rationale for the project is established and the scope of the project is decided. In the *elaboration* phase, more detailed requirements are collected. Also,

a high-level analysis and design is done to determine a baseline architecture, and the plan for construction is established [Fow97].

The *construction* phase consists of a number of *iterations*, in which every iteration builds production-quality software, satisfying a subset of the requirements of the project. Every iteration contains all the usual life-cycle *phases* of analysis, design, implementation, and testing. Within an iteration, the refinement process takes place before the implementation phase.

In the *transition* phase, the final work is done, for example, beta testing, performance tuning, and user training.

2.4 Overview of phases

A brief overview of the phases of the general software life cycle models as illustrated by Figures 2.1 to 2.4 is presented below:

2.4.1 Requirements definition phase

The developers determine the needs of the client and the constraints that exist on the proposed system [Sch99]. Davis [Dav90] states that the software requirements phase includes an analysis of the problem and concludes with a complete specification of the desired external behaviour of the system to be built. He combines the requirements and specification phases outlined by Schach [Sch99] into one phase, namely *software requirements*.

2.4.2 Requirements specification phase

Once the developers understand the requirements, the specification document is drawn up (Schach [Sch99]). This document explicitly describes the functionality of the product and must satisfy two mutually contradictory requirements. It must be clear and intelligible to the client, who is probably not a computer expert. It must also be comprehensive since it is virtually the sole source of information available to the design team for drawing up the design.

According to Wordsworth [Wor92], a specification describes:

- the state on which the program is to operate
-

- the inputs to be supplied by the caller of the program
- the outputs to be returned by the program
- the preconditions that the writer of the program can assume
- the post-conditions that the outputs and the final state must satisfy.

When a detailed system specification is derived, some design activities are necessary to structure the specification and to establish its feasibility. A specification is produced at a number of different levels, from an abstract requirements definition through a contractual requirements specification to a detailed software specification which is sometimes seen as the first stage in the design process (see e.g. [Som92]).

The specification document can be an informal document written in, for example, English, or it can be written in a formal specification language, e.g. Z [Spi92], or Vienna Definition Method (VDM) [Jon90].

Sommerville [Som92] defines a *requirements definition* to be a natural language statement of what user services the system is to provide. A follow-up document, namely the *requirements specification*, sets out the system services in more detail.

A formal specification gives an unambiguous description of the behaviour of the proposed system. The specification can be reasoned about formally and validated to check that it possesses certain desirable properties, and that undesirable consequences cannot be derived from it [Woo96, Rat94].

2.4.3 Design phase

The design phase can be divided into two sub phases, the preliminary design and the detailed design [Sch99, Dav90].

- *Preliminary design*: The software system is iteratively decomposed in a top-down manner until the leaves of the resulting design tree are small enough to map into a manageable number of lines of code. Each such module is documented in terms of its inputs, outputs, functions, high-level design, architectural design and functional design.
 - *Detailed design*: This design defines and documents algorithms for each module in the design tree that will be realised as code (i.e. the design of the program).
-

The design process involves describing the system at a number of different levels of abstraction. As the design is decomposed, errors and omissions in earlier stages are discovered and earlier design stages are then refined by this feedback. The output of each design activity is a specification, which can be an abstract, formal specification that clarifies the requirements, or a specification of how part of the system is to be realised [Som92].

More and more detail is added to the specification as the design progresses. This leads to the ultimate outputs that are specifications of algorithms and data structures to be used as a basis for system implementation.

2.4.4 Implementation phase

During the implementation phase, the various component modules of the design are coded into a set of programs or program units. In other words, implementation is the process that takes a design and transforms it into actual code, ensuring conformity that is judged against the design. This is usually performed in two steps:

- The conversion of the algorithm into a high-level language, such as Cobol, C, C++, and Java, etc.
- The conversion of the high-level language into a machine language (i.e. compilation).

2.4.5 Integration phase

During this phase the modules are combined and it is determined whether the product as a whole functions correctly. It is recommended that implementation and integration be performed in parallel [Sch99].

Davis [Dav90] refers to integration as the testing phase. During this phase (see Figure 2.2), each module is checked to ensure that it behaves according to its specification. During system testing, the entire (i.e. fully integrated) software system, embedded in its actual hardware environment, is tested to ensure that it is a reasonable reflection of the requirements specification. Following final testing, the system becomes operational and is delivered to the customer.

The testing stages referred to above are to uncover and remove errors. Unit testing checks each coded module for the presence of errors. The integration testing interconnects suites of previously tested modules to ensure that these behave as well as they did as independently

tested modules. Then finally system testing checks that the fully integrated software system embedded in its actual hardware environment behaves according to the Requirements Specification.

2.4.6 Maintenance phase

Once the client has accepted the product, any changes constitute maintenance. These also involve correcting of errors not discovered in earlier stages of the life cycle.

Maintenance involves the continued detection and correction of errors after deployment, while the enhancement of a product (system) is the addition of new features. The maintenance and enhancement phases are actually full development life cycles. The reason is that if a *coding* change is made, then the coding and subsequent testing stages must be performed. However, if a *design* change is made, then the design, coding and testing stages must be performed. When a *requirement* change has occurred, then all the stages must be performed [Dav90].

2.5 Refinement process

Refinement is the process that takes place between the specification and the implementation phases. The refinement process is in general not indicated in any of the traditional models.

The refinement process is generally defined (informally) as the process of moving from the specification towards the implementation language. In other words, a specification is developed in such a way that it leads towards a suitable implementation. Refinement can enable development decisions to be verified formally (see e.g. Woodcock and Davies [Woo96], Back and Von Wright [Bac97], etc.).

Barden, Stepney and Cooper [Bar94] define refinement as the process of turning an *abstract* specification into a more *concrete* specification (see Chapter 3 of this dissertation for the definitions of abstract and concrete specifications).

2.6 Comparison of traditional life cycle models

There are many similar and overlapping characteristics between the various life cycle models. Some differences can also be highlighted.

The need for iteration between phases or portions of phases of the life cycle appears to be more common with the object-oriented paradigm than with the structured paradigm.

The idea of minimising risk via the use of prototypes and other means is the concept underlying the spiral model [Boe88]. This life-cycle model can be viewed as a waterfall model with each phase preceded by risk analysis. Before each phase is commenced, an attempt is made to control (or resolve) the risk. If all the risks cannot be resolved at that stage, the project is terminated [Sch99, Som92].

Prototypes are used in the spiral model to provide information about certain classes of risk. If all risks are successfully resolved, the next step in the development process is started. The phases that follow may correspond to the pure waterfall model. The results of each phase are evaluated and the next phase planned [Sch99, Som92].

Inherent in every phase of the waterfall model is testing. Testing proceeds continuously throughout the process. Of course, spending too much time on testing could be a waste of money, and the delivery of the product delayed. If too little testing is performed, the delivered software may contain residual faults. The spiral model with its inherent risk-analysis of each phase avoids therefore the problems of too much or too little testing [Sch99].

Within the structure of the spiral model, maintenance is another cycle of the spiral. There is no distinction between maintenance and development. Maintenance is treated the same way as development. For the waterfall model, once the client has accepted the product, any changes to the product constitute maintenance [Sch99]. For the fountain model, because the various phases overlap, the maintenance effort is reduced when the object-oriented paradigm is used [Hen90].

For the spiral model the emphasis on alternatives and constraints supports the reuse of existing software and the incorporation of software quality as a specific objective [Sch99].

The spiral model also has its restrictions on applicability. In its present form the model is intended exclusively for internal development [Boe88]. A second restriction of the spiral model is that it is only applicable to large-scale software [Boe88, Sch99].

All the features of the other life cycle models are incorporated in the Outline Development Process, because the construction phase of the Outline Development Process consists of many iterations, each of which contains all the usual life cycle phases of analysis, design, implementation, and testing [Fow97] (refer to Section 2.3.1).

The following table contains a summary of some of the main differences, weaknesses and strengths of the waterfall model, the spiral model and the object-oriented model. The object-oriented model of the table refers to generic characteristics of the fountain model, the objectory life cycle model, the recursive/parallel life cycle model, the round-trip gestalt design and the outline development process.

Life cycle model	Strengths	Weaknesses
Waterfall model	<ul style="list-style-type: none"> • Disciplined approach • Document-driven 	<ul style="list-style-type: none"> • Delivered product may not meet client's needs
Spiral model (Incorporates features of several models including the waterfall model)	<ul style="list-style-type: none"> • The emphasis on alternatives and constraints supports the reuse of existing software • Disciplined approach • Document-driven • Ensures that delivered product meets client's needs • Maximises early return on investment • Promotes maintainability • Future user's needs are met • Ensures components can be successfully integrated 	<ul style="list-style-type: none"> • Can be used only for large-scale, in-house products • Developers have to be competent in risk analysis and risk resolution
Object-oriented models (Fountain model, the objectory life cycle, the recursive/parallel life cycle, the round-trip gestalt design and the outline development process)	<ul style="list-style-type: none"> • Supports iteration within phases, parallelism between phases • Can be used for in-house or external users. • Amenable to systems developed in the object-oriented paradigm. 	<ul style="list-style-type: none"> • May degenerate into CABTAB (code a bit, test a bit)

Table 2.1 Comparison of some life cycle models

2.7 Summary and conclusion

In this chapter the waterfall model [Roy70], as described by Schach [Sch99] and Davis [Dav90], and the spiral life cycle model [Boe88] were discussed for the non-object-oriented paradigm. For the object-oriented paradigm the fountain model [Hen90] and the Outline Development Process as described by [Fow97] were examined.

The position of refinement within these software development life cycles was indicated. It can be concluded that each life cycle model is suited for different types of systems and circumstances. To decide whether or not to use a particular model for a project, it is necessary to understand both its strengths and weaknesses. Until the early 1980s the waterfall model was the only widely accepted life cycle model.

Although the object-oriented paradigm is nowadays often the preferred model for system development (see e.g. [Sch99]), a problem is that frequent iterations and refinements are often required. However, as the object-oriented paradigm matures, the need for repeated review and revision could decrease. The necessity for iteration between life cycle phases appears to be more common with the object-oriented paradigm than with the structured paradigm.

In Chapter 7 the software system life cycle is revisited with an object-oriented design of the ITEM system, illustrating the process as it moves from requirements to implementation.

Chapter 3

Refinement: Non-object-oriented systems using Z

This chapter examines the process of refinement for Z applications. This includes the data refinement, operation refinement and operation decomposition. Prior to the refinement process the specification is validated and verified.

3.1 Introduction

In this chapter the process of refinement from specification to implementation is demonstrated using Z as the specification language for non-object-oriented designs. Z can be classified as a formal specification language.

This chapter starts with a brief introduction to Z, followed by a discussion of the verification and the validation of specifications prior to refinement.

We conclude with a discussion on the refinement of Z specifications comprising data refinement, operation refinement and operation decomposition for basic constructs, considering each construct in isolation and not in combination with other constructs.

3.2 Z specifications

3.2.1 Introduction

Z is perhaps the most widely used formal language of its type, and has been successful, especially on large-scale projects [Ran94, Rat94]. Writing Z specifications requires the specifier to be mathematically precise, with the result that there are fewer ambiguities, contradictions, and omissions than with informal specifications. The fact that Z is a formal language allows a developer to reason about the correctness of the specification when necessary. The use of Z has often decreased the cost of software development [Sch99].

Because Z is a specification language with an explicit semantics, every construct used in Z has a definite interpretation and the interpretation is given unambiguously [Spi88, Kot97]. Z defines a set of built-in mathematical data types to model the data in the system. These data

types are not oriented towards a specific implementation language, but adhere to a rich collection of mathematical laws that lend themselves to an effective reasoning about the way a specified system will behave.

According to Kotzé [Kot97] the following illustrates the elegance of Z:

- A fixed interpretation of data types that promotes communication of ideas between people.
- Because of the built-in types, a more concise and economical specification is possible.
- Z objects are interpreted as objects in a typed, constructive set theory.
- The view that functions are explicitly to be represented by their graphs.
- Predicates define subsets.

A Z specification consists of a number of schemata. Each schema consists of a group of variable declarations, and a list of predicates that constrains the possible values of the variables. Z lends itself to a highly modified form of specification via the *schema* construct. Both the static and the dynamic aspects of a system can be described by schemas [Cra91, Spi92, Kot97].

The static aspects include [Cra91, Spi92, Kot97]:

- The states that a system can have.
- The state invariant relationship that must be maintained as the system moves from state to state.

The dynamic aspects include:

- The operations that can be performed on the system.
- The relationship between the inputs and outputs of operations.
- The state changes that can or are allowed to take place.

The *state invariant* is a statement about what is always true of the state before and after every valid operation. This is made up of predicates which are described both explicitly (beneath the divider line of a schema) and implicitly through type declarations. An *abstract data type* is a set of values that a variable of that type can take and a set of operations that can be carried out on the values [Ran94].

A Z specification contains state and operator schemas, as well as axiomatic descriptions. Each of these aspects is discussed in more detail below.

More information on Z can be found in Spivey [Spi92], Woodcock et al. [Woo96], Ratcliff [Rat94], Potter et al. [Pot96], Wordsworth [Wor92], Kotzé [Kot97], Rann et al. [Ran94], and Jacky [Jak97]. Appendix A provides a glossary of the Z notation used in this study.

3.2.2 Discussion

3.2.2.1 Types

A variable has an identifier and an associated value, taken from a range of possible values described by the variable's *type*. Types are similar to sets, in that they are collections of values sharing similar characteristics. If a variable is described as being of a particular type, it means that the variable can only hold one of the possible values at a time from the collection of values that are described by the type, and cannot hold a value that is not contained in that type. Therefore, types, amongst other things, describe sets of values, giving a variable a choice of a range of possible values.

Consider the following example:

A certain garage owns a number of different coloured cars. Together they constitute the total number of cars *max_cars*. This total number of cars is a natural number. The value can vary according to the number of cars owned by the garage at that point in time. The type representing this information can be represented as the type *max_cars* declared by:

$max_cars: \mathbb{N}$

where $\mathbb{N} = \{0, 1, 2, \dots\}$ is the set of natural numbers.

It is customary in Z to build a specification starting with a set of *basic types*, i.e. the sets from which all variables in the specification may take their values. A basic type is declared by writing the type name within square brackets, e.g. the basic type CARS is indicated by [CARS].

The declaration $max_cars: \mathbb{N}$ is an example of an *abbreviated* description. The set \mathbb{N} is not a basic type. Types in Z are maximal sets and the set of all integers, namely \mathbb{Z} , is a valid type in Z. Therefore, the complete definition of *max_cars* requires the specifier to state that this set is of type \mathbb{Z} and that $max_cars > 0$.

Suppose *sell_cars* is the set of all cars presently for sale. Only red and white cars are for sale. The cardinality of *sell_cars* is less than or equal to the total number of cars in storage.

The sets *white_cars* and *red_cars* are elements of the type \mathbb{P} CARS. The sets *white_cars* and *red_cars* together constitute the set of cars *sell_cars*.

Note that the variables *sell_cars*, *white_cars*, and *red_cars* are elements of a power set \mathbb{P} CARS. The power set $[\text{Der01}] \mathbb{P}$ CARS represent the set of all subsets of CARS.

sell_cars: \mathbb{P} CARS
white_cars: \mathbb{P} CARS
red_cars: \mathbb{P} CARS

3.2.2.2 Axiomatic descriptions

Global variables are declared in definition paragraphs called axiomatic descriptions. A global variable is a variable whose scope covers the whole of the specification text from its declaration onwards.

<i>declaration</i>	
--------------------	--

<i>predicate</i>	
------------------	--

The names declared above the horizontal must be new to the specification [Rat94, Bar94].

Axiomatic descriptions may be used to define *constants* [Ran94, Sch93]. Within the *cars* example, the maximum number of cars in storage is given a constant value of 50. In the Z specification, the constant *max_cars* is defined by:

<i>max_cars</i> : \mathbb{N}	(declaration of constant)
--------------------------------	---------------------------

<i>max_cars</i> = 50	(value of constant)
----------------------	---------------------

Axiomatic definitions are also used to describe *functions* expressed in Z [Jak97]. For example if $iroot(a)$ is the square root of a , then a square root function can be expressed as follows:

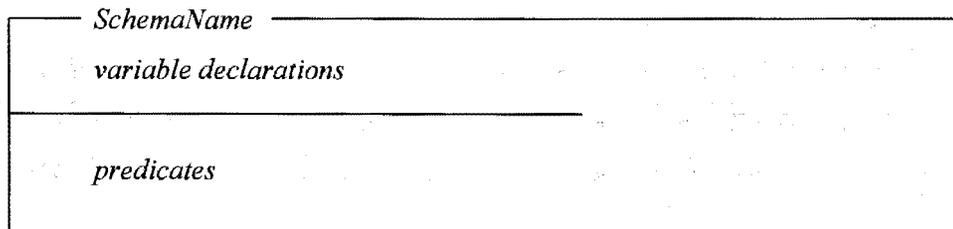
$iroot : \mathbb{N} \rightarrow \mathbb{N}$
$\forall a : \mathbb{N} \bullet iroot(a) * iroot(a) \leq (iroot(a) + 1) * (iroot(a) + 1)$

3.2.2.3 Schemas

3.2.2.3.1 State schemas

State schemas define variables and the relationships among the variables. Each time a variable changes its value it is said to change its state. The relationships between the variables in a state schema are written as predicates. These predicates, also known as *system invariants*, hold in every valid state of the system [Ran94, Sch93].

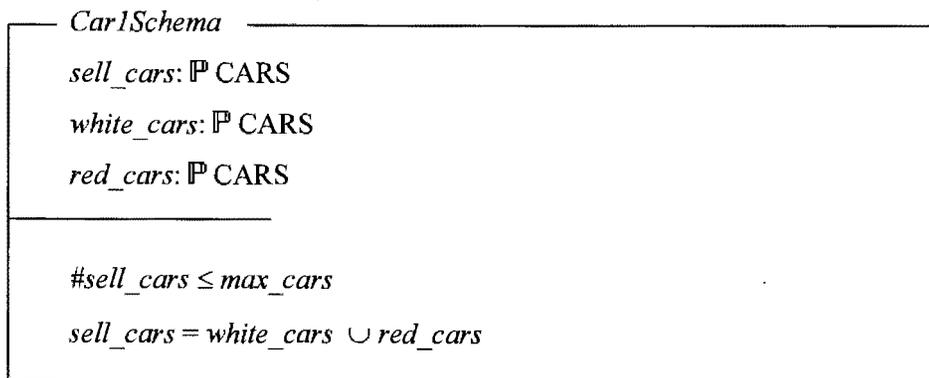
A schema has the following form:



Example:

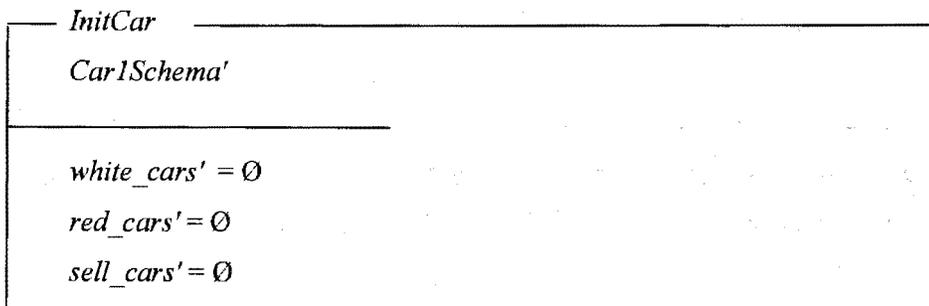
In the *cars* example, the definition of the variable *sell_cars* is given in a state schema named *Car1Schema*. Above the 2nd horizontal line are the declarations of the objects or variables used in the system. The variables *sell_cars*, *white_cars*, and *red_cars* are declared as sets of elements of type \mathbb{P} CARS. The value of *max_cars* is a non-negative integer.

Below the 2nd horizontal line the relationships among the variables are given. There is one relationship that is expressed as a predicate between the variables *sell_cars* and *max_cars*.



This predicate $\#sell_cars \leq max_cars$ states that the number of cars for sale must not be greater than the total number of cars in storage, that is, the number of elements in the set $sell_cars$ must not exceed the value of max_cars . The set $sell_cars$ constitute a union (\cup) of the sets $white_cars$ and red_cars . The symbol $\#$ is used to denote the cardinality of a set.

The initial state of the system is given by the schema:



3.2.2.3.2 Operation schemas

Operation schemas model changes of state as a result of a specified operation on the state. The precondition represents the constraints on the operations, and the post-condition describes the state after the operation.

For an operation schema:

- Any variables affected by an operation must be declared.
- An input to the operation is given by an input variable followed by a question mark, e.g. *new_car?*
- An output from the operation ends in an exclamation mark, e.g. *sell_cars!*

- State schema variables are imported into the schema by use of the symbols Δ and Ξ . Δ indicates that variables which are declared in a state schema may change their value as a result of the operation. Ξ indicates that variables which are declared in a state schema will not change their values as a result of the operation.
- Messages are introduced by using a *free type* (also known as a data type), for example $\text{RESPONSE} ::= \text{car_removed} \mid \text{new_car_added}$.

Next, consider the operation of purchasing a new red car for the show room. This car will be added to the set of all red cars for sale after the operation. Details of this purchasing operation are given in the operation schema *PurchaseRedCar* below.

Consider the actions associated with the schema *PurchaseRedCar*. Before the new car can be purchased, it must be checked that the car is not already owned, that it is not in the collection of all cars. If the car is not already owned it may be purchased, and the variable *red_cars* is updated. $\Delta\text{Car1Schema}$ in the operation schema *PurchaseRedCar* asserts that the schema *PurchaseRedCar* may change the state of the variables declared in the state schema *Car1Schema*.

<i>PurchaseRedCar</i>	
$\Delta \text{Car1Schema}$	
<i>redcar?</i> : CARS	
<i>reply!</i> : RESPONSE	
<i>redcar?</i> \notin <i>sell_cars</i>	(precondition)
<i>red_cars'</i> = <i>red_cars</i> \cup { <i>redcar?</i> }	(postcondition)
<i>reply!</i> = <i>new_car_added</i>	(postcondition)

The set *red_cars'* represents the updated set for *red_cars*.

The precondition *redcar?* \notin *sell_cars* states that the new car must not already be an element of the collection of cars. If the pre-condition is satisfied the new value of the variable *red_cars* is the union of the set *red_cars* and {*redcar?*}. The second predicate is a postcondition that must be satisfied afterwards. A further postcondition asserts that the response is *new_car_added*.

3.3 Specification, validation and verification

Before launching into design and implementation, a Z specification needs to be verified and validated [Rat94, Wor92].

3.3.1 Verification

At each development stage the progression of the software product can be checked to ensure that it is a correct and consistent representation of the previous stage. This constitutes verification.

Verification is also the carrying out of reasoning tasks that establish that specifications possess certain desirable properties [Rat94, Wor92] or that certain undesirable properties are absent. One of the great strengths of a formal method (e.g. Z) is that its mathematical underpinnings provide the basis for verification or proof. The main areas of specification consistency addressable by reasoning about the specification are [Rat94].

- verifying consistency of global definitions
- verifying consistency of state models
- verifying consistency of operations.

3.3.1.1 Terminology

Before commencing with the discussion on verification, we introduce some terminology.

- *A sequent.* A sequent is a claim or conjecture that its conclusion part follows (i.e. can be deduced) from its hypothesis part [Rat94]. The general format is:

Hypothesis \vdash *Conclusion*.

The symbol \vdash is called a 'syntactic turnstile'. Once a sequent has been proven, it becomes a theorem in the system.

- *A consistent Z specification*

This is a specification that is free of logical defects and 'can be realised' as a mathematical object. The specification can then be refined into code.

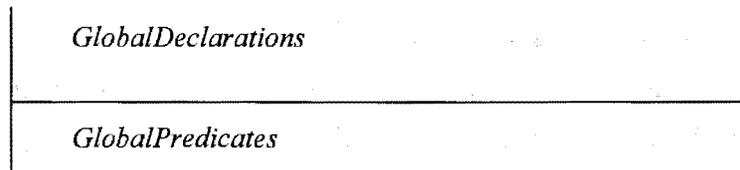
3.3.1.2 Methods of verification

A number of verification methods can be performed on the specification:

- Peer review. The work of an individual or a team is reviewed by a panel made up of project members who can make an effective contribution. Peer reviews can detect errors in the way that the language is used, or in the understanding of the application. Peer reviews ensure that the specification is clear and meaningful and that the accompanying documentation is sensible.
- Checking syntax and types: Z has a well defined syntax, hence a specifier can use software tools for syntactic analysis and type checking (e.g. CaDiZ – [Toy00]). Since this is an automated process it is a quick and reliable method for detecting errors.
- Reasoning about the correctness of the specification. This activity can be used to find conflicts between one part of the specification and another, or differences between specified actions. An interactive (e.g. CaDiZ) or automated (e.g. OTTER – see [vdP00]) reasoning assistant can be employed for this task.

3.3.1.3 Verifying consistency of global definitions

Refer to the axiomatic description

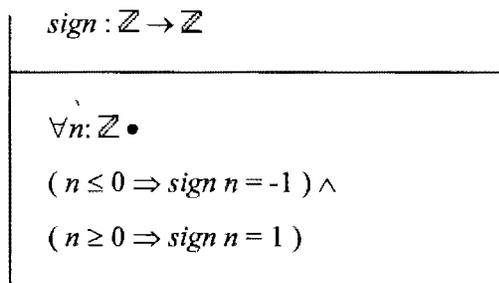


which can be written horizontally as:

GlobalDeclarations | *GlobalPredicates*.

It must be established that $\vdash \exists \textit{GlobalDeclarations} \bullet \textit{GlobalPredicates}$ which means that there exist values for *GlobalDeclarations* which satisfy predicate *GlobalPredicates*.

For example, consider the following (inconsistent) axiomatic description [Rat94, p.238]:



This description is inconsistent since it has a consequence $sign\ 0 = -1$ and $sign\ 0 = 1$, which contradicts the assertion that $sign$ above is a (total) function.

The following theorem must be established:

$$\vdash \exists \text{GlobalDeclarations} \bullet \text{GlobalPredicates}$$

The consistency theorem for the operation would be:

$$\vdash \exists \text{sign} : \mathbb{Z} \rightarrow \mathbb{Z} \bullet \forall n : \mathbb{Z} \bullet n \leq 0 \Rightarrow \text{sign}\ n = -1 \wedge n \geq 0 \Rightarrow \text{sign}\ n = 1$$

Eliminating $<$ and $>$

$$\vdash \exists \text{sign} : \mathbb{Z} \rightarrow \mathbb{Z} \bullet \forall n : \mathbb{Z} \bullet n = 0 \Rightarrow \text{sign}\ n = -1 \wedge n = 0 \Rightarrow \text{sign}\ n = 1$$

By properties of \mathbb{Z}

$$\vdash \exists \text{sign} : \mathbb{Z} \rightarrow \mathbb{Z} \bullet \forall n : \mathbb{Z} \bullet n = 0 \Rightarrow \text{sign}\ n = -1 \wedge \neg (\text{sign}\ n = -1)$$

By a simple property of logic

$$\vdash \exists \text{sign} : \mathbb{Z} \rightarrow \mathbb{Z} \bullet \text{false}$$

By a simple property of logic

$$\vdash \text{false}$$

Therefore, the sequent predicate is a contradiction, hence $\text{sign} : \mathbb{Z} \rightarrow \mathbb{Z}$ is inconsistent. Of course one way of avoiding this inconsistency is to change $(n \leq 0 \Rightarrow \text{sign}\ n = -1)$ to $(n < 0 \Rightarrow \text{sign}\ n = -1)$.

3.3.1.4 Verifying consistency of state models

A check must be performed to ensure that the state model is consistent. This check can be expressed as a theorem which has the following general form (see e.g. [Rat94]):

$$\vdash \exists \text{State}' \bullet \text{InitStateSchema}$$

which can be extended to

$$\vdash \exists \text{State}'; \text{Inputs?} \bullet \text{InitStateSchema}$$

if there are input variables present for the initial state schema InitStateSchema .

This theorem states that:

'There is a state of the general model (and inputs if InitStateSchema has any) that satisfies the initial state description'. Proving this theorem establishes consistency between State and InitStateSchema and consistency of State itself in the sense that it describes a non-empty state space, i.e. its predicate does not collapse to *false* [Rat94]. The predicate of Section 3.3.1.3 is an example of a predicate that collapses to *false*.

3.3.1.5 Verifying consistency of operations

For an operation that is defined as: $OperationDeclarations \mid OperationPredicates$, the consistency theorem is

$$\vdash \exists OperationDeclarations \bullet OperationPredicates$$

Calculating its precondition can check an operation's consistency. If the operation is inconsistent, its precondition will be *false*. A false precondition strongly suggests a defect in the operation description [Rat94].

As an example, it can be proved that operation *Exchange* in the following is inconsistent:

$$Schemal \hat{=} [x, y : \mathbb{Z} \mid x > y]$$

$$Exchange \hat{=} [\Delta Schemal \mid x' = y \wedge y' = x]$$

The consistency theorem for the operation would be:

$$\vdash \exists \Delta Schemal \bullet Exchange$$

Unfold $\Delta Schemal$ and *Exchange*

$$\vdash \exists x, y, x', y' : \mathbb{Z} \bullet x > y \wedge x' > y' \wedge x' = y \wedge y' = x$$

By onePtRule¹ on: x', y'

$$\vdash \exists x, y : \mathbb{Z} \bullet x > y \wedge y > x$$

By properties of \mathbb{Z}

$$\vdash \exists x, y : \mathbb{Z} \bullet x > y \wedge \neg(y > x)$$

By a property of logic

$$\vdash \exists x, y : \mathbb{Z} \bullet false$$

By a property of logic

$$\vdash false$$

That means the sequent predicate is a contradiction, hence that *Exchange* is inconsistent.

¹ The One Point Rule says that if we have an existentially quantified variable in a statement, part of which gives us an exact value for the quantified variable, then the quantification can be removed, replacing the variable by its known value wherever it appears [Pot96].

The onePtRule:

$$\exists x : X \bullet P \wedge x = t \equiv P[t/x]$$

$$[x \notin \phi t; t \in X]$$

$x \notin \phi t$: x must not occur freely in term t

$t \in X$: t 's value must belong to the set X .

[Rat94, Pot96, Der01]

3.3.2 Validation

A specification can be consistent, but it still might not be 'valid' in the more general sense of describing the system that should be built. In other words, the specification could describe something that is implementable, but contain features (or properties) that are not in accordance with the requirements set out by the customer.

One cannot *prove* that a formal specification is a correct formalisation of a (natural language) requirements specification. As Vienneau [Vie97] puts it: You cannot go from the informal to the formal by formal means. In particular, formal methods can prove that an implementation satisfies a formal specification, but they cannot prove that a formal specification captures a user's intuitive informal understanding of a system [vdP00].

Unlike consistency, validity cannot be 'proved' because it requires reference to the client, domain and/or requirements [Rat94, Wor92, Jak97].

Conjectures can be internal or external.

- Internal conjectures are constructed by the specifiers to increase their confidence in the specification's validity. For example, a collection of operations over a state should be checked for properties that lead to the conclusion that they describe the wanted behaviour.
- External conjectures are inspired by the client. It must be established that a state model is a valid abstraction of the reality it is intended to model, and that the operations over the state represent a functionality that the client actually wants [Rat94].

Validation techniques applicable to specifications include review, constructing prototypes and even specification execution with some development methods. Proving well-chosen validity-supporting conjectures results in:

- exploiting the full potential of formal notations
- a specification that is consistent and implementable.

If the requirements are clear, and the formal specification is well organised much of the specification can be validated by inspection [Jak97].

3.4 Refinement

Once a specification has been validated the design and implementation can begin. The design is the process of deriving concrete, computer-oriented specifications (though still not containing code) from validated, abstract user-oriented specifications to which the former must conform [Rat94]. Refinement includes the latter process, as well as the subsequent development of code from the concrete, computer-oriented specifications.

The three stages of refinement are data refinement, operation refinement and operation decomposition. Data and operation refinement can be looked at as that part of the development process that corresponds to the design phase of the traditional software life cycle. Ways to represent the abstract data structures that will be more amenable to computer processing are chosen, as well as the translation of abstract operations into corresponding concrete operations. The concrete operations are, however, still expressed in the language of schemas and describe only the relationship among the components of before and after states. This does not indicate how such changes of state are to be expressed in an implementation language.

Operation decomposition is the process of conversions of descriptions of state changes into executable instruction sequences. Operation decomposition can be carried through to the level of individual programming language instructions in the Z notation by using the schema calculus. With the addition of further schema operators, the decomposition process can continue until schemas are produced corresponding directly to programming language instructions. Z is a language for describing requirements formally, whereas programming languages describe sequences of computer-executable instructions. The goal of operation decomposition is to produce a program from which all schemas have been removed. A schema can be replaced by programming language text only when the change of state described by the schema is of such a simple kind that it corresponds to an instruction or a group of instructions in that language [Pot96, Woo96, Jak97].

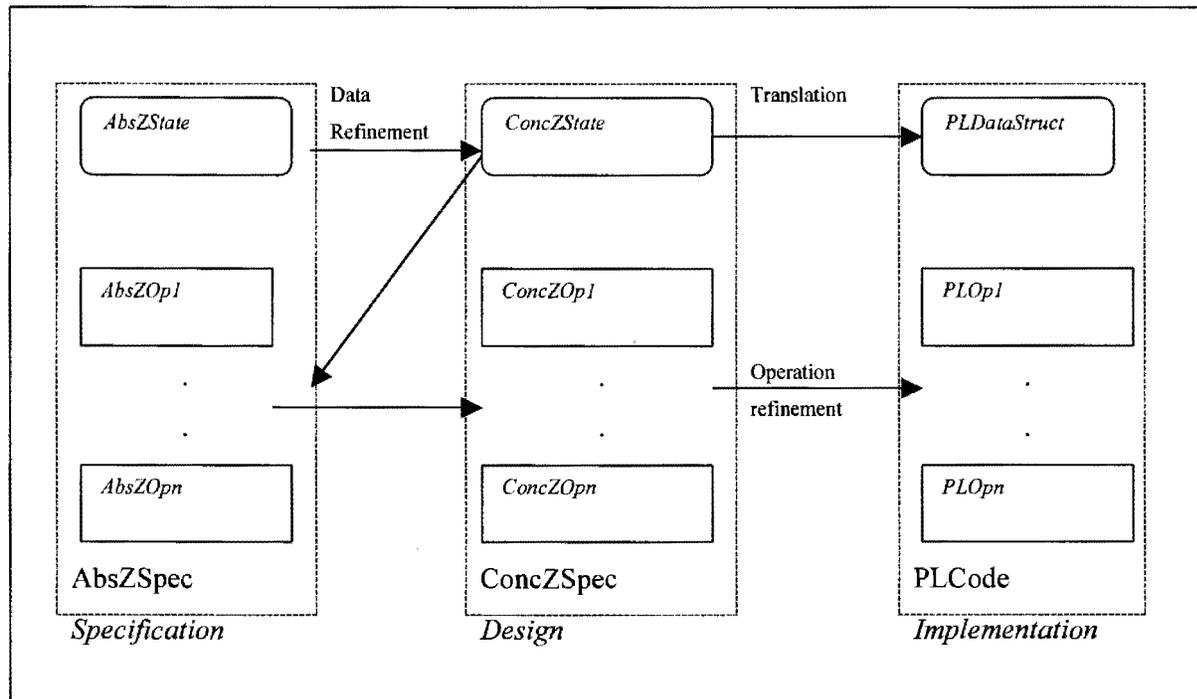


Figure 3.1 A global view of the refinement task [Rat94, p.243]

The process of refinement will be discussed using a number of examples. For the sake of consistency, the examples used will be predominantly from one source, but supported by the views of different other authors.

3.4.1 Global view of refinement

When a program is developed from a specification, two sets of design decisions usually need to be taken:

- The operations described by predicates in the specification must be implemented by algorithms expressed in a programming language.
- The data described by mathematical data types in the specification must be implemented by data structures available in the programming language [Spi92].

The overall refinement scenario can be expressed as follows [Rat94, Pot96]:

$$AbsZSpec \sqsubseteq PLCode$$

where *AbsZSpec* is the first, most abstract Z specification constructed, the relation \sqsubseteq denotes 'refined by', and *PLCode* is the final concrete form of the specification implemented in some chosen programming language (*PL*).

The refinement task is divided into two main phases as illustrated in Figure 3.1 above.

- Phase 1: The *AbsZSpec* is taken down to a level *ConcZSpec*, which is still written in *Z* but which is expressed in terms of data types that are suitable for direct translation into the PL. This design phase is driven by a process called *data refinement*.
- Phase 2: The components of *ConcZSpec* are gradually transformed into *PL* constructs until a complete translation into PL algorithms has been obtained. This implementation phase is driven by a process called *operation refinement*.

Sometimes, all state components in *AbsZSpec* could be adequately represented directly as *PL* data types and data structures. In such a case no data refinement is necessary. However, certain preliminary refinements to operations might still need to be performed before reaching a stage in the development where actual code begins to emerge [Rat94, Pot96, Woo96].

3.4.2 Data refinement

The data refinement phase is essentially the process of constructing a concrete data type that simulates an abstract one [Rat94, Pot96, Woo96, Der01].

The data refinement phase consists of two main components [Rat94]:

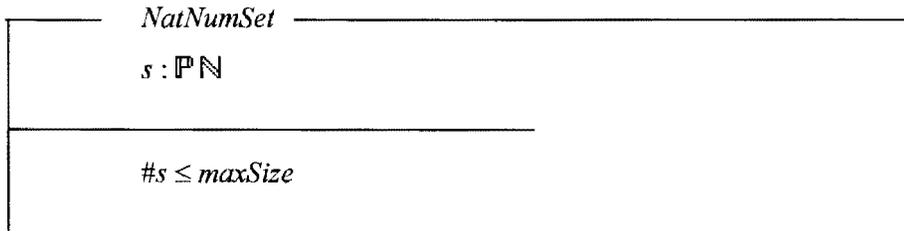
- The abstract state of *AbsZSpec* must be ‘adequately represented’ by concrete types available in the PL, i.e. by a concrete state, say *ConcZState*.
- Each operation *AbsZOp* over *AbsZState* must be correctly recast as an operation *ConcZOp* over *ConcZState* where *AbsZOp* represents abstract *Z* operations and *ConcZOp* represents concrete *Z* operations.

Consider the following data refinement example [Rat94, p.244]. In this example the following definitions are used:

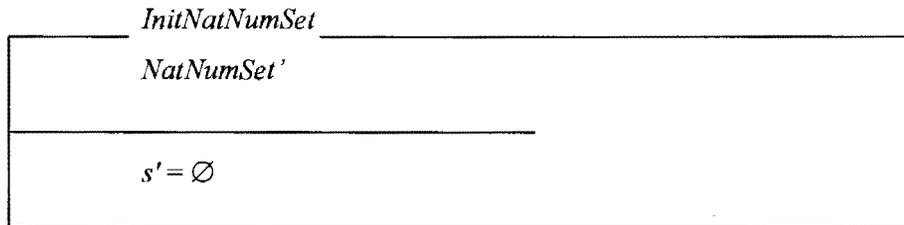
\mathbb{N} :	Natural numbers
s :	A set of natural numbers
$maxSize$:	Upper limit on the size of s
$\#s$:	Number of elements in the set s
\emptyset :	Empty set
x :	A variable that is a natural number

Given the abstract state *NatNumSet* with:

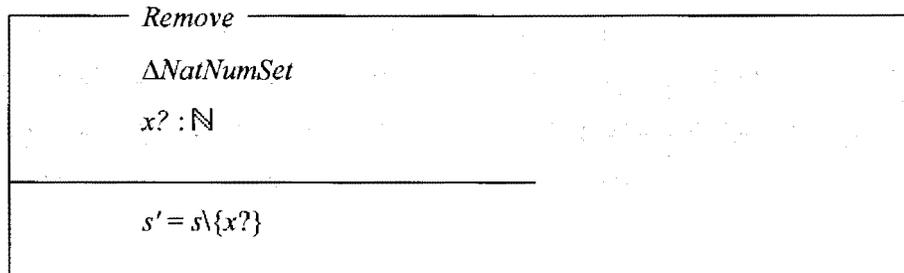
| *maxSize* : \mathbb{N}



with an initial state



and with one of its operations being



Suppose we decide to implement the state component $s : \mathbb{P}\mathbb{N}$ by an array with an index variable:

$a : \text{array } [1..\text{maxSize}] \text{ of Natural}$

Not all elements in the array need to be used. The variable *nels* is used to indicate the number of elements in use (the first *nels* elements of the array).

$nels : 0..\text{maxSize}$

It is assumed that the *nels* elements are sorted in ascending sequence to ensure that the array contains no duplicates.

Our array is modelled as a mathematical sequence. This choice of data structure is described in Z as (where $\text{dom } a$ denotes the domain of the array a):

NatNumArray <hr/> $a : \text{seq } \mathbb{N}$ $\text{nels} : 0..maxSize$
<hr/> $\#a = \text{nels}$ $\forall i, j : \text{dom } a \bullet i < j \Rightarrow a(i) < a(j)$

The concrete initial state is:

InitNatNumArray <hr/> $\text{NatNumArray}'$
<hr/> $a' = \langle \rangle$

$\langle \rangle$ indicates an empty sequence.

The abstract Remove operation is re-expressed as the following concrete operation:

RemElem <hr/> $\Delta \text{NatNumArray}$ $x? : \mathbb{N}$
<hr/> $a' = a \uparrow (\text{ran } a \setminus \{x?\})$ $\text{nels}' = \text{nels} - 1$

The symbol \uparrow denotes a sequence filtering: For a sequence s and a set of values v , $s \uparrow v$ creates a new sequence that contains precisely those entries in sequence s that are elements of set v , and in the same order. For example: $\langle a, b, c, d, e \rangle \uparrow \{b, d, f\} = \langle b, d \rangle$ [Bar94]. The function $\text{ran } a$

denotes the range of a relation. f is not an entry in sequence s and is therefore not present in the new sequence.

To determine whether the design correctly simulates what is abstractly described, the verification task is split into three main checks [Rat94]:

- Concrete state adequacy
- Concrete operation applicability
- Concrete operation correctness.

Note that this verification differs from the verification described in Section 3.3.1 in the sense that this time verification is performed on the concrete state, *ConZState*, of the specification, while the verification described in Section 3.3.1 is on the abstract state *AbsZState* of the specification. The *AbsZState* specification has been refined to the *ConcZState* using data refinement (refer to Figure 3.1). Therefore, the verification is on a refined version of *AbsZState*.

3.4.2.1 Concrete state adequacy

Concrete state adequacy is to determine whether the concrete state ‘adequately represents’ the abstract state. Three aspects are involved [Rat94]:

- The concrete state must be consistent.
- It must be determined whether every abstract state has at least one concrete representative.
- The correctness of the concrete initial state must be verified.

3.4.2.1.1 The concrete state must be consistent

The general format of this proof obligation is:

$$\vdash \exists \textit{ConcState}' \bullet \textit{InitConcState} \quad (\textit{InitConcState} \text{ represents the initial concrete state.})$$

For our example we have:

$$\vdash \exists \textit{NatNumArray}' \bullet \textit{InitNatNumArray}$$

NatNumArray' is a state of the general model *NatNumArray*. To show that it is consistent refer to *NatNumArray'*:

$\text{NatNumArray}'$
$a' : \text{seq } \mathbb{N}$ $\text{nels}' : 0..maxSize$
$\#a' = \text{nels}'$ $\forall i, j : \text{dom } a' \bullet i < j \Rightarrow a'(i) < a'(j)$

Proof:

$\vdash \exists a' : \text{seq } \mathbb{N}$

$\text{nels}' = 0..maxSize$

$\forall i, j : \text{dom } a' \bullet i < j \Rightarrow a'(0) < a'(1) < a'(2), \dots < a'(maxSize)$

$\#a' = \text{nels}'$

If $\text{nels}' = 0$ then

$a' = \langle \rangle$

which implies that there is a state ($\text{NatNumArray}'$) of the general model NatNumArray that satisfies the initial state description InitNatNumArray .

3.4.2.1.2 It must be determined whether every abstract state has at least one concrete representative

This can be achieved by determining if each abstract variable can be derived or ‘retrieved’ from the concrete variables by writing down equalities of the form:

$$\text{AbsVar} = \text{Expr}(\text{ConcVars})$$

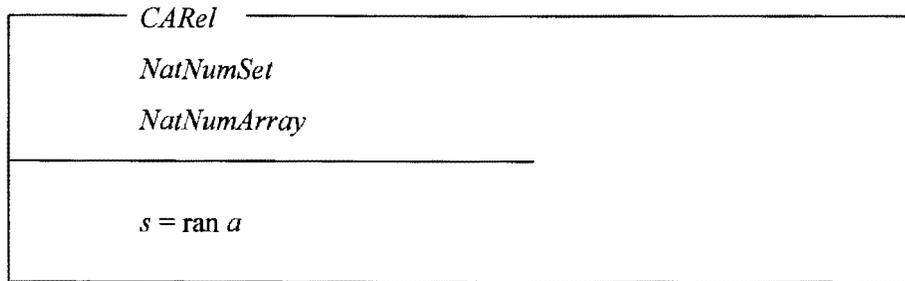
where AbsVar represents an abstract variable of the abstract state, Expr an expression and ConcVars the concrete variable of the concrete state representing the abstract state.

For the example, the predicate

$$s = \text{ran } a$$

will be referred to as the ‘retrieve relation’ CAREl (concrete-to-abstract relation) that brings together the abstract and the concrete states [Rat94, p.246]:

The equality means that CAREl is in effect a total function when viewed as ‘calculating’ the abstract state from the concrete one. Being total implies that every concrete state maps to some abstract state. This implicit property of the retrieve relation being functional and total, characterises the fact that a simplified form of data refinement is discussed.



Suppose, however, the ‘sorted’ invariant was removed from *NatNumArray* so that the array element order was immaterial. Assume that no duplicates are stored in the array. The design will now include some redundancy in that each non-empty, non-singleton set in the abstract state would have more than one concrete representation. [Rat94].

For example, the abstract state

$$\langle s \Rightarrow \{ 0,1 \} \rangle$$

would have two concrete representatives:

$$\langle a \Rightarrow \langle 0,1 \rangle, nels \Rightarrow 2 \rangle$$

$$\langle a \Rightarrow \langle 1,0 \rangle, nels \Rightarrow 2 \rangle$$

In general, assuming no duplicates, there would be *nels!* (!denoting a factorial) concrete representatives for a single abstract state. The implicit functionality of a retrieve relation such as *CARel* is not compromised because the relation expresses a calculation from *concrete to abstract*. That is, *CARel* will hold even if array *a* contained duplicates [Rat94].

3.4.2.1.3 Verifying the correctness of the concrete initial state

The concrete initial state must not describe initial states that have no counterpart in the abstract model.

To illustrate, a theorem of the following form is proved:

Given the retrieve relation then:

$$InitConcState \vdash InitAbsState \tag{3.1}$$

which says that ‘for each concrete initial state, there is a corresponding abstract one from which it is derived’.

$CARel'$ $NatNumSet'$ $NatNumArray'$
$s' = \text{ran } a'$

We need to prove that given a state $CARel'$ of the general model $CARel$ then:

$$InitNatNumArray \vdash InitNatNumSet$$

That is

$$NatNumArray' \mid a' = \langle \rangle \vdash NatNumSet' \bullet s' = \emptyset$$

$CARel'$ acts as an extra hypotheses (given $CARel'$) and the declarative part of the right-hand side schema text is just $NatNumSet'$ (section 3.4.2), which is provided by $CARel'$ on the left.

The sequent is then unfolded into

$$CARel' ; NatNumArray' \mid a' = \langle \rangle \vdash s' = \emptyset$$

(that is $NatNumArray' \wedge NatNumSet' \mid s' = \text{ran } a'; NatNumArray' \mid a' = \langle \rangle \vdash s' = \emptyset$)

which holds because $a' = \langle \rangle$ on the left and $s' = \text{ran } a'$ in $CARel'$.

By substitution $s' = \text{ran } \langle \rangle$, and $s' = \emptyset$ immediately follows.

Since in a theorem, the weaker predicate appears on the right hand side of the turnstile (see e.g. (3.1)) the concrete initial state space is *narrower* than its abstract model. It is sufficient if every concrete initial state has an abstract counterpart - it is not necessary for every possible abstract initial state to have a concrete representative.

3.4.2.2 Concrete operation applicability

It must be determined whether for each concrete-abstract operation pair, the former (concrete) is applicable over its state whenever the latter (abstract) is applicable over its state [Rat94, Pot96]. This check ensures that each concrete operation has an applicability that at least encompasses its abstract partner. It must be shown that given the retrieve relation then:

$$\text{pre } AbsOp \vdash \text{pre } ConcOp \quad (3.2)$$

[Rat94, p.247]

In the example discussed above, we need to show that given $CARel$, then:

$$\text{pre } \text{Remove} \vdash \text{pre } \text{RemElem}$$

The right-hand side is schema text, therefore its declaration part (the schema reference *NatNumArray* and variable $x? : \mathbb{N}$), as well as its predicate, must be derivable from the left-hand side [Rat94]. *CARel* acts as an extra hypothesis, and so *NatNumArray* in *pre RemElem* is provided by *CARel*. Variable $x?$ in *pre RemElem* is the same as the $x?$ of *pre Remove*. Therefore, is it valid to re-arrange the sequent by unfolding its left-hand side and writing just the conclusion's *predicate* on the right-hand side of the turnstile? We reason as follows [Rat94].

$$CARel; NatNumSet; x? : \mathbb{N} \mid$$

$$\text{true} \vdash \text{true}$$

$$s' = s \setminus \{x?\} \vdash a' = a \uparrow (\text{ran } a \setminus \{x?\})$$

which holds because we have

$$s = \text{ran } a \quad \text{in } CARel$$

The above turnstile can be proven as follows: By substitution

$$s' = (\text{ran } a \setminus \{x?\})$$

$$a \uparrow s' = a \uparrow (\text{ran } a \setminus \{x?\})$$

$$a \uparrow \text{ran } a' = a \uparrow (\text{ran } a \setminus \{x?\})$$

$$\text{ran } (a \uparrow \text{ran } a') = \text{ran } (a \uparrow (\text{ran } a \setminus \{x?\}))$$

(If $x \subseteq \text{ran } s$ then $\text{ran}(s \uparrow x) = x$)

Therefore

$$\text{ran } a' = \text{ran } (a \uparrow (\text{ran } a \setminus \{x?\}))$$

$$a' = a \uparrow (\text{ran } a \setminus \{x?\})$$

which proves that if given the hypotheses

$$s' = s \setminus \{x?\} \text{ then it can be concluded that } a' = a \uparrow (\text{ran } a \setminus \{x?\})$$

This sequent is trivially a theorem and so *RemElem* passes its applicability check.

3.4.2.3 Concrete operation correctness

It has to be determined for each concrete-abstract operation pair whether the concrete operation, when applied in circumstances conforming to its abstract partner's applicability, produces behaviour which its abstract partner would be capable of producing [Rat94, Pot96]. Behaviour correctness is ensured by proving, given the retrieve relation, then [Rat94, p.248]:

$$\text{pre } AbsOp \wedge ConcOp \vdash AbsOp \quad (3.3)$$

where $AbsOp$ represents the abstract operation applicable to the state and $ConcOp$ the concrete operation applicable to the state.

In the example it must be shown that given $\Delta CRel$, then:

$$\text{pre } Remove \wedge RemElem \vdash Remove$$

Unfold the sequent into

$$\begin{array}{l} \Delta CRel; NatNumSet; \Delta NatNumArray; x? : \mathbb{N} \mid \\ \text{true} \\ a' = a \uparrow (\text{ran } a \setminus \{x?\}) \\ \vdash \\ s' = s \setminus \{x?\} \end{array}$$

From the hypothesis equality, it follows that

$$\text{ran } a' = \text{ran } (a \uparrow (\text{ran } a \setminus \{x?\}))$$

(If $x \subseteq \text{ran } s$ then $\text{ran}(s \uparrow x) = x$)

By properties of sets

$$\text{ran } a \setminus \{x?\} \subseteq \text{ran } a$$

Therefore

$$\text{ran } a' = \text{ran } a \setminus \{x?\}$$

The conclusion of the sequent follows because of the equalities $s = \text{ran } a$ and

$$s' = \text{ran } a' \text{ in } \Delta CRel$$

So the correctness theorem holds for $RemElem$.

For (3.3) the reference to the concrete operation appears on the *left* of the turnstile. Behaviour of a concrete operation cannot produce more than its abstract partner in equivalent circumstances; however, it can be more deterministic. The concrete operation's behaviour must be the behaviour the abstract operation could produce [Rat94].

3.4.3 Operation refinement

After obtaining a validated, fully data-refined specification, the next step is to implement it. 'Fully data-refined' means that the whole of the state is now expressed at a level of

abstraction corresponding to the data structures of the target language. The operation specifications must now be correctly turned into code [Rat94, Der01].

Potter et al. [Pot96] describes operation refinement as the provision of concrete operations for each of the abstract operations. It will be required to prove for each concrete operation that it is a refinement of a corresponding abstract operation.

Ratcliff [Rat94] gives two approaches to operation refinement:

- Devise algorithms and then verify by some suitable reasoning mechanism that they conform to the concrete operation specifications.
- Gradually transform each concrete operation into an algorithm using rules and checks which guarantee that correctness is being maintained in the step-by-step decision-making.

The main problem with the *first* approach lies in producing *post hoc* correctness proofs. Rules will be needed to enable the reasoning about the behaviour of the constructs used to construct algorithms in the target language, and some way of relating the reasoning back to the concrete Z specification. Of course, the concrete Z specification needs to be changed if the *post hoc* correctness proofs discover errors.

The *second* approach guarantees correctness *as the code emerges*. This approach is referred to as *operation refinement* [Rat94], or *operation decomposition* [Pot96] because of the top-down nature of the process. The term ‘operation refinement’ will be used when refinement over the *same* state space is taking place.

For operation refinement to work the following must be adhered to [Rat94, Der01]:

- Refinement rules to introduce correct fragments of greater operational detail as expressed by the various constructs of the target language.
- A way of writing mixed descriptions: until the operation has been fully refined into code, parts of the operation description (initially none of it) will be described in target code (the bits already fully refined), and parts of it (initially all of it) will be the specification text (the bits still to be refined).

In any system of operation refinement, typical procedural constructs are [Rat94]:

- assignment statements
-

- various algorithmic structures: sequential composition, selection (multi *if-then* kind of statements), looping (usually the *while* variety), etc.
- the introduction of blocks and local variables
- the declaration of procedures, with or without parameters.

The following example illustrates the refinement of an operation schema into an algorithm.

Refer to *RemElem* (from Data Refinement Section 3.4.2):

<p style="text-align: center;"><i>RemElem</i></p> <hr style="border: 0.5px solid black;"/> <p>$\Delta \text{NatNumArray}$ $x? : \mathbb{N}$</p> <hr style="border: 0.5px solid black;"/> <p>$a' = a \uparrow (\text{ran } a \setminus \{x?\})$ $nels' = nels - 1$</p>

This is a data-refined operation because the abstract *Remove* operation (Section 3.4.2) is re-expressed as the concrete operation *RemElem*. The predicate still looks abstract and ‘unprogram-like’, though. *RemElem* is refined into *RemElem1*:

<p style="text-align: center;"><i>RemElem1</i></p> <hr style="border: 0.5px solid black;"/> <p>$\Delta \text{NatNumArray}$ $x? : \mathbb{N}$</p> <hr style="border: 0.5px solid black;"/> <p>$x? \notin \text{ran } a \Rightarrow$ $\theta \text{NatNumArray}' = \theta \text{NatNumArray}$ $x? \in \text{ran } a \Rightarrow$ $(\exists i : 1..nels \bullet$ $a(i) = x? \wedge a' = ((1..i-1) \uparrow a) \hat{\sim} ((i+1..nels) \uparrow a) \wedge$ $nels' = nels - 1$</p>

where $\hat{\ }^{\frown}$ denotes sequence concatenation and $\hat{\ }^1$ denotes a sequence extraction: For a set of numbers n and a sequence s , $n \hat{\ }^1 s$ creates a sequence containing only those elements in s that appear in the indexes given in n , and in the same order. For example $\{2,4,6\} \hat{\ }^1 \langle a,b,c,d,e \rangle = \langle b,d \rangle$ [Bar94].

The first two lines of the predicate in *RemElem1* state that if $x?$ is not in the array, nothing changes. The last four lines state that if $x?$ is in the array, it gets removed from its position i , leaving an updated but still contiguous sequence. The operation $(1..i-1) \hat{\ }^1 a$ extracts all the elements of array a from 1 to $i-1$: $a(1), \dots, a(i-1)$. The operation $(i+1..nels) \hat{\ }^1 a$ extracts all the elements of array a from $i+1$ to $nels$: $a(i+1), \dots, a(nels)$. The two extracted arrays are then concatenated. The only array element that was not extracted was $a(i)$, which has effectively now been removed. The new number of elements in the array $nels'$ is now 1 less than the original number $nels$.

RemElem1 is typical of moving towards code – it becomes more detailed [Rat94]. *RemElem1* can be refined into the following algorithm [Rat94, p.251]:

<i>[lookFor(x?, a, i, found)]</i>	[Check if $x?$ is in the array a]
if \neg found then skip	[If not found (not in the array) go to endif]
elseif found then	[If found (in the array) then
<i>[shiftDownOne(a, i)]</i>	elements in positions $i+1..#a$ are moved
	successively down one, thus getting rid of $x?$]
nels := nels - 1	[Derived from the predicate $nels' = nels - 1$]
endif	

The bracketed italicised parts indicate that the constructs involved are specification components, which need to be subjected to further refinement into code.

The theorems to prove to verify that *RemElem1* is a correct refinement of *RemElem* are the applicability and correctness theorems (refer to sections 3.4.2.2 and 3.4.2.3). Because *RemElem1* and *RemElem* are two operations over the same state space, no data refinement is involved, which means that no retrieve relation has to be involved in the proof of the two theorems. If, however, data refinement was involved, the retrieve relation would have been involved in the expression of the applicability and correctness theorems (refer to sections 3.4.2.1.3, 3.4.2.2 and 3.4.2.3).

Therefore, the expression of the applicability and correctness theorems for *RemElem1* and *RemElem* are simply:

$\text{pre } RemElem \vdash \text{pre } RemElem1$ (applicability)

$\text{pre } RemElem \wedge RemElem1 \vdash RemElem$ (correctness)

3.5 Operation decomposition

Data and operation refinement can be looked at as that part of the development process that corresponds to the design phase of the traditional software life cycle. It involves choosing ways of representing the abstract data structures that will be more amenable to computer processing, and the translation of abstract operations into corresponding concrete operations. Those concrete operations are still expressed in the language of schemas and describe only the relationship amongst the components of before and after states. *Operation decomposition* is the process of conversion of descriptions of state changes into executable instruction sequences (implementation language instructions) [Pot94]. Woodcock et al. [Woo96] describes operation decomposition as moving from the specification schemas to the programming language code.

Operation decomposition uses refinement laws that show how predicates can be replaced by programming language statements [Jak97]. Every refinement law can be written as:

$P \wedge Q \sqsubseteq PL$, with P the precondition, Q the post condition and PL a programming language fragment. The left side of a refinement law is a mathematical formula, and the right side is a code fragment, joined by the refinement symbol \sqsubseteq . This symbol is read as 'is implemented by' or 'translates to'. Also refer to Section 3.4.1 where the overall refinement scenario is expressed as $AbsZSpec \sqsubseteq PLCode$ where *AbsZSpec* is the first, most abstract Z specification constructed, the relation \sqsubseteq denote 'refined by', and *PLCode* is the final concrete form of the specification implemented in some chosen programming (implementation) language (PL).

Refinement laws show how predicates can be replaced by programming language statements.

For example: $(p \wedge q) \vee (\neg p \wedge r) \sqsubseteq p ? q : r$ (if p then q else r) [Conditional predicate]

The conditional predicate is a refinement law that shows how disjunction can be translated to a C conditional expression.

This is a further development from the original refinement scenario of Section 3.4.1.

In Section 3.5.1 below the processes of decomposing or refining the Z specification into code are illustrated, with implementation into C and Cobol used as examples. Some standard Z constructs will be discussed. Table 3.1 gives the mathematical and logical operators used by Z and C.

3.5.1. Sets

We start with a discussion of sets, followed by discussions of relations, functions, bindings, state schemas, operation schemas, schema expressions, guarded commands, assignment, disjunction, conjunction, new variables, quantifiers, modules and programs.

3.5.1.1 Smaller sets

The smaller sets are discussed to distinguish them from the larger sets in Section 3.5.1.2. Smaller sets are implemented by arrays of boolean flags, where there is an array index for each element in the type, and the flag tells whether the element is present in the set or not [Jak97].

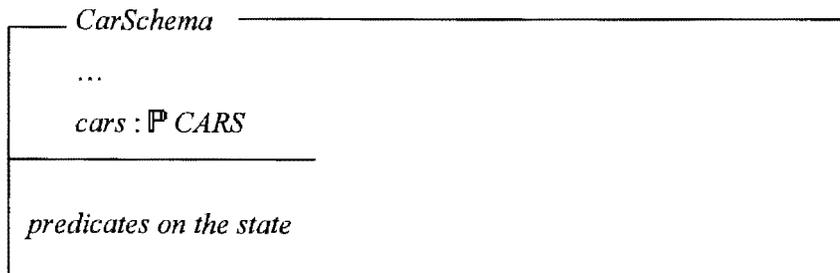
Z	C
+	+
*	*
<i>div</i>	/
<i>mod</i>	%
=	==
≠	!=
∧	& &
∨	
<	<
≤	<=
false (Boolean flag)	0 (Boolean flag)
true (Boolean flag)	nonzero integer (e.g. 1) (Boolean flag)

Table 3.1 Mathematical and logical operators used by Z and C [Jak97]

For example, consider the data type definition

$CARS ::= Toyota | Ford | Mazda | Chrysler$

together with schema *CarSchema* below:



3.5.1.1.1 Implementation into C

```
typedef enum { TOYOTA, FORD, MAZDA, CHRYSLER } car;
#define N_CARS SIZE 4 // The size of the array, i.e. number of car models.
int cars[N_CARS];
```

3.5.1.1.2 Implementation into Cobol

```
.....
01  N_CARS                pic 9(2) value 4.          [The size of the array]
.....
01  table1.
    05  car_table1.
        10  filler  pic x(8) value 'TOYOTA'.
        10  filler  pic x(8) value 'FORD  '.
        10  filler  pic x(8) value 'MAZDA  '.
        10  filler  pic x(8) value 'CHRYSLER'.
    05  car_table2 redefines car_table1.
        10  cars    pic x(8) occurs N_CARS times.    [Array definition]
```

3.5.1.2 Larger sets

Larger sets are usually modelled by other data structures [Jak97]. For example, the set of all the subscribers to some service could be implemented by a data file. In addition to the actual text of the subscriber's name, each entry might include additional information, so each element of *NAME* would be implemented by a C structure (called a record in many other

languages, e.g. Cobol). To enable a large file to be searched quickly, keys (in this case the name-key) can be included.

For example, the set of all subscribers to some service (assuming *NAME* is a basic type):

```
[NAME]
| subscriber :  $\mathbb{P}$  NAME
```

In addition to the text of the subscriber's name, each entry includes additional identifying information, namely the ID and the name_key of the subscriber.

3.5.1.2.1 Implementation into C

In addition to the text of the subscriber's name, each C structure includes additional identifying information, namely the ID and name_key of the subscriber. [Jak97, p.267]

```
/* NAME */
typedef struct
{
    char name_string[name_length];
    int id;
    int name_key;
} Name;

/* Subscriber set, a file of Name */
FILE *subscriber;
```

The variable *subscriber* (from the C implementation) is a pointer to a type called FILE (hence the singular form *subscriber*), defined in the `stdio.h` header file. *subscriber* is also called a file handle and it has the address of the package of information about the file: where the buffer is, where the file is found, and so on [Ram98].

3.5.1.2.2 Implementation into Cobol

In addition to the text of the subscriber's name, each Cobol record description includes additional identifying information, namely the ID and name_key of the subscriber.

```
.....
fd    subscriber_file.                                [Subscriber file]
```

```

01    subscriber_record.
      05    name_string      pic x(20).
      05    id               pic 9(10).
      05    name_key        pic 9(3).

```

3.5.1.3 Set membership

Each set membership test depends on how the set is implemented. When the set is implemented by a data structure the membership test is implemented by searching for the element in the structure.

$x \in s \sqsubseteq \text{Search for } x \text{ in data structure } s$ [Membership (data structure)]

For example, when a set is implemented by a file, as in Section 3.5.1.2 the membership predicate $name? \in subscriber$ will be implemented by code that searches the file for a particular record.

3.5.1.3.1 Implementation into C

The set is implemented by an array of flags indexed by the element names. The membership test is the value of the array element at the index, where a nonzero value indicates the element is a member of the set (refer to Table 3.1).

$x \in s \sqsubseteq s[x]$ [Membership (Boolean array)]

3.5.1.3.2 Implementation into Cobol

.....

*Assume that there are 10 elements in the array s.

```

01    x          pic 9(2)    value 10.
01    s          pic x(2)    occurs x times.
01    i          pic 9(2).

```

.....

perform search_array varying i from 1 by 1 until i = x.

search_array.

if s(i) = 0

display 'element' i ' not a member of set s'

else if s(i) ≠ 0

display 'element' i 'member of set s'.

.....

3.5.2 Relations

Z relations often represent data structures, as indicated by the following telephone directory example:

$phone: NAME \leftrightarrow \mathbb{N}$ $subscriber: NAME$	<hr style="width: 100%;"/> $dom\ phone = subscriber$
--	--

The component *phone* is a relation of type $\mathbb{P}(NAME \times \mathbb{N})^2$, that can be implemented by a C structure where the relation can be a file or an in-memory data structure that is organised to permit rapid search and retrieval.

3.5.2.1 Implementation into C

```

/* Phone record */
typedef struct
{
    Name name;
    int phone_num;
} phone_rec;

/* Phone relation; a file of phone_rec */
FILE * phone;

```

3.5.2.2 Implementation into Cobol

```

.....
fd    phone_file.                                [Phone file]
01    phone_record.
      05    name                                pic x(20).
      05    phone_num                           pic 9(10).

```

² Note that for any 2 sets X and Y, $X \leftrightarrow Y$ is defined as $\mathbb{P}(X \times Y)$

For every name (`name_string`) of a subscriber from file `FILE *subscriber` (in C) (Section 3.5.1.2.1) or `subscriber_file` (in Cobol) (Section 3.5.1.2.2) the corresponding phone number can be obtained from the phone file `FILE *phone` (in C) (Section 3.5.2.1) or `phone_file` (in Cobol) (Section 3.5.2.2), when the `name_string = name`.

3.5.3 Functions

The translation of Z axiomatic definitions may involve the definition of functions in the programming language. The refinement of the bodies of these functions may result in new refinement sub problems. During the operation decomposition phase further useful functions may be identified as a result of uncovering recurring patterns of evaluation.

A function is often just a binary relation. Since the first element of each such pair in a function is unique, it can act as an 'index'. Functions can be implemented by files or data structures where each item has a unique key. The keys correspond to the domain of the function. The items stored in the structure correspond to the range.

An example is array u below, with the array indices the domain of the function, and the array elements the range of the function. In Z we could specify this in an *axiomatic* description (\mathbb{Z} represents a set of integers):

$$| u : \mathbb{Z} \rightarrow \mathbb{Z}$$

3.5.3.1 Implementation into C

```
int u[n];           /* i.e. u is an array from 1 .. n of integers. */
```

where n is the domain of the function.

3.5.3.2 Implementation into Cobol

```

      ....
01   domain.
      05   n                pic 9(2).
01   table1.
      05   range occurs n times.
      10   u                pic 9(2).
```

3.5.4 Function application

For functions we use the following informal laws [Jak97]:

$u(x) \sqsubseteq$ Find item in u with key x [Function application using a key into an array]
 $f(x) \sqsubseteq$ Evaluate f with argument x [Function application, i.e. function call]

3.5.4.1 Implementation into C

The data structure u and function f are implemented by an array and a C function, respectively:

$u(x) \sqsubseteq$ $u[x]$ [Function application (array)]
 $f(x) \sqsubseteq$ $f(x)$ [Function application (C function)]

3.5.4.2 Implementation into Cobol

Arrays:

```
01  x                pic 9(2).
   01  table1.
       05  u  occurs x times      pic x(5).      [Array]
```

Functions:

For example: Function $\max(2,8,5,-7) = 8$

This function returns the maximum of the values between the parentheses. [Par96].

Or the function $f(x) = 3x + 4$ returns the value of $f(x)$, i.e., $3x + 4$.

```
.....
01  f                pic 9(2).
01  x                pic 9.
.....
      accept x.
      compute f = 3 * x + 4.      [ Function application ]
      display f.
.....
```

3.5.5 Functions and relations

Z specifications are structured into formal texts, collected into paragraphs such as axiomatic definitions and schemas, interspersed by natural language prose. These Z paragraphs can be made to correspond to programming language units in the implementation such as functions, procedures and methods.

Z relations that are used as predicates can be implemented in C by functions that return integer values [Jak97]. Zero indicates false and nonzero indicates true. For example, the Z relation $odd(x)$ is true when x is an odd integer. ($odd_$) can be implemented by a C function that tests whether $odd(x)$ is true. The following is a possible Z definition of $odd_$:

$$\frac{odd_ : \mathbb{Z} \rightarrow \{0, 1\}}{\forall i : \mathbb{Z} \bullet odd(i) \Leftrightarrow i \bmod 2 \neq 0}$$

The definition of the corresponding C function is

```
int odd(int i) {return i%2; }
```

Therefore, the Z predicate $odd(x)$ can be implemented by the C code fragment `x%2`.

From Table 3.1, page 28 it can be seen that logical *true* is represented in C by any nonzero integer.

3.5.5.1 Implementation into C

Statement	Refinement law
$odd(x)$	[Given]
$\Leftrightarrow x \bmod 2 \neq 0$	[Comprehension]
$\sqsubseteq x\%2 \neq 0$	[C operators]
$\equiv x\%2$	[final C]

We start with a standard Z-formula. Identifiers are replaced by their definitions and manipulated using ordinary laws. The translation is refined into code, which is simplified. Another example of the final step is given by (e represents an evaluable expression):

$$e \neq 0 \equiv e \quad \text{[i.e. simply the value of } e \text{]}$$

3.5.5.2 Implementation into Cobol

```

.....
01  answer      pic 9(2).
01  rem         pic 9.
01  result      pic x(5).
01  x           pic 9(2).
```

```

.....
divide x by 2 giving answer remainder rem.
if rem not equal to zero
move 'true' to result [True]
display 'number is odd '
else move 'false' to result [False]
display 'number is not odd '.
.....

```

3.5.6 Bindings

θ is the Z schema binding expression. Whenever a value is written in Z, it must belong to some type, i.e., be a member of some set that is its type. Since a schema represents a type in Z, a new type is introduced when a schema is defined. Values of that type are called bindings, and contain a (name, value) pair corresponding to each component of the corresponding schema. A schema represents a set, the set of all its bindings. A binding is an assignment of values to a schema's components such that they obey its predicate [Sch93, Bar94].

The purpose of θ is to construct a binding value. The schema that is referenced by the argument to θ determines the names from which the binding is to be constructed. However, the values to be associated with these names are taken not from the referenced schema but from the declarations of those names that are in scope in the context in which the θ is used [Sch93, Bar94]. The expression θS gives a specifier access to all the components of S.

Consider the following example ([Jak97]):

Given the following Z schemas:

$$S \hat{=} [x, y : \mathbb{Z}]$$

$$Op \hat{=} [\Delta S \mid x' = x + y]$$

$$SumOp \hat{=} [\Delta S \mid x' = sum(\theta S)]$$

$$\frac{sum : S \rightarrow \mathbb{Z}}{\forall S \bullet sum(\theta S) = x + y}$$

The operation *SumOp* has the same effect as *Op*, however, all the work is done by applying the function *sum* to a binding of *S*. In this example *x* and *y* are in scope in the function *sum*. The access to the binding is implicit in references to global variables. (In Z, variables defined in axiomatic definitions are global, i.e. they can be used anywhere in a Z document after their definition).

3.5.6.1 Implementation into C

When there is only a single instance of *S* implemented by a global variable, the implementations of *sum* and *SumOp* don't need any passed parameters, and are therefore declared as void in C.

For example [Jak97]:

```
int x, y; /* Single binding of schema type S */
int sum(void) { return x + y; }
        /* theta S is global */
void sum_op(void) { x' = sum( ); }
        /* apply to theta S */
```

3.5.6.2 Implementation into Cobol

```
.....
01 x          pic 9.
01 y          pic 9.
01 sum1       pic 9(3).
01 sum_op     pic 9(3).
01 x2         pic 9(3).
.....
compute sum1 = (x + y).
compute x2 = sum1.
move x2 to sum_op.
.....
```

sum1 represents int sum(void) (in C) and x2 represents sum_op(void) (in C).

3.5.7 State Schemas

State schemas are usually implemented by *mutable* data structures [Jak97]. Mutable data structures are structures whose contents can change frequently. Refer to the following two examples:

3.5.7.1 A system with a single instance (binding) of a schema type

A state schema can be implemented by ordinary program variables. The binding is just the values that those variables hold, e.g. schema S with two components x and y :

$$S \hat{=} [x, y : \mathbb{Z}]$$

3.5.7.1.1 Implementation into C

The state schema S refines to the C declaration:

```
int x, y; /* variables in schema S */
```

3.5.7.1.2 Implementation into Cobol

```
.....
01    x           pic 9.
01    y           pic 9.
.....
```

3.5.7.2 State schema implemented by the declaration of a C structure

A C structure is a record in most other languages. The members of the structure (fields of the record) are the state variables in the schema. The bindings of the schema type are implemented by variables that are instances of the structure (record) type. It is then possible to have many bindings of a single type [Jak97].

For example, state schema S and schema *BASIC_SALESPERSON*.

```
BASIC_SALESPERSON
sales_id : SALESPERSON_ID
sales_name : NAME
sales_age : AGE
```

3.5.7.2.1 Implementation into C

Schema *S* (Section 3.5.7.1) is implemented as follows:

```
/* Schema type S */
typedef struct
{
    int x,y;
} S;
```

Schema *BASIC_SALESPERSON*:

```
`typedef struct
{
    int sales_id;
    char sales_name;
    int sales_age;
} Basic_salesperson;
```

3.5.7.2.2 Implementation into Cobol

Schema *S*:

```
.....
Fd    S_file.
01    S_record.
      05    x                pic 9.
      05    y                pic 9.
.....
```

Schema *BASIC_SALESPERSON*:

```
.....
Fd    Basic_salesperson_file.
01    Basic_salesperson_record.
      05    Sales_id         pic 9(5).
      05    Sales_name      pic x(20).
      05    Sales_age       pic 9(2).
.....
```

3.5.8 Operation Schemas

Operation schemas model changes of state as a result of the specified operation on the state. The precondition represents the constraints on the operations, and the post condition describes the state after the operation.

Operation schemas are implemented by procedures that can change the program state by the assignment of new values to global variables. Procedures are implemented by the same construct as functions in C. The type of a procedure is often declared as ‘void’ (see e.g. Jacky [Jak97]). In the case where there is a single binding of a schema type and it is implemented by a program variable, it is not necessary to pass a parameter to the procedure that implements the operation schema. The procedure simply updates the global variable. This is indicated in C by using a ‘void’ parameter list.

Consider the following state schema S and operation schema Op :

$$S \hat{=} [x, y : \mathbb{Z}]$$

$$Op \hat{=} [\Delta S \mid x' = x + y]$$

3.5.8.1 Implementation into C

```
int x, y; /* variables in schema S */

void op{void} { x = x + y; }
```

3.5.8.2 Implementation into Cobol

```
.....
01    x                pic 9.
01    y                pic 9.
.....
      compute x = x + y.
.....
```

3.5.9 Schema Expressions

A schema expression can be expanded to a single schema using the Z schema calculus. A single procedure that implements the expanded schema can then be written. For example, the final state of operation schema $Op12$ is state $S1$ or state $S2$ depending on a predicate p :

$$\begin{aligned}
 S &\hat{=} [x, y : \mathbb{Z}] \\
 Op1 &\hat{=} [\Delta S \mid p \wedge S1'] \\
 Op2 &\hat{=} [\Delta S \mid \neg p \wedge S2'] \\
 Op12 &\hat{=} Op1 \vee Op2
 \end{aligned}$$

Expansion of the schema expression:

$Op12$
ΔS
<hr style="border: 0.5px solid black;"/>
$(p \wedge S1') \vee (\neg p \wedge S2')$

3.5.9.1 Implementation into C

```
void op_12{void}
{ if (p) S1; else S2; }
```

3.5.9.2 Implemenation into Cobol

```
.....
if (p)
    then perform S1_rtn
    else perform S2_rtn.
.....
```

3.5.10 Assignment

The assignment statement is used to refine specifications where one or more variable(s) in the program state is to change, and the new values are easily computed in the target language or where equality of states are to be indicated [Wor92, Jak97].

3.5.10.1 Assignment where only one variable changes value

The *Assignment* refinement law is used where only one variable can change value. Refer to the following example [Jak97]:

$$x' = e \wedge y' = y \wedge z' = z \wedge \dots \sqsubseteq x = e \quad \text{[Assignment]}$$

where $x, y : \mathbb{Z}$, x' and y' after state variables (postconditions) and e is an expression. In this example only x' changes.

3.5.10.1.1 Implementation into C

$x = e$ [Assignment]

3.5.10.1.2 Implementation into Cobol

```

.....
01 x          pic 9.
01 y          pic 9.
01 z          pic 9.
01 e          pic 9(3).
.....
    compute e = x*y.           [Calculate e]
    move e to x.              [Assignment]
.....
    
```

3.5.10.2 Assignment where several variables can change

Equations can appear in any order, but a data flow analysis must be done to determine the order in which the assignments are to be performed [Jak97].

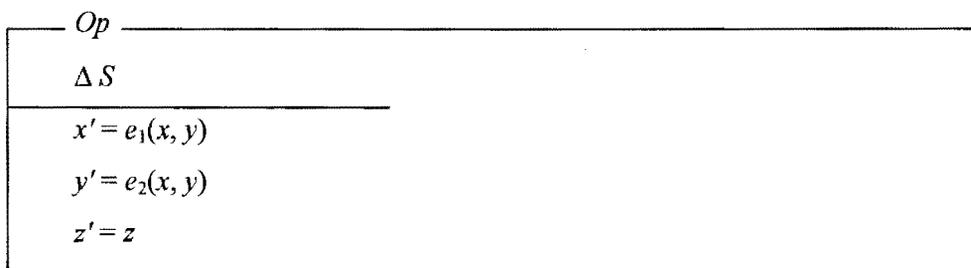
As an example refer to the following operation schema (Op) where at each transition, new values are assigned to each state variable, based only on the previous values of one or more variables. Suppose

$$| z : \mathbb{Z}, S \hat{=} [x, y : \mathbb{Z}]$$

and e_1 and e_2 are expressions involving x and y .

that is,

$$x' = e_1(x, y) \wedge y' = e_2(x, y) \wedge z' = z$$



3.5.10.2.1 Implementation into C

```
t = x; x = e1(x, y); y = e2(t, y)      /* t is a temporary storage value.*/
```

3.5.10.2.2 Implementation into Cobol

```
.....
01 x          pic 9.
01 y          pic 9.
01 z          pic 9.
01 e1         pic 9(3).
01 e2         pic 9(3).
01 temp       pic 9.
.....

      move x to temp.
      compute e1 = x*y.                [Compute e1]
      move e1 to x.                    [Assignment]
      compute e2 = temp + y.           [Compute e2]
      move e2 to y.                    [Assignment]
.....
```

3.5.11 Guarded Command

A concrete design is translated into an abstract programming notation and then further refined to yield a description in the language of guarded commands [Woo96, Der01]. A guarded command can be implemented by an if statement as described by the *Guarded Command* refinement law:

$$p \wedge s \sqsubseteq \text{if } (p) \text{ } s \quad [\text{Guarded command}]$$

where p is the descriptive predicate and s is the prescriptive predicate (defined below). This guarded command law says that if p , then the state is changed to produce the post condition prescriptive predicate s .

Descriptive predicates (p) describe situations where specifications are applicable, and they might be true or false. In operation schemas, preconditions that only contain unprimed state variables and input variables are examples of these descriptive predicates. Prescriptive predicates (s) are always true; they assert that the variables in the predicate must have values

that make the predicate true. In operation schemas, post conditions that contain primed variables and output variables are examples of prescriptive predicates [Jak97, Der01].

For example, refer to the following implementations:

3.5.11.1 Implementation into C

From 3.5.10.1 (assignment) we get the C construct:

$x = e_1 \wedge x' = e_2$	[Given]
$x : \mathbb{Z}, e_1$ and e_2 are expressions.	
\sqsubseteq if $(x = e_1) x' = e_2$	[Guarded command]
\sqsubseteq if $(x == e_1) x' = e_2$	[C operator] (1)
\sqsubseteq if $(x == e_1) x = e_2$	[Assignment] (2)

The first equation (1) becomes a test, and the second equation (2) becomes an assignment.

3.5.11.2 Implementation into Cobol

```

.....
01 x          pic 9.
01 e1         pic 9(3).
01 e2         pic 9(3).
.....
      if x = e1
      move e2 to x.
.....

```

3.5.12 Disjunction

A disjunction is satisfied by any situation that satisfies any of its disjuncts: it is *true* when either, of its disjuncts is *true*. Case analysis is applied where situations can be classified into cases and all the situations are handled the same way.

Disjunction is implemented in, for example, C and Cobol, by conditional branching. This can be expressed in the *Case Analysis* refinement law [Jak97, p.276]:

$$(p \wedge s) \vee (q \wedge t) \sqsubseteq \text{if } (p) s; \text{ else if } (q) t \quad \text{[Case analysis law]}$$

p and q are descriptive predicates and s and t are prescriptive predicates. Here p and q must describe distinct (non overlapping) situations (cases), so that only one of the predicates can be

true [Jak97]. If t itself is another disjunction, the case analysis law can be applied repeatedly to obtain the results

$$(p \wedge s) \vee (q \wedge t) \vee (r \wedge u) \dots \sqsubseteq$$

if $(p) s$; else if $(q) t$; else if $(r) u \dots$

3.5.12.1 Implementation into C

First case: $(p \wedge s) \vee (q \wedge t) \sqsubseteq$ if $(p) s$; else if $(q) t$

Second case: $(p \wedge s) \vee (q \wedge t) \vee (r \wedge u) \dots \sqsubseteq$
 if $(p) s$; else if $(q) t$; else if $(r) u \dots$

3.5.12.2 Implementation into Cobol

First case: if p
 perform s_rtn
 else if q
 perform t_rtn.

Second case: if p
 then perform s_rtn
 else if q
 then perform q_rtn
 else if r
 then perform u_rtn

3.5.13 Conjunction

The guarded command and sequential assignment laws (sections 3.5.10 and 3.5.11) show how to implement certain conjunctions. However, if both the conjuncts are prescriptive predicates that constrain the same variables, then often no generally applicable refinement law exists.

The following example illustrates this difficulty [Jak97]:

A number n and divisor d are given. d is divided into n giving quotient q' and remainder r' . *Division* is defined by conjoining *Quotient* and *Remainder*, where the predicate of *Quotient* is $d \neq 0 \wedge n = q' * d + r'$
 the predicate for *Remainder* is

$$r' < d$$

$$S2 \hat{=} [d, n, q, r : \mathbb{N}]$$

$$Quotient \hat{=} [\Delta S2 \mid d \neq 0 \wedge n = q' * d + r']$$

$$Remainder \hat{=} [\Delta S2 \mid r' < d]$$

$$Op \hat{=} Quotient \wedge Remainder$$

None of the assignments laws are matched by these because q' needs r' in its calculation (i.e. $q' = ((n - r')/d)$, while r' is not available beforehand because it in turn needs q' in its calculation (i.e. $r' = n - q'*d$).

3.5.14 New Variables

There are various reasons for adding new program variables to the implementation that are not present as mathematical variables in the specification. For example, the sequential assignment law (Section 3.5.10.2) uses the new program variable t to store the initial value of the mathematical variable x . Another reason for introducing new variables is to factor out repeated expressions which are too lengthy to write or costly to evaluate more than once. The refinement of an operation often requires the use of extra variables, the scope of which should be restricted to the section of the program corresponding to that refinement [Pot96, Jak97, Woo96].

New variables in programs correspond to local definitions in Z. The *local definition law* says that an expression e can be factored out and a new variable x can replace all of its occurrences:

$$s \Leftrightarrow (\mathbf{let} \ x == e \bullet s[x/e]) \quad \text{[Local definition]}$$

where $s[x/e]$ is a predicate with all occurrences of expression e being replaced by variable x .

The *New Variable* refinement law shows how predicates with local definitions are implemented [Jak97].

$$(\mathbf{let} \ x == e \bullet s(x)) \sqsubseteq x = e; s(x) \quad \text{[New variable } x\text{]}$$

3.5.14.1 Implementation into C

The local definition and new variable laws and other refinement laws are used to implement the guarded command $p(y') \wedge y' = fx$, (Section 3.5.11) where the primed variable y' appears in the guard [Jak97, p.284]:

Statement	Refinement law
$p(y') \wedge y' = fx$	[Given]
$\Leftrightarrow p(fx) \wedge y' = fx$	[Expand first y']
$\Leftrightarrow (\text{let } t = fx \bullet p(t) \wedge y' = t)$	[Local definition]
$\sqsubseteq t = fx; p(t) \wedge y' = t$	[New variable]
$\sqsubseteq t = fx; \text{if } (p(t)) y' = t$	[Guarded command]
$\sqsubseteq t = fx; \text{if } (p(t)) y = t$	[Assignment]
$\sqsubseteq t = f(x) ; \text{if } (p(t)) y = t$	[C operators]

```
void partial {void}
{
    int t;
    t = f(x);
    if (p(t)) y = t;
}
```

The state remains unchanged if $p(y')$ is false.

3.5.14.2 Implementation into Cobol

Assume that $f(x)$ is the function $f(x) = 3x + 4$, and $p(t)$ is the predicate $((2 + f(x)) > 3)$.

```
.....
01 temp                pic 9(5).
01 f                   pic 9(5).
01 y                   pic 9(5).
01 p                   pic 9(5).
01 x                   pic 9(2).
.....
compute f = 3x + 4.    [Function application (executable code)]
move f to temp.       [Assignment]
compute p = 2 + temp.
```

```

if (p > 3)
    move temp to y                                [Assignment]
else next sentence.
.....
    
```

3.5.15 Quantifiers

Predicates that involve sets and quantifiers are implemented by *loops* that iterate over the elements of the set. The elements are tested by the descriptive predicates. The universal quantifier requires that every element passes the test. The existential quantifier requires that at least one element passes the test. In the implementation into C the truth value of the quantified predicate is assigned to the boolean flag *b*, where *b=0* indicates *false* and *b=1* indicates *true*. (Refer to Table 3.1 page 3-28).

Z specifications:

```

S ≐ [ x , y : Z ]
∀ x : S • p(x) ≐ b = 1; for (x in S) if (! p(x)) b = 0           [Universal test]
∃ x : S • p(x) ≐ b = 1; for (x in S) if (p(x)) b = 1           [Existential test]
    
```

Prescriptive predicates assign new values to elements. While universally quantified predicates assign a value to every element, existentially quantified predicates can non-deterministically assign a value to any one or more elements.

```

∀ x : S • s(x) ≐ for (x in S) s(x)                               [Universal assignment]
∃ x : S • s(x) ≐ Choose any or all x in S; s(x)                 [Existential assignment]
    
```

3.5.15.1 Implementation into C

```

∃ x : S • p(x) ≐ b = 1; for (x in S) if (p(x)) b = 1           [Existential test]

∀ x : S • p(x) ≐ b = 1; for (x in S) if (! p(x)) b = 0         [Universal test]
                    ≐ b=1; for (i=0; i<n; i++) if (!p(s[i])) b=0
    
```

For example: If $p(x)$ is $odd(x)$ (Section 3.5.5.1) then replace $p(s[i])$ with $s[i]\%2$.

3.5.15.2 Implementation into Cobol

This implementation is illustrated for the example (above) where $p(x)$ is $odd(x)$.

```

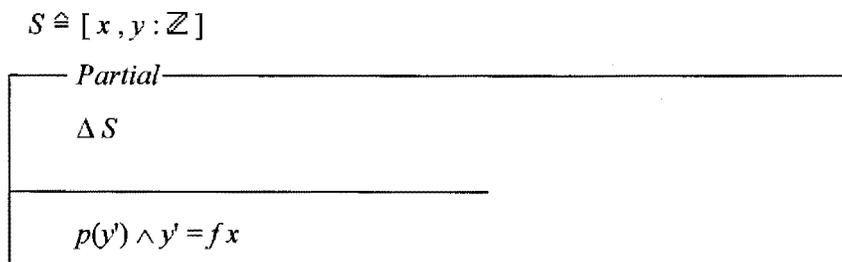
.....
01 rem          pic 9.
01 answer      pic 9(2).
01 i           pic 9(2) value 0.
01 b           pic 9 value 1.
01 n           pic 9(2) value 10.
*Assume 10 numbers (n=10)
01 array1.
    05 s        pic 9(2) occurs n times.
.....
    perform test_rtn varying i from 1 by 1 until i=n.      [Universal test]
test_rtn.
    divide s(i) by 2 giving answer remainder rem.
    if rem not equal to zero
    move 1 to b
    display s(i) ' is odd'
    else move 0 to b
    display s(i) ' is not odd'.
.....

```

3.5.16 Partial Operations

A partial operation is an operation that is applicable in some situations and its effects are undefined in others. Partial operations must be combined in schema expressions so that the combined operation is *total*, that is, its final state is defined for all possible input states. In code, partial operations are sometimes implemented by separate procedures whose results are combined elsewhere. It is then essential that each such procedure leaves the state unchanged if its precondition is not satisfied.

For example, the predicate in *Partial* below says that the assignment to *y* can only be made if the final value *y'* satisfies predicate *p*:



3.5.16.1 Implementation into C

The implementation into C is done in a similar fashion as in Section 3.5.14.1.

3.5.16.2 Implementation into Cobol

The implementation into Cobol is done in a similar fashion as in Section 3.5.14.2.

3.5.17 Modules and programs

A collection of Z related paragraphs can be implemented by the larger programming language constructs such as modules, packages or classes. For languages such as C that do not provide such constructs, large-scale structuring can be achieved by placing the declarations of related program units together in a single header file and defining them together in a single file of related variables, function definitions, and supporting code.

A Z state schema, together with the operation schemas and the definitions they use, can be implemented by one module. In the case where one Z schema includes another, this can be implemented by a module dependency, or inheritance in an object-oriented programming language. It is often implemented in C by the inclusion of a header file.

The Z notation has no structuring construct for collecting together related Z paragraphs, since it is up to the specifier to choose a layout and use the informal prose to make the intended organization of the paragraphs clear. Z has no built-in concept of a program or any explicit control structure. A top level system definition in Z is often determined by a collection of operation schemas. The operation schemas at the top level are not included in any others. For example, if the top level operation schemas are called $Op1$, $Op2$, ..., OpN then the formal definition of the main program is the disjunction of these operations:

$$Main \hat{=} Op1 \vee Op2 \vee \dots \vee OpN \vee Exception$$

The *Exception* operation handles any cases that might have been overlooked [Jak97].

The *Case Analysis* refinement law is an effective programming strategy to implement each operation such as, for example, $Op1$ with two program units: a Boolean function `test_1` that tests its precondition and a procedure `do_1` that performs the state change by executing assignments. The heart of the main program is a loop that repeatedly tests preconditions and executes any enabled operation until some exit condition is reached. This is made into an event-driven system by adding a statement at the beginning of the body of the loop that waits for events to be triggered.

3.5.17.1 Implementation of the *Case Analysis* refinement law into C

```
#include ...  
main (...)  
{  
    ...  
    while (ok)  
    {  
        get_event();  
        if ( test_1 ( ) ) do_1();  
        else if ( test_( ) ) do_2();  
        .  
        .  
        .  
        else if ( test_n ( ) ) do_n();  
        else exception ( );  
    }  
}
```

The above program is an example of an event-driven state transition system. Each test function examines the most recent event to determine whether its do operation should be invoked. By executing each operation's do procedure the program state can be changed so that a different operation becomes enabled (a different test function will succeed on the next pass through the body of the loop). The control reaches exception if none of the other preconditions are satisfied.

3.5.17.2 Implementation into Cobol

```
.....  
    perform event_tn until not_ok  
event_tn.  
    perform get_event.  
    if test_1 is true perform do1_rtn  
    else if test_2 is true perform do2_rtn  
    .  
    .
```

```
else if test_n is true perform don_rtn  
else perform exception_rtn  
.....
```

3.6 Summary

Before a specification can be refined and implemented, the specification needs to be verified and validated. In this chapter the main methods for these two processes were discussed. Some of the main ideas of refinement have been given. The data refinement, operation refinement, and operation decomposition processes were examined.

The examples of data and operation refinement showed how data is represented in a way that is more suitable for translation into programming language code. Often the distance between the level of specification and that of the programming language is too great for this to be accomplished in a single step.

Whenever the concrete operation is applied in a state representing an abstract state for which the abstract operation is applicable, the action of the concrete operation gives rise to a concrete state that represents an abstract state. This abstract state could have resulted from the action of the abstract operation. When there is output from the operation the output produced by the concrete system is a possible output of the abstract operation.

The operation decomposition process has been illustrated, with the accompanying examples of implementations in C and Cobol. This was done for sets, relations, functions, state schemas, operation schemas, schema expressions, assignments, guarded commands, disjunction, conjunction, new variables and quantifiers. Typical implementations into C and Cobol for modules and programs were presented.

A comparison between C and Cobol as far as the refinement and implementation from Z is concerned, follows:

Construct	Z	Refinement and implementation into C	Refinement and implementation into Cobol	Comment
Integers	In Z integer numbers are indicated by \mathbb{Z} and natural numbers by \mathbb{N} .	An integer variable y in C is indicated by the definition <i>int y</i> .	An integer variable y in Cobol is defined by the <i>pic y</i> clause: <i>y pic 9</i> . <i>pic 9(2)</i> indicates 2 integers, that is, a value between 00 and 99.	
Alphanumeric variables	An alphanumeric (any character including numbers) value can be defined by for example, a set, e.g. $A = \{p,q\}$, $B = \{1,2,3\}$	An alphanumeric value y (character string) is indicated in C by the definition <i>char y</i> .	An alphanumeric value y in Cobol is defined by the <i>pic x</i> clause: <i>y pic x(2)</i> , where the x indicates that y is alphanumeric characters and the 2 indicates the number (2) of alphanumeric characters.	
Relations (data structures)	Z relations that are represented by axiomatic definitions can represent data structures in Z.	Data structures in C can be represented by record structures of a file. A record structure of a file is indicated in C with the <i>typedef struct</i> definition, with the relevant fields between the $\{$ and $\}$ symbols. The corresponding file is indicated by, for example <i>FILE*phone</i> (phone file). (Refer to Section 3.5.2).	In Cobol a file is defined by the <i>FD</i> (file description) definition in the <i>FILE SECTION</i> of the program, with the corresponding record and file descriptions following in the <i>01</i> and higher level definitions of the <i>FD</i> . (Refer to Section 3.5.2).	
Functions	Functions in Z are indicated with axiomatic definitions.	Function applications are implemented by the evaluation of the function $f(x)$ with argument x , where $f(x)$ is the range depending on the value x (domain). (Refer to Section 3.5.4).	Function applications are implemented by the evaluation of the function $f(x)$ with argument x , where $f(x)$ is the range depending on the value x (domain). (Refer to Section 3.5.4).	The refinement into C from Z is more direct than into Cobol.
Assignment (equality)	A Z assignment is indicated as for example $x' = e$ (e is an expression).	The Z assignment is implemented into C by the statement $x = e$ (e is an expression)	The Z assignment is implemented into Cobol by the statements: <i>move e to x</i> or <i>compute x = e</i> . (Refer to Section 3.5.10.1).	The refinement for assignment is equally direct from Z into C and Cobol.

Construct	Z	Refinement and implementation into C	Refinement and implementation into Cobol	Comment
Disjunction	Disjunction is represented in Z by, for example $(p \wedge s) \vee (q \wedge t)$ where p and q are descriptive predicates and s and t are prescriptive predicates.	Disjunction is implemented by conditional branching: the conditional expression (<i>IF</i> statement) is implemented in C by <i>if (p) s;</i> where p is a descriptive predicate and s is a statement that is executed when the predicate is true. (Refer to Section 3.5.12).	In Cobol the corresponding implementation is: <i>if (predicate p is true) then perform s_rtn.</i> s_rtn is a routine that is executed if predicate p is true. (Refer to Section 3.5.12).	The refinement for disjunction from Z into C and Cobol is equally direct and clear.
Loops	Loops in Z are given by, for example, the disjunction of top level operation schemas: $Main \cong Op1 \vee Op2 \vee \dots \vee OpN \vee Exception.$	To implement a loop in C, the statement <i>while (ok)</i> (the loop is executed while ok is true) with the body of the loop nested inside a pair of braces “{ }”. (Refer to Section 3.5.17)	In Cobol the loop is executed with the statement: <i>perform x_rtn until not_ok.</i> The loop follows in x_rtn . (Refer to Section 3.5.17).	It can be concluded that the implementation into C and Cobol is equally direct and clear.
Variables and the relationship between the variables	State schemas	State schemas in Z are implemented into C by ordinary program variables, that can be included into a C structure. (Refer to Section 3.5.7).	State schemas in Z are implemented into Cobol by ordinary program variables, that can be included in a Cobol record description.	The refinement into C and Cobol is equally direct and clear.
Operations	Operation schemas	Operation schemas in Z are refined and implemented into C by the definition and subsequent operation of the variables used.	Operation schemas in Z are refined and implemented into Cobol by the definition and subsequent operation of the variables used.	The refinement into C and Cobol is equally direct and clear.

Chapter 4

Refinement: From UML to non-object-oriented implementation languages

This chapter illustrates the refinement of a UML specification to the two non-object-oriented implementation languages C and Cobol.

4.1 Introduction

In structured systems analysis, the emphasis is on the actions, rather than the data, of the product to be built [Sch99]. The data modelling of the data is secondary to that of the actions. Entity-relationship modelling (ERM), however, is a semiformal data-oriented technique for specifying a product. It is amongst others, widely used for specifying databases where the emphasis is on the data. Although actions are needed to access the data, these are less significant when drawing up the model. Entity-relationship modelling is also an element of object-oriented analysis.

There are, however, limitations to the ERM. For example, it is an oversimplification to represent the data model of an organisation solely through entities and their relationships with one another. There is a school of thought that the UML (Unified Modelling Language) class diagrams is an extension of the ERD (Entity Relationship Diagram) notation. UML is a logical modelling tool just as ERDs are. Many modellers choose to stay with ER diagrams because of their simplicity and elegance. However, UML is able to describe relationships that ERDs cannot. Moreover, UML adds *process information* to the data model. UML is considered a more flexible modelling tool than ERDs. The ‘entity’ in ERM is replaced by a ‘class’ (group of ‘objects’) in UML [Dor99].

UML is used mainly for representing an object-oriented analysis and design methodology. But, because UML is considered an extension of ERM, structured systems analysis can also be implemented using UML, if the implementation is done in a non-object-oriented language. This principle is illustrated in the current chapter: We present an overview of

UML, followed by the transformation of a design into an implementation for non-object-oriented languages, namely C and Cobol.

The development of UML, past present and future is discussed in Appendix B. Included in Appendix B are issues regarding a possible standardisation of UML.

4.2 UML in general

4.2.1 Background

UML is the successor to the wave of object-oriented analysis and design (OOA&D) methods that appeared in the late 1980's and early '90s [Fow97]. UML is a modelling language, which in its current state, defines a notation and a meta-model. UML is, however, more than simply a standardisation and discovery of a 'unified' notation [Eri98]. UML also contains new and interesting concepts that are not generally found in the object-oriented community, such as:

- how to describe and use patterns in a modelling language.
- how to use the concept of stereotypes to extend and adapt the language.
- how to provide complete traceability from conceptual models of a system to the executable components in the physical architecture.

One of the main goals of UML is to be independent of particular programming languages and development processes. It supports various development methods without 'excessive difficulty' (see e.g. [Boo97, Rum97, Jac97]). However, understanding UML means not only learning the symbols and their meaning, it means learning object-oriented modelling in the state-of-the-art mode.

In terms of the views of a model, UML defines the following graphical diagrams [Dor99]:

- use case diagrams
 - class diagrams
 - behaviour diagrams
 - statechart diagrams
 - activity diagrams
 - interaction diagrams
 - sequence diagrams
 - collaboration diagrams
-

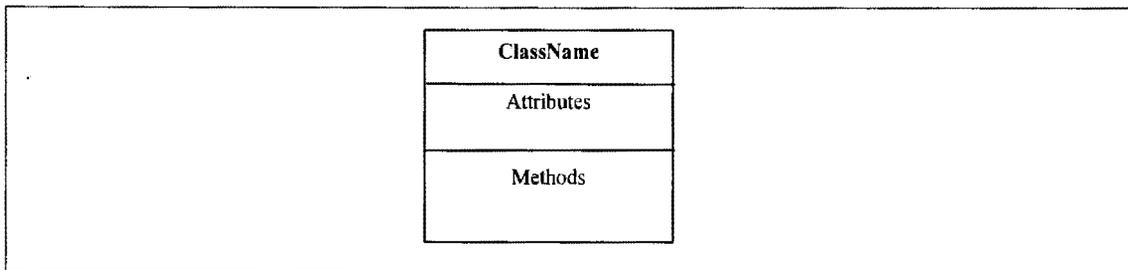


Figure 4.1 Class diagram

- implementation diagrams
- component diagrams
- deployment diagrams.

4.3 Modelling with UML

4.3.1 Class diagrams

The class diagram is a tool for illustrating a system's classes and the relationships between those classes. UML suggests the following structure for the class diagram, where a class is represented by a rectangle divided horizontally into three parts (see Figure 4.1): the class name (top section), the attributes of the class (middle section), and the methods of the class (bottom section) [Pri97].

A class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them. A class diagram shows the static structure of the classes in the system [Eri98, Fow97].

4.3.2 Relationships

Class diagrams are made up of classes and the relationships between them. Eriksson et al. [Eri98] and Douglass [Dou98] identifies the different types of relationships between classes that can be used. These are associations, generalizations, dependencies, refinement, aggregations, and inheritance.

4.3.2.1 Association

An association is a connection between classes, a semantic connection (link) between objects of the classes involved in the association. An association is normally bi-directional, which

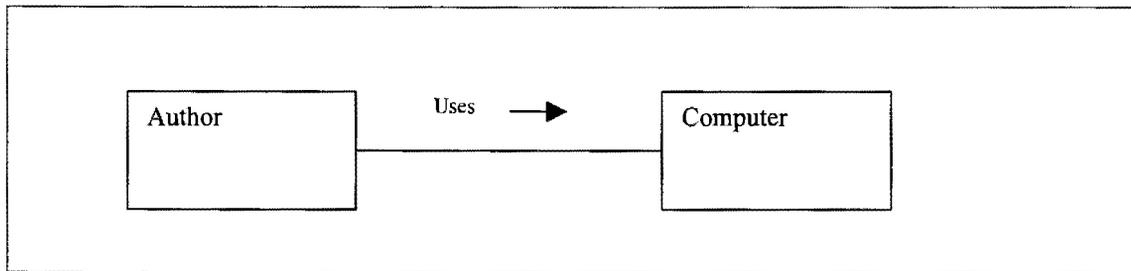


Figure 4.2 An author uses a computer. The Author class has an association to the Computer class [Eri98].

means that if an object is associated with another object, both objects are aware of each other [Eri98].

Adjacent to each connecting line is a label indicating the association between the two classes. An arrowhead can optionally be included indicating the direction of the association. (Uses Figure 4.2).

4.3.2.2 Aggregation

An aggregate is an object that has been decomposed into its component parts. For example, in a graphic user interface system (refer to Figure 4.3) a window may be broken into four parts: title bar, pane, scrollbar, and border [Pri97]. A small diamond shows an aggregate relationship

4.3.2.3 Inheritance

Sometimes, classes differ only in a few attributes. Refer to Figure 4.4. A company rents both lecture rooms and lab rooms.

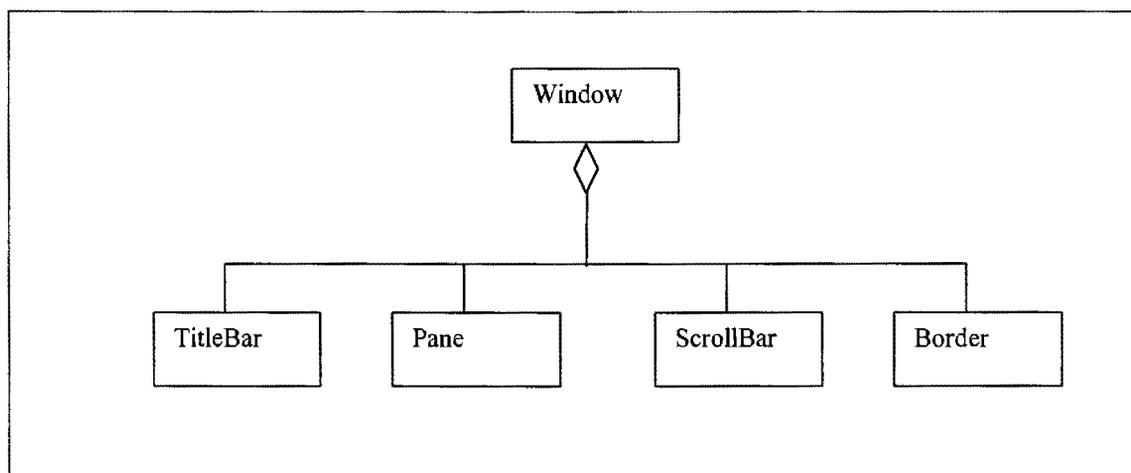


Figure 4.3 An aggregation structure

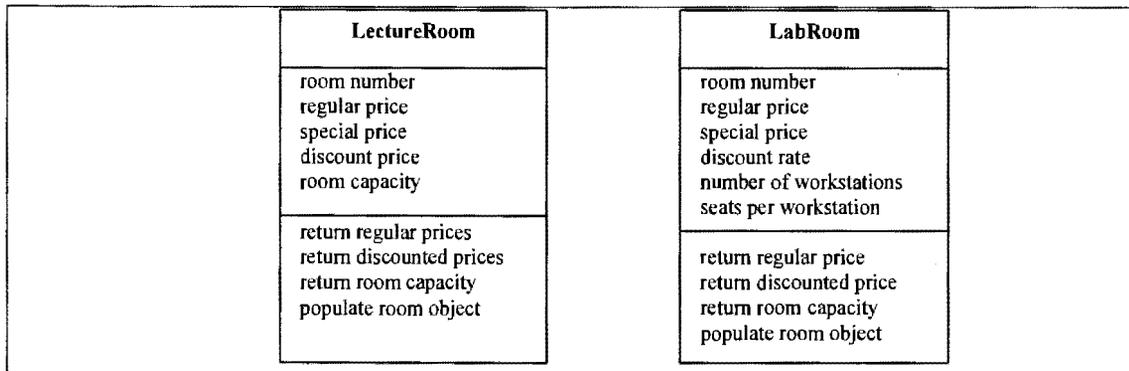


Figure 4.4 Two similar classes

The object data and the methods are almost the same in the two classes. LectureRoom includes the room capacity and LabRoom includes the number of workstations and the seats per workstation. Room capacity for LabRoom must be computed (number of stations time seats per station) so that the method will differ from the method with the same name in LectureRoom. The object data is different for the two classes, therefore, the methods to populate the object will differ, resulting in an undesirable redundancy. For instance, a change in the algorithm for calculating the discounted prices would require changes in the identical code of two different classes.

Object technology resolves this problem with a *structure hierarchy* which, when applied to this model, yields the following form:

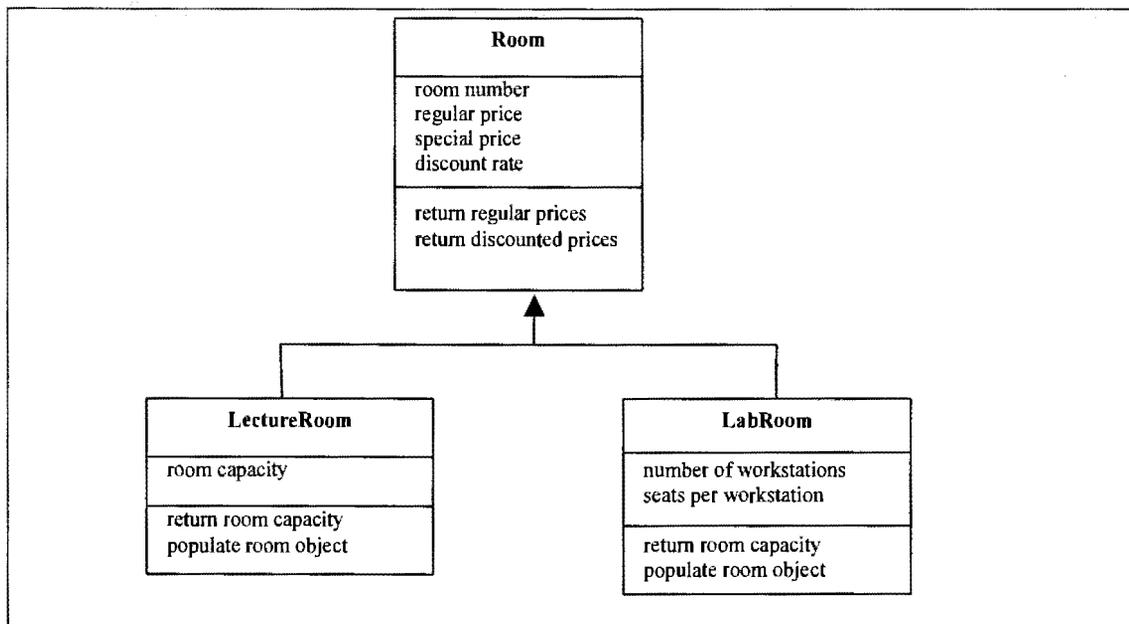


Figure 4.5 Class diagram illustrating the inheritance relationship

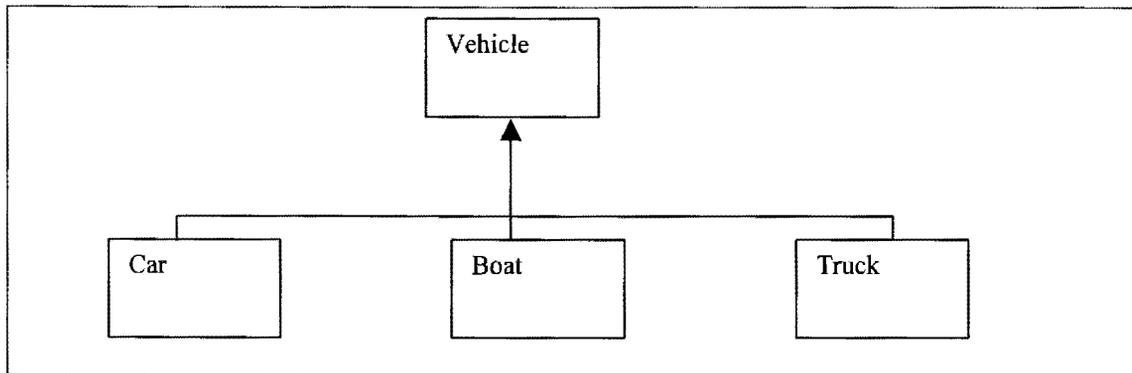


Figure 4.6 Vehicle is a general class (superclass) derived to specific classes (subclasses) via inheritance [Eri98].

Data and methods common to both LectureRoom and LabRoom are moved to a separate class entitled Room. Data and methods unique to each specialized class remain in that class. Room is called the *superclass* and LectureRoom and LabRoom are called *subclasses*. Through a feature called *inheritance*, methods of the superclass are available to the subclasses [Pri97].

By using the subclasses, the programmer sees each subclass as in Figure 4.5. The arrow represents an inheritance hierarchy.

4.3.2.4 Generalisation

A generalisation is a relationship between a general and a specific class. The specific class, called the subclass, inherits everything (e.g. attributes, operations, and all associations) from the general class, called the superclass. For example, in Figure 4.6 the superclass Vehicle is made up of 3 subclasses, namely Car, Boat, and Truck.

4.3.2.5 Dependencies

A dependency relationship is a semantic connection between two *model elements*: one an independent and one a dependent model element. A change in the independent element affects the dependent element. A model element can be a class, a package, a use case (see Section 4.3.4 for a discussion of use cases), and so forth (Fowler [Fow97]).

Dependencies between classes can exist for various reasons, for example:

- One class sends a message to another one.

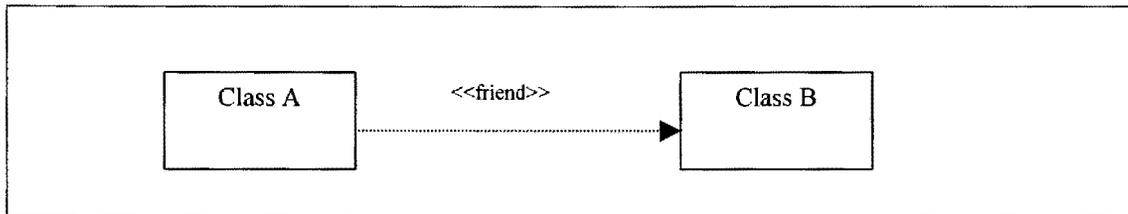


Figure 4.7 A dependency relationship between classes. The type of the dependency is a <<friend>> dependency [Eri98].

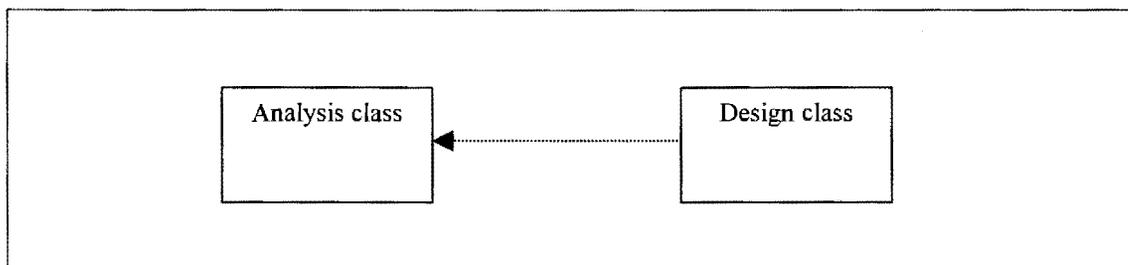


Figure 4.8 Refinement relationship.

- One class has another one as part of its data.
- One class mentions another one as a parameter to a method, etc.

Refer to Figure 4.7 ([Eri98]). A friend dependency means that one model element (Class A) gets special access to the internal structure of the other model element (Class B).

Ideally, only changes to a class's interface should affect any other class. The art of large-scale design involves minimizing dependencies [Fow97]. A dependency between two packages exists if any dependency exists between any two classes in the packages.

4.3.2.6 Refinement

A UML refinement is a relationship between two descriptions of the same entity, but at different levels of abstraction. A refinement relationship can be between a type and a class that realizes it, in which case it is called a realization. Other refinements are relationships between an analysis class and a design class modeling the same thing (see Figure 4.8), or between a high-level description and a low-level description [Eri98].

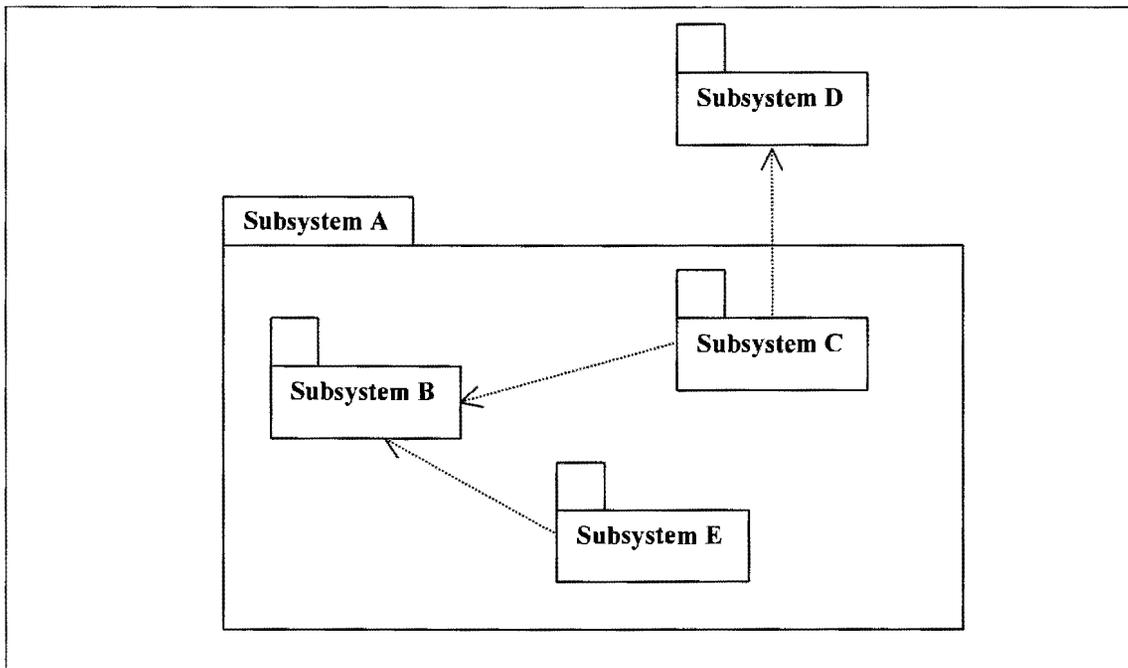


Figure 4.9 Packages

4.3.3 Packages

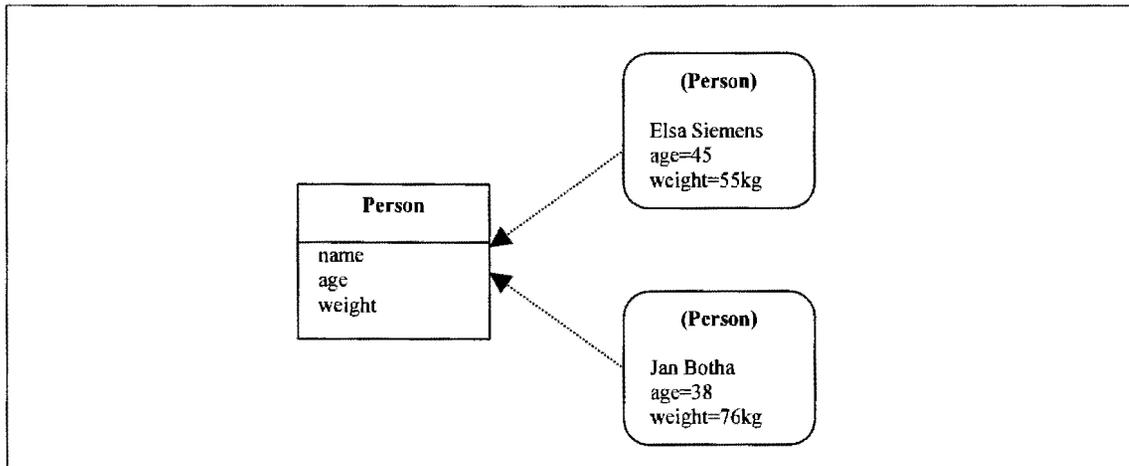
Classes are grouped into higher-level units. In UML this grouping mechanism is called the *package*. Package diagrams show packages of classes and the dependencies among them (Figure 4.9). Subsystem E is dependent on subsystem B. Subsystem C is dependent on subsystem B and D. Subsystem B, C, and E are inside subsystem A. All subsystems are represented as packages.

4.3.4 Instantiations

A class describes a set of object instances of a given form. *Instantiation* relates a class to its instances [Rum91]. Refer to Figure 4.10 for an example: The two persons Elsa Siemens and Jan Botha are instantiations of the class Person.

4.3.5 Use Case Diagrams

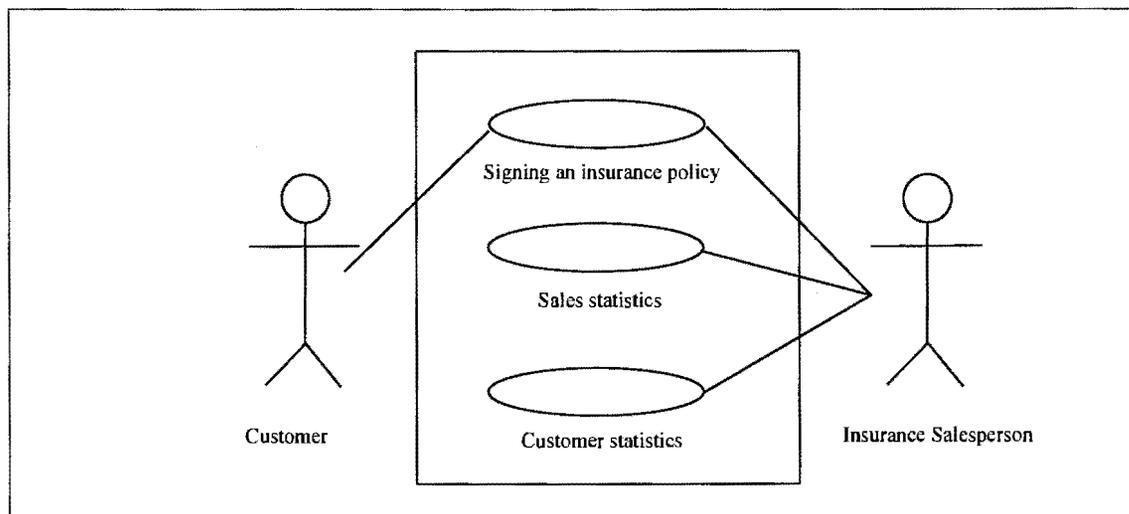
Moving from a requirements definition to a requirements specification (see Chapter 2, page 2-9) is often an *iterative* process and one such method is the construction of *use cases*. The technique of use-case analysis is basically process driven in the sense that the specifier constructs various scenarios on how the system will be used by the users or clients, and in so

**Figure 4.10 Instantiates**

doing identifies various use cases. Modelling with use cases is one way of discovering objects, object classes, and operations during the analysis phase.

A use-case model is described in UML by a *use-case diagram*. A use-case model can be divided into a number of use-case diagrams [Eri98]. A use-case diagram contains model elements for the system like the actors (who play in the use-case scenarios), and the proposed operations identified through a use-case analysis. (Refer to Figure 4.11 for an example of a use-case diagram for an insurance business).

The use cases are represented as ellipses inside a system boundary and have associations with the actors.

**Figure 4.11 Use-case diagram for an insurance business [Eri98]**

4.3.6 Interaction Diagrams

Interaction diagrams are models that describe how groups of objects collaborate in some behaviour [Fow97]. An interaction diagram captures the behaviour of a single use-case. The diagram shows a number of objects and the message that are passed between these objects within the use case. An interaction diagram is used to analyse the behaviour of several objects within a single use-case. There are two kinds of interaction diagrams: sequence diagrams and collaboration diagrams.

4.3.6.1 Sequence diagram

A sequence diagram is also called a message trace diagram. An example is shown in Figure 4.12 [Eri98]. The sequence diagram shows the dynamic collaboration between a number of objects. It indicates a sequence of events or messages sent between the objects. The interaction between the objects are shown at a specific point in the execution of the system. The objects are indicated by vertical lines. Time passes downward in the diagram. The diagram shows the exchanges of messages between the objects over time. The messages are indicated as lines with message arrows between the vertical object lines.

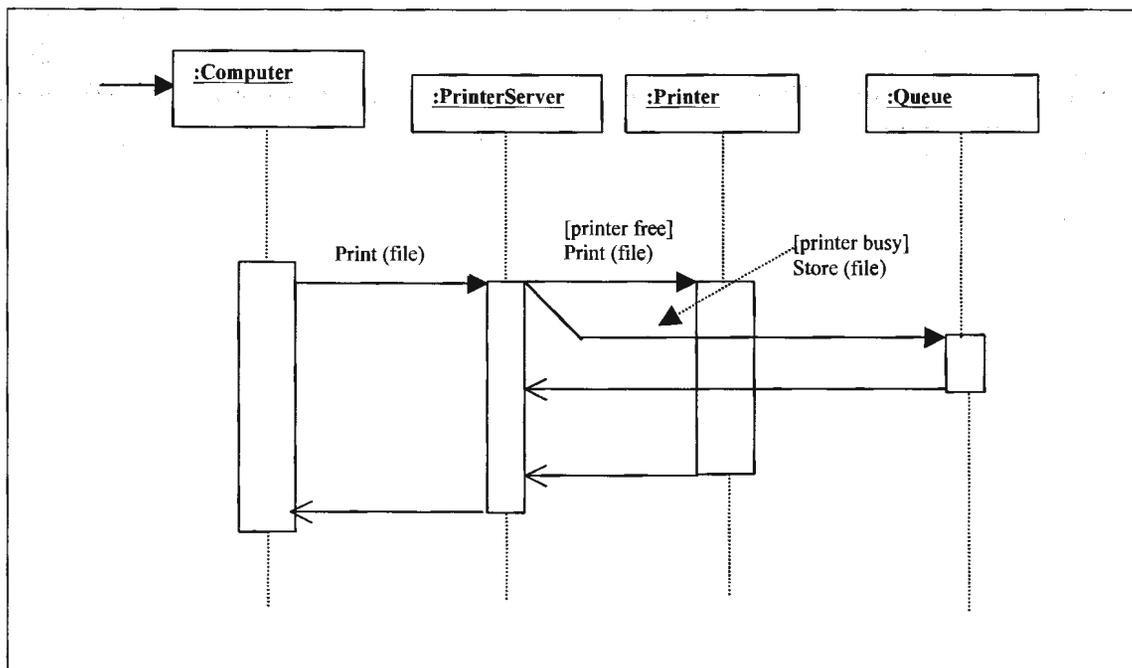


Figure 4.12 [Eri98] A sequence diagram for a print server

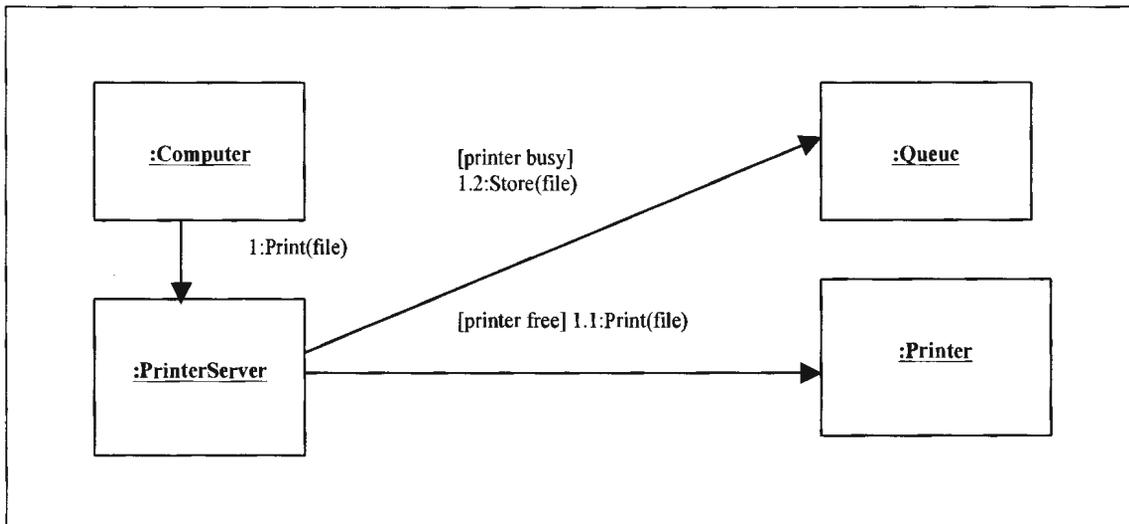


Figure 4.13 [Eri98] A collaboration diagram for a printer server

4.3.6.2 Collaboration Diagram

A collaboration diagram shows a dynamic cooperation between objects. It is drawn as an object diagram, where a number of objects are shown along with their relationships [Eri98] (refer to Figure 4.13 for a collaboration diagram for a printer server).

Message arrows between the objects show the flow of messages between the objects. Labels on the messages indicate the order in which the messages are sent. The labels can also show conditions, iterations, return values, and so on.

4.3.7 State Diagram

State diagrams describe the possible states a particular object can get into and how the object's state changes as a result of events that reach the object [Fow97, Eri98]. Typically, a state diagram complements the description of a class [Eri98]. State diagrams are drawn for classes that have a number of well-defined states and where the behaviour of the class is affected and changed by the different states.

Consider Figure 4.14 for a state diagram of an elevator [Eri98]. The diagram shows all the possible states that objects of the class (elevator) can have, and the events that cause the states to change (A, B, C, D, E, F, and G).

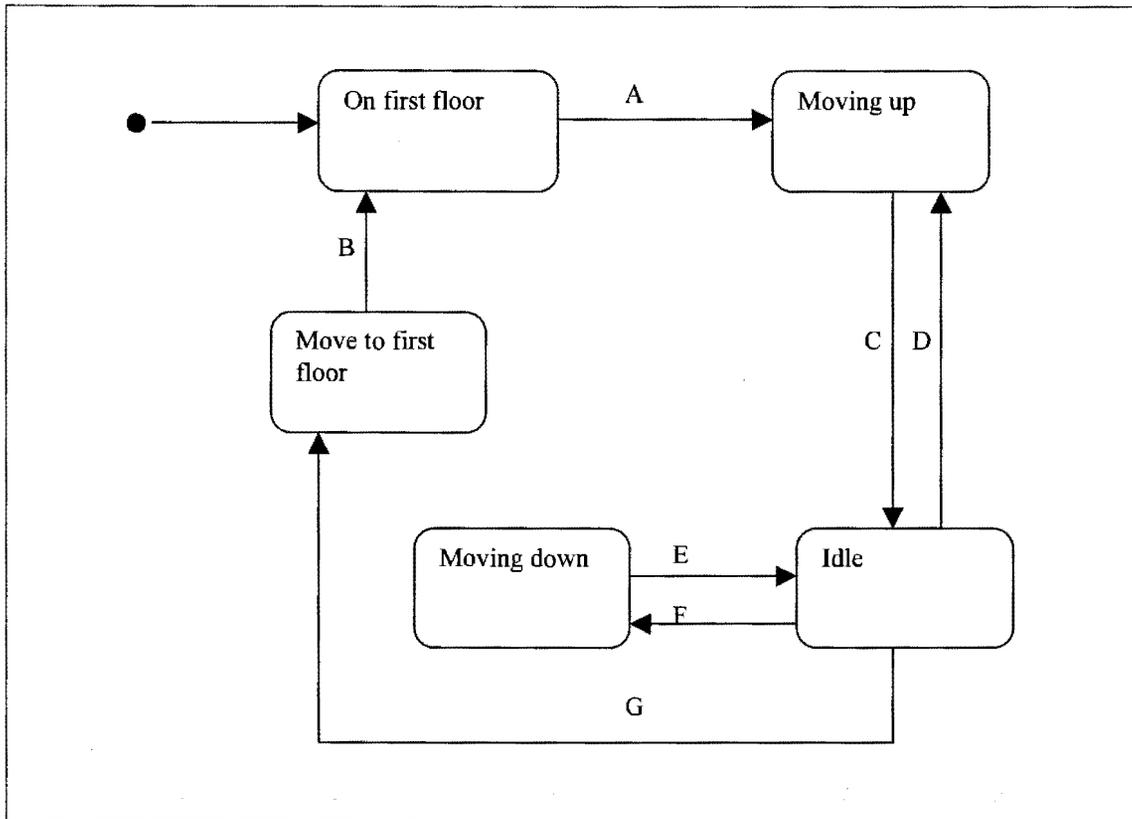


Figure 4.14 A state diagram for an elevator [Eri98, p20]

The elevator starts on the first floor and moves up to some other floor after which it becomes idle or moves down. Eventually it moves back to the first floor. Provision for the top floor has not been made, but it can be deduced that the top floor is a floor from which the elevator can only move down.

The different events are:

- A: go up (floor)
- B: arrive at first floor
- C: arrive at floor
- D: go up (floor)
- E: arrive at floor
- F: go down (floor)
- G: time-out.

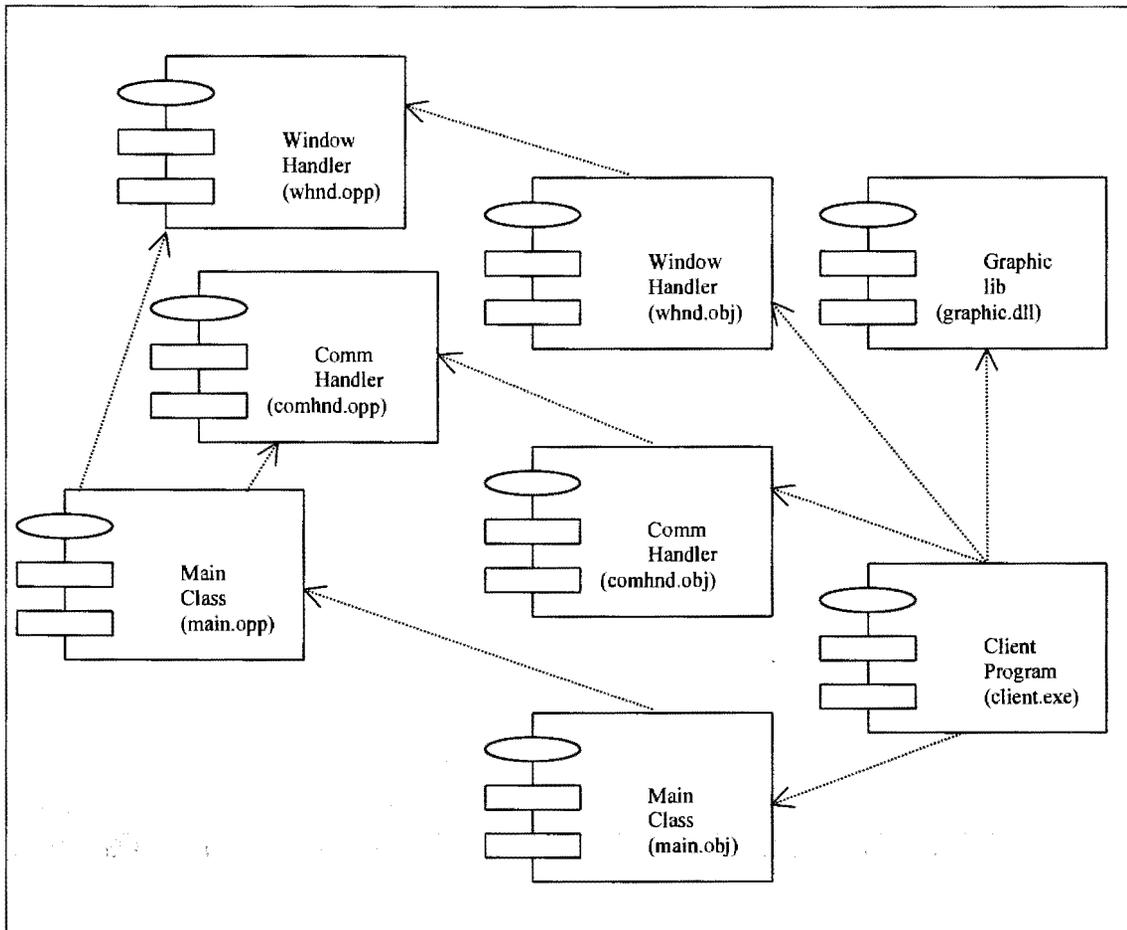


Figure 4.15 A component diagram showing dependencies between code components [Eri98, p.24]

4.3.8 Component Diagram

A component diagram shows the physical structure of the code in terms of code components [Eri98]. A component can be a source code component, a binary component, or an executable component. A component contains information about the logical class(es) it implements, thereby creating a mapping from the logical view to the component view [Eri98].

Dependencies between the components are shown, making it easy to analyse how other components are affected by a change in one component.

Refer to Figure 4.15 for a component diagram showing dependencies between code components. These components are shown as types. A dashed line with an open arrow indicates a dependency connection between components. This means that one component uses another one in its definition.

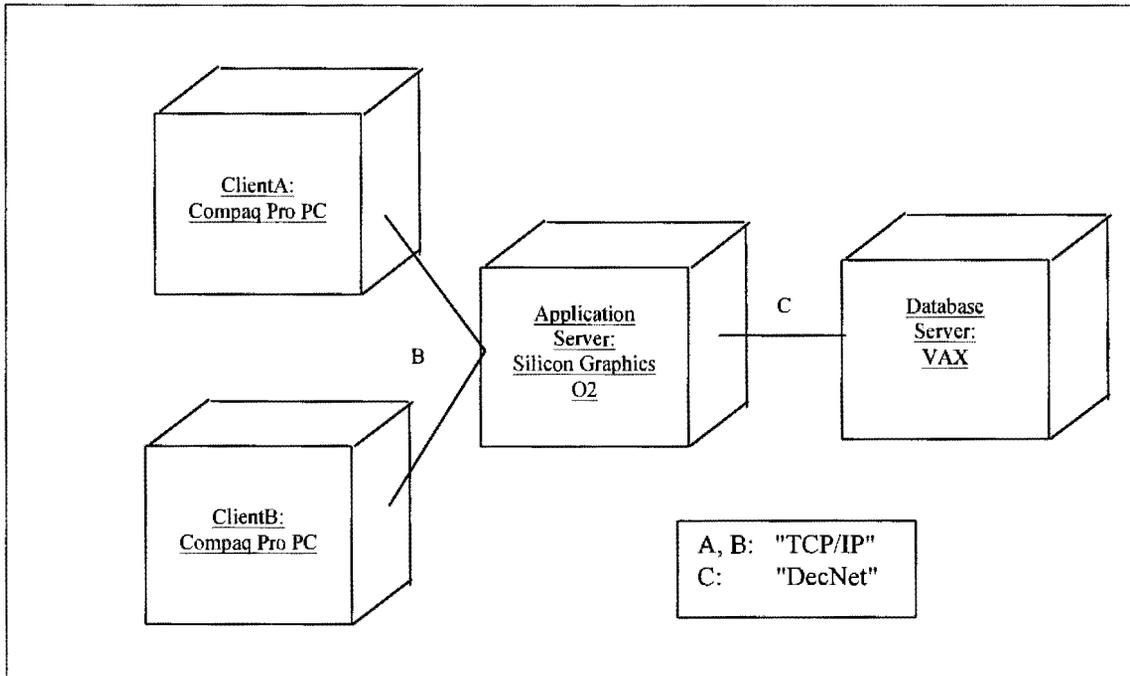


Figure 4.16 A deployment diagram showing the physical architecture of a system

[Eri98, p.25]

4.3.9 Deployment diagram

A deployment diagram shows the physical relationships among software and hardware components in the delivered system. Inside the nodes, executable components and objects are

allocated to show which software units are executed on which nodes. Dependencies between the components are shown (Refer to Figure 4.16).

4.3.10 CRC Cards

A Class-Responsibility-Collaboration (CRC) card describes (on a 3X5 index card) the purpose (responsibility) of the class. With each responsibility, the other classes (collaborators) needed to fulfil the purpose of the class are shown [Fow97].

The CRC card is a responsibility-driven design tool. The front of the card is organised as follows [Pri97]:

- The class name is written at the top of the card.
- The left column contains the class's responsibilities.
- The right column contains the class's collaborators.

A responsibility of a class is reflected by a method of that class.

Refer to the CRC cards of the ITEM system illustrated in Figure 4.17 and described in more detail in Chapter 7 of this work.

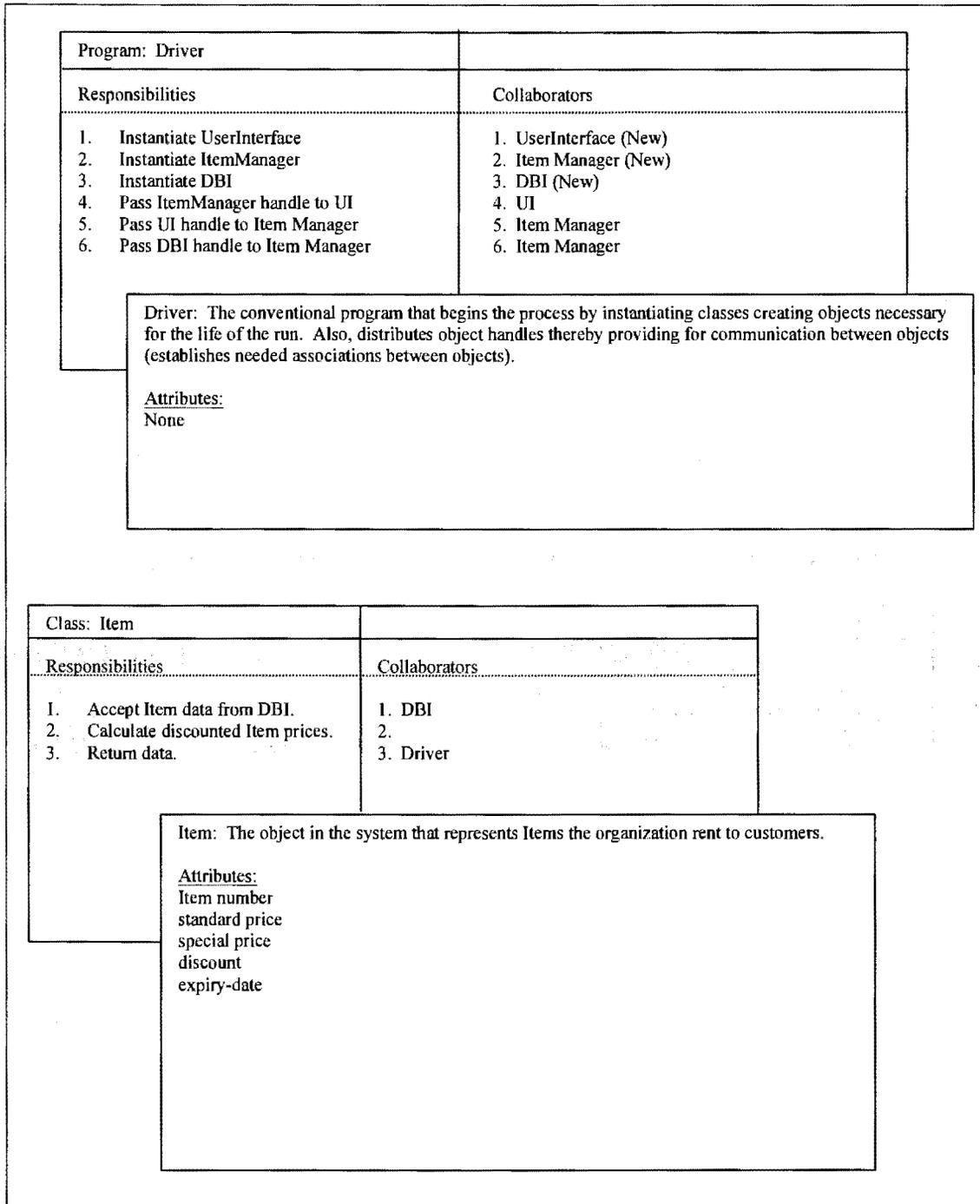


Figure 4.17 CRC cards for the driver and for ITEM (from the ITEM system Chapter 7 [Pri97])

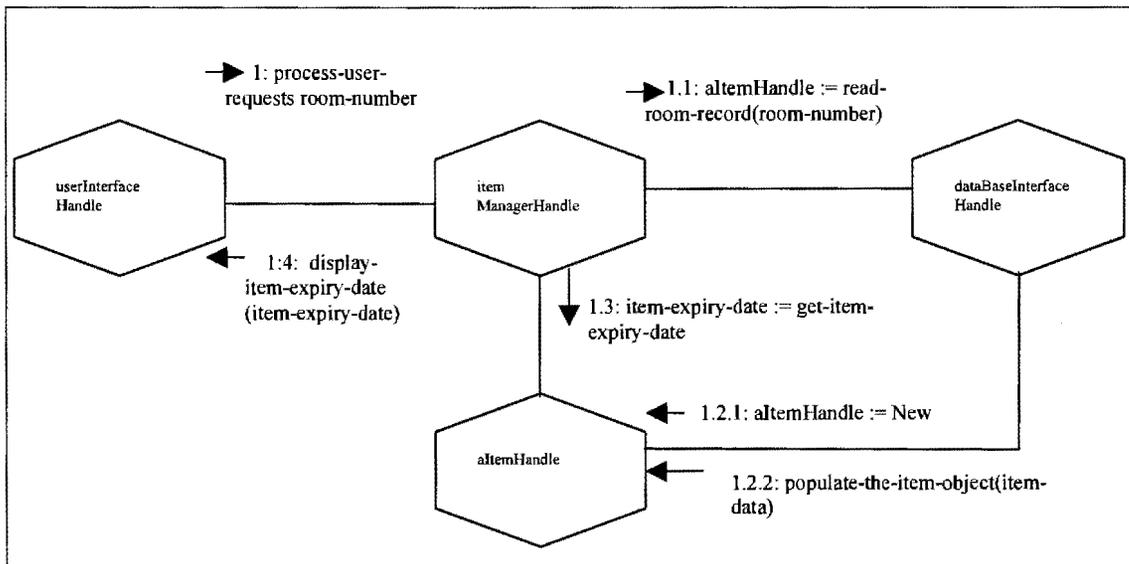


Figure 4.18 Object message diagram: normal request processing (ITEM system, Chapter 7 [Pri97])

4.3.11 Object Message Diagram

An object message diagram is a scenario diagram showing the sequence of messages that implement an operation or a transaction. It shows the objects and links that exist before an operation begins and the objects and links created during the operation [Pri97]. An example is given in Figure 4.18. We will discuss this diagram in Chapter 7.

4.4 Refinement: Translating a Design into an Implementation

For the purposes of this discussion, the refinement process, where a design is implemented into a non-object-oriented language, is illustrated through the following steps:

- Translate classes into structures
- Pass arguments to methods
- Allocate storage for objects
- Implement inheritance in data structures
- Implement method resolution
- Implement associations
- Deal with concurrency
- Encapsulate internal details of classes

The implementation languages used are C and Cobol. Cobol is still widely used in industry (e.g. refer to any recent Information Technology employment advertisement in for example Computing SA, Jobmail, The Star Workplace and PC Format; also it is the main language used by ABSA - a local South African bank), while C with its loose type checking provides the flexibility that allows several important object-oriented and non-object-oriented concepts to be efficiently implemented. The C pointer mechanism and run-time memory allocation facilitate this implementation. Classes, instances, single inheritance and run-time method resolution are implemented in C with little loss of efficiency [Rum91].

For the rest of the discussion we refer to the object model for the graphics editor case study as illustrated in Figure 4.19. The editor permits recursive groups of shapes to be constructed from boxes, circles, diamonds, and lines. A drawing is made up of a set of shapes. Groups can be built from shapes or smaller groups. Items (shapes or groups) that are not part of a group are root items in the drawing and are available for manipulation. A root item is selected by picking one of its embedded shapes with a locator cursor controlled by a mouse. A shape is picked if the cursor point lies within its boundaries. Selected items can be grouped, ungrouped, cut from the drawing, or moved by an offset. Commands are also provided to clear the selections or create new shapes. Shapes are erased by writing over them in the background colour.

Figure 4.19 shows the attributes and operations of each class:

- Abstract operations are indicated by the word *abstract* in the origin class.
 - The  represents a many-to-one association.
 - The  represents an aggregation, where a group consists of a number of items.
 - A  represents generalisation and inheritance, where Box and Circle are refined versions of Shape (super class).
 - Attributes and operations common to a group of subclasses are attached to the super class and shared by each subclass.
-

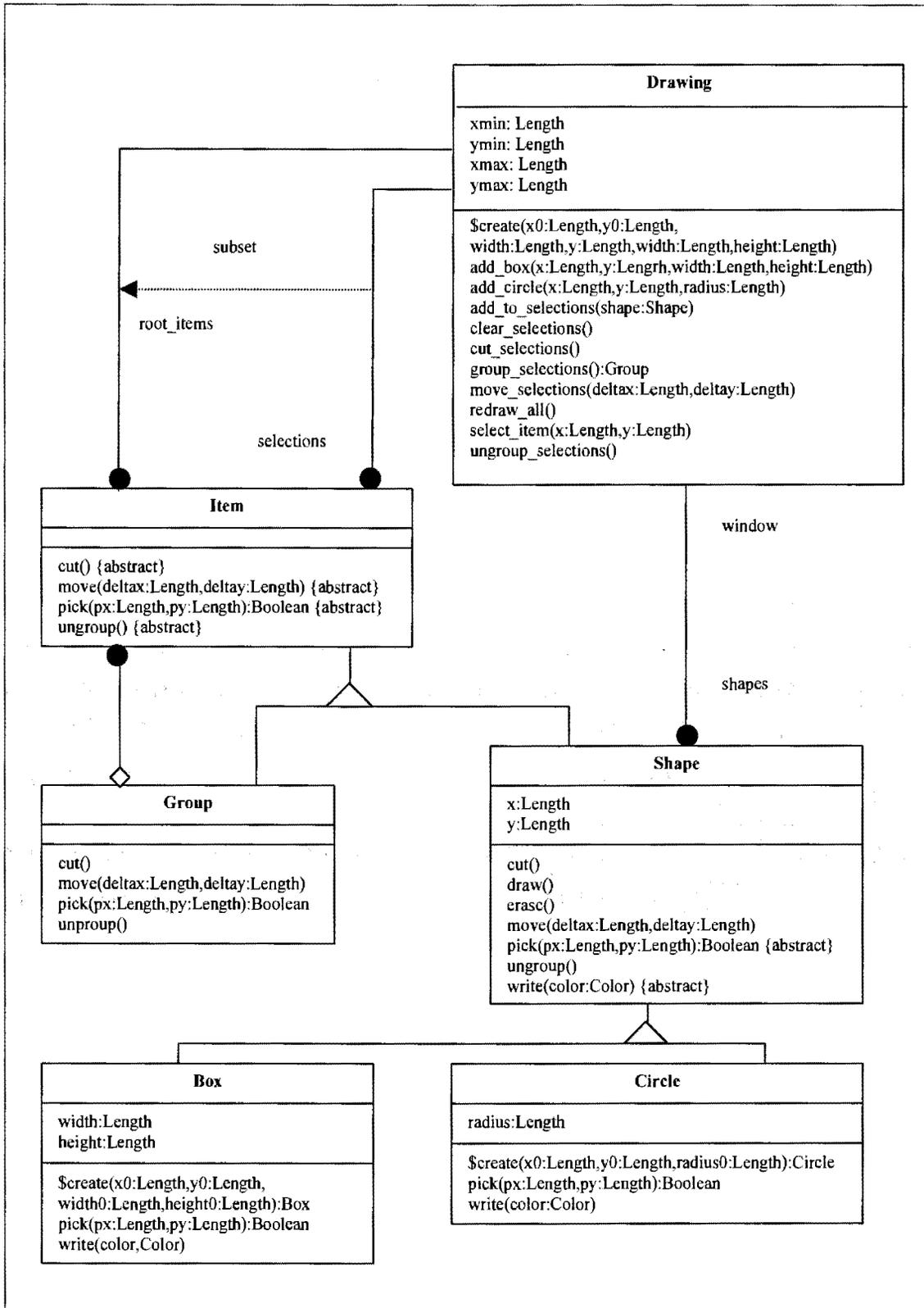


Figure 4.19 Object model for a graphics editor [Rum91, p.298]

4.4.1 Translating classes into data structures

From Figure 4.19 implementations into C and Cobol are deduced. UML does not have a separate 'refinement' stage, and the class names, attributes and methods defined in the UML diagrams in Figure 4.19 are used for the data definitions and methods used in the programming languages.

Each attribute in a class becomes an element in a record. Each attribute belongs to a type, which can be a primitive type, such as *integer*, *real*, or *character*, or it can be a structured type, such as an embedded record structure or a fixed-length array.

4.4.1.1 Implementation into C

Each class in the design becomes a C *struct*. Each attribute defined in the class becomes a field in the C *struct*. The structure for the *Drawing* class is declared as:

```

Struct Drawing
{
    Length xmin;
    Length ymin;
    Length xmax;
    Length ymax;
};

```

Length is a C type (not a class) defined with a C *typedef* statement to provide greater modularity in the definition of values:

```

Typedef float Length;

```

In C a reference to an object can be represented by a pointer to its object record:

```

struct Drawing * drawing;
Length x1 = drawing -> xmin;

```

An object can be allocated statically, automatically, or dynamically because C can compose a pointer to any object, including one embedded within another structure.

4.4.1.2 Implementation into Cobol¹

```

01  Drawing_record.
    03  Adrawing.
        05  Length_xmin          pic 9(2)v99.
        05  Length_ymin          pic 9(2)v99.
        05  Length_xmax          pic 9(2)v99.
        05  Length_ymax          pic 9(2)v99.
    03  Adrawing2 redefines Adrawing.
        05  Adrawing3 occurs 4 times pic 9(2)v99.
    ....
    move Adrawing3(1) to length_x1.          [Assignment]
    ....

```

4.4.2 Passing arguments to methods

Every method has at least one argument, the implicit *self* argument. This argument must be made explicit in a non-object-oriented language. Methods may have additional objects as arguments. Some of the arguments may be simple data values and not objects. In passing an object as an argument to a method, a reference to the object must be passed if the value of the object can be updated within the method [Rum91].

4.4.2.1 Implementation into C

An object should always be passed by a pointer in C, since passing a pointer to an object structure is more efficient and provides a uniform access mechanism for both query and update operations.

The objects Drawing and Shape are used as arguments in the Drawing_add_to_selections method:

```

Drawing_add_to_selections (self, shape)
    struct Drawing * self;
    struct Shape * shape;

```

¹ For the purposes of this chapter 'Cobol' refers to structured Cobol and not Object-Oriented.

4.4.2.2 Implementation into Cobol

The Drawing_record (01 Drawing_record.) and Shape_record (01 Shape_record.) are sent as arguments from the calling program to the subroutine sub_add_to_selections with the call “sub_add_to_selections” statement.

```

01  Drawing_record.
    03  Adrawing.
        05  Length_xmin          pic 9(2)v99.
        05  Length_ymin          pic 9(2)v99.
        05  Length_xmax          pic 9(2)v99.
        05  Length_ymax          pic 9(2)v99.
    03  Adrawing2 redefines Adrawing.
        05  Adrawing3  occurs 4 times pic 9(2)v99.
01  Shape_record.
    05  Length_x          pic 9(2)v99.
    05  Length_y          pic 9(2)v99.
    .....
100_process-records.
    .....
    call “sub_add_to_selections” using Drawing_record, Shape_record.
    .....

Subroutine
    .....
program-id.      sub_add_to_selections.
    .....

Linkage section.
01  Ls_drawing_record.
    (The same as Drawing_record)
01  Ls_shape_record.
    (The same as Shape_record)
procedure division using Ls_drawing_record, Ls_shape_record.
100_add_to_selections.

```

4.4.3 Allocating objects

Objects can be allocated statically at compile time, or dynamically during run time (from a stack or a heap). Statically allocated objects are implemented as global variables. Their lifetime is the duration of the program [Rum91].

Temporary and intermediate objects will be implemented as stack-based variables. Stack-based variables are automatically allocated and de-allocated.

Dynamically allocated objects are used, normally, when their specific size is not known at compile time. A general object can be implemented as a data structure allocated on request at run time from a heap.

Sets of related objects can be implemented as arrays, linked lists, or other data structures.

4.4.3.1 Implementation into C

Global objects can be declared as top-level *struct* variables. These can be initialised at compile, e.g:

```
struct Drawing outer_drawing = {0.0, 0.0, 8.5, 11.0};
```

When a method is called, the address of the variable (indicated by *&*, e.g. *&outer_drawing*) must be passed.

Objects are allocated dynamically using the C functions *malloc* or *calloc*:

```
struct Drawing * create_drawing (xmin, ymin, width, height)
    Length xmin, ymin, width, height;
{
    struct Drawing * drawing;
    drawing= (struct Drawing *) malloc (sizeof (struct Drawing));
    drawing->xmin = xmin;
    drawing->ymin = ymin;
    drawing->xmax = xmin + width;
    drawing->ymax = ymin + height;
```

```

    return drawing;
}

```

An object is de-allocated when it is no longer needed, using the C *free* function. Temporary and intermediate objects are allocated as ordinary C *automatic* variables within a function, a body, or a block.

4.4.3.2 Implementation into Cobol

```

.....
01  Drawing_record.
    03  Adrawing.
        05  Length_xmin          pic 9(2)v99.
        05  Length_ymin          pic 9(2)v99.
        05  Length_xmax          pic 9(2)v99.
        05  Length_ymax          pic 9(2)v99.
    03  Adrawing2 redefines Adrawing.
        05  Adrawing3 occurs 4 times pic 9(2)v99.
01  Outer_drawing_record.
    03  Bdrawing.
        05  Blength_xmin          pic 9(2)v99 value 0.0.
        05  Blength_ymin          pic 9(2)v99 value 0.0.
        05  Width                  pic 9(2)v99 value 8.5.
        05  Height                  pic 9(2)v99 value 11.0.
    03  Bdrawing2 redefines Bdrawing.
        05  Bdrawing3 occurs 4 times pic 9(2)v99.
.....
compute Length_xmin = Blength_xmin.
compute Length_ymin = Blength_ymin.
compute Length_xmax = Length_xmin + width.
compute Length_ymax = Length_ymin + height.
.....

```

4.4.4 Implementing inheritance

There are several ways to implement data structures for inheritance in a non-object-oriented language [Rum91]:

- *Avoid it.* A Class not needing an inheritance can be implemented simply as a record.
- *Flatten the class hierarchy.* Each concrete class is expanded as an independent data structure during implementation and every inherited operation must be implemented as a separate method on the particular concrete class. Flattening the hierarchy introduces duplication, but the use of language constructs such as C macros can alleviate this.
- *Break out separate objects.* A common group of attributes can be pulled out of all the sub-classes containing such group and implemented as a separate object with a reference to it stored within each sub-class.

4.4.4.1 Implementation into C

Inheritance can be simulated in C by duplicating the structure of a super class in a subclass and passing a sub-class object to a super class method, and can be implemented using variant records. The declaration for the super class is embedded as the first part of each sub-class declaration. The first field of each *struct* is a pointer to a class descriptor object, shared by all direct instances of a class. The class descriptor object is a *struct* that contains the class attributes, including the name of the class and the methods for the class [Rum91].

The class **Shape** is an abstract class, with concrete sub-classes **Box** and **Circle**. The C declarations for the classes **Shape**, **Box** and **Circle** are:

```

struct Shape
{
    struct ShapeClass * class;
    Length x;
    Length y;
};

```

```

struct Box
{
    struct Boxclass * class;

```

```

    Length x;
    Length y;
    Length width;
    Length height;
};
struct Circle
{
    struct CircleClass * class;
    Length x;
    Length y;
    Length radius;
};

```

Note that the first field of each structure is a pointer to the class descriptor for the actual class of each object instance.

4.4.4.2 Implementation into Cobol

For structured Cobol, Box_record and Circle_record ‘inherit’ SLength_x and SLength_y from Shape_record as indicated. (Note that Object-Oriented Cobol implements inheritance, but structured Cobol does not.)

```

.....
01  Shape_record.
    05  SLength_x          pic 9(2)v99.
    05  SLength_y          pic 9(2)v99.
01  Box_record.
    05  BLength_x          pic 9(2)v99.
    05  BLength_y          pic 9(2)v99.
    05  B_width            pic 9(2)v99.
    05  B_height           pic 9(2)v99.
01  Circle_record.
    05  CLength_x          pic 9(2)v99.
    05  CLength_y          pic 9(2)v99.
    05  C_radius           pic 9(2)v99.

```

```

.....
move SLength_x to BLength_x, CLength_x.
move SLength_y to BLength_y, CLength_y.
.....

```

4.4.5 Implementing method resolution

Method resolution can be addressed in several ways [Rum91]:

- *Avoid it.* When each operation is defined only once in the class hierarchy and not overridden, then there is no polymorphism², and therefore no need for run-time method resolution.
- *Resolve methods at compile time.* If the class of each object is known at compile time, then the correct method can be determined and called directly, avoiding the need for run-time method resolution.
- *Resolve methods at run time.* Dynamic method resolution is required if there is a collection of mixed classes to which an abstract operation needs to be applied.

4.4.5.1 Implementation into C

Run-time method resolution is efficient in C, using function pointers stored in a class descriptor object for each class. Any method that can be resolved at compile time can be implemented as straight C function calls. Many operations are implemented only once as methods and never overridden, and therefore do not need run-time method resolution. In most applications operations can be resolved at compile time.

Each class descriptor is a C *struct* containing all the operations defined in the class or inherited from a super class. The following code shows the declaration for the class descriptors for *Item*, *Shape*, *Box* and *Circle* (Figure 4.19):

² Polymorphism means that the same operation may behave differently on different classes, e.g. the addition of 2 integers or 2 real numbers.

```
struct ItemClass
{
    char * class_name;
    void (* cut) ();

    void (* move) ();
    Boolean (* pick) ();
    Void (* ungroup) ();
};
```

```
struct ShapeClass
{
    char * class_name;
    void (* cut) ();
    void (* move) ();
    Boolean (* pick) ();
    void (* ungroup) ();
    void (* write) ();
};
```

```
struct BoxClass
{
    char * class_name;
    void (* cut) ();
    void (* move) ();
    Boolean (* pick) ();
    void (* ungroup) ();
    void (* write) ();
};
```

```
struct CircleClass
{
    char * class_name;
    void (* cut) ();
    void (* move) ();
    Boolean (* pick) ();
    void (* ungroup) ();
```

```

void (* write) ();
};

```

The correct methods can be determined by examining the object model. For example, class *Box* and *Circle* inherit operations *move*, *cut* and *ungroup* from class *Shape* but override operations *pick* and *write* with their own methods.

4.4.5.2 Implementation into Cobol

Routines *Box_rtn* and *Circle_rtn* ‘inherit’ operations *move*, *cut* and *ungroup* from routine *Shape_rtn* but override operations *pick* and *write* with their own methods.

```

.....
01  Drawing_record.
.....
01  Item_record.
.....
01  Shape_record.
.....
01  Box_record.
.....
01  Circle_record.
.....
perform Item_rtn.
perform Shape_rtn.
perform Box_rtn.
perform Circle_rtn.
.....
Item_rtn.
perform cut_rtn.
perform move_rtn.
perform pick_rtn.
perform ungroup_rtn.
Shape_rtn.  Generic shape operations.
perform shape_cut_rtn.

```

perform shape_move_rtn.
perform shape_pick_rtn.
perform shape_ungroup_rtn.
perform shape_write_rtn.

Box_rtn. *Draw a box.*

perform shape_cut_rtn.
perform shape_move_rtn.
perform box_pick_rtn.
perform shape_ungroup_rtn.
perform box_write_rtn.

Circle_rtn. *Draw a circle.*

perform shape_cut_rtn.
perform shape_move_rtn.
perform circle_pick_rtn.
perform shape_ungroup_rtn.
perform circle_write_rtn.

.....

4.4.6 Implementing associations

4.4.6.1 Mapping associations to pointers

To implement binary associations, each role of an association is mapped into an object pointer stored as a field of the source object record. Each object contains a pointer to an associated object (for one or zero-one multiplicity), or a pointer to a set of associated objects (for multiplicity greater than one). A set may be implemented using an appropriate available data structure (e.g. linked list or array). The association can be implemented in one direction or in both directions [Rum91, Hel98].

4.4.6.2 Implementing association objects

Associations are implemented as pointers from one object to another. Associations can also be implemented as distinct objects containing a set of paired values. However, if an association relates more than two classes, it cannot be mapped into pointers and a separate object must be used.

4.4.6.3 Implementation into C

A binary association is usually implemented as a field in each associated object, containing a pointer to the related object or to an array of related objects. The many-to-one association between *Item* and *Group* are implemented as [Rum91, Hel98]:

(Refer to figure 4.19)

```

struct Item
{
    struct ItemClass * class;
    struct Group * group;
};

struct Group
{
    struct GroupClass * class;
    int item_count;
    struct Item * * items;
};

```

4.4.6.4 Implementation into Cobol

Structured Cobol has no pointers. The number of items in a group can be established by adding 1 to the item count (nitem) each time an item record is read. The group to which an item belongs can, for example, be identified in the item and group record descriptions with an additional field.

```

.....
Fd   Item_file.
01   Item_record
      03   group_number                pic 9(2).
.....
Fd   Group_file.
01   Group_record.
      03   group_number2              pic 9(2).
.....
01   group_no                          pic 9(2) value 0.
01   nitem      occurs 1100 times      pic 9(2) value 0.
.....

```

To determine the number of items for a specific group:

```
if group_number = group_number2
move group_number to group_no
add 1 to nitem(group_no).
.....
```

4.4.7 Dealing with concurrency

Many languages do not explicitly support concurrency. C is a language that lacks concurrent constructs. The only way to create concurrency is to ask the operating system to create a parallel task. Concurrency is handled by operating calls in most languages [Rum91].

4.4.8 Encapsulation

Encapsulation of data representation and method implementation is one of the major themes of object-oriented programming. Object-oriented languages provide constructs to encapsulate implementation.

4.4.8.1 Implementation into C

C encourages a loose programming style that may be harmful to encapsulation. The following steps improve encapsulation:

- Avoid the use of global variables.
- Package the methods for each class into a separate file.
- Treat objects of other classes as type "void *".

4.4.8.2 Implementation into Cobol

Encapsulation is available in Object-Oriented Cobol. In structural Cobol encapsulation may be simulated with the use of subprograms and nested subprograms.

4.5 Comparison between C and Cobol

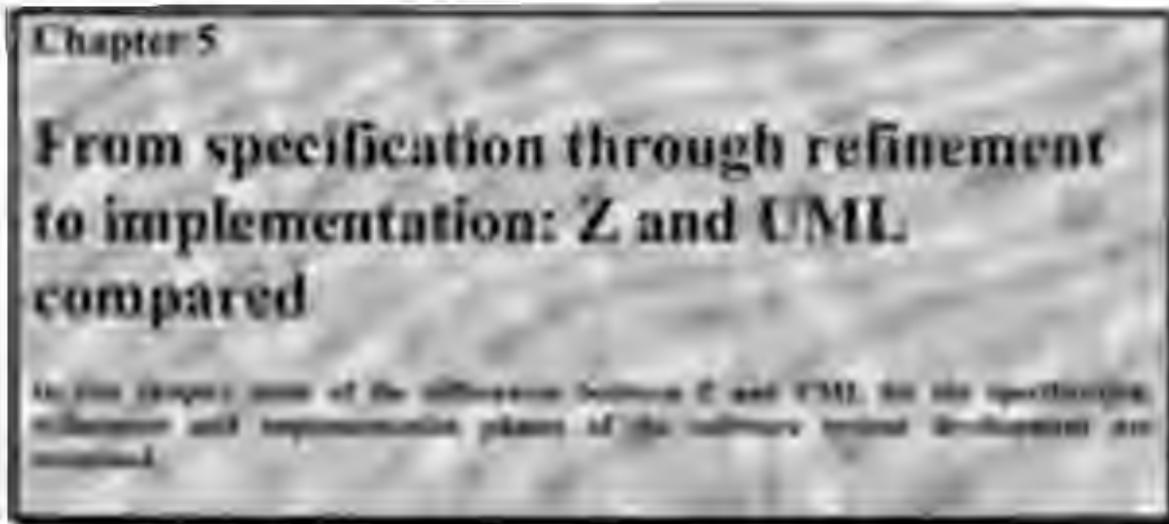
Because the refinement as such is the same whether C or Cobol is used, no comparison between those two languages as far as the refinement is concerned can be deduced. However, from the above refinement from UML structures into both C and Cobol the following similarities and differences between the two languages emerge:

- C implements pointers that are absent from Cobol. Pointers are, however, available in object-oriented Cobol.
 - The C *struct* is implemented as a record structure in Cobol.
 - In C an object can be allocated statically, automatically, or dynamically because C can compose a pointer to any object, including one imbedded within another structure. For structural Cobol to allocate an object imbedded within another structure, the use of tables (or arrays) can be used with a redefines clause (refer to Sections 4.4.1.1 and 4.4.1.2).
 - To pass an argument in C, a pointer must be used. Arguments (objects, data structures, or record structures) in structured Cobol can be passed from a calling program to a called program via the 'call' and 'using' statements (refer to Sections 4.4.2.1 and 4.4.2.2).
 - Variables in a C *struct* can be initialised with, for example, the following type of statement: **struct** Drawing outer_drawing = {0.0, 0.0, 8.5, 11.0}. In structured Cobol variables are initialised with the 'value' clause, or with 'move' statements in the procedure division. (Refer to sections 4.4.3.1 and 4.4.3.2).
 - Inheritance is not directly implemented in structured Cobol. Inheritance can be simulated with move statements in structured Cobol. (Refer to sections 4.4.4.1 and 4.4.4.2).
 - Inheritance of operations are not automatic in structured Cobol. If one routine inherits a statement from another routine, it has to be explicitly stated, as indicated in section 4.4.5.2. In C subclasses implicitly inherits operations from super classes (refer to Section 4.4.5.1).
 - There is no pointer mechanism to implement associations in structured Cobol. The use of arrays can be employed to implement associations. (Refer to sections 4.4.6.3 and 4.4.6.4).
 - In C, arrays can change in length and increase in size dynamically. In Cobol, an array is sized at compile time according to some constant, or a variable that receives a value from the outside (refer to sections 4.4.6.3 and 4.4.6.4).
 - Cobol's English-like syntax may be easier to follow than the more cryptic C syntax.
 - The decision as to whether a programmer should use C or Cobol is dependent on the system, and the personal preference of the programmer, as each language has its own advantages and disadvantages.
-

4.6 Summary

A broad background on the UML modelling technique was presented.

Guidelines for mapping concepts into an non-object-oriented language were discussed. The concepts of translating classes into data structures, passing arguments to methods, allocating objects, implementing inheritance, method resolution and associations, dealing with concurrency and encapsulation were investigated. Examples were given in both C and structured Cobol. A brief comparison between C and Cobol was given. Concluding as to which of C or Cobol is the preferred language to use for the implementation of system is not always that easy, since each has its own strengths and weaknesses.



5.1 Introduction

This chapter focuses on non object-oriented concepts, starting with a general comparison between Z and UML. This is followed by a broad overview on specification, refinement and implementation in Z and UML where the main differences are highlighted. Thereafter the comparison between UML and Z focuses mainly on specification, refinement and implementation for:

- UML classes and subclasses (schemas representing states in Z)
- Relationships (also called associations):
 - * one-to-one relationships
 - * one-to-many and many-to-one relationships
 - * resolved many-to-many relationships.

Our specifications are refined into both C and Cobol.

For the comparison selected examples are taken. Because of the limited scope of this study, in the comparison not all the possibilities are explored. For every category an example is discussed. The main purpose of this comparison is to illustrate the main trends and threads that run throughout the specification phase, the refinement process and the implementation phase as far as the comparison of Z and UML is concerned.

In the refinement and implementation of specifications into programming code, it is rarely necessary to develop an entire system in a completely formal way as shown in Chapter 3. The programming problems that arise within a single project usually present a range of

difficulty. Often large parts of the project may be so routine that there is no need for any formal description in Z, hence only a fraction might be refined to a detailed Z specification. In this fraction only a page or two of code might be derived and verified. The rest can be translated to code intuitively and then verified by inspection [Jak97].

Therefore, in translating a Z specification into C or Cobol, it is not always necessary to follow all the steps indicated in Chapter 3.

This chapter focuses on non-object-oriented concepts.

5.2 A general comparison: Z and UML

The Z notation uses mathematical concepts, particularly set theory, to specify data and operations. This allows for reasoning about systems, e.g. checking the consistency of the data and the various operations, as well as verifying the correctness of subsequent system development during refinement. The specifications of the system's static state and operations are precise and abstract and allow no ambiguity. The most common approach to writing Z is to specify and initialise the system state and then to define operations on it [Pol92, Fow97].

UML is based on graphical notations for expressing a wide variety of concepts relevant to a problem domain. Visualisation and modelling are often useful techniques in the battle against system complexity and UML is a well-defined and widely accepted response to that need. Diagrammatic structured analysis techniques have a proven record as a communication medium for non-experts and are used to present the specification even after formal definition. While these notations are intuitive and easy to understand by users, they often lack sufficient expressive power [Boo97, Ham97, Jac97, Pol92, Rum97].

Neither Z nor UML is a method – both are modelling languages (see e.g. [Bar94] and [Fow97]).

Combining Z and a graphical notation might be the solution to overcome some negative aspects of both approaches. For UML, the consistency of the data and operations on the data is not as easy to check as is the case for Z. Also, according to *Polack* [Pol92], a SSADM (structured systems analysis and design method) specification is improved by incorporating the precision of Z and by insights gained into the system during formalisation. The

production and accessibility of the formal specification are helped by the structure and easily read graphical products of SSADM (e.g. ER diagrams).

5.3 A comparison of Z and UML in terms of specification, refinement and implementation

This section takes the form of a brief discussion of Z and UML specification, refinement and implementation, followed by comparisons between Z and UML. The basic refinement steps for Z and UML are as given in chapters 3 and 4, respectively. The chief objective of this discussion is to compare some of the main refinement and implementation issues of Z and UML, for non-object-oriented designs.

The main areas of comparison are:

- Specification, refinement and implementation in general
- Implementing the classes in UML and the schemas in Z
- Implementing the class invariant (in UML) and the state invariant (in Z)
- Implementing associations (i.e. relationships).

Before commencing with the discussion, the following definitions are given:

For UML

A *class diagram* describes the types of objects in the system and the various kinds of static relationships that exist among them [Fow97].

A *class invariant* is a condition that a class must satisfy at all stable times [Rum91].

For Z

A *state schema* defines variables and the relationship between the variables [Ran94] (refer to Section 3.2.1).

An *abstract data type* defines the structure of variables belonging to the type as well as a set of permissible operations that can be carried out on these variables [Ran94].

5.3.1 Specification, refinement and implementation in general

5.3.1.1 UML applications

UML specifications can be verified by for example peer review, checking syntax and types and reasoning about the correctness of the specification (refer to Section 3.3). If the requirements are clear, and the formal specification (UML) is well organised much of the specification can be validated by inspection. If a UML specification is not formalised, the refinement of the specification is done directly into a suitable implementation language, e.g. C (see chapters 3 and 4 of this work).

5.3.1.2 Z applications

In refining a Z specification we perform the following steps:

- Validate the specification to ensure its internal consistency (refer to Section 3.3).
- Transform the abstract data types into concrete ones (e.g. move from a total function to an indexed array).
- Transform the abstract operations into concrete operations (e.g. move from a set-theoretic union to an array insertion).

The applicability property ensures that when the abstract operation can be used, so can the concrete one (see sections 3.3, 3.4 and 3.5). The correctness preserving property of the refinement process ensures that the concrete operations have the same effect as the abstract ones. [Ran94].

In Z, according to Potter [Pot96] and Woodcock [Woo96], data and operation refinement can be looked at as that part of the development process that corresponds to the design phase of the traditional software life cycle. Ways to represent the abstract data structures that will be more amenable to computer processing are chosen, as well as the translation of abstract operations into corresponding concrete operations. The concrete operations are still, however, expressed in the language of schemas and describe only the relationships among the components of before and after states. This does not indicate how such changes of state are to be expressed in an implementation language. Operation decomposition is the subsequent process of converting state descriptions into executable instruction sequences.

Operation decomposition can be carried through to the level of individual programming language instructions in the Z notation by using the schema calculus. With the addition of further schema operators, the decomposition process can continue until schemas are produced that correspond directly to programming language instructions. The goal of operation decomposition is to produce a program from which all schemas have been removed. A schema can only be replaced by programming language text when the change of state described by the schema is so simple that it corresponds to an instruction or a group of instructions in that language [Pot96, Woo96].

5.3.1.3 Comparison

For Z, the specifications are validated and verified, followed by data and operation refinement, after which the operation decomposition follows. The result is a program from which all schemas have been removed. For UML, the refinement leads directly into the implementation language, without the processes of validation, verification, data refinement, operation refinement and operation decomposition. The only time when the latter five processes are included in the UML refinement into an implementation language is when the UML is combined with a formal notation (see e.g. [Pai99b]).

5.3.2 Implementing Z states and UML classes

5.3.2.1 UML applications

Each class is implemented as a single contiguous block of attributes – a record structure [Rum91] (refer to Section 4.4.1 for the details of the implementation). Each class in the design becomes a C *struct*. Each attribute defined in the class becomes a field in the C *struct*.

5.3.2.2 Z applications

State schemas are usually implemented by data structures whose contents can change frequently [Jak97] (refer to Section 3.5.7 for the details).

5.3.2.3 Comparison

- The state schema in Z and the class in UML can both be implemented by a C structure (or record in other languages).
 - For Z, the members of the structure are the state variables of the schema. For UML, each attribute defined in the class becomes a member in the C structure, that is, becomes fields in the C *struct*.
-

5.3.3 Implementing a Z state invariant and a UML class invariant

5.3.3.1 UML applications

The class invariant must be satisfied at all times. Conditions and invariants are a part of the class declaration and must be obeyed by all descendent classes. If they are violated at runtime, an exception occurs. This exception causes the faulty operation to fail or executes an exception handler for the class if the programmer provides one [Ran94].

5.3.3.2 Z applications

Similarly, a Z state invariant must hold before and after every valid operation. This applies to any subsequent implementation as well. To ensure this, a function that returns true if a state satisfies the state invariant and false otherwise, is created.

5.3.3.3 Comparison

- In Z the state invariant is implemented as a function that returns true if a state is satisfied and false otherwise.
- In UML, the class invariant is implemented as part of the class declaration and must be obeyed by all descendent classes.

5.3.4 Implementing associations (relationships)

5.3.4.1 UML applications

Rumbaugh et al. ([Rum91]) claim that the implementation of associations in a non-object-oriented language presents the same two possibilities as in an object-oriented language, namely mapping them into pointers or implementing them directly as association container objects. (Refer to Section 4.4.6 for the implementation details and Section 5.6 for examples.)

5.3.4.2 Z applications

Associations are specified as indicated in Section 5.6. Refinement and implementation (i.e. data refinement, operation refinement and operation decomposition) follow the route described in Chapter 3. Refer to Section 5.6 for examples.

5.3.4.3 Comparison

- In Z, relationships are modeled as mathematical relations (or functions) between entity types.
-

- In UML a relationship is modeled as a semantic connection among model elements. The relationship is specialized as a generalization, dependency, association, transition, or link (see [Eri98]).

5.4 Classes and subclasses: Z and UML compared

For this comparison, the following example is used: Suppose a salesperson is an element of the class `BASIC_SALESPERSON` and assume the two subclasses `PERMANENT_SALESPERSON` and `TEMPORARY_SALESPERSON` inherit the attributes and operations of `BASIC_SALESPERSON`.

5.4.1 Specifications in UML

Refer to Figure 5.1. `BASIC_SALESPERSON` is sub typed into subclasses `PERMANENT_SALESPERSON` and `TEMPORARY_SALESPERSON`.

`PERMANENT_SALESPERSON` and `TEMPORARY_SALESPERSON` inherit the *sales_id*, and *sales_name* of the `BASIC_SALESPERSON`. Added to the `PERMANENT_SALESPERSON` is the *annual_salary* and added to the `TEMPORARY_SALESPERSON` is the *unit_price* and the *units_sold*.

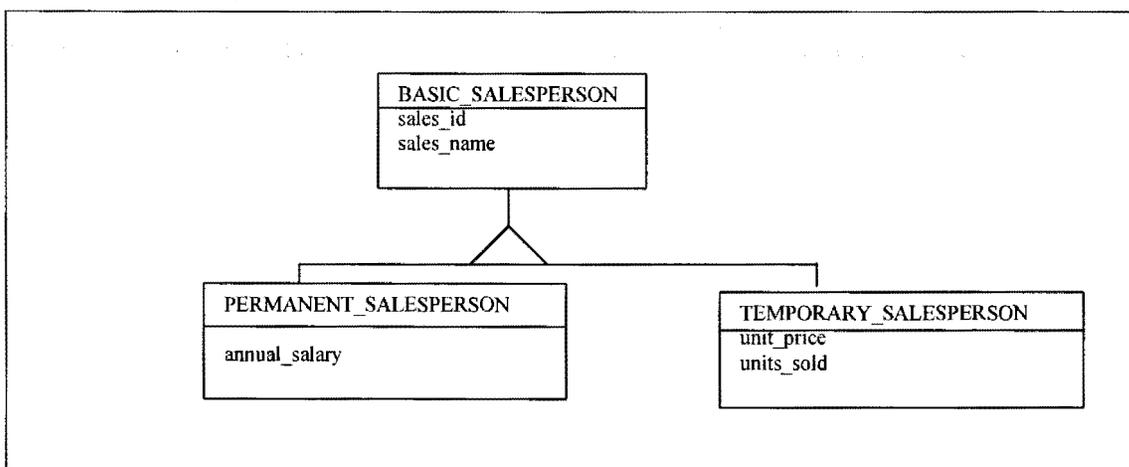


Figure 5.1 Subtyping in UML

5.4.2 Specifications in Z

In Z, specifying the same example would result in the definition of the basic types [SALESPERSON_ID, RAND¹, NAME] with the schema definitions given below.

The predicates define a set partition. The elements of the subtypes comprise all the elements of the super type. No super type instance can be present in more than one sub type.

```
BASIC_SALESPERSON
-----
sales_id : SALESPERSON_ID
sales_name : NAME
```

A salesperson is either a permanent salesperson or a temporary salesperson:

```
PERMANENT_SALESPERSON
-----
BASIC_SALESPERSON
annual_salary : RAND
```

```
TEMPORARY_SALESPERSON
-----
BASIC_SALESPERSON
unit_price : RAND
units_sold : ℕ
```

The subtype relationships are defined in one schema *SALESPERSON_SUBTYPES_1* (see Figure 5.2). This brings together the specification of the sets of instances of all the classes, the function defining the common unique identifier and the one-to-one relationships.

¹ Using the South African currency

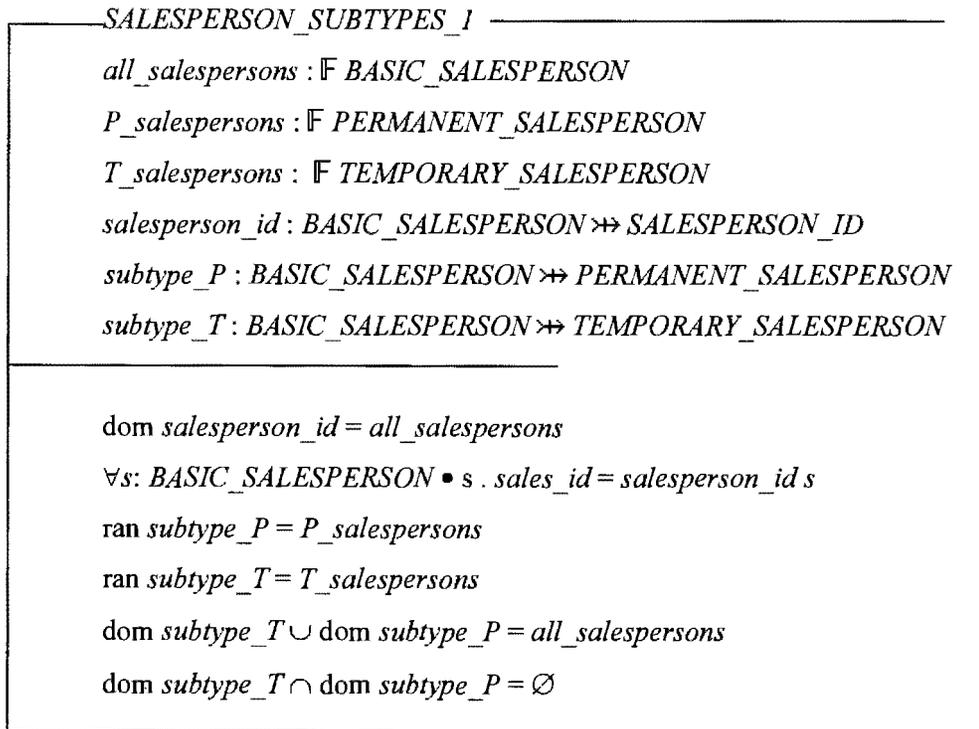


Figure 5.2 Subtyping in Z

5.4.3 Verification and Refinement

5.4.3.1 Verification and Refinement (Z)

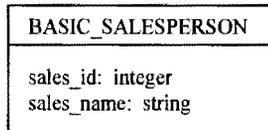
A number of consistency checks can be performed on the above state space. For example, a specifier can show that there exists a basic salesperson who is also a temporary salesperson for certain values of the variables.

$$\vdash \exists BASIC_SALESPERSON' \bullet sales_id' \in SALESPERSON_ID \wedge sales_name' \in NAME \wedge sales_id = 45321 \wedge sales_name = 'Peter\ Scott' \wedge unit_price = 25.00 \wedge unit_sold = 15$$

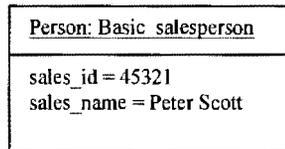
An analogous instantiation can be written down for a permanent salesperson. The above mentioned check can also serve as a data refinement where it must be determined whether every abstract state has at least one concrete representative.

5.4.3.2 Verification and Refinement (UML)

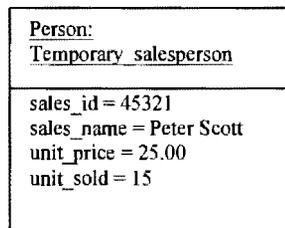
In UML, for example, the BASIC_SALESPERSON class can be represented by



and can be instantiated as follows [Eri98, Fow97]:



Similarly, a temporary salesperson can be instantiated by:



Therefore, the abstract state has at least one concrete representative.

5.4.4 Implementation into C

The following is a trivial example. The program is for illustrative purposes and performs more functions and calculations than what are reflected in the specifications. No files are used. A salesperson is either a permanent or a temporary salesperson that inherits the salesperson ID and salesperson name from the basic salesperson (refer to Line 1 and Line 2 in the program). For this example 3 permanent and 2 temporary salespersons are entered which together constitute the 5 basic salespersons.

```
#include <stdio.h>           //This program also reads in and prints the data, which is
#include <string.h>         //implied by the specifications, but not explicitly indicated.
#include <conio.h>

/*BASIC_SALESPERSON*/
struct basic_salesperson
{
```

```
int sales_id[5];
char sales_name[5][10];
char sales_ind[5][1]; //The indicator is either T for a temporary, or P for a
}; //permanent salesperson.

/*PERMANENT_SALESPERSON*/
struct permanent_salesperson
{
    struct basic_salesperson; // Line 1: Inherits from the basic salesperson
    int annual_sal[3];
};

/*TEMPORARY_SALESPERSON*/
struct temporary_salesperson
{
    struct basic_salesperson; // Line 2: Inherits from the basic salesperson
    float unit_price[2];
    int unit_sold[2];
};

void main(void)
{
    int i, temp_id, temp_sal, temp_unit;
    float temp_price;
    char temp_name[10], temp_ind[1];
    struct basic_salesperson BASIC;
    struct permanent_salesperson PERMANENT;
    struct temporary_salesperson TEMPORARY;
    printf("Enter the information for the 2 temporary salespersons\n");

    for (i=0; i<2; i=i+1) //Data is entered
    {
        printf("\nEnter the ID number\n");
        scanf("%d", &temp_id);
        BASIC.sales_id[i] = temp_id;
```

```
printf("Enter the name\n");
scanf("%s",temp_name);
strcpy(BASIC.sales_name[i], temp_name);
printf("Enter T for the indicator\n");
scanf("%s", temp_ind);
strcpy(BASIC.sales_ind[i],temp_ind);
printf("Enter the unit price\n");
scanf("%f", &temp_price);
TEMPORARY.unit_price[i] = temp_price;
printf("Enter the units sold\n");
scanf("%d", &temp_unit);
TEMPORARY.unit_sold[i] = temp_unit;
}
printf("\n\nPrint the information of the 2 temporary salespersons\n");
for (i=0; i<2; i=i+1) //From this routine it can be seen that the basic salesperson
{ //data has been inherited by the temporary salesperson.
printf("Id %d Name %s Price %f Units %d\n", BASIC.sales_id[i],
BASIC.sales_name[i], TEMPORARY.unit_price[i], TEMPORARY.unit_sold[i]);
}
printf("\n\nEnter the information for the 3 permanent salespersons\n");
for (i=0; i<3; i=i+1)
{
printf("\nEnter the ID number\n");
scanf("%d", &temp_id);
BASIC.sales_id[i] = temp_id;
printf("Enter the name\n");
scanf("%s",temp_name);
strcpy(BASIC.sales_name[i], temp_name);
printf("Enter P for the indicator\n");
scanf("%s", temp_ind);
strcpy(BASIC.sales_ind[i],temp_ind);
printf("Enter the annual salary\n");
scanf("%d", &temp_sal);
PERMANENT.annual_sal[i] = temp_sal;
```

```

    }
    printf("\n\nPrint the information of the 3 permanent salespersons\n");
for (i=0; i<3; i=i+1)          //From this routine it can be seen that the basic salesperson
    {                          //data has been inherited by the permanent salesperson.
    printf("Id %d Name %s Salary %d\n", BASIC.sales_id[i],
    BASIC.sales_name[i], PERMANENT.annual_sal[i]);
    }

getche();
};

```

5.4.5 Implementation into Cobol

This example reflects the main aspects of the C program of 5.4.4. Because structured Cobol do not have built in inherits features, inheritance has to be simulated by using the MOVE statement.

```

.....
01 basic_salesperson1      occurs 5 times.
   05 sales_id             pic 9(5).
   05 sales_name          pic x(10).
   05 sales_ind           pic x.

01 Permanent_salesperson  occurs 3 times.
   05 basic_salesperson2  pic x(16).
   05 annual_sal          pic 9(6).

01 Temporary_salesperson  occurs 2 times.
   05 basic_salesperson3  pic x(16).
   05 unit_price          pic 9(4)v99.
   05 unit_sold           pic 9(3).
.....

```

Choose either a temporary or permanent salesperson:

if Permanent_salesperson

move basic_salesperson1() to basic_salesperson2()

else if Temporary_salesperson

```
    move basic_salesperson( ) to basic_salesperson3( )
    else perform error-rtn.
.....
```

From the above it can be deduced that a person is either a permanent or temporary salesperson. Because there are only permanent or temporary salespersons, the number of salespersons is the sum of the permanent plus temporary salespersons. This information can be obtained either from files or entered from the keyboard.

5.4.6 Comparison

5.4.6.1 Specification

- For both the Z and UML a supertype entity comprises attributes common to all its subtypes and has relationships with its subtypes. Each subtype entity has additional information.
- For this specific example it appears that the information conveyed by the specifications in UML and Z are equally comprehensive and clear for the purposes of an implementation. For example, in UML the inheritance of properties of the two subclasses PERMANENT_SALESPERSON and TEMPORARY_SALESPERSON from the supertype BASIC_SALESPERSON is indicated diagrammatically while in Z it is indicated by schema inclusion.

5.4.6.2 Verification and Refinement

- For the verification and refinement both the UML and Z have concrete representations for the abstract states. Although a matter of taste, the UML diagram appears to be easier to grasp at a glance, while the Z definitions are harder to interpret at a first glance. However, the Z definitions are more precise which allows for greater precision (e.g. the specifier can reason about them in a formal way). Refer to Section 5.4.3.

5.4.6.3 Implementation

- The class diagrams in UML and the schemas in Z for BASIC_SALESPERSON, PERMANENT_SALESPERSON and TEMPORARY_SALESPERSON were implemented into the C structures BASIC_SALESPERSON, PERMANENT_SALESPERSON and TEMPORARY_SALESPERSON above. The 'struct basic_salesperson' statement in the C implementation of
-

PERMANENT_SALESPERSON and TEMPORARY_SALESPERSON imports the *sales_id* and *sales_name* of the BASIC_SALESPERSON structure. Therefore, implementation from the UML classes and Z schemas into the C structures are the same as far as the data type setting and inheritance are concerned.

- The class diagrams in UML and schemas in Z are implemented into Cobol in the FD (File Description) entries (refer to the Cobol program). For the Cobol program, the inheritance is accomplished by the *move basic_salesperson1() to basic_salesperson2() or basic_salesperson3()* statements, which also result into the same implementation from the UML classes and Z schemas into the Cobol statements as far as the data type setting and inheritance are concerned.

5.5 Operations: Z and UML compared

The operations are going to be illustrated by means of the following example: A constituent can have many voters and a voter belongs to one constituent. Two operations can be performed: the ID (identification number) of a voter can be displayed, and then that voter is removed from his constituency. The *REGISTERED_REL* is a separate schema to correspond with the *is_registered_with* relation in the UML specification.

5.5.1 Specifications in UML

For the details of the Display voters() and Remove voters() operations refer to figures 5.4.1, 5.4.2, and 5.4.3 (pages 5-27 and 5-28).

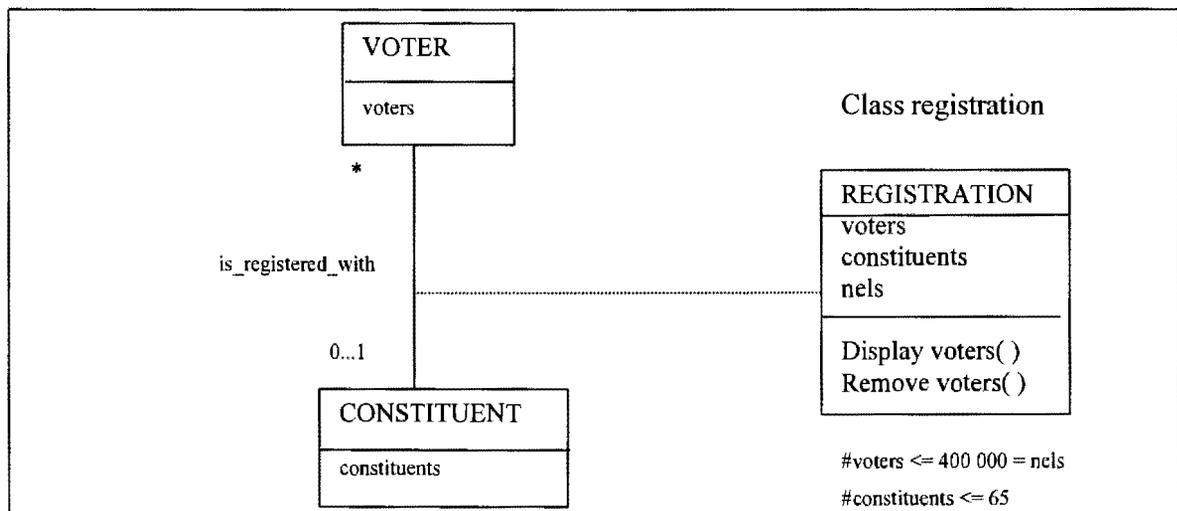
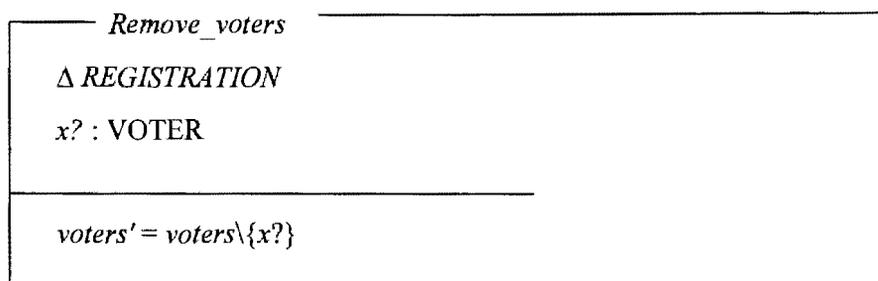
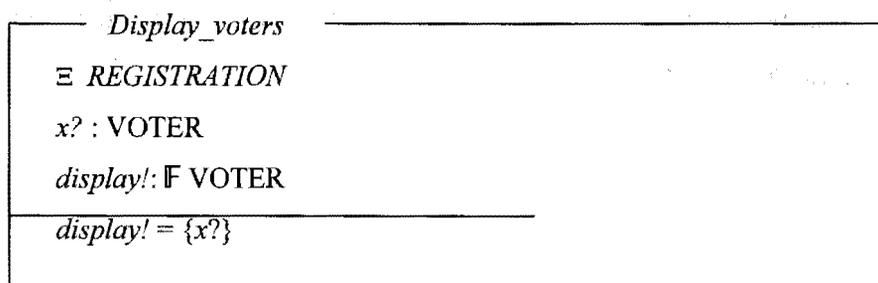
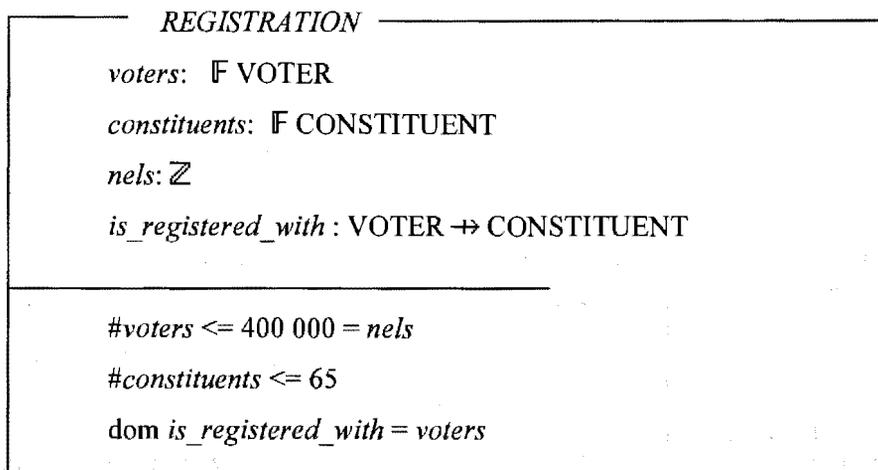
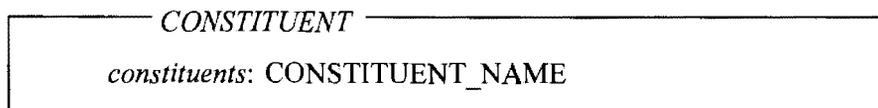
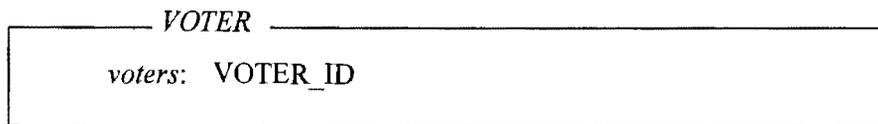


Figure 5.3 Specifications in UML

5.5.2 Specifications in Z

The basic types are: [VOTER_ID, CONSTITUENT_NAME]



5.5.3 Verification and Refinement

As an example of verification, we look at the verification of global definitions (Refer to Chapter 3 Section 3.3.1.3). Assume that for constituent PAARL, the constituent number is 52236, the number of voters is just 5 (which is less than 20000 (required for PAARL)) and the ID numbers of the voters are contained in the set $VOTER = \{111222, 111228, 125678, 135000, 140000\}$.

Then, for the *REGISTRATION* schema:

$voters: \quad \mathbb{F} \text{ VOTER}$ $constituents: \mathbb{F} \text{ CONSTITUENT}$
$VOTER = \{111222, 111228, 125678, 135000, 140000\}$ $constituents = \{PAARL\}$ $\#voter \leq 20\ 000 = 5 = nels$

or

$$\begin{aligned} & \vdash \exists voters: \mathbb{F} \text{ VOTER} \wedge \exists constituents: \mathbb{F} \text{ CONSTITUENT} \bullet \forall voters: \text{VOTER} \wedge \forall \\ & constituents: \text{CONSTITUENT} \bullet voters \in VOTER \wedge \#voters \leq 20\ 000 \wedge constituents = \\ & \{PAARL\} \in CONSTITUENT \end{aligned}$$

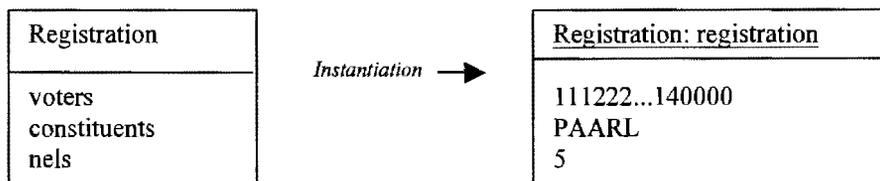
which for the axiomatic description is true, therefore

$$\exists voters: \mathbb{F} \text{ VOTER} \wedge \exists constituents: \mathbb{F} \text{ CONSTITUENT} \bullet true$$

and by a simple property of logic

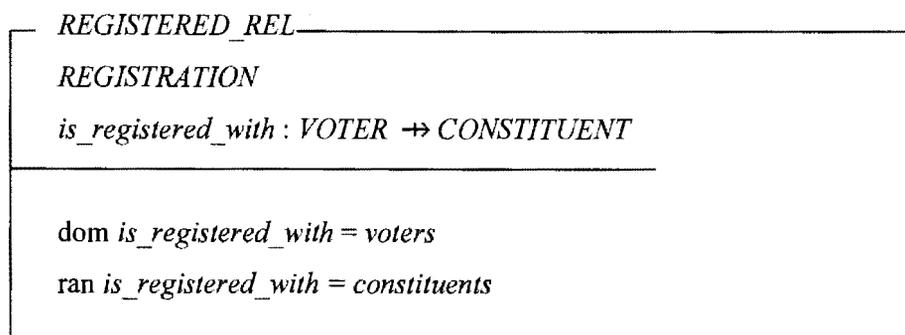
$$true$$

- In UML the class Registration can be instantiated for constituent PAARL as follows:

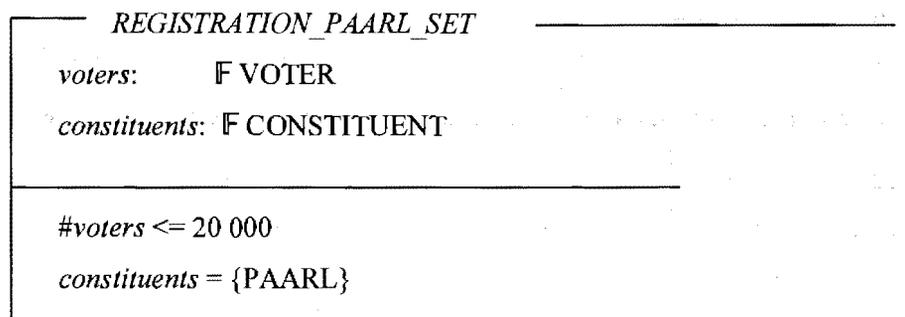


This instantiation of the class can also serve as a verification of global definitions.

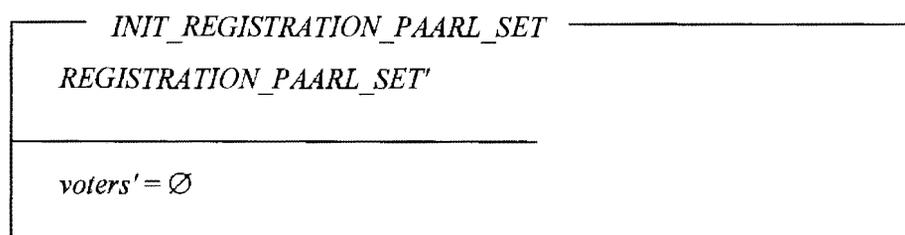
- A concrete representation of the abstract *REGISTERED_REL* (data refinement) in Z is:



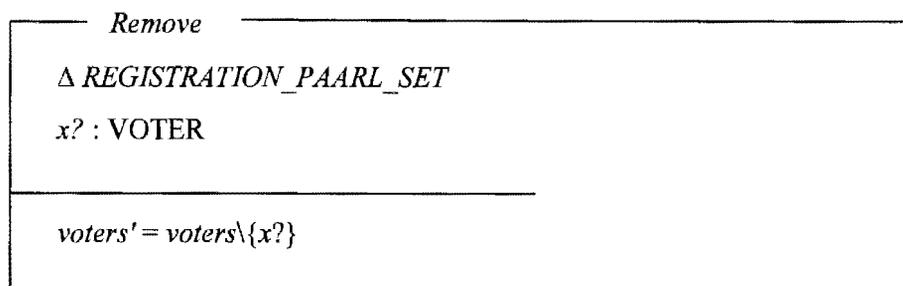
The verification of the global definitions can also serve as part of the data refinement process, because part of the data refinement process is to determine whether the design correctly simulates what is abstractly described through three verification tasks (see [Rat94], Chapter 3 Section 3.4.2). One of these tasks is the concrete state adequacy verification which determines whether the concrete state adequately represents the abstract state. A check is to determine whether every abstract state has at least one concrete representative, which has been done in the verification of the global definitions for the *REGISTRATION* schema.



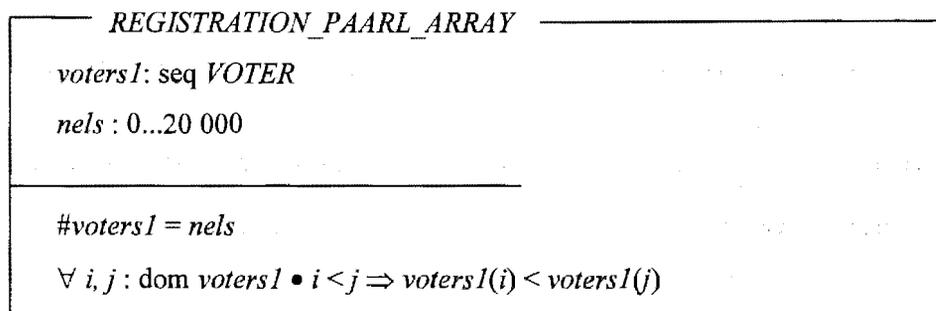
with an initial state:



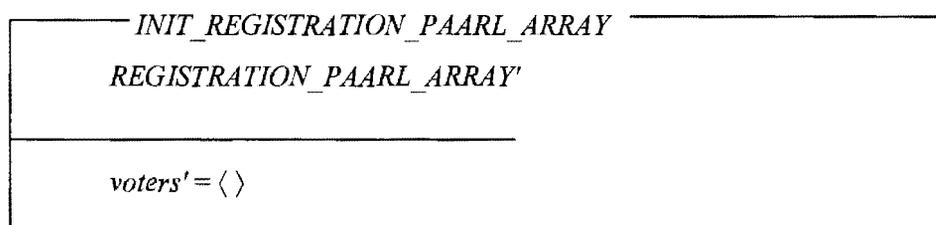
and with (for example) one of its operations being



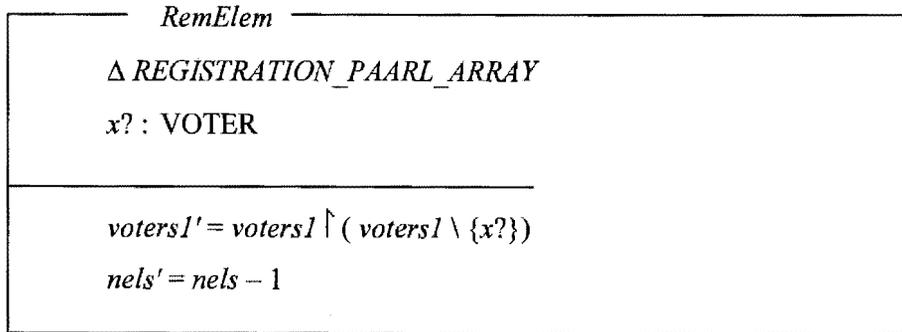
The state set *REGISTRATION_PAARL_SET* can be implemented by an array with an index variable *votersI* array[1..20 000] of *VOTER*, and *nels*: 0..20 000. (*nels* will be an array if there are more than one constituent; *nels* in this example is only for PAARL). In the case of zero constituents, there will be zero voters and the system will be terminated. It is assumed that the *nels* elements are sorted in ascending sequence to ensure that no duplicates are in the array, and to facilitate fast lookup of the array. This choice of data structuring can be described in Z as:



The concrete initial state is:



The abstract Remove operation is re-expressed as the concrete operation:



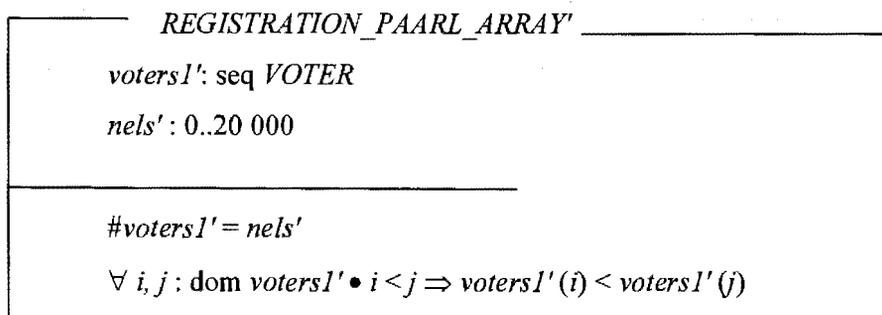
The symbol \uparrow denotes a sequence filtering: For a sequence s and a set of values v , $s \uparrow v$ creates a new sequence that contains precisely those entries in sequence s that are elements of set v , and in the same order. For example: $\langle a, b, c, d, e \rangle \uparrow \{b, d, f\} = \langle b, d \rangle$ [Bar94]. The function $\text{ran } a$ denotes the range of a relation.

It can then be verified that the concrete state is consistent (refer to Section 3.4.2). It has to shown in general that

$\vdash \exists \text{ConcState}' \bullet \text{InitConcState}$ (InitConcState represents the initial concrete state)

so for our example:

$\vdash \exists \text{REGISTRATION_PAARL_ARRAY}' \bullet \text{INIT_REGISTRATION_PAARL_ARRAY}$



REGISTRATION_PAARL_ARRAY' is an after state of the general model *REGISTRATION_PAARL_ARRAY*.

To show that it is consistent:

Refer to *REGISTRATION_PAARL_ARRAY'*

$\vdash \exists \text{voters1}' : \text{seq VOTER} \wedge \exists \text{nels}' = 0..20\ 000 \bullet \#\text{voters1}' = \text{nels}' \wedge$

$\forall i, j : \text{dom voters1}' \bullet i < j \Rightarrow$

$\text{voters1}'(0) < \text{voters1}'(1) < \text{voters1}'(2), < \dots \text{voters1}'(\text{nels}')$

If $nels' = 0$ then $votersI' = \langle \rangle$

which implies that there is a state *REGISTRATION_PAARL_ARRAY* of the general model *REGISTRATION_PAARL_ARRAY* that satisfies the initial state description *INIT_REGISTRATION_PAARL_ARRAY*.

Furthermore, it must be determined whether every abstract state has at least one concrete representative (refer to Section 3.4.2). This can be achieved by determining if each abstract variable can be derived or 'retrieved' from the concrete variables by writing down equalities of the form:

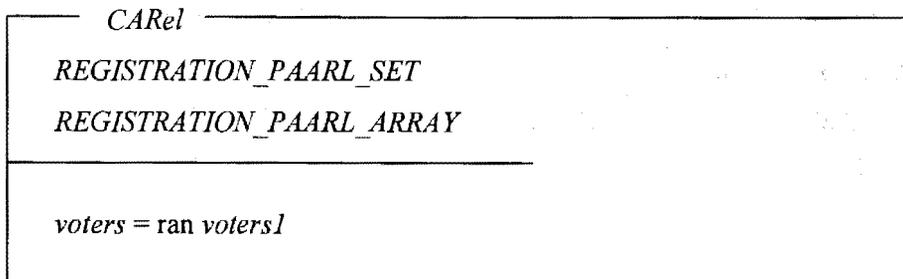
$$AbsVar = Expr(ConcVars)$$

where *AbsVar* represents an abstract variable of the abstract state, *Expr* the expression and *ConcVars* the concrete variable of the concrete state representing the abstract state.

For the example the predicate (see schema *CARel* below)

$$voters = \text{ran } votersI$$

will be referred to as the 'retrieve relation' *CARel* (concrete-to-abstract relation) that brings together the abstract and the concrete states:



The above equality implies that *CARel* is in effect a total function when viewed as 'calculating' the abstract state from the concrete one [Rat94]. Being total means that every concrete state maps to some abstract state [Rat94, Der01].

Suppose, however, the 'sorted' invariant was removed from *REGISTRATION_PAARL_ARRAY* so that the array element order was immaterial and that no duplicates are stored in the array. The design will now include some redundancy in that each non-empty, non-singleton set in the abstract state would have more than one concrete representation (indicated by \Rightarrow) [Rat94, Der01].

For example, the abstract state

$$\langle voters \Rightarrow \{111222, 111228, 125678, 135000, 140000\} \rangle$$

would have 120 concrete representatives (of which two is shown):

$$\langle voters1 \Rightarrow \langle 111222, 111228, 125678, 135000, 140000 \rangle, nels \Rightarrow 5 \rangle$$

.....

$$\langle voters1 \Rightarrow \langle 111228, 111222, 125678, 135000, 140000 \rangle, nels \Rightarrow 5 \rangle$$

Since, in general, assuming no duplicates, there would be $nels!$ concrete representatives for a single abstract state. The implicit functionality of a retrieve relation such as $CARel$ is not compromised because the relation expresses a calculation from *concrete to abstract representations* [Rat94].

- **The correctness of the concrete initial state must be verified**

(Refer to Section 3.4.2.1.3). The concrete initial state must not describe initial states that have no counterpart in the abstract model [Rat94, Pot96, Der01].

To illustrate, a theorem of the following form is proved:

In general given the retrieve relation then:

$$InitConcState \vdash InitAbsState$$

which says that ‘for each concrete initial state, there is a corresponding abstract one’.

In our example, this means proving the sequent

Given $CARel'$ then

$$INIT_REGISTRATION_PAARL_ARRAY \vdash INIT_REGISTRATION_PAARL_SET$$

$CARel'$ <hr style="border: 0; border-top: 1px solid black; margin: 5px 0;"/> $REGISTRATION_PAARL_SET'$ $REGISTRATION_PAARL_ARRAY'$ <hr style="border: 0; border-top: 1px solid black; margin: 5px 0;"/> $voters' = \text{ran } voters1'$

The declarative part of the right-hand side schema text is just $NatNumSet'$ (refer to Section 3.4.2), which is provided by $CARel'$ on the left.

The sequent is then unfolded into

$$CARel' ; REGISTRATION_PAARL_ARRAY \mid \\ votersI' = \langle \rangle \vdash voters' = \emptyset$$

which holds because $votersI' = \langle \rangle$ on the left and $voters' = \text{ran } votersI'$ in $CARel'$.

By substitution $voters' = \text{ran } \langle \rangle$, therefore $voters' = \emptyset$ immediately follows.

- **Concrete operation applicability**

(Refer to Section 3.4.2.2) It must be determined whether for each concrete-abstract operation pair, the former is applicable over its state whenever the latter is applicable over its state [Rat94, Pot96, Der01]. This check ensures that each concrete operation has an applicability that at least encompasses its abstract partner. To show this requires proving theorems of the form given the retrieve relation.

$$\text{pre } AbsOp \vdash \text{pre } ConcOp$$

In the example discussed above, we need to show that

Given $CARel$ then:

$$\text{pre } Remove \vdash \text{pre } RemElem$$

The right-hand side of the above theorem is schema text, therefore its declaration part (the schema reference $REGISTRATION_PAARL_ARRAY$ and variable $x? : VOTER$), as well as its predicate, must be derivable from the left-hand side [Rat94]. $CARel$ acts as an extra hypothesis, and so $REGISTRATION_PAARL_ARRAY$ in $\text{pre } RemElem$ is provided by $CARel$. Variable $x?$ in $\text{pre } RemElem$ is the same as the $x?$ of $\text{pre } Remove$. Therefore it is valid to rearrange the sequent by unfolding its left-hand side and writing just the conclusion's *predicate* on the right-hand side of the turnstile:

$$CARel; REGISTRATION_PAARL_SET; x? : VOTER \mid \\ \text{true} \vdash \text{true} = \\ voters' = voters \setminus \{x?\} \vdash votersI' = votersI \uparrow (\text{ran } votersI \setminus \{x?\})$$

which holds because we have

$$voters = \text{ran } votersI \quad \text{in } CARel$$

By substitution

$$voters' = (\text{ran } votersI \setminus \{x?\}) \\ votersI \uparrow voters' = votersI \uparrow (\text{ran } votersI \setminus \{x?\})$$

$$\text{voters}I \uparrow \text{ran voters}I' = \text{voters}I \uparrow (\text{ran voters}I \setminus \{x?\})$$

$$\text{ran} (\text{voters}I \uparrow \text{ran voters}I') = \text{ran} (\text{voters}I \uparrow (\text{ran voters}I \setminus \{x?\}))$$

(In general if $\{X\} \subseteq \text{ran voters} \Rightarrow \text{ran}(\text{voters} \uparrow X) = \{X\}$)

Therefore,

$$\text{ran voters}I' = \text{ran} (\text{voters}I \uparrow (\text{ran voters}I \setminus \{x?\}))$$

$$\text{voters}I' = \text{voters}I \uparrow (\text{ran voters}I \setminus \{x?\})$$

This sequent is trivially a theorem and so *RemElem* passes its applicability check.

- **Concrete operation correctness**

(Refer to Section 3.4.2). It has to be determined for each concrete-abstract operation pair whether the concrete operation, when applied in circumstances conforming to its abstract partner's applicability, produces behaviour which its abstract partner would be capable of producing [Rat94, Pot96, Der01].

Behaviour correctness is ensured by proving:

$$\text{pre } AbsOp \wedge ConcOp \vdash AbsOp$$

given the retrieve relation where *AbsOp* represents the abstract operation applicable to the state and *ConcOp* is the concrete operation applicable to the state.

In the example it must be shown that

Given $\Delta C A R e l$ then:

$$\text{pre } Remove \wedge RemElem \vdash Remove$$

which unfolds into

$$\Delta C A R e l; REGISTRATION_PAARL_SET; \Delta REGISTRATION_PAARL_ARRAY;$$

$x? : VOTER \mid$

$$\text{true} \wedge \text{voters}I' = \text{voters}I \uparrow (\text{ran voters}I \setminus \{x?\})$$

\vdash

$$\text{voters}' = \text{voters} \setminus \{x?\}$$

From the hypothesis equality, it follows that

$$\text{ran voters}I' = \text{ran} (\text{voters}I \uparrow (\text{ran voters}I \setminus \{x?\}))$$

(In general if $\{X\} \subseteq \text{ran voters} \Rightarrow \text{ran}(\text{voters} \uparrow X) = \{X\}$)

By properties of sets

$$\text{ran voters}I \setminus \{x?\} \subseteq \text{ran voters}I$$

Therefore

$$\text{ran voters}I' = \text{ran voters}I \setminus \{x?\}$$

The conclusion of the sequent follows because of the equalities $voters = \text{ran voters1}$ and

$$voters' = \text{ran voters1}' \text{ in } \Delta \text{CAREl}$$

So the correctness theorem holds for *RemElem*.

Behaviour of a concrete operation does not produce more than its abstract partner in equivalent circumstances; however, it can be more deterministic. The concrete operation's behaviour must be behaviour the abstract operation could produce.

For the operation refinement, the operation schema is refined into an algorithm. The algorithm can be refined into code (refer to Section 3.4.3).

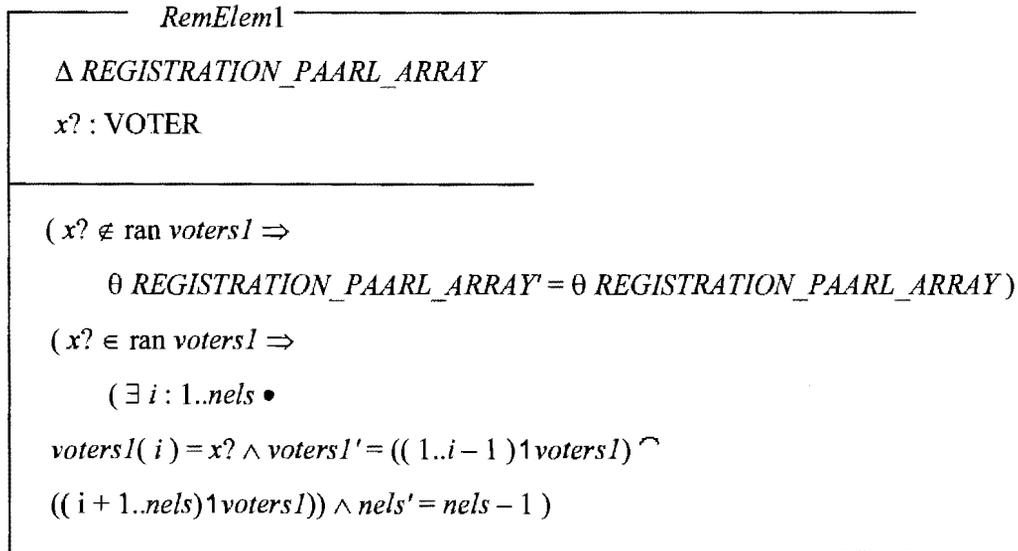
Refer to *RemElem* :

<i>RemElem</i>
$\Delta \text{REGISTRATION_PAARL_ARRAY}$
$x? : \text{VOTER}$
$voters1' = voters1 \uparrow (voters1 \setminus \{x?\})$
$nels' = nels - 1$

The first two lines of the predicate state that if $x?$ is not in the array, nothing changes. The last four lines state that if $x?$ is in the array, it gets removed from its position i , leaving an updated but still contiguous sequence. $(1..i - 1) \uparrow voters1$ extracts all the elements of array $voters1$ from 1 to $i - 1$: $voters1(1), \dots, voters1(i - 1)$. $(i + 1 \dots nels) \uparrow voters1$ extracts all the elements of array $voters1$ from $i + 1$ to $nels$: $voters1(i + 1), \dots, voters1(nels)$. The two extracted arrays are then concatenated. The only array element that was not extracted was $voters1(i)$, which has effectively now been removed. The new number of elements in the array $nels'$ is now 1 less than the original number $nels$.

The abstract operation, *Remove*, is re-expressed as the concrete operation *RemElem*.

RemElem is refined into *RemElem1*:



RemElem1 can be refined into the following algorithm:

<i>lookFor</i> (<i>x?</i> , <i>voters1</i> , <i>i</i> , <i>found</i>)	[Check if <i>x?</i> is in the array <i>voters1</i>]
if \neg <i>found</i> then skip	[If not found in the array go to endif]
elseif <i>found</i> then	[If found (in the array) then
<i>shiftDownOne</i> (<i>voters1</i> , <i>i</i>)	elements in positions <i>i</i> + 1...# <i>voters1</i> are moved
	successively down one, thus getting rid of <i>x?</i>]
<i>nels</i> := <i>nels</i> - 1	[Derived from the predicate <i>nels'</i> = <i>nels</i> - 1]
endif	

The bracketed italicised parts indicate that the constructs involved are specification components which need to be subjected to further refinement.

For UML we found no published refinement procedure in the literature. (UML has some notion of refinement between a use case and its implementation, which is a different notion of the refinement discussed in this work.) However, Figure 5.4.1 to Figure 5.4.3 below gives a broad description of what should happen in the implementation.

5.5.4 Implementation into C

This program lets a user select a voter in a constituency by index number and the program responds by printing the ID number of that particular voter. This is a trivial example. In practice the constituency would be initialised from a file of values. (continued on page 28)

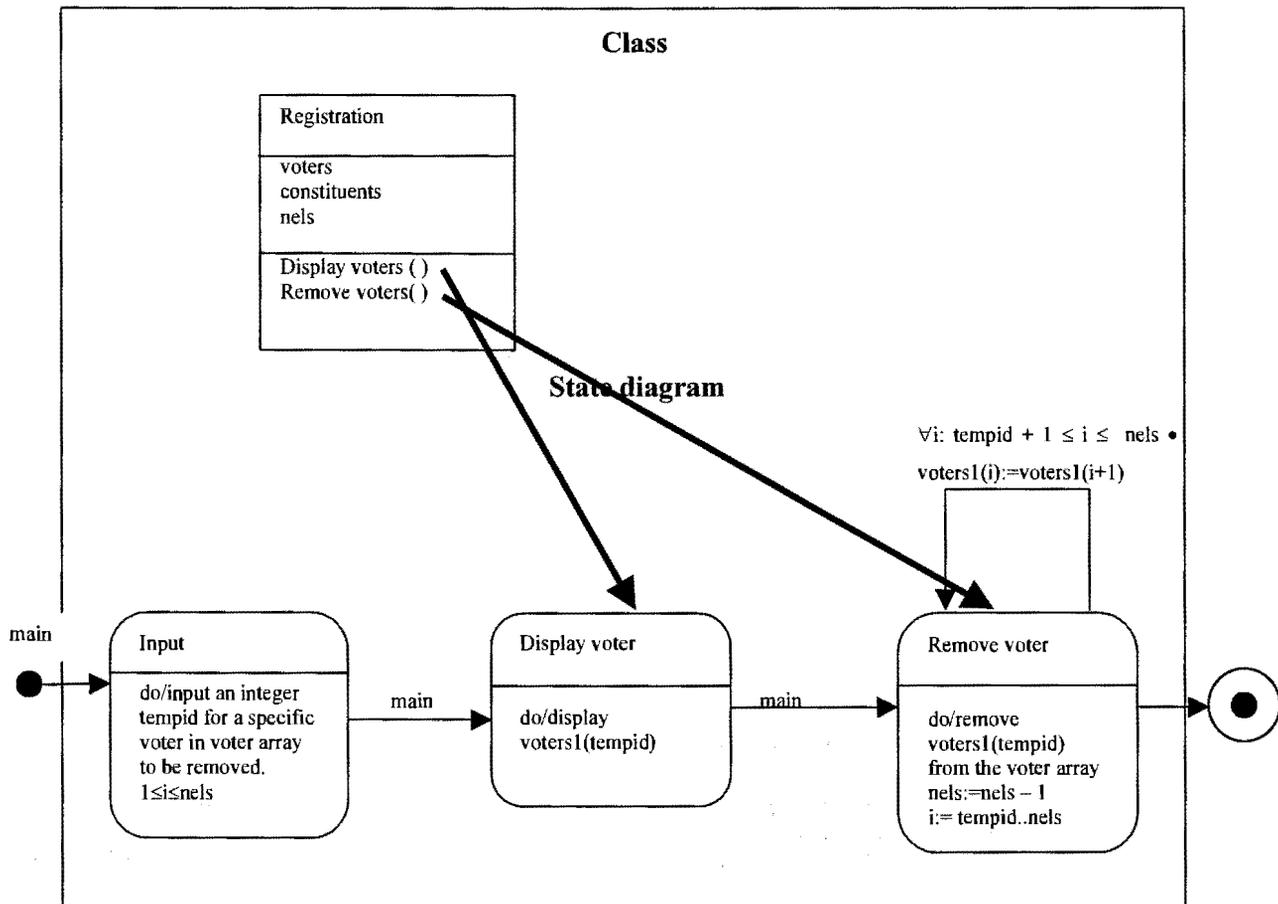


Figure 5.4.1 Registration class with its corresponding state diagram

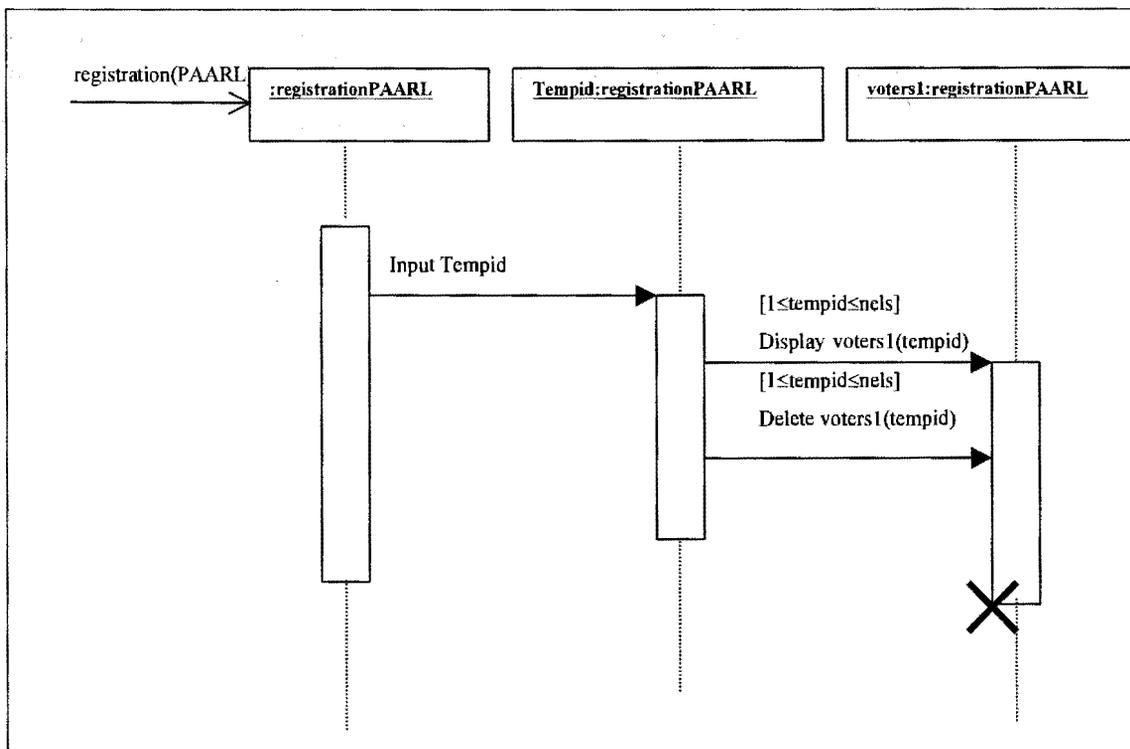


Figure 5.4.2 Sequence diagram

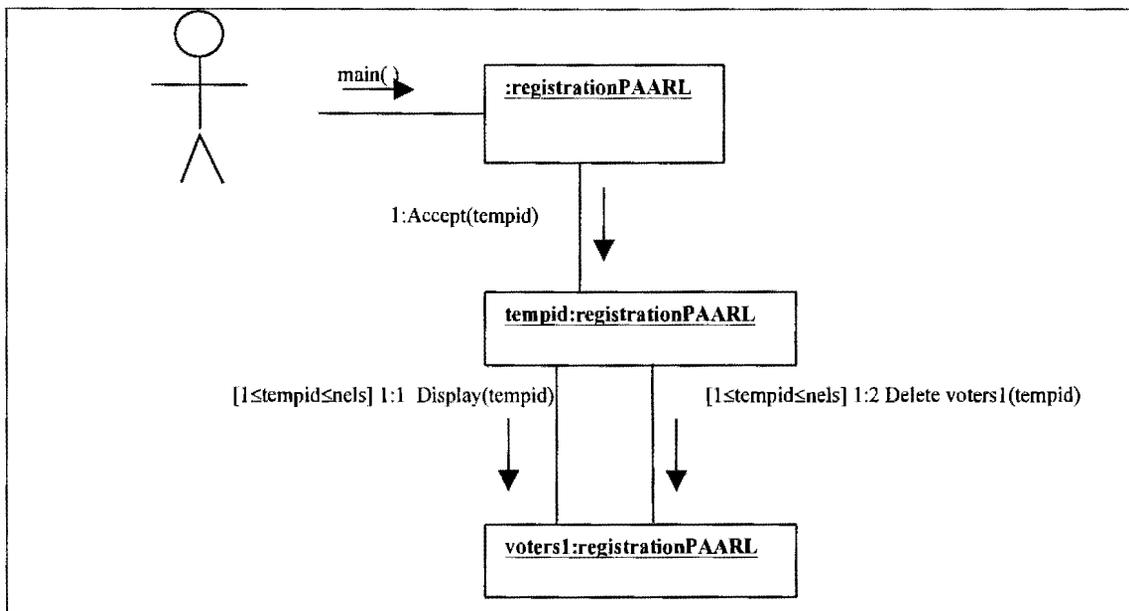


Figure 5.4.3 Collaboration diagram

Functions would need to be written to update the file. Tempid is a temporary variable to hold an index number typed by the user. Five voter IDs – refer to the verification and refinement of Section 5.5.3 - are read in for a constituent whose name PAARL has also been read in. Refer to the output of the program. The association between voter and constituent is accomplished by the struct registration that includes both the voter and constituent information, and the voter and constituent arrays.

```

#include <stdio.h>
#include <string.h>
#include <conio.h>
/* Voters */
struct voter
{
    int voters[5][20]; //Line 1: Assume a 2D array of maximum of 20 voter ID numbers
                      //per constituent and 5 constituents.
};

/* Constituents*/
struct constituent
{

```

```
char constituents[5][10]; //Line 2: Assume each constituent name is maximum 10
                          // characters long for 5 constituents.
};

/* Registration*/

struct registration      //Line 3: This struct handles the association between struct voter
                          // and struct constituent.
{
    struct voter;
    struct constituent;
    int nels;
};

void main(void)
{
    /* i is the number of the constituent for this program and nels is the number of voters for */
    /* that constituent. */
    /* Read in or assign initial integer value for nels and i: Assume that nels is 5 and i is 1.*/
    /* Read in an integer value and display the ID of the voter from the value in the voter array */
    /* then remove that voter from the voter array. */
    int tempid, j, j1;
    char tempname[10];
    struct registration REGISTRATION;
    struct voter VOTER;
    struct constituent CONSTITUENT;
    int nels = 5;
    int i = 1;
    /*Load the first constituent array with 5 voter IDs */
    for (j = 0; j < nels; j++)
    {
        j1 = j + 1;
        printf(" Load the ID number for voter number %d of constituent number %d\n ", j1, i);
        scanf("%d", &tempid);
        VOTER.voters[i][j] = tempid;
```

```

    }
    /*Read in the name of the constituent */
    printf(" Load the name for constituent number %d\n",i);
    scanf("%s", tempname);
    strcpy(CONSTITUENT.constituents[i], tempname);
    /*Print the ID numbers of the voters and the name of constituent number i */
    printf("The name of constituent %d is %s\n", i, tempname);
    printf("The ID numbers of the %d voters for constituent %d is\n",nels,i);
    for (j = 0; j < nels; j++)
    {
        printf("%d\n",VOTER.voters[i][j]);
    }

    /*Type in one of the ID numbers to be removed from the voters array*/
    printf("Type in one of the five voter ID numbers\n");
    scanf("%d", &tempid);
    for (j = 0; j < nels && VOTER.voters[i][j] != tempid; j++)
    {
        j1 = j + 1;
        printf("This voter for %s is voter number %d\n", CONSTITUENT.constituents[i], j1);
    }
    /* Now remove that voter from the voter array */
    nels = nels - 1;
    for (j = j1 - 1; j < nels; j++)
    {
        VOTER.voters[i][j] = VOTER.voters[i][j + 1];
    }
    /* Now print the updated array */
    printf("The updated voter array follows:\n");
    for (j = 0; j < nels; j++)
    {
        printf("%d\n",VOTER.voters[i][j]);
    }
};

```

Output:

Load the ID number for voter number 1 of constituent number 1

111222

Load the ID number for voter number 2 of constituent number 1

111228

Load the ID number for voter number 3 of constituent number 1

125678

Load the ID number for voter number 4 of constituent number 1

135000

Load the ID number for voter number 5 of constituent number 1

140000

Load the name for constituent number 1

PAARL

The name of constituent 1 is PAARL

The ID numbers of the 5 voters for constituent 1 is

111222

111228

125678

135000

140000

Type in one of the five voter ID numbers

125678

This voter for PAARL is voter number 3

The updated voter array follows:

111222

111228

135000

140000

5.5.5 Implementation into Cobol

The association between voters and constituents is also effectuated through the registration record (if files are used), that incorporates both the voter and constituent information. The association is further accomplished by the use of the single and double level arrays.

.....

```

01 registration_record.
    05 voter_record.
        10 voters2 occurs 5 times.
            15 voters3 occurs 20 times.
                20 voters                                pic 9(6).
    05 constituent_record.
        10 constituent1 occurs 5 times.
            15 constituents                                pic x(10).
    05 nels                                              pic 9(2).
.....
working-storage section.
*tempid is a number between 0 and 20 000
01 tempid                                              pic 9(6).
01 i                                                  pic 9(2) value zero.
01 j                                                  pic 9(2) value zero.
01 j1                                                 pic 9(2) value zero.
.....

```

main-rtn.

```

* The information of registration_record is either accepted from the console or read in from
* a file. After the tables have been loaded, one of the voters are removed from the voter
* array in remove_rtn. Assume that only one constituent namely PAARL is read in for this
* example..
.....

```

```

* Assume that there are only 5 voters in constituent number 1.

```

```

move 5 to nels.
.....

```

```

* Assume voter number j1 is to be removed – then that voter is first displayed
* and then removed in the following two routines.

```

display_rtn.

```

....
display voters(i, j1).

```

remove_rtn.

```

....
compute nels = nels -1.

```

```
perform calc-rtn varying j from j1 by 1 until j = nels.  
.....  
calc-rtn.  
    compute voters(i, j) = voters(i, j + 1)  
.....
```

5.5.6 Comparison

5.5.6.1 Specification

- For the specifications the one-to-many relation *is_registered_with* in UML is indicated by a * at one end and a (0..1) at the other end (see Figure 5.3). This relation is indicated in Z by a partial function: *is_registered_with* : *VOTER* \rightarrow *CONSTITUENT*
- The Registration class in UML with the voter and constituent attributes are represented in Z by the *REGISTRATION* schema. The relation *is_registered_with* in UML is represented by the *REGISTERED_REL* schema in Z.
- The UML operations Display voters() and Remove voters() are represented by the *Display_voters* and *Remove_voters* schemas in Z.

5.5.6.2 Verification and Refinement

A detailed comparison between the verification and refinement of Z and UML is part of the descriptions of Section 5.5.3.

5.5.6.3 Implementation

- For the implementation into C and Cobol, programs have been written that accepts 5 voters for one constituent. An integer is read in or accepted by the program, which gives the relevant voter ID number in the voter array. This particular voter is then removed from the voter array. This implementation corresponds with the specifications in both Z and UML, and also with the refinement in Z.

5.5.6.4 Conclusion

- It can be concluded that through the refinement process of Z, a more direct implementation into the program code can be demonstrated. However, the refinement process is long and tedious which can result in many people not following that route.
 - UML does not have an equivalent refinement process, however, for experienced programmers, the program code can also be adequately inferred from the UML
-

specifications, namely the class-, state-, sequence and collaboration diagrams. Even for a non-object-oriented language such as structured Cobol, the necessary code can be inferred from the UML diagrams. Therefore it depends on the programmer to ensure that the implementation corresponds to the intention.

5.6 Relationships: Z and UML compared

Class diagrams in UML consist of classes and the relationships between them. The relationships that can be used are associations, generalizations, dependencies, and refinements. The most common association is a connection between classes in the form of a solid line [Eri98, Fow97].

Multiplicity, which is a range that tells how many objects are linked, is used to express the relationship between classes. The range can be zero-to-one (0..1), zero-to-many (0..* or just *), one-to-many (1..*), five to eleven (5..11), etc. It is also possible to express sequences of numbers such as (1, 4, 6, 8..12). If no multiplicity is specified, (1..1) is the default. The multiplicity is shown near the ends of the association line, at the class to which it is applicable [Eri98, Fow97].

Relationships are modelled in Z as mathematical relations (or functions) between entity types. An optional relationship is modelled in Z using set inclusion as a restricting predicate: For example, for the declarations

```

voters           F VOTER
constituents :   F CONSTITUENT
is_registered_with : VOTER  $\leftrightarrow$  CONSTITUENT

```

the following relationship holds

$$\text{dom } \textit{is_registered_with} \subseteq \textit{voters}$$

A voter is, therefore, allowed to exist without a constituency.

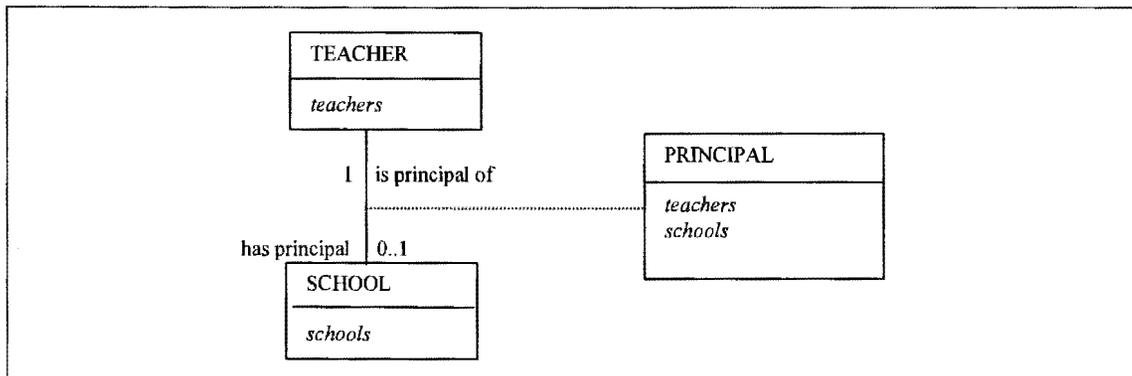


Figure 5.5 One-to-one relationship mandatory on SCHOOL side in UML

5.6.1 One-to-one relationships

Consider the following example. There are a number of schools. Every school has a number of teachers. Every school must have one teacher who is the principal. There are teachers who are not the principal of any school.

** Note that although *has_principal* (Figure 5.6) is actually a *total* injection (every school has a principal), we prefer to model it by a *partial* injection, since the set of all total functions from one set (say X) to another set (say Y) is a subset of the set of all partial functions from X to Y.

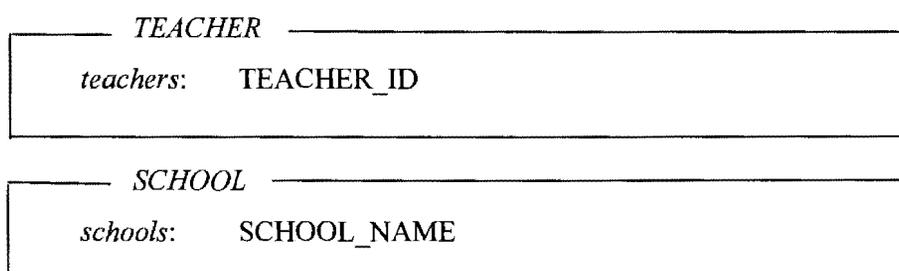
For Z the one-to-one relationship can be modelled in either direction as a *partial* injection (i.e. \rightarrow). An additional predicate specifies that one relationship is the inverse of the other (\sim):

5.6.1.1 Specifications in UML

Refer to Figure 5.5.

5.6.1.2 Specifications in Z

The basic types are: [TEACHER_ID, SCHOOL_NAME]



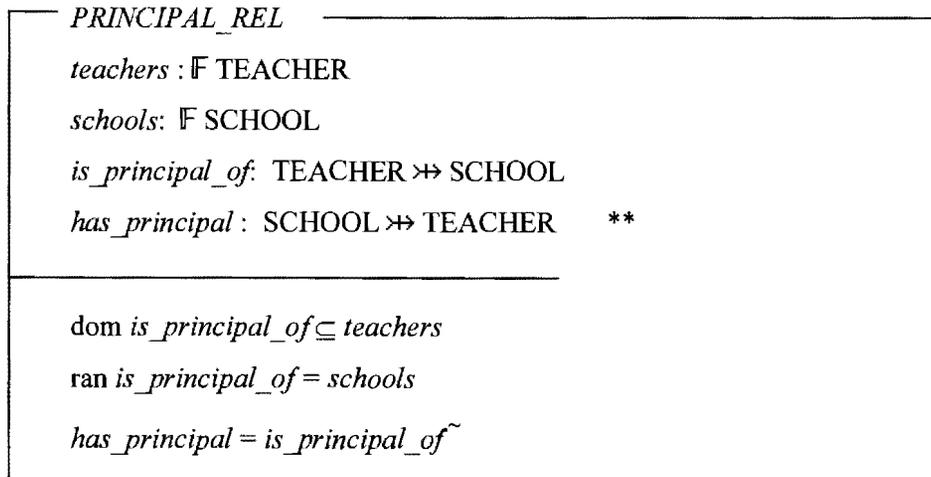


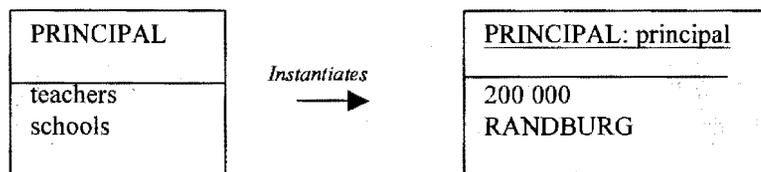
Figure 5.6 One-to-one relationship mandatory on SCHOOL side in Z

5.6.1.3 Verification and Refinement

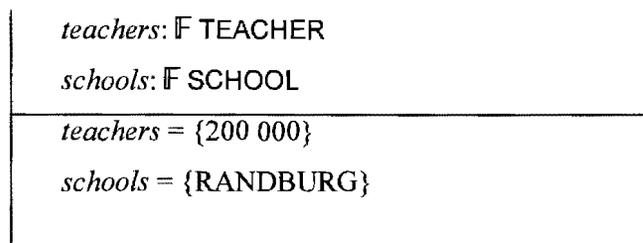
- Assume the following (sorted in ascending sequence) set of teachers employee numbers that contains 5 elements: $\{200\ 000, \dots, 240\ 000\}$.

Only one teacher with employee number 200 000 is the principal of the school RANDBURG.

In UML the class Principal can be instantiated as follows:



The axiomatic description of the *PRINCIPAL* (Section 5.6.1.6.1) schema is:



This axiomatic description is also a verification of the consistency of the global definitions

or

$$\vdash \exists teachers : \mathbb{F} TEACHER \wedge schools : \mathbb{F} SCHOOL \bullet \forall teachers : TEACHER \wedge \forall schools : SCHOOL \bullet teachers \in TEACHER \wedge schools \in SCHOOL$$

which for the axiomatic description is true, therefore

$$\vdash \exists \text{ teachers: } \mathbb{F} \text{ TEACHER} \wedge \text{ schools: } \mathbb{F} \text{ SCHOOL} \bullet \text{ true}$$

and by a simple property of logic

$$\vdash \text{ true}$$

This satisfies the verifications for global definitions, which can also serve as part of a data refinement because a concrete data type was constructed that simulates the abstract one. (Refer to Section 3.4.2 and Section 5.5.3).

Data refinement and operation refinement can be performed on the specifications in the same manner as demonstrated in Section 5.5.3.

5.6.1.4 Implementation into C

This is a very basic program that does not include data validation and error checking.

Comments: If an association is implemented as a separate object (or struct in C), then no attributes need be added to either class in the association. If the association has a small number of objects participating in it, then a separate association object uses less storage than a pointer in each object [Rum91].

It is assumed that there are 5 teachers, of whom 2 are principals of 2 schools. The teachers array contains the employee numbers of the teachers. Assume that the 2 teachers that are the 2 principals are teachers[1] and teachers[2]. In the main method the employee numbers of the 2 teachers and the names of the 2 schools are printed.

The associations and functions are effectuated by the principal relation struct and the use of arrays for the teachers and schools (refer to Lines 1, 2, 3, 4 and 5). Also refer to the output after the program.

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
```

```
struct teacher
{
```

```
    int teachers[5];          //Line 1
};

struct school
{
    char schools[2][10];     //Line 2
};

struct principal_rel        //Line 3
{
    struct teacher;         //Line 4
    struct schools;         //Line 5
};

void main(void)
{
    int i, temp_emp_no, j;
    char temp_school_name[10];
    struct principal_rel PRINCIPAL;
    struct teacher TEACHER;
    struct school SCHOOL;
    printf("Enter the 5 teacher employee numbers\n");
    for (i=0; i<=4; i=i+1)
    {
        scanf("%d", &temp_emp_no);
        TEACHER.teachers[i] = temp_emp_no;
    }

    printf("Enter the 2 school names\n");

    for(i=0; i<=1; i=i+1)
    {
        scanf("%s", temp_school_name);
        strcpy(SCHOOL.schools[i], temp_school_name);
    }
}
```

```
for (i=0; i<=1; i=i+1)
{
    j = i + 1;
    printf("The name of school number %d is %s\n",j,
        SCHOOL.schools[i]);
    printf("The employee number of the school principal is %d\n",
        TEACHER.teachers[i]);
}
printf("The program has been executed\n");
getche();

};
```

Output

Enter the 5 teacher employee numbers

200000

210000

220000

230000

240000

Enter the 2 school names

RANDBURG

LINDEN

The name of school number 1 is RANDBURG

The employee number of the school principal is 200000

The name of school number 2 is LINDEN

The employee number of the school principal is 210000

The program has been executed

5.6.1.5 Implementation into Cobol

As with the C program, the associations and functions are accomplished by the inclusion of teachers and schools in the principal relation structure, and the use of arrays (Refer to lines 1, 2 and 3).

.....

.....

working-storage section.

* Line 1 *****

01 principal_rel.

* Line 2 *****

05 teachers pic 9(6) occurs 5 times.

* Line 3 *****

05 schools pic x(10) occurs 2 times.

01 i pic 9(2).

01 temp_emp_no pic 9(6).

01 temp_school_name pic x(10).

procedure division.

main_rtn.

perform load_rtn1 varying i from 1 by 1 until i > 5.

perform load_rtn2 varying i from 1 by 1 until i > 2.

stop run.

load_rtn1.

display "Enter employee number ", i.

accept temp_emp_no.

move temp_emp_no to teachers(i).

display "The employee number of teacher number ",

i, " is ", teachers(i).

load_rtn2.

display "Enter the name of school number ", i.

display "Fill to 10 characters with spaces."

accept temp_school_name.

move temp_school_name to schools(i).

display " School ", schools(i),

"has got principal ", teachers(i).

.....

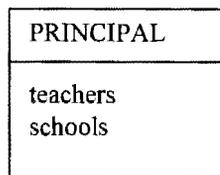
5.6.1.6 Comparison

5.6.1.6.1 Specification

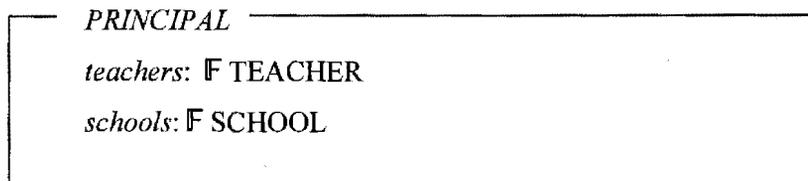
- In Z the one-to-one relationship can be modelled in either direction as a partial injection (\rightarrow). An additional predicate specifies that one relationship is the inverse of the other. In

the UML specification, this one-to-one relationship is as indicated in Figure 5.5 (i.e. (1..1)).

- In UML, one-to-one relationships are used to model a special type of subtype. A supertype entity comprises attributes and relationships common to all subtypes. There is no notion of sub typing in Z [Pol92].
- For this specific example, in my opinion, the specification in UML is much more clear, graphic and concise than that of the Z specification, because the main features of the specification can be immediately seen from one glance at the picture(s).
- An object class for a principal can for example be defined in UML as follows (refer to Figure 5.5):



The corresponding *PRINCIPAL* schema in Z is



5.6.1.6.2 Verification and Refinement

A comparison between Z and UML for the verification and refinement is given in Section 5.6.1.3.

5.6.1.6.3 Implementation

- For this example, the teachers (teacher employee numbers) are implemented in the C and Cobol programs in arrays containing five items, each with their own unique key. The arrays are defined as int teachers[5] (in C), and teachers pic 9(6) occurs 5 times (in Cobol).
- The dom *is_principal_of* is one teacher, for example the first teacher in the array with employee number 200 000. The ran *is_principal_of* is one school, namely RANDBURG. Therefore teachers[1] = 200 000 is the principal of school RANDBURG.

- The predicate *has_principal = is_principal_of* indicates that RANDBURG has one teacher who is the principal, but not every teacher is the principal of the school.

5.6.1.6.4 Conclusion

- For the UML and Z the consistency of the global definitions are verified by the instantiation of the classes and axiomatic descriptions respectively.
- From the specifications of Z and UML, each is clear and descriptive enough to facilitate the implementation into C and Cobol. Although the specifications are different in each case, they convey the same information. The specification in UML is in my opinion easier to read and understand because of the graphic nature of UML – the specifications can be absorbed at a glance and do not need the detailed study of Z.
- The refinement process of Z (not illustrated in detail in this section) provides a bridge from the specification into the programming code that UML does not have.

5.6.2 One-to-many and many-to-one relationships

Consider the following example: There are a number of schools. Every school has a number of teachers. Every teacher is employed by only one school. In Z (Figure 5.8) an additional predicate specifies that one relationship is the inverse of the other ($\bar{\cdot}$): the inverse is the function representing the equivalent many-to-one relationship. An alternative specification style in Z for one-to-many relationships is a partial function mapped onto a single instance from the domain to a set of instances from the range (refer to Figure 5.8) [Sem90]: $SCHOOL \rightarrow TEACHER$

5.6.2.1 Specifications in UML

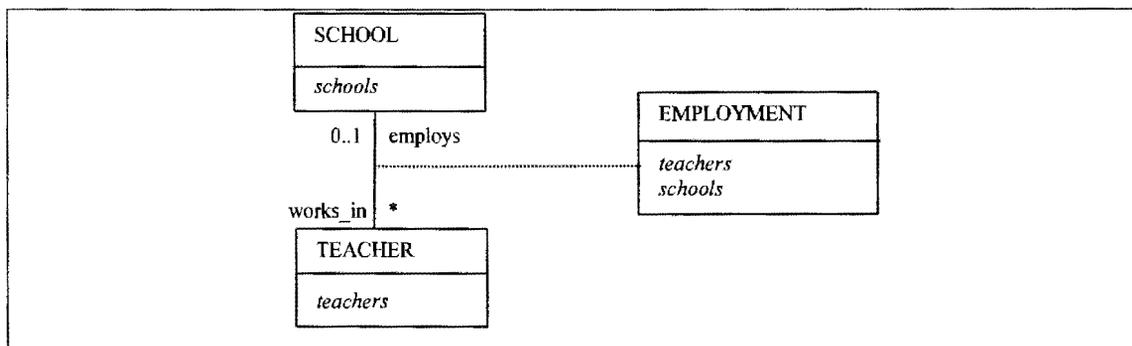


Figure 5.7 One-to-many relationship and its inverse (mandatory on Teacher side) in UML

5.6.2.2 Specifications in Z

The basic types are: [TEACHER_ID, SCHOOL_NAME]

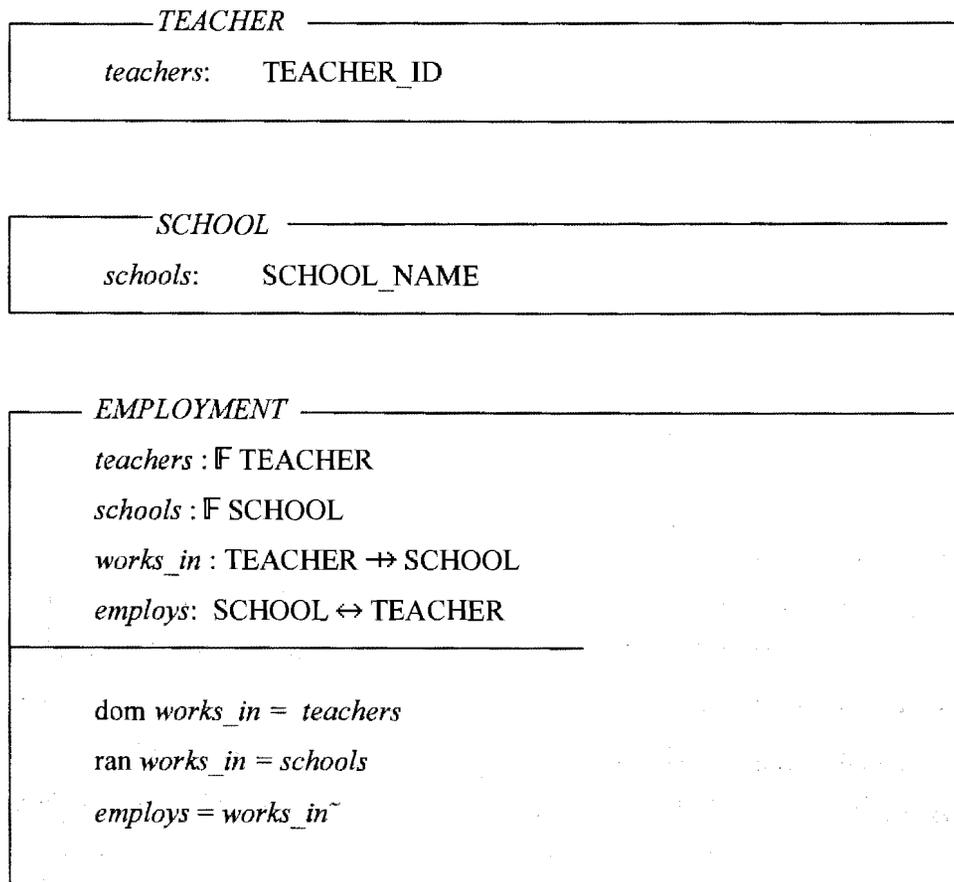


Figure 5.8 One-to-many relationship and its inverse (mandatory on Teacher side) in Z

5.6.2.3 Verification and Refinement

- Assume the following set of five teacher employee numbers: $TEACHER = \{200000, 210000, 220000, 230000, 240000\}$. Assume that school number 1 is RANDBURG, school number 2 is LINDEN. Therefore $SCHOOL = \{RANDBURG, LINDEN\}$.

In Z a concrete representation for one teacher in RANDBURG of the *EMPLOYMENT1* (Section 5.6.2.6.1) schema can be the axiomatic definition

$teachers : \mathbb{F} \text{TEACHER}$ $schools : \mathbb{F} \text{SCHOOL}$
$teachers = \{200000\}$ $schools = \{\text{RANDBURG}\}$

or

$$\vdash \exists teachers: \mathbb{F} \text{TEACHER} \wedge schools: \mathbb{F} \text{SCHOOL} \bullet \forall teachers: \text{TEACHER} \wedge \\ \forall schools: \text{SCHOOL} \bullet teachers \in \text{TEACHER} \wedge schools \in \text{SCHOOL}$$

which for the axiomatic description is true, therefore

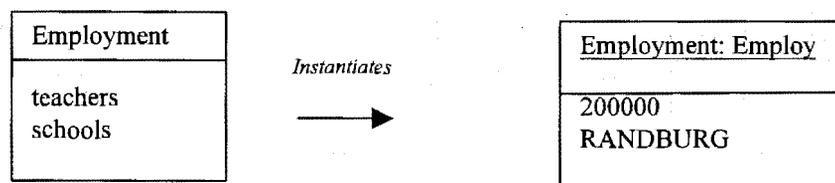
$$\vdash \exists teachers: \mathbb{F} \text{TEACHER} \wedge schools: \mathbb{F} \text{SCHOOL} \bullet true$$

and by a simple property of logic

$$\vdash true$$

This satisfies the verifications for the consistency of the global definitions, which can also serve as part of a data refinement because a concrete data type was constructed that simulates the abstract one. (Refer to Section 3.4.2 and Section 5.5.3).

In UML the class Employment can be instantiated as follows:



A detailed data and operation refinement process for Z can also be provided (refer to Section 5.5.3).

5.6.2.4 Implementation into C

This is a very basic program and does not include data validation and error checking.

Comments: Assume there are 5 teachers with employee numbers {200000, 210000, 220000, 230000, 240000}. The first 2 teachers in the set work for school 1 (RANDBURG) and the last 3 teachers work for school 2 (LINDEN).

The associations and functions are implemented by the use of the employment structure that includes the teachers and schools structures, and by the use of arrays for representing the teachers in the schools (refer to lines 1, 2, 3, 4, and 5).

```
#include <stdio.h>
#include <string.h>
#include <conio.h>

struct teacher
{
    int teachers[5];          //Line 1
};

struct school
{
    char schools[2][10];     //Line 2
};

struct employment          //Line 3
{
    struct teacher;         //Line 4
    struct school;         //Line 5
};

void main(void)
{
    int i, temp_emp_no, j;
    char temp_emp_name[10];
    struct employment EMPLOYMENT;
    struct teacher TEACHER;
    struct school SCHOOL;

    printf("Enter the 2 school names\n");

    for(i=0; i<=1; i=i+1)
```

```
{
    scanf("%s", temp_emp_name);
    strcpy(SCHOOL.schools[i], temp_emp_name);
    j = i + 1;
    printf("The name of school number %d is %s\n", j, SCHOOL.schools[i]);
}

printf("Enter the 5 teacher employee numbers for schools 1 and 2\n");

for (i=0; i<=4; i=i+1)
{
    scanf("%d", &temp_emp_no);
    TEACHER.teachers[i] = temp_emp_no;
}

printf("The employee numbers of the teachers for school 1 are\n");
for (i=0; i<=1; i=i+1)
{
    printf("%d\n", TEACHER.teachers[i]);
}

printf("The employee numbers of the teachers for school 2 are\n");
for (i=2; i<=4; i=i+1)
{
    printf("%d\n", TEACHER.teachers[i]);
}

printf("The program has been executed\n");
getche();
};
```

Output:

Enter the 2 school names

RANDBURG

The name of school number 1 is RANDBURG

LINDEN

The name of school number 2 is LINDEN

Enter the 5 teacher employee numbers for schools 1 and 2

200000

210000

220000

230000

240000

The employee numbers of the teachers for school 1 are

200000

210000

The employee numbers of the teachers for school 2 are

220000

230000

240000

The program has been executed

5.6.2.5 Implementation into Cobol

The associations and functions are implemented using the 01 employment level that includes the teachers and schools, as well as the use of the arrays for the teachers and schools. Refer to lines 1, 2, and 3.

.....

working-storage section.

* Line 1 *****

01 employment.

* Line 2 *****

05 teachers pic 9(6) occurs 5 times.

* Line 3 *****

05 schools pic x(10) occurs 2 times.

01 i pic 9(1) value 0.

01 temp_emp_no pic 9(6).

01 temp_school_name pic x(10).

procedure division.

main_rtn.

```
perform load_rtn1 varying i from 1 by 1 until i > 2.
display "Enter employee number(s) for school 1 ".
perform load_rtn2 varying i from 1 by 1 until i > 2.
display "Enter employee number(s) for school 2 ".
perform load_rtn2 varying i from 3 by 1 until i > 5.
display "The 2 schools are:".
perform display_rtn1 varying i from 1 by 1 until i > 2.
display "The teachers for school 1 are: ".
perform display_rtn2 varying i from 1 by 1 until i > 2.
display "The teachers for school 2 are: ".
perform display_rtn2 varying i from 3 by 1 until i > 5.
stop run.
```

load_rtn1.

```
display "Enter the name of school number ", i.
display "Fill to 10 characters with spaces.".
accept temp_school_name.
move temp_school_name to schools(i).
display " School number ", i, " is ", schools(i).
```

load_rtn2.

```
accept temp_emp_no.
move temp_emp_no to teachers(i).
```

display_rtn1.

```
display schools(i).
```

display_rtn2.

```
display teachers(i).
```

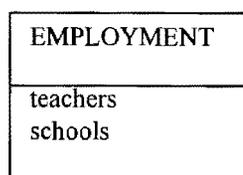
.....

5.6.2.6 Comparison

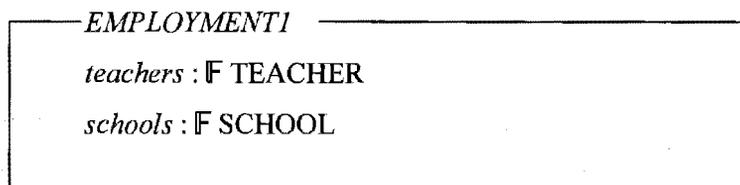
5.6.2.6.1 Specification

- In Z the one-to-many relationship (*employs*) is indicated by the relation symbol \leftrightarrow . The inverse of this relation is a function representing the equivalent many-to-one relationship (*works_in*). Refer to the predicate $employs = works_in\tilde{}$.

- An alternate specification style for one-to-many relationship is: $employs: SCHOOL \rightarrow F TEACHER$. A partial function maps a single instance from the domain ($SCHOOL$) to a set of instances from the range ($TEACHER$). In UML the one-to-many relationship is indicated by (0..1..*) (Figure 5.7).
- In Z the many-to-one relationship can be modelled as a partial surjection, or partial function (\rightarrow). In UML the many-to-one relationship is indicated by (*..0..1) (see Figure 5.7).
- An Employment class can be defined in UML as follows (refer to Figure 5.7):



The corresponding $EMPLOYMENT1$ schema in Z is



5.6.2.6.2 Verification and Refinement

A comparison between Z and UML for the verification and refinement is given in Section 5.6.2.3.

5.6.2.6.3 Implementation

- The UML and Z specifications are implemented into C and Cobol programs where the five teacher employee numbers and two school names stored in arrays. The classes (schemas) are represented by the C struct employment and 01 employment construct in Cobol.
- The dom $works_in$ (relation $works_in$ in UML) for example the school RANDBURG are 2 teachers (indicated by * (more than one) in UML) with employee number 200000 and 210000. The ran $works_in$ is one school (indicated by 0..1 in UML), namely RANDBURG.
- The predicate $employs = works_in$ indicates that RANDBURG has two teachers who work there, but not every teacher works in RANDBURG.

5.6.2.6.4 Conclusion

- For the UML and Z the consistency of the global definitions are verified by for example, the instantiation of the class Teacher and the axiomatic descriptions respectively.
- From the specifications the implementations into C and Cobol for Z and UML provide about the same amount of detail. However, the additional refinement steps (which follow a similar pattern as Section 5.5.3 – not given here because of lack of space) of Z results in the capability of Z being more directly implementable into programming code. This does however add to the complexity of Z.

5.6.3 Many-to-many relationships

Consider the following example. There are many magazines and each magazine has many advertisements. For example, 5 different types of magazines have been printed, each containing a random choice of 6 advertisements from a collection of 10 advertisements. Each chosen advertisement is placed in any 3 of the 5 magazines. As before, for the Z specification an additional predicate specifies that one relationship is the inverse of the other ($\bar{}$).

5.6.3.1 Specifications in UML

Refer to Figure 5.9.

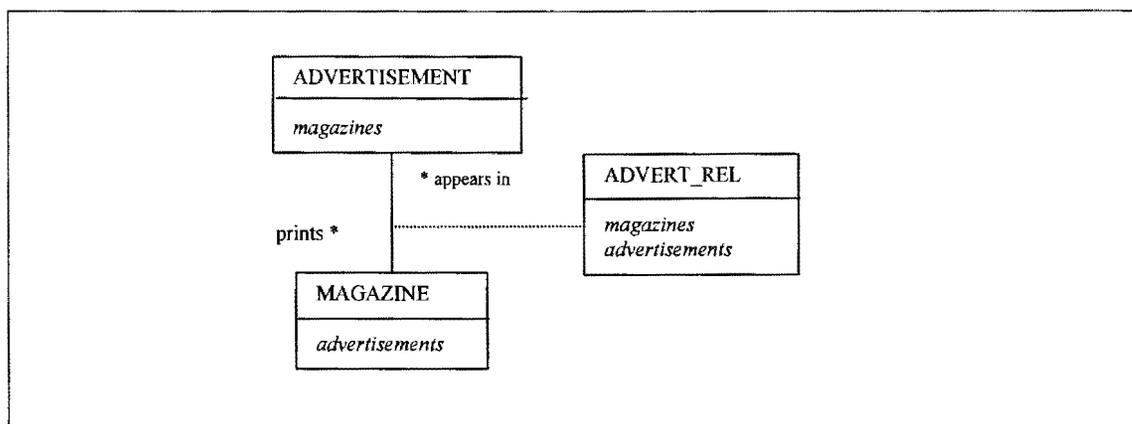


Figure 5.9 Many-to-many relationships in UML

5.6.3.2 Specifications in Z

[MAGAZINE, ADVERTISEMENT]

The basic types are: [MAGAZINE, ADVERTISEMENT]

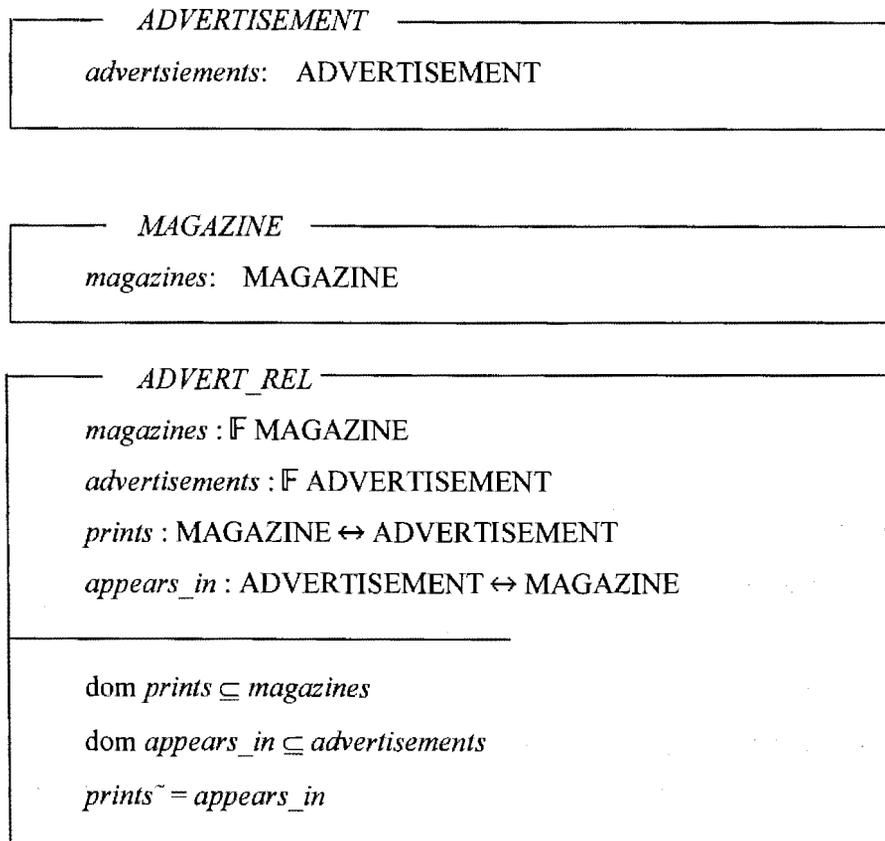


Figure 5.10 Many-to-many relationships in Z

5.6.3.3 Verification and Refinement

- Assume the following sets:

MAGAZINE = {SCIENCE_SA, TRAVEL_SA, ..., COMPUTER_SA}

ADVERTISEMENT = {STUDENTBANK1, FNB1, ..., ORACLE1}

In Z a concrete representation of the *ADVERT_REL1* (Section 5.6.3.6.1) schema can be the axiomatic definition:

$magazines: \mathbb{F} \text{ MAGAZINE}$ $advertisements: \mathbb{F} \text{ ADVERTISEMENT}$
$magazines = \{\text{SCIENCE_SA}\}$ $advertisements = \{\text{STUDENTBANK1}\}$

or

$$\vdash \exists magazines: \mathbb{F} \text{ MAGAZINE} \wedge advertisements: \mathbb{F} \text{ ADVERTISEMENT} \bullet \forall$$

$$magazines: \text{MAGAZINE} \wedge \forall advertisements: \text{ADVERTISEMENT} \bullet magazines \in$$

$$\text{MAGAZINE} \wedge advertisements \in \text{ADVERTISEMENT}$$

which for the axiomatic description is true, therefore

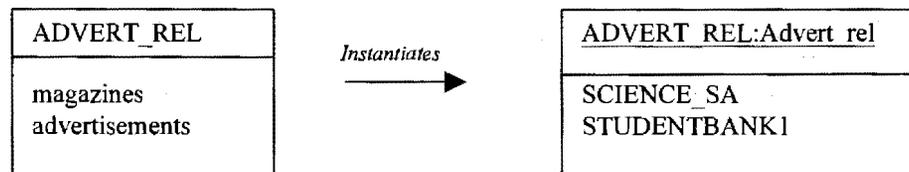
$$\vdash \exists magazines: \mathbb{F} \text{ MAGAZINE} \wedge advertisements: \mathbb{F} \text{ ADVERTISEMENT} \bullet \text{true}$$

and by a simple property of logic

$$\vdash \text{true}$$

This satisfies the verifications for global definitions, which can also serve as part of a data refinement because a concrete data type was constructed that simulates the abstract one. (Refer to Section 3.4.2 and Section 5.5.3).

- In UML the class ADVERT_REL can be instantiated as follows:



Data and operation refinement in Z can be provided, similar to Section 5.5.3.

5.6.3.4 Implementation into C

```

#include <stdio.h>
#include <string.h>
#include <conio.h>
/*Magazine*/
struct magazine
{
    int magazines[2][3]; //Line 1: There are 2 magazines with up to 3 advertisements
  
```

```
    char magname[6][20];                // in each.
};
/*Advertisement*/
struct advertisement
{
    int advertisements[3][2]; //Line 2: There are 3 advertisements placed in up to
    char advertname[6][20];        // 2 magazines.
};
/*advert_rel*/
struct advert_rel          //Line 3: This is the association structure
{
    struct magazine;
    struct advertisement;
};

void main(void)
{
    int i, j, j1, i1;
    char temp_mag_name[20], temp_advert_name[20];
    struct magazine MAGAZINE;
    struct advertisement ADVERTISEMENT;
    struct advert_rel ADVERT_REL;
    printf("Enter the 2 magazine names\n");
    for(i=0; i<=1; i=i+1)
    {
        scanf("%s", temp_mag_name);
        strcpy(MAGAZINE.magname[i], temp_mag_name);
    }
    printf("Enter the 3 advertisments\n");
    for (i=0; i<=2; i=i+1)
    {
        scanf("%s", temp_advert_name);
        strcpy(ADVERTISEMENT.advertname[i], temp_advert_name);
    }
    for (i=0; i<=1; i=i+1)
```

```
{
    i1 = i + 1;
    printf("The advertisements in magazine number %d name %s are\n",
        i1, MAGAZINE.magname[i]);
    for (j=0; j<=2; j=j+1)
    {
        j1 = j + 1;
        printf("Advertisement no %d %s \n", j1, ADVERTISEMENT.advertname[j]);
    }
}
for (i=0; i<=2; i=i+1)
{
    i1 = i + 1;
    printf("The magazines that published advertisement number %d name %s are:\n",
        i1, ADVERTISEMENT.advertname[i]);
    for (j=0; j<=1; j=j+1)
    {
        j1 = j + 1;
        printf("Magazine no %d %s \n", j1, MAGAZINE.magname[j]);
    }
}
getche();
};
```

Output

Enter the 2 magazine names

SCIENCE_SA

TRAVEL_SA

Enter the 3 advertisements

STUDENTBANK1

FNB1

ORACLE1

The advertisements in magazine number 1 name SCIENCE_SA are

Advertisement no 1 STUDENTBANK1

Advertisement no 2 FNB1

Advertisement no 3 ORACLE1

The advertisements in magazine number 2 name TRAVEL_SA are

Advertisement no 1 STUDENTBANK1

Advertisement no 2 FNB1

Advertisement no 3 ORACLE1

The magazines that published advertisement number 1 name STUDENTBANK1 are:

Magazine no 1 SCIENCE_SA

Magazine no 2 TRAVEL_SA

The magazines that published advertisement number 2 name FNB1 are:

Magazine no 1 SCIENCE_SA

Magazine no 2 TRAVEL_SA

The magazines that published advertisement number 3 name ORACLE1 are:

Magazine no 1 SCIENCE_SA

Magazine no 2 TRAVEL_SA

5.6.3.5 Implementation into Cobol

For the purposes of this program there are 2 magazines that publishes 3 advertisements each.

The many-to-many relationship is accomodated by the 01 advert_rel and the arrays.

.....

01 advert_rel.

05 magazines1 occurs 2 times.

10 advertisements1 pic x(20) occurs 3 times.

05 advertisements2 occurs 3 times.

10 magazines2 pic x(20) occurs 2 times.

.....

The names of the advertisements and magazines were loaded into the tables and are displayed.

.....

perform display_rtn1 varying i from 1 by 1 until i = 2.

perform display_rtn3 varying i from 1 by 1 until i = 3.

.....

display_rtn1.

display "Magazine number ", i, " has the following advertisements: ".

perform display_rtn2 varying j from 1 by 1 until j = 3.

display_rtn2.

display advertisements1(i, j).

display_rtn3.

display "Advertisement number", i , " appears in the following magazines: "

perform display_rtn4 varying j from 1 by 1 until j = 2.

display_rtn4.

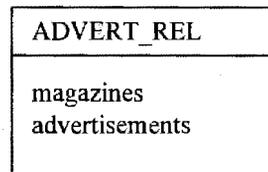
display magazines2(i , j).

.....

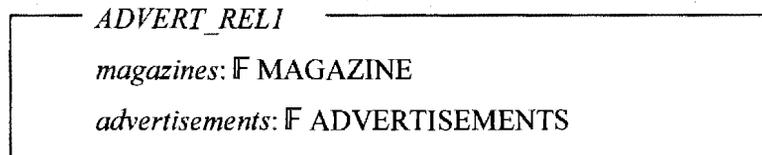
5.6.3.6 Comparison

5.6.3.6.1 Specification

- A many-to-many relationship in Z is modelled by a relation (i.e. \leftrightarrow).
- One relationship (*appears_in*) is represented formally by a mathematical relation where the inverse is the function representing the equivalent many-to-many relationship (*prints*). Refer to the predicate $prints^{\sim} = appears_in$.
- In UML the many-to-many relationship is indicated by (*..*) (Figure 5.9).
- An ADVERT_REL class, is for example defined in UML as follows (refer to Figure 5.9):



with the corresponding *ADVERT_REL1* schema in Z:



5.6.3.6.2 Verification and Refinement

A comparison between Z and UML for the verification and refinement is given in Section 5.6.3.3.

5.6.3.6.3 Implementation

- The UML and Z specifications are implemented into C and Cobol programs where for example, the advert_rel struct (C) and advert_rel record (Cobol) contain the magazine names and the names of the advertisements that are placed in the magazines.

5.6.3.6.4 Conclusion

- For the UML and Z the consistency of the global definitions are verified by, for example, the instantiation of the class magazine and the axiomatic descriptions respectively.
- Z provides more detailed refinement than UML, but the essence of the requirements for an implementation can be adequately inferred from the UML specifications e.g. the advert_rel class in UML that was implemented as the advert_rel struct and the advert_rel record in C and Cobol respectively. The Z specifications and refinement provide more detailed information, but the UML is more concise, graphic and easier to read.

5.6.4 Resolved many-to-many relationships

Many-to-many links are resolved by the addition of a new entity, with many-to-one relationships to each of the existing entities.

5.6.4.1 Specifications in UML

Refer to Figure 5.11.

5.6.4.2 Specifications in Z

Refer to Figure 5.12 and the Z specifications in Section 5.6.4.3.

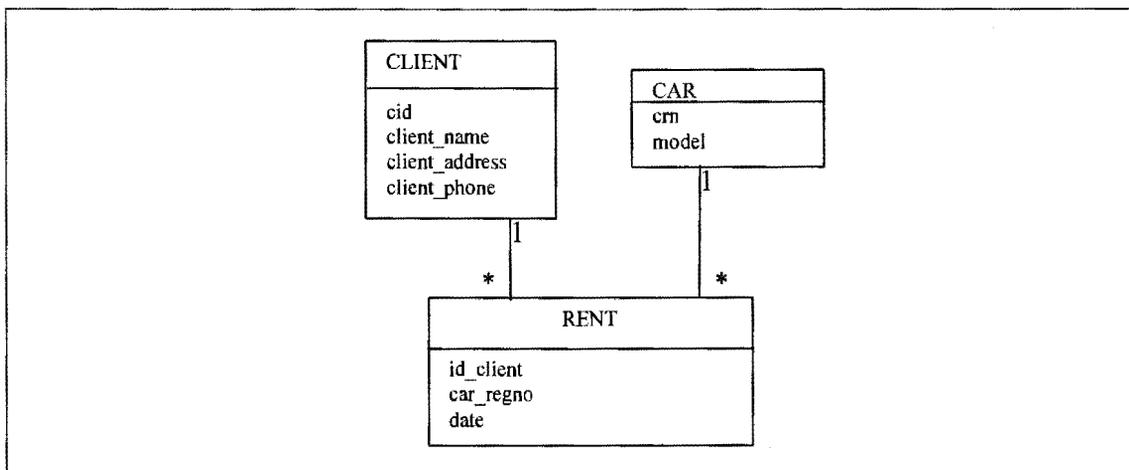


Figure 5.11 A resolved many-to-many relationship in UML

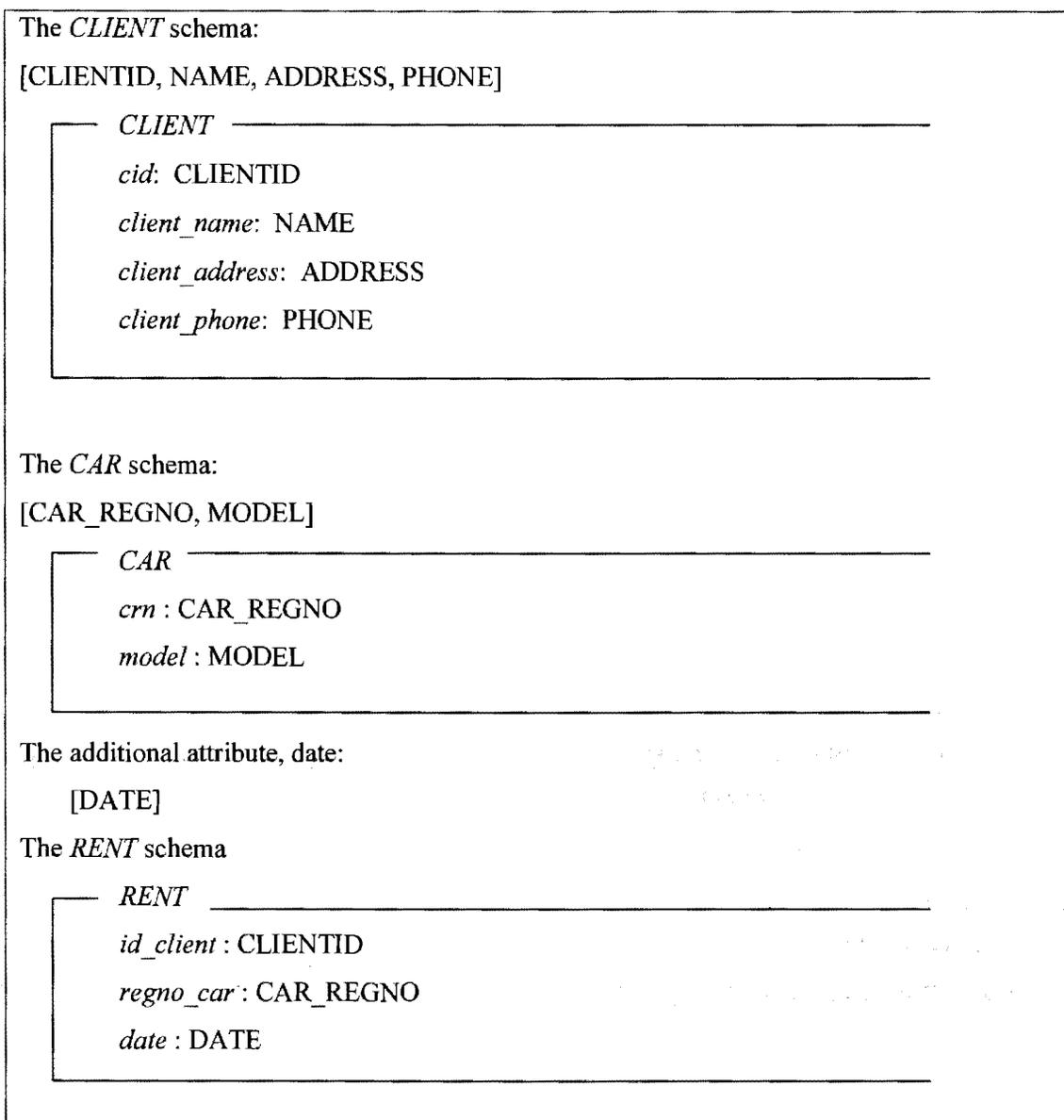
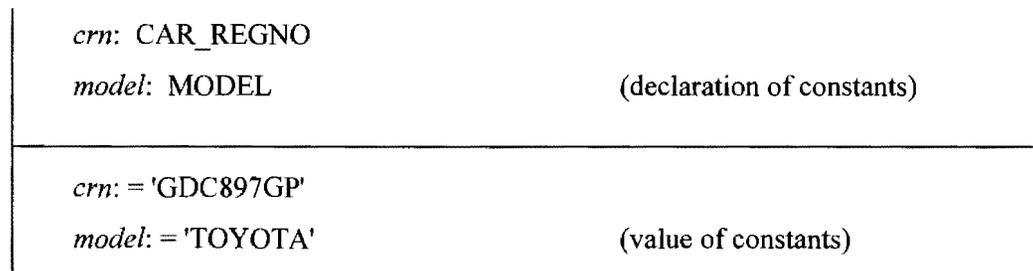


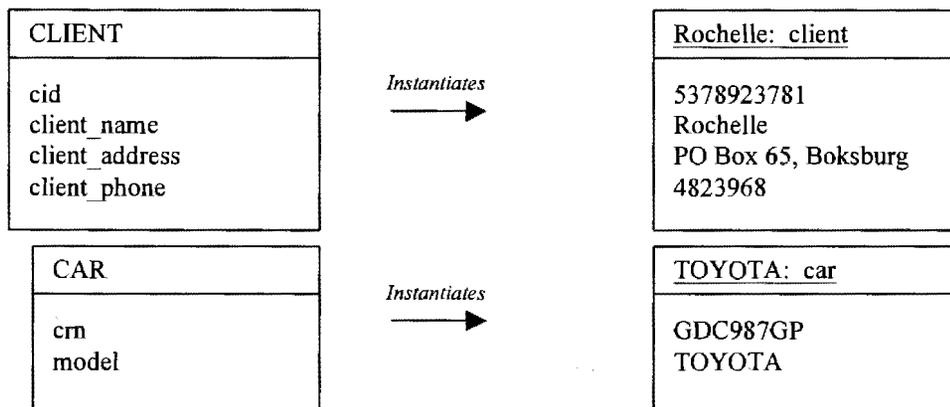
Figure 5.12 A resolved many-to-many relationship in Z

5.6.4.3 Verification and Refinement

- For both the Z and UML there are representative concrete data types for the abstract data types.
- For the Z specification, a concrete representation of the abstract schemas can be provided by axiomatic descriptions giving a constant value to each of the variables. For example:

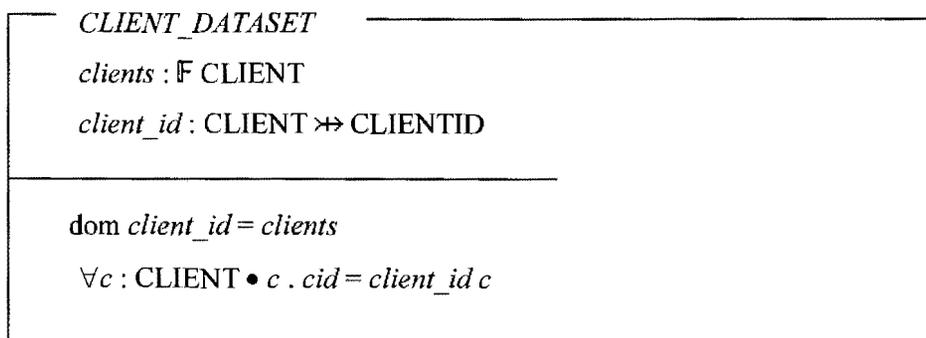


- For the UML specification the classes CLIENT and CAR can be instantiated as follows:



- The *CLIENT* schema can be data refined into the *CLIENT_DATASET* schema:

Sets of instances of the master entities are defined:



The first variable declaration is a finite set of instances of the entity type CLIENT. The second variable declaration is a function linking the entity type to the type of the key attribute CLIENTID. The function is a partial injection that maps exactly one instance of the domain type to each participating instance of the range.

The first predicate restricts the domain to the set of known clients. The second predicate specifies that for every instance of CLIENT, the key attribute is the same as its unique identifier, in the range of the `client_id` function.

- The *CAR* schema can be refined into the *CAR_DATASET* schema:

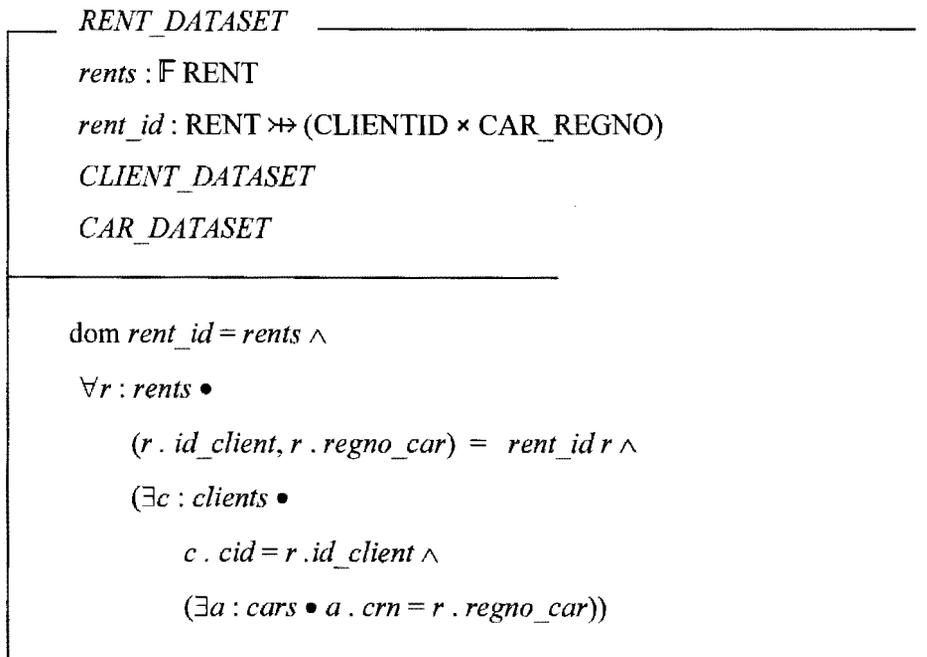
<i>CAR_DATASET</i>
$cars : \mathbb{F} CAR$ $car_regno : CAR \rightsquigarrow CAR_REGNO$
$dom\ car_regno = cars$ $\forall a : CAR \bullet a.crn = car_regno\ a$

The first variable declaration is a finite set of instances of the entity type. The second variable declaration is a function linking the entity type to the type of the key attribute *CAR_REGNO*. The function is a partial injection that maps exactly one instance of the domain type to each participating instance of the range.

The first predicate restricts the domain to the set of known cars. The second predicate specifies that for every instance of *CAR*, the key attribute is the same as its unique identifier, in the range of the `car_regno` function.

The integrity of the system is maintained by the specification of the uniqueness of the key attribute *id_client* and *regno_car* of the associative entity, and therefore ensures that its components are identifiers of clients and cars known to the system.

- The *RENT* schema can be data refined into the *RENT_DATASET* schema:



5.6.4.4 Implementation into C

Consider the following example. There are a number of clients who rents a number of cars on different dates. Given two classes CLIENT and CAR. The resolution of the relationship between classes CLIENT and CAR is done by the addition of an associative entity RENT.

.....

```
/*CLIENT*/
```

```
struct client
```

```
{
```

```
int cid;
```

```
char client_name;
```

```
char client_address;
```

```
int client_phone;
```

```
};
```

```
/*CAR*/
```

```
struct car
```

```
{
```

```
char crn;
```

```
char model;
```

```
};
```

```
/*DATE*/
```

```
struct date
```

```
{
```

```
int date1;
```

```
};
```

```
/*RENT*/
```

```
struct rent
```

```
{
```

```
int id_client;
```

```
char regno_car;
```

```
int date2;
```

```
};
```

//The relation can be implemented by a C structure and the relation could be a file or in-memory data structure organised to permit rapid search and retrieval.

```
void main(void)
```

```
{
```

```
.....
```

```
    struct client CLIENT;
```

```
    struct car CAR;
```

```
    struct date DATE;
```

```
    struct rent RENT;
```

```
    RENT.id_client = CLIENT.cid;
```

```
    RENT.regno_car = CAR.crn;
```

```
    RENT.date2 = DATE.date1;
```

```
.....
```

```
};
```

5.6.4.5 Implementation into Cobol

```
.....
```

```
Fd client_file.
```

```
01 client_record.
```

```
    05 cid                pic 9(10).
```

```
    05 client_name        pic x(10).
```

```
    05 client_address     pic x(20).
```

```
    05 client_phone       pic 9(7).
```

```
.....  
Fd car_file.  
01 car_record.  
    05 crn                pic x(10).  
    05 model              pic x(10).  
.....  
01 date1.  
    05 day                pic 9(2).  
    05 month              pic 9(2).  
    05 year               pic 9(2).  
.....  
Fd rent_file.  
01 rent_record.  
    05 id_client          pic 9(10).  
    05 regno_car          pic x(10).  
    05 date2              pic 9(6).  
.....  
    move cid to id_client.  
    move car_regno to regno_car.  
    move date1 to date2.  
.....
```

5.6.4.6 Comparison

5.6.4.6.1 Specification

- In UML many-to-many links are resolved by adding a new detail entity, with many-to-one relationships to each existing entity. It may be implied by using foreign keys from the master entities as the composite key of the new entity. The Z notation, however, emphasises the form of the original relationship [Pol92].
 - In the UML notation, the resolution of the relationship between entities CLIENT and CAR is done by the addition of an associative entity RENT. In Z, the type of the associative entity comprises a compound key, made up of the types of the key attributes *cid* and *crn* of the two master entities CLIENT and CAR respectively (refer to the *RENT* schema).
-

A foreign key is an attribute (possibly composite) of one relation, R2 (say) whose values are required to match those of the primary key of some relation R1 (say). (R1 and R2 are not necessary distinct).

5.6.4.6.2 Verification and Refinement

A comparison between Z and UML for the verification and refinement is given in Section 5.6.4.3.

5.6.4.6.3 Implementation

- The CLIENT, CAR, RENT classes (UML) and the *CLIENT*, *CAR*, *RENT* schemas (Z) are implemented respectively into the client, car and rent structs (C) and the client_file, car_file and rent_file (Cobol).
- The integrity and uniqueness of the composite keys are ensured by the = statements in C and the move statements in Cobol.

5.6.4.6.4 Conclusion

- In conclusion it can be mentioned that for the implementation of the different relationships (one-to-one, many-to-one, many-to-many, resolved many-to-many), is the UML specification easier to read and understand, but because the Z specification provides more detail, it is easier to implement into an implementation language from the Z specifications.
 - The Z refinement provides, strictly speaking, prove of the correctness than is the case for UML, but the generalisation structures in the UML specification ensures that only information of clients and cars known to the system is used for the composite key of the rent class.
 - It appears that the axiomatic descriptions (Z) and class instantiations (UML) provide equal amount of verification of the global definitions.
 - It can be concluded that both Z and UML provides each in their own unique way, an equal amount of clarity as far as the specification is concerned. Z provides more detailed refinement than UML, but Z can become very mathematical, which can make it hard to understand.
-

5.7 Summary and conclusions

In this chapter Z and UML have been compared for some aspects regarding specification, refinement and implementation.

From the opinions of the different authors, it can be concluded that both Z and UML have their own particular advantages and disadvantages. As far as the specification notations are concerned, these strengths and weaknesses can be summarised as follows [Fow98, Sch99, Pol92, Boo97, Rum97, Jac97]:

UML

Category: Semiformal

- Strengths:*
- Can be understood by the client
 - More precise than informal methods
 - Many people find these methods useful
 - These methods have little rigour and the notation appeals to intuition rather than formal definition
 - Visual and graphic
 - Communication medium for non-experts.
- Weaknesses:*
- The specification is not as precise as formal methods
 - The specification generally cannot handle timing
 - The consistency of the data and processing is not so easy to check.

Z

Category: Formal

- Strengths:*
- Mathematically precise
 - Can reduce specification faults
 - Can reduce development cost and effort
 - Can support correctness proving
 - Allows no ambiguity
 - Mathematically rigorous.
- Weaknesses:*
- Potentially difficult to learn
 - Difficult to use
 - Often hard for clients to understand
-

- There is no way to prove that the mathematical specification actually meets the real requirements of the system
- Formal methods are hard to understand and manipulate, often more so than the programming languages itself.

It can be concluded that for the detail implementation, a more detailed specification such as Z is more appropriate. The specifications in UML, however, provide a more broad and easily comprehensible picture. Therefore, it can be concluded that a possible combination between Z and UML will provide the ideal solution if both the detail and graphic understanding is needed.

The implementation languages used to illustrate the implementation from the specifications are C and Cobol. These two languages were compared in Chapter 4.

Chapter 6

From specification through refinement to implementation for object-oriented systems: Object-Z and UML compared

In this chapter some of the differences between Object-Z and UML for the specification, refinement and implementation phases of the software system development cycle are examined.

6.1 Introduction

Because the refinement processes for Z and UML have already been discussed extensively in chapters three and four, it will not be repeated in this chapter. This chapter starts with an overview of an object-oriented approach to modeling systems, followed by a short discussion on object-oriented refinement. Then follows a brief overview of Object-Z and C++. Then follows a comparative study between Object-Z and UML as far as specifications, refinements and the implementation of the object-oriented design into an object-oriented language.

A basic knowledge of Z, Object-Z, C++ and UML is assumed in this chapter. Z and UML have been discussed in chapters three and four respectively.

6.2 An object-oriented approach to modeling systems

6.2.1 Terminology

The following definitions related to systems, object-oriented modeling and specifications are of importance to this chapter [vHa93, p.200]:

- A *system* is some part of the world that is of interest to a designer for a particular period.
 - A designer can create a *model* to record the behaviour of an existing system, or to specify the behaviour of a system which is still to be built.
 - Models are created using a *notation*. The particular notation used depends on the designer's view of the world and systems in the world.
-

- In the object-oriented view, systems are composed of *objects* which may have relations between them.
- A *part-of-relation* indicates that one object is part of another object.
- A *general relation* indicates some other more general relation between objects.
- Objects which have the same characteristics have a common *type* which defines the objects' characteristics and behaviour.
- An *inheritance relation* between two types indicates that the *subtype* in the relation inherits the characteristics of the *supertype*.
- Each object has *attributes* which are inaccessible from outside the object, and *methods* which provide services, including changing attribute values.
- Any method can utilize the services of another method that is in the same object or in an object connected by a part-of or a general relation.
- In moving from the general object-oriented approach to object-oriented specification, there are *operations* upon types, rather than methods.
- Any specification of the behaviour of operations can be expressed in terms of the effects on a *state*. The state consists of *state variables*, which will be called *attributes*.

6.2.2 Object-oriented methods, analysis and design

According to Booch [Boo94], for all things object-oriented, the conceptual framework is the *object model*. There are four major elements of this model:

- *Abstraction*. This denotes the essential characteristics of an object that distinguish it from all other kinds of objects and provides therefore clearly defined conceptual boundaries, relative to the perspective of the viewer.
- *Encapsulation*. This is the process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour. Encapsulation separates the contractual interface of an abstraction and its implementation.
- *Modularity*. This is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.
- *Hierarchy*. This constitutes a ranking or ordering of abstractions.

There are three minor (useful but not essential) elements of the object model:

- *Typing*. Typing enforces the class of an object, such that objects of different types may not be interchanged, or only interchanged in very restricted ways.
-

- *Concurrency*. This is the property that distinguishes an active object from one that is not active.
- *Persistence*. This is the property of an object through which its existence transcends time (that is, the object continues to exist after its creator ceases to exist) and/or space (that is, the object's location moves from the address space in which it was created).

'Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.' (Booch [Boo94]).

Booch [Boo94] further states that 'object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.'

Object-oriented (OO) methods unify data and the functions that operate on them into software components called *objects*. For example in the real-time world, objects are models of things such as sensors, motors, and communication interfaces. As an example, the Table 6.1 provides an informal object-oriented description of these kinds of objects [Dou98]:

Object Type	Data	Functions
Temperature Sensor	Temperature	Acquire()
	Calibration Constant	Set Calibration()
Stepper Motor	Position	Step forward()
		Step Backward()
		Park()
		Power()
RS232 Interface	Data to be Transmitted	Send Message()
	Data Received	Receive Message()
	Baud Rate	Set Comm Parameters()
	Parity	Pause()
	Stop Bits	Start()
	Start Bits	Get Error()
	Last Error	Clear Error()

Table 6.1 Object examples

Some objects in a system can be replicates of one another, therefore it would be redundant to specify the data and functions of each. OO methods use the notion of *class* to capture, in one place, the *structure* (for example, the common data fields) and the *behaviour* (for example, common functions callable) of such objects [Boo94].

A class contains both data and function (pointer) fields. A class is like a C-language *struct* declaration. Structs are definitions of groups of fields that are closely related.

An object has a definition that is prescribed by its class. In UML terminology, the data portion of an object is defined by a set of *attributes*, and the functional portion of an object is defined by a set of *operations*. [Rum91]. An object is 'an entity that has a unique identity, specific data values, and specific behaviours of program code' [COBOL 2000 standard] – refer to chapter 7. Objects store and process data. Within an OO system, objects pass data directly and interactively to each other without the use of files [Gra98]. An object is a packaging of procedural code and data, complete within itself, to represent a meaningful entity in the context of an application [Pri97].

Every object of a given class has exactly the same structure and behaviour. Each object is unique in that changes to one do not automatically affect others. For example, every object of the *Temperature Sensor* (see example table by [Dou98]), class has its own copy of the data items *Temperature* and *Calibration Constant*, as well as its own access to the operations *Acquire()* and *Set Calibration()* [Dou98].

A fundamental approach of OO is to structure data and functions into classes. Another key principle of object orientation is *encapsulation*: the data of an object is accessible only through the functions defined by the object's class. Encapsulation makes objects more reliable and safe [Boo94]:

- Users of an object must go through a controlled interface, such as a function call to affect the state of the object.
- Developers of an object's class can make changes to the data portion often with little or no impact on the users.

Inheritance is another essential element of object-oriented systems. Inheritance defines a relationship among classes, wherein one class shares the structure or behaviour defined in one or more classes (single and multiple inheritance respectively). Inheritance therefore represents a

hierarchy of abstractions, in which a subclass inherits from one or more superclasses. A subclass augments or redefines the existing structure and behaviour of its superclasses [Boo94].

According to Grauer [Gra98, Gra00] OO has become an important new way to develop information systems. Systems are developed faster, program code can be reused, and data is better managed.

Booch [Boo94] says that object-oriented analysis and design leads to an object-oriented decomposition. By applying object-oriented design, software is created that is resilient to change and written with economy of expression. A greater level of confidence in the correctness of the designed software is achieved through an intelligent separation of its state space. The risks that are inherent in developing complex software systems are reduced.

6.2.3 The object-oriented versus structured paradigm

When developing an OO system, the designer tries to identify and represent the *nouns* of the system. The nouns come from names of entities necessary to accomplish the system's purpose, such as Student, Invoice, or Employee. These entities become candidates for *classes* in the system. The designers identify and refine the classes, they specify the types of data belonging to, and the behaviour associated with the class. Additional data items and methods are added as additional requirements become apparent. By looking at the nouns, the OO designer can determine how a class should behave in general without regard to any specific system. Therefore, these general class behaviours or methods can be used by many systems. When system-specific requirements dictate the need for additional methods or data items, they can be added to the class without affecting the previously defined data items and methods in the class [Gra98, Gra00].

The structured approach, in contrast however, focuses on the *verbs* of the system. Verbs identify the things a system must do. As each activity of the system is identified, the designer specifies a program(s) to carry it out. These programs are custom designed for their specific system [Gra98, Gra00].

Object-oriented design uses class and object abstractions to logically structure systems. Structured design uses algorithmic abstractions. Traditional structured analysis techniques focus upon the flow of data within a system. Object-oriented analysis emphasizes the building of real-world models, using an object-oriented view of the world [Boo94].

OO systems are more flexible than structured systems [Gra98, Gra00]. By placing procedures and methods contained in classes rather than programs, OO allows common routines to be written only once. Methods and classes can be reused. A change in a method is made only once in a class and systems using that class will use the revised method. New methods can be added to classes for different systems.

Structured systems consist of custom-designed programs. Even when common routines occur, they cannot easily be copied into other programs, nor can other systems just use part of a structured program. This results in new programs being built from scratch with little use of previously developed routines. Programs run in a standalone mode with one program operating at a time. Linkages between programs are maintained by passing files. Objects, unlike programs, are aware of other objects. Messages, rather than files, provide the linkage between objects. Files only store data while the system is not running [Gra98, Gra00].

6.3 Object-Oriented Refinement

6.3.1 Refining classes

Classes within a product can be refined in isolation and thereby simplifying the overall refinement process. In order to simplify the identification of classes and aid the process of refinement, subtasks corresponding to the refinement of classes could be introduced. For example, the operator *Specify Class* could be modified to produce the workplan reduction shown in Figure 6.1.

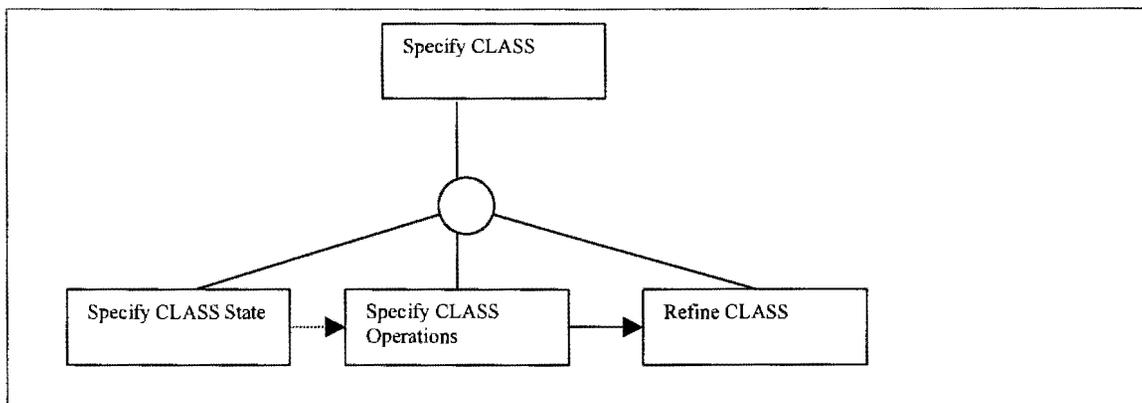


Figure 6.1. The operator Specify Class [Smi95a]

The operator *Specify Subclass* can be similarly modified (see Smith [Smi95a]). Smith continues by saying that the reduction of the task *Refine CLASS* and its subsequent subtasks would require a set of development operators reflecting the refinement methods suitable for the specification language being used.

Such operators, similarly as for specification, are grouped into three categories:

- those that apply a particular refinement strategy
- those that identify components of the product to be refined
- those that correspond to the task of refinement itself.

These final operators produce the refined version of the specification, and introduce subtasks corresponding to the necessary refinement proof obligations.

6.3.2 Refining modules

For the enabling of refinement of modules in isolation, classes first identified within a module must not be used outside the module. A module contains information about the logical class or classes it implements, thus creating a mapping from the logical view to the modular view [Eri98]. Allowing modules to be refined, in addition to classes, results in a more flexible and practical notion of refinement [Smi95a]. Subtasks can be added to simplify the identification of modules and aid in their refinement. The operators that are concerned with the task of refinement are identical to those for class refinement, i.e. they would simply apply refinement techniques suitable for the specification language being used. The operators for applying particular refinement strategies would apply strategies relevant to groups of classes. Such strategies are:

- distributing an invariant between classes
- the addition and deletion of classes to restructure modules
- sharing objects between classes, when refining from a specification language where objects values are present within classes to an implementation language where only pointers or object identities are present.

6.3.3 Functional Refinement

Jalote [Jal89] claims that functional refinement for object-oriented design is an iterative process, with the input to the first refinement coming from the initial design. For each of the operations marked for functional refinement in the last refinement, an informal strategy is written, the

objects are identified, and the operations to be further refined in the next refinement. When no operations for further refinement are identified, this refinement process stops.

As the functional refinement proceeds, new objects and operations on these will be identified [Jal89]. During a refinement step, operations may be identified on objects that were identified in earlier refinement steps.

At the termination of the refinement process, the transformation function needed to solve the problem would have been decomposed to a required level, with each operation in the algorithm being an operation on some object. Also, all the objects in the problem space will be identified. More objects and their functions may be uncovered when object-refinement takes place in the next phase. The objects identified in the next phase, would be exclusively for implementing the objects identified so far, and need not be visible outside. The set of all the objects identified when the functional refinement process terminates are called the *Problem Space Object Set (PSOS)* [Jal89].

6.3.4 Object refinement

When an object is refined, the informal strategy of all the operations defined so far are written. The new objects that are required to implement these operations are then identified, and the operations on these objects. New operations on the objects that have been previously identified (other members of PSOS) are identified. This process is repeated for each object, and is called *object refinement* [Jak89]

6.4 Modelling with Object-Z

Z used in the conventional style can be used to specify objects, but it does not 'support objects as a language feature', and objects are not 'first-class values' [Ste92]. Therefore, it cannot truly be called even object-based by Wegner's definition [Weg87a, Weg87b]. It can be said that Z is approximately object-based.

However, Object-Z is fully object-oriented; it has classes and inheritance. Object-Z extends Z by introducing a class construct, which encapsulates state and operation schemas. Classes, and hence state and operations, can be inherited by other classes [Ste92, Raf93, Fuk94, Der01].

Object-Z extends the graphical component of Z, the boxes, to define its classes. This gives an immediate visual indication of the scope of the definitions, as contrasted with the need to search for keywords such as 'begin' and 'endclass' in textual-based variants. This makes it easier to navigate specifications. Fully object-oriented Object-Z specifications are eminently readable.

6.4.1 Overview of Object-Z

Object-Z was proposed by researchers of the University of Queensland as an extension of Z [Ste92a]. Object-Z augments the class concept as a structuring facility. Refer to figure 6.2. The structure of an Object-Z class has as components constants, types, state schemas and operation schemas of Z. (Multiple) class inheritance is available by specifying base classes [Ste92, Raf93, Fuk94, Smi95a, Der01].

Object-Z embraces object-oriented concepts such as object, class, inheritance and composition. In order to accomplish the schema type class is introduced. Each Object-Z class definition as illustrated in Figure 6.2 consists of class constants characterising fixed object attributes, inherited classes, a state schema with predicates (invariants), and initialisation schema which defines the initial state of each object declared, and operation schemas which correspond to the methods for each object [Ste92, Raf93, Fuk94, Der01].

There are differences between the forms of schemas in a class of Object-Z and Z:

- A state schema in an Object-Z class does not have a schema name.
- In the Δ notation in operation schemas each state to be modified is specified in Object-Z, while in Z the state schema name is specified in a Δ list. Operation schemas indicate the change of values of states by the operation [Fuk94].

Inheritance in Object-Z is incremental. The subclasses have access to all contents of the super classes. No provision has been made for operations to be defined private to a class (used for internal manipulations), to be confined to objects of that class [Ste92, Raf93, Fuk94, Der01].

Object-Z does not specifically distinguish between attributes of objects (fixed) and their state variables. Object-Z accesses variables directly. This operator ignores the defined interface and breaks down encapsulation of data [Raf93].

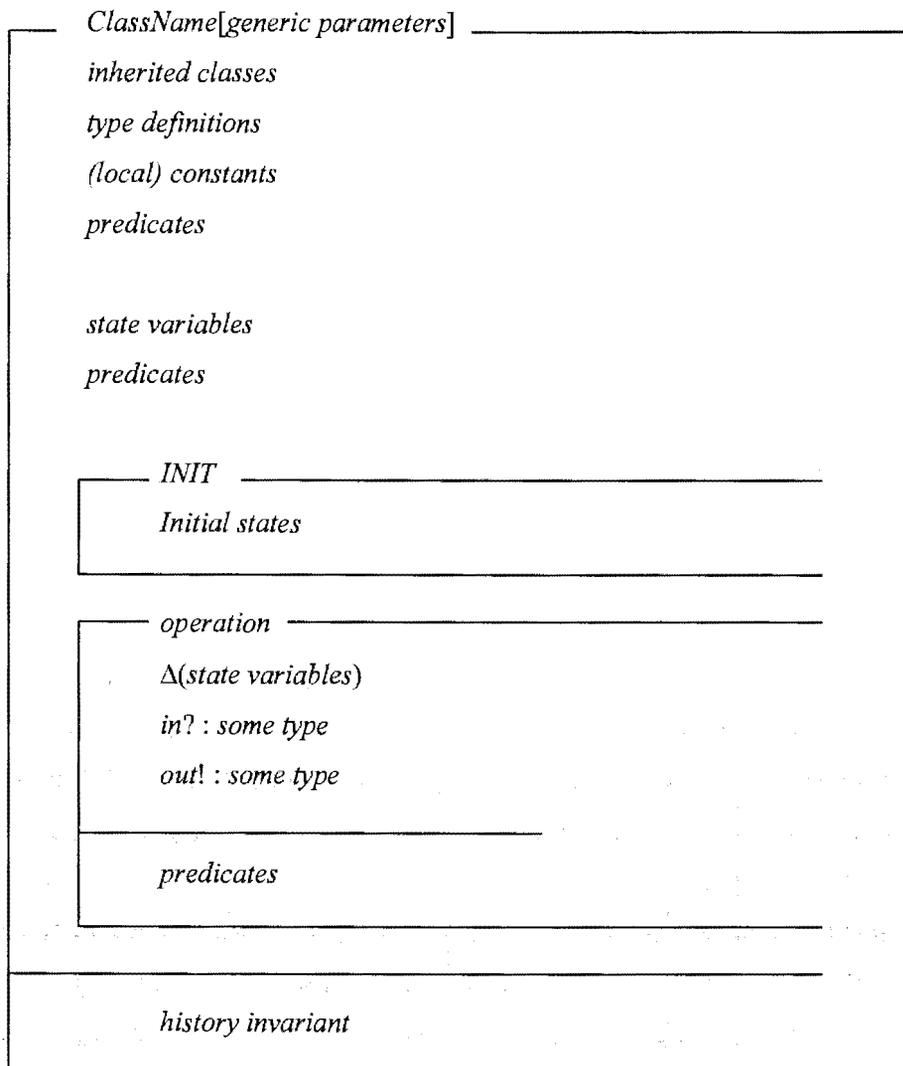


Figure 6.2. Object-Z specification [Fuk94 p.221]

In Object-Z a class is defined by [Fuk94, Ste92, Der01]:

- *Inherited classes*: The names of the superclasses to be inherited.

A subclass incorporates all the features of its superclasses, including their constants, state and operations. Operations and variables may be renamed in this list.

According to [Ste92]:

- *Local constants*: Cannot be changed by any operations, but different instances (objects) can have different values of the constants.
- The unnamed state schema declares state variables and a state invariant that constrains the constants and variables. Together these give the class attributes.

- *Initial state schema*: Defined in a way similar to plain Z; unprimed variable names are used.
- *Operation schemas*: Define the operations in a way very similar to plain Z, defining a relation between the before and after state.
- *A Δ list*: Lists those variables that may be changed by the operation; the other variables remain unchanged.

The declaration and predicate of any inherited operation with the same name are implicitly conjoined with this definition.

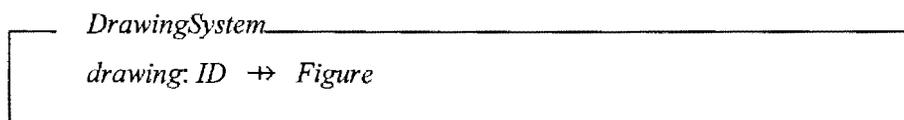
Since class attributes can be objects, it is often necessary to apply their operations on them:

- *Obj.Op*: is an object of the same class as *obj* resulting from performing the operation *Op* on *obj*.
- The history invariant enables constraints to be included in the allowable order of operations, using notation from temporal logic [Ste92, Smi95a].

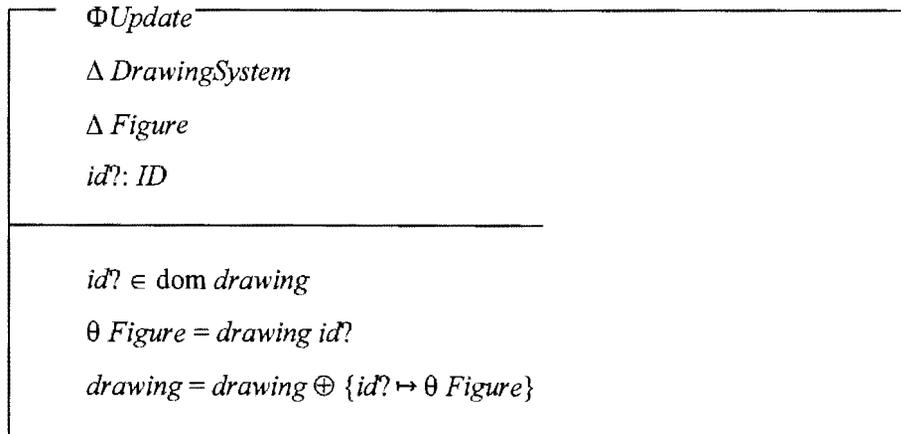
6.4.2 Moving from conventional Z to Object-Z

To illustrate the difference between the specification of a system in Z and Object-Z the following drawing system, as given by Stepney [Ste92] pages 151-157 is used as an example.

A drawing system consists out of a number of figures. e.g. quadrilateral (a figure that has non-zero edges), parallelogram, rhombus, rectangle and square. Different operations are performed on the figures, e.g. moving the figure, querying the angle between two edges, shearing the figure. The state of the drawing system consists of a mapping from identifiers to figures.



A general updating schema that performs an as yet unspecified change to a particular figure in the drawing system is defined.



The Greek letter Φ indicates that a schema is a framing schema. A framing schema introduces the before and after versions of the larger state [Pot96].

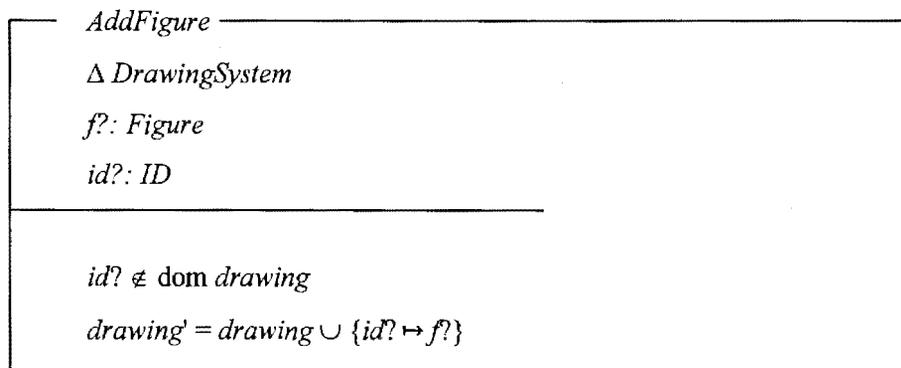
The operations on an individual figure can be promoted to operations on a figure in the drawing system, by conjoining them with the general updating schema and hiding the *Figure* [Ste92] as follows:

$$MoveDS \hat{=} (\Phi Update \wedge Move) \Delta Figure$$

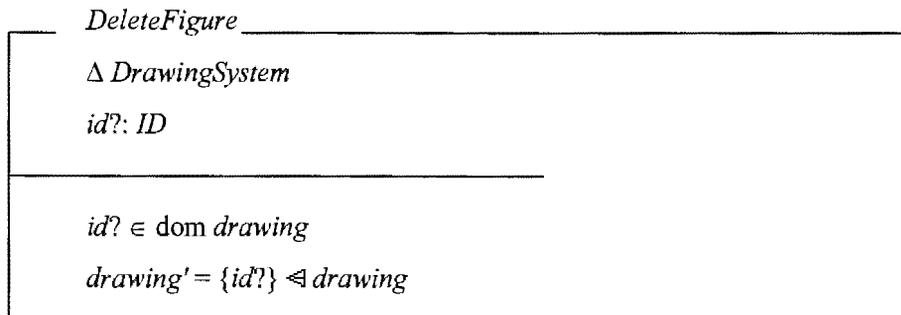
$$AngleDS \hat{=} (\Phi Update \wedge Angle) \Delta Figure$$

$$ShearDS \hat{=} (\Phi Update \wedge Shear) \Delta Figure$$

A new figure is added to the drawing system by *AddFigure*



The operation *DeleteFigure* deletes an existing figure from the state *DrawingSystem*.



The operation *AddFigure* adds a figure to the *DrawingSystem*. The two promotion schemas Φ *Update* and Φ *Lookup* are used to select a particular figure for updating or for interrogating, and *MoveDS*, *AngleDS* and *ShearDS* define the promoted operations. These operations are defined in terms of *Quadrilaterals*, but can also be applied to instances of subclasses.

In Object-Z the drawing system is represented in Figure 6.3. This system implements and illustrates the differences between Z and Object-Z described in Section 6.4.

6.5 Overview of C++

C++ was designed by Bjourne Stroustrup of AT&T Bell Laboratories. C++ is largely a superset of C. C++ is in a sense a better C by providing type checking, overloaded functions, and many other improvements. C++ adds object-oriented programming features to C [Boo94]. C++ corrects many of the deficiencies of C, and adds to the language support for classes, type checking, overloading, free store management, constant types, references, inline functions, derived classes, and virtual functions [Gor89].

C++ has been described as 'C plus classes' because C++ compilers compile C. The **class** construct in C++ is an extension of the C *struct* to include possible hiding of the data stored in class instantiations (i.e. objects), and a record of the operations that can be validly invoked on that data (the member functions). C++ does not distinguish between attributes and states of objects [Raf93, Fuk94, Ram98, Zak01].

C++ supports inheritance. There are three parts in the members of a class of C++ concerning accessibility to these from outside: *private*, *protected*, and *public*. The accessibility in a derived class can be controlled when inheriting the members of a base class, using the same keywords *public*, *protected* and *private* [Raf93, Fuk94, Ram98, Zak01].

```
    char dept[20];  
public:           //Interface section  
    Employee (char *n)  
    void show (void)  
    ~Employee()  
};
```

The class is separated into three sections, called private, public and protected. The private members are not freely accessible to any function. Only the functions in the public sections are allowed to access them. The public members can be accessed from any part of the program. The public section provides the interface to the class, while private members are the hidden implementation. The protected member *dept* implies that any member function in a class derived from it may access *dept*. For the functions of the derived classes, *dept* acts as if it is public. For anyone else, *dept* acts as if it is private. This means that other classes not derived from Employee may not access *dept* [Ram98].

A pointer to an object of a derived class can be coerced to that pointing to the object of a base class. Similarly, a pointer to an object of a base class can be coerced to that pointing to the object of a derived class. If a member function of a base class is declared to be *virtual*, when this function is called, the actual member function in a derived class is selected according to the class to which the object belongs. In this way polymorphism is realised. Multiple inheritance is realised by declaring a base class to be a *virtual* base class [Raf93, Fuk94, Ram98, Zak01].

6.6 Classes: Object-Z and UML compared

6.6.1 Specifications in UML

As an example refer to the Medicine_Catalogue class in Figure 6.4 below. The name and expiry dates of the medicines are given, and the operations are the adding of an expiry date, the finding of an expiry date and the reminding of an expiry date.

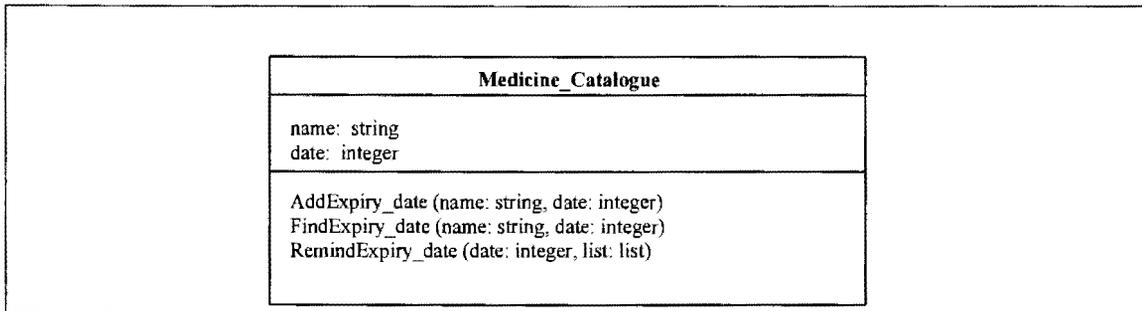


Figure 6.4 Medicine_Catalogue class

A class diagram describes the static view of a system in terms of classes and relationships among the classes [Eri98, Fow97, Rum91]. A class in a class diagram can be directly implemented in an object-oriented programming language that has direct support for the class construct. A class diagram box contains the name compartment, the attribute compartment, and the operation compartment. The syntax used for the compartments is independent of programming languages.

6.6.2 Specifications in Object-Z

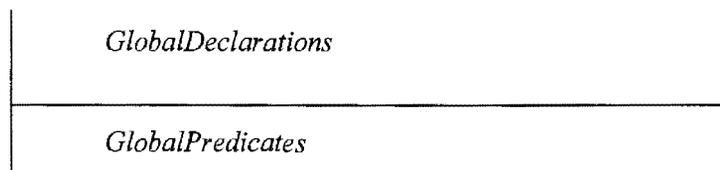
A Z state schema together with the operation schemas on that state define a class [Jak97]. The state variables in the state schema are the attributes or instance variables of that class. The operation schemas are the methods of that class [Jak97, Raf92, Ste92, Fuk94]. A class definition is a named box where the constituents of the class are defined and related. Refer to Figure 6.5.

6.6.3 Verification and Refinement

6.6.3.1 Verification (Object-Z)

Verifying consistency of global definitions:

For the axiomatic description



it must be established that there exists values for *GlobalDeclarations* that satisfy *GlobalPredicates*. For example from the type definitions, the state variables can be defined as follows:

$known: \mathbb{P} \text{NAME}$	(declaration of constant)
$known = \text{ABPainpills}$	(value of constant)

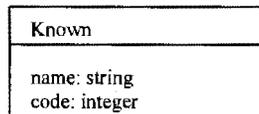
This description is consistent because it does not contradict *known*'s declaration. The same verification can be performed on the rest of the data items of the Z specifications.

6.6.3.2 Refinement (Object-Z)

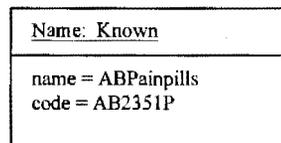
The above mentioned verification can also serve as the data refinement where it must be determined whether every abstract state has at least one concrete representative.

6.6.3.3 Verification and Refinement (UML)

In UML, for example, the Known class can be represented by



and can be instantiated as follows [Eri98, Fow97]:



Therefore the abstract state has at least one concrete representative..(This can be applied to the other classes as well).

- For the verification and refinement both the UML and Object-Z have concrete representations for the abstract states and provide an equal amount of information and clarity.

6.6.4 Implementation into C++

In implementing an object-oriented design, the first step is to declare object classes. Each attribute and operation in an object diagram must be declared as part of its corresponding class.

In C++ both attributes and methods are declared as *members* of a class.

These members are either public or private. Public members can be accessed by any function, private members can be accessed only by methods on the same class [Rum91, Raf93, Fuk94, ram98]. A method is not allowed to have the same name as an attribute. For the mapping of Object-Z to C++ class structures, class constants (object attributes) and state variables are mapped to protected class variables [Raf92].

Classes Power and Pfun (actually, templates) are prepared in C++ as a class library for realising the Object-Z operators related to power sets and partial functions, respectively [Fuk94]. Other related Z symbols and operators like *dom* and *#* are prepared in these classes.

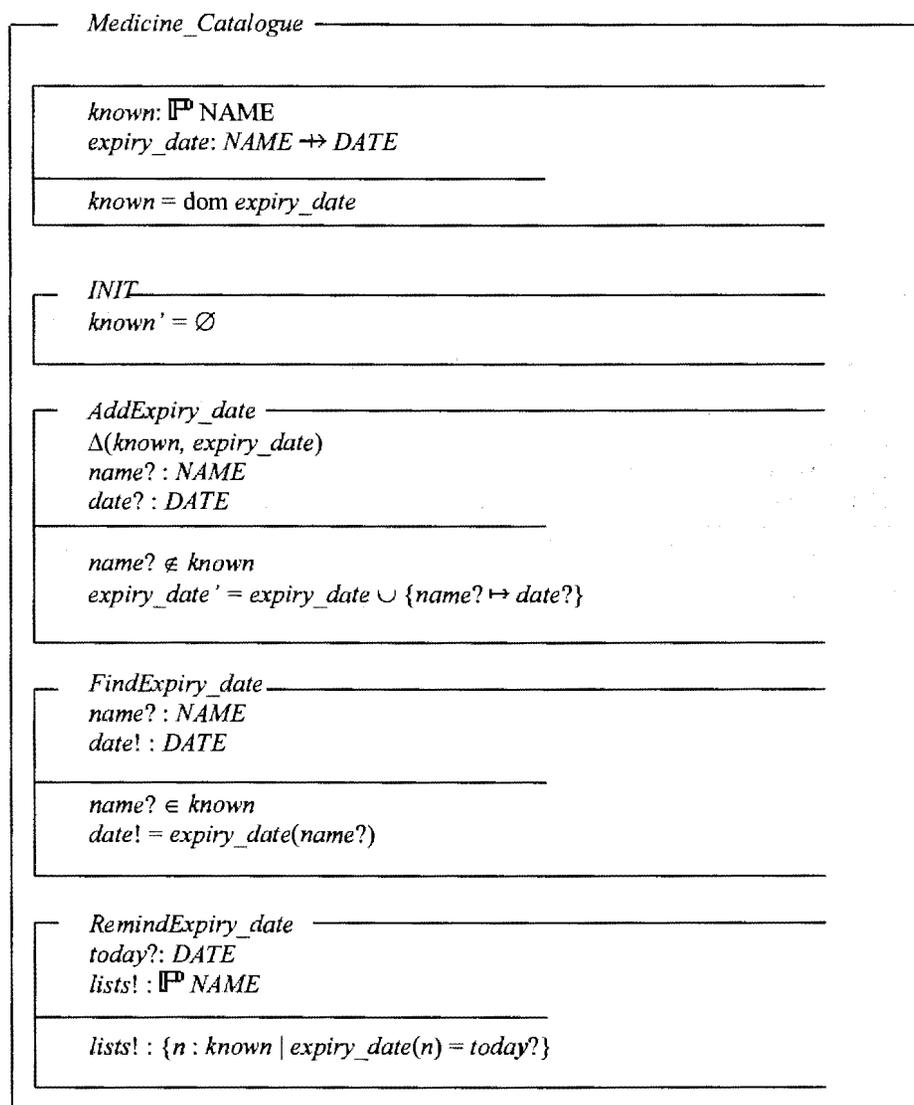


Figure 6.5 *Medicine_Catalogue* class in Object-Z

All member functions (corresponding to Object-Z operations) are declared as virtual functions in C++. The suffixes '_q' and '_x' of the parameters of the operations AddExpiry_date and the others correspond to the '?' and '!' of variables in operation schemas of Object-Z.

```
# include "GlobalDefs.h"
class MedicineCatalogue
{
protected:
//DeclPart
    Power< NAME > known;
    Pfun< NAME, DATE > expiry_date;
public:
    MedicineCatalogue( ); //Null Constructor
    MedicineCatalogue(MedicineCatalogue the_MedicineCatalogue); //Copy Constructor
    virtual ~MedicineCatalogue( ); //Destructor
    MedicineCatalogue
        operator = (MedicineCatalogue& the_MedicineCatalogue); //Assignment Operator

    virtual void
        AddExpiry_date(NAME& name_q, DATE& date_q);

    virtual void
        FindExpiry_date(NAME& name_q, DATE& date_x);

    virtual void
        Remind(DATE& today_q, Power< NAME >& lists_x);
};
```

6.6.5 Comparison

6.6.5.1 Specification

- For both Object-Z and UML, the class definition becomes a C++ class definition.
 - Although there are differences between the specifications in Object-Z and UML, there are also similarities:
-

- Constants and state variables are collectively called *attributes*.
- For Object-Z the *state invariant* is a statement about what is always true of the state before and after every valid operation on the state. This is made up of predicates which are described both explicitly (under the middle line of a schema) and implicitly through type declarations [Ran94].
- For UML a *class invariant* is a condition that a class must satisfy at all stable times.
- The state variables and state invariants in the state schema of Object-Z are the equivalent of the attributes defined in UML, which in the first example are declared in the C++ class `Drawing` as *private*.
- For Object-Z the attributes and class invariant are implicitly included in the initial state schema and each operation schema. UML does not have an initial class diagram.
- The class definition of Object-Z contains a history invariant that is a predicate over histories of objects of the class (typically in temporal logic) that further constrains the possible behaviours of such objects [Ste92]. UML does not have a way of visually identifying time-related classes.

6.6.5.2 Verification and Refinement

For a comparison between Object-Z and UML for the verification and refinement, refer to Section 6.6.3.

6.6.5.3 Implementation

- The `Medicine_Catalogue` class and accompanying operation definitions in UML and the `Medicine_Catalogue` class in Object-Z with the operation schema definitions are in both cases implemented into the class `MedicineCatalogue` in C++ (refer to Section 6.6.3).

6.6.5.4 Conclusion

- In this example for the implementation of classes in UML and Object-Z into C++ 2 routes are followed that are different, but equally clear and concise.
 - The main differences in the specifications are that for Object-Z the attributes and class invariant are implicitly included in the initial state schema and each operation schema. UML does not have an initial class diagram. Also the class definition of Object-Z contains a history invariant that is a predicate over histories of objects of the class (typically in temporal
-

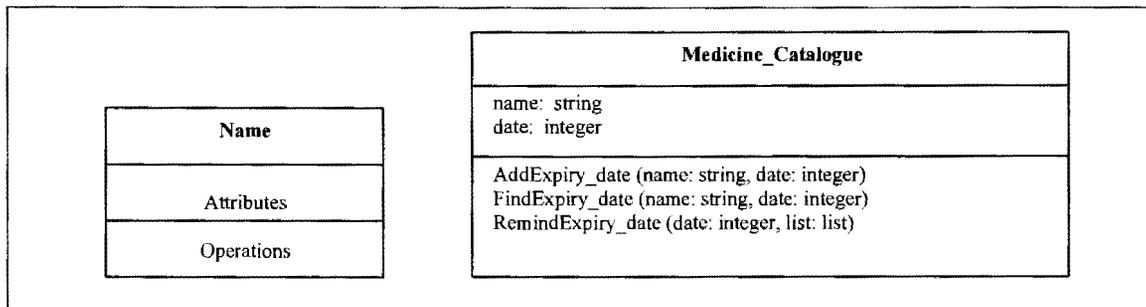


Figure 6.6 UML Class with operations [Eri98, p.73]

logic) that further constrains the possible behaviours of such objects [Ste92, Der01]. UML does not have a way of visually identifying time-related classes.

6.7 Operations: Object-Z and UML compared

6.7.1 Specifications in UML¹

A class in UML consists out of the class name, class attributes and class operations (Refer to Figure 6.6). Attributes are values that characterise objects of the class. Sometimes the values of the attributes are a way to describe the state of the object. Operations are used to manipulate the attributes or to perform other actions. Normally operations are called functions, but they are inside a class and can be applied only to objects of that class. An operation is described with a return-type, a name, and zero or more parameters [Rum91, Fow97, Ode98, Eri98].

(The class is further operation refined in figures 6.7.1, 6.7.2, and 6.7.3)

6.7.2 Specifications in Object-Z

Operations in Object-Z are specified as operation schemas, the same as for Z (refer to Chapter 4). Refer to Figure 6.5 for examples of operation schemas *AddExpiry_date*, *FindExpiry_date*, *RemindExpiry_date*.

¹ Not all UML diagrams are required in specifications, only the ones that are needed [Eri98, Fow97, Dor99].

6.7.3 Verification and Refinement

6.7.3.1 Refinement (UML)

- The UML operations specified in the class can be further refined as follows (figures 6.7.1, 6.7.2, and 6.7.3). (Note that we consider the process depicted in the state diagram below as a valid *refinement*, although the UML community does not explicitly refer to this process as refinement).

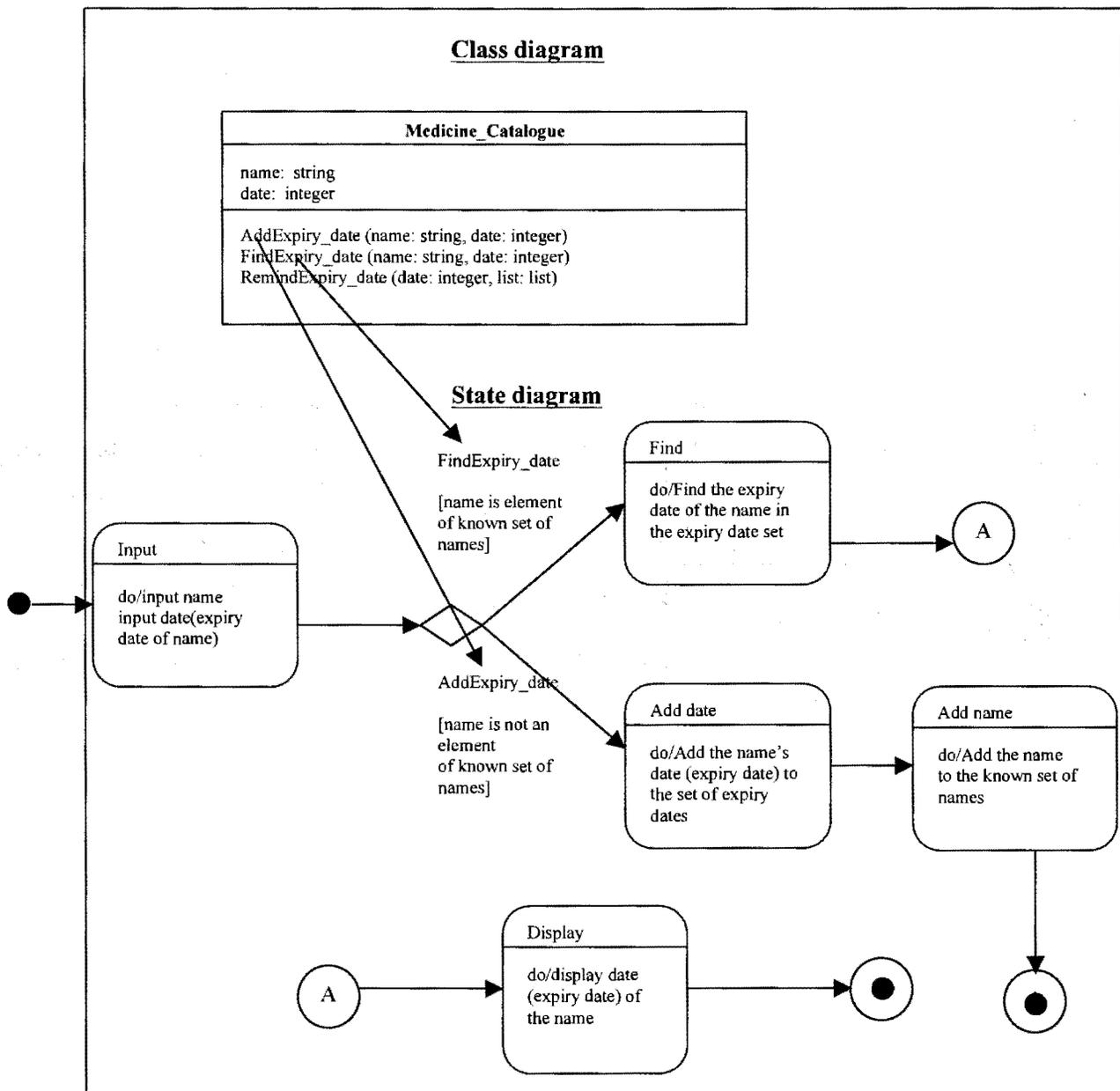


Figure 6.7.1 Class diagram with the corresponding State diagram

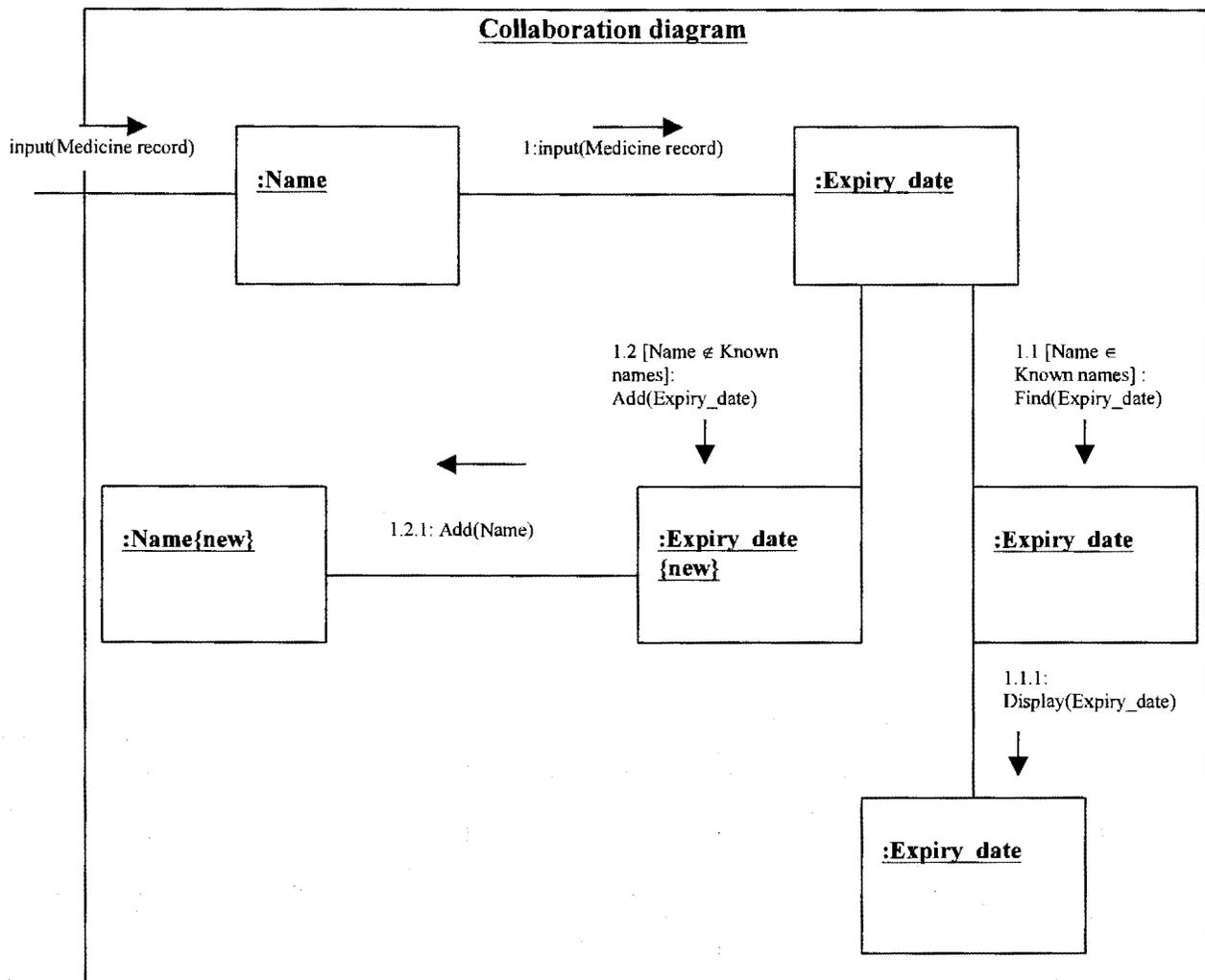


Figure 6.7.2 Collaboration diagram

6.7.3.2 Verification and Refinement (Object-Z)

The Object-Z operation specifications for the schema *AddExpiry_date* can be refined as follows:

Verifying consistency of global definitions

It must be established that $\vdash \exists \text{GlobalDeclarations} \bullet \text{GlobalPredicates}$ which means that there exist values for *GlobalDeclarations* which satisfy predicate *GlobalPredicates* (refer to Section 3.3.3).

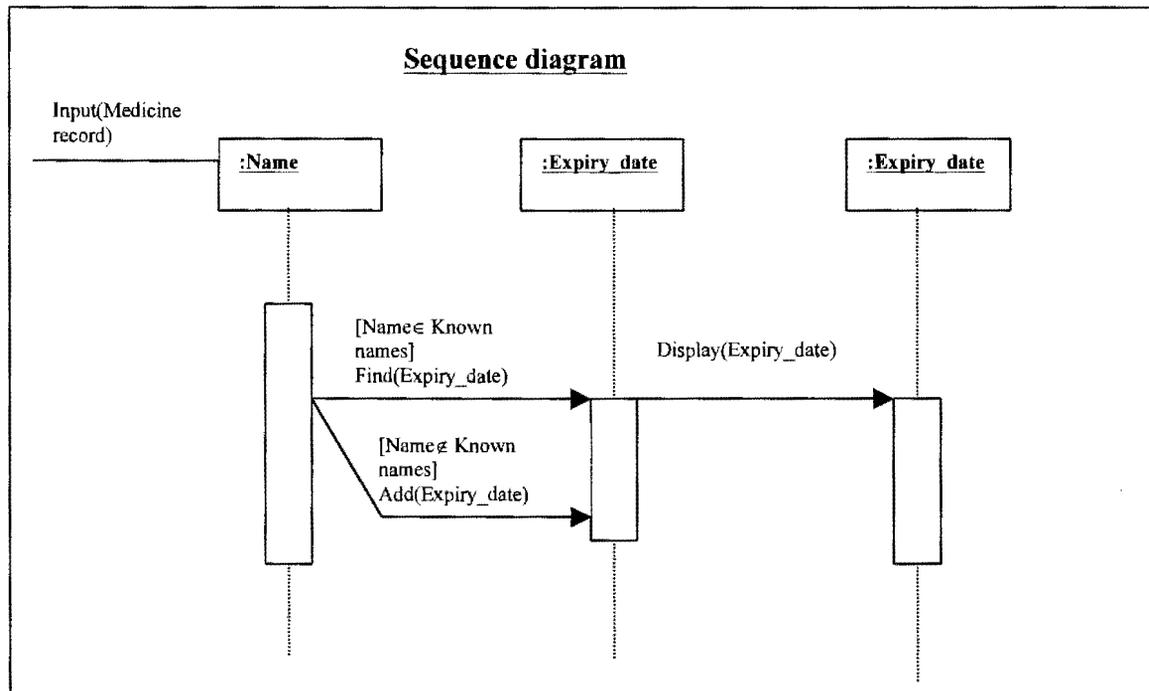
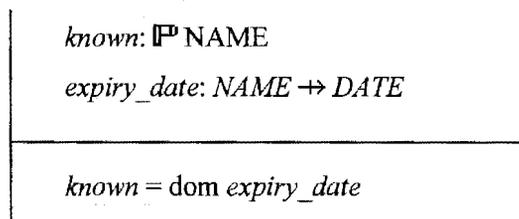


Figure 6.7.3 Sequence diagram



Suppose $known = \{Aaa, Bbb, Ccc, Ddd, Eee\}$

and

$expiry_date = \{(Aaa,030202), (Bbb,240502), (Ccc,151102), (Ddd,080702), (Eee,180902)\}$

Therefore

$$known = \text{dom } expiry_date.$$

Therefore

$$\vdash \exists known: \mathbb{P} \text{ NAME} \wedge expiry_date: \text{NAME} \leftrightarrow \text{DATE} \bullet true$$

By a simple property of logic

$$\vdash true$$

Verifying consistency of an initial state

A check must be done to ensure that a consistent initial state exists. This check can be expressed as the *initialisation theorem* which has the following general form:

$$\vdash \exists \text{State}' \bullet \text{InitStateSchema}$$

which can be extended to

$$\vdash \exists \text{State}'; \text{Inputs}' \bullet \text{InitStateSchema}$$

if there are input variables for the initial state schema *InitStateSchema*.

Refer to Figure 6.5. The concrete initial state is:

$$\boxed{\begin{array}{l} \text{INIT} \\ \text{known}' = \emptyset \end{array}}$$

To show that it is consistent:

$$\vdash \exists \text{known}': \mathbb{P} \text{NAME}; \text{expiry_date}': \text{NAME} \rightarrow \text{DATE} \bullet \text{known}' = \text{dom expiry_date}'$$

If $\text{dom expiry_date}' = \emptyset$ then

$$\text{known}' = \emptyset$$

which implies that there is a state *TYPEDDEF'* of the state definition schema that satisfies the initial state description.

Verifying consistency of operations

For an operation that is defined as: *OperationDeclarations* | *OperationPredicates*, the consistency theorem is

$$\vdash \exists \text{OperationDeclarations} \bullet \text{OperationPredicates}$$

Calculating its precondition can check an operation's consistency. If the operation is inconsistent, its precondition will be *false*. A false precondition strongly suggests a defect in the operation description [Rat94].

The consistency theorem for the operation *AddExpiry_date* (refer to *AddExpiry_date* schema below) will be:

$$\vdash \exists \text{known}': \mathbb{P} \text{NAME}; \text{expiry_date}': \text{NAME} \rightarrow \text{DATE}; \text{name}' : \text{NAME}; \text{date}' : \text{DATE} \bullet \text{name}' \notin \text{known}' \Rightarrow \text{expiry_date}' = \text{expiry_date} \cup \{\text{name}' \mapsto \text{date}'\}$$

Now assume that $\text{name}' \notin \text{known}' \Rightarrow \text{false}$

then $\text{name}' \notin \text{known}' \Rightarrow \text{expiry_date}' \neq \text{expiry_date} \cup \{\text{name}' \mapsto \text{date}'\}$

But according to the verified initial state schema, $\text{known} = \text{dom expiry_date}$

therefore if $name? \notin known \Rightarrow expiry_date' = expiry_date \cup \{name? \mapsto date?\}$

Then according to a simple property of logic

$\vdash \exists known': \mathbb{P} NAME; expiry_date': NAME \mapsto DATE; name? : NAME; date? : DATE \bullet false$

and then according to a simple property of logic

$\vdash false$

This means that the assumption was *false*, and that the sequent predicate is not a contradiction, hence that *AddExpiry_date* is consistent.

Data refinement

Consider the abstract state in our *MedicineCatalogue* above. This state is a set *known* of elements of type *NAME*, and a set *expiry_date* of elements of type *DATE*:

$known: \mathbb{P} NAME$ $expiry_date: NAME \mapsto DATE$
$known = \text{dom } expiry_date$

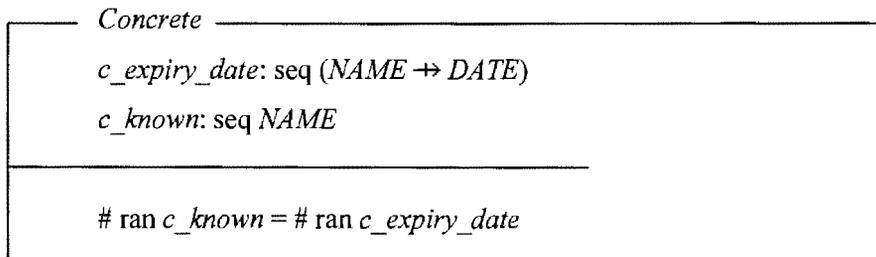
An abstract operation *AddExpiry_date* is defined that stores an element in the set using the set union operator:

$AddExpiry_date$ $\Delta(known, expiry_date)$ $name? : NAME$ $date? : DATE$
$name? \notin known$ $expiry_date' = expiry_date \cup \{name? \mapsto date?\}$

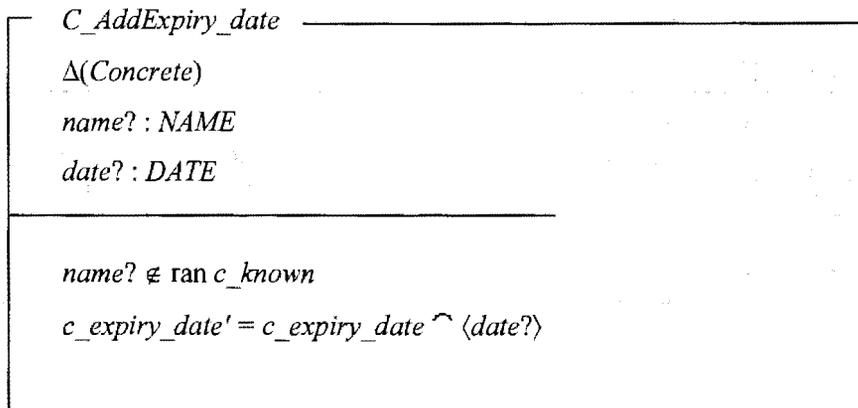
We plan to implement this system into a programming language that support arrays and lists. We decide to refine the abstract specification to a detailed design based on sequences because we expect this will be easier to map into the target programming language. The reason for this is

because sequences are sorted lists and we are going to use sorted arrays in the programs to represent the sequences.

For the refinement the concrete representation of *expiry_date* is not a set but a sequence c_expiry_date of elements of type *DATE*. The concrete representative of *known* is not a set but a sequence c_known .



Here is the concrete operation that adds an element to the sequence (if the corresponding medicine name is not in the medicine array) using the concatenation operator:



The sequence should always hold the same elements as the set. A sequence is a function from natural numbers to elements, so the elements stored in the sequence are the range of this function.

The range of the sequence must be the same as the set.

$$expiry_date = \text{ran } c_expiry_date \wedge$$

$$expiry_date' = \text{ran } c_expiry_date'$$

also

$$known = \text{ran } c_known \wedge$$

$$known' = \text{ran } c_known'$$

therefore

$$\begin{aligned} & name? \notin known \\ \Leftrightarrow & name? \notin \text{ran } c_known \end{aligned}$$

This must be true before and after any operation, so equations appear for unprimed and primed (') variables.

We can now form the implication that expresses the refinement. The predicate of the abstract operation *AddExpiry_date* appears on the right of the implication arrow, and the predicate of the concrete operation *c_AddExpiry_date* is on the left, along with the equations relating *expiry_date* and *c_expiry_date*

$$\begin{aligned} & \text{When } name? \notin \text{ran } c_known \\ \text{then} \\ & \quad c_expiry_date' = c_expiry_date \hat{\ } \langle date? \rangle \wedge \\ & \quad expiry_date = \text{ran } c_expiry_date \wedge \\ & \quad expiry_date' = \text{ran } c_expiry_date' \Rightarrow \\ & \quad expiry_date' = expiry_date \cup \{name? \mapsto date?\} \end{aligned}$$

Proof of the refinement:

$$\begin{aligned} & \text{When } name? \notin \text{ran } c_known \\ \text{then} \\ & \quad c_expiry_date' = c_expiry_date \hat{\ } \langle date? \rangle \wedge \\ & \quad expiry_date = \text{ran } c_expiry_date \wedge \\ & \quad expiry_date' = \text{ran } c_expiry_date' \Rightarrow \\ & \quad expiry_date' = expiry_date \cup \{name? \mapsto date?\} \quad \text{[Given]} \\ \Leftrightarrow & \quad expiry_date' = expiry_date \cup \{name? \mapsto date?\} \quad \text{[Assume antecedent]} \\ \Leftrightarrow & \quad \text{ran } c_expiry_date' = expiry_date \cup \{name? \mapsto date?\} \quad \text{[Antecedent } expiry_date' = \\ & \quad \text{ran } c_expiry_date]} \\ \Leftrightarrow & \quad \text{ran}(c_expiry_date \hat{\ } \langle date? \rangle) = expiry_date \cup \{name? \mapsto date?\} \quad \text{[Antecedent } c_expiry_date' \\ & \quad = c_expiry_date \hat{\ } \langle date? \rangle]} \\ \Leftrightarrow & \quad \text{ran } c_expiry_date \cup \text{ran} \langle date? \rangle = expiry_date \cup \{name? \mapsto date?\} \quad \text{[Law about } \text{ran}(s \hat{\ } t) = \\ & \quad (\text{ran } s) \cup (\text{ran } t)] \\ \Leftrightarrow & \quad \text{ran } c_expiry_date \cup \text{ran} \{name? \mapsto date?\} = expiry_date \cup \{name? \mapsto date?\} \quad \text{[Law about} \\ & \quad \text{ran} \langle x? \rangle = \{x?\}] \\ \Leftrightarrow & \quad expiry_date \cup \{name? \mapsto date?\} = expiry_date \cup \{name? \mapsto date?\} \quad \text{[Antecedent} \end{aligned}$$

$expiry_date = \text{ran } c_expiry_date]$

$\Leftrightarrow true$ $[e = e \Leftrightarrow true]$ $[e = \text{an expression}]$

This concludes the proof of correctness for this refinement.

Verifying the correctness of the concrete initial state

The concrete initial state must not describe initial states that have no counterpart in the abstract model [Rat94, Pot96, Der01]. A theorem of the following form is to be proved:

Given the retrieve relation then:

$InitConcState \vdash InitAbsState$

which says that 'for each concrete initial state, there is a corresponding abstract one'.

Refer to the following schema definitions:

<i>INIT</i>
<i>TYPEDEF'</i>
$known' = \emptyset$

and

<i>Concrete</i>
$c_expiry_date: \text{seq } (NAME \rightarrow DATE)$
$c_known: \text{seq } NAME$
$\# \text{ran } c_known = \# \text{ran } c_expiry_date$

and

<i>Concrete'</i>
$c_expiry_date': \text{seq } (NAME \rightarrow DATE)$
$c_known': \text{seq } NAME$
$\# \text{ran } c_known' = \# \text{ran } c_expiry_date''$

and

C_INIT <i>Concrete'</i> $c_known' = \langle \rangle$
--

and

<i>TYPEDEF</i> $known: \mathbb{P} NAME$ $expiry_date: NAME \rightarrow DATE$
$known = \text{dom } expiry_date'$

and

$CARel$ <i>TYPEDEF</i> <i>Concrete</i>
$known = \text{ran } c_known$

and

$CARel'$ <i>TYPEDEF</i> <i>Concrete'</i>
$known = \text{ran } c_known$

It must be proved that there is a state $CARel'$ of the general model $CARel$ (concrete to abstract relation) that satisfies the following:

$$INIT \vdash C_INIT$$

The declarative part of the right-hand side schema text is just $TYPEDEF'$ which is provided by $CARel'$ on the left. The sequent is then unfolded into

$$CARel'; Concrete' \mid$$

$$c_known' = \langle \rangle \vdash known' = \emptyset$$

which holds because $c_known' = \langle \rangle$ on the left and $known' = \text{ran } c_known'$ in $CARel'$.

By substitution $known' = \text{ran } \langle \rangle$, and $known' = \emptyset$ immediately follows.

The concrete state must be consistent

It has to be shown in general that

$$\vdash \exists ConcState' \bullet InitConcState \quad (InitConcState \text{ represents the initial concrete state})$$

or for our example:

$$\vdash \exists Concrete1' \bullet C_INIT$$

$Concrete1'$ is a state of the general model $Concrete1$.

From the *data refinement* it is concluded that the state sets $known: \mathbb{P} NAME$ and $expiry_date: NAME \leftrightarrow DATE$ are implemented as arrays with an index variable:

$c_known: \text{array } [1..maxSize]$ and
 $c_expiry_date: \text{array}[1..maxSize]$ respectively and
 $n: 0..maxSize$

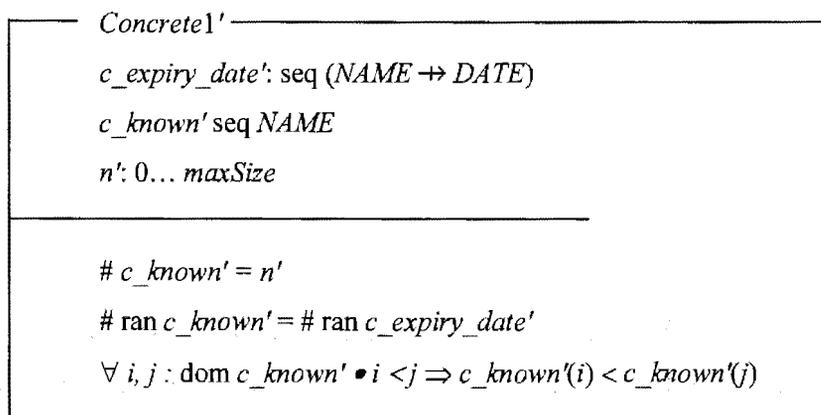
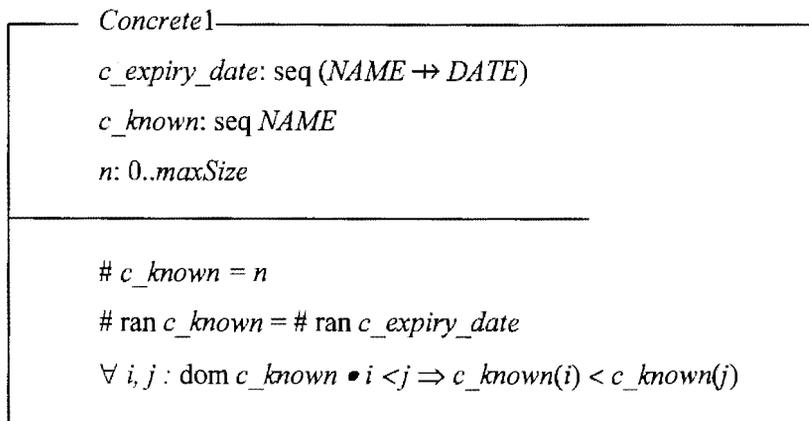
It is assumed that the n elements of the c_known array are sorted in ascending sequence to ensure that no duplicates are kept in the array and to facilitate fast lookup of the array. For all the n elements of the c_known array, the corresponding element in the c_expiry_date array represents the expiry date of the c_known medicine. For example, $c_expiry_date(5)$ is the expiry date of the medicine represented by $c_known(5)$.

Add the following to *TYPEDEF*:

$|maxSize: \mathbb{N}$

<i>TYPEDEF1</i>
$known: \mathbb{P} NAME$ $expiry_date: NAME \leftrightarrow DATE$
$known = \text{dom } expiry_date'$ $\# known \leq maxSize$

Add the following to *Concrete*, and *Concrete'*:



To show that it is consistent:

Refer to *Concrete1'*:

$\vdash \exists c_known' : \text{seq } NAME$

$n' : 0... maxSize$

$\forall i, j : \text{dom } c_known' \bullet i < j \Rightarrow$

$c_known'(0) < c_known'(1) < c_known'(2), < \dots < c_known'(maxSize)$

$c_known' = n'$

If $n' = 0$ then

$c_known' = \langle \rangle$

which implies that there is a state (*Concrete1*) of the general model *Concrete1* that satisfies the initial state description *C_INIT*.

Determine whether every abstract state has at least one concrete representative:

This can be achieved by determining if each abstract variable can be derived or 'retrieved' from the concrete variables by writing down equalities of the form:

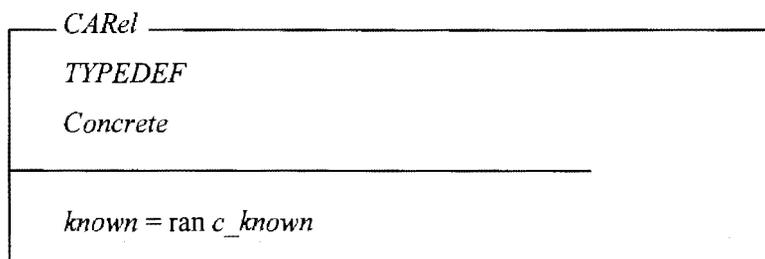
$$AbsVar = Expr(ConcVars)$$

where *AbsVar* represents an abstract variable of the abstract state, *Expr* the expression and *ConcVars* the concrete variable of the concrete state representing the abstract state.

For the example the predicate

$$known = \text{ran } c_known$$

will be referred to as the 'retrieve relation' *CARel* (concrete-to-abstract relation) that brings together the abstract and the concrete states:



The equality means that *CARel* is in effect a total function when viewed as 'calculating' the abstract state from the concrete one [Rat94]. Being total means that every concrete state maps to some abstract state. This implicit property of the retrieve relation being functional and total, characterises the fact that a simplified form of data refinement is discussed.

Suppose, however, the 'sorted' invariant was removed from *NatNumArray* so that the array element order was immaterial. Assume that no duplicates are stored in the array. The design will now include some redundancy in that each non-empty, non-singleton set in the abstract state would have more than one concrete representation. [Rat94].

For example, the abstract state

$$\langle known \Rightarrow \{ Aspirin, Brufen \} \rangle$$

would have two concrete representatives:

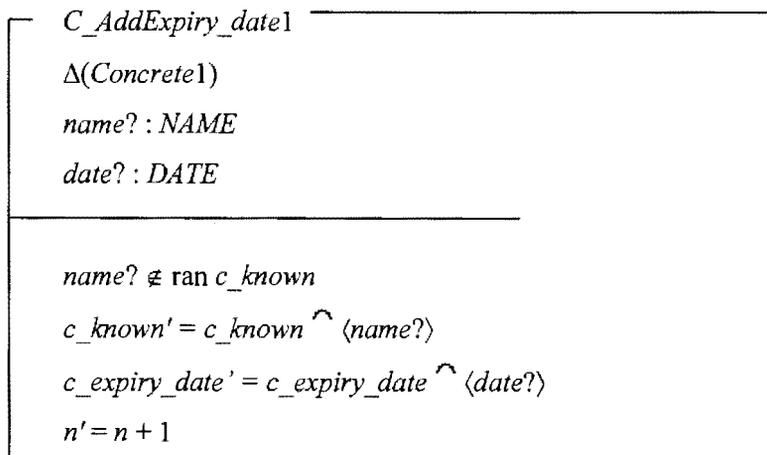
$$\langle c_known \Rightarrow \langle Aspirin, Brufen \rangle, n \Rightarrow 2 \rangle$$

$$\langle c_known \Rightarrow \langle Brufen, Aspirin \rangle, n \Rightarrow 2 \rangle$$

In general, assuming no duplicates, there would be $n!$ concrete representatives for a single abstract state. The implicit functionality of a retrieve relation such as *CARel* is not compromised because the relation expresses a calculation from *concrete to abstract* [Rat94].

Operation Refinement

Refer to *C_AddExpiry_date* from the *Data Refinement* with n' and n added:



This is a data-refined operation because the abstract *AddExpiry_date* operation is re-expressed as the concrete operation *C_AddExpiry_date1*. *C_AddExpiry_date1* is refined into *C_AddExpiry_date2*:

The first two lines of the predicate (of the schema *C_AddExpiry_date2*) state that if *name?* is in the array, nothing changes. The last four lines state that if *name?* is not in the array, the name gets added to the *c_known* array and its corresponding expiry date gets added to the *c_expiry_date* array. This is achieved by the insertion of the new name to the existing name array and the insertion of the new existing expiry date to the existing expiry date array.

C_AddExpiry_date2 can be refined into the following algorithm:

<i>[lookFor(name?, c_known, i, found)]</i>	[Check if <i>name?</i> is in the array <i>c_known</i>]
if found then skip	[If found (in the array) go to endif]
elseif \negfound then	[If not found (not in the array) then

<i>C_AddExpiry_date2</i>	
$\Delta(\text{Concrete1})$	
<i>name?</i> : NAME	
<i>date?</i> : DATE	
$name? \in \text{ran } c_known$	
$\theta \text{Concrete1}' = \theta \text{Concrete1}$	
$name? \notin \text{ran } c_known \Rightarrow$	
$(\exists i : 1..n+1 \bullet$	
$c_expiry_date(i) = date? \wedge c_expiry_date' = ((1..i-1) \uparrow c_expiry_date) \hat{\wedge}$	
$c_expiry_date(i) \hat{\wedge} ((i..n) \uparrow c_expiry_date) \wedge$	
$c_known(i) = name? \wedge c_known' = ((1..i-1) \uparrow c_known) \hat{\wedge} c_known(i) \hat{\wedge}$	
$((i..n) \uparrow c_known) \wedge$	
$n' = n + 1$	
<i>[AddExpiryDate(c_known, i)]</i>	[The new medicine is added to the <i>c_known</i> array]
<i>[AddExpiryDate(c_expiry_date, i)]</i>	[The corresponding expiry date is added to the <i>c_expiry_date</i> array]
$n = n + 1$	[Derived from the predicate $n' = n + 1$]
endif	

The bracketed italicised parts indicate that the constructs involved are specification components which need to be subjected to further refinement to reach code.

6.7.4 Implementation into C++

In most OO (object-oriented) languages, each operation has at least one implicit argument, the target object, indicated with a special syntax [Rum91]. Operations can have additional arguments. Some languages permit a choice between passing arguments as read-only values or as references to values that can be updated by a procedure.

Object-Z and UML operations are directly implemented as C++ virtual member functions. The virtuality is necessary to support the extension of operations. The return values of the functions are of type void, and their parameters are passed by reference [Raf93, Fuk94].

The constraints that are inherited from super classes in Object-Z and UML are implemented in C++. If they are not implemented, there is no guarantee that the state invariant (also inherited) would be preserved as operations are invoked during the lifetime of the object. To compensate for the loss of direct variable access in Object-Z, standard C++ functions are provided to retrieve these values as required. These functions are not labeled as virtual as there is no intention to allow them to be redefined in derived classes. The keyword *const* indicates that these functions have no side-effects on the object's state.

A C++ operation is declared as a member of a class along with attributes [Rum91]. An operation is invoked using a similar notation to attribute access: the member selection operator '->' is applied to an object pointer.

One possible implementation for part of the virtual void

AddExpiry_date(NAME& name_q, DATE& date_q) function is:

```
virtual void
AddExpiry_date(NAME& name_q, DATE& date_q)
{
....
    i = 0;
    ...
    while (i <= n)
    {
        ....
        if name_q == name[i]
        {...execute virtual void
            FindExpiry_date(NAME& name_q, DATE& date_x) and return to
            main()...
        }
        else
            i = i + 1;
    }
}
```

```
}
while (i <=n)
{
    if name_q < name[i]
    {name[i] = name[i];}
    else
    temp = name[i];
    name[i] = name_q;
    temp2 = date[i];
    date[i] = date_q;
    name[i + 1] = temp;
    date[i + 1] = temp2;
}
n = n + 1;
.....
```

6.7.5 Comparison

6.7.5.1 Specification

- For both the UML and Object-Z, the implementation of an operation is called a *method*. An operation is specified with precondition, post condition, algorithm and the affect it has on an object, that is, the input, algorithm and the output that results. In Object-Z the input are indicated by the ? data types, and the output by the ! data types. In the UML diagrams 6.7.1, 6.7.2, and 6.7.3 the input and output from the operations are explicitly indicated in the figures.
- Object-Z operations are directly implemented as C++ virtual member functions [Raf93] e.g. virtual void AddExpiry_date(NAME& name_q, DATE& date_q) in 6.7.3. The UML operations are also implemented into the virtual member functions. The operations are more refined in figures 6.7.1, 6.7.2, and 6.7.3, which facilitate the implementation.
- The difference in the specifications are as indicated in the figures 6.6, 6.5, 6.7.1, 6.7.2, and 6.7.3, with both the pre- and post conditions (input and output) shown.

6.7.5.2 Verification and Refinement

A detailed comparison between Object-Z and UML is given in Section 6.7.3.

6.7.5.3 Implementation

As indicated in Section 6.7.4.

6.7.5.4 Conclusion

- It can be concluded that the Object-Z refinement is more comprehensive and complete, and includes a comprehensive preliminary verification. However, the eventual C++ code that emerges could also have been derived from the state diagram given in the UML specification.
- The concrete refined Object-Z statements could almost directly have been implemented into C++ code. However, the amount of time spent on the Object-Z refinement in comparison to the UML refinement neutralizes the advantages thereof.
- The UML specifications and refinements are, because they are more graphic, easier to read and comprehend.
- It can therefore be concluded that both UML and Object-Z have got their own advantages and disadvantages.

6.8 Using Inheritance

6.8.1 Specifications in UML

The same example as for the comparison between Z and UML is used, therefore from Figure 5.1: The relationship between a class and one or more refined versions of it is called *generalisation*. The class that is refined is called the *superclass* and each refined version is called a *subclass*. A subclass is called a *derived* class [Rum91, Fow97, Eri98, Ode98]. For example, in Figure 6.8 PERMANENT_SALESPERSON is a subclass of BASIC_SALESPERSON.

Attributes and operations common to a group of subclasses are attached to the superclass and shared by each subclass [Rum91, Eri98, Fow97, Ode98]. Each subclass then *inherits* the features of its super class.

In Figure 6.8 BASIC_SALESPERSON is the general (super) class derived to specific class (subclasses) via inheritance (generalisation – specialisation) [Eri98].

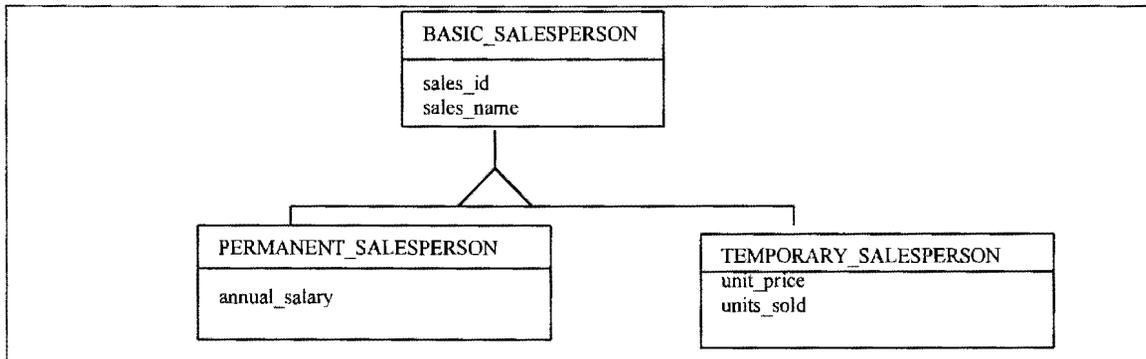


Figure 6.8 Inheritance in UML

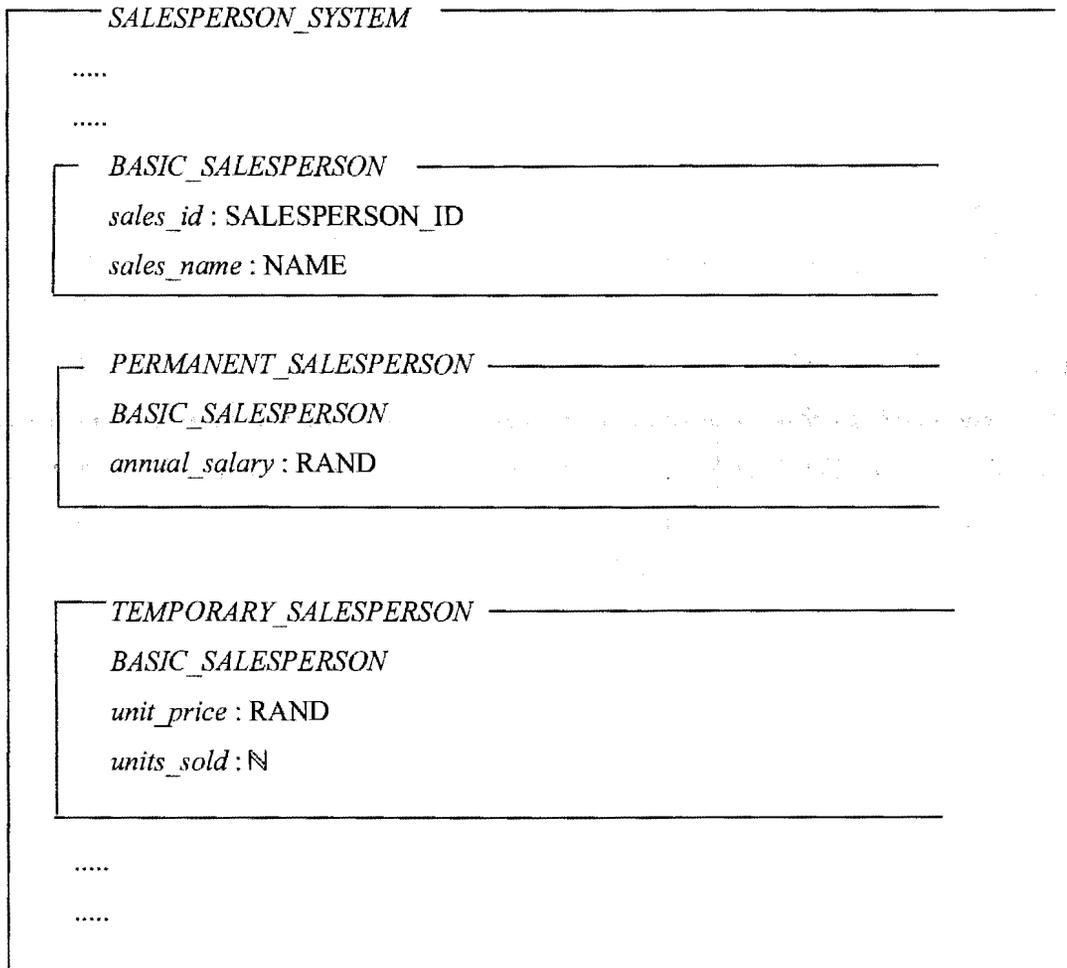


Figure 6.9 Inheritance in Object-Z

6.8.2 Specifications in Object-Z

Object-Z allows multiple inheritance. Classes may be related to each other by an *inheritance* relation. Once a class inherits another class, it is allowed to define operations affecting states as given in the inherited class. In multiple inheritance in Object-Z specifications, a common ancestor class as a base class must be determined [Raf93, Fuk94, Der01].

Refer to Figure 6.9. A salesperson is either a permanent salesperson or a temporary salesperson: Schema *PERMANENT_SALESPERSON* inherits from schema *BASIC_SALESPERSON*. Schema *TEMPORARY_SALESPERSON* inherits from schema *BASIC_SALESPERSON*. The dots indicate other possible schemas of the system that are not relevant to this discussion.

6.8.3 Verification and Refinement

6.8.3.1 Verification and Refinement (Object-Z)

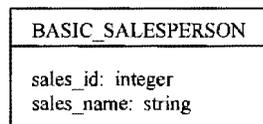
A number of consistency checks can be performed on the above (Figure 6.9) state space. For example, a specifier can show that there exists a basic salesperson who is also a permanent salesperson for certain values of the variables.

$$\vdash \exists BASIC_SALESPERSON' \bullet sales_id' \in SALESPERSON_ID \wedge sales_name' \in NAME \wedge sales_id = 45321 \wedge sales_name = 'Peter Scott' \wedge salary = 30000$$

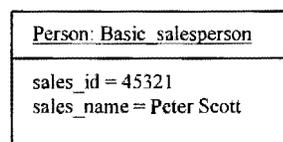
An analogous instantiation can be written down for a temporary salesperson. The above mentioned check can also serve as a data refinement where it must be determined whether every abstract state has at least one concrete representative.

6.8.3.2 Verification and Refinement (UML)

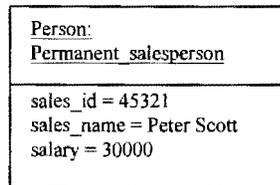
In UML, for example, the *BASIC_SALESPERSON* class can be represented by



and can be instantiated as follows [Eri98, Fow97]:



Similarly, a permanent salesperson can be instantiated by:



Therefore, the abstract state has at least one concrete representative.

6.8.4 Implementation into C++

From Object-Z and UML inheritance is mapped to public inheritance in C++. In the case of multiple inheritance the base class is labeled virtual.

Three independent dimensions for classifying inheritance mechanisms are

- static (inheritance is bound at compile time) or
- dynamic, implicit (behaviour of an object depends on its class, which cannot be changed) or
- explicit, per object or per group (inheritance characteristics are specified for a class, not for specific objects).

Many of the popular languages are static, per group and implicit. In most languages, the declaration of each class includes a list of super classes from which it inherits attributes and methods [Rum91, Fuk94, Ram98].

The superclass(es) of a class are specified as part of the class declaration. A subclass is called a *derived* class. The C++ code declares *Permanent_salesperson* to be a subclass of *Basic_salesperson*. In `main()`, *e1* is an object of class *Basic_salesperson* and *x1* is an object of class *Permanent_salesperson*. For the functions of the derived class (*Permanent_salesperson*), *dept* acts as if it is public. *p1* and *p2* are pointers to objects of class *Basic_salesperson* or to objects of classes derived from *Basic_salesperson*. *px* can only point to *x1*, an object of class *Permanent_salesperson*, it can't point to objects of class *Basic_salesperson*.

```
#include <iostream.h>
#include <string.h>
class Basic_salesperson
{
private:
```

```

    char sales_name[20];
protected:
    int sales_id;                //Accessible by base and derived class members.
public:
    Basic_salesperson(char *n) { strcpy(sales_name, n);}           // Ctor (Constructor)
    void show (void) { cout << "Name " << sales_name << endl; }
    ~Basic_salesperson()    { cout << "Basic_salesperson destructor \n"; } //Dtor (Destructor)
};

class Permanent_salesperson: public Basic_salesperson           //Line 1
{
private:
    float salary;

public:
    Permanent_salesperson (char *n, float s) : Basic_salesperson(n) //Derived ctor calls
    {
        // the base ctor first.
        salary = s;
        sales_id = 45321; // Accessing a protected
    } // member.
    void show (void) // Permanent_salesperson::show() first
    { // calls
        Basic_salesperson::show(); // Basic_salesperson::show()
        cout << "Salary " << salary //then it displays its
            << " ID " << sales_id << endl; // own members.
    }
    ~Permanent_salesperson()
    { cout << "Permanent_salesperson destructor\n"; }
};

void main (void)
{
    Basic_salesperson e1("Adams"), *p1, *p2;
    Permanent_salesperson x1("Peter Scott", 30000.0), *px = &x1;
    //x1 is an object of class Permanent_salesperson

```

```

Basic_salesperson e2 = e1;      //Line 2
p1 = &e2;                       //Line 3    p1 points to Basic_salesperson object
p2 = &x1;                         //Line 4    p2 points to Permanent_salesperson object
e1.show();                       //Line 5    Shows Basic_salesperson Adams
(*p1).show();                   //Line 6    e1 is an object of class Basic_salesperson
p2 -> show();                   //Line 7    p2 is a Basic_salesperson pointer
px -> show();                   //Line 8    px is a Permanent_salesperson pointer
}                                //All destructors are called here.

```

Output

```

Name Adams                       //From Line 5
Name Adams                       //From Line 6
Name Peter Scott                 //From Line 7, p2 is a Basic_salesperson pointer
Name Peter Scott                 //From Line 8
Salary 30000 ID 45321           //Also from Line 8
Basic_salesperson destructor     //Destructor for e2
Permanent_salesperson destructor //Derived class destructor for x1
Basic_salesperson destructor     //Base class destructor also for x1
Basic_salesperson destructor     //Destructor for e1
Press any key to continue

```

The attributes declared in a super class are inherited by its subclasses and are not repeated. These can be accessed from any subclass unless they are declared *private*. Only methods of a class can access its private attributes. The attributes that are declared *protected* are accessible to subclasses but not to client classes [Rum91, Ram98].

The methods that are declared in a super class are also inherited. If a method can be overridden by a subclass, then it is declared *virtual* in its first appearance in a super class [Rum91, Fuk94, Ram98].

Methods that override inherited methods must be re-declared in the subclass, but those that are inherited (and not overridden) need not be repeated. Inherited attributes need not be repeated. Virtual operations are called using the same syntax as non-virtual operations [Rum91, Ram98].

If a super class derivation is *private*, then clients of the class cannot call inherited operations directly nor access attributes of the ancestors. C++ supports multiple inheritance, by specifying a list of super classes in the derivation statement. C++ supports several complicated variations on multiple inheritance, including a chain of constructors that are automatically invoked when a new instance is created [Rum91, Fuk94, Ram98].

6.8.5 Comparison

6.8.5.1 Specification

- For UML the inheritance is indicated with the generalisation relationship. For Object-Z the inheritance is indicated by the inclusion of the name of the inherited class schema into the schema of the subclass.
- For both UML and Object-Z the superclass(es) of a class are specified as part of the class declaration. The attributes and methods (UML) and state variables, constants and operations (Object-Z), that are declared in a super class are inherited by its subclasses (refer to the discussion on the implementation into C++).
- For both the Object-Z and UML a supertype entity comprises attributes common to all its subtypes and has relationships with its subtypes. Each subtype entity has additional information.
- For this specific example it appears that the information conveyed by the specifications in UML and Object-Z are equally comprehensive and clear for the purposes of an implementation. For example, in UML the inheritance of properties of the two subclasses PERMANENT_SALESPERSON and TEMPORARY_SALESPERSON from the supertype BASIC_SALESPERSON is indicated diagrammatically while in Object-Z it is indicated by schema inclusion.

6.8.5.2 Verification and Refinement

For the verification and refinement both the UML and Object-Z have concrete representations for the abstract states. Although a matter of taste, the UML diagram appears to be easier to grasp at a glance, while the Object-Z definitions are harder to interpret at a first glance. However, the Object-Z definitions are more precise which allows for greater precision (e.g. the specifier can reason about them in a formal way).

6.8.5.3 Implementation

Both UML and Object-Z provide equally clear specifications and refinements for the implementation into C++. We cannot state categorically that the implementation is easier for the one above the other.

6.9 Associations

6.9.1 Specifications in Object-Z and UML

The associations are treated in the same way for Object-Z as for Z. The comparison for Object-Z and UML is therefore similar to a large extent as for Z and UML. Refer to Chapter 5 for a discussion of these.

6.9.2 Implementation into C++

There are two approaches to implement associations and aggregations [Rum91, Hel98]:

- Buried pointers

This is used for languages that do not explicitly support association objects. Attributes needed to implement the buried pointers must be added to the class definitions. A binary association is implemented as an attribute in each associated object, containing a pointer to the related object or to a set of related objects. If the association is only traversed in one direction, a pointer need only be added to one of the classes. Pointers in the 'one' direction are simple to implement; they simply contain object references. In the 'many' direction, pointers require a set of objects, or an array of the association is ordered, implemented with a collection class object from a class library.

- A distinct container object.

An explicit association object is conceptually a set of tuples, each tuple containing one value from each associated class. A binary association object can be implemented as two *Dictionary* objects, each dictionary mapping in one direction across the association.

- Associations cannot be simulated by attributes on classes without violating encapsulation of classes because the paired attributes composing an association are not independent [Rum91, Hel98].
 - When one pointer in the implementation of an association is updated, it is implied that the other pointer must be updated as well to keep the implementation consistent. The individual
-

attributes should not be made freely available externally because they ought not be updated separately [Rum91, Hel98].

- An externally available method to update the attributes cannot be attached to either class of the association without access to the internal implementation of the other class, because the attributes are mutually constrained [Rum91, Hel98].

The many-to-one association between TEACHER and SCHOOL (refer to figures 5.7 and 5.8, Chapter 5) are implemented in C++ using pointers. This is an arbitrary example. Dynamic objects are used and the arrays can change in length.

```
#include <iostream.h>
```

```
#include <string.h>
```

```
class Teachers
```

```
{
```

```
private:
```

```
    char *line;
```

```
    int ID;
```

```
public:
```

```
    Teachers(void);
```

```
    ~Teachers(void);
```

```
    int get_ID(void) {return ID;}
```

```
};
```

```
Teachers::Teachers (void)
```

```
{
```

```
    int qty;
```

```
    cout<<"enter Teacher ID number :";
```

```
    cin >> ID;
```

```
    line = new char [qty];
```

```
}
```

```
Teachers::~~Teachers (void)
```

```
{
    cout << "de-constructing " << ""
         << ID << endl;
    delete [] line;
}
// End of Teachers class

const int SIZE = 3; //Amount by which the "p" array is increased

class Schools
{
private:
    Teachers **p;
    int max, top;
public:
    Schools() { p = new (Teachers *[SIZE]);
//Initializing Schools
    max = SIZE;
    top = -1;
    }
    ~Schools(void);
    void add_another(Teachers *teachers_ptr);
};

Schools::~~Schools(void)
{
    int i;
    for ( i = 0; i <= top; ++i)
//Delete all the Schools
        delete p[i];
    delete [] p;
}
void Schools::add_another(Teachers *teachers_ptr)
{
```

```
if (top < max - 1)
{
    ++top;
    //There is still an unused element
    // in the "p" array to add the new Teacher
    p[top] = teachers_ptr;
}
else
{
    cout << "Please wait while adjusting memory\n";
    Teachers **new_t; int i;
    new_t = new(Teachers *[max + SIZE]);
    //First, create a new and longer array. Then copy
    //the addresses from the old array, "p[],"
    for (i = 0; i <= max; ++i)
        new_t[i] = p[i];
    max = max + SIZE; //to the new array, "new_t[]."
    ++top; //Adjust "max" and increment "top."
    new_t[top] = teachers_ptr; //Place the new teacher there.
    delete [] p; //Delete the old array.
    p = new_t;
}
}

void main (void)
{
    Schools schools;
    int school_no, choice, qty;
    cout << "Enter the school number ";
    cin >> school_no;
    cout << "Enter the number of teachers ";
    cin >> qty;
    Teachers *ptr;
    cout << "1. To add another teacher, anything else to quit: ";
```

```
cin >> choice;
for ( ; choice == 1; )
{
    ptr = new Teachers;
    schools.add_another(ptr);
    cout << "1. To add another teacher, anything else to quit: ";
    cin >> choice;
}
}
```

If an association is implemented as a separate object, then no attributes need be added to either class in the association. If the association involves a small fraction of objects, then a separate association object uses less storage than a pointer in each object [Rum91].

6.9.3 Comparison

- The comparison for associations is the same as that given in Chapter five, Section 5.6, for the comparison between Z and UML for relationships.
 - The refinement of and accompanying comparison between UML and Object-Z for associations are the same as indicated in Chapter 5, Section 5.6.
 - In Object-Z the one-to-many relationship is represented formally by a mathematical relation where the inverse is the function representing the equivalent many-to-one relationship [Pol92].
 - For both Object-Z and UML, one-to-one relationships are also used to model subtypes.
 - In Object-Z the one-to-one relationship can be modelled in either direction as a partial injection ($\rightarrow\rightarrow$).
 - In UML, one-to-one relationships are used to model subtypes. A supertype entity comprises attributes and relationships common to all subtypes. There is no notion of subtyping in Object-Z [Pol92].
 - The formal specification in Object-Z of a many-to-many relationship uses a mathematical relation.
 - For both UML and Object-Z a binary association is implemented as an attribute in each associated object, containing a pointer to the related object or to a set of related objects.
-

6.10 Summary and conclusions

In this chapter Object-Z and UML have been compared for some aspects regarding specification, refinement and implementation. From the opinions of the different authors, it is clear that both Object-Z and UML have their own particular advantages and disadvantages. As far as the specification notations are concerned, these strengths and weaknesses can be summarised as follows [Fow98, Sch99, Pol92, Boo97, Rum97, Jac97]:

UML

Category: Semiformal

- Strengths:*
- Can be understood by the client
 - More precise than informal methods
 - Many people find these methods useful
 - These methods have little rigour and the notation appeals to intuition rather than formal definition
 - Visual and graphic
 - Communication medium for non-experts
 - Present the specification even after formal definition
- Weaknesses:*
- The specification is not as precise as formal methods
 - The specification cannot be refined into implementation code
 - The consistency of the data and processing is not so easy to verify

Object-Z

Category: Formal

- Strengths:*
- Mathematically precise
 - Can reduce specification faults (i.e. reduces ambiguity)
 - Supports correctness proofs
 - Can reduce development cost and effort
- Weaknesses:*
- Hard to learn and use
 - Often difficult for clients to understand
 - There is no way to prove that the mathematical specification actually meets the real, often informally stated, requirements of the system
 - Formal methods are hard to understand and manipulate, often more so than the programming languages itself.
-

Both Object-Z and UML embrace object-oriented concepts: e.g. object, class, inheritance and composition. In UML the class is defined with its associated attributes and operations. For Object-Z, the constant and state variables are collectively called *attributes* found in the state schema. The operations in Object-Z are defined in the operation schemas.

The refinement of Object-Z progresses through the validation, verification, data refinement, operation refinement and refinement calculus phases. Unless UML is formalised in a predicative notation, the refinement of UML will be a direct translation into the implementation language from the specification in UML.

The implementation into C++ of an Object-Z specification is arguably easier than an implementation starting from UML, owing to the more precise nature of Object-Z. UML though, is easier to read and understand than Object-Z.

Chapter 7

Case Study: From specifications in UML and Object-Z to implementation in Object-Oriented COBOL: The ITEM system

In this chapter a system, the ITEM system demonstrates how a system is developed from the original specifications written in English, through the analysis, design, formal specification and implementation phases to the complete and running system.

7.1 Introduction

This chapter describes a case study where an intended system (ITEM system) described in English, can in the first analysis and modeling phase, be analysed by the OOA&D (object oriented analysis and design) technique to find the right classes of the application. From this the use-case scenarios and the class diagrams in UML, and the class schemas in Object-Z are constructed. In the second high-level phase a control class is added and the CRC cards in UML and operation schemas in Object-Z are drawn. In the third low-level phase the deliverables are the CRC cards, completed class diagrams and the object-message diagrams (and/or object-trace diagrams) in UML. In Object-Z a complete Object-Z system diagram is given.

The programs (ITEMDriver, the ITEMClass and the ITEMDatabaseInterface) are then written and compiled using the Micro Focus Cobol compiler. The programs are compiled and run using some test data (refer to the DISPLAYED output given after the program listings).

This case study is a culmination of the work of the previous chapters in that the differences between UML and Object-Z (and indirectly Z) are illustrated in a practical case study. Also, in this case study, because the ITEM system is taken from the analysis phase through to the implementation phase, the position of specification, refinement and implementation within the systems software life cycle, as described in Chapter 2, are indicated.

Extra UML specifications with the corresponding Object-Z specifications that were not examined in Chapter 6 are discussed, for example CRC cards, Object-message diagrams, object creation, and message trace diagrams.

In chapters 3, 4 and 5 the refinement and implementation of specifications into the non-object-oriented language C were illustrated. In Chapter 6 the implementation of specifications into an object-oriented language C++ was shown. The case study in Chapter 7 uses Object-Oriented Cobol to serve two main purposes:

- To show the implementation of specifications into one more of the major third generation object-oriented languages.
- Object-oriented Cobol is written in English, and much easier to understand and follow than C++ (refer also to Section 7.2).

This ITEM system is derived from the ROOM system by Wilson Price [Pri97]. The Object-Z specifications and refinement is my own interpretation, and so are the compilation and running of the programs on the Micro Focus Personal COBOL 2.0.

7.2 The next generation of Cobol

One of the most exciting new features of the COBOL 2000 Standard is the incorporation of *Object-Orientation* (OO) into the language. OO Cobol is now available and offered by several vendors including IBM, Hitachi, and Micro Focus. Cobol is the dominant business language, and now have object-oriented capabilities while retaining Cobol's traditional strengths: readability, easy maintenance, powerful file handling, and good reporting [Gra98, Gra00].

The concept of object-oriented programming will be added to the Cobol standard as an *extension*, so that any new compiler incorporating object-oriented techniques will be compatible with previous standards. Object-Oriented Cobol and traditional Cobol will be usable together, making it possible to migrate to object-oriented Cobol in stages [Ste97, Gra98, Gra00].

Object technology and Object-Oriented Cobol will play increasingly important roles in software development in coming years. It incorporates ideas and concepts advanced in database design and artificial intelligence, and it embody the best practices of software engineering [Pri97].

The Micro Focus Cobol compiler was used to compile and run the Object-Oriented Cobol programs of the ITEM system.

In his latest book Wilson Price [Pri98] shows how Cobol can be used in creating Web applications. With the Micro Focus NetExpress compiler, legacy applications can be brought to

the Web using Cobol. Relatively basic extensions to Cobol from a number of vendors provide Web programming capabilities.

7.3 The Micro Focus Cobol compiler

The Micro Focus Personal COBOL is a tool to help people learn about today's COBOL and about how to apply object-oriented programming to business applications. The Personal COBOL kit contains three main tools, namely the Animator, the Class Browser, and the Personal Dialog System.

The Personal COBOL Browser is a viewing and editing tool designed for managing multiple programs. The Browser makes it easy to manage a large number of source code files at once, without having to think about opening and closing each file individually.

The class browser is an environment for editing, compiling, and running programs. The Class Browser allows the user to group files for a given application as a project. When the Class Browser is used, a project is first created. In this project the programs and files to be used are entered, compiled and run.

The Personal COBOL programming environment is called Animator. Animator enables you to edit, compile, and execute your program in one window. It is an interactive debugging environment which assists in the detection of difficult-to-find errors. Animator allows the user to set breakpoints: these are points in the program at which Animator halts execution when you select the run icon. This is useful if the program executes a large number of statements before encountering the statement that interests the user.

Micro Focus Personal Dialog System is a tool for creating sophisticated Windows based GUIs (graphical user interface) for an application. With Personal Dialog System, the management of the interface is done for the user, leaving the user with a shorter and less complex COBOL program to process the data and control the logic. In Personal Dialog System, the user can easily define a number of alternative interfaces for the same application, without needing to change the main application programs. The interfaces can be prototyped without writing any application programs, which makes Personal Dialog System a powerful tool for system designers.

7.4 The ITEM system

7.4.1 Introduction

The ITEM system is a system that consists of three Object-Oriented Cobol programs namely the driver (Item08Driver), the ItemClass (Item05cl.cbl) and the ItemDatabaseInterface (Item08da.cbl). The programs (ItemDriver, the ItemClass and the ItemDatabaseInterface) are then written and compiled using the Micro Focus Cobol compiler. The programs are compiled error free, and then run using some test data (refer to the DISPLAYED output given after the program listings).

The database is an indexed sequential file (ITEM2.DAT) that contains ten records. Each record contains the item number, standard configuration price, special configuration price, discount and item expiry-date for a certain item. The item numbers range from 200 to 209. Any other item number is invalid.

The Item08driver interactively asks the user which item he wants to enquire about. The user responds by entering a item number. Then Item08driver asks the user whether he wants to enquire about the item expiry-date, standard prices or discount prices of the item. The user responds by entering either a(n) 'e', 's', or 'd'. The information is sent by the driver program to the ItemClass program. The ItemClass program invokes the ItemDatabaseInterface program that reads the record from the indexed file (Item2.dat). The item object in the ItemClass program is then loaded with this information. Any necessary calculations that needs to be done, is executed in the ItemClass program, that sends the information back to the driver program, that displays the information interactively to the user. Refer to figures 7.2 and 7.3.

If the user enters an item number < 200 or > 209 , it is an invalid item number. If the user enters a 0 item number, the processing is complete. If the user enters a valid item number and 'e' for expiry-date, the item number and item expiry-date will be displayed. If the user enters a valid item number, and a 's' for standard prices, the item number, the standard configuration price and the special configuration price will be displayed. If the user enters a valid item number and a 'd' for discount prices, the discount will be read from the relevant record and the discount prices will be calculated by multiplying the standard and configuration prices by the discount in the

ItemClass program. The discounted standard and special configuration prices will then be displayed by the driver program to the user.

7.4.2 Phase 1: Analysis and Modelling

7.4.2.1 A brief description of the ITEM system

The Sales Department of Company ABC handles ITEM sales of its inventory to customers. Both ITEM data and customer data are stored in indexed files. A basic information system is needed for users to display ITEM data such as prices and ITEM expiry-date.

The first phase of the system comprises the following:

- A problem description defining what is expected of the intended software.
- Usage scenarios to drive design and establish a basis for later testing.
- A list of candidate classes required to satisfy the needs of the system. A class diagram identifies relationships between classes.

A difficult task of OOA&D is to find the right classes of an application. One technique is to focus on nouns and noun phrases in the problem statement. For example, for the ITEM system the nouns are given in italics:

The *Sales Department* of *Company ABC* handles *item* sales of its *inventory* to *customers*. Both *item data* and *customer data* are stored in *indexed files*. A basic information system is needed for users to display item data such as *prices* and *item expiry-dates*.

7.4.2.2 Next Step - Examination of Candidate Classes

Ascertain whether or not each of the candidate classes is needed to accomplish the goals of this application. The goal of the system is to display the item data such as prices and item expiry-date.

Upon consideration of each of the identified candidate classes, the following classes do not appear necessary to achieve the goals of this application:

Sales Department, Company ABC, customers, customer data.

The candidate classes include (obtained from the last paragraph Section 7.4.2.1):

- *Item*: On a broad basis, the entire application is an information system for items.
- *Items and inventory* are synonymous. Users might be queried at this point whether or not other facilities exist that might affect the design.
- *Indexed file*: The source of item data is an indexed file. A database interface class will be required.
- *"Users to display"*: A user interface class will be required.
- *Item data, prices, and item expiry-date*: Data (possibly attributes of a candidate class). They are not classes.

7.4.2.3 Next Step - Use-Case Scenarios

This is an external view of the system describing interactions between system components in response to an event - a sequence of events. For example what happens when the user requests an item that is not in the database; what if the user wants information on another item. (Refer to Section 7.5).

7.4.2.4 Next Step - Class Diagrams

For the class diagrams refer to the UML diagrams for the ITEM system. Also refer to the Object-Z ITEM system schema (Section 7.5).

7.4.3 Phase 2: High-Level Design

Responsibilities are allocated to individual classes. Class and object definitions developed from analysis are refined to meet constraints of the implementation. The designer assigns a name to each class that reflects its function. Classes are introduced to serve as needed go-betweens to isolate application logic from machine dependencies.

The CRC card is a commonly used tool. Deliverables of this phase are the documenting CRC cards and further refined class diagrams, the Object-Z class schemas and the operation schemas (Section 7.5).

7.4.3.1 Adding a Control Class

In phase 2 the need for a control class is recognized to coordinate services between objects of the system.

7.4.3.2 CRC Cards

The responsibilities and collaborators of each class are summarized in table form. This lead to the Class-Responsibility-Collaborators which is a responsibility-driven tool called the CRC card. (Refer to the description of the CRC cards and the UML diagrams for the ITEM system).

Figure 7.1 is a set of class responsibilities and collaborators for the ITEM system. During a CRC session, participants record candidate classes on these cards. Any class (not just DBI) can invoke ITEM's methods; the invoking class need only provide the necessary input.

7.4.4 Phase 3: Low-Level Design

7.4.4.1 Phase 3 Deliverables

Phase 3 adds the necessary detail to turn a design into an application. The class definitions must have a sufficient level of detail to describe messages, attributes, object creation, and relationships [Pri97]. The deliverables of Phase 3 include the following:

- CRC cards.
- Completed class diagrams and Object-Z system schema and class schemas.
- Object-message diagrams (and/or object-trace diagrams), Object-Z class and operation schemas.

Together, these deliverables provide sufficient detail so that a programmer can generate code that tests both usability and performance.

7.4.4.2 Responsibilities of the Driver

The responsibility of the Driver is to instantiate the UI (User Interface), the ItemManager and the DBI (Database Interface). The Item is instantiated only if DBI finds the requested item in its database.

<p>Driver</p> <p><u>Responsibilities</u></p> <p>Create ITEM manager object</p> <p>Create DBI object</p> <p>Create UI object</p>	<p><u>Collaborators</u></p> <p>ITEM manager</p> <p>DBI</p> <p>UI</p>
<p>ITEM</p> <p><u>Attributes</u></p> <p>ITEM number</p> <p>Standard price</p> <p>Special price</p> <p>Discount rate</p> <p>ITEM expiry-date</p> <p><u>Responsibilities</u></p> <p>Populate the ITEM object</p> <p>Calculate discounted ITEM prices</p> <p>Return data</p>	<p><u>Collaborators</u></p>
<p>UI</p> <p><u>Attributes</u></p> <p>ITEM object handle</p> <p><u>Responsibilities</u></p> <p>Get ITEM number from the user</p> <p>Process requested ITEM number</p> <p>Provide program repetition control</p> <p>Query user for menu choice</p> <p>Display messages</p> <p>Display ITEM data</p>	<p><u>Collaborators</u></p> <p>ITEM manager</p>
<p>ITEM manager</p> <p><u>Attributes</u></p> <p>UI object handle</p> <p>DBI object handle</p> <p><u>Responsibilities</u></p> <p>Read a ITEM record</p> <p>Access ITEM data</p> <p>Display data values</p>	<p><u>Collaborators</u></p> <p>DBI</p> <p>ITEM</p> <p>UI</p>
<p>DBI</p> <p><u>Responsibilities</u></p> <p>Create the ITEM object</p> <p>Accept ITEM number</p> <p>Return ITEM object handle</p> <p>Access the desired record</p> <p>Populate the ITEM object</p>	<p><u>Collaborators</u></p> <p>ITEM</p> <p>ITEM manager</p> <p>ITEM manager</p> <p>ITEM</p>

Figure 7.1. A set of class responsibilities and collaborators for the ITEM system

7.5 UML and Object-Z specifications for the ITEM system

7.5.1 Class diagram for ITEM system

7.5.1.1 Specifications in UML

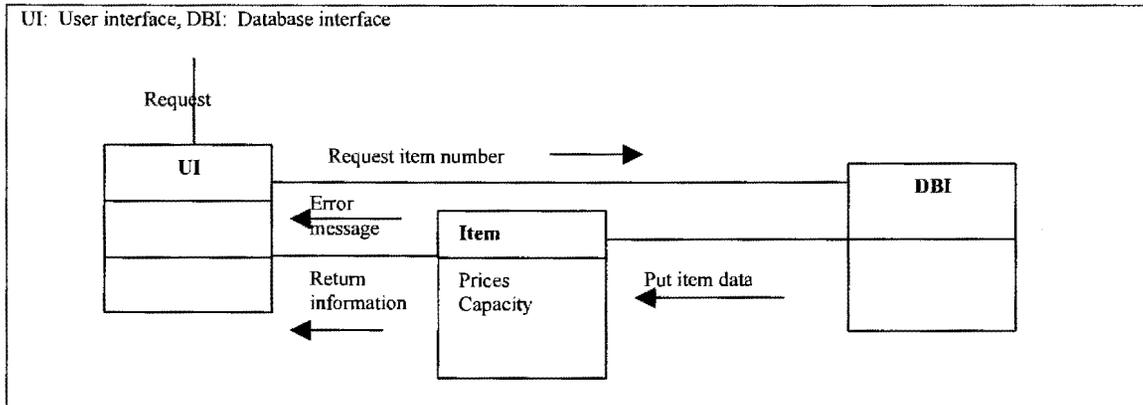


Figure 7.2. A simplified class diagram for the ITEM system

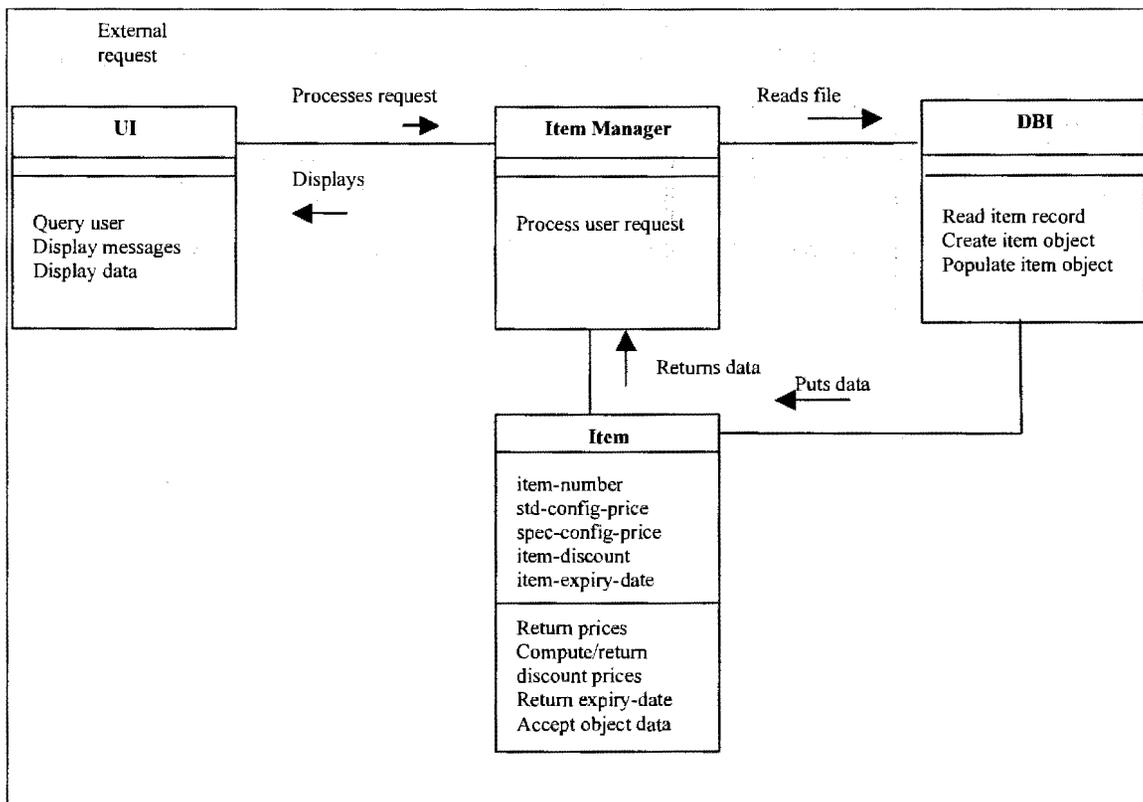


Figure 7.3 A more detailed class diagram for the ITEM system

7.5.1.2 Specifications in Object-Z

7.5.1.2.1 Basic types, sets, data, constants, choices and messages

Basic types (given sets):

[NUMBERS, SPRICE, PPRICE, DISCOUNT, EXPIRY-DATE, NUMBERS2, RECORD]

The following are examples of these sets:

NUMBERS = {200,201,202,203,204,205,206,207,208,209} $\subseteq \mathbb{N}$

SPRICE = {225,240,255,280,305,340,350,375,380,400} $\subseteq \mathbb{N}$

PPRICE = {385,405,425,465,505,565,610,615,625,655} $\subseteq \mathbb{N}$

DISCOUNT = {.05,.05,.05,.05,.05,.05,.07,.07,.10,.10} $\subseteq \mathbb{N}$

EXPIRY-DATE =

{130899,241000,250499,180901,141200,091199,121199,150599,130899,151200} $\subseteq \mathbb{N}$

NUMBERS2 = {1,2,3,4,5,6,7,8,9,10} $\subseteq \mathbb{N}$

Record layout for the indexed file ITEM2.DAT

Record number	ITEM number	Standard configuration price	Special configuration price	ITEM discount	ITEM expiry-date
1	200	225	385	.05	130899
2	201	240	405	.05	241000
3	202	255	425	.05	250499
4	203	280	465	.05	180901
5	204	305	505	.05	141200
6	205	340	565	.05	091199
7	206	350	610	.07	121199
8	207	375	615	.07	150599
9	208	380	625	.10	130899
10	209	400	655	.10	151200

The input and output corresponds with the input and output displays (refer to the DISPLAYED OUTPUT following the printout of the last program).

Input option constants:

CHOICE1 ::= number | 0

CHOICE2 ::= e | s | d

number: ITEM number

0: ITEM number indicating that the processing stops

e: expiry date

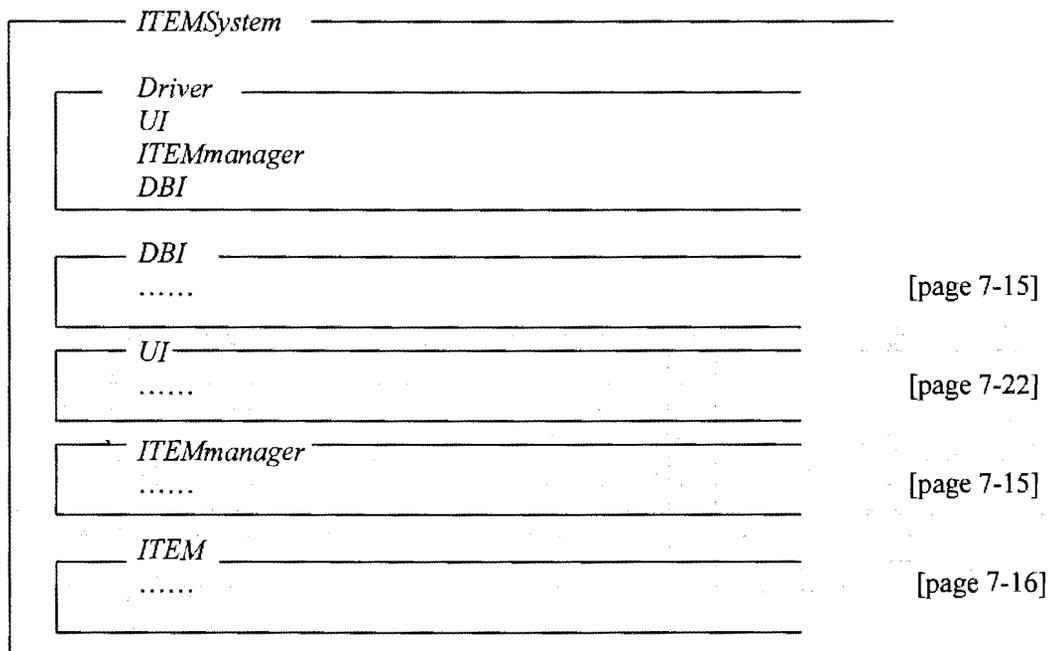
s: standard price
d: discounted price

Message options:

*RESPONSE::= Processing_complete | Invalid_ITEM_number | Standard_configuration_price:
 | Special_configuration_price: | ITEM_expiry_date*

7.5.1.2.2 Object-Z ITEM system

Below is the skeleton of the ITEM system. The schemas *DBI*, *UI*, *ITEMmanager* and *ITEM* are developed on the pages indicated.



7.5.1.3 Operation decomposition: implementation into Object-Oriented Cobol

This is the system that consists out of all the Cobol programs as described in Appendix C.

7.5.1.4 Comparison

- In the specifications each UML class in *DBI*, *UI*, *ITEMmanager* and *ITEM* is represented by a respective schema in Object-Z namely *DBI*, *UI*, *ITEMmanager* and *ITEM*.
- The flow of information between the classes is seen more clearly in the UML specifications than the Object-Z specifications.

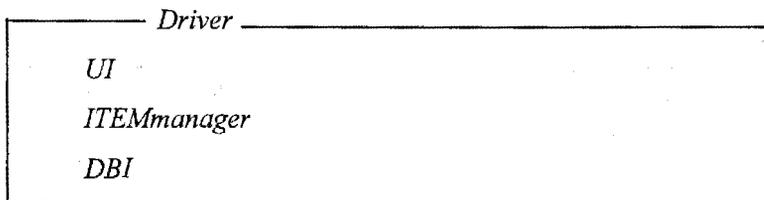
- Because of the detailed nature of Object-Z, easier implementation into Object-Oriented Cobol is facilitated (refer to sections 7.5.2.3, 7.5.3.3, 7.5.4.3).

7.5.2 CRC cards for the Driver

7.5.2.1 Specifications in UML

Program: Driver	
Responsibilities	Collaborators
<ol style="list-style-type: none"> 1. Instantiate UserInterface 2. Instantiate ItemManager 3. Instantiate DBI 4. Pass ItemManager handle to UI 5. Pass UI handle to Item Manager 6. Pass DBI handle to Item Manager 	<ol style="list-style-type: none"> 1. UserInterface (New) 2. Item Manager (New) 3. DBI (New) 4. UI 5. Item Manager 6. Item Manager
<p>Driver: The conventional program that begins the process by instantiating classes creating objects necessary for the life of the run. Also, distributes object handles thereby providing for communication between objects (establishes needed associations between objects).</p> <p><u>Attributes:</u> None</p>	

7.5.2.2 Specifications in Object-Z



The driver instantiates the Userinterface, the ITEMmanager and the DataBaseInterface. (Refer to the class diagram, the object message diagram and the set of class responsibilities and collaborators for the ITEM system).

7.5.2.3 Operation decomposition: implementation into Object-Oriented Cobol

For the Cobol programs refer to Appendix C.

Object-Z

From Schema Driver:

UI

Object-Oriented Cobol statements

From ITEM08DR:

```
45 Perform with test after
46 until ITEM-number = 0
47 Display " "
48 Display "ITEM number <0 to terminate>? "
49 with no advancing
50 Accept ITEM-number

89 250-get-menu-selection.
90 Display " "
91 Display "The options are:"
92 Display " E - ITEM expiry-date (default)"
93 Display " S - Standard ITEM prices"
94 Display " D - Discounted ITEM prices"
95 Display " "
96 Display "Your choice <E,S,D>? " no advancing
97 Accept menu-choice
98
```

Object-Z

From Schema Driver:

ITEMmanager

Object-Oriented Cobol statements

ITEMmanager is instantiated only if DBI finds the requested ITEM in its database

From ITEM08DA:

```
77 Read ITEM-File
78 Invalid key
79 Set ls-theITEMHandle to null
80 Not invalid key
81 Invoke ITEMClass "New"
82 Returning ls-theITEMHandle
81 Invoke ls-theITEMHandle
82 "populate-the-ITEM-object"
83 Using ITEM-record
```

Object-Z

From Schema Driver:

DBI

Object-Oriented Cobol statements

From ITEM08DR:

```
42 Invoke DatabaseInterface "New"
43 Returning theDBIHandle
```

7.5.2.4 Comparison

- For both the UML and Object-Z specifications it is clearly stated that the Driver instantiates the classes creating objects necessary for the life of the run.
- For the UML specifications the responsibilities and collaborators columns add more detail, which can only be achieved by the Object-Z specifications if the individual schemas are studied.
- For both the UML and Object-Z specifications the objectives of the specifications are clearly indicated.
- The detailed nature of the Object-Z specifications facilitate easier implementation into Object-Oriented Cobol.

7.5.3 CRC cards for ITEM

7.5.3.1 Specifications in UML

Class: Item	
Responsibilities	Collaborators
<ol style="list-style-type: none"> 1. Accept Item data from DBI. 2. Calculate discounted Item prices. 3. Return data. 	<ol style="list-style-type: none"> 1. DBI 2. 3. Driver
<p>Item: The object in the system that represents Items the organization rents to customers.</p> <p><u>Attributes:</u> Item number standard price special price discount expiry-date</p>	

7.5.3.2 Specifications in Object-Z

DBI

```

ITEMnumber: ℙ NUMBERS
n: ℙ NUMBERS2
standardprice: ℙ SPRICE
specialprice: ℙ PPRICE
discount: ℙ DISCOUNT
expiry-date: ℙ EXPIRY-DATE
sprice: NUMBERS → SPRICE
ddiscount: NUMBERS → DISCOUNT
cexpiry-date: NUMBERS → EXPIRY-DATE
pprice: NUMBERS → PPRICE
record: ℙ RECORD

```

```

#dom ITEMnumber =
  #ran standardprice; #ran discount; #ran expiry_date;
  #ran special_price

```

The ITEMnumber, standardprice, specialprice, discount and expiry-date are the sets as specified (refer to the printout of the indexed sequential file ITEM.DI). There is a one-to-one mapping of the domain of the ITEM numbers to the ranges of the prices, capacities and discounts. That is, if the ITEM number is 103, then it is the fourth number in the ITEM number sequence, and the corresponding values of the prices, capacities and discounts will be the fourth position in their respective sequences. The following schema defines a database record:

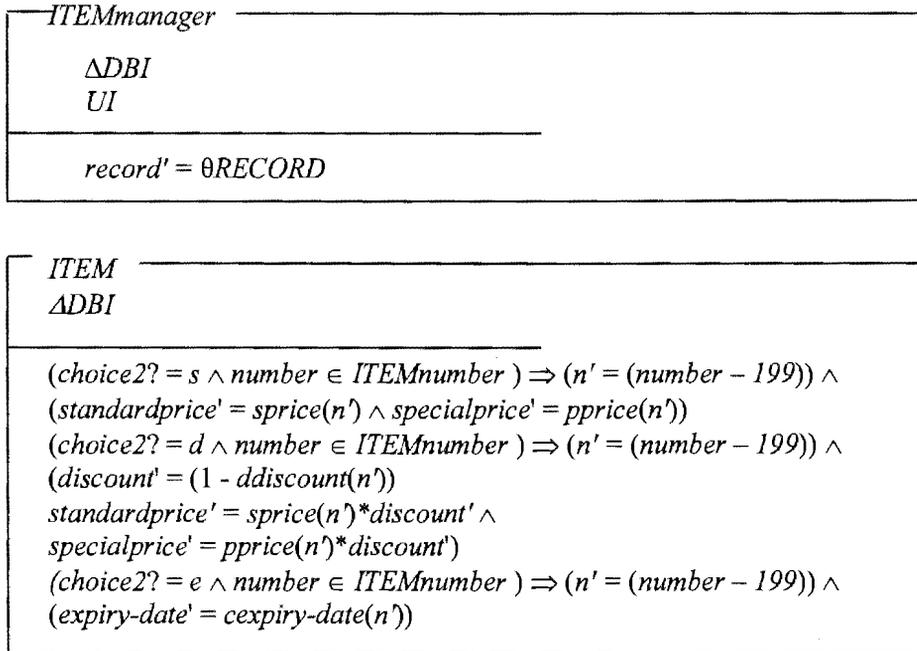
RECORD

```

ITEMnumber: ℙ NUMBERS
pprice: NUMBERS → PPRICE
sprice: NUMBERS → SPRICE
ddiscount: NUMBERS → DISCOUNT
cexpiry-date: NUMBERS → EXPIRY-DATE

```

Schema RECORD is made up of five database fields (refer to Section 7.5.1.2.1).



The *ITEMmanager* populates the ITEM data object from the corresponding database record. Such information is then further restricted by *ITEM* above (e.g. the necessary calculations, in the case of the discounted prices, are made). The output display to the user from the *UI* (UserInterface) is the standard and special prices for the particular ITEM. If the user chooses c (for expiry-date), the *ITEMmanager* populates the ITEM-data object from the corresponding record from the database, and sends the information to *ITEM*, from where the information is sent to the *UI* that displays the ITEM expiry-date to the user.

7.5.3.3 Operation decomposition: implementation into Object-Oriented Cobol

Object Z

From Schema DBI:

ITEMnumber: \mathbb{P} NUMBERS
n: \mathbb{P} NUMBERS2
standardprice: \mathbb{P} SPRICE
specialprice: \mathbb{P} PPRICE
discount: \mathbb{P} DISCOUNT
expiry-date: \mathbb{P} EXPIRY-DATE

Object-Oriented Cobol statements

From ITEM05CL:

30 OBJECT.
 31

```

32 Data Division.
33 Object-Storage Section.  *> OBJECT DATA
34 01 ITEM-data.
35 10 ITEM-number          pic 9(03).
36 10 std-config-price    pic 9(03).
37 10 spec-config-price   pic 9(03).
38 10 ITEM-discount       pic v9(02).
39 10 ITEM-expiry-date    pic 9(02).

```

Object-Z

From Schema DBI:

```

sprice: NUMBERS ↔ SPRICE
ddiscount: NUMBERS ↔ DISCOUNT
cexpiry-date: NUMBERS ↔ EXPIRY-DATE
pprice: NUMBERS ↔ PPRICE
record: P RECORD

```

Object-Oriented Cobol statements

From ITEM08DA:

```

20 File-Control.
21 Select ITEM-File assign to disk "a:ITEM2.dat"
22 organization is indexed
23 access is random
24 record key is rr-ITEM-number.
25
26 Object Section.
27 Class-Control.
28 ITEMDatabaseInterface is class "ITEM08da"
29 ITEMClass is class "ITEM05cl"
30 Base is class "base"
31 .
32
33 Data Division.
34 File Section.
35 FD ITEM-File.
36 01 ITEM-record.
37 10 rr-ITEM-number          pic 9(03).
38 10 rr-std-config-price    pic 9(03).
39 10 rr-spec-config-price   pic 9(03).
40 10 rr-ITEM-discount       pic v9(02).
41 10 rr-ITEM-expiry-date    pic 9(02).
42

```

Object-Z

From Schema DBI:

```
#dom ITEMnumber = #ran standardprice; #ran discount; #ran expiry_date; #ran special_price
```

Object-Oriented Cobol statements

ITEM2.DAT (Indexed file)

```

1002253850538
1012404050540
1022554250540
1032804650543
1043055050543
1053405650547
1063506100749

```

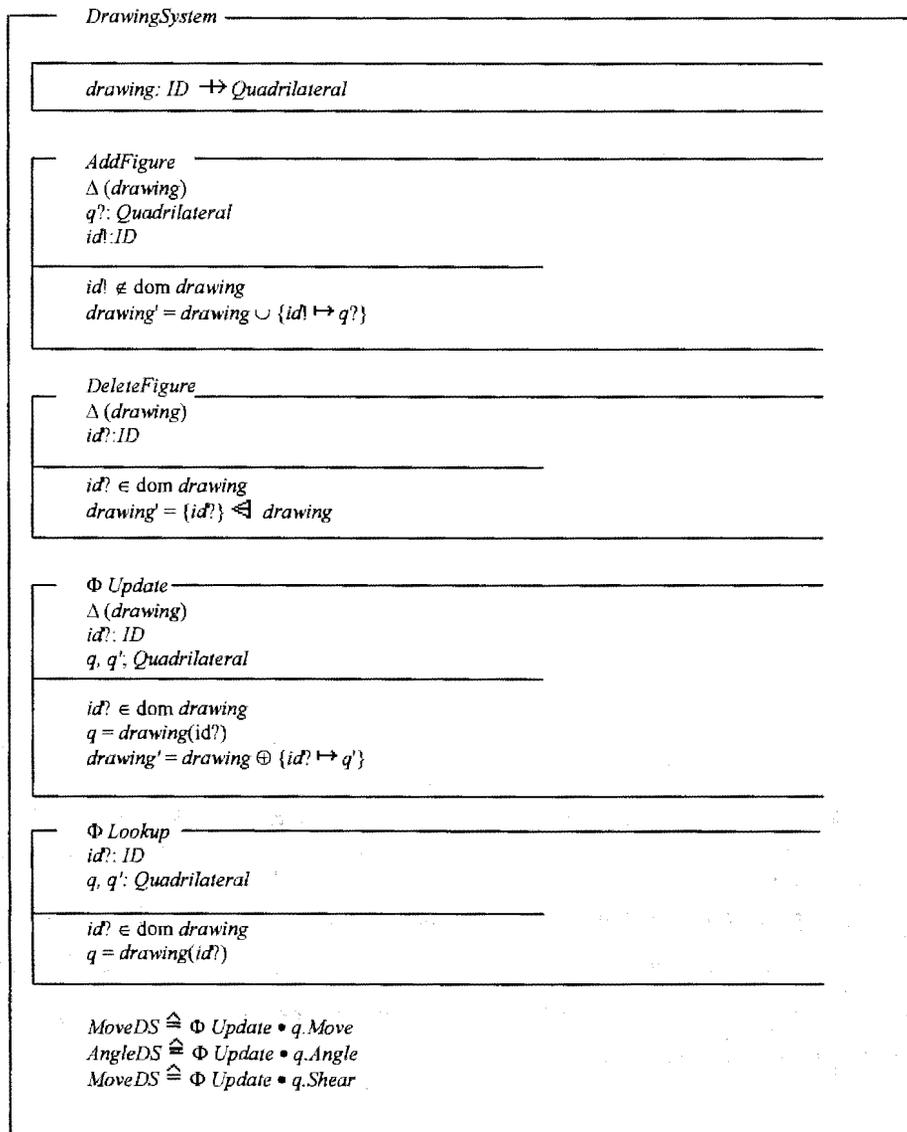


Figure 6.3 The drawing system

Refer to the following example [Ram98]:

Base class Employee:

```
class Employee
{
  private:                //Implementation section
    char name[20];
  protected:            //Accessible by base and derived class members
```

1073756150745
 1083806251045
 1094006551050

Object-Z

From Schema DBI:

record = \emptyset RECORD

Object-Oriented Cobol statements

From ITEM08DR:

```
61 200-process-user-request.
62 Invoke theDBIHandle "read-ITEM-record"
63 Using ITEM-number
64 Returning aITEMHandle
```

From ITEM08DA:

```
59 Method-ID. "open-file".
60 *>-----
61 Procedure Division.
62 Open input ITEM-File
63 .
64 End Method "open-file".
65
66 *>-----
67 Method-ID. "read-ITEM-record".
68 *>-----
69 Data Division.
70 Linkage Section.
71 01 ls-ITEM-number          pic 9(03).
72 01 ls-theITEMHandle object reference.
73
74 Procedure Division Using ls-ITEM-number
75 Returning ls-theITEMHandle.
76 Move ls-ITEM-number to rr-ITEM-number
77 Read ITEM-File
78   Invalid key
79     Set ls-theITEMHandle to null
80   Not invalid key
81     Invoke ITEMClass "New"
82     Returning ls-theITEMHandle
83 Invoke ls-theITEMHandle
84 "populate-the-ITEM-object"
85 Using ITEM-record
86 End-Read *>ITEM-File
87 .
88 End Method "read-ITEM-record".
```

Object-Z

From Schema ITEMmanager:

Δ DBI
 UI

record' = \emptyset RECORD

Object-Oriented Cobol statements**From ITEM05CL:**

```

30 OBJECT.
31
32 Data Division.
33 Object-Storage Section.  *> OBJECT DATA
34 01 ITEM-data.
35 10 ITEM-number          pic 9(03).
36 10 std-config-price     pic 9(03).
37 10 spec-config-price    pic9(03).
38 10 ITEM-discount        pic v9(02).
39 10 ITEM-expiry-date     pic 9(02).

106 Method-id. "populate-the-ITEM-object".
107 *>-----
108 Data Division.
109 Linkage Section.
110 01 ls-ITEM-data.
111 10 ls-ITEM-number       pic 9(03).
112 10 ls-std-config-price  pic 9(03).
113 10 ls-spec-config-price pic 9(03).
114 10 ls-ITEM-discount    pic v9(02).
115 10 ls-ITEM-expiry-date pic 9(02).
116
117 Procedure Division Using ls-ITEM-data.
118 Move ls-ITEM-data to ITEM-data
119 .
120 End Method "populate-the-ITEM-object".

```

Object-Z**From Schema ITEM:**

ADBI

$$(choice2? = s \wedge number \in ITEMnumber) \Rightarrow (n' = (number - 199)) \wedge (standardprice' = sprice(n') \wedge specialprice' = pprice(n'))$$
Object-Oriented Cobol statements**From ITEM05CL:**

```

74 Method-id. "get-ITEM-prices".
75 *>-----
76 Data Division.
77 Linkage Section.
78 01 ls-ITEM-number       pic 9(03).
79 01 ls-ITEM-data.
80 10 ls-std-config-price  pic 9(03).
81 10 ls-spec-config-price pic 9(03).
82
83 Procedure Division Returning ls-ITEM-data.
84 Move std-config-price
85   to ls-std-config-price
86 Move spec-config-price
87   to ls-spec-config-price
88 .
89   End Method "get-ITEM-prices".

```

Object-Z

From Schema ITEM:

$$(choice2? = d \wedge number \in ITEMnumber) \Rightarrow (n' = (number - 199)) \wedge$$

$$(discount' = (1 - ddiscount(n'))) \wedge$$

$$standardprice' = sprice(n') * discount' \wedge$$

$$specialprice' = pprice(n') * discount'$$
Object-Oriented Cobol statements

From ITEM05CL:

```

47 Method-id. "get-discounted-ITEM-prices".
48 *>-----
49 Data Division.
50 Working-Storage Section.
51 01 price-factor                pic 9(01)v9(02).
52
53 Linkage Section.
54 01 ls-ITEM-number            pic 9(03).
55 01 ls-ITEM-data.
56 10 ls-std-config-price       pic 9(03).
57 10 ls-spec-config-price      pic 9(03).
58
59 Procedure Division Returning ls-ITEM-data.
60 Move std-config-price
61 to ls-std-config-price
62 Move spec-config-price
63 to ls-spec-config-price
64 Subtract ITEM-discount from 1.0
65                                giving price-factor
66 Multiply price-factor
67 by ls-std-config-price
68 Multiply price-factor
69 by ls-spec-config-price
70
71 End Method "get-discounted-ITEM-prices".

```

Object-Z

From Schema ITEM:

$$(choice2? = c \wedge number \in ITEMnumber) \Rightarrow (n' = (number - 199)) \wedge$$

$$(expiry-date' = cexpiry-date(n'))$$
Object-Oriented Cobol statements

From ITEM05CL:

```

92 Method-id. "get-ITEM-expiry-date".
93 *>-----
94 Data Division.
95 Linkage Section.
96 01 ls-ITEM-number            pic 9(03).
97 01 ls-ITEM-expiry-date       pic 9(02).
98
99 Procedure Division
100 Returning ls-ITEM-expiry-date.
101 Move ITEM-expiry-date to ls-ITEM-expiry-date
102 .
103 End Method "get-ITEM-expiry-date".

```

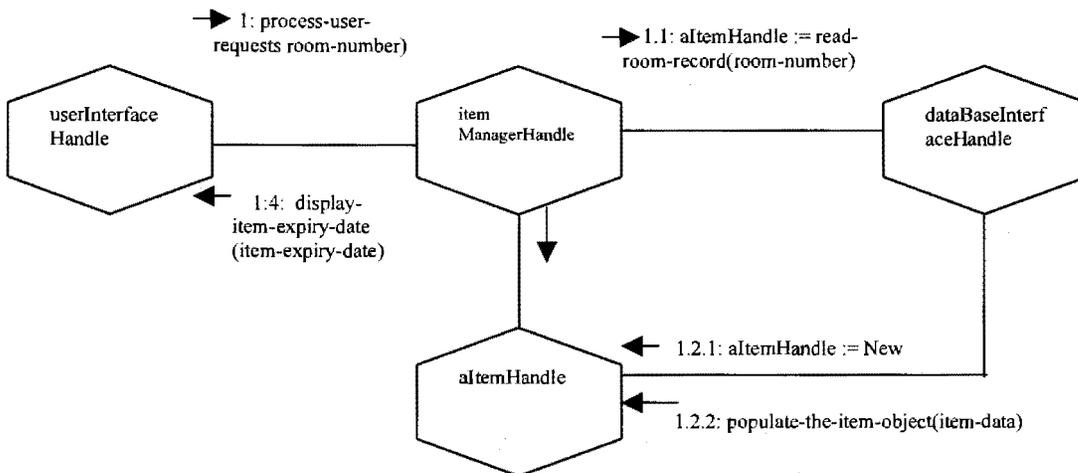
7.5.3.4 Comparison

- The UML specifications are more concise and clear than the Object-Z specifications.
- The Object-Z specifications are more detailed than the UML specifications.
- The detailed nature of the Object-Z specifications facilitate easier implementation into object-oriented Cobol.

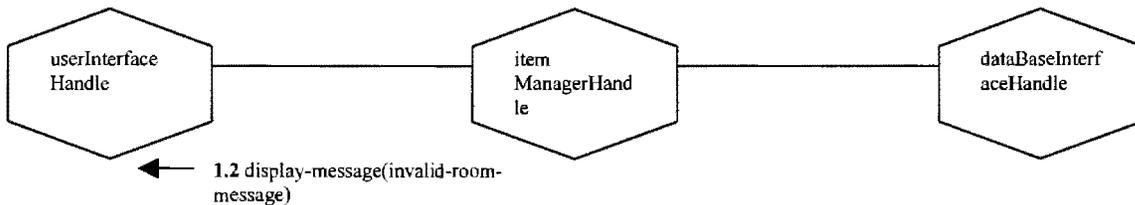
7.5.4 Object-message Diagrams

7.5.4.1 Specifications in UML

7.5.4.1.1 Object-message diagram: normal request processing (UML)



7.5.4.1.2 Object-message diagram: invalid ITEM number (UML)



7.5.4.2 Specifications in Object-Z

UI

choice1? : CHOICE1
choice2? : CHOICE2
message! : RESPONSE
ITEMnumber : \mathbb{P} NUMBERS
ITEMmanager
number : \mathbb{N}

$(choice1? = number \wedge number \in ITEMnumber \wedge choice2? = s) \Rightarrow$
 $(n' = (number - 199) . \{\emptyset ITEM\}) \wedge$
 $message! = Standard_configuration_price: standardprice' \wedge$
 $message! = Special_configuration_price: specialprice'$
 $(choice1? = number \wedge number \in ITEMnumber \wedge choice2? = d) \Rightarrow$
 $(n' = (number - 199) . \{\emptyset ITEM\}) \wedge$
 $message! = Standard_configuration_price: standardprice' \wedge$
 $message! = Special_configuration_price: specialprice'$
 $(choice1? = number \wedge number \in ITEMnumber \wedge choice2? = c) \Rightarrow$
 $(n' = (number - 199) . \{\emptyset ITEM\}) \wedge$
 $message! = ITEM_expiry_date: expiry-date'$
 $(choice1? = number \wedge number = 0) \Rightarrow$
 $(message! = Processing_complete)$
 $(choice1? = number \wedge number \notin ITEMnumber) \Rightarrow$
 $(message! = Invalid_ITEM_number)$

The user is prompted for a ITEM number. If the ITEM number is 0, the processing stops. The relevant record from the database, that is the indexed file ITEM.DI, is read by the DBI (DataBaseInterface). If the ITEM number is >109 or <100, it is an invalid ITEM number. If the ITEM number is valid, and if the user chooses the standard or discounted prices, the *ITEMmanager* is invoked.

ITEM

ΔDBI

$(choice2? = s \wedge number \in ITEMnumber) \Rightarrow (n' = (number - 199)) \wedge$
 $(standardprice' = sprice(n') \wedge specialprice' = pprice(n'))$
 $(choice2? = d \wedge number \in ITEMnumber) \Rightarrow (n' = (number - 199)) \wedge$
 $(discount' = (1 - ddiscount(n'))$
 $standardprice' = sprice(n') * discount' \wedge$
 $specialprice' = pprice(n') * discount'$
 $(choice2? = e \wedge number \in ITEMnumber) \Rightarrow (n' = (number - 199)) \wedge$
 $(expiry-date' = cexpiry-date(n'))$

7.5.4.3 Operation decomposition: implementation into Object-Oriented Cobol

Object-Z

From Schema UI:

```
choice1? : CHOICE1
choice2? : CHOICE2
message! : RESPONSE
ITEMnumber :  $\mathbb{P}$  NUMBERS
ITEMmanager
```

```
(choice1? = number  $\wedge$  number  $\in$  ITEMnumber  $\wedge$  choice2? = s)  $\Rightarrow$ 
(n' = (number - 199) . { $\emptyset$ ITEM}  $\wedge$ 
message! = Standard_configuration_price: standardprice'  $\wedge$ 
message! = Special_configuration_price: specialprice')
```

Object-Oriented Cobol statements

choice1 and choice2

From ITEM08DR:

```
45 Perform with test after
46 until ITEM-number = 0
47 Display " "
48 Display "ITEM number <0 to terminate>? "
49 with no advancing
50 Accept ITEM-number
51 Evaluate ITEM-number
52 When 0
53 Continue
54 When other
55 Perform 200-process-user-request
56 End-Evaluate *>ITEM-number
57 End-Perform *>with test after

89 250-get-menu-selection.
90 Display " "
91 Display "The options are:"
92 Display " C - ITEM expiry-date (default)"
93 Display " S - Standard ITEM prices"
94 Display " D - Discounted ITEM prices"
95 Display " "
96 Display "Your choice <E,S,D>? " no advancing
97 Accept menu-choice
98
```

From ITEM08DR:

```
62 Invoke theDBIHandle "read-ITEM-record"
63 Using ITEM-number
64 Returning aITEMHandle
```

From ITEM08DR:

```
61 200-process-user-request.

67 Evaluate aITEMHandle

78 When standard-prices
79 Invoke aITEMHandle "get-ITEM-prices"
80 Returning ITEM-prices
81 Perform 300-display-ITEM-prices
```

From ITEM08DR:

```

99 300-display-ITEM-prices.
100 Display " "
101 Display "Standard configuration price: "
102  std-config-price
103 Display "Special configuration price:  "
104  spec-config-price
105 .

```

Object-Z

From UI Schema:

$$\begin{aligned}
 &(\text{choice1?} = \text{number} \wedge \text{number} \in \text{ITEMnumber} \wedge \text{choice2?} = d) \Rightarrow \\
 &(\text{n}' = (\text{number} - 199) . \{\text{ITEM}\} \wedge \\
 &\text{message}' = \text{Standard_configuration_price: standardprice}' \wedge \\
 &\text{message}' = \text{Special_configuration_price: specialprice}')
 \end{aligned}$$
Object-Oriented Cobol statements

Same as above, except as from ITEM08DR:

```

61 200-process-user-request.
.
67 Evaluate aITEMHandle
.
73 When discounted-prices
74 Invoke aITEMHandle
75 "get-discounted-ITEM-prices"
76 Returning ITEM-prices
77 Perform 300-display-ITEM-prices

```

From ITEM08DR:

```

99 300-display-ITEM-prices.
100 Display " "
101 Display "Standard configuration price: "
102  std-config-price
103 Display "Special configuration price:  "
104  spec-config-price
105 .

```

Object-Z

From UI Schema:

$$\begin{aligned}
 &(\text{choice1?} = \text{number} \wedge \text{number} \in \text{ITEMnumber} \wedge \text{choice2?} = c) \Rightarrow \\
 &(\text{n}' = (\text{number} - 199) . \{\text{ITEM}\} \wedge \\
 &\text{message}' = \text{ITEM_expiry_date: expiry-date}')
 \end{aligned}$$
Object-Oriented Cobol statements

Same as above except as from ITEM08DR:

```

61 200-process-user-request.
.
67 Evaluate aITEMHandle
.
82 When other
83 Invoke aITEMHandle "get-ITEM-expiry-date"
84 Returning ITEM-expiry-date

```

85 Perform 350-display-ITEM-expiry-date

and from ITEM08DR:

```
106 350-display-ITEM-expiry-date.
107 Display " "
108 Display "ITEM expiry-date: " ITEM-expiry-date.
```

Object-Z

From Schema UI:

$(choice1? = number \wedge number = 0) \Rightarrow$
 $(message! = Processing_complete)$

Object-Oriented Cobol statements

Same as above, except as from ITEM08DR:

```
41 000-process-ITEM-data.
.
47 Display " "
48 Display "ITEM number <0 to terminate>? "
49 with no advancing
50 Accept ITEM-number
51 Evaluate ITEM-number
52 When 0
53 Continue

56 End-Evaluate *>ITEM-number
57 End-Perform *>with test after
58 Display "Processing complete"
58 Invoke theDBIHandle "close-file"
59 Stop run
60 .
```

Object-Z

From Schema UI:

$(choice1? = number \wedge number \notin ITEMnumber) \Rightarrow$
 $(message! = Invalid_ITEM_number)$

Object-Oriented Cobol statements

From ITEM08DR:

```
61 200-process-user-request.
.
67 Evaluate aITEMHandle
68 When null
69 Display "Invalid ITEM number."
```

From ITEM08DA:

```
51 Method-ID. "close-file".
52 *>-----
53 Procedure Division.
54 Close ITEM-File
55 .
56 End Method "close-file".
```

Object-Z

Schema ITEM

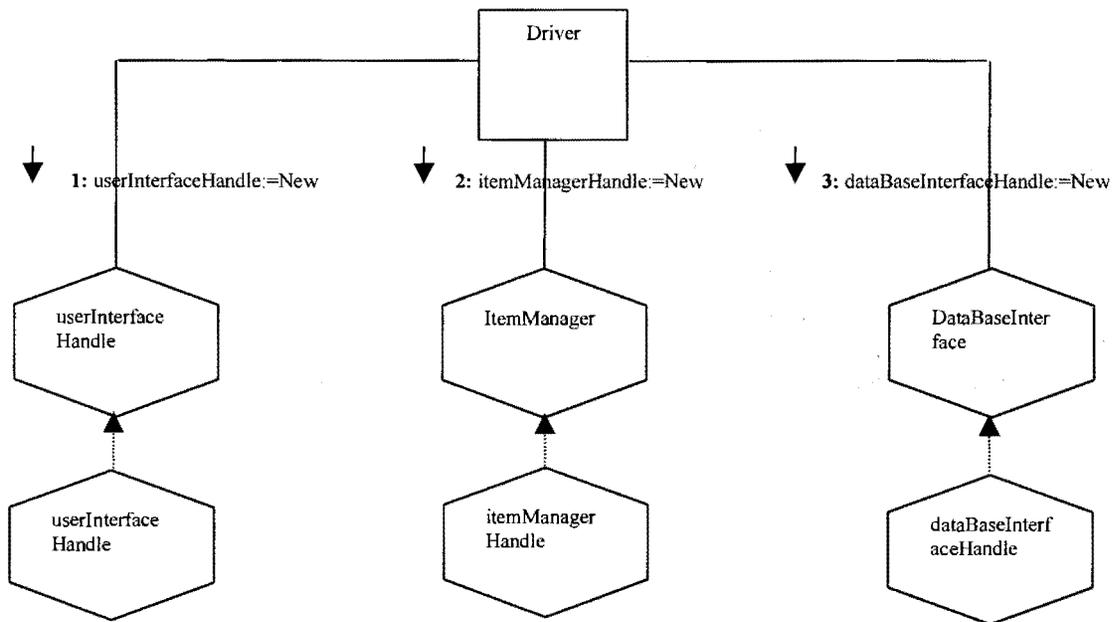
Object-Oriented Cobol statements: The same as given in Section 7.5.3.3.

7.5.4.4 Comparison

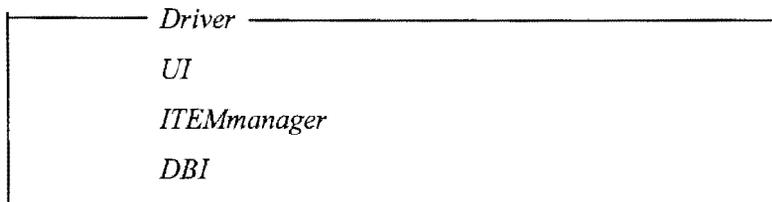
- The movement of the data between the classes is indicated more clearly in the UML specifications.
- The Object-Z specifications are more detailed.
- Because of the more detailed nature of Object-Z, easier implementation into Object-Oriented Cobol is facilitated.

7.5.5 Object creation by Driver

7.5.5.1 Specifications in UML



7.5.5.2 Specifications in Object-Z



The driver instantiates the Userinterface, the ITEMmanager and the DataBaseInterface. (Refer to the class diagram, the object message diagram and the set of class responsibilities and collaborators for the ITEM system).

7.5.5.3 Operation decomposition: implementation into Object-Oriented Cobol

For this operation decomposition please refer to Section 7.5.2.3.

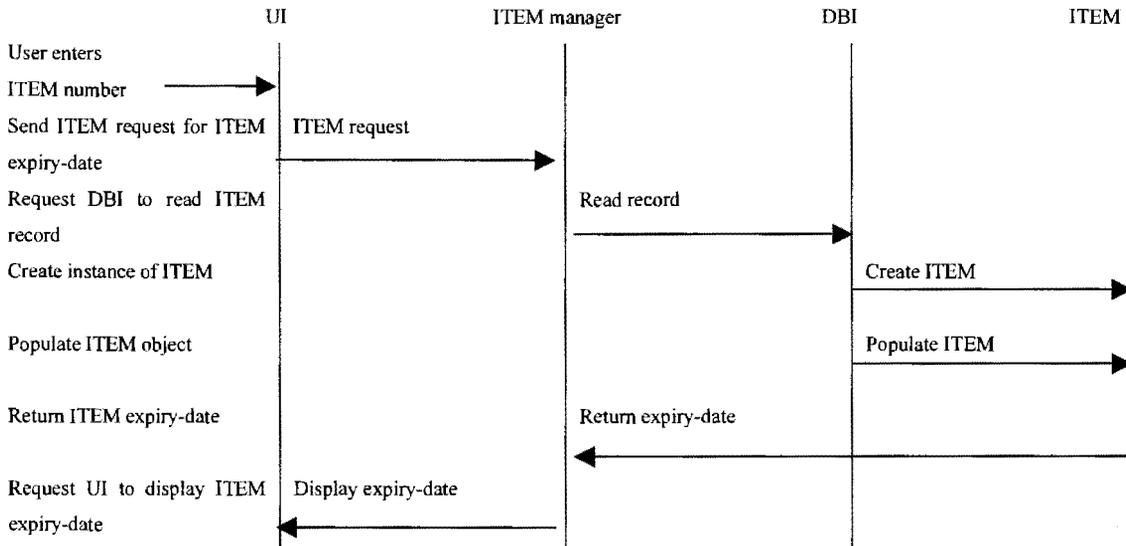
7.5.5.4 Comparison

- It is equally obvious from both the UML and the Object-Z specifications that the UserInterface (UI), the ITEMmanager and the DataBaseInterface (DBI) are instantiated by the Driver.
- The UML specifications add the extra detail of the UserInterfaceHandle, the ItemManagerHandle and the DataBaseInterfaceHandle.
- Because of the detailed nature of Object-Z, the implementation into Object-Oriented Cobol is facilitated.

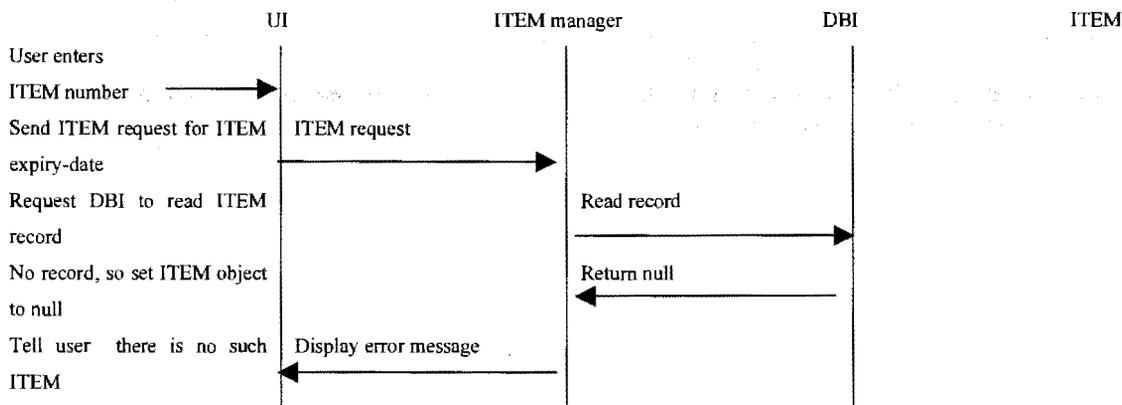
7.5.6 A message trace diagram (sequence diagram)

7.5.6.1 Specifications in UML

7.5.6.1.1 A message trace diagram: ITEM exists (UML)



7.5.6.1.2 A message trace diagram: ITEM does not exist (UML)



7.5.6.2 Specifications in Object-Z

UI

choice1? : CHOICE1
choice2? : CHOICE2
message! : RESPONSE
ITEMnumber : \mathbb{P} NUMBERS
ITEMmanager
number : \mathbb{N}

$(\text{choice1?} = \text{number} \wedge \text{number} \in \text{ITEMnumber} \wedge \text{choice2?} = s) \Rightarrow$
 $(n' = (\text{number} - 199) . \{\emptyset \text{ITEM}\} \wedge$
 $\text{message!} = \text{Standard_configuration_price: standardprice}' \wedge$
 $\text{message!} = \text{Special_configuration_price: specialprice}')$
 $(\text{choice1?} = \text{number} \wedge \text{number} \in \text{ITEMnumber} \wedge \text{choice2?} = d) \Rightarrow$
 $(n' = (\text{number} - 199) . \{\emptyset \text{ITEM}\} \wedge$
 $\text{message!} = \text{Standard_configuration_price: standardprice}' \wedge$
 $\text{message!} = \text{Special_configuration_price: specialprice}')$
 $(\text{choice1?} = \text{number} \wedge \text{number} \in \text{ITEMnumber} \wedge \text{choice2?} = c) \Rightarrow$
 $(n' = (\text{number} - 199) . \{\emptyset \text{ITEM}\} \wedge$
 $\text{message!} = \text{ITEM_expiry_date: expiry-date}')$
 $(\text{choice1?} = \text{number} \wedge \text{number} = 0) \Rightarrow$
 $(\text{message!} = \text{Processing_complete})$
 $(\text{choice1?} = \text{number} \wedge \text{number} \notin \text{ITEMnumber}) \Rightarrow$
 $(\text{message!} = \text{Invalid_ITEM_number})$

The user is prompted for a ITEM number. If the ITEM number is 0, the processing stops. The relevant record from the database, that is the indexed file ITEM.DI, is read by the DBI (DataBaseInterface). If the ITEM number is >109 or <100, it is an invalid ITEM number. If the ITEM number is valid, and if the user chooses the standard or discounted prices, the *ITEMmanager* is invoked.

ITEM

 Δ DBI

$(\text{choice2?} = s \wedge \text{number} \in \text{ITEMnumber}) \Rightarrow (n' = (\text{number} - 199)) \wedge$
 $(\text{standardprice}' = \text{sprice}(n') \wedge \text{specialprice}' = \text{pprice}(n'))$
 $(\text{choice2?} = d \wedge \text{number} \in \text{ITEMnumber}) \Rightarrow (n' = (\text{number} - 199)) \wedge$
 $(\text{discount}' = (1 - \text{ddiscount}(n'))$
 $\text{standardprice}' = \text{sprice}(n') * \text{discount}' \wedge$
 $\text{specialprice}' = \text{pprice}(n') * \text{discount}'$
 $(\text{choice2?} = e \wedge \text{number} \in \text{ITEMnumber}) \Rightarrow (n' = (\text{number} - 199)) \wedge$
 $(\text{expiry-date}' = \text{cexpiry-date}(n'))$

7.5.6.3 Operation decomposition: implementation into Object-Oriented Cobol

For this operation decomposition please refer to sections 7.5.3.3 and 7.5.4.3.

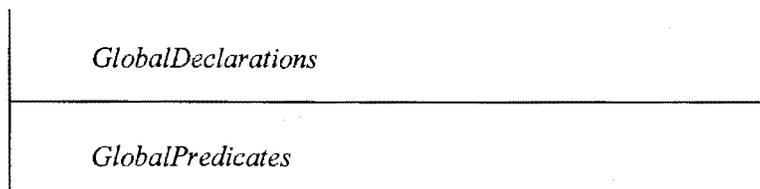
7.5.6.4 Comparison

- The movement of data and messages between the UI, ITEMmanager, DBI and the ITEM classes are easier to follow in the UML specifications.
- The detailed nature of the Object-Z specifications facilitates easier implementation into Object-Oriented Cobol.

7.6 Validation and verification of the Object-Z diagrams

7.6.1 Verifying consistency of global definitions

Refer to the text *GlobalDeclarations* | *GlobalPredicates* or the axiomatic description



It must be established that $\vdash \exists \textit{GlobalDeclarations} \bullet \textit{GlobalPredicates}$ which means that there exist values for *GlobalDeclarations* which satisfy predicate *GlobalPredicates* [Rat94].

For the schema *DBI* :

```

ITEMnumber: {200,201,202,203,204,205,206,207,208,209}
n: {1,2,3,4,5,6,7,8,9,10}
standardprice: {225,240,255,280,305,340,350,375,380,400}
specialprice: {385,405,425,465,505,565,610,615,625,655}
discount: {.05,.05,.05,.05,.05,.05,.07,.07,.10,.10}
expiry-date:
{130899,241000,250499,180901,141200,091199,121199,150599,130899,151200}
sprice: {305}
ddiscount: {.05}
cexpiry-date: (141200)
pprice: {505}

```

```

#dom ITEMnumber = 10 = #ran standardprice;
                = #ran discount;
                = #ran expiry_date;
                = #ran special_price
record = 20430550505141200

```

For the *ITEM* schema:

ΔDBI

```

(choice2 = s ∧ number = 204 ∈ {200,201,202,203,204,205,206,207,208,209})
⇒ (n' = (number - 199)) = 5 ∧
standardprice' = sprice(n') = 305 ∧ specialprice' = pprice(n') = 505)
(choice2 = d ∧ number = 204 ∈ {200,201,202,203,204,205,206,207,208,209})
⇒ (n' = (number - 199)) = 5 ∧
   (discount' = (1 - ddiscount(n')) = 1 - 0.05 = 0.95
   standardprice' = sprice(n') * discount' = 305 * 0.95 = 289.75 ∧
   specialprice' = pprice(n') * discount' = 505 * 0.95 = 479.75)
(choice2 = e ∧ number = 204 ∈ {200,201,202,203,204,205,206,207,208,209})
⇒ (n' = (number - 199)) = 5 ∧
(expiry-date' = cexpiry-date(n') = 141200)

```

For the *UI* schema:

```

choice1 : 204
choice2 : s
ITEMmanager

```

```

choice1 = 204 ∈ {200,201,202,203,204,205,206,207,208,209} ∧ choice2 = 5)
⇒ n' = 5 ∧
message! = Standard_configuration_price: 305 ∧
message! = Special_configuration_price: 505

```

Invalid item:

$choice1 : 232$ $choice2 : d$ $ITEMmanager$

$choice1 = 232 \notin \{200,201,202,203,204,205,206,207,208,209\} \wedge choice2 = 5$ $\Rightarrow message! = Invalid_ITEM_number$

For the *ITEMmanager* schema:

ΔDBI

UI

$record' = (20430550505141200)$

7.6.2 Verifying consistency of state models

A check must be done to ensure that the state model is consistent. This check can be expressed as a theorem which has the following general form:

$$\vdash \exists State' \bullet InitStateSchema$$

which can be extended to

$$\vdash \exists State'; Inputs? \bullet InitStateSchema$$

InitState: Initial state schema

if the initial state schema *InitStateSchema* has input variables.

This theorem states that 'There is a state of the general model (and inputs if *InitStateSchema* has any) that satisfies the initial state description.'. Proving this theorem establishes consistency between *State* and *InitStateSchema* and consistency of *State* itself in the sense that it describes a non-empty state space, i.e. its predicate does not collapse to *false* [Rat94].

For the *InitStateSchema (Driver)* schema:

<i>Driver</i> <i>UI</i> <i>ITEMmanager</i> <i>DBI</i>
--

This theorem is true, because it is true for *UI*, *ITEMmanager*, and *DBI*. which can be verified by inspection.

The initial state for *DBI*:

<i>InitDBI</i> <i>DBI'</i>
$\#dom\ ITEMnumber' = \#ran\ standardprice'; \#ran\ discount'; \#ran\ expiry_date'; \#ran\ special_price' = \emptyset$ $record' = \emptyset RECORD = \emptyset$

A state of the general model:

<i>DBI1</i> <i>ITEMnumber</i> : $\mathbb{P}\ NUMBER$ <i>n</i> : $\mathbb{P}\ NUMBER^2$ <i>standardprice</i> : $\mathbb{P}\ SPRICE$ <i>specialprice</i> : $\mathbb{P}\ PPRICE$ <i>discount</i> : $\mathbb{P}\ DISCOUNT$ <i>expiry-date</i> : $\mathbb{P}\ EXPIRY-DATE$ <i>sprice</i> : $NUMBER \rightarrow SPRICE$ <i>ddiscount</i> : $NUMBER \rightarrow DISCOUNT$ <i>cexpiry-date</i> : $NUMBER \rightarrow EXPIRY-DATE$ <i>pprice</i> : $NUMBER \rightarrow PPRICE$ <i>record</i> : $\mathbb{P}\ RECORD$
$\#dom\ ITEMnumber = \#ran\ standardprice; \#ran\ discount; \#ran\ expiry_date; \#ran\ special_price$ $record = \emptyset RECORD$

Assume that all the sets contain only zeroes:

$$NUMBER = \{0\}$$

$$SPRICE = \{0\}$$

$$PPRICE = \{0\}$$
$$DISCOUNT = \{0\}$$
$$EXPIRY-DATE = \{0\}$$
$$NUMBERS2 = \{0\}$$

Then by substitution the *InitDBI* schema is satisfied. Therefore

$$\vdash \exists \textit{State}; \textit{Inputs?} \bullet \textit{InitStateSchema}, \text{ that is}$$
$$\vdash \exists \textit{DBII}; \textit{Inputs?} \bullet \textit{InitDBI}$$

7.6.3 Verifying consistency of operations

Because the verification of the consistency of the global definitions also included the operation schemas, the consistency of the operations has also been proven. Therefore

$\vdash \exists \textit{OperationDeclarations} \bullet \textit{OperationPredicates}$, that is, the operation's predicates satisfies the operation's declarations.

7.7 Summary and Conclusion

Object-oriented modeling and design promote better understanding of requirements, cleaner designs, and more maintainable systems [Rum91, Pri97].

The ITEM system was discussed, from its requirements in English, through the specification in UML and Object-Z, into implementation into Object-Oriented Cobol. The Cobol programs were compiled using the Micro Focus Cobol compiler. Test data was used to test the system, and the output results were displayed.

From this chapter we conclude that the UML specifications are more graphic and easier to read than the Object-Z specifications. However, the Object-Z specifications are closer to the implementation language and a more direct implementation from Object-Z to Object-Oriented Cobol was possible. The direct implementation from the UML to the Object-Oriented Cobol is not so easy and straightforward. Therefore, both the UML and the Object-Z have each their own advantages and disadvantages. Once again, the combination of these two specification modelling techniques will provide the best specification modelling - this can also be concluded from the articles by Jackson [Jac95], Kronlöf [Kro93], Polack, Wiston & Mander [Pol93], Semmens [Sem92], MetaPHOR [Met94], Paige [Pai97], [Pai97b], [Pai99], [Pai99b], Sacki [Sae98], Zave [Zav93] and Hall [Hal96].

Chapter 8

Summary and Conclusion

In this chapter a brief summary is given of the previous seven chapters, and the achievements and conclusions of these chapters are investigated. Possibilities of future research are explored.

8.1 Introduction

Chapter 1 formulated three research questions based on the research problem:

1. *How systems are transformed (refined) from specification methods in Z, Object-Z and UML to implementation in implementation languages such as C, C++, and Object-Oriented Cobol.*
2. *What the differences (strengths and weaknesses) are between Z, Object-Z, and UML for the specification and refinement of systems that are implemented into non-object-oriented and object-oriented implementation languages.*

Chapter 8 also looks back on the previous seven chapters and reappraise the results and conclusions that have been reached. A brief summary of the previous seven chapters are given, and we look at what has been achieved. From the study certain possibilities for future research have emerged, which are put under the spotlight.

8.2 Brief Summary

A brief summary on what was discussed and investigated in each chapter follows:

Chapter 1 introduced the problem statements and aims of this study. Included were the method of study, the specific questions (problems) investigated as well as an overview of all eight chapters that constitute the body of the study.

In Chapter 2 we examined the software systems development life cycle for both the structured and object-oriented paradigms. A comparison of the life cycle models was made and an overview of the life cycle development phases was given. The position of refinement within the software systems development life cycle was indicated.

The process of refinement from specification to implementation was demonstrated using Z as the specification language for non-object-oriented designs in Chapter 3. Prior to the refinement we discussed the validation and verification of the specifications.

In Chapter 4 we presented an overview of UML, followed by the transformation of a design into an implementation for non-object-oriented languages, namely C and Cobol.

A comparative study between Z and UML applications from specification through refinement to implementation for non-object-oriented implementation languages (in this case C and Cobol) was made in Chapter 5.

Object-Z and C++ were introduced in Chapter 6. In a background study on Object-Z and C++ some of the main features of these two languages were described. For various aspects of design, Object-Z and UML were compared for the specification, refinement and implementation into an object-oriented language (C++).

The software systems development life cycle of Chapter 2 was revisited in Chapter 7 in the form of a system, the ITEM system, where this system was developed from the requirements phase through to the implementation phase. Specification languages used were Object-Z and UML, while the implementation language was Object-Oriented Cobol.

This study was concluded in Chapter 8 by referring to Chapter 1 and stating the major findings and conclusions that could be drawn as a result of this study. The research objectives were re-appraised. Suggestions for future research were offered.

Three appendices are included. Appendix A summarises some of the features of the Z notation. Appendix B describes some UML detail. Appendix C contains the Object-Oriented Cobol programs for Chapter 7.

8.3 What has been achieved?

From this study the importance of the refinement stage in the software systems development cycle has been demonstrated. This is because both the specification languages and the implementation languages are becoming more and more sophisticated. Just as fast is the development of systems development tools that aid in the design, specification and implementation of software systems. This is a dynamic field of study that demands constant re-evaluation and updating.

It has also emerged, as confirmed by Jacky [Jak97], that in a single project, the programming problems that arise present a range of difficulty. Therefore, a large percentage of the project may be so routine, that no formal description other than the code itself is required. Only a part of the project will require specification in a formal notation such as Z, and from this part, only a fraction might be refined to a detailed design in Z. Of this fraction, only a page or two of code might be derived and verified. The rest of the project, because it is so obvious, can be translated to code by intuition and then verified by inspection.

The usual strengths and weaknesses of both Z (Object-Z) and UML are demonstrated in this study, but beyond that, as can be seen in chapters 5 and 6, even though UML does not have a refinement process such as Z, different UML diagrams e.g. the sequence and collaboration diagrams, can be used to simulate a refinement process, where the finer detail of the specification increases towards an implementation. Therefore, even in the refinement process, for both Z and UML, each in its own way, have their strengths and weaknesses.

Because of the difference in notation of Z and UML, where Z is more mathematical and UML more graphic, it became more and more apparent that a combination of Z and UML as a specification language will obviate the disadvantages and combine the advantages of both the specification languages.

In the study it has also been shown that UML can be used for both structured and object-oriented designed systems. Z is usually used for systems designed in the structured paradigm, and Object-Z for systems designed in the object-oriented paradigm.

The ITEM system in Chapter 7 demonstrates the application of all the techniques described in the previous chapters, and how they contribute to the eventual implementation of a system that is working correctly according to the original specifications in English.

8.4 Conclusions

The conclusions reached during the research project suggest the following solutions/answers for these questions.

- (1) **The transformation (refinement) of systems from specification methods in Z, Object-Z and UML to implementation in implementation languages such as C, C++, and**

Object-Oriented Cobol. In chapters 3, 4, 5 and 6 this transformation process was extensively looked at. Chapter 3 gives a detailed refinement process from specifications in Z through to implementation into C and Cobol. From the following three chapters it can be seen that UML does not have an equivalent refinement process such as Z, but by using and extending existing UML diagrams that include more and more detail that will facilitate the implementation process, a refinement process can be simulated. Chapter 7 gives an example of a system that is specified in both Object-Z and UML, refined and implemented into Object-Oriented Cobol. This ITEM system includes all the stages of the system development process, as outlined in Chapter 2.

(2) **The differences (strengths and weaknesses) between Z, Object-Z, and UML for the specification and refinement of systems that are implemented into non-object-oriented and object-oriented implementation languages.** From the discussions of the chapters it emerged that both Z and UML have each got their own unique strengths and weaknesses. Often the choice as to which specification language to choose is dependent on the system involved, the people involved, whether it is object-oriented or not, and the complication level of the system. It can't be concluded categorically that the one method is preferable to the other. This study however, gives an extensive insight into the versatility of both the specification languages involved.

8.5 Future Research

From this study, we believe that future research can be directed in a number of directions, for example: By using and extending existing UML diagrams, a UML refinement process can be simulated that can be considered an extension of UML towards the emerging of implementation language code.

Secondly, with the combination of specifications in Z and UML, all the advantages, including the extensive Z refinement process, of these two specification languages can be incorporated.

Research in that direction has already begun: Because of the complexity of problems being solved, it has been suggested that a single software development method is insufficient for all situations [Jac95]. Method integration (combining specific methods), is a technique that can be used to aid in multiple-method use [Kro93, Plo93, Sem92, Met94, Pai97, Sae98].

A heterogeneous specification is composed from parts written in two or more different notations [Sem92, Zav93, Hal96]. Heterogeneous notations are ideally applied where the

complimentarity of the separate notations is evident, and when the notations being combined can be used together without significant alteration.

To formalize UML specifications in predicative notation, a simple heterogeneous basis is constructed consisting of selected UML constructs and predicative notation.

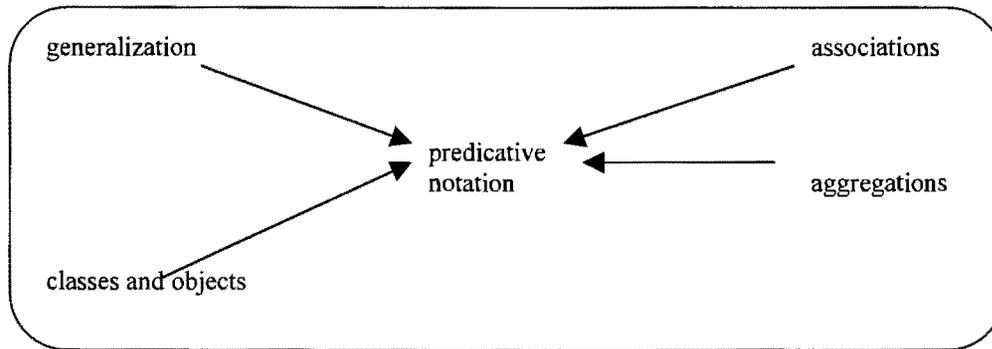


Figure 8.1 The heterogeneous basis with UML concepts

Predicative programming [Heh93] is a program design calculus in which programs are specifications. Programs and specifications are predicates on pre- and post states. As in Z, final values of variables are annotated with a prime, initial values of variables are undecorated.

To use formal methods in cooperation with other methods requires a formal semantics for all specifications that are produced, otherwise proof rules, *refinement* and other formal analyses cannot be carried out with full rigour [Pai99].

To *refine* specifications by parts over a specification combinator requires a proof that *refinement* is monotonic over the combinator. Without a formal semantics for a heterogeneous specification such a proof cannot be carried out.

Integrating predicative programming with *OOM* demonstrates how an object-oriented method can be extended to work with a formal method. It shows how a program design calculus can be made more accessible and attractive to developers through its combination with an object-oriented method, which allows restrictable application of the formal method's *refinement techniques*. It provides an example of how to scale *refinement-based methods* up to deal with large-scale problems, by integrating them with object-oriented techniques.

For the integration, a subset of UML is chosen as the notation for *OOM* because it is becoming a standard object-oriented modeling notation and is process independent. Predicative programming is chosen because it is general and *refinement rules* are powerful and simple enough to prove useful theorems about specifications.

Predicative programming and *OOM* are complementary methods [Pai99]. Complementarity of methods can be explained in terms of notation and process. UML notations are visual, while predicative notations are text-based.

As another example, refer to the Object Constraint Language, the precise modelling with UML. OCL complements the UML by providing a language for formally expressing the constraints of a model, a facility useful in user models as well as in the definition of the UML itself [War99].

The Object Constraint Language (OCL) is a new notational language, a subset of the industry standard Unified Modeling Language, that allows software developers to write constraints over object models. These constraints are particularly useful, as they allow a developer to create a highly specific set of rules that governs the aspect of an individual object. As many software projects today require unique and complex rules that are written specifically for business models, OCL is fast becoming an integral facet of object development [War99].

What we are advocating in this study, is a specific combination of *Z* and UML, which will also have the advantages as that described for OCL (object constraint language) and the more general predicative programming, but will have the added advantage that *Z* is a well known and proven predicative specification language.

8.6 Conclusion

In conclusion we hope that this comparative study has brought the versatility of both *Z* and UML to the fore, with the prediction that their potential and range can be extended even more, to include for example refinement in UML, and a specification language that includes both *Z* and UML.

Appendix A

Z notation

For a comprehensive introduction to Z, refer to [Spi92].

TYPES

$[T]$	- definition of T as a given type
$x:T$	- declaration of x as a variable of type T
$x_1, x_2, \dots, x_n:T$	- compound declaration of multiple variables of type T
$x_1:T_1; \dots; x_m:T_m$	- compound declaration of variables of differing types

SETS

\emptyset	- the empty set
P	- powerset constructor: all sets that are subsets of...
PS	- the set of subsets of S
FS	- the set of finite subsets of S
\cup, \cap, \setminus	- set union, intersection and difference
$S_1 \times S_2$	- cartesian product
$t \in S$	- set membership, t is a member of S
$t \notin S$	- non membership, i.e. $\neg(t \in S)$
$S_1 \subseteq S_2$	- Subset: each element of S_1 is also a member of S_2

RELATIONS AND FUNCTIONS

A relation between two sets is represented in Z by a subset of the cartesian product of the two sets, i.e. by a set of pairs. Any operation that can apply to a set can also be applied to a relation.

(x,y)	- a pair drawn from the cartesian product $X \times Y$
$X \leftrightarrow Y$	- the set of relations between X and Y (equivalent to $P(X \times Y)$)
$X \rightarrow Y$	- the set of total functions from X to Y
$X \rightharpoonup Y$	- the set of partial functions from X to Y
$F(x)$	- function application, the (unique) y such that $(x,y) \in f$
$f \oplus g$	- function overriding, $f \oplus g(x) = g(x)$ if $x \in \text{dom } g$ or $f(x)$ otherwise
$R \triangleright S$	- range subtraction, $R \triangleright S = \{x: X, y: Y \mid (x,y) \in R \wedge y \notin S\}$

SEQUENCES

$\text{Seq } X$	- the set of sequences over the set X
$\langle \rangle$	- the set of sequences over the set X
$s \wedge t$	- concatenation: $\langle x_1, \dots, x_n \rangle \wedge \langle y_1, \dots, y_m \rangle = \langle x_1, \dots, x_n, y_1, \dots, y_m \rangle$

PREDICATES

D is a declaration and P and Q are predicates.

$\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$	- standard logical connectives
\exists, \forall	- existential and universal quantification
$\forall D \mid P \bullet Q$	- for all values of the variables declared in D
$\exists D \mid P \bullet Q$	- there exists some values for the variables in D such that P is true and it is also the case that Q is true

A single quantifier \exists or \forall is permitted for the outer scope of each predicate. Thus

$\forall x: X: y: Y$ universally quantifies both x and y . If the predicate P is empty then it is assumed to be *true*, i.e.

$$\forall D \bullet = \forall D \mid \text{true} \bullet Q$$

$$\exists D \bullet = \exists D \mid \text{true} \bullet Q$$

The symbol \uparrow denotes a sequence filtering: For a sequence s and a set of values v , $s \uparrow v$ creates a new sequence that contains precisely those entries in sequence s that are elements of set v , and in the same order. For example: $\langle a, b, c, d, e \rangle \uparrow \{b, d, f\} = \langle b, d \rangle$ [Bar94]. The function $\text{ran } a$ denotes the range of a relation.

SCHEMAS

θ Z schema binding expression. Whenever a value is written in Z , it must belong to some type, i.e., be a member of the set which is its type. When a schema is written, a new type is declared. Values of that type are called bindings, and contain a (name, value) pair corresponding to each component of the corresponding schema. The purpose of θ is to construct a binding value. The schema that is referenced by the argument to θ determines the names from which the binding is to be constructed. However, the values to be

associated with these names are taken not from the referenced schema but from the declarations of those names that are in scope in the context in which the θ is used [Sch93].

Φ Commonly used name prefix convention for framing schemas.

\equiv This symbol is used when a new schema is defined, either by giving an explicit schema text or by an expression involving other schemas and schema operators [Pot96].

OPERATORS

- Minus (subtraction)
- * Multiplication

Appendix B

UML – Past, Present and Future

The UML was developed by Rational Software and its partners. The development of UML began in October of 1994 when Grady Booch and Jim Rumbaugh of Rational software Corporation began their work to unify the Booch and OMT (Object Modelling Technique) methods [OMG99]. The Booch and OMT methods were already independently growing together and were collectively recognized as leading object-oriented methods worldwide, Booch and Rumbaugh joined forces to forge a complete unification of their work. A draft version 0.8 of the (then called) Unified Method, was released in October of 1995. Ivar Jacobson and his Objectory company joined Rational and this unification effort in the fall of 1995, merging in the OOSE (Object-Oriented Software Engineering) method. As the primary authors of the Booch, OMT, and OOSE methods, Grady Booch, Jim Rumbaugh, and IVar Jacobson were motivated to create a unified modeling language [OMG99].

By 1996, several organizations saw UML as strategic to their business. A Request for Proposal (RFP) issued by the Object Management Group (OMG) provided the catalyst for these organisations to join forces around producing a joint RFP response. Rational established the UML Partners consortium with several organizations that were willing to dedicate resources to work toward a strong UML definition. Those contributing most to the UML definition included: Digital equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, and Unisys. UML was produced by this collaboration. UML is a modeling language that is well defined, expressive, powerful, and generally applicable [OMG99].

In January 1997 IBM & ObjecTime; Platinum Technology; Ptech; Taskon & Reich Technologies; and Softeam also submitted separate RFP responses to the OMG. These companies joined the UML partners to contribute their ideas, and together the partners produced the revised UML 1.1 response [OMG99].

After the UML 1.1 proposal was adopted by the OMG membership in November 1997, the OMG chartered a revision task force (RTF) to accept comments from the general public and to make

revisions to the specifications for the handling of bugs, inconsistencies, ambiguities, and minor omissions that could be handled without a major change in scope from the original proposal. The RTF issued a final report containing UML 1.3 due for the second quarter of 1999. In the big picture this version should be considered a minor upgrade to the original proposal [OMG99].

Although the UML defines a precise language, it is not a barrier to future improvements in modeling concepts. Many leading-edge techniques have been addressed, but expect additional techniques may influence future versions of the UML [OMG99].

Standardisation of the UML

Many organizations have already endorsed the UML as their organisation's standard, since it is based on the modeling languages of leading object methods. The Unified Modelling Language version 1.1 specification was added to the list of OMG Adopted Technologies in November 1997. Since then the OMG has assumed responsibility for the further development of the UML standard [OMG99].

All the UML examples used in this dissertation comply with the UML version 1.1 standard even though some dates back as far as 1991.

Focus of UML Cooperation

The UML Partners contributed a variety of expert perspectives, including, but not limited to the following: OMG (Object Management Group) and RM-ODP technology perspectives, business modeling, constraint language, state machine semantics, types, interfaces, components, collaborations, refinement, frameworks, distribution, and metamodel. UML 1.1 is the result of a collaborative team effort. The UML partners have worked hard as a team to define UML 1.0 and 1.1. While each partner came in with their own perspective and areas of interest, the result has benefited from each of them and from the diversity of their experiences [Fow97].

The contribution of the different partners can be defined as follows:

Hewlett-Packard provided input on the relationship between UML models and reuse issues and in the use of packages and “facades” to facilitate the construction of reusable component-based application frameworks. They advocate a layered UML structure and mechanisms to define extensible, modular method subsets for HP Fusion, reuse, and domain-specific methods. They

also contributed to how inter-model relationships are modeled. They also focused on the use of patterns and the relationship to CORBA services. (See www.hp.com, [Mal96, Gri96].)

IBM's primary contribution to the UML is the Object Constraint Language (OCL). OCL was developed at IBM as a language for business modeling within IBM and is derived from the Syntropy method. It is used within UML both to help formalize the semantics of the language itself and to provide a facility for UML users to express precise constraints on the structure of models. IBM's contributions to UML have also included fundamental concepts in the semantics of refinement and templates. (See www.ibm.com, [Coo94])

i-Logix contributed with expertise in the definition, semantics, and use of executable behavior and the use of events and signals within the UML. The UML incorporates Harel statecharts to provide a hierarchical specification of concurrent behavior. i-Logix also focused strongly on the relation between object and behavioral models in UML. (See www.ilogix.com, [Har87, [Har96a, Har96b].)

ICON Computing contributed in the area of precise behavior modeling of component and framework-based systems, with a clear definition of abstraction and refinement from business to code. Their primary technical areas in UML have been types, behavior specifications, refinement, collaborations, and composition of reusable frameworks, adapted from the Catalysis method. (See www.iconcomp.com, [D'So97a, D'So97b].)

IntelliCorp contributed to business modeling aspects of the UML, activity diagrams and object flow in particular, as well as formalization for the concept of roles, and other aspects of object and dynamic modeling. (See www.intellicorp.com, [Mar95, Boc94].)

MCI Systemhouse - As a leading systems integrator MCI Systemhouse has worked to ensure that the UML is applicable to a wide range of application domains. Their expertise in distributed object systems has made the UML more scalable and better able to address issues of distribution and concurrency. MCI Systemhouse played an important role in defining the metamodel and the glossary, especially in their leadership of a semantics task force during the UML 1.1 phase. They also assisted in aligning the UML with other OMG standards and RM-ODP. (See www.systemhouse.mci.com.)

Microsoft provided expertise with the issues of building component-based systems, including modeling components, their interfaces and their distribution. They have also focused on the relationship between UML and standards such as ActiveX and COM and use the UML with their repository technology.. (See www.microsoft.com.)

ObjecTime contributed in the areas of formal specification, extensibility, and behavior. They played a key role in definitions for state machines, common behavior, role modeling, and refinement. They also contributed to the RM-ODP comparison. (See www.objectime.com.)

Oracle helped in the definition and support for modeling business processes and for describing business models in UML. They focused on support for workflow descriptions and activity diagrams as well as business objects, and have prepared stereotypes for tailoring the UML for business modeling. (See www.oracle.com, [Ram95, Ram96].)

Platinum Technology contributed in the areas of extension mechanisms, alignment with the MetaObject Facility, metamodeling, CDIF perspectives, and tool interoperability. (See www.platinum.com.)

Ptech contributed expertise in metamodels, distributed systems, and other topics. (See www.ptechinc.com.)

Rational Software defined the original UML and led the UML 1.0 and 1.1 projects, technically and managerially. Rational's diverse experience in object-oriented, component-based, and visual modeling technology has contributed greatly to the UML. (See www.rational.com/uml, [Boo97, Rum97, Jac97].)

Reich Technologies and **Taskon** contributed their expertise on collaborations and role modeling. (See www.sn.no.)

Softeam provided detailed reviews of the UML during its evolution. (See www.softeam.fr.)

Sterling Software contributed with their expertise on the modeling of components and types. They focused on type models and specifications, on business modeling, and on the relationship of

the UML definition to standards. Texas Instruments Software, a UML Partner, was acquired by Sterling Software during the UML 1.1 definition phase. (See www.sterling.com.)

Unisys has a strong interest in the meta-metamodels and their relationship to the UML, including the formalization of relationships and constraints at the meta-level and meta-meta-level consistently. Within the UML proposal they have particularly focused on the integration of the UML and the OMG's Meta-Object Facility and CORBA IDL. They were instrumental in the IDL generation for the UML CORBA facility. (See www.unisys.com.)

Appendix C

Object-Oriented Cobol programs of the ITEM system compiled and run using the Micro Focus Cobol compiler

The terms below are described by Price [Pri97]:

Program. A conventional Cobol program; it includes a Program-ID paragraph in the Identification Division.

Class program or Class definition. The source unit that defines a class. It includes a Class-ID paragraph in the Identification Division.

Object definition. That portion of code in the class program between the reserved words OBJECT and END OBJECT.

Object data. The data definitions of the Object-Storage Section immediately following the OBJECT header and preceding the method definitions.

Object, object instance or instance. The data definitions created by instantiating an object. This data is accessible only through the methods of the class (and inherited classes) from which the object was created. These methods are commonly considered to be part of the object.

Source unit. A complete program, whether a conventional program or class definition.

There are three Cobol programs:

1. ITEM08DR.CBL. This is the driver program. This is a conventional program that begins the process by instantiating classes creating objects necessary for the life of the run. Also, the program distributes object handles thereby providing for communication between objects (establishes needed associations between objects). The item number is obtained from the user, and the user is queried for menu choice. The requested item number is processed. Messages and item data are displayed.

2. ITEM05CL.CBL. This is the ItemClass program. The user request is processed. The item object is populated. The discounted item prices are calculated and the data returned to the driver program.
3. ITEM08DA.CBL. This is the database interface for the ITEM system. The desired record is read from the indexed file ITEM2.DAT. The item object is created and populated.

Creating a project

A project is created using the Micro Focus compiler. This project will then contain the three programs and the indexed file for the ITEM system. After the project has been created, the programs of the ITEM project are loaded into the created project. (Refer to a Micro Focus (Personal COBOL for Windows with Object Orientation) manual). The screen will, for example, look as follows:

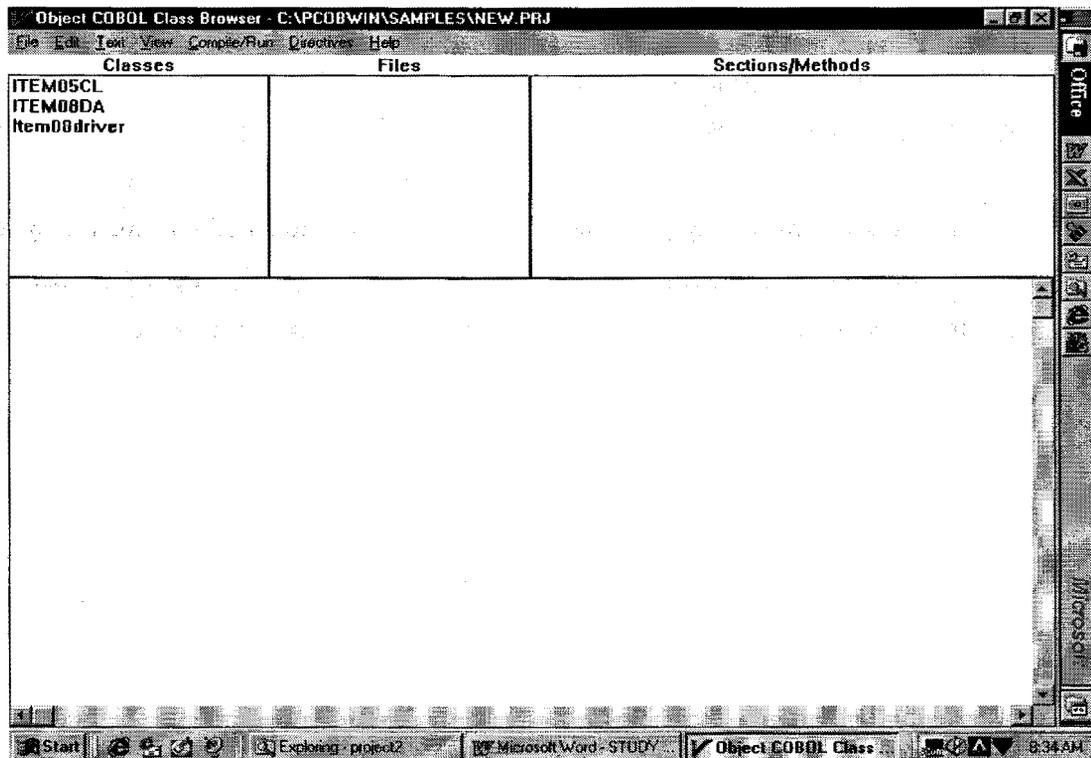


Figure C.1 Programs of the ITEM project

After compilation for each of the three programs the following messages appear:

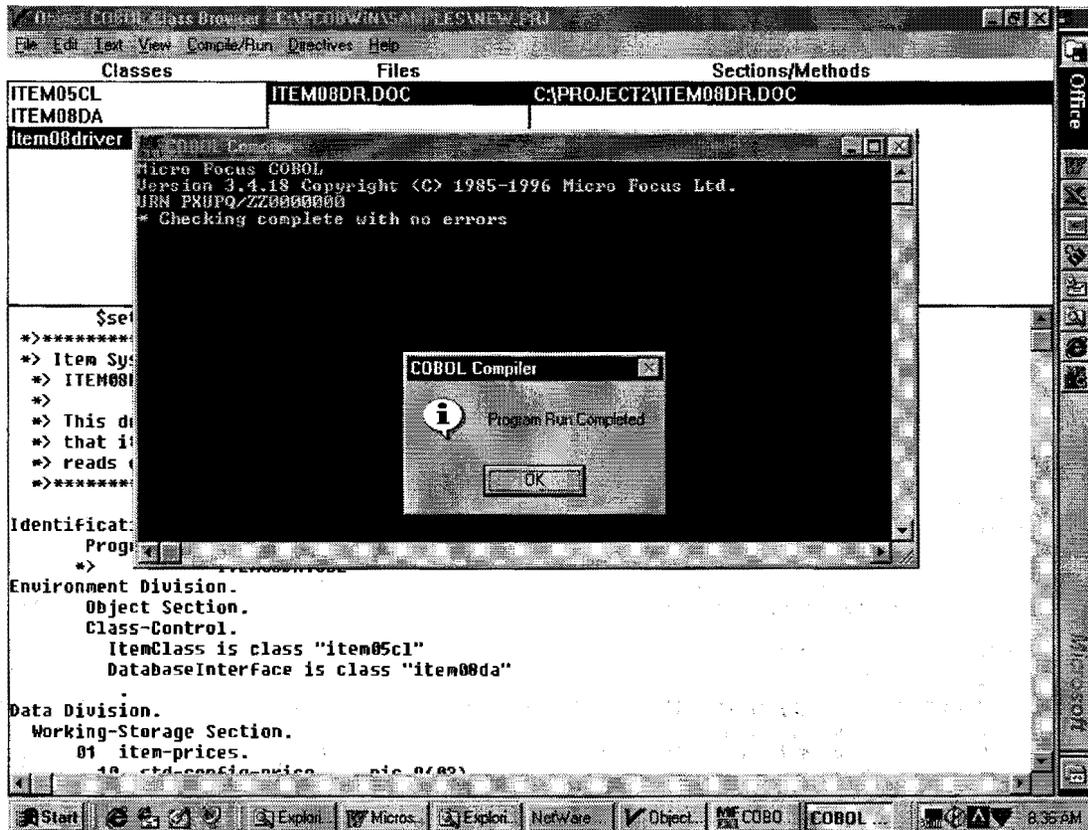


Figure C.2 Compiled program messages

After the programs have been compiled error free, the project is run using item numbers 200, 201 and 202.

Output:

```

Object COBOL Class Project - A... PRJ
COBOL Run Time Shell
Your choice <E,S,D>? s
Standard configuration price: 248
Special configuration price: 485
Item number <0 to terminate>? 202
The options are:
E - Item expiry-date (default)
S - Standard item prices
D - Discounted item prices
Your choice <E,S,D>? d
Standard configuration price: 242
Special configuration price: 481
Item number <0 to terminate>? 230
Invalid item number.
Item number <0 to terminate>? 0
*****

Identification Division.
  Program-ID. Item08driver.
  *->      ITEM08DR.CBL
Environment Division.
  Object Section.
  Class-Control.
    ItemClass is class "item05cl"
    DatabaseInterface is class "item08da"

Data Division.
  Working-Storage Section.
    01 item-prices.
      10 std-config-price pic 9(02)
  
```

COBOL Run Time Shell

Program Run Completed

OK

Figure C.3 Program run completed

Creating an instance of a class in Object-Oriented Cobol

When the ITEM system is run (executed), the driver program (ITEM08DR) and the two class programs (ITEM05CL) and (ITEM08DA) are loaded into memory from the disk storage. The class programs are only templates of the classes and cannot itself be executed. During execution, the driver program must first create an instance of a class program. This instance of a class program can then be processed.

When the following statement is executed in the driver program (ITEM08DR)

```

Invoke DatabaseInterface "New"
    Returning theDBIHandle
  
```

(lines 42-43, ITEM08DR).

An instance (object) of a class (ITEM08DA), is created by invoking the method *New*. *New* is one of many methods available from the system class library. Access to the methods is through a class, named *Base*. Refer to lines 20-21 of ITEM05CL:

```
20    Class-ID. ItemClass
21          inherits from Base
```

and lines 15-16 of ITEM08DA:

```
15    Class-ID.    ItemDatabaseInterface
16          inherits from Base
```

Base includes methods to create and destroy instances of classes, handle errors, and perform many other basic functions. *New* returns the memory address of the newly created object (instance) in the designated object handle, *theDBIHandle*.

When the statement

```
81    Invoke ItemClass "New"
82          Returning ls-theItemHandle
```

(lines 81-82, ITEM08DA) is executed an instance of the class ITEM05CL is created and the memory address placed in the object handle *ls-theItemHandle*.

The driver program (ITEM08DA) invokes the methods "open-file" (line 44) and "read-item-record" (line 62), from the class instance of ITEM08DA. The class instance of ITEM08DA invokes the method "populate-the-item-object" (lines 83-84) from the class instance of ITEM05CL where the data is read from the file ITEM2.DAT and the item-data object (lines 30-39 from ITEM05CL) is loaded with data.

```
30    OBJECT.
31
32    Data Division.
33      Object-Storage Section.    *> OBJECT DATA
34      01  item-data.
35          10  item-number          pic 9(03).
36          10  std-config-price     pic 9(03).
37          10  spec-config-price    pic 9(03).
38          10  item-discount        pic v9(02).
39          10  item-expiry-date     pic 9(06).
40
```

The ITEM system is then further executed (refer to the programs), until an item number of 0 is entered, which terminates the processing.

Listings of compiled Object-Oriented Cobol programs and output of the ITEM system

ITEM08DR program

```

1  *>*****
2  *> Item System
3  *> ITEM08DR.CBL
4  *>
5  *> This is the driver program.
6  *> It designates the DBI ITEM08DA which
7  *> reads data from an indexed file.
8  *>*****
9
10 Identification Division.
11     Program-ID. Item08driver.
12     *>          ITEM08DR.CBL
13
14 Environment Division.
15     Object Section.
16     Class-Control.
17     ItemClass is class "item05c1"
18     DatabaseInterface is class "item08da"
19
20 Data Division.
21     Working-Storage Section.
22
23     01 item-prices.
24         10 std-config-price      pic 9(03).
25         10 spec-config-price     pic 9(03).
26     01 item-expiry-date         pic 9(06).
27
28     01 item-number              pic 9(03).
29         88 terminate-processing  value 0.
30
31     01 menu-choice              pic X(01).
32         88 discounted-prices    value "D" "d".
33         88 standard-prices      value "S" "s".
34
35     01 object-handles           object reference.
36         10 aItemHandle.
37         10 theDBIHandle.
38
39 Procedure Division.
40
41     000-process-item-data.
42     Invoke DatabaseInterface "New"
43         Returning theDBIHandle
44     Invoke theDBIHandle "open-file"
45     Perform with test after
46         until item-number = 0
47     Display " "
48     Display "Item number <0 to terminate>? "
49         with no advancing
50     Accept item-number
51     Evaluate item-number
52     When 0
53         Continue
54     When other
55         Perform 200-process-user-request
56     End-Evaluate *>item-number
57     End-Perform *>with test after
58     Invoke theDBIHandle "close-file"
59     Stop run
60
61     200-process-user-request.
62     Invoke theDBIHandle "read-item-record"
63         Using item-number
64         Returning aItemHandle
65     *> If the record was not found, the item

```

```

66     *> object handle is returned as null.
67     Evaluate aItemHandle
68         When null
69             Display "Invalid item number."
70         When other
71             Perform 250-get-menu-selection
72             Evaluate TRUE
73                 When discounted-prices
74                     Invoke aItemHandle
75                         "get-discounted-item-prices"
76                         Returning item-prices
77                 Perform 300-display-item-prices
78                 When standard-prices
79                     Invoke aItemHandle "get-item-prices"
80                         Returning item-prices
81                 Perform 300-display-item-prices
82                 When other
83                     Invoke aItemHandle "get-item-expiry-date"
84                         Returning item-expiry-date
85                 Perform 350-display-item-expiry-date
86             End-evaluate *>TRUE
87     End-evaluate *>aItemHandle
88     .
89     250-get-menu-selection.
90     Display " "
91     Display "The options are:"
92     Display " E - Item expiry-date (default)"
93     Display " S - Standard item prices"
94     Display " D - Discounted item prices"
95     Display " "
96     Display "Your choice <E,S,D>? " no advancing
97     Accept menu-choice
98     .
99     300-display-item-prices.
100    Display " "
101    Display "Standard configuration price: "
102                                std-config-price
103    Display "Special configuration price: "
104                                spec-config-price
105    .
106    350-display-item-expiry-date.
107    Display " "
108    Display "Item expiry-date: " item-expiry-date
109    .

```

ITEM05CL program

```

1  *>*****
2  *> Item System
3  *> ITEM05CL.CBL
4  *>
5  *> This is the basic item class
6  *> Object data is:
7  *>   item-number
8  *>   std-config-price   Standard config item price
9  *>   spec-config-price  Special config item price
10 *>   item-discount     Allowable item discount
11 *>   item-expiry-date   Item expiry-date
12 *> Methods are:
13 *>   get-discounted-item-prices
14 *>   get-item-prices
15 *>   get-item-expiry-date
16 *>   populate-the-item-object (invoked by DBI)
17 *>*****
18
19 Identification Division.
20     Class-id.   ItemClass
21     inherits from Base.
22
23 Environment Division.
24     Object Section.
25     Class-Control.
26     ItemClass is class "item05cl"

```

```

27         Base      is class "base"
28
29 *>=====
30 OBJECT.
31
32 Data Division.
33   Object-Storage Section.  *> OBJECT DATA
34   01 item-data.
35       10 item-number          pic 9(03).
36       10 std-config-price     pic 9(03).
37       10 spec-config-price    pic 9(03).
38       10 item-discount        pic v9(02).
39       10 item-expiry-date     pic 9(06).
40
41 Procedure Division.
42 *>-----<*
43 *> Object Methods <*
44 *>-----<*
45
46 *>-----
47 Method-id. "get-discounted-item-prices".
48 *>-----
49   Data Division.
50     Working-Storage Section.
51       01 price-factor          pic 9(01)v9(02).
52
53     Linkage Section.
54       01 ls-item-number        pic 9(03).
55       01 ls-item-data.
56         10 ls-std-config-price pic 9(03).
57         10 ls-spec-config-price pic 9(03).
58
59     Procedure Division Returning ls-item-data.
60       Move std-config-price
61         to ls-std-config-price
62       Move spec-config-price
63         to ls-spec-config-price
64       Subtract item-discount from 1.0
65         giving price-factor
66       Multiply price-factor
67         by ls-std-config-price
68       Multiply price-factor
69         by ls-spec-config-price
70
71   End Method "get-discounted-item-prices".
72
73 *>-----
74 Method-id. "get-item-prices".
75 *>-----
76   Data Division.
77     Linkage Section.
78       01 ls-item-number        pic 9(03).
79       01 ls-item-data.
80         10 ls-std-config-price pic 9(03).
81         10 ls-spec-config-price pic 9(03).
82
83     Procedure Division Returning ls-item-data.
84       Move std-config-price
85         to ls-std-config-price
86       Move spec-config-price
87         to ls-spec-config-price
88
89   End Method "get-item-prices".
90
91 *>-----
92 Method-id. "get-item-expiry-date".
93 *>-----
94   Data Division.
95     Linkage Section.
96       01 ls-item-number        pic 9(03).
97       01 ls-item-expiry-date   pic 9(06).
98
99     Procedure Division
100       Returning ls-item-expiry-date.
101       Move item-expiry-date to ls-item-expiry-date
102
103   End Method "get-item-expiry-date".

```

```

104
105 *>-----
106 Method-id. "populate-the-item-object".
107 *>-----
108 Data Division.
109 Linkage Section.
110 01 ls-item-data.
111 10 ls-item-number pic 9(03).
112 10 ls-std-config-price pic 9(03).
113 10 ls-spec-config-price pic 9(03).
114 10 ls-item-discount pic v9(02).
115 10 ls-item-expiry-date pic 9(06).
116
117 Procedure Division Using ls-item-data.
118 Move ls-item-data to item-data
119
120 End Method "populate-the-item-object".
121 *>-----
122 END OBJECT.
123 END CLASS ItemClass.

```

ITEM08DA program

```

1 *>*****
2 *> Item System
3 *> ITEM08DA.CBL
4 *>
5 *> This class definition is the database
6 *> interface for the ITEM system. It reads a
7 *> requested record from the indexed file ITEM2.DAT
8 *> Object methods
9 *> close-file
10 *> open-file
11 *> read-item-record
12 *>*****
13
14 Identification Division.
15 Class-ID. ItemDatabaseInterface
16 inherits from Base.
17
18 Environment Division.
19 Input-Output Section.
20 File-Control.
21 Select Item-File assign to disk "a:item2.dat"
22 organization is indexed
23 access is random
24 record key is rr-item-number.
25
26 Object Section.
27 Class-Control.
28 ItemDatabaseInterface is class "item08da"
29 ItemClass is class "item05c1"
30 Base is class "base"
31 .
32
33 Data Division.
34 File Section.
35 FD Item-File.
36 01 item-record.
37 10 rr-item-number pic 9(03).
38 10 rr-std-config-price pic 9(03).
39 10 rr-spec-config-price pic 9(03).
40 10 rr-item-discount pic v9(02).
41 10 rr-item-expiry-date pic 9(06).
42
43 *>=====
44 OBJECT.
45
46 Procedure Division.
47 *>-----<*
48 *> Object Methods <*
49 *>-----<*
50 *>-----

```

```

51 Method-ID. "close-file".
52 *>-----
53 Procedure Division.
54 Close Item-File
55 .
56 End Method "close-file".
57
58 *>-----
59 Method-ID. "open-file".
60 *>-----
61 Procedure Division.
62 Open input Item-File
63 .
64 End Method "open-file".
65
66 *>-----
67 Method-ID. "read-item-record".
68 *>-----
69 Data Division.
70 Linkage Section.
71 01 ls-item-number pic 9(03).
72 01 ls-theItemHandle object reference.
73
74 Procedure Division Using ls-item-number
75 Returning ls-theItemHandle.
76 Move ls-item-number to rr-item-number
77 Read Item-File
78 Invalid key
79 Set ls-theItemHandle to null
80 Not invalid key
81 Invoke ItemClass "New"
82 Returning ls-theItemHandle
83 Invoke ls-theItemHandle
84 "populate-the-item-object"
85 Using item-record
86 End-Read *>Item-File
87 .
88 End Method "read-item-record".
89 *>-----
90 END OBJECT.
91 END CLASS ItemDatabaseInterface.

```

ITEM2.DAT (Indexed file, sequentially printed)

```

20022538505130899
20124040505241000
20225542505250499
20328046505180901
20430550505141200
20534056505091199
20635061007121199
20737561507150599
20838062510130899
20940065510151200

```

Displayed output

The following displayed output is for item numbers 200, 201 and 202. An invalid item number 230 is tested, and the program is terminated by typing in item number 0.

The input from the user is given in italics.

Item number 200

Item number <0 to terminate>? *200*

The options are:

- e - Item expiry-date <default>
- s - Standard item prices
- d - Discounted item prices

Your choice <e,s,d>? *e*

Item expiry-date: 130899

Item number 201

Item number <0 to terminate>? *201*

The options are:

- e - Item expiry-date <default>
- s - Standard item prices
- d - Discounted item prices

Your choice <e,s,d>? *s*

Standard configuration price: 240

Special configuration price: 405

Item number 202

Item number <0 to terminate>? *202*

The options are:

- e - Item expiry-date <default>
- s - Standard item prices
- d - Discounted item prices

Your choice <e,s,d>? *d*

Standard configuration price: 242

Special configuration price: 403

Invalid item number 230

Item number <0 to terminate>? 230

Invalid item number

Terminate

Item number <0 to terminate>? 0

Processing complete

oooOOOooo

References

- [Aba96] Abadi M. and Cardelli L., *A Theory of Objects*. Springer, New York. 1996.
- [Ach97] Achatz K. and Schulte W., *A Formal OO Method Inspired by Fusion and Object-Z*. Department of Computer Science, University of Ulm. 1997.
- [Bac97] Back R.J.R. and von Wright J., *Refinement Calculus: A Systematic Introduction*. Springer Verlag. 1997.
- [Bah99] Bahrami A., *Object Oriented Systems Development using the Unified Modeling Language*. Irwin/McGraw-Hill. 1999.
- [Bar94] Barden R. Stepney S. and Cooper D., *Z in Practice*. Prentice Hall. 1994.
- [Ber93] Berard E.V., *Essays on Object-Oriented Software Engineering, Volume 1*, Prentice Hall, Englewood Cliffs, N.J. 1993.
- [Boe81] Boehm B.W., *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, N.J. 1981.
- [Boe88] Boehm B.W., *A Spiral Model of Software Development and Enhancement*, IEEE Computer 21 (May 1988), pp. 61-72. 1988.
- [Boc94] Bock C. and Odell J., "A Foundation For Composition," *Journal of Object-oriented Programming*. October 1994.
- [Boo94] Booch G., *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin/Cummings, Redwood City, CA. 1994.
- [Boo97] Grady B., Jim R., and Ivar J., *Unified Modeling Language User Guide*, Addison Wesley. December 1997. See www.awl.com/cp/uml/uml.html.
- [Bou95] Bourdeau R. and Cheng B.H.C.. A formal semantics for object model diagrams. *IEEE Trans. Software Eng.*, 21, 799-821. 1995.
- [Bri96] Brinkkemper S., Lytinen K. and Welke R., *Method engineering*. Chapman & Hall, New York. 1996.
- [Buz94] Buzzi-Ferraris G., *scientific C++*. Addison-Wesley 1994.
- [Coo94] Cook S. and Daniels J., *Designing Object Systems: Object-oriented Modelling with Syntropy*, Prentice-Hall Object-Oriented Series. 1994.
- [Cra91] Craig I.D., *Formal Specification of Advanced AI architectures*. Chichester: Ellis Horwood. 1991.
- [Dav90] Davis A. M., *Software Requirements, Analysis and Specification*. Prentice-Hall. 1990.
- [Der01] Derrick J. and Borten E., *Refinement in Z and Object-Z*. Foundations and Advanced Applications. Springer-Verlag London Limited. 2001.
- [D'So97a] D'Souza D. and Wills A., "Input for the OMG Submission." 1997. www.iconcomp.com/catalysis
- [D'So97b] D'Souza D. and Wills A., "Catalysis: Component and Framework based development." 1997. www.iconcomp.com/catalysis
- [Dij76] Dijkstra E. W., *A Discipline of Programming*. Prentice Hall, Hemel Hempstead. 1976.

- [Dor99] Dorsey P. Dr. and Hudicka J.R., *Oracle8 Design Using UML Object Modeling*, Osborne/Mc Graw-Hill. 1999.
- [Dou98] Douglass B., of i-Logix. 1998. <http://www.rational.com/support/techpapers/umlrt/>
- [Eri98] Eriksson H.E. and Penker M., *UML Toolkit*. John Wiley & Sons, Inc. 1998.
- [Eva98] Evans A., Reasoning with the unified modeling language. In *Proc. WIFT '98*. IEEE Press, Los Alamitos, CA. 1998.
- [Eva98b] Evans A. and Clark A., Foundations of the Unified modeling language. In *Proc. 2nd Northern Formal Methods Workshop*. Springer, Berlin. 1998.
- [Fow97] Fowler M. and Scott K., *UML Distilled: Applying the Standard Object Modeling Language*, ISBN 0-201-32563-2, Addison-Wesley. 1997. <http://www.awl.com/cp/uml/uml.html>
- [Fra97] France R.B., Bruel J. M., Larrondo-Petrie M.M. and Shroff M., *Exploring the Semantics of UML Type Structures with Z*. Chapman & Hall, IFIP. 1997.
- [Fuk94] Fukagawa M., Hikita T. and Yamazaki H., *A Mapping System from Object-Z to C++*. IEEE. 1994.
- [Gor89] Gorlen K., *An Introduction to C++, in UNIX System V AT&T C++ Language System, Release 2.0 Selected Readings*. Murray Hill, N.J.: AT&T Bell Laboratories, p. 2-1. 1989.
- [Gra97] Graham I., Henderson-Sellers B. and Younessi H., *The OPEN Process Specification*, Addison-Wesley, Harlow. 1997.
- [Gra98] Grauer R.T., Villar C.V., and Buss A.R., *COBOL From Micro to Mainframe*. 3rd ed. Prentice-Hall. 1998.
- [Gra00] Grauer R.T., Villar C.V., and Buss A.R., *COBOL From Micro to Mainframe*. 3rd ed. Prentice-Hall. 2000.
- [Gri96] Griss M., "Domain Engineering And Variability In The Reuse-Driven Software Engineering Business." *Object Magazine*. Dec 1996. (See www.hpl.hp.com/reuse)
- [Hai98] Hamie A., Howse J. and Kent S., *Navigation Expressions in Object-Oriented Modelling*. University of Brighton. 1998.
- [Hal94] Hall A., Specifying and interpreting class hierarchies in Z. In *Proc. ZUM '94*. Springer, Berlin. 1994.
- [Hal96] Hall A., Using formal methods to develop an ATC information system. *IEEE Software*, 13, 66-76. 1996.
- [Ham94] Hammond J., "Producing Z specifications from object-oriented analysis." In *Proc. ZUM '94*. Springer, Berlin. 1994.
- [Ham95] Hamilton V., *The use of Z within a safety-critical software system*. In: Applications of Formal Methods, edited by Hincley M.G. and Bowen J.P. Hemel Hempstead: Prentice Hall International. 1995.
- [Har87] Harel D., "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming* 8, 231-274. 1987

- [Har96a] Harel D. and Gery E., "Executable Object Modeling with Statecharts." Proc. 18th Int. Conf. Soft. Eng., Berlin, IEEE Press, March, pp. 246-257. 1996.
- [Har96b] Harel D. and Naamad A., "The STATEMATE Semantics of Statecharts," ACM Trans. Soft. Eng. Method 5:4 Oct. 1996.
- [Heh81] Hehner E.C.R., "Bunch theory: a simple set theory for computer science." Inform. Process. Lett., 12, 26-30. 1981.
- [Heh93] Hehner E.C.R., *A Practical Theory of Programming*. /Springer, New York. 1993.
- [Hel98] Hellwig F., Implementing Associations. Maintaining pointer integrity the easy way. Dr. Dobb's Journal. June 1998.
- [Hen90] Henderson-Sellers B. & Edwards J. M., The Object-Oriented Systems Life Cycle. Communications of the ACM 33 pp. 142-59. September 1990.
- [Hoa89] Hoare C. A. R., An axiomatic basis for computer programming. *Communications of the ACM*, 12. 1989.
- [Jac92] Jacobson I., Christenson M., Jonsson P., and Overgaard G., *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press, New York. 1992.
- [Jac97] Ivar J., Grady B., and Jim R., *The Objectory Software Development Process*, ISBN: 0-201-57169-2, Addison Wesley est. December 1997. See www.awl.com/cp/uml/uml.html and the "Rational Objectory Process" on www.rational.com.
- [Jak95] Jackson M.A., *Software Requirements and Specifications*. Addison-Wesley, Wokingham, UK. 1995.
- [Jak97] Jacky J., *The Way of Z*. Cambridge University Press. 1997.
- [Jal89] Jaloti P., *Functional Refinement and Nested Objects for Object-Oriented Design*. IEEE 1989.
- [Jon90] Jones C.B., *Systematic software development using VDM*. Prentice Hall, New York. 1990.
- [Kot97] Kotzé P., *The use of Formal Models in the Design of Interactive Authoring Support Environments*. DPhil Thesis, University of York (U.K.) 1997.
- [Kro93] Kronlöf K., *Method Integration: Concepts and Case Studies*. Wiley, New York. 1993
- [Kru99] Kruchten P., *The Rational Unified Process*. Addison-Wesley, Reading. 1999.
- [Mal96] Malan R., Coleman D. and Letsinger R. et al., "The Next Generation of Fusion." *Fusion Newsletter*. Oct 1996. (See www.hpl.hp.com/fusion.)
- [Mar95] Martin J. and Odell J., *Object-oriented Methods, A Foundation*, ISBN: 0-13-630856-2, Prentice Hall. 1995.
- [Met94] MetaPHOR Project Group *MetaPHOR: Metamodeling, Principles, Hypertext, Objects and Repositories*. Technical Report TR-7, University of Jyvaskyla. 1994.
- [Mey97] Meyer B., *Object-oriented Software Construction* (2nd edn). Prentice-Hall, Upper Saddle River, NJ. 1997.
- [Mis97] Mistic V. and Moser S., "Formal approach to metamodeling: a generic object-oriented perspective." In *Proc. ER '97*, LNCS 1331. Springer, Berlin. 1997.

- [Ode98] Odell J.J., *Advanced Object-Oriented Analysis and Design using UML*. Cambridge University Press. 1998.
- [OMG99] Object Management Group (Report) <http://www.omg.org/>. 1999.
- [Pai97] Paige R. F., A meta-method for formal method integration. In *Proc. FME '97*, LNCS 1313. Springer, Berlin. 1997.
- [Pai97b] Paige R.F., *Formal Method Integration via Heterogeneous Notations*. PhD Dissertation, University of Toronto, November. 1997.
- [Pai98] Paige R.F. and Ostroff J.S., From Z to BON/Eiffel. In *Proc. automated Software Engineering*. IEEE Press, Los Alamitos, CA. 1998.
- [Pai98b] Paige R.F., Comparing extended Z with a heterogeneous notation for reasoning about time and space. In *Proc. ZUM '88*, LNCS 1439, Springer, Berlin. 1998.
- [Pai99] Paige R.F., When are methods complementary? *Inform. Software Technol.*, **41**, 157-162. 1999.
- [Pai99b] Paige R.F., *Integrating a Program Design Calculus and a Subset of UML*. The Computer Journal, Vol. 42, No. 2. 1999.
- [Pol92] Polack F., *Integrating formal notations and systems analysis: using entity relationship diagrams*. Software Engineering Journal, p.363-371. September 1992.
- [Pol93] Polack F., Whiston, M. and Mander, K.C. "The SAZ Project: integrating SSADM and Z." In *Proc. FME '93: Industrial-strength Formal Methods*, LNCS 670. Springer, Berlin. 1993.
- [Pot96] Potter B., Sinclair J. and Till D., *An Introduction to Formal Specification and Z*. 2nd Ed. Prentice Hall. 1996.
- [Pri97] Price W., *Elements of Object-Oriented COBOL*. Object-Z Publishing. 1997.
- [Pri98] Price W., *Elements of Cobol Web Programming: with Micro Focus and NetExpress*. Object-Z Publishing. 1998.
- [Raf93] Rafsnajani B., Colwill S.J., *From Object-Z to C++: A Structural Mapping*. 1993.
- [Raa94] Rann D., Turner J. and Whitworth J., *Z: A Beginner's Guide*. Chapman & Hall. 1994.
- [Ram95] Ramackers G. and Clegg D., "Object Business Modelling, requirements and approach" in Sutherland, J. and Patel, D. (eds.), *Proceedings of the OOPSLA95 workshop on Business Object Design and Implementation*, Springer Verlag, publication pending. 1995.
- [Ram96] Ramackers G. and Clegg D., "Extended Use Cases and Business Objects for BPR," *ObjectWorld UK '96*, London, June 18-21, 1996.
- [Ram98] Ramteke T., *Introduction to C and C++ for Technical Students. A Skill Building Approach*. Prentice-Hall. 1998.
- [Ran94] Rann D., Turner J., and Whitworth J., *Z: A Beginner's Guide*. Chapman & Hall. 1994.
- [Rat94] Ratcliff B., *Introducing Specification using Z*. McGraw-Hill Book Company. 1994.
- [Roy70] Royce W.W., Managing the Development of Large Software Systems: Concepts and Techniques," *WESCON Technical Papers, Western Electric Show and Convention*,
th

- Los Angeles, August 1970, pp. A/1-1-A/1-9. Reprinted in *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, May 1989, pp. 328-338. (Chapter 3) 1970.
- [Rum91] Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W., *Object-Oriented Modeling and Design*. Prentice_Hall. 1991.
- [Rum97] Rumbaugh J., Jacobson I., and Booch G., *Unified Modeling Language Reference Manual*, ISBN: 0-201-30998-X, Addison Wesley, est. publication December 1997. See www.awl.com/cp/uml/uml.html. [UML Web Site] www.rational.com/uml
- [Sae98] Saeki M., A meta-model for method integration. *Inform. Software Technol.*, **39**, 925-932. 1998.
- [Sch96] Schach S.R., *Classical and Object-Oriented Software Engineering*. Irwin. 1996.
- [Sch99] Schach S.R., *Classical and object-oriented software engineering with UML and C++*. 4th ed., Mc Graw-Hill. 1999.
- [Sek97] Büchi M. and Sekerinski E., *Formal Methods for Component Software: The Refinement Calculus Perspective*. From: Object-Oriented Technology ECOOP'97 Workshop Reader. Springer. 1997.
- [Sem90] Semmens L. and Allen P., *Using Yourdon and Z: an approach to formal specification*. (Draft) Proc. Fifth Annual Z User Meeting, Oxford, December 1990.
- [Sem92] Semmens L.T., France R.B. and Docker T.W., Integrated structured analysis and formal specification techniques. *Comp. J.* **35** 600-610. 1992.
- [Shl92] Shlaer S. and Mellor S.J., *Object Lifecycles – Modeling the World in States*. Prentice-Hall. New York. 1992.
- [Shr97] Shroff M., and France R.B., *Towards a Formalisation of UML Class Structures in Z*. IEEE. 1997.
- [Smi95a] Smith G., *A Development Framework for Object-Oriented Specification and Refinement*. 1995.
- [Smi95b] Smith G., *Extending W for Object-Z*. In J. Bowen and M. Hinchey, editors, *9th International Conference of Z Users (ZUM'95)*, volume 967 of Lecture Notes in Computer Science, pages 276-295. Springer-Verlag, 1995.
- [Som92] Sommerville I., *Software Engineering*. 4th Ed. Addison-Wesley. 1992.
- [Som97] Sommerville I. and Sawyer P., *Requirements Engineering. A good practice guide*. John Wiley & Sons. 1997.
- [Spi88] Spivey J.M., *Understanding Z. A Specification language and its formal semantics*. Cambridge University Press. 1988.
- [Spi89] Spivey J.M., *The Z Notation. A Reference Manual*. Prentice Hall. 1989.
- [Spi92] Spivey J.M., *The Z Notation: A Reference Manual*, 2nd Edn. New York: Prentice Hall. 1992.
- [Ste92a] Stepney S., Barden R. and Cooper D., *Object Orientation in Z*. Springer-Verlag. 1992.
- [Ste92b] Stepney S., Barden R. and Cooper D., *A Survey of object orientation in Z*, Software Engineering Journal, March 1992.

- [Ste97] Stern N. and Stern R.A., *Structured COBOL Programming*. 8th ed. 1997.
- [Toy00] Toyn I., CaDiZ Type checker and Theorem Prover Home Page. <http://www.cs.york.ac.uk/~ian-/cadiz/home.html>. 2000.
- [vdP00] Van der Poll, J. A., *Automated support for Set-Theoretic Specifications*. PhD thesis, Department of Computer Science & IS, University of South Africa. June 2000.
- [vHa93] van Harmelan M., *Object-Oriented Modeling and specification for User Interface Design*. 1993.
- [Vie97] Vienneau R., "A Review of Formal Methods." In Richard H. Thayer and Merlin Dorfman, editors, *Software Requirements Engineering*, pages 324 – 335. IEEE Computer Society, 2nd edition, 1997.
- [Wal95] Walden K. and Nerson J.M., *Seamless Object-Oriented Software Architecture*, Prentice-Hall, New York. 1995.
- [War99] Warmer J. and Kleppe A., *Object Constraint Language, the Precise Modelling with UML*. Addison-Wesley Object Technology Series. 1999
- [Weg87a] Wegner P., *Dimensions of object-based language design*. OOPSLA '87 Proc, (ACM SIGPLAN Not., 1987, 22,(12), pp. 168-182). 1987.
- [Weg87b] Wegner P., *The object-oriented classification paradigm in Shriver B., and Wegner P. (Eds): Research directions in object-oriented programming*. (MIT Press). 1987.
- [Woo96] Woodcock J., Davies J., *Using Z Specification, Refinement, and Proof*. Prentice Hall. 1996.
- [Wor92] Wordsworth J. B., *Software Development with Z*. Addison-Wesley. 1992.
- [Zav93] Zave P. and Jackson M., Conjunction as composition. *ACM Trans. Software Eng. Method.*, 2, 379-411. 1993.