Design and implementation of a Multi-Purpose Object-Oriented Spatio-Temporal (MPooST) data model for Cadastral and Land Information Systems (C/LIS)

Christoforos Vradis

Submitted for the title of Master of Science

Department of Geography and Topographic Science
University of Glasgow

January 2000

ProQuest Number: 13834007

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 13834007

Published by ProQuest LLC (2019). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

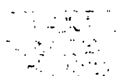
GLASGOW
UNIVERSITY
LIBRARY

11996 - Copy 1

Declaration

The work presented in this thesis is my own unaided work, except where acknowledgedement is given, and has not been submitted for a higher degree in this or any other University.

Christoforos Vradis



Abstract

The application of the object-oriented methodology in geospatial information management has significantly increased during the last 10 years and tends to gradually replace the *status quo* relational technology. In general, object orientation offers a flexible and adaptable modelling framework to satisfy the most demanding complex data structuring requirements.

The objective of this thesis is to determine how a modern Land Information System used for cadastral purposes can benefit from an object-oriented methodology. To this aim, a Multi-Purpose, Object-Oriented Spatio-Temporal (abbreviated as MPOOST) data model has been developed. In brief, the MPOOST data model embodies spatial data and their temporal reference in the form of objects which contain their attributes as well as their behaviour. The design of the MPOOST data model has been specified in such a way that it enables other data models to exploit its functionality, therefore enabling the multi-purpose aspect.

At first, the requirements of Land Information Systems are being examined. Next, the functionality that is offered by the object-oriented methodology is being analysed in detail. Even if the bibliography is quite rich in relevant research, however there seems to be no starting point regarding the application of OO in LIS. Hence, a whole chapter of this thesis has been dedicated in an extended bibliographic research. Finally, the OO methodology is applied for the design and implementation of the MPOOST data model.

The outcome of the design and the implementation is the first version of the MPOOST data model written using the Java object-oriented programming language.

In this way, it is proven that:

- the relational technology has significant drawbacks which prohibit it from being applied in conceptually demanding information systems; and that
- object-orientation can fully satisfy the most complex data structuring requirements posed in modern geographic information systems.

Contents

Acknowledgements **15**

Finance, resources and support of the research 17

Introduction **19**

About the research 19

Research objectives and aim 20

Research structure 21

Thesis structure 22

Case study: the Hellenic Cadastre 24

1 Information Systems **27**

1.1 Information systems 27

1.2 Software Engineering 28

1.3 Geographical Information Systems (GIS) 28

1.4 Cadastre and Land Information Systems (LIS) 32

1.4.1 Data modelling 36

1.4.1.1 Providing support for multiple application domains and platform independence 40

1.5 Data in a Geoinformation system 41

1.5.1 Spatial Data 42
1.5.2 Spatial data models and their functionality 44
1.5.2.1 Field versus object data models 44
1.6 Required functionality in a GIS 45
1.7 Data concepts 50
1.7.1 Spatial and temporal data modelling 50
1.7.2 Data concepts 55
1.7.2.1 Spatial data and their geometric space 55
1.7.2.1.1 Coordinate systems 57
1.7.2.2 Vector data 58
1.7.2.2.1 Topology of vector data 58
1.7.2.2.2 Modelling primitives 59
1.7.2.3 Raster Data 61
1.7.2.4 Temporality 61
1.7.2.5 Metadata 73
1.7.3 Functionality requirements 80
1.7.3.1 Operations 84
1.8 The object-oriented approach 90
1.8.1 Object-oriented analysis and design 90
1.8.2 Summary of object-oriented concepts 92
1.8.3 Object-oriented database management systems 94
1.8.4 Functionality issues summary 95

2 Review of Object Orientation 96

2.1 Introduction 96

2.2 Moving from conventional models and technology to object-orientation 97

2.3 Object orientation 99

2.3.1 General concepts of object orientation 107

2.3.1.1 Abstraction and Abstract data types (ADT) . . . 108

2.3.1.2 Classes, Hierarchy and Modules 111

2.3.1.3 Inheritance 116

2.3.1.4 Structure of classes - Attributes 124

2.3.1.5 Structure of classes - Behaviour of objects (Interface, Methods/ Operations, Messages) 131

2.3.1.6 Associations 137

2.3.1.7 Encapsulation 141

2.3.1.8 Polymorphism 142

2.3.1.9 Objects as instances of classes, state and behaviour of objects 146

2.3.1.10 Metaclasses 149

2.4 Analysis and Design Techniques 151

2.4.1 Introduction 151

2.4.2 The UML approach 154

2.4.3 UML elements 156

2.4.3.1 Classes 157

2.4.3.2 Metaclasses 160

2.4.3.3 Interfaces 161

2.4.3.4 Use cases 163

2.4.3.5 Objects 164
2.4.3.6 Packages 164
2.4.3.6.1 Logical packages 165
2.4.3.6.2 Component packages 165
2.4.3.7 Operations 166
2.4.3.8 Components 166
2.4.3.9 Processors 167
2.4.3.10 Devices 167
2.4.3.11 Relationships 168
2.4.3.12 Messages 173
2.4.3.13 General extensibility mechanisms 173
2.4.3.14 Constraints 174
2.4.3.15 Stereotypes 174
2.4.4 UML diagrams 175
2.4.4.1 Class and object diagrams 176
2.5 Implementation - Languages and databases 176
2.5.1 Object Oriented Programming Languages 177
2.5.2 Object Oriented Databases 179
2.5.2.1 Special characteristics of object-oriented databases 181
2.5.2.1.1 The transaction management mechanism 181
2.5.2.1.2 Support of integrity constraints 183
2.5.2.1.3 Concurrency 184
2.5.2.1.4 Recovery 184
2.5.2.1.5 Versioning 185
2.5.2.1.6 Performance enhancement 186

2.5.2.1.7 Query processing 189

2.5.2.1.8 Schema Modifications 192

2.5.2.1.9 Security and authorization 193

2.5.2.2 UML in database modelling. 194

2.5.2.3 Usability of OO databases in GIS 194

2.6 Conclusions 195

3 Analysis 197

3.1 Introduction 197

3.2 Organization of terminology 198

3.3 Basic requirements 199

3.4 Requirements and terminology in the real world domain 200

3.5 Terminology and requirements in the system domain 202

3.6 Terminology and requirements in the map domain 203

3.7 Data input 204

3.8 Spatial information 205

3.8.1 Geometry 205

3.8.2 Coordinate systems 207

3.9 Spatial topology 209

3.10 Aspatial data related to geometry 210

3.10.1 Attributes 210

3.10.2 Data quality 212

3.10.3 Positional accuracy 213

3.10.4 Attribute accuracy 214

3.10.5 Completeness 214

3.10.6 Complete list of required metadata 215

3.10.7 Spatial resolution 219

3.11 Temporality 219

3.12 Representation 222

3.12.1 Behaviour 224

3.13 Enabling action in space: spatio-temporal entities 225

3.13.1 Application domains 228

3.13.1.1 Case study: the Hellenic Cadastre 228

3.13.1.1.1 Entities in the cadastre 230

3.14 Analysis of the required multi-purpose model characteristics . . . 232

4 Design of the MPOOST model **236**

4.1 Introduction 236

4.2 Overall structure of the MPOOST model 237

4.3 Stereotypes defined in the UML model 240

4.3.1 Stereotyped classes 241

4.3.2 Stereotyped relationships 241

4.4 Detailed package structure and class description 242

4.4.1 The `MPOOST` package 242

4.4.2 Package `MPOOST.Spatial` 243

4.4.2.1 Package `MPOOST.Spatial.RealWorld` 243

4.4.2.2 Package `MPOOST.Spatial.Cartographic.Geometry` 252

4.4.2.3 Package `...Cartographic.MapComposition` . . . 254

4.4.3 Package `MPOOST.Representational` 255

4.4.3.1 Package `MPOOST.Representational.Symbol` . 255

4.4.4 Package MPOOST.Behavioural 257
4.4.4.1 Package MPOOST.Behavioural.Utility 257
4.4.5 Package MPOOST.ASpatial 258
4.4.5.1 Package MPOOST.ASpatial.Primitive 258
4.4.5.2 Package MPOOST.ASpatial.Metadata 260
4.4.5.2.1 Package ...Metadata.ReferenceSystems 263
4.4.5.2.2 Package ...Metadata.CoordinateSystems 264
4.4.6 Package MPOOST.Temporal 266
4.4.7 Package MPOOST.GUI 267
4.4.8 Package MPOOST.ApplicationDomains 267
4.4.8.1 Package MPOOST.ApplicationDomains.Cadastre 268
4.4.8.2 Package MPOOST.AppilicationDomains.Topography 268

5 Implementation of the MPOOST model 272
5.1 Java as the implementation language 272
5.2 Code conventions 275
5.3 MPOOST Package Index 275
5.4 Class Hierarchy 275
5.5 The MPOOST Graphical User Interface 277
5.6 MPOOST GUI Snapshots 278

6 Conclusions and further research 282
6.1 Data model assessment 282
6.2 Future work 286

Appendices **304**

A Object Oriented Programming Languages **304**

A.1 Smalltalk 304

A.2 PS-ALGOL 304

A.3 C++ 305

A.4 Java 305

B Interoperability Standards In Object Orientation **310**

B.1 CORBA 310

List of Figures

1.1 Building up an information system 28

1.2 Domains and the modelling stages in a multipurpose GIS 37

1.3 Functionality of a GIS. (Modified from [82]) 47

1.4 Discrete modelling sequence, spaces and topological models 56

2.1 Specialization relationships among classes. 112

2.2 Hierarchy of classes 113

2.3 Is object A1:X an instance of class A1 only, or both A1 and A? . 117

2.4 Class B will inherit members belonging to class C twice: once from super-class C1 and another from super-class C2. Which of the two should be preserved? (italics denote inherited members.) 122

2.5 Class C2 contains the attribute A, which is a reference to a class X. In class C2, attribute A can receive as values not only objects belonging to class X, but objects belonging to X sub-classes, namely X1 and X2 124

2.6 Class icon 157

2.7 Abstract class icon 158

2.8 Interface icon in component diagrams 162

2.9 Interface icon in class diagrams 162

2.10 Use case icon 163

2.11 Single (left) and Multiple (right) Object icons 164

2.12 Logical (left) and Component (right) package icons 165

2.13 Component icon 166

2.14 Processor icon 167

2.15 Device icon 168

2.16 Notation used for Relationships 169

4.1 The overall structuring of the data model into packages 237

4.2 Package MPOOST.Spatial.RealWorld 269

4.3 Package MPOOST.Representational.Symbol. For visual clarity, thick lines represent specialization and thin lines represent association. 270

4.4 The parent class for all application domain classes. 271

5.1 The main part of the MPOOST GUI. 279

5.2 Class selection window 280

5.3 Class schema browser window 281

A.1 Workaround for multiple inheritance in the UML class diagram when not supported by the implementation language. Text in italics denotes class members inherited from the superclass(es). See text for explanation 308

List of Tables

2.1 Object Orientation in IT 103

2.2 Major concepts of Object Orientation 106

2.3 UML elements 156

3.1 Domains and concepts 204

A.1 Comparison of OO programming languages. Based on comparison found in Booch (1994), augmented with additional features and facts about Java. 309

Acknowledgements

The author would like to thank the following individuals, institutions and companies:

- The **Hellenic State Foundation of Scholarships**, for the funding of the research, including tuition fees and living expenses for the period of 16 months at the University of Glasgow.
- **Dr. Jane Drummond**, lecturer of the Geography and Topographic Science Department for help on geo-information technology issues, as well for the funding provided a) to be trained on Laser Scan products, and b) to attend the 6th GISRUK conference in Edinburgh (for which there was also RRL.net funding).
- **Mr. Ian Gordon**, lecturer of the department and supervisor of the research, mainly on issues regarding Cadastral and Land Information Systems.
- **Matt Duckham**, PhD student in Topographic Science Section, for his valuable help and recommendations on Gothic ADE, object-orientation and the *Linux* operating system.
- **Brian Black** and **Stephen McGinley**, IT technicians at the Geography and Topographic Science Department, for their IT support.
- Staff of the **Geography and Topographic Science Department**, for the feedback I received from the presentation of the initial research proposal, in April 1998.

- Staff of the **Computing Science Department**, Glasgow University, for their help on the object-oriented approach and Java programming language.
- **Caroline Hogan**, from SDS company at Bo'ness in Scotland for kindly preparing and providing the topographic dataset used to evaluate the Java application of the MPOOST model.
- The **Laser Scan Ltd** company, for their training and support on Gothic Application Development Environment.
- The companies **Rational** and **Valtech**, for the free seminars on the Unified Modeling Language.

Finally, I would like to thank anyone that helped me throughout the research elaboration, but who is not mentioned above.

Finance, resources and support of the research

Funding for the research was provided by the Hellenic State Foundation of Scholarships, for the period of 16 months in total, including both tuition fees as well as living expenses. Additional funds have been kindly provided by the Department of Geography and Topographic Science, for training purposes.

Computational resources used during the research include:

- A Fujitsu personal computer, running at 133 MHz, with 32 MB of RAM and 2 GB of hard disk space.
- My perosnal Mitac notebook equipped with a Pentium processor running at 133MHz, with 1.3GB hard disk and 32 MB RAM
- Operating systems including Microsoft Windows 95, installed by the department's IT technicians, and S.u.S.E Linux 5 and RedHat Linux 5.2, installed by the researcher. Both OSs were running on the same machine.
- Software for Linux including:
  1. GIMP v. 1.0.1 image manipulation program.
  2. XEmacs 20.4 text and code editor.
  3. Xfig 3.2 graphics program.
  4. LaTeX 2_εtypesetting environment
  5. KDE v1.0 X Windows manager
  6. Netscape Navigator 4.5
  7. Java Development Kit v1.1.7/1.2, including compiler, debugger and code documentation parser.
  8. html2tex , HTML to LaTeXconverter
- A Sun SPARCstation 20 computer, with 192MB RAM, 4GB hard disk space, running on Solaris OS.
- Software for Solaris OS including Laser Scan's Gothic ADE GIS, PJama for Java compiler and classes.

Usage of Laser Scan's Gothic ADE was done remotely on the PC through Linux's X Windows environment.

Technical support was provided by:

- the department's IT support group
- Computing Science Department of Glasgow University
- Laser Scan Ltd.
- Usenet newsgroups available through the Internet (e.g. comp.infosystems.gis) as well as support offered by companies in the form of newsgroups (such as Sun's Java Developer Connection)

Introduction

About the research

This dissertation documents the research that the author has undertaken during the period from February 1998 to May 1999, in the Geography and Topographic Science Department of the University of Glasgow, as a research student for the degree of the Master of Science by research in Geoinformation Technology. In brief, it involves research into the *design and implementation of a multipurpose object-oriented spatio-temporal data model* (abbreviated as MPOOST) which can effectively be used in the context of a modern land information system.

The motive for this research was primarily the recent developments in the Hellenic National Cadastre, a project suggested and promoted by the Hellenic Ministry for the Environment, Physical Planning and Public Works. In brief, an initial two-phase pilot project started in 1994 and is now being implemented as this dissertation is being written. So far, property registration (that includes topographic mapping) for a part of the country is taking place (approximately 5 million acres), thus producing an extremely large volume of geographical and georeferenced data, the management of which is in need of a robust, flexible, effective, multipurpose, nation-wide Land Information System. The current proposal by the fairly recent established company "Cadastre S.A" regarding the information technology support, involves the adoption of the relational technology in the database management system, which will be used as the underlying computerized environment, mostly because it has been widely used in similar information

systems as a safe and proven solution. However, it is anticipated that this approach will definitely come to a point where the required functionality can not be satisfied. Thus, the research focuses on the object-oriented approach, and how it can be used to develop and implement such a multipurpose spatio-temporal data model so as to meet the requirements posed by this demanding application.

Research objectives and aim

Knowledge and theoretical concepts that have been devised so far in the context of Geographical Information Science are voluminous. During the first two months of the time allocated, a literature research has resulted in a record of the most important theoretical and practical issues on these theoretical concepts, which later are addressed in the context of spatio-temporal modelling. The aim of the research is primarily to propose an object-oriented design of a multipurpose spatio-temporal data model which will be used in the context of a modern cadastral information system. As a secondary stage, the research involves the implementation of the design which will be as independent of any specific computer platforms as possible. Finally, through the resulting model, an effort will be made to identify the degree by which current and future functionality requirements of a cadastral information system can be satisfied. Existing research work found in the literature that remains unimplemented will be considered in the literature review phase and part of it may be incorporated in the object-oriented design phase of the model.

Methodologically, the research work is divided into four major parts:

1. Literature research on object orientation in GIS.
2. Analysis in the GIS/LIS context.
3. Design of the MPOOST model.
4. Implementation and testing of the MPOOST model.

The system design phase involves aspects other than the data model itself (such as the hardware component). Some of these aspects will be of minor importance for the context of this thesis.

One of the main concerns of the research is to propose a design, and tools to work with which are as much as possible hardware and software independent as well as easy to become familiar with and be straightforwardly implementable. An aid to achieve this is the adoption of the current standards that have been developed so far (e.g. OMG). Whereas specific software platforms are considered and used, they are chosen with regard to the degree of independence that they have relative to the hardware platform.

Research structure

In order to better address the problems introduced and organize the research, five major aspects were identified and treated both separately during the analysis phase, and later in combination for the purposes of the design and implementation. These are:

1. analysis and design methodologies in information systems;
2. multipurpose systems and data models;
3. object-orientation;
4. spatial and temporal data, information and models;
5. cadastre and land information systems.

Research work started with familiarization of information technology tools and methodologies (such as object-orientation and object-oriented programming). Literature, as it is being accessed today, not only through any local university library but also through the Internet, allows one to tap an enormous amount of existing research work. It is noteworthy to mention that only in one week's time, a total of well over 300 references were collected. This can be better understood,

if one thinks of how many different science and engineering fields are involved in the research topic, such as mathematics, information technology, programming and modelling. However, not all of those references were used in the research, as firstly not all of them were of interest, and secondly, the time allocated to complete the research did not allow all references to be read and therefore included. Nonetheless, a significant amount of time (more than 8 months) was spent on the literature search itself. This was not obvious right from the beginning, hence an extension of 4 months was necessary, in order that the thesis be completed. The development of the model took approximately 5 months to complete, including the time necessary for familiarization with the object-oriented approach, the Unified Modelling Language and with the Java programming language.

Thesis structure

Chapter 1 (Introduction, pages 27-95) is an introduction to information technology, geoinformation systems, their structure, and the information that they can manage. The requirements of a modern geoinformation system are of main concern, and problems that are encountered are examined, along with solutions that have been suggested and used so far. A brief introduction in object orientation is given as an answer to how it can tackle all addressable issues. The literature search contributed significantly in the contents of this chapter.

Chapter 2 (Review of Object Orientation, pages 96-196) contains an extended discussion about object orientation, what it involves, where and how it may be used. Specific problems regarding geographic information that were encountered in chapter one are addressed in more depth. This chapter is purely an outcome of the literature research involved.

Chapter 3 (Analysis, pages 197-235) documents the analysis phase of the research. Analysis includes topics on the manner that real world entities and phenomena can be decomposed into primary building blocks (such as spatial and temporal

information, representation *etc.*) This chapter is based on the previous two and its outcome is the primary source from which the MPOOST model was developed.

Chapter 4 (Design of the MPOOST model, pages 236-268) explains how the MPOOST model was built. A documentation of the MPOOST model is also included, using not only textual descriptions but the Unified Modeling Language as a visual and definitely more comprehensible tool to the reader.

Chapter 5 (Implementation of the MPOOST model, pages 272-278) is about how the MPOOST model was implemented using the Java programming language. A detailed documentation of all Java packages, classes along with their methods and attributes may be found here. However, none of the Java code itself has been inlcuded in the thesis text, since it is anticipated that the reader will find more useful to interact with the Java GUI application (MPOOST GUI), rather than browse through the code. For reference purposes, all Java code is included on the accompanying compact disc (CD).

Chapter 6 (Conclusions and further research, pages 282-287) is a discussion on the functionality provided by the MPOOST model and how it can effectively address all the problems encountered. Any further research or development work that is considered necessary is also mentioned.

Finally, *Appendices* (pages 304-312) are on technical issues, and mostly on object oriented programming languages.

Additionally, a compact disc is accompanying the thesis which contains:

1. Code written in Java, in ASCII format (.java files), with all implemented packages and classes. Code files also contain documentation for classes and class members.
2. Documentation of the code in HTML format (.html files), which can be browsed using **any** world wide web browser (such as Netscape Navigator).
3. Compiled Java bytecode (.class files), which can be parsed by any Java interpreter.

4. The "MPOOST GUI" Java applet(MPOOSTGUI.html file), as a demonstration of the implemented MPOOST design, which is viewable within any world wide web browser with Java support (highly recommended is Netscape Navigator version 4.5 and higher).
5. All necessary additional Java classes (.class files) used (e.g. JFC Swing 1.1)
6. The actual thesis text, in postcript and device independent format (.dvi and .ps respectively), as well as all figures in postscript format.

It must be noted that the format of the documentation and code files (.html, .java) is platform independent, hence they can be browsed with any appropriate software within any operating system. When it comes to the execution of the byte code (.class files) this is not always the case. More specifically, every operating system requires a special version of the Java Virtual Machine used to execute the application. It is usually the case that a WWW browser includes a JVM so that it can parse Java applet code. When Java applications are involved, executable stand-alone files are required. As there is a plethora of operating systems, only the Windows 95/98 JVM files have been included on the CD. Instructions on how to launch the MPOOST GUI application may be found in the README.txt file.

Case study: the Hellenic Cadastre

For the testing of the research outcome, the case study chosen was the Hellenic Cadastre. It is a fairly recent established system (1997) replacing the older Land Registry system that was used so far. The official Hellenic institution responsible for the management of the cadastral information is the "Hellenic Cadastre and Mapping Organization - HCMO" (OKXE). Officially issued documents so far include:

– Technical specifications regarding data collection using photogrammetric and land surveying methods.

- Technical specifications regarding cartographic production.

Specifications found in the above documents were used as part of the design, mainly to incorporate the appropriate cadastral and legislative information in the final structure of the implemented data model.

Some official documents under consideration that still remain:

- The National Transfer Format for cadastral data submission from various private surveying companies to HCMO. At the time this dissertation was written, the format used is a set comprising three ASCII files, which contain a) the geometry, b) the topology and c) the attribute data.
- The national cadastral database system. Although the official initial proposal involves the usage of relational database management systems, however by the time of the thesis completion, no implementation had taken place.

The document involving the database management system relates strongly to the work undertaken in this thesis, and it can be said that it can contribute significantly towards the development of such a system in practice.

The phase of cadastral and topographic data collection has been recently initialized. For the purpose of data collection, Greece has been partitioned into 30 surveying areas where topographical mapping and later cadastral data collection (mainly property registration) will take place by a number of different surveying companies, both Hellenic and International. No specifications have been issued yet regarding details about the update of the data, how frequent this should happen etc.

The above specifications were collected during the early stage of the research and they were incorporated in the initial LIS domain analysis. Additionally, a sample vector data containing only topographic features was obtained from the SDS company, which were used:

- to analyze their inherent characteristics; and
- to test the implementation of the object model using the Java application.

However, no cadastral data relating to the topographic dataset acquired could be used, due to the organization's (HCMO) current policy, which does not allow any exportation of cadastral data out-with the Hellenic State.

The acquired data set can be browsed using the Java applet version of the MPOOST data model on the accompanying CD.

Chapter 1

Information Systems

1.1 Information systems

This section introduces the reader to the concepts used throughout the research. In brief, designing a spatio-temporal model should be done by taking into account the context in which the model will exist and function, namely a **geoinformation system**, which is a special kind of information system. An *information system* can be defined as a set of organized procedures and data, that, when executed, provides information. Information is some tangible or intangible entity that reduces uncertainty about a state or event [100]. In other words, it is a system capable of data input, data processing and information output. The hierarchical context of an information system could be shown as in figure 1.1.

As depicted above, an information system is an extension of a computer-based system, with the addition of the information handling procedures and technology. It also involves the human factor, that is the set of people that interact with the system, either as system developers or end users. It is the content of the information handling procedures and the information itself that add to the hardware capabilities to meet the demands for a specific application.

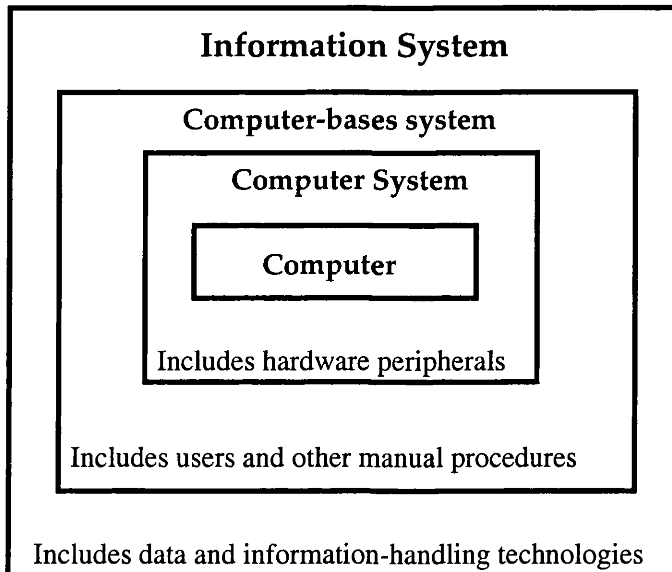


Figure 1.1: Building up an information system

1.2 Software Engineering

For any information system to be developed, some kind of software engineering methodology must be adopted and used. *Software engineering* for information systems is considered to be an iterative process which initiates from the application that poses the requirements, moves to the analysis and design phases, then to the implementation and testing stage, and goes back to the first stage, iterating as many times as necessary to refine any stage and apply the modifications to the consequent phases. This sequence is repeated as many times as necessary throughout the life cycle of the information system. For every different stage of this process, humans are involved with different roles, the most important of which are the *application domain expert*, the *architect*, the *designer*, the *implementer* and the *system tester* [22].

1.3 Geographical Information Systems (GIS)

A *geographical information system* or *geo-information system* (both abbreviated as GIS) can be defined as an information system that specializes in the manage-

ment of geographically referenced information. It provides the implementation environment for mechanisms such as collecting, storing, retrieving, processing, updating, querying and visualizing geographical data. This geographical component of information is related to any physical or artificial real world object or phenomenon. As Worboys [135] comments: '" GIS are not just systems, but also constitute an interdisciplinary endeavour involving many people in industry and academia. The aim of this endeavour is to provide better solutions to geo-spatial problems ". In this manner, a GIS can be thought of as an academia/industry enterprise system, in which a core database exists. Around and out of this database, users belonging to different application domains, store and retrieve data, through interfaces, that may or may not be spatially related. It is of great importance for the database model not to exclude possible user views but rather to incorporate them or at least provide the ability for expansion with the purpose of a future incorporation. Story [122] when defining a GIS, talks about "...integration of geo-scientific, [and] space refered information. [Moreover] The GIS can be extended by the fourth dimension (time). Thus a Geo-Information System is a space refered information system for the geosciences". Early stages of GIS are characterized by their *ad hoc* nature, and the wide variety of disciplines involved in the attempt to use computer technology, since solutions that GIS provided were very closely related to the application involved.

From a resource management perspective, the main four components that a GIS includes, are:

1. hardware
2. software
3. information and
4. the users that will interact with the system.

Software along with information are the parts that are stored, some way, within the hardware, although they are different in structure and behaviour. Software

has a dynamic behaviour, as it can instruct the hardware and manipulate the information. Information on the other hand is static relative to the hardware component, signifying that it is meaningful only to the users and possibly to software designed for information processing. Moreover, software is a series of machine level instructions stored permanently. Before this final status of storage, an interface (usually a high level language) has to be used by the developer to create it. Major software components in an information system (that are also relevant to a GIS as explained later in this chapter) are the *database management system* (DBMS), the *programming language* used for developing applications, and the *graphical user interface* that stands between any user and the system.

Information, spatial or not, originates from raw data through a series of processes. Along with raw data, it is organised via a *data model* which serves as an organisational framework, it is stored permanently into a database and thereafter it is manipulated through special software called the *database management system.* In any state of transformation, it can be visualized through the graphical functionality of the computer system.

Users may be categorised generally as either *end users* or *system developers.* End users are the persons that finally utilize the system, with significant limitations to alter the system's functionality. Their requirements usually include adding, storing, retrieving, updating and visualizing data and/or information out of the system. In an application specific GIS, end-user groups may be defined, with respect to their security access level, application domains, geographic location etc. The more generic and integrated a GIS is, the more end user sub-categories that will be found. This sub-grouping is necessary to be deployed prior to system implementation, at least in a very generic way so as to aid the system developers later to build application and/or data specific user group views and interfaces. System developers groups contain people whose responsibilities involve system activities with regard mainly to system functionality development itself, such as system analysis, programming, database development, data base administration and others. Sub-grouping here is also necessary in order to assign responsibilities

prior to and after the system development. The taxonomy of the users is not being addressed in the current thesis. In any category, a user interacts with the system via an *interface*. An interface provides access functionality at a specific conceptual level and hides the rest of the system architecture. User interfaces are developed based on the functional requirements of a user group.

In many cases, GI systems are being developed in the context of specific application domains (e.g. like environmental impact analysis, topographic mapping or mining exploration, to name but a few). There are cases though, when multidisciplinary applications are involved, such as governmental decision-making, where information from different application domains must be correlated, combined and used. In this case the GIS may be composed of many other modular sub-systems, connected into a network, each one focused on handling a specific part out of the total information set, thus presenting a *distributed environment*. In any case, important factors in building a GIS are the development time necessary and the assets involved for software development. Therefore, as every organization's will is to maintain as long as possible the core parts of the system, it is obvious that the underlying data model should be capable of expanding its structure whenever new information is to be stored. In this manner a GIS should be considered as an integrated environment, in need of an open architecture. Both will be satisfied to a great degree if the data model involved is intelligent and flexible enough. The role of *standardization* is very important in order to achieve open architecture within such a system.

As for the software components of a GIS, an architecture must be chosen. This can be of three major types [25]:

- *Dual* or *geo-relational* architecture, where a file system is used to store spatial data and a relational database management system is used to store aspatial data. Several commercial systems provide such a solution. A well known example in this category is ESRI's Arc/Info, which holds all the spatial data, while aspatial data that are stored in an external DBMS (e.g.

Oracle) may link to the spatial, through their unique identity. Another example is ESRI's ArcView which may be used with Microsoft's Access DBMS.

- *Layered* architecture, where both spatial and thematic data are stored in relational database management system. An example in this category is Oracle's Spatial Data Cartridge (SDC), which is built on top of Oracle relational DBMS, and provides functionality for storing spatial and aspatial data in the same database.
- *Integrated* architecture, where spatial and thematic data are kept in an extended relational database management system (ERDBMS).
- *Object-oriented* architecture, where spatial and thematic data are stored in an object oriented database. Example systems are LaserScan's Gothic ADE and Smallworld GIS.

1.4 Cadastre and Land Information Systems (LIS)

A *cadastre* may be narrowly defined as a record of interests in land, encompassing both the nature and extent of these interests [108]. An *interest* may be broadly interpreted to include any uniquely recognized relationship among people with regard to acquisition and management of land. A *land information system* (LIS) or a *cadastral information system* is a specialized GIS in which data are strongly related to the concept of legal rights to land. According to the Fédération Internationale des Géomètres (FIG) [41]:

> A Land Information System is a tool for legal administrative and economic decision-making and an aid for planning and development which consists on the one hand of a data base containing spatially referenced land-related data for a defined area, and on the other hand, of procedures and techniques for the systematic collection, updating, processing and distribution of the data. The base of a land information system is a

uniform spatial referencing system for the data in the system which also facilitates the linking of data within the system with other land-related data.

Since land related information is definitely geographically related, an LIS can be seen as a subset of a GIS regarding the type of data that are manipulated. On the other hand, due to the legal, administrative and economic impacts of an LIS, it becomes necessary to ascertain its functionality and effectiveness in relation to a GIS, as an LIS plays an important role within the decision making of a country. In an LIS database, in contrast to a generic GIS (in which effectiveness depends on the agreement between the data inside the database and the respective real world phenomena that they model), information in the database, once stored is reversely related to the real world phenomena and sometimes used to transfer these features back into the real world. In effect, the information in the system is of crucial importance: for example, the information regarding an ownership parcel along with its respective owner(s), once it has been verified and stored in the database, is considered to be legally correct as well as absolute, and provided with this information the specific parcel of land can be legally characterized as belonging to the specific owner. Unless the contents of the database are not correctly updated and verified whenever necessary, legal inconsistencies may occur.

The *multipurpose cadastre* may be defined as a large scale, community-oriented land information system designed to serve both public and private organizations and individual citizens [42]. This definition clarifies the multiplicity in the purpose it will serve. Its distinguishing characteristics [111] are:

1. Employment of a proprietary land unit (the cadastral parcel) as the fundamental unit of spatial organization;
2. Correlation of a series of land records (such as land tenure, land value and land use) to this land unit, the parcel;
3. Completeness, wherever possible, in terms of spatial cover;

4. Provision of a ready and efficient mean of access to the data.

Chrisman [37] identifies that the fundamental framework for a multipurpose land information system is the underlying *geodetic framework* and the relevant knowledge of its quality. This is quite obvious, if we take into account what different GIS applications have in common: that is basically the spatial data themselves, which are bound to a geodetic framework. It is quite often the case that spatial data refer to a different spatial framework in terms of its coordinate system definition, however this does not impose a serious problem when it comes to the combination of the data, which is feasible through coordinate system transformation.

Moreover, for a cadastre to serve as a multipurpose one, then the underlying data model (discussed further below in section 1.4.1) must be multipurpose as well, so that different applications that employ different data structures and information may have access to the same dataset, without imposing serious operability issues. For this goal to be achieved, many aspects have to be discussed first. The way that the proposed model is designed to be a multipurpose one, is discussed in section 3.14, but it is recommended that the reader should not skip the text in between, as important issues are discussed.

A cadastre usually contains a core data set, mostly related to land tenure rights and aspects of value, as well as additional data that are useful and can be referenced to the parcel. Core items may include land rights and restrictions, land values and tax assessments, land use, housing and buildings, population and census data, administrative information, archaeological sites. Additional information [42] might be the topography, geological data, soil classification, vegetation, hydrology, wildlife, meteorological information, pollution health and safety, industry and employment, transport systems, utilities (gas, electricity and telephones), emergency services. The primary linkage mechanism among geographical information, as mentioned already, is the underlying *geodetic framework*, which will enable the correct overlay of data and secondary mechanism is the identity of the

land parcel in the cadastral database. The implementation of these mechanisms is done through the *data model*, which will enable the unified storage and processing of data from different sources and perhaps from different geodetic reference systems.

In any of the concepts mentioned earlier about the cadastre, the basic unit of reference is the *land* or *cadastral parcel*, which is an unambiguously defined unit of land, within which homogeneous rights and interests are legally recognized. It envelops a continuous area of land with a history of legal interests [102], [108]. As a three-dimensional division of the earth, the parcel may include super-adjacent and sub-adjacent rights in addition to surface rights. *Homogeneity* regards the nature of rights that will define the land encompassed by the parcel, while *continuity* is concerned with the spatial extent of the property interests [102]. Parcel-based land information systems can be classified according to the information contained in the system and/or the primary purpose of the system. McLaughlin [102] identifies three categories:

1. *fiscal* cadastres which are developed primarily for property valuation and taxation;
2. *juridical* cadastres which serve as a legally recognized record of land tenure; and
3. *multipurpose* cadastres which encompass both fiscal and juridical cadastres.

The definition of a multipurpose cadastre given by McLaughlin and Nichols [102] is an attempt to unify the two different kinds of cadastre. However, information that exists as part of a cadastre can prove useful to a multitude of other applications. In this way, a multipurpose cadastre encompasses all possible applications that may retrieve or contribute information. It is considered, that this kind of a multipurpose cadastre, can serve as the basis of a *national GIS*, not only due to the possible country-wide extent of the data, but also because of the plethora of information held.

1.4.1 Data modelling

In most cases, GI systems are built *ad hoc* for a specific application, since the assets and time allocated for the application development may usually restrict the functionality of the system. In the future, the need for the system to expand will usually occur and the degree will mostly depend on the development strategy that has been followed initially, which seriously affects, the so called open architecture of the system. Different system development methods have been proposed and followed throughout the technological evolution of recent years. The thesis purpose regarding the open architecture issue is to introduce the object-oriented approach to the development of a multipurpose data model. It is clearly the researcher's opinion that a multipurpose data model serves as the basis for a multipurpose land information system which is used to manipulate cadastral information.

According to Worboys [135], "... a *model* is an artificial construction in which parts of one domain (source) are represented in another domain (target). ... The purpose of the model is to simplify and abstract from the source domain." Whatever belongs to the source domain is transformed into the target and vice versa. Models are useful if they simulate the source domain as closely as possible. For this to be feasible, some kind of mechanism that moves from one domain to another is necessary. This movement is called a *morphism* [135], which is defined as "...a function from one domain to another by preserving some of the structure of the source domain in the translation."

In the context of a GIS, models can be operated in many situations, from the application for which a GIS may be built, right down to the lowest level where the information is stored in the computer. Each application, whether it is related or not with the geographic space, has a domain, called the *application domain*, which in turn has a respective *domain model*. It is the way by which concepts are organised. Application domains may be information technology related. In the case that an application needs computerized support, then a connection from

this application domain to the physical location of the information has to be established. This linkage mechanism goes through three major stages, by mapping the conceptual models of every stage. Thus, (assuming that the application itself is stage 1), starting from the application domain model (stage 2), and well before any computer support is involved, a *conceptual model* must be used (stage 3). Examples of such models are the network, the hierarchical, the relational and the object-oriented. If this model has to be permanently stored in the computer, then the next stage is to manipulate it through a *logical computational model* (stage 4). This can be a database management system or a persistent programming language (as discussed in chapter 2). Finally, since information has to reside on some kind of physical media (e.g. a hard disk, or an optical disc) then the *physical computational model* is involved (stage 5). This model implies a strong, and usually not very flexible, connection with a specific hardware/software platform. To summarize, a series of mappings from the application domain model to the other end, the physical computational model must be defined and implemented. This modelling sequence can be schematically represented as in figure 1.2 (modified from [135]).

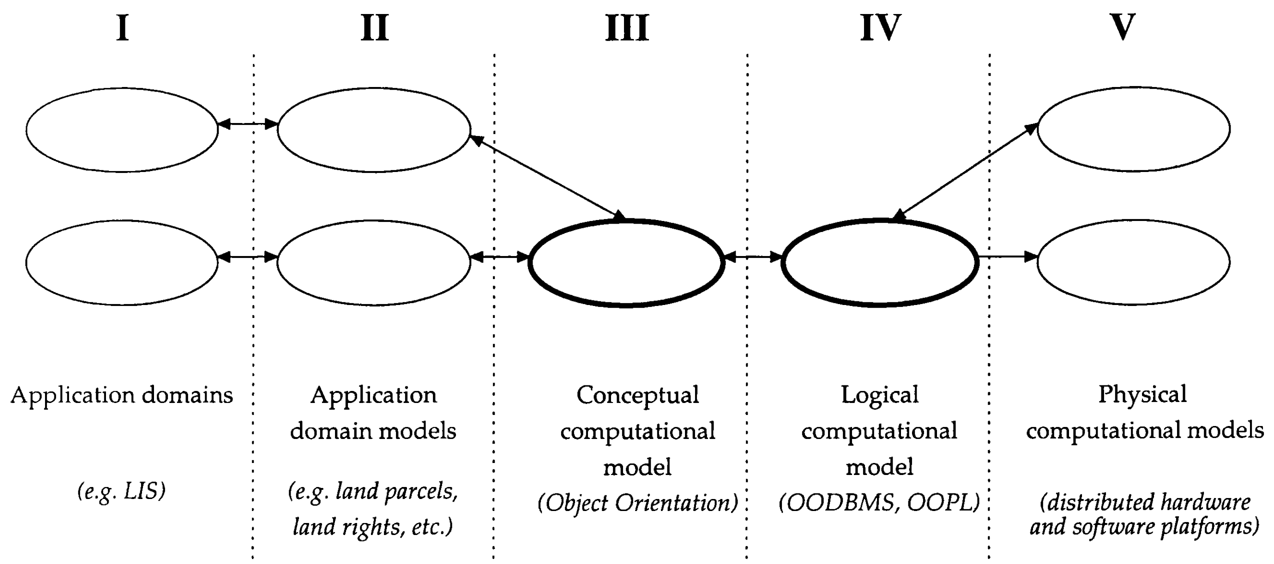


Figure 1.2: Domains and the modelling stages in a multipurpose GIS

In figure 1.2, arrows represent the translation that takes place from one do-

main to another and ellipses represent domains. The constituents of any source domain may, for example, be entities, relationships, processes or any other phenomena of interest. A model is particularly useful when it simulates efficiently the source domain and the "movement" between the domains (source-target) is fairly straightforward. Examples of application domains are the administrative areas domain, electrical supply networks, road and railway network management, cadastral information management, to name but a few. The last one, as an application domain is the focus of the research. Each of these application domains includes a domain model that is specific to the domain itself. The conceptual computational model takes into account the computational model. In this domain the relational or the object model can play a significant role. The logical computational model takes into account a specific paradigm. For the entity-relationship model the conceptual model primitives are mapped into entities and relations. For the object model the primitives are individual objects. The physical computational model is a specific combination of a hardware and software platform. The rapid technology evolution in this domain, results in a diverse variety of products, each designed for different application domain requirements. The role of standardization is also important in this phase. The actual presence of data is at the physical model, while the rest of the models are used to describe the domain data type and behaviour.

When a data model, regardless of the level involved, is built to manipulate information that has some kind of geometry as a part, then the model is called a *spatial data model.* If this spatial information refers to more than one timestamp, then the model should be called a *spatio-temporal data model.* It must be noted that usually a spatial or spatio-temporal model is mostly used within a conceptual computational model.

Application domains and their models are the application expert's task to define. The concern of this thesis is that a multipurpose data model at this level should be capable of supporting many application domains, mostly in terms of their spatial component. This is feasible by defining a finite set of application domains,

specifying their domain models and finding their union regarding the geometrical information they involve. This union of conceptual entities will be the basis of the development of a multipurpose model. This, however simple it may sound, in fact is rather difficult to achieve, simply because of the large amount of different application domains that are based on spatial data, along with the many variations found among data models of the same approach (not to mention the different data structures that these domains may employ). An alternative to this is to employ a flexible conceptual computational model, which contains a basic set of objects[1] that include some kind of geometry, and additionally provides support for the definition and creation of higher level, aggregate objects. Moreover, these aggregate objects should be able to communicate themselves, regardless of the data model schema used to define them. The object-oriented model is considered to be one of the most suitable for this purpose.

In order that an initial model can be built, a single application domain has been chosen, namely a land information system for the cadastre. It is envisaged that later, this model may be augmented with additional features and artifacts so as to provide support to more than one higher-level model from different application domains. This incorporation will definitely affect the model, but only in terms of its contents and not of its structure. Schema modification is always part of model development, both during development and during utilization within a computerized system.

The *computational model* concerns the computational context within which the transition from the application into the computerized system takes place. Examples of such models are the entity-relationship (ER) or the object-oriented (OO), which has been adopted in this thesis. The latter, as mentioned earlier, is considered to provide support for modelling multiple application domains. Many different approaches can be found, in composing the basic artifacts of an object-oriented model, as discussed in chapter 2. For this thesis, the Unified Modelling Language (UML) has been adopted as a tool to visually construct the data model.

[1]The term here is used in the sense of an identified unit of information within the domain.

UML has been adopted as a standard by the Object Management Group (OMG) and is spreading and being adopted fairly quickly for similar data and software modelling purposes. Specifications of the UML can be found in chapter 2.

The *logical computational model* represents a lower-level modelling than the computational model and regards specific computational paradigms (like relational against object-oriented databases). In this thesis, the object-oriented paradigm has been adopted, whenever applicable (that is: object-oriented DBMS, object-oriented programming language, object-oriented GIS). The main reason for this selection is because it eliminates the mismatch between object-oriented designs and non object-oriented platforms, as discussed in chapter 2.

The *physical computational model* is of no particular concern within this thesis, since it regards low-level implementations closely coupled with a specific computer platform (software and hardware). Moreover, interoperability is one of the key goals of the proposed model, which requires loose coupling with the physical implementation. However, as already mentioned, this is not always feasible since the final implementation in terms of testing and evaluating the design, involves specific platforms. This research involves Java as the object oriented programming language, Java's serialization mechanism which is used to achieve persistence, and LaserScan's Gothic ADE, as the GIS platform. All of these, individually, constitute specific physical computational models. In this way, interoperability relies on the functionality found in these platforms. This issue is discussed thoroughly in the relevant appendixes.

1.4.1.1 Providing support for multiple application domains and platform independence

For a successful open-architecture GIS environment, multiple application domain models must be able to "morph" quite straightforwardly, into the same conceptual computational model. In other words, they should be able to organize the concepts and information using similar artifacts. Additionally, different applica-

tions should be capable of using the same software, regardless of the way they handle data. It is inevitable therefore that the conceptual model must be flexible, complex and rich enough to anticipate similar needs found in the application domain model. Consecutively, for maximum hardware and software independence, the logical computational model must provide the capability of "morphing" into multiple physical computational models. In other words, the design methodology must be as independent as possible from the software and hardware employed. Finally, the software itself should also be loosely coupled to the hardware. This can be achieved either with a single and very efficient logical model on which the design must be based on, or with a series of interfaces from the various conceptual and logical models to the lower level physical ones. The later approach has been proposed by the OpenGIS consortium. Therefore the conceptual computational model with the respective logical computational model are of unambiguous importance in the modelling process for an open architecture and multipurpose data model. The link between multiple application domains and different hardware/software platforms within the same environment is considered to be the *object model.* The MPOOST model has been designed to serve multiple application domains and uses the Java language to achieve this goal. This is discussed in section 3.14.

1.5 Data in a Geoinformation system

With regard to the processing stage and the system, the data that are stored in a database can be (figure 1.3, page 47):

1. *Raw data*, which have not undergone any kind of processing.
2. *Structured data*, which have only undergone processing regarding their structure and not their content.
3. *Interpretations*, which have undergone processing which affects both the structure and their content.

Data in the later two categories are derived from the previous ones through specific processes provided by the functionality of the system. Data in all categories must be stored and later retrieved and visualized not only in isolation, but with respect to their origin.

Regardless of the conceptual data model that a system works with (such as relational/object), data can always be divided into two main categories: *spatial data* are those that have a spatial dimension relating to the real world (the earth) and *aspatial data* those that do not have. This distinction has proven to be necessary and it is only a matter of the level in which it takes place. For instance, in the relational model paradigm the distinction lies within the user's initial conceptual view, while in the object model (in which an object as a concept may have spatial and non-spatial data), the distinction may be done in the lowest level, the physical. As a result, robust and effective, query oriented, indexing techniques can be applied. Moreover, for an effective data-level analysis such a distinction is essential. This distinction however does not mean that spatial and aspatial data will be treated differently, as one of the basic requirements is to use a single store which can hold both types of data.

1.5.1 Spatial Data

Spatial data may be characterized as either *raster* or *vector*. The difference between the two data models is mainly based upon the consideration to discretize or not the spatial framework to which data are referring. The two models are strongly related to the instrumentation and methodology used for acquiring them. Vector data can also be thought as objects, which are discrete entities with sharp boundaries. With the respect to the Euclidean space, vector data in a GIS may include up to 3 dimensions and may refer to a specific date/time. We could therefore speak of spatio-temporal data/framework, or perhaps for 4D vector data or framework. Raster data are continuous by nature and are related to a specific date/time on which these were captured. Moreover, for vector data be-

sides geometrical information, the topological relations must be built and stored as well. *Topology* refers to the connectivity of the geometric objects in a space where the objects are embedded. Therefore concepts like continuity, boundary and connectedness may be defined, stored and used within the vector component of a GIS. Since vector objects can be aggregated, thus comprising more than one part, topology may be divided into *local*, refering to the internal structuring of a complex object, and *global* or *external* refering to the connectedness of a geometric object to its neighbourhing objects [25].

Independently of the model employed, spatial data always represent attempts to capture the truth about a real-world phenomenon. Since, axiomatically truth is never known, therefore spatial data are liable to errors and inaccuracies. Error may refer either to the deviation from the truth, which is described by the term *accuracy*, or the precision degree of the measurements, which is expressed by the term *precision*. The term error here also includes the statistical concept of *variation*.

In the context of an integrated GIS, both vector and raster data must not only be incorporated but also be interrelated. This implies an efficient data model which should capable of storing both kinds of spatial data as well as the processing history of derived information and optionally the computational process itself. Therefore, the need for storing information on how information is derived from data is necessary. An example is the raster to vector transformation (and vice-versa), where the derived vector data set must refer to the original raster data set, whence it originates.

In conclusion, there is a strong need for an integrated, generic, modular, open architecture data model which should be capable of including such spatial information as vectors, the third dimension and time, raster data and provide the facility to easily expand itself to suit specific application modelling needs that will occur in the future. However, it has been proven difficult by a model itself to incorporate efficiently the diversity of data, concepts and their inter-relationships

that are encountered within this heavily demanding requirement. In effect, a combination of many models, or perhaps the expansion of a single model has to be considered. The most promising of the conceptual and logical models that can incorporate all the above required functionality is considered to be the object model.

1.5.2 Spatial data models and their functionality

1.5.2.1 Field versus object data models

A commonly encountered debate in GIS centers on the dichotomy of spatial models. The effect of this cascades down to spatial structures and implementations. Many researchers (Chrisman [36], Peuquet [116], and Worboys [135]) have discussed these two different ways of modelling. *Field based* models (in the bibliography also encountered as raster models) treat information as a continuous distribution of an attribute value over a specific spatial extent, which has been canonically tessellated. This distribution can be mathematically formalized as a function. *Object based* models (or vector) in contrast, are sets of discrete, identifiable and tangible entities, separate from each other, which usually do not imply any kind of canonical spatial tessellation.

An interesting commonality between the two models is that they both treat the geometric space in a discrete way, since otherwise any manual computation and therefore any computer-based implementation can not take place. In the case of raster data, a canonical segmentation of space, is basic prior to data capture and storage. *Cells* (or pixels) are the fundamental unit that raster models are using to store data. The respective unit in the object model is the *point*, where the segmentation of space is irregular and defined by the existence of identifiable objects of interest in the real world. Effectively, points and cell centroids are of the same geometrical nature, but only the spatial pattern in which they form upper level entities (e.g. lines/polygons or images respectively) is the feature

that makes the distinction. In the proposed object model, this is the key feature that links both raster and vector data in the same conceptual model.

Particular to object based models is the *topology*, that is the connectedness of objects in the spatial framework. Topology in this case has to be defined *ad hoc* and it is strongly related to the object contents themselves. In the case of raster data, topology is known *a priori*, since any kind of grid patterns are of known connectedness. Therefore raster topology is not of major concern in the field model paradigm.

Usage of either of the two models strongly depends on the real world phenomenon or entity that is being modelled. As Couclelis [39] notes: "The points, lines, and polygons that do exist in the geographic world are practically all human artifacts, falling into two broad categories: a) engineering works such as roads, dykes, runways, railway lines, and surveying landmarks and b) administrative and property boundaries.". The concept of an artificial entity is clear, as opposed to a natural entity or phenomenon. For any artificial phenomenon it is more convenient to use the vector model, while for natural phenomena, sometimes, is more appropriate to use the raster model [2].

1.6 Required functionality in a GIS

Regardless of the aforementioned characteristics of spatial data, what makes a geo-information system more demanding than other information systems is mainly the idiomatic nature of spatial datasets, which include [122]:

- Extremely large volumes, which are introduced by the recent developments in data collection techniques, such as GPS, photogrammetry, satellite images etc., in which a significant enhancement in precision has been achieved over the past decade. The main issue posed here is how this vast volume

[2] However, this is not a rule, as for example polygons can adequately model lakes (sharp boundaries) but not forests (fuzzy boundaries), although both objects may be natural.

of data can be manipulated efficiently in terms of access, processing and visualization. Hence, the subsystems of a GIS affected here are the database management system as well as the graphics kernel, which both should be incorporating intelligent algorithms which provide the required fastness. As it it discussed further below in this section, mechanisms like indexing and clustering found in object oriented databases may help to tackle the problem of fast access. Additionally, hardware development may also be the answer, mainly for visualization performance enhancement.

- Complex modelling. Spatial data, regardless of being in a vector or raster format, are known to be composed of other parts either of spatial data or aspatial(attributes). The data structure used to model real world geographic objects should be able to include recursively defined objects as well as transitive closure operations. Sometimes the degree of complexity and recursiveness is increased, and the relational model is known not to provide such facility (this is discussed in chapter 2, section 2.2). In object-orientation, complex modelling is an inherent feature.

- Variability of the data associated with the entities of the same type (e.g. georeferenced objects, temporal objects). This characteristic, introduces the need of associating an entity of a certain type with a multitude of different other entities, when all data are modelled as entities. Object orientation provides the built-in feature of object links, which properly addresses this issue.

In order to tackle the issue of required functionality, a classification is necessary, based on the type of operations on data. The functions that a geographical information system provides may be divided into five general categories [82]:

1. Data acquisition
2. Preliminary data processing
3. Data storage and retrieval

4. Spatial search and analysis
5. Display and interaction

The above categorization is illustrated in figure 1.3.

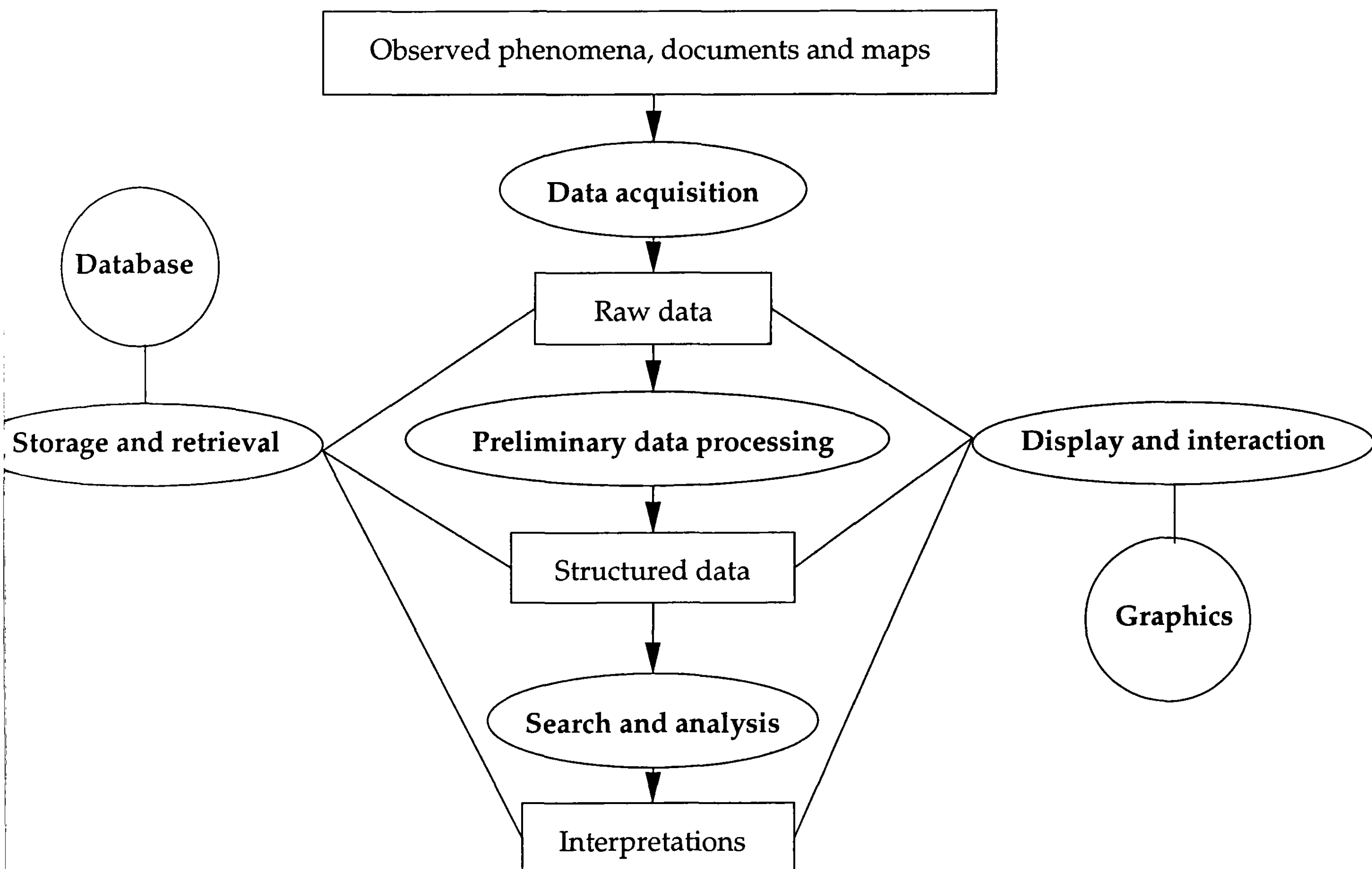


Figure 1.3: Functionality of a GIS. (Modified from [82])

Data acquisition includes all the spatial data collection mechanisms such as topographic surveys, satellite imagery, photogrammetric procedures or even socio-economic surveys. Data in this stage may be either in digital format or in any conventional format, such as documents, lists, maps, questionnaires, charts etc. The first stage of data processing is therefore to transform them into a digital format within the computer system, namely the database. By the term *data*, the set of human concepts and observations that are related to the system itself are also included. The term has to be broad enough in order to include information such as rules (real world or data integrity constraints), real world and system

events, facts, metadata, such as procedural information (lineage) about the data collection fashion of a specific technique or the processing algorithm used to produce a specific information set. In order to manipulate such a complicated set with its inter-relationships in the database context it is necessary to thoroughly analyze it and find a suitable model, initially for the organization (conceptual model) and later for the storage (logical and physical model).

As already mentioned, data that are stored within the GIS database may belong to one of the following three main categories:

1. raw data;
2. structured data;
3. interpretations.

Each category results after some processing of its predecessors. This process flow is always one way, hence information will never be transformed back into the source data. However, information as output from one process may serve as data for another process, either within the same system or when exported to an external system. Therefore all data must be stored in the database always along with the processing information involved. In this way, the user is aware of the origin of information, and may issue queries that are related to all three categories *and* their processing history. It is, therefore, of great importance for a GIS to be built upon an intelligent data model, which has to incorporate processing information storage. This historical information about data is called *lineage* and it is considered to be a *metadata* component. The object-oriented model and its artifacts *do* provide this kind of functionality.

Data collection and update is an expensive phase of the GIS development. Therefore, the database must have such functionality in order to preserve and exploit the assets involved to built such a system. As Egenhofer and Frank [59] comment, a geo-information system integrates data from various resources into a single and homogeneous system, therefore it requires powerful and flexible data models to serve tasks. Such data model characteristics can be:

1. Sophisticated treatment of real-world complex geometry. The term complex is used to denote the aggregational nature of real-world objects.

2. Representation of the same data at different conceptual levels of resolution and detail. This signifies the need for abstraction and generalization support.

3. Management of history and versions of objects (versioning).

4. Combinations of measurements with different resolution and accuracy.

5. Correctness of the information within the database. This is a major concern in the computer science field. It primarily suggests that any data or information stored must be of known correctness degree, whether this is high or low. Guaranteeing correctness involves mainly the utilization of formal specifications. In the context of geographic information systems this is still under research, since no specific formal methodologies have been developed. So far, the most popular way that correctness information is being stored is through metadata.

   Specific GIS requirements have direct impact on the mechanisms used within the database management system. Authors such as [35], Frank and Egenhofer [64] and Egenhofer and Frank [57] suggest some of them along with proposals towards providing mechanisms to enhance the functionality.

6. Clustering and indexing enhancement. The data set stored in the database should be seamless and not divided into user-visible map sheets, unless for the stage of visualization or hardcopy production. The performance penalties that are introduced are significant in the case of large data sets, mainly because of the frequency of disk access necessary to retrieve the required data through a user defined query. In order to minimize the number of accesses, spatial storage clusters must be integrated into the functionality of the database subsystem. These techniques are used to store spatial data on the disk so that spatial neighbors are also neighbors on the storage device [57]. Obviously, this offers an enhancement to the problem, however it does

not fully resolve the issue of clustering, since objects are accessed by their non-spatial components as well. Since simultaneous multiple clustering can not be implemented (it would result in data being duplicated) the alternative is to use *indices* based on different user needs when querying and accessing objects.

7. Single database: both spatial and non-spatial data should be stored and manipulated within the same database.

8. Nested transactioning to support nested objects: for this requirement the basic principles of a transaction should be augmented with more than the basics (atomicity, consistency, isolation and durability).

1.7 Data concepts

This section discusses in more detail the main research issues that were examined during the elaboration of the research. Issues include characteristics of the data and the concepts, spatial and non-spatial data, temporality, metadata and the required functionality.

1.7.1 Spatial and temporal data modelling

According to Egenhofer *et al.* [55] a *spatial data model* is "... a formalization of the concepts humans use to conceptualize space". It is very important that the model is readily transferable to the computer. Two basic different approaches are well known within spatial modelling, namely *layer based* versus *object based* approaches. In the earlier, a unit may have several values for every layer, while in the latter complicated data structures are allowed.

A data model is a means of representation for database design. It provides a tool for specifying the structural and behavioural properties of a database and ideally should provide a language which allows the user and database designer

to express their requirements in ways that they find appropriate, while being capable of transformation to structures suitable for implementation in a database management system [140]. As a result, a proper data model should:

- be efficient;
- support data integrity checks;
- offer a natural logical structure to its user;
- demand easy maintenance and extension.

Choi *et al.* [35] divide the overall object-oriented model into two main modules: the geographic and geometric data model, with the former being on top of the latter. This distinction is based primarily on the fact that geography involves the passage of time, while geometry is rather static. Additionally, the models are different not only because of the possible user views, but on the computational functions that are used within each module. Almost every class that models a geographic object has a corresponding class in the geometric data model, which contains the geometrical information. This very basic distinction between time-varying geography and static geometry provided the incentive to start the component based analysis in chapter three.

Data modeling can be seen as a three-staged process [131]. In short, real-world entities are part of the continuous space. These entities have the property of being indefinite in terms of being computationally defined in a precise manner. Through the abstraction approach, they are transformed into discrete objects and therefore are part of a discrete space. In this way, they loose their indefiniteness and become computationable. However, in order for these objects to become storable an additional transformation is necessary called representation. The objects resulting from this transformation are part of the representation space. To make a relation with the modelling stages described earlier, continuous space belongs to the application domain model, discrete space is part of the conceptual model and representation space is part of the logical model.

Two geographical objects may have identical geometry, and therefore refer to identical geometrical objects. Their identity, however, remains distinct, therefore resulting in two different geographical objects in the database pointing to the same geometrical object. This approach is widely used ([35]) and has the advantage that geometrical data redundancy is attenuated. Moreover, indexing on geometry is enhanced.

Geo-referenced objects in a database may employ four major properties (Egenhofer and Frank [59]):

1. Persistence
2. Geometry
3. Graphical representation
4. Temporal reference

These three latter properties suggest the main modules that classes should be grouped in as it is discussed in chapter four. Persistence of objects, refers to whether they are stored in permanent storage media or they remain transient, and therefore destroyed whenever the application that these originate stops executing. It is moreover considered to be a property strongly coupled with the other three mentioned above. It is discussed in more detail in chapter two, section 2.5.2.

Worboys [131] uses the notion of *simplicial complexes* to represent geographic data embedded in a plane (two-dimensional). He additionally proposes a classification scheme for these geometric artifacts. These include 0-extent, 1-extent and 2-extent objects, having 0, 1, or 2 spatial dimensions respectively. The formal definitions of every spatial class that he proposes are based on homeomorphism that they show to a mathematically defined geometric object. *Homeomorphism* relates two spatial objects (one is homeomorphic to another) if they are topologically equivalent, i.e. one is transformable into the other using a reversible topological transformation. In this way, a strand is homeomorphic to a straight line segment, a loop is homeomorphic to a circle. Simplicial complexes are likely

to be used in the case of modelling discretized continuous surfaces like Triangulated Irregular Networks.

Geo-referenced objects are constructed as aggregates of geometric objects, the former as a group holding an ordered set of the latter. Regarding their geometries, geo-referenced objects may employ a degree of complexity e.g. they are allowed to cross themselves. This results in the geo-referenced objects having a reference to the same geometric object more than once. If such a case is allowed, it means that they should be treated as sets. Therefore, set-based operations must also be included.

When attempting to model a group of objects to create a category in an application domain, this should be done with respect to three properties they must hold [101]:

- Be identifiable.
- Be of interest to the design.
- Be describable.

An initial categorization for geo-referenced real world objects is necessary to be addressed. More specifically such objects may be categorized:

1. In terms of their precise geometrical definition:
   - *Fuzzy*, if they are not precisely defined in terms of their geometry. Precise definition refers not only to the simple aggregation, specialization and association relationships between complex and primitive spatial objects, but also to the degree of uncertainty by which a higher level geo-referenced object refers to its geometric component (e.g. point A is within a forest area). The existence of such objects causes problems to object-oriented modelling and therefore has to be simplified, either by a) eliminating geometric fuzziness, so that sharp boundaries are defined for fuzzy objects through rules or functions, or b) by allowing a degree of uncertainty when referring to their geometry (possibility range).

- *Discrete*, if they involve precisely defined geometry with no uncertainty regarding their geometric parts.

2. In terms of their existence in the real world:

   - *Abstract*, if they do not exist as tangible objects but as concepts. Examples are land property, right-of-way, etc.
   - *Tangible*, if they exist as real world objects (e.g. tunnel, road, building etc.).

3. In terms of the human role on their existence:

   - *Physical*, if they exist as tangible objects, without the human intervention. An example is the physical drainage network (rivers, lakes), .
   - *Technical*, if they exist as tangible objects, only because of human intervention. Examples are road and railway networks, power distribution facilities etc.

Additional classification may be imposed by other application domains and how they group spatially related information. These classes will actually act as container classes (as described in chapter two).

It is obvious that an object may belong to more than one category. It might be physical objects which humans have modified in a way (like the diversion of a river), or objects that exist as technical or physical but are considered to be abstract as well (as administrative/property boundaries which coincide with rivers or fences). This type of classification is very similar to the one that Couclelis [39] has proposed. If this type of classification is used, then the concept of multiple inheritance must be supported not only by the high level conceptual model but by the implementation environment as well.

1.7.2 Data concepts

As it has been stated, data are considered to include more than static attributes but also the rules, facts, constraints or logical procedures that may be encountered in the context of spatial information. Therefore, prior to any data model construction, a thorough examination of the data themselves has to be performed.

1.7.2.1 Spatial data and their geometric space

Chrisman [36] has recognized that space may be conceptualized in two distinct ways, either as a set of locations with properties (absolute space, existent in itself) or as a set of objects with spatial properties (relative space, dependent upon other objects). The implementation of the two different views of spatial embedding can be done using the relational or the object-oriented data models correspondingly. Regardless of the concept of space that may be adopted, it is necessary to distinguish amongst a) the spatial framework domain, b) the attribute domain and c) the object domain that references the first two domains. Effectively an integral view of data, that involves both the spatial and the aspatial aspects can be achieved. Thus, real world objects can "point" to other objects, which in turn have their spatial embedding and their own set of properties. Worboys *et al.* [141] have proposed a general object model for planar geographic information where objects are spatially referenced to two kinds of objects: spatial and aspatial. The spatial framework that they propose is embedded in the Euclidean 3D or 2D space ($R^3$and R^2 respectively), which results in a model of space described by records of coordinates [135]. Since the purpose of this research is not to examine mathematical spaces, but rather to adopt one, the Euclidean metric space for the geometry along with the common coordinate geometric calculus and formulas will be used whenever necessary.

With regard to space, spatial data that are stored via the model which will be developed, will be embedded either in the Euclidean three-dimensional space (R^3) or in two dimensional space (R^2), whichever is applicable, depending on the cat-

egorization of the spatial data. Real world objects (RWO) that will be stored in the database, will be embedded in the R^3 space, while the cartographic representations of the database objects will be embedded in the R^2 space (see figure 1.4). The adoption of the two-dimensional geometric space is based primarily on the media that a cartographic end product is produced, usually either on screen or as a hardcopy.

In order to establish a terminology convention, for the rest of the text the term *entity* will refer to a real world entity, tangible or not. An *object* is defined as the representation of an entity in the database system, *feature* is the geometric simplification of an object in the cartographic database, and *symbol* is the cartographic depiction of a feature in a hardcopy item or on the computer screen [3]. The relation between real world entities, database objects and cartographic features has to be made clear: the earlier, correspond to real world objects while the later correspond to database objects. This sequence does not always apply reversely, *e.g.* as a 2D cartographic feature may correspond to more than one database objects, due to the elimination of the third dimension. Schematically, figure 1.4 shows this modelling sequence.

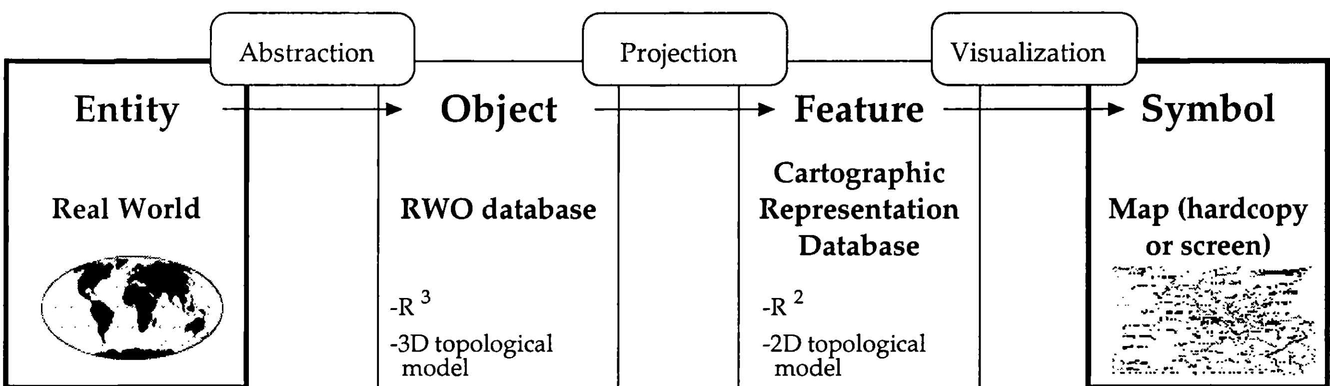


Figure 1.4: Discrete modelling sequence, spaces and topological models

Real world objects are embedded in a three dimensional space. Cartographic end products, in their majority, are two dimensional, including even modern virtual reality interfaces, which nonetheless result in a 2D projection of 3D objects. In

[3]This convention is quite similar to the proposal by NCDCDS

this way, the representation of a real world entity is at the same time separate but linked from the cartographic depiction. An advantage is that digital cartographic end products can be used in isolation for analysis purposes. The down side is that the transformation of objects from 3D to the 2D topological model is necessary, therefore some of the information may be lost. Entities are referring to real world entities that are tangible or not. Database objects are the representation of the entities that are stored in the database, while cartographic features are the depiction of database objects on the map, either on a hardcopy or on the screen.

1.7.2.1.1 Coordinate systems The two major coordinatizations of the three-dimensional Euclidean space that the spatial sciences have adopted are:

- *Geographical coordinates* (longitude and latitude). This kind of system requires the definition of a mathematical parametric surface (either a sphere, ellipsoid or geoid). A point definition, besides longitude and latitude might include the distance from this reference surface. The definition of the surface requires the specification of its parameters along with a major orientation. In the case of the geoid, which shows a significant more complex degree in its definition, a number of parameters must also be defined thus it is highly unlikely to be used as a reference system. It is considered that the most appropriate mathematical surface to be used is the ellipsoid.
- *Cartesian coordinates*, which can be either three- (e.g. x, y, z) or two- (e.g. x, y) dimensional. The earlier might be used in geocentric systems, which do not involve any kind of distortions, while the latter in map projections which introduce three types of distortions: angular, areal and linear. The first type of coordinate system requires the definition of an axis triplet, the location of the axes start point, along with the major orientation, since axes are considered to be vertical to each other. The coordinate system used in a map projection requires more parameters, such as the surface that is used to project the earth on *etc.*

The selection of the coordinate system mainly depends on the extent of the geographical dataset [48]. When large in extent, for example national datasets are considered, it is most likely either the geographic or the geocentric coordinate system will be employed, since both do not introduce any kinds of distortion. However, map features are considered to use a map projection coordinate system so that they can be drawn on a two-dimensional media, such as paper or screen. Both types of systems refer to the same Euclidean space, and there is a one-to-one transformation between them, so that a point has a set of geographical coordinates and a set of Cartesian coordinates. Operations and algorithms must be supplied to transit from one system to another.

1.7.2.2 Vector data

This category of spatial data includes discrete entities which in turn have a geometry, that derives from the coordinatized representation of the space and a topology, which describes the spatial relations between them. An effort will be made to incorporate the third dimension of the data, thus mathematically speaking, working in the R^3 space.

1.7.2.2.1 Topology of vector data Space can be theoretically viewed as an infinite set of points. Some phenomena though, are not defined for all points in a continuous space. Only a subset of them is of interest [97]. Topology is about the relationships between geometric entities in a discretized view of space.

According to Worboys [135], *topology* is the study of topological transformations and the properties that are left invariant by them, and he identifies a set of topological properties. In order to define topological relations between spatial objects, the modelling primitives must be specified as well as the set of possible operations between them. Topological relationships can be modelled using the graph theory [97], where a *graph* is defined as a finite non-empty set of nodes together with a set of unordered pairs of distinct nodes, called as edges.

1.7.2.2.2 Modelling primitives Theoretically, the notion of the simplicial complex is used as a fundamental unit to build up a formal model ([76], [135], [56], [25], [105]) as a finite set of simplices. A simplicial complex is any complex geometry, which is composed of simpler geometric structures, the simplices. Simplexes are to be the building blocks of larger structures. These are defined with regard to their dimension as a set of connected points.

The simplices which are usable in a GIS environment are [135] :

- A 0-simplex is a set consisting of a single point. Geometrically it is equal to a point.
- A 1-simplex is a finite straight-line segment. It is composed of 0-simplexes. Geometrically it is equal to a line segment.
- A 2-simplex is a set consisting of all the points on the boundary and the interior of a triangle whose vertices are not collinear. It is composed of 1-simplexes, and geometrically equals to a triangle.
- A 3-simplex is a set of all points on the boundary and the interior of a tetrahedron (a volume with 4 faces). Graphically, in 2D space equals to a set of triangles that in pairs share a common edge.

A more formal definition of the simplexes can be found in Breunig [25].

Definition 1.1 (simplicial complex) *A simplicial complex is a finite set of simplices with the properties that:*

1. *the intersection of the two respective simplices is either empty or a face of both simplices.*
2. *with every simplex each of its faces is also defined.*

The boundary ∂C of a simplicial complex is a simplicial complex of dimension (d-1).

A simplicial 3-complex is a set of connected 3-simplices. Each 2-simplex is bordered by at most two 3-simplices and every 1-simplex is surrounded by at most

two 2-simplices. Every 1-simplex connects two 0-simplices and the boundary of a simplicial 3-complex is a connected sequence of 2-simplices.

Simplicial complexes employ their geometry by using coordinates, their topology by maintaining references to other complexes, and their metrics (distance, angle) by having them derived from its geometry. Metrics are defined in the metric space, which satisfies the properties of identity, symmetry and triangle inequation. It is noteworthy to mention that both topology and metrics may be derived from geometry, although the reverse does not apply.

Other spatial object types can be generalized into 0-extent, 1-extent, 2-extent and 3-extent, with the prefix number denoting the dimensionality of the type. 0-extent types include sub-types like point and node. 1-extent types include subtypes like line segment, line, arc, chain, directed or undirected loop. 2-extent types include subtypes as area, region, cell.

These are similar to simplicial complexes, with the addition that the latter can be aggregated to form the previous. These primitives can effectively be used to model complex 3D structures.

The bibliography is rich in work for modelling and classifying spatial primitives. Major contributions are work from Egenhofer *et al.* [60], De Floriani *et al.* [44], Guting and Schneider (1993), Worboys [132], Worboys and Bofakos [138], Worboys [134].

According to National Committee's for Digital Cartographic Data Standards NCDCDS *et al.* [109], definitions of fundamental spatial objects:

- A point is a zero-dimensional spatial object with coordinates and a unique identifier within the map.
- A line is a sequence of ordered points, where the beginning of the line may have a special start node and the end a special end node.
- A chain is a line which is part of one or more polygons and therefore also has a left and right polygon identifier in addition to the start and end node.

- A node is a junction or endpoint of one or more lines or chains.
- A polygon consists of one outer and zero or more inner rings.

These types of geometric entities can be also modelled using simplicial complexes. However, in some cases it is considered to be costly, e.g. when maintenance of the internal triangles of a polygon is required.

The geometric model involved in the research is partially based on the above proposal by NCDCDS as it has been augmented with more features (e.g. polygons with an indefinite level of recursively inner polygons).

1.7.2.3 Raster Data

Raster data need to be transformed and interpreted as objects, so that effective object-based analysis is possible [7].

1.7.2.4 Temporality

Time is inherent within spatial data. It may be the date that the entities (represented by data) existed in the real world, the date that they were captured or the date that they where stored in the database. Usually, systems are built to take into account only the date regarding the existence of entities in the real world. This does not allow the user to have an overview of previous states of the information. Temporal support, enables an audit trail on the date. Moreover it allows analysis to be performed of the information known by the database at a particular time.

Temporal logic is a tool that enables its users to achieve better understanding of the nature of time. Its primary aim is to clarify the content, to elaborate the consequences, and to elucidate the interrelationships among the axioms of time in general (Rescher and Urquhart 1971, from Al-Taha [2]). Unlikely with standard first-order logic, where a sentence is either true or false exclusively, in modal logic the truth of a statement is closely related to its temporal component, and can

only be answered with a temporal context. By the term *temporality* we address the aspect of information which is a consequence of the effect of time.

Two basic approaches have been used in temporal reasoning for databases: a *change based approach* and a *time based approach.* However, the two concepts are inseparable, since change occurs over time, and time is meaningless without any change [2]. This is also the conclusion from the conference held in 1992, regarding "Methods of Spatio-Temporal Reasoning in Geographic Space" (Frank *et al.* 1992), where participants agreed that space and time reasoning can be inherently linked to each other, since we can infer time from space and vice versa. This distinction on time and change based views of temporality serves as the basis that temporal reasoning models have been built: whenever a change occurs in the real world, this is surveyed and stored in the database, creating a single snapshot. Eventually, the database will soon evolve to a series of snapshots, an approach which is change-based. The latter concentrates on recording changes valid at a certain time point.

The change based approach shows two major modelling shortcomings, namely:

- *Instantaneous actions*: actions are supposed to be of zero duration, therefore their results are immediately apparent to the system. This is not true, however, for most real-world actions that cause change to happen. In a more formal way, a wait operator is not allowed in this approach.
- *Lack of concurrency and overlapping actions*: actions cannot be simultaneous or overlap among them.

Other limitations involve future prediction not based on changes in the past, complex action results (ramification problem) and selectivity of change on complex objects (frame problem).

The time-based approach recognizes the passage of time as the only change which is unaffected by anything else. In this manner, time can be considered regardless of the space, and the state of the space can be thought of as a consequence of change due to time passage.

In any case, when a GIS is enabled with a temporal extension, it might be called a *temporal GIS* (there is more than one suitable adjective as discussed later) and it is therefore able to answer questions of type "when" *and* "where" instead of only "where". Temporal extensions also serve as the basis of handling historical data and make change management feasible. It must be noted that at the time that this chapter was written, temporal GI systems are not very common in the market, and the few available are based on OO technology (e.g. Laser Scan, Smallworld). This is mainly because the relational model does not provide artifacts to directly implement versioning, hence change management is not straightforwardly supported. However, it is possible to use relational technology to construct temporal GI systems and the literature is rich in relevant work (such as in Langran [94]).

The simplest way of introducing temporality to spatial information is through *time stamping,* which is the attachment of a time value to any other data value. This is very generic and it implies that all values are subject to change, something that it is usually partly true since some of the data remain unchanged for the whole life cycle of an object. In any case, time stamping only takes into account the time that data were captured. This directly points to the date that they existed in the real world. Of major importance here, is the level at which time stamping should occur, especially when aggregate objects are involved, so as to avoid data redundancy: should aggregate parts be time stamped or should the whole object? In the first case, the aggregate parts specify the time stamping of the whole, therefore a generalization relationship is implied, while in the latter case, the whole object is timestamped, resulting in transfering this information to its parts. This implies a kind of inheritance function from the whole to its aggregates. Both solutions are viable, although only one should be followed throughout a design. The answer usually relies on the frequency of updates and the nature of queries. Snapshot approach can be found in the relational model approach. Al- Taha[2], uses multivalued attributes to represent temporal data within the context of a relational DBMS. Each time a new change occurs to an attribute, a

new value will be added to its multivalued data set, and only changed attributes will receive a new data field. This approach saves a considerable amount of disk space, however it results in performance cost whenever a time-specific snapshot of the database must be reconstructed. This problem occurs regardless of the type of model used (relational or object-oriented). He suggests that a solution to this problem would be a hybrid snapshot and multivalued attribute model, so that the base snapshot from where the reconstruction begins is as close as possible to the query time. He concludes that this kind of snapshot approach when modeling temporality may result in data redundancy. However, as if to make things more complicated, there's more to the time that geo-referenced objects existed in the real world, and this is immediately obvious if we think of the computer system within which the information is entered. In a more holistic view, Worboys [135] considers temporality in spatial databases as a two dimensional space: it is composed of the *transaction* time dimension, which is the time when transactions take place within an information system and the *valid* time dimension, where the events occur in the application domain. Effectively, every spatio-temporal object stored in a system, is related to a bi-temporal interval, that is a two dimensional array, where the real world events and the system events are captured. Langran [95] names these two aspects of time as *logical* and *physical.* She separates the concept of space from time and introduces the concept of *temporal topology* where versions of real world objects have a state in cartographic time, and mutations of these objects correspond to events in cartographic time. The concept of bi-temporality is still being researched, although through this literature review it can be seen that the concepts have been both individually and together considered for quite a while now. Valid time can be found as real world time [94], effective time (Ben-Zvi 1982), extrinsic time (Bubenko 1977), logical time (Dadam, Lum and Werner 1984). The concept of transaction time can be found also in the literature under different names: database time (Langran 1992), registration time (Ben-Zvi 1982), physical time (Dadam et al. 1984). Snodgrass [121] introduces the notion of *bi-temporal time domain* as a set of bi-temporal

chronons. This concept is furthermore enhanced by Worboys [135], as mentioned earlier. Every bi-temporal chronon represents a portion of valid time and transaction time, measured along orthogonal axes. He notices that these two different times require different bounds and granularities. Additionally, transaction time is derived from the system clock and therefore independent of any application [122], a fact that separates design from implementation. Arctur [6] suggests the additional concept of *user defined* time, which can be of a specific data type (e.g. string), not interpreted by the system, and therefore not indexed. However, when the enhancement of value-based user query is involved, indexes should exist for all types of attributes. Other notions of time might be *survey time*, when the change was observed. It is usually treated as user defined time and does not require any special treatment by the DBMS, so it can be modeled as a standard attribute [122].

Normally, time-stamping values should not be constrained. However, Story [122] suggests that transaction time is always upper bounded by the variable instant known as *now*, which is nothing more than the system's own clock time, as no database transactions can be known to have occurred in the future. It is the researcher's opinion that this is not the case always, since there is information which is stored in the database and refers to a future time, e.g. foreseen future transactions such as triggers and scheduled events. Nonetheless, this observation raises the issue of *constraints* on time values, which should be addressed with respect to the granularity under which a time-stamp value is being recorded. These constraints are highly application specific, and are discussed in chapter three, which deals with the analysis of the model, where the context of a cadastral information system is being examined.

Consequently, not all systems supporting time serve the same purpose since there are different kinds of time. Regarding the kind of time involved, temporal databases are divided into four main designs, according to the kind of time-stamping supported [135]:

- *static*, when there's no time support. Such a system stores a snapshot of space-related information at a moment in time;
- *rollback*, (Snodgrass and Ahn 1985) when there's only system time support. These systems store the time when the transaction was done, usually providing the capability of restoring the old state of information;
- *historic*, when only real world time is supported. This approach enables the development of historical GIS, since space related information is time stamped with respect to when it existed in the real world; and
- *temporal*, when both system and real-world time are supported. This combination of the two different kinds of time is called *bi-temporal* [135].

When building a temporal type within the model it is is considered good practice, mainly for consistency reasons, that this type remains constant throughout the model as well as the level at which it is integrated [122]. The data model developed is considered to be on a temporal system, therefore supporting bi-temporal space. Additionally, the bi-temporality is built-in in the early stage of design.

One of the major conclusions from the 1990 NCGIA workshop [12] was the identification of two significantly different paradigms of time: (1) time as a *continuum* and (2) time as a *sequence of intervals and changes* caused by events. This differentiation of the concept of time is quite similar to that which separates temporality, based on the application involved, into scientific and engineering [2] which is discussed later on in this section. *Mutations* (or events) are concepts of zero dimensionality, similar to the concept of point in space, which switch from one version (or state) to another. The latter are considered to span one dimension. Changes in real world entities through time are reflected in the database by changes in the attributes or relations of database objects. Storing the same object twice with changed attributes would result in major data redundancy. By *versioning* database objects, only changes are stored, therefore redundancy is eliminated. Hornsby and Egenhofer [77] have proposed a classification of temporal change based on object identity along with a set of operations that either

preserve or change the object identity. These operations can also be applied to composite objects. A similar approach has been adopted in the object design phase of the model.

If a system supports temporal data, then the user will definitely require to query the database against not only the values themselves but with respect to the (single if not bi-) temporal domain stored. This introduces the need for fast data access techniques to be devised so as to create indexes based on the temporal data component. If different granularity is to be used, then this should differentiate data, which may be indexed on the time value using a quad tree model [7]. Additionally, indexes should be built for both world and transaction times. Indexing is part of any object-oriented RDBM system, where one can find features like custom indexing structures, which are considered to be useful. However, all the above make obvious that introducing temporality is imposing storage and performance cost on creating and maintaining indexes, which may very often be complicated mainly due to the possible large amount of data.

In a bi-temporal GIS, where both system and real-world time are supported, every value that is subject to change in the future or has undergone changes during the past, is associated with a time stamp. This time stamp is nothing more than a value denoting either a specific moment in time (instant) or a period that the value refers to. Hence, types of time-stamp values can be [7]:

1. *Instants* at different granularities.
2. *Spans*, *intervals* (or periods). These may include the instant limits that define them, therefore being called *closed intervals* or may not, and thus being called *open intervals.*
3. *Relative times*, where there is a base time instant and a period refering to this instant (e.g. a week since 13th of August 1998).

The precision that time stamps are recorded is called *granularity.* Different levels of granularity are usually required indicating that this should be recorded

separately from the time stamp value itself, instead of using a global fine granularity throughout the dataset. In the case where granularity is not known or not provided, it can be a derived attribute, calculated from the precision of the time stamp. The accuracy of time stamps is an issue similar to any numeric or attribute value found in a GIS and should be treated accordingly. A direct impact of granularity, in combination with how instantaneous time stamps are considered to be, is that an instant can be an interval when the granularity increases. E.g. a day such as 13th of August 1998 may be an instant, while this day may include a whole 24h period. It is obvious that granularity precision is determining whether an instant may be considered to be a period at the same time. Nonetheless, additionally to the aggregative link between a period and the instants that are used to define it, some kind of generalization/specialization relationship exists between intervals and instants. However, a more simplified view might be that of *periodic time*, since in the specific example "every day" has an associated 24h cycle, just as an hour has a period of 60 minutes and so on. Temporality is also related to the application domain to which the data refer. At the workshop held by NCGIA at the University of Maine ([12] from [2]) it was concluded that there are two distinct applications for a temporal GIS: *science-oriented* and *engineering-oriented*. The major differences between the two types of applications are based on:

- specific requirements, where engineering applications aim at solving specific problems with clearly defined conceptual entities, while scientific applications are characterized by their genericity and fuzziness of the concepts; and
- incremental changes, where updates in engineering applications are expressed in discrete time units, where in scientific applications time is fairly parametric and quite often not well defined.

This distinction should be always taken into account prior to the design of the temporal module of a system. Specific and parametric time in engineering appli-

cations do not necessarily need a mathematical formalism, unlike science oriented problems, where analysis cannot be initiated without it.

Al-Taha [2] notes some of the main issues in temporal reasoning:

- *The imprecision of expressions*: keywords such as "now" or "today" are quite similar but, depending on the context, point to different levels of granularity. Therefore similar keywords should be avoided as much as possible in a potential temporal language of any kind within a GI system. If used, this should be done with caution, and sentences made out of these keywords should always result in a single interpretation.
- *Variety of points of view*: different applications demand different timestamping granularity. Therefore a system should be capable of handling multigranular information. This is similar to multiscale spatial information, since time granularity can be parallelized to space precision.
- *Variety of actions*: actions are different in their duration, sequence, and effect. Their classification and formalization are therefore compulsory.
- *Conceptual issues*: namely the debate between points versus time intervals and their association. An important issue posed here is that of inheritance, meaning that a time point might or might not inherit from the time interval it belongs to. In this way time extrapolation is not always applicable to time intervals.
- *Structural issues*: issues here are related to the mathematical model for the representation of temporal entities, like precedence, discrete time versus dense, complete vs. incomplete and bounded vs. unbounded.
- *Logical issues*, which have to do with the formal logic involved when defining the temporal behaviour of a conceptual model.

How are temporality issues tackled using an object-oriented approach? Object orientation, as it is discussed in chapter two, is a conceptual modelling procedure where most of the required temporal features such as identity, events, states, and

versions are inherent, therefore time and change modelling of spatially referenced objects is considered to be an extremely straightforward procedure. One of the problems best addressed using the object-oriented methodology (rather than any other approach) is that of versioning. In an object-oriented environment, the version of an object is implemented as another object of the same class and created whenever a change occurs in its state. Of course, not all state changes should produce a new version. In any case, succeeding versions should be linked, therefore the class of the objects should include a self-referenced relationship pointing to the previous version of the object. In this way, every versioned object should be associated with a *version chain* which holds all the information about the history of the object. Moreover, temporality is not only an aspect of the data and information stored in a system. The data model itself is subject to changes, as it is never found to be satisfactory throughout the existence of a system, therefore the need for the data model evolution is constantly present. Consequently, a way of modelling the model change itself is necessary. This change towards aligning the model with the real-world phenomena and current required system functionality is called *schema evolution*, and as far as design is concerned, it should be taken into account before this stage initiates. To better explain the problem, let us consider the following: in an object-oriented data model, a class is a structure which consists of attributes and operations definitions. It is assumed here that references to other classes are implemented as attributes as well, containing the class name that they point. When the definition of a class is modified, one (or more) new class(es) is(are) produced. It is apparent that, a version tree of this class must be maintained. An object as an instance of a class, is a structure containing attribute values and operations which conform to the definitions found in its class. Some of these attributes are references to other objects. Values are subject to change, therefore a version tree of the object history must be maintained as well. Among the special attributes found in an object are its identity as well as its class definition. Therefore, every object should include not only the class name that it belongs to, but the class version

as well. Computationally, the platform used should directly support schema evolution, otherwise the designer is responsible for implementing such a strategy, which is far more complicated, disfunctional and design-dependent than with a built-in support. Fortunately, most OO methodologies support such a concept either directly or indirectly, through extensibility mechanisms. Some, but not all object-oriented computer environments directly support evolution management of the model (e.g. Java). Schema evolution is discussed in detail in chapter two.

In the context of the current research, and with regard to multipurpose design, some of the issues that must be addressed regarding object and schema versioning are:

- Versioning of real-world objects is not part of the adopted UML approach. It has to be implemented either implicitly, through self associations between the same object, or by introducing new artifacts which is feasible through extensibility mechanisms such as *stereotypes.* In both cases, regardless of the manner it is implemented, the extension is considered to be very straightforward.
- Classes should be classified into versionable and non-versionable. This is considered to be additional work for both the analyst and the designer. It is also a starting point to argue whether an object should be versioned and when this should occur. And when multipurpose design is involved, it is noticed that different applications may require different approaches regarding when a spatial object should or should not be versioned.
- Time stamping might be a problem among different versions in a distributed environment, when each version uses its own clock [7]. The issue of *time synchronization* arises among systems with different clocks. An obvious solution to this might be that all systems would be synchronized to a common clock in the network, an approach which is widely encountered.
- Modeling temporal data requires the introduction of appropriate concepts for reasoning about change. A conceptual framework is therefore necessary

whereupon operations and predicates will be based [7].

- Should time granularity be stored within data, or be a derivative parameter? An approach might be to assume the degree of time granularity given the precision of timestamps (e.g. "12:00" is a value with a time granularity of 1 minute, while "twelve o'clock" implies a time granularity of one hour). However, this might be a problem whenever this kind of information is not known precisely, since the above assumption is not always correct.
- When should a new object be created? In other words, which are the attribute changes that signify when a new version of the object should be created with a new and different identity? The answer to this question is extremely application dependent, something that poses a serious obstacle to the multipurpose design. In order to address this issue, the data model must support parametric class versioning.

Version information regarding objects is included in the same module of a system as the predecessor object. This means that an object A belonging to a class C may be versioned to produce a newer object B, which also belongs to class C. But when the schema itself changes, i.e. when the class definition is altered, any class version information should be part of the schema itself as well as part of the metadata [7]. Versions of objects (including classes) are usually different from their predecessors in one or more attribute values. Some of the values might remain the same. If all of the object attributes are to be stored this might result in major data redundancy. To avoid this situation, recording changes only to a complete base state for each version, is a good practice since data duplication is avoided [122]. In this way, only the changed attribute values are stored, while the unchanged ones point to the immediately older object. A whole object can later be reconstructed by tracing the version tree and combining all the changes from the base state. This solution is considered to be useful in order to achieve databases small in size. Issues that arise here are that the number of steps required to reconstruct an object must be the minimum, so as to decrease access

time and storage overhead. More complicated issues arise when geo-referenced objects are found to have multiple geometries as part of their structure, with each geometry refering to a single and different timestamp. This case is common to aggregate spatial objects, which often change partially in one of their components (e.g. geometry). Multiple geometries are also useful to model a geographic object in different scale, whenever this spatial object can be generalized. Issues that arise here are that:

1. any non-spatial attributes related to this object should be the same throughout all geometries, hence no attribute-based change modelling is allowed;
2. time stamping of each geometry is rather awkward, if not impossible, since each geometry should have a 1-1 relationship with a valid timestamp, either world time or database time [7].

Worboys [137] when discussing the difference between objects and events comments that although events can be treated as objects within the object-oriented approach, they do belong to distinct categories since events *occur*, but objects do not. He considers space as the container for objects while time is the equivalent container for events. A process can be defined as a composite event.

1.7.2.5 Metadata

Metadata are any data that are used to describe a data set. Jones [82] has identified the following component hierarchy of metadata for spatial datasets:

1. **Data exchange format**
   (a) Specification of data storage format
2. **Data summary**
   (a) Source, classes of data, areal coverage, date, scale
3. **Lineage**
   (a) Agency of origin

   (b) Method of data collection:
      i. Primary survey techniques
      ii. Secondary data sources
         A. Digitizing method
   (c) Dates updated
   (d) Processing history:
      i. Coordinate transformation
      ii. Data model transformations
      iii. Attribute transformations

4. **Coordinate system**
   (a) Type of coordinate system
   (b) Map projection parameters
5. **Spatial data model**
   (a) Specification of primitive spatial objects
   (b) Topological data stored
6. **Feature coding system**
   (a) Definition of feature codes and classification system
7. **Classification completeness**
   (a) Documentation on extent of usage of classification system
8. **Geographical coverage**
   (a) Overall extent
   (b) Detailed specification of coverage if not complete
9. **Positional accuracy**
   (a) Statistics on coordinate error
10. **Attribute accuracy**
   (a) Statistics on attribute error

11. **Topological accuracy**
    (a) Methods of topology validation employed
12. **Graphical representation**
    (a) Graphical symbolism for each feature class
    (b) Text fonts for annotation

A serious issue here is whether the values that these fields may take, have to be standard or not. This is not always applicable. For instance, the spatial data model can be generally either vector or raster. However, the topological accuracy information can be expressed in various ways. If multiple datasets must coexist in the same database, then metadata information must be standardized. Nonetheless, if this type of data are stored along with the core dataset, they can describe satisfactorily the contents of it. Concepts of metadata as discrete entities relating to spatial objects can be easily modelled and manipulated within the context of the object-oriented paradigm.

Data quality is a major subcomponent of metadata, relating to errors and the sources of them. Burrough and McDonell [30] identify seven main groups of factors that affect the quality of data: currency, completeness, consistency, accessibility, accuracy and precision, sources of errors in data, and sources of errors in derived data and in the results of modelling and analysis. Chrisman [37] proposed that data quality is a measure of *fitness for use* of data for a particular task. Moreover, data quality information provides the handle for long-term maintenance. This kind of measure gives the degree to which a particular data set can be used for a particular purpose, and it is considered to be of major importance in a data model that functions as a multipurpose one. Different applications will require to use the same dataset. For this to be successful, it will depend strongly on the quality requirements posed by the application in conjunction with the stored quality information about a specific dataset.

The International Cartographic Association[70] proposal based on analysis of spatial data from different countries has reached a consensus and identified six

major components of data quality:

1. *Positional accuracy*, which applies for spatial data only. The number of its components is as many dimensions as the spatial object is embedded in. In the case of the usual 3D Euclidean space positional accuracy consists of 3 components, x,y,z in the case of Cartesian coordinates or f,l,h in the case of geographical coordinates. Positional accuracy should definitely be an input to the system rather than derived, since derivative accuracy values are in most cases incorrect, as they strongly depend on the processing history of the spatial dataset.

2. *Attribute quality*, which applies for any non-spatial attributes of an object. In this category, the temporal quality of the spatial dataset may also belong, if the temporality of the geo-referenced entities is considered to be an attribute.

3. *Logical consistency*, which encompasses any constraints that the data should follow. This includes topological constraints among spatial objects.

4. *Completeness*, which is defined as "...an attribute describing the relationship between the objects represented in a data set and the abstract universe of all objects" [104]. It describes the exhaustiveness of a set of features, including both spatial and attribute properties [109]. For the degree of completeness to be assessed, a detailed description of this abstract universe is necessary. This term is used here to denote all possible real world entities or phenomena that may be perceived by a human observer. Moreover, it is obvious that such a concept is quite general, as it contains a multitude of entities, therefore its detailed and complete definition is considered to be extremely time-consuming. The answer to this is the adoption of standards within every application domain that the entities belong.

5. *Lineage*, which refers to the process responsible for creating a dataset. This includes a description of the source material from which the data were derived, the methods of derivation, all transformations involved, reference to

control information used, and a description of the mathematical transformations of coordinates.

6. *Temporal information*, which is about the date or period that information refers and regards their validity in combination with the date that the data are being examined.

Within the above six major categories, some of the most important data quality research issues that have been suggested recently (International Symposium on Spatial Data Quality 1999) are:

1. Uncertainties in real world entities.
2. Uncertainty propagation in spatial operations.
3. Uncertainty in remotely sensed images and classification.
4. Error in Digital Elevation Models.
5. Spatial querying with uncertain data.
6. Spatial reasoning with uncertain information.
7. Uncertainty in geographical and environmental analysis.
8. Visualizing uncertainty in spatial data and analysis.
9. Meta-data and model for GIS data.
10. Spatial data models for uncertain objects in GIS.

Not all of the above are always significant. It always depends on the application domain which of the above components are addressed and which are the relevant data to be stored. Each one of the above components may use continuous or discontinuous variables to quantify quality. Every measured continuous variable has additionally three aspects: *precision*, *resolution* and *accuracy*. It is very common mistake to confuse these three concepts and to misuse them. The term *error* is also widely used and it is mostly connected to the concept of accuracy as it refers to the deviation of a single feature's value (e.g. estimated or measured) from its true value, or a value which is axiomatically set to be true. More specifically:

- Precision refers to the repeatability of a value which has been reported. As such, it is used to characterize measurements (like lengths, angles or coordinates) and always applies to an individual feature, or a set of features that have been collected using the same methodology. Most commonly the *standard deviation* σ is used to quantify precision. Precision itself is a continuous variable, usually taking integer values.
- Accuracy refers to the "nearness to the truth", and always applies to a set of features (dataset). Accuracy is quantified by means of the error concept and most commonly the root mean square error (RMSE) is used. Accuracy is also a continuous variable, with values taken from the domain of real numbers.
- Resolution refers to the minimum unit of measurement. It applies only to continuous variables (like coordinates, angles, lengths, areas, volumes etc). Resolution is also a continuous variable.
- Error is a quantitative indication of the accuracy. It is a statistical concept based on some assumption of the nature of the measurement process.

.

Precision depends mostly on the computer system used to store the data. It has been stressed earlier that there is a need for space discretization in order that computation may take place. However, this results in inaccuracies. We must also take into account that any computer implementation involves integer arithmetic. As [135] states, "... the Euclidean space (2- or 3- dimensional), is topologically dense and capable of an infinite amount of precision." (i.e. resolution). It is obvious that coordinates will be expressed as real numbers, which support infinite levels of resolution. In this way, however, space is not computable in terms of integer arithmetic. Therefore, it has to be discretized into computable objects, namely points. In this way, the geometric domain used is a "...finite connected portion of the discrete Euclidean space" , either 2 or 3 dimensional, consisting of a set of points, called P. Any line segment that belongs to this space, must

have its endpoints as members of P. Consequently, if two line segments intersect, their common intersection point must also be a member of P. It has been shown [135] that this results in accuracy loss, since straight line segments cease to be "absolutely" straight, but these are only a rough approximation to the real straight line they model. This also depends on the resolution used in the model.

Regarding models employed to quantify positional error, three types can be identified: point, line and polygon error models.

From the data collection stage, right through to using a map as an end product, information is not only liable to inherent errors but to errors being propagated from one stage to the next. Jones [82] identifies the five stages that may increase the error of the information:

1. primary data acquisition errors, attributed to human usage and interpretation, resolution of instrumentation, or possible misuse of instrumentation;
2. secondary data acquisition due to shortcomings in data collection method (e.g. digitizing), shortcomings of the source document, failure to obtain adequate metadata;
3. data manipulation and analysis due to changes in the coordinate system, changes between spatial models, interpolation procedures, integration of both geometrical and thematic data from various sources (e.g. overlay), compromises and generalizations of classes;
4. data transmission due to numerical degradation because of inadequacies of the transmission media;
5. usage due to data misuse by the user.

Errors introduced during any of the above stages, need to be mathematically modelled not only individually but in combination as well.

An interesting question that arises regards how different applications that access the same dataset can be aware of the degree of fitness for use for the purpose they serve.

Finally, it must be emphasized that, regardless of how metadata information is modelled, space, time and attributes interact and it is considered that quality information plays an important role in these interactions [37].

1.7.3 Functionality requirements

During the development process of modelling for a geo-information system, many technical issues that arise in a low level stage of the design (*e.g.* implementation), are reflected in a higher level (*e.g.* design), therefore making the application domain expert to interact with, and modify the conceptual model that it is used. For example, some operations on objects, like splitting or versioning, require the introduction of new object identifiers. As already mentioned, it is the application domain expert's decision to specify the cases when this should be done.

Jones [82] groups functions within a GIS into three categories (figure 1.3):

1. Data acquisition.
2. Preliminary data processing.
3. Search and analysis.

Data acquisition involves the process of obtaining data from various sources. The data collection phase is also a part of acquisition (primary data acquisition). The concern of the thesis is the final digital format in which they are stored temporarily, prior to final database storage. This phase involves other techniques and method specifications (e.g. photogrammetric procedures, land-surveying techniques, GPS usage etc.). Secondary data acquisition involves data that are in various digital formats. The system must be capable of transforming these formats to the internal data representation, first conceptually and later logically. Examples of such formats are the DXF, DLG, NTF, for vector-based data and TIFF, ERDAS for raster based data.

Data storage and retrieval involves the mechanisms provided by the GIS for the persistence of the information. Data, in all levels of processing, are stored and

later retrieved. Vector and raster, object or relational are all conceptual data models that enable the organization of spatial information prior to storage. The logical data model is the way that the database management system organizes the conceptual data models in a specific class of hardware and software configuration using database specific concepts as files records and indexes. In most cases of a commercial platform, these tools are built in, therefore little concern is given as to how they operate, unless the performance of the system is of an unsatisfactory level for a specific application. In this case, storage techniques must be reviewed and new ones must be proposed. Within the object paradigm, where encapsulation restricts all the information of an object and makes it available only through its interface, a workable solution is imperative for an effective indexing storage and retrieval.

White [125] classifies user queries out of a data set into topological, metrical, about consistency, purely geometrical, and geometrical/geographical. Examples of these types of queries are (referring mostly to 2D entities):

- **Topological queries**. These are related to the connectedness among the objects, including the possible internal structure of the object. Examples are:
  1. What 0-, 1- and 2D elements does the map comprise of?
  2. Which 2-cells cobound a particular 1-cell?
  3. Which 0-cells terminate particular 1-cells?
  4. For a particular 0-cell, which 1-cells are incident?
  5. For a 2-cell, which 1-cells are incident (both bounding and interior)?
- **Metrical queries**. These are related to the geometrical attributes of an object. Examples are:
  1. What is the location of a particular 0-cell?
  2. What is the shape of a 1-cell? The answer might include metric details about length.

3. What is the shape of a 2-cell in three-dimensional space? The answer might include metric details about perimeter, 2D area and surface area (e.g. upon the physical earth surface).

- **Geographical queries**. These focus on the combination (overlay) of objects that do not necessarily belong to the same thematic layer or object class. Examples are:

  1. For any two regions A and B, does A equal B?
  2. What regions does a given region cover?
  3. What regions cover a given region?
  4. What is the join of two regions?
  5. What is the meet of two regions?

- **Geometrical and Geographical queries**. These are combined questions both on the geometry and the overlay of two or more objects. Examples are:

  1. For a particular region, what 2-cells are included?
  2. For a particular 2-cell, what regions include it?
  3. For a particular linear feature, what 1-cells are included?
  4. For a particular 1-cell, what features include it?
  5. For a particular set of points what 0-cells are included?
  6. For a particular 0-cell, what set of points include it?

Analysis functions are part of the core function group in a GIS, which emphasises the decision-making aspect of the system. Jones [82], summarises these query types as:

1. Containment search within a spatial region.
2. Proximal search.
3. Phenomenon based search and overlay processing.
4. Interpolation and surface modelling.

5. Best path analysis and routing.
6. Spatial interaction modelling.
7. Correlations, associations, patterns and trends.
8. Map algebra with gridded data.

The taxonomy presented by Jones refers to both raster and vector data but mainly for the 2D representation of space. Incorporating 3D spatial data sets into these categories, includes the consideration of volumes instead of regions and voxels instead of pixels.

Additionally to the above categorization, the *visualization* of information in any level is considered to be the visual way for information dissemination from the system to the user, in a specific spatial data set. Graphics functionality is of major importance in a GI system. Visualization techniques are developed in conjunction with the technological context. Recent advances include virtual reality systems. Visualization of 3D data incorporates techniques such as transformations, projections, 3D clipping, hidden surface removal, viewshed analysis, surface shading and rendering. The major concern in cartographic visualization is the adoption of proper symbols and rules that cartographic features are depicted and processed. In the object paradigm, these can be encapsulated in the static and dynamic behaviour of the cartographic object within the database. Generalization is a challenging aspect of the cartographic production process, where the geometry of a data set is modified in a manner appropriate to the visualization scale. Jones [82] summarises generalization concepts as *semantic*, which includes processes such as aggregation and classification, and *geometrical generalization* which includes elimination of objects, amalgamation of objects, dimensionality collapse, exaggeration, enhancement, typification, and displacement.

It must be noted that parcel-based cadastral digital outputs are not always in real need of very rich graphics kernel functionality (e.g. rendering support), but rather in need of algorithms fast enough to tackle the problem of large data volumes. When it comes to linking and combining spatial information from other

application domains (e.g. raster orthophoto images) and incorporating raster data, then this kind of functionality will be determinant for the successful visualization of these derived complex data sets. Generalization is also a procedure, which might not be of importance in large-scale parcel based diagrams. It will be of concern when small-scale representations are involved, where the abstraction of information has to take place.

An extended classification and taxonomy in the context of the LIS paradigm will be considered in the early phases of the thesis research phase. Special focus will be given on the three-dimensional and temporal nature of the required functionality.

1.7.3.1 Operations

Preliminary data processing functions involves operations such as structuring, classifying and transforming representations of data to make them suitable for further analysis. Examples are topological structuring, image classification, vector to raster (rasterisation), raster to vector (vectorization), interpolation to grid, triangulation, reclassification, and projection [82].

Operations can also be broadly grouped as attribute, distance/location, and using built-in spatial topology. Attributes are properties of real world entities that define their static nature. Burrough and McDonell [30] groups mathematical operations on attributes as:

1. Logical (Boolean)
2. Arithmetical
3. Trigonometric
4. Data type
5. Statistical
6. Multivariate

A taxonomy of spatial operations based on topology can be found in Worboys [135], with a classification of basic static spatial operations (unary and binary)

into general, set-oriented, topological and Euclidean. The static nature implies that operands are not affected by the application of the operation. His major concern is 2D space, which necessitates the expansion of these operators into 3D space. These operations may well work in object-oriented models, as a part of the dynamic structure of objects (methods). He also separates dynamic spatial operations on entities into dependent and independent of the creation of a spatial object. Dependent creation (when an object is created with reference to another object) includes operations as reproduce, generate, split and merge. Additionally, dynamic operations might be transformations, which are treated as update operations such as translate, rotate, scale, reflect and shear, which mostly effect the geometry of a spatial object.

Operations on spatial objects must be identified and classified. The initial proposal used is that of Worboys [131], which proposes four different categories of operations upon spatial object classes, mostly in the two dimensional space. In the current research, this proposal has been augmented to incorporate three-dimensional spatial classes.

Worboys [135] identifies four different types of spaces that spatial objects can be embedded within: Euclidean, metric, topological, and set-oriented. This classification dictates also the categories that operations upon embedded spatial classes are classified into. In short, such operations might be set-oriented, topological, metric and Euclidean. Some of these operations are members of spatial classes, while some other belong to utility classes, for the reasons already mentioned. More specifically, every category includes the following operations:

1. **Set-oriented operations:**
   (a) Equals
   (b) Member
   (c) Subset
   (d) Intersection
   (e) Union

   (f) Difference
   (g) Cardinality

2. **Topological operations:**
   (a) Interior
   (b) Closure
   (c) Boundary
   (d) Components
   (e) Extremes
   (f) Begin
   (g) End
   (h) Inside
   (i) Clockwise

3. **Metric operations:**
   (a) Distance
   (b) Length
   (c) Perimeter
   (d) Volume

4. **Euclidean operations:**
   (a) Bearing
   (b) Area

Additionally and with respect to the temporal aspect of a system, Langran [94] classifies the six fundamental functions of a temporal GIS as:

- inventory;
- analysis;
- updates;
- quality control;

- scheduling;
- display.

Al-Taha [2] also proposes a classification for operations (he calls then actions) on objects with temporal extensions, depending on their characteristics according to:

- their starting effect as:

1. immediate effect
2. delayed effect
3. earlier effect
4. periodic effect

- their permissibility as:

1. permissible act
2. non-permissible act
3. must action

- their results as:

1. single change
2. multiple changes

- other attributes

1. instantaneous
2. consecutive
3. parallel

It must be noted that some of the above operations may be solely related to time but it is mostly likely to be invoked on complex data with more than a temporal component, resulting in *spatio-temporal operations.* As far as the implementation is concerned, operations are an inherent part of object-oriented modelling, according to which, objects are modelled having attributes *and* operations. An important question is whether spatial classes should include general methods regarding geometric operations. If these operations are part of the spatial classes they are usually modelled as class members (which are common to all object instances) and they are only invoked through them, therefore tightly coupled with the spatial classes. If such operations are part of the so-called *utility* classes, they are independent of the spatial classes they may receive as arguments and therefore not requiring their presence, in terms of whenever their invocation is necessary. This finally results in general geometric methods independent of the structure of the spatial classes, and definitely fosters software reusability.

Regarding the existence of objects in a system, Worboys [135] identifies three fundamental dynamic operations, upon which all the rest are derived from:

- create
- destroy and
- update

Whenever a user retrieves information from a database, a transient copy is made, which matches the criteria set by the user. This is done in the form of *queries.* A query language is usually used as the primary user interface to the database, so that operations can be implemented and executed. Choi *et al.* [35] use the concept of *query-generated* objects which are transient objects that result from the processing of a query against the OO database. These type of objects may belong to a predefined class found in the schema or in a query generated class. Functions, invoked during run-time, are usually responsible for the generation of these objects. These objects might be stored, so that they can be retrieved later without having to re-execute the query. This of course does not apply whenever

object attributes are changed and have to be re-read from the database. Query modelling can be considered as the classification and parameterization of possible user queries, and it is closely connected to:

- the data model used to store information and
- the application domain(s) that the data are coming from

Current approaches on incorporating any kind of operations as programs in a GIS involved the separation of these programs from the data that they manipulate. Hence, programs were made tailored to the specific data structure that the application involved. The object-oriented paradigm suggests that programs are very coupled with the data that they operate on, since they are both part of an object. However, as far as the core geometrical algorithms are concerned, many of them remain the same either in procedural or object oriented approaches.

1.8 The object-oriented approach

This section aims to introduce the reader to the paradigm of object-orientation, what it involves and how it can be employed in a GIS. Chapter two is a more detailed discussion on object orientation.

1.8.1 Object-oriented analysis and design

Long before object-orientation evolved into a usable solution for information technology, the available modelling techniques were models such as the network, the hierarchical model and then the relational model with its relatively recent evolution into the extended relational model. The object oriented approach, although it is considered to be effectively used by the academics for the past 20 years or so, however it is only until the current decade that it gained popularity within the software market. The major difference between an object-oriented data model and the relational data model is what the latter had to offer: the introduction of concepts such as encapsulation, inheritance and polymorphism [88]. For this reason the object oriented model can be seen as the natural evolution of the extended relational model. Object orientation has proved attractive to certain GIS users because of the intuitive manner in which it offers the modelling functionality [30]. A characteristic of the object its modularity. It is comprised of a state, at a given time period and a functionality, that is a set of operations which impose a dynamic nature. In object-oriented design, the computer is divided into a number of smaller computers, or objects, each of which can be given a role like that of an actor in a play [84]. An *object* can be defined as something that plays a role with respect to a request for an operation. The request invokes the operation that defines some service to be performed [79].

In general, object-oriented development is a conceptual process, which is independent of any specific programming language or database management system. It refers to identification and organization of application domain concepts, rather

than their final representation in a language or database [120]. In contrast with the functional methodology, where the primary concern is to specify and decompose functionality, the object-oriented approach focuses on the identification of objects and on fitting the proper procedures around them. The object-oriented methodology consists of building a model of an application and then adding implementation details. The four stages are:

1. *analysis*, where the specifications of the functionality for the desired system are defined;
2. *system design*, where the overall architecture of the system is specified;
3. *object design*, where implementation details are added to the conceptual model from the analysis stage, and
4. *implementation*, where the object model from the previous stage is translated to a specific configuration of hardware and software platform.

Many different methodologies have been devised, proposed and used throughout the existence of object-orientation. One of the these methodologies used by the researcher initially is by Rumbaugh *et al.* [120] which is called the Object Modelling Technique (OMT). This technique uses three kinds of models to describe a system, as it comprises of the *object model*, describing objects and their relationships, the *dynamic model*, describing interactions between objects in a system, and the *functional model*, which describe the data transformations of the system. Each of the models mentioned above has a respective diagram, to graphically represent the concepts that it models. The object diagram is a graph whose nodes are object classes and whose arcs are relationships among classes. There are two types of object diagrams, namely the class diagram, which represents many possible instances of data and the instance diagram, which refers to a specific set of object instances. A state diagram is a graph whose nodes are states and whose arcs are transitions between states caused by events. The data flow diagram is a graph whose nodes are processes and whose arcs are data flows. In this way the three models are orthogonal parts of the description of

a complete system. Similar object-oriented modelling and design technique has been proposed by [21] and widely used in many system development examples. Jacobson *et al.* [80] have proposed a similar methodology which focuses mostly on the use case aspect of the object model. The most recent proposal for an object-oriented modelling technique has been by Booch *et al.* [22] and it is called the *Unified Modelling Language* (UML). It is considered to be a fusion of the three aforementioned methodologies proposed by each author individually. This approach is being used for the analysis and design of the model, and it is thoroughly explained in chapter two.

1.8.2 Summary of object-oriented concepts

An *object class* describes a group of objects having similar properties, common behaviour, common relationships to other objects and common semantics. The identification and creation of classes is done through *abstraction* and *generalization* from a specific case to a host of similar cases. Every object class has a set of *attributes* that characterize the class. For a given *class instance* (object) each one of its attributes has a value. A special attribute of an object instance is its *identity*, a unique identifier for the object. Additionally, every object class has a set of *operations*, which are functions or transformations that may be applied to or by objects in a class. An operation can be *polymorphic*, when the same operation name takes places on several classes but in different forms. An operation is implemented via a *method*. A *link* is a connection between object instances. An *association* is a group of similar links between object classes. Associations are possible to be modelled as classes, therefore having attributes and operations. *Aggregation* is the assembly of an object class from other classes. *Generalization* is the relationship between a class and one or more refined versions of it (superclass, subclass). Through *inheritance* classes may be sharing attributes and operations in a hierarchical manner. It is also possible for an operation to be overridden from a superclass to a subclass. Attributes and operations are *encapsulated* within an

object class but some operations define the interface of the object, that is the way operations may be invoked from other objects via messages. According to Cox [40] encapsulation is the foundation of the object-oriented approach shifting emphasis from coding technique to packaging, while inheritance builds on encapsulation to make reuse of code practical. A *module* is a logical construct for grouping classes. According to Mattos *et al.* [101], an entity, in order to be modelled, must be identifiable, relevant and desirable in the context of a specific GI application. Communication among objects instantiated from classes which are made by models written in different languages , is considered to be achievable by incorporating the CORBA standard in the early object-oriented data model design. This however, is not possible for all aforementioned platforms, namely Gothic ADE, since the platform that was used (version 3.0a.18) CORBA standard has not been incorporated into the system functionality. Other specific programming environment employ their own interoperability mechanisms, such the Remote Method Invocation in Java. These issues are discussed in more detail in chapter two as well as in the relevant appendix (chapter seven).

According to proper object-oriented design, five steps are involved in the specification of the model:

1. Requirements analysis within the application domain.
2. Identification of objects, classification, and extraction of classes. Specification of the class hierarchy.
3. Specification of relationships among classes, apart from their hierarchy, as well as their role and cardinality.
4. Specification of the structure of the classes (attributes and methods).
5. Specification of the signatures in the methods.
6. Specification of messages among classes.

This sequence of steps is supposed to be iterative throughout the whole development process. This will guarantee the *correctness* of the model contents, and

its *integrity* always according to the requirements that are to be met within the application domain. There is no specific limitation in the number of iterations necessary, since this may continue even when the system is operative. The only factor that imposes limitations is the time necessary to complete a full iteration in relation with the total amount of time allocated for a project. Additionally, the above sequence applies for every different viewpoint of the model.

1.8.3 Object-oriented database management systems

An *object-oriented database* is a collection of persistent objects whose behaviour and state, and the relationships are defined in accordance with an object-oriented data model. Object-oriented databases integrate object orientation with database capabilities [88]. One of the most important database capabilities is the *persistence* of objects. Khoshafian [86] suggests that functionality in an object-oriented database comprises mainly of persistence, concurrency, transactions, recovery, querying, versioning, integrity, security and performance. One of the most serious deficiencies within an object-oriented database management system is the performance under which it retrieves persistent objects. This originates from the concept of encapsulation and information hiding: an object's attributes are only accessible from the available interfaces it provides. Effectively searching for an object within a database requires sending messages to it and waiting for the response. The time required is more than it takes for a procedural program to access properly indexed data (e.g. SQL queries from a table in a relational database). Proposals to work out this problem involve the separation of the object identity from the rest of the attributes so as to stop being hidden. However efficient this solution is, it violates the principle of encapsulation and has been declined by a few researchers. Alternative viable solutions involve indexing and storage of objects based on their identity or using internal system addresses as object identifiers. All relevant issues are discussed in more detail in chapter two.

1.8.4 Functionality issues summary

It must be noted that in the context of the object-oriented system, embedding the computational process as an object is better in many ways with regard to a relational database combined with a procedural program. Because of the nature of raster data, it is very difficult to break down the continuity into separate units, such as objects [30]. Working out this mismatch involves treating raster images and data sets as objects composed of cells. Even when fuzzy boundaries are involved[29], the object-based model offers the required functionality for modelling. Burrough and McDonell [30] have also summarized the advantages and disadvantages of the object-based paradigm.

Chapter 2

Review of Object Orientation

2.1 Introduction

The purpose of this chapter is to investigate the effects of adopting the object-oriented paradigm in the context of a Geographical Information System. This is accomplished in two steps: initially, through an introduction to the approach of object orientation, in general, as it is being utilized within the information technology domain. Secondly, a special focus is given, whenever necessary, on the context of an object-oriented Geographical Information System, through discussion of the issues (benefits and problems) that are encountered.

The structure of this chapter is as follows: first, the reasons for moving from conventional methodologies to object orientation are discussed. Next, through a somewhat extended, literature review, an attempt is made to capture the concepts of object orientation, as they have been devised and used by various authors and methodologists worldwide. The third section is about how these concepts are used and implemented within specific computing environments, such as database management systems, programming languages and GIS platforms. Finally, a short description of the most popular and historically important object oriented environments is included at the end of the chapter. Special attention has been given to the environments used for the implementation stage of the research.

2.2 Moving from conventional models and technology to object-orientation

In general, it can be said that object orientation represents an attempt to overcome the problems encountered within the conventional approaches, namely functional analysis and design techniques, relational data models and databases and procedural programming. More specifically, several studies [59] have shown that conventional models and systems (including the relational) are not suitable to be used within the context of geographical data management. In short, when conventional approaches are involved, the major limitations to be confronted are:

- limited data types provided, hence difficulty when modeling complex nested entities;
- limitation of semantic expression;
- lack of grouping code with data;
- impedance mismatch: lack of support for data types found in programming languages such as structured or long unstructured data;
- lack of support for concepts like aggregation and generalization;
- lack of support for temporal data;
- lack of support for recursion, a crucial feature in the GIS domain e.g. for modeling multi-scale data [59] or using tree shaped index methods;
- lack of support in DBMS for multidimensional searches and search structures [112];
- very poor support for single storage space, where both spatial and aspatial data may reside.

It must be noted that almost all of the above problems are encountered when attempting to model the real-world geometry in terms of well defined objects. Many proposals regarding the extension of the relational model to incorporate

the storage of spatial data can be found in the literature (Egenhofer and Frank [59] include a large list with references). Many of these proposals claim to overcome the aforementioned problems by suggesting techniques at the design conceptual level. They, sometimes, are awkward to implement, however efficient they might be. Hence, they only pinpoint the difficulty of the conventional methodologies to handle real-world complexity.

Conventional databases are good at managing large amounts of data, sharing data among programs, and fast value-based queries. They are not very good at modeling the relationships among data, since everything must be represented as a series of two-dimensional tables.

Worboys [135] clarifies the notion of instance uniqueness in the relational and object models: "In E-R model, each entity should be identified uniquely, (a.n. via providing unique entity names in the entity schema definition) . Each occurrence of an entity is uniquely identified by giving values for the identifying attributes. Thus, the E-R model is a "value-based" model. Additionally, in relational databases the uniqueness of each record is characterized by the *primary key*, which is usually dependent on other attributes within the same record, and results in a unique combination of values for each record. If attribute values change then the primary key value changes as well. Within the GIS context, the attributes of many spatial objects do not exist independently, which is a requirement of the relational model [122]. However, in the OO model, an object occurrence retains its identity even if all its attributes change their values. An object ID is independent of its attribute values".

A solution to the problems that the relational model imposes, is to extend a relational database to accommodate geometry. However, join operations are expensive both time-wise and storage-wise, since semantic links have to be maintained through integrity constraints. This idea has resulted into the Extender Entity Relation Model (EER).

Object orientation disregards the way that the EER is based upon (tables) and provides support for general data types and nested objects. It includes the concepts of encapsulation, inheritance, and polymorphism. Object-oriented databases extend the relational data model by relaxing the first normal form condition[1], since they support sets of atomic values, tuple-valued attributes, sets of tuples (nested structures), set and tuple constructors and object identity which is independent of any object-related data[2].

2.3 Object orientation

There is no consensus about what precisely object-orientation means, or even how it is defined. Many ideas exist throughout the literature and it is only recently that standards have been proposed [22]. It is generally recognized [115] that there is a clear ascending chain from the relational model through the earlier semantic models.

Kim [88] observes that if one examines existing object-oriented programming languages, knowledge representation languages, databases and data models, a set of common, core concepts can be identified. It is the approach that has been followed in this literature review.

The most general definition about object orientation is given by Khoshafian and Abnous [87]. He defines *object-orientation* as a set of "...software modeling and development (engineering) disciplines that facilitate the construction of complex systems from individual components". He therefore identifies the three main components of object orientation:

Object Orientation = Abstract Data Typing + Inheritance + Object

[1]According to the First Normal Form: "Each value in a tuple is an atomic value; that is, it is not divisible into components within the framework of the relational model. Hence composite and multi-valued attributes are not allowed." [63]

[2]It must be noted that within the ER approach a record identity independent of the record values is feasible, although it is hardly encountered.

Identity

In summary, the fundamental concept that object orientation is based upon is that of the *object*: it is a unique real world entity and the designer is attempting to model its static structure (also known as attributes) as well as its dynamic component (also know as behaviour), through abstractions (also called classes). Classes may be organized in a hierarchy, so that through the inheritance mechanism they can share their members.

The object oriented approach was introduced and used for many reasons. Egenhofer and Frank [59], in their literature search, suggest the major ones:

- Object oriented data models have been developed to capture more semantics than the relational model.
- Object oriented user interfaces make systems appear more natural and easier to use.
- Object oriented database management systems have been investigated to provide the corresponding features for storage and retrieval of complex objects.
- Object oriented software engineering techniques and programming languages have been developed to support the implementation of software systems that were designed following an object oriented approach. They allow for immediate implementations of object concepts rather than simulating them with traditional programming languages.

The functional benefits from the adoption of object-oriented methods are many. Object-orientation, in general, provides ([87, 88, 21, 22, 6]) the advantages listed below:

- Modeling of the real world close to the user's perspective. The result is that the normal radical transformation from system requirements, defined in user's terms, to system specification, defined in computer terms is greatly

reduced. This benefit is commonly known as removal of the so-called "semantic gap".

- Interaction with the computational environment using familiar metaphors: the notion of object is originating from real world situations, hence it will be easier for the user to model complex structures, without worrying about the internal mapping to the data structuring in the computing environment.
- Reusable software components: Software that is built in an object oriented fashion can be reused "as is" within different applications. Code alteration, whenever it is necessary is definitely minimized, and may be limited only within the object.
- Extensible software libraries: new modules are built on top of old ones, without the need for modifications within the old modules. This is feasible either by behavioural extension or through inheritance.
- Ease in the modification and the implementation of the extension of software modules: through the notion of encapsulation, modifications in the software code of a module are independent of any changes occurring in other modules. What should remain unchanged is the interface.
- Rapid prototyping.
- Incremental refinement of the system behaviour. The four stage development iterative process (discussed later in this chapter) allows the model to be defined in different levels of abstraction. As the process moves, the model is defined in more detail. In this way, the developer can focus on what they think is of interest, without worrying about the lower or upper levels of definition.
- The intuitive way of describing systems leads to flexibility when used for customization.
- Support for collaborative decision making across geographically distributed systems.
- Improved application design and implementation practices.

Moreover, the major benefits in the adoption of object-orientation are directly on the system users, either they are developers, programmers or end-users. These can be ([87, 40]):

- benefits to end-users through friendlier user interfaces;
- benefits to application developers through semantically richer models; and
- benefits to system programmers through code reusing, clarity in syntax, easier debugging, improved code maintenance, and time reduction when building complex systems.

Some of the benefits particularly in the context of GIS are ([6]):

- Improved capabilities for data modeling and spatial phenomena simulation.
- Treatment of attribute values as complex objects.
- Modeling of relationships as objects.
- Modeling of abstract concepts.
- Support for versioning, multi-user concurrency, feature-object change tracking.
- Spatial and non-spatial data are modeled using the same conceptual model and stored within the same database space.

Booch [21] endorses that object orientation's main component is the *object model*, the basic artifact which incorporates object-oriented concepts. When this model serves as the basis for a specific paradigm, this paradigm can therefore be characterized as object-oriented. The four major areas that are adopting the object-oriented model, and, as a result, can be characterized object-oriented, are shown in table 2.1:

How are these four categories related? As Booch [21] suggests "...the products of object-oriented analysis serve as the models from which we may start

Table 2.1: Object Orientation in IT

Analysis and Design Techniques	**OOADT**
Programming Languages	**OOPL**
Database Management Systems	**OODBMS**
Graphical User Interfaces	**OOGUI**

an object-oriented design; the products of object-oriented design can then be used as blueprints for completely implementing a system using object-oriented programming methods." In addition to programming languages, the implementation might take place in the environment of a database management system, be it object-oriented or not. As it is explained further in the relevant section, DBMS usually extend the functionality of OO programming languages by adding the concept of data persistence, where objects survive beyond the termination of the program in which they belong, usually in a permanent media space. Moreover, the implementation of the design might be the graphical (or not) user interface, between the computer and those humans exploiting the system. User interfaces are beyond the scope of this chapter and therefore not discussed in detail.

It must be noted that, regardless of the target implementation, the stages of analysis and design are always present and preceding any other phase. Hence, a sharp boundary has to be drawn between the analysis and design techniques (1) and the implementation environments (2 and 3) mentioned earlier: OOADT output refers to a higher conceptual model than the other three and are prerequisites in object-oriented system construction. Moreover, it can be viewed as the link between the real world phenomena, that are being modeled and the computing environment, where the implementation will eventually (but not inevitably!) take place. It must be emphasized that the first stage of analysis and design must not, in any case, be influenced by any disadvantages that a specific OO implementation environment naturally shows. In other words, the designer should not consider the way that the design is about to be implemented. This gives the freedom to the designer to incorporate as many object-oriented con-

cepts as possible, making the design more powerful, flexible and adaptable to complex application domains. However, if the implementation environment does not support directly the theoretical concepts that have been adopted, it is the user's responsibility to devise implementation artifacts, as an interface, so as to enforce the design to "fit" the language or the database. In this situation, the user can be either the designer or the developer of a commercial system. One such paradigm, recently developed, is the so-called "object-relational database management system", where the high level object-oriented design model, with objects, attributes, methods and associations fits into the relations (or tables) of a relational database management system, with records, tuples, attributes and links. This is discussed in section 2.5.2.3.

How does a geographic information system relate to these four areas under examination? The answer lies purely on the architecture of a GIS as a computerized information system. As mentioned earlier in chapter one a GIS includes:

- a database kernel which stores and retrieves the data. It is usually based on a pre-defined data model.
- a programming language, which may be used as an application development interface, a query language or as a simple customization tool;
- a graphical user interface, which may be used to provide simplicity and friendliness in the human-computer interaction process;
- an analysis and design technique, which may be used in the high level stage, to map the real-world phenomena being modeled, into the computer data model, although this is not encountered very often in commercially available GISs. It is a phase usually external to the GIS itself.
- an interface that is used to define the application domain model into a computational conceptual model. Some environments (e.g. Smallworld) provide CASE tools to significantly aid the user towards this direction. Other platforms (e.g. LaserScan Gothic) simply provide a generic model definition GUI.

Since, object-orientation can be applied to any of the above components, hence a GIS can be object-oriented in more than one way. However, characterizing a GI system as object-oriented simply because it involves the usage of an object-oriented language does not necessarily mean it provides full object-oriented capabilities [3].

In this chapter, the focus is mainly upon the first three areas, namely analysis and design techniques, programming languages and their use in combination with database management systems.

As mentioned earlier, the common part that the four different areas share is the object-oriented data model, which is the model that captures the concepts of object orientation and uses them to organize data. Codd [38] defines a data model as a collection of:

- data structure types;
- operators or inferencing rules and
- general integrity constraints.

If the data model is to be object-oriented, then this should reflect the data structure employed in the model itself, since operators, rules and constraints are revolving around the structures. One important question posed at this point is when should a data model be characterized as object-oriented, since object-orientation involves more than a few concepts? According to Brodie [27] the object oriented model is based on the principles of abstraction, classification, generalization and aggregation. Booch [21] agrees on that issue as he suggests that for a model to be characterized as object-oriented, it must comprise of these four major concepts (table 2.2).

[3] a characteristic example is ESRI's Arc/Info (all versions prior to 8.0) which can be customized using the object-oriented language C++. However the database kernel involves the relational model only.

Table 2.2: Major concepts of Object Orientation

Abstraction
Encapsulation
Modularity
Hierarchy

The adjective "major", denotes that without all four concepts mentioned earlier, a model can not be characterized as an object oriented one.

The three minor elements of an object-oriented model that he suggests are:

- Typing
- Concurrency
- Persistence

Minor means that these concepts are useful but not essential for an environment to be object oriented. The last element is encountered mostly within database management systems.

Any environment can be characterized as object-oriented if it incorporates all of the major concepts mentioned earlier. Some other systems might be simply characterized as *object-based*, when they offer less semantics than object-orientation. For example, a computing environment may be characterized as object based when it provides the capability to create new objects out of existing classes without the option to create new classes. Booch [21] suggests that when a programming language does not support the notion of inheritance it is then an object-based language. Such an example is the Avenue macro language which is part of ESRI's ArcView GIS.

As a conclusion, it must be noted that object-oriented design is the most natural of the design approaches at the conceptual level. And when implementation is involved, the full cooperation between the conceptual design and the underlying computing environment becomes the most persuasive argument for the adoption

of an object-oriented database or programming language. The latter two are discussed in detail in sections 2.5.1 and 2.5.2 respectively.

The structure of this chapter is as follows:

The first section is an introduction to the general concepts of object-orientation along with various definitions and views found in the literature. The following three sections are devoted to each of the specific groups of object-oriented environments (OOADT, OOPL, and OODBMS) and the focus is upon the individual and special problems and issues encountered when object-oriented concepts are incorporated within them. It must be noted that there are many OO implementation environments for both languages and databases. Since the purpose of this review is not to focus on the platforms themselves, but on the theoretical concepts of OO, not all platforms are mentioned but only those ones which introduce support for OO concepts. Moreover, special focus is upon platforms that are to be used at the final implementation of the spatio-temporal model in this work, such as Java and LaserScan's Gothic ADE.

Henceforth, the abbreviation OO will point to the term object-orientation as a noun and to the term object-oriented as an adjective. The context of the abbreviation (i.e. the sentence) will clarify whether the abbreviation is used as a noun or as an adjective term.

2.3.1 General concepts of object orientation

It must be noted that the literature on object-orientation is vast, and there is a considerable variation in the naming and the notation of the concepts. More or less, the semantics remain the same, and it is only the naming of the terminology that varies from author to author or among different implementation platforms that support OO concepts. As already mentioned in chapter one, although the OO paradigm spans a period of almost 20 years, it is only recently "official" standards have been released. Such standards are the work of the Object Management Group (OMG) and the Unified Modeling Language (UML), as well as

the CORBA standard (both are explained in further detail later in this chapter). The term "official" is used mostly here in terms of the interoperability of commercial products, since that was the initiative of this kind of work.

The theoretical foundation for relational models and databases is the relational algebra. However, no formal theoretical approach exists so far within the OO paradigm, which remains a research area. A mathematical formalism in OO is the so-called *λ-calculus*. Moreover, a mathematical concept close to object orientation is that of *multi-sorted* or *heterogeneous algebras* [17]. It must be noted that the focus of the current text is more on practical issues rather than any surrounding theory, hence no formal approach is being discussed, since it is considered to be well beyond the scope of the research. The variation that occurs makes a literature search important. It will show how concepts are used within the object-orientation paradigm in general and the specific areas that it can be implemented. Hence, this chapter's aim is to give a survey of the OO terminology and concepts and to built the foundations of the OO reasoning that has been followed in the rest of the thesis. There are many other literature surveys that elaborate in more depth upon OO concept variations and they are mentioned later in this chapter.

Finally, it must be noted that the notation used in figures throughout the next sections is according to the proposal by the UML authors, which is documented in section (section 2.4.2).

2.3.1.1 Abstraction and Abstract data types (ADT)

As mentioned earlier, one of the key elements of object orientation is abstraction. *Abstraction* is a conceptual mechanism via which the human brain copes with complexity. This is achieved through the recognition of similarities between object, situations, or processes in the real world. The goal of abstraction is to simplify a complex system. It is the specific context within which the focusing on specific details is done. Booch [21] suggests that abstraction "...denotes the

essential characteristics of an object that distinguish it from all other kinds of objects and thus provides crisply defined conceptual boundaries, relative to the perspective of the user". He, therefore, points out that "...objects as abstractions of entities in the real world, represent a particularly dense and cohesive clustering of information."

The goal of abstraction within the OO paradigm is firstly to separate individual types of objects so as to produce categories (called the classes) and secondly to distinguish between the object's structure and behaviour in a general level and later from its implementation. Moreover, abstraction is always dependent on the domain and the user's view, as mentioned earlier.

The closest concept to abstraction, which is found in the OO paradigm, is that of data types. [87] uses the notion of an *Abstract Data Type* (ADT) and defines it as "...an encapsulated set of similar objects with the same representation". The specification of an ADT usually includes the structure, that is the static part, as well as the behaviour, or the dynamic part of an object. A clear separation is made between the visible part of an ADT, which is the interface, and its implementation. The implementation of an ADT consists of:

- the representation (data structures); and
- the operations (algorithms) which constitute the interface of an ADT.

As the principle of encapsulation applies in the design (explained later) and functioning of ADT's, the interfaces are kept public, but the implementations remains private, not known or visible to other ADTs (usually called the *clients*). The final implementation of an ADTs is done through classes.

Booch [21] uses the term *type* to denote an ADT. He also uses it interchangeably with the term class (explained later) and abstraction. He therefore defines *typing* as "... the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways". The notion of type makes necessary the distinction

between strong typing and weak typing: in strong typing type conformance is strictly enforced, meaning that operations can not be called upon an object unless the exact signature of that operation is defined in the object class. In weak-typing a client can send any message to any class, even if the class might not know how to respond to that message.

In summary, the benefits of Abstract Data Typing are that:

- it allows better conceptualization and modeling of the real world and enhances representation and reusability;
- it enhances the robustness and the performance of the system;
- the capturing of the semantics is done through grouping of operations and attributes;
- the specification from implementation are kept separate. Modification in implementation is done without affecting the specification of the interface;
- it allows extensibility of the system.

Abstract data types, in terms of their members, need to be complete and correct [87]. Both terms are used literally. Some language constructs are necessary to achieve this goal, in order to help the designer to fully control the behaviour of the abstraction. Two approaches can be found in the literature and both include the usage of constraints:

1. Constraints on objects, which are similar to integrity constraints on tables in relational databases. They usually force the values of the object's attributes to follow the constraint. Constraints can be attached predicates [87] which are usually triggers that get fired up whenever an event associated with the object and the constraint occurs, e.g. on update of a value, on the access of a value etc.
2. Pre- and post-conditions on methods: preconditions apply constraints upon an object's attributes that must be satisfied before a method is executed.

Post-conditions apply on the same attributes but are checked after a method has finished executing.

Genericity [32] has proven to be a powerful method to reduce redundant definitions of ADTs. A generic type or object is a definition which is a backbone for a series of detailed and specified definitions. Through generic types programming is greatly reduced [57] since operations applied to all types are coded only once. The choice between the two approaches is mainly upon the user for *ad hoc* semantic convenience.

The major concept that abstract data types are lacking is that of inheritance, which is supported by the concept of classes, explained in the next section.

2.3.1.2 Classes, Hierarchy and Modules

The concept of the class can be seen as a blueprint or a template according to which objects are instantiated. It is the most fundamental in the object orientation paradigm. A class is the definition of an abstract data type. This can be seen clearly in the environment of an object-oriented programming language, where the class is a language construct to define and implement ADTs. Booch [21] defines the concept of the *class* as a "...set of objects that share a common structure and a common behaviour". A class includes the name, the external operations to manipulate the class, also known as the interface, the internal representation, namely the attributes and the internal implementation of the interface, which are the methods of the class. The set of structure (attributes) and behaviour (operations) within a class, may also be called members of the class. A class may be instantiated to produce a set of objects. Usually, an object belongs to a single class, although there are exceptions, which are mentioned later in this chapter.

Classes are linked together via the sub-class/super-class relationship, also known as the "IS-A" relationship, resulting in a class hierarchy graph, usually called the *class schema*, a special type of "node-and-link" semantic network. Nodes in this

graph are the classes and links are the sub/super-class relationships. When a class A_1 is a sub-class of another class A, then it is said that class A_1 *specializes* class A (figure 2.1).

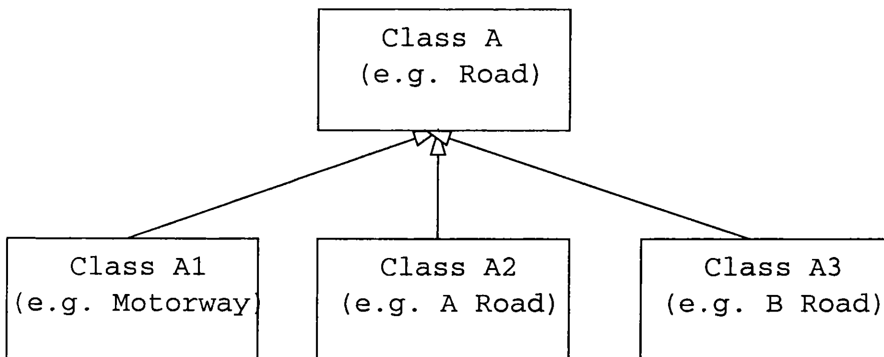


Figure 2.1: Specialization relationships among classes.

Hierarchy can be viewed as a ranking or ordering of the abstractions [21]. It mainly refers to how classes are related to each other, with the child/parent relationship. One of the main reasons that classes are embodied in a hierarchy is for the inheritance mechanism to work: sub-classes will be inheriting members from super-classes and super-classes will be sharing their members with sub-classes. A secondary reason is that some class members are other classes, therefore the aggregation relationships must be shown. Kim [88] separates the notion of the class hierarchy, which captures the generalization/specialization relationships, from that of class-composition hierarchy, which captures the aggregation relationships only, among the classes. Nonetheless, within a single class hierarchy, both types of relationships should be modeled, as well as any other types of associations (explained later in this chapter) for a clearer overview of the schema. All OO environments use a single class hierarchy to model the entirety of class associations. Additionally, for modularization reasons, they use the notion of *modules*, where classes belong and are grouped logically. It is necessary to mention that for the inheritance concept to work, the specialization relationship between two classes must be defined explicitly.

In a more formal definition [88], a class hierarchy can be defined as a "...rooted

directed acyclic graph, such as for a class C and a set of lower classes $\{S_i\}$ connected to C, a class in the set $\{S_i\}$ is a specialization of class C, and conversely the class C is the generalization of the classes in the set $\{S_i\}$. The classes in $\{S_i\}$ are sub-classes of the class C; and the class C is a super-class of the classes in $\{S_i\}$. Sub-classes inherit structure and behaviour from their parent, super-classes." (figure 2.2)

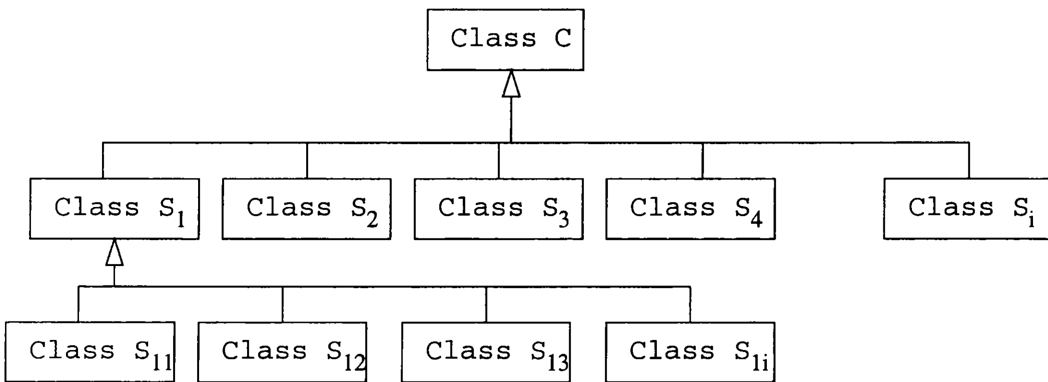


Figure 2.2: Hierarchy of classes

Classes themselves are very often treated conceptually as objects in many object-oriented environments. It is then that they are also called *containers* (as in Java) or *collection objects*, containing not only the class definition, but the instances that are being produced from it as well. The class specification that a class itself belongs to is called a *metaclass*, which is a "higher level" class describing a class that is used to instantiate objects. The notion of metaclass is explained later in this chapter.

A class can be *abstract* if no object is allowed to be instantiated from it. It is usually the sub-classes of an abstract class that, after being enriched in structure and behaviour, can be instantiated. Instantiable classes are called *concrete* or *leaf* classes. It is the designer's choice within the application domain that they work in, to incorporate abstract classes in the class hierarchy. Most OO environments implicitly consider a newly created class as concrete, unless the user explicitly declares it as abstract. The most generalized class of a hierarchy is called the *base* or *root* class. It is very common that there might be more than one base class

in a class hierarchy. However, in some OO environments the unique existence of such a class is necessary.

When a class is created to incorporate program code and attributes that are not part of any other class but are useful throughout the system in a generic way, the class is called a *class utility* ([21]). Class utilities are not instantiated to produce objects, but serve as both the class and the single instance of it. There are further explanations in section 2.3.1.5 about the class behaviour.

Creating classes to model real world objects in large projects, eventually results in a large number of classes that might form one or more class schemata. Therefore, a more elegant manner of grouping schemata is necessary, and that is done through the concepts of modules. *Modularization* according to Liskov [99] consists of dividing a program into *modules* which can be compiled separately, but which have connections with other modules. The main purpose of partitioning a program into individual components is because it can reduce its complexity to some degree. Therefore, the module, can be the basic element upon which a system's physical architecture can be based. A module (or *package*, like in Java) is an artifact used as a container for classes that model similar real world objects. It is very closely connected to the notion of abstraction, since it is the implementation means. Moreover, modules may contain more than one abstraction, providing the means to group them as well. Modules usually contain classes, which are considered more specific and a smaller unit of decomposition. In this way, modularity helps the management of a system's complexity by providing the means to cluster logically related classes. Booch [21] defines modularity as "...the property of a system that has been decomposed into a set of cohesive and loosely coupled modules". He therefore observes that the concepts of encapsulation, abstraction and modularity are synergistic with regard to building individual components, since an object provides a well-defined boundary around a single class and both encapsulation and modularity serve as a shelter around this class. Coad and Yourdon (1991) use the notion of *viewpoints* in the early stages of analysis, to divide the overall model into smaller units and to organize the

phases of analysis and design. The general viewpoint may include several other viewpoints according to the application domains that the model encompasses.

One of the first stages in analysis and design is to identify objects from the domain that is being modeled and to construct classes. Booch [21] proposes five concepts against which the class design should be checked:

- *Coupling*: denotes the measure that modules are interconnected. Strong coupling makes a system more complicated since strongly interconnected modules are difficult to understand and maintain. It is worth noting that inheritance, which connects classes, is quite a barrier to loose coupling.
- *Cohesion*: measures the degree of inner connectivity among the elements of a class. The most desired form of cohesion is *functional*, in which the elements of a class or module work together to provide well-bounded behaviour, so that the connection with other classes is minimized.
- *Sufficiency*: characterizes the degree to which a class captures enough characteristics of an abstraction to allow efficient and functional interaction with the class/object. It usually denotes the minimal interface required.
- *Completeness*: measures the degree to which the interface of a class fully covers and captures the behaviour of the abstraction.
- *Primitiveness*: refers to how operations can be implemented using an already existing set of classes (Booch calls them "...the underlying representation"). It is quite often that many high-level operations can be composed of many low-level ones, eliminating the need to write new operations from scratch. A proper way of forming classes should start with primitive ones, so that later classes can be based on them.

In general, object orientation tends to loosen the coupling among the components of a complex system. Therefore, effects that are due to individual module changes have little or no impact on other modules and their relations with them. However,

due to the flexibility of object orientation, a poor design might end up in strongly connected classes in the system.

2.3.1.3 Inheritance

Inheritance is the concept of code (methods) and structure (attributes) being shared among classes in the same class hierarchy. It resembles to a great extent the notion of inheritance found in the natural world: a living organism inherits its characteristics from its parents. The term can also be encountered with the adjective class (class inheritance), which emphasizes that inheritance refers to classes and their members, in a class hierarchy. Inheritance is based on the so-called "IS-A" relationship between two classes. A super-class represents an abstract (but real) world object, while the sub-classes are more specialized objects, in a specific way. By inheriting the behaviour, the goal of *code sharing* is achieved, and by inheriting representation, the goal of *structure sharing* is achieved. Inheritance works in a top-down wise fashion: sub-classes within a class-hierarchy inherit the structure (attributes) and code (methods) from its super-class (es). In the case that a subclass inherits from more than one super-class, then the inheritance is called *multiple*, a powerful semantic concept which is not widely supported by OO systems (e.g. Java). Cox [40] observes that without inheritance classes would be stand-alone units each of them developed from ground up. He therefore emphasizes the significance of inheritance since it "...makes it possible to define new software in the same way we introduce any concept to a newcomer, by comparing it with something that is already familiar. " Booch [21] views inheritance as a generalization/specialization hierarchy: as the inheritance hierarchy evolves in a class schema, the structure and behaviour that are common for different classes will tend to migrate to common super-classes.

A question posed at this point [88], is that, since a class hierarchy captures the "IS-A" relationship between a class and its super-class, should an instance of a class belong to its super-class as well? An example is given in figure 2.3. The

answer is both yes and no for different reasons: since a class is also its super-class via the "IS-A" relationship (but more specialized), an instance should also be its class' super-class instance as well. This happens only up to a point, since any instance contains an instance of its super-class (es) in terms of structure and behaviour. An exception to this fact is the case when the sub-class has modified the members of its ancestors, either by overriding attributes and/or operations or by restricting members in general. In this case, the instance belonging to the modified sub-class can not be an instance of its super-class precisely, since instance members are different. In both cases mentioned above, another question is when an instance has to be deleted, should it continue to exist as an instance of the super-class of the class it used to belong to? This question is mainly posed in the environment of an OO database management system. If the answer to the earlier question is no, then a deleted instance should not be kept as an instance of its super-class. Nonetheless, the answer to this depends on what exactly is being deleted: if e.g. an object which has a single member that purely belongs to its class (that it is not a member of one of its super-classes) is being deleted, but members that have been inherited then this instance may change its class, or it may not.

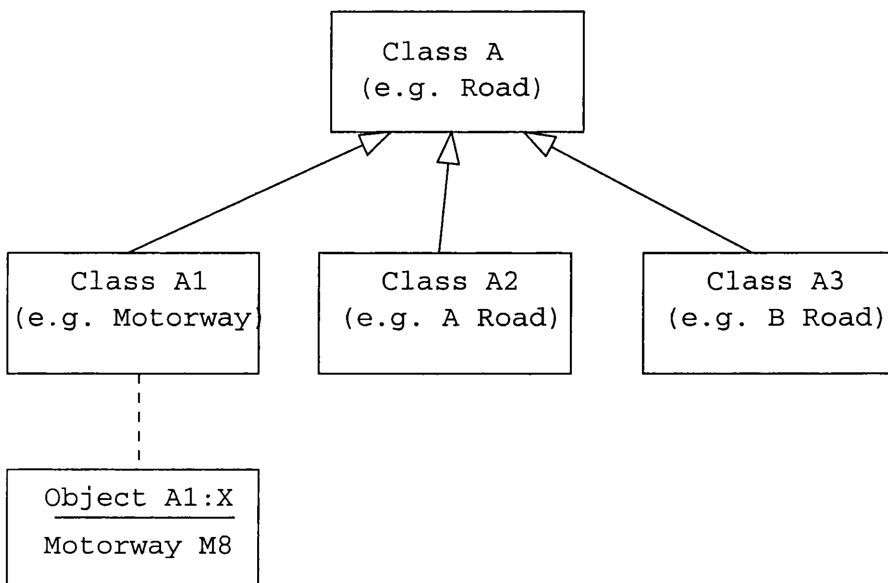


Figure 2.3: Is object A1:X an instance of class A1 only, or both A1 and A?

The three main parts that are inherited from a super-class by its sub-classes are:

- The interface of the super-class (method definitions).
- The code that implements the methods.
- The attributes of the class.

It is worth noting that aggregation associations also follow the inheritance rule, since they are stored in attributes as references to object identities (from one object to another).

Inheritance and specialization are very often confused and misused as terms: through specialization, a set of classes of the same level, become members of a higher set, that of the their common super-class. In this way, every class is the superset of its sub-classes and the topmost (root) class is the superset of all classes, in a class hierarchy. Through inheritance, sub-classes can be enriched in terms of their structure and behaviour, by accumulatively inheriting new members as the class hierarchy graph is traced downwards. In this way, the structure (behaviour) of a sub-class, as a set of attributes (methods), is a superset of the respective one defined in its super-class. Although it may not seem obvious, in terms of inclusivity, inheritance and specialization are working reversely. The sub-class that enriches the structure and behaviour of its super-class (es) is said to use inheritance for extension. Sub-classes might also restrict the structure and behaviour of their super-classes, therefore using inheritance for restriction [21]. It is very common, though, for a sub-class to both restrict and extend its super-class. Extension and restriction should be applied in different members.

The concept of inheritance is quite similar to that of sub-typing, although there are a few differences. Khoshafian and Abnous [87] compare the concepts of inheritance with that of sub-typing.

A major issue that arises is which methods (and/or attributes) should be inherited and which should not, by sub-classes in a class-hierarchy schema. This makes imperative the introduction of the *selective inheritance* concept [88], which

applies for both methods as well as attributes. Selective inheritance is the prohibition of a sub-class from inheriting a specific subset of the structure and/or the behaviour from its super-class (es). This concept not only provides maximum flexibility, but it also gives a solution to the problem of any naming clash that might occur. This is explained in more detail later in this chapter.

Class inheritance, as explained earlier, is a very fundamental concept in OO environments. However, it only regards the sharing of structure and attributes and behaviour, but not the actual values among instances, whose classes have the sub/super-class relationship. Through value inheritance, an instance C can inherit the values of an instance C', only if the class of C' is a super-class of the class of instance C. A dilemma that arises with this concept is whether within the child instance, the attribute values should change, whenever this occurs within the parent instance. This kind of inheritance between instances is called *value inheritance* and it is a very useful feature for OO databases.

With *multiple inheritance*, a class can have more than one super-class from which it inherits structure and behaviour. In this situation, the class hierarchy graph becomes a Directed Acyclic Graph (DAG): It has the properties of being directed, acyclic and connected: Directed means that inheritance between classes works only from top (super-classes) to bottom (sub-classes). Acyclic means that any sub-class can not have any of its super-classes as a sub-class within the same class hierarchy. And connected means that every class (node) on the graph is reachable through the root class (node). Every class has a name, which ought to be unique throughout the class hierarchy schema. Multiple inheritance allows a class to logically belong to more than one class, which otherwise would be too costly to maintain [88].

A common problem encountered with multiple inheritance is that of name clashes: whenever a given set of super-classes (two or more) have the same member name but different interfaces and/or implementations, and a sub-class inherits members from these super-classes, a rule should be specified so that only one member

among members with the same name, from different super-classes, will be inherited. An example is given in figure 2.4. This problem is similar to the one that is encountered in single inheritance in combination with polymorphism (as explained later in this chapter), where a class may inherit two methods with the same name but different signatures and implementations. A rule commonly used is that of *ordered parent classes* [4]. In this technique, the set of super-classes of a class is explicitly ordered in a linear fashion. Members are inherited from the last class in order and move upwards. If the same method or attribute is encountered more than once, then only that of the first super-class in order is inherited, thus prohibiting methods and attributes with the same name being inherited from the next super-classes in order. However convenient this might be, in practice it results in structure or data loss, since some methods or attributes will eventually not be inherited, probably without the designer's intention. Moreover, when inheritance is partial, as when a class inherits all methods, but not all attributes, because of a name clash among attributes, it is possible that references to attributes from within a method will become invalid. Another implication of this approach is that if the specified order is not similar to the class hierarchy, then the intended class design becomes invalid. It is also certain that indirect access to methods belonging to super-classes will be necessary. If this technique is to be used, it is obvious that the user is responsible for avoiding the usage of methods with the same name, throughout the class schema. LaserScan's Gothic ADE uses this technique to order the parent classes so as to avoid name clashes.

Other ways to resolve such conflicts can be [87]:

- By forbidding conflicts whenever there is a name conflict in inheritance, by e.g. issuing an error message. The user is then responsible for renaming any of the members in the super-classes definitely before inheritance is allowed. However flexible this might seem, as it provides the user with the freedom of choosing the qualification strategy, it is also an encumbrance for the

[4] Khoshafian and Abnous [87] uses the term linearization

designer.

- By renaming members. In this case, inherited members with the same name are renamed in the class where they are inherited (as opposed to the previous way).
- By qualifying the inherited conflicting member names, something which is done by adding the class name as a prefix (any of the super-classes that members originate from).

It must be noted that the problem does not arise whenever the source of similar structure or behaviour is a common ancestor of two (or more) classes. Then the inherited attribute(s) and/or method(s) are identical. Thereby, only one of the identical constructs is used. This is a similar problem to that of name clashes, but in a simpler form. It is called *repeated inheritance* [21], where two or more peer sub-classes share a common super-class. In this case (where the class hierarchy will be diamond shaped) the issue is whether the lowest sub-class, which inherits from both peer super-classes should inherit one or two copies of the common members. In some OO languages, this is prohibited before it occurs. In C++, however it is the programmer's decision. Anyhow, it is necessary from the programmer's side to explicitly qualify the inherited copies to differentiate between them. Otherwise, if prevention of inheriting all members is adopted, it will result in data loss.

Three strategies exists to confront the issue of repeated inheritance [21]:

a) by prohibiting its occurrence, therefore treating it as illegal. Smalltalk follows this approach. The user therefore is responsible for member renaming.

b) by permitting repeated inheritance, provided that the inherited method or attribute names are qualified before used. This approach has been taken by C++.

c) by treating multiple references to the same class as denoting the same class. This approach is also found in C++, where the ancestor common super-class is declared within a sub-class as a virtual base class.

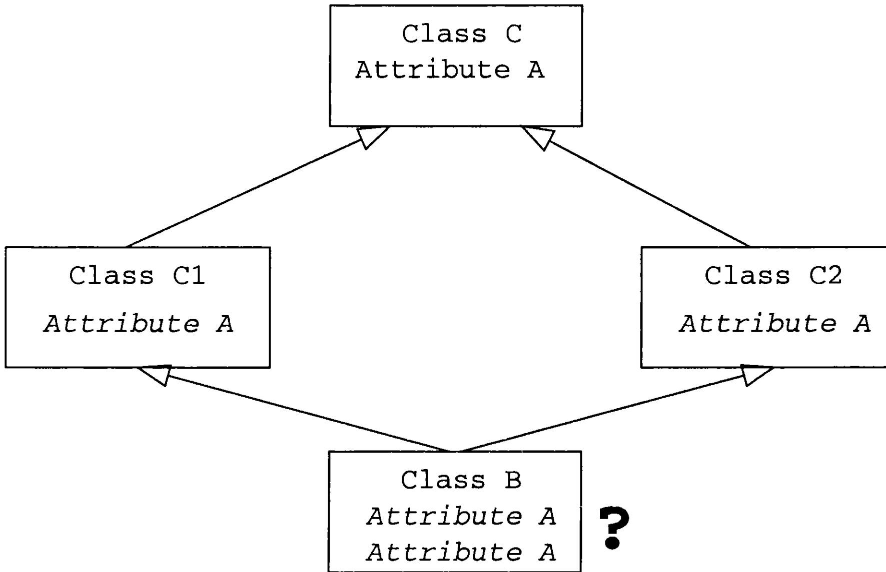


Figure 2.4: Class B will inherit members belonging to class C twice: once from super-class C1 and another from super-class C2. Which of the two should be preserved? (italics denote inherited members.)

Quite often multiple inheritance is overused in the OO design, and this may complicate the schema itself. Booch [21] suggests that it may be avoided, by replacing one (or more) of the hierarchy associations of a class with one (or more) of the aggregation association. He also names classes that inherit only their members without adding new structure or behaviour as aggregate classes. The issue here is that a programming language deficiency has an effect on the way that the model is built, which according to proper OO design should not occur, since high level design is independent of the functionality offered by the implementation environment. However, this is considered to be a convenient artifice in languages like Java, which does not support multiple inheritance. As it is explained in detail in chapter 4 in the design stage of the model, cases where multiple inheritance was necessary are transformed to aggregation relationships between classes.

An extreme view of the combination of inheritance and encapsulation is that these two concepts invalidate each other: since structure and behaviour are hidden

in a class, by allowing inheritance by sub-classes, encapsulation is violated. If encapsulation is enforced, then sub-classes should inherit neither structure nor behaviour. Liskov [99] notes three ways in which encapsulation can be violated, always in coexistence with inheritance. A sub-class might:

1. access an instance variable of its super-class,
2. call a private operation of its super-class, or
3. refer directly to super-classes of its super-classes.

It must be noted though, that in most of the OO environments, encapsulation is forced only on instances of classes, therefore allowing the combination of the two concepts to coexist and cooperate within the class hierarchy. This issue gives rise to the characterization of the members of a class as *private, protected* and *public*, which are explained later in this chapter.

So far, inheritance refers to code (methods) and attribute specification sharing between parent and child classes. When classes are instantiated to produce objects, it may be useful to not only inherit the structure of its super-classes but the attribute values of an object belonging to its direct super-class or super-classes, in the case of single or multiple inheritance respectively. This concept of *value inheritance* is considered to be of use, to model temporal data, especially in OODBMS environments, and in combination with object versioning, although currently it is not explicitly supported by any commercial OO environment. This mechanism may be implicitly implemented with pre-conditions found in constructor methods of a class. Value inheritance has meaning only among objects and not classes. It must be also noted, that value inheritance imposes once more the question of how encapsulation and inheritance can co-exist without violating each other, since it *does* violate the principle of encapsulation. This can be avoided by including methods that access the attribute values, instead of directly accessing the attribute themselves. In this way, value inheritance may conform more easily with any interoperability mechanisms (discussed later in this chapter). Anyhow,

value inheritance should be used with caution and only whenever it is absolutely necessary.

2.3.1.4 Structure of classes - Attributes

Classes comprise structural specifications that describe what objects look like. These are called the *attributes* of the class. When the class is instantiated and an object is created, attributes contain values that conform to the specifications of their attributes. Every attribute has an associated domain, which is actually the class, where the values of the attribute belong. Values can be other objects or primitive classes, such as integers, real numbers, strings etc. These classes are often called *base objects.* It must be emphasized that an attribute's domain is not specific, as is the case with the relational model. Since an attribute can receive a class as its domain, it can also receive all its sub-classes as well, expanding in this way the range of values it can receive (see figure 2.5). This is considered to be an extremely powerful mechanism that provides a significant degree of flexibility both to the design and the implementation.

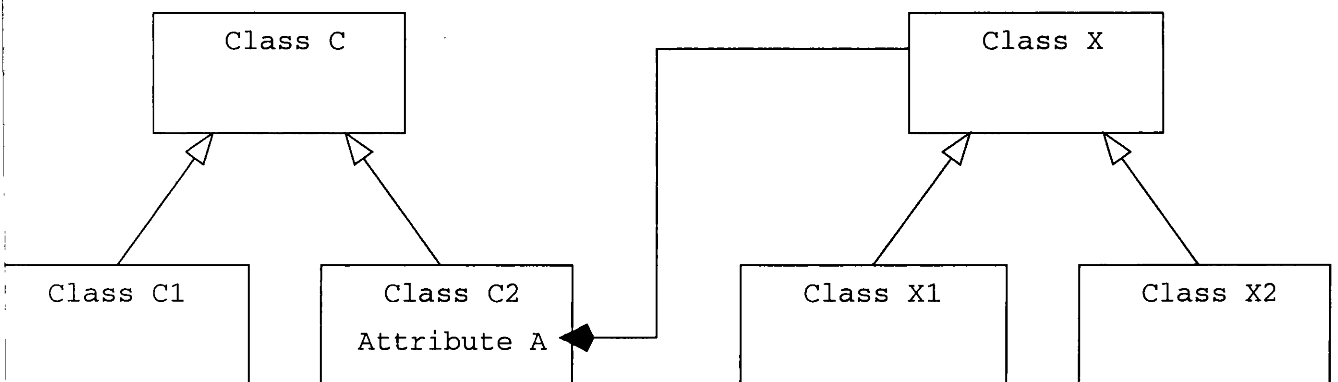


Figure 2.5: Class C2 contains the attribute A, which is a reference to a class X. In class C2, attribute A can receive as values not only objects belonging to class X, but objects belonging to X sub-classes, namely X1 and X2

Kim [88] distinguishes between two types of attributes depending on the way that their values are set: *shared* are those attributes whose values are set by the

user explicitly in every instance of the class. *Default*, are those attributes whose values are set implicitly when the class is instantiated (usually by the constructor method of the class, explained further below in this chapter).

As mentioned earlier, associations are finally implemented as attributes within a class, and are called *links*. They contain the reference to the object with which they are related (a sub/super class, aggregate part or other). In other words, links are object attributes that usually contain the identity of the referenced object. The *identity* (ID) of an object, is a special attribute that distinguishes it from other objects and makes it unique within a given object space. In contrast to the relational databases[5], object identity is created with total disregard to the object's attribute values. Hence, whenever a value of an attribute in an object changes, its identity remains the same, unaffected.

A single identity is associated with every object in one-to-one relationship, so that no identity can exist that does not point to an object, and *vice versa*. The identity should be created simultaneously with the object.

Additionally, through identity, objects can reference other objects, establishing the basis on which complex (or aggregate) objects may be built. Moreover, objects can be stored persistently in a database system (as explained later in this chapter) and therefore be referenced, distinguished and retrieved, only by their identity.

Effective utilization of an object's identity (OID) poses a question as to how it should be formed. In a truly object-oriented environment, object identity should be constructed based upon the following principles [87]:

- Uniqueness throughout the object space (transient or persistent).
- Independence of any particular computer implementation.
- Independence of where the object is stored (the physical address).

[5]In relational databases the uniqueness of each record is characterized by the primary key, which is usually dependent of other attributes within the same record, and results in a unique combination of values in each record. If attribute values change then the primary key value changes as well.

- Independence of its attribute contents (object state) or even the way it is accessed.
- Uniformity of object identity type across the same object space.

Any technique used to create object identities should follow and preserve the above principles, in order to be used effectively.

Many OO environments, though, relate the OID to the physical address of an object, whether the object is stored temporarily (within the execution of a program, in RAM) or permanently (within a database system, in permanent storage media). The main dichotomy here is: should the OID be the same as the physical object address or not? Should there be any relation between these two? According to the principles of constructing an OID posed earlier, this should not be the case, but why?

The answer lies in the *address* of an object, which is a lower-level, usually environment-dependent mechanism, used to reference an object within a particular environment. If the object moves physically, within the same environment, or the environment of the object changes (e.g. the object is exported out of the system) the address will change as well. Therefore, relation to the physical address compromises the notion of identity.

Other techniques of forming object identities, commonly found in OO environments, include user-specified names or the usage of identifier keys in object collections. An example of a user-specified name is the hierarchical directory structure in most operating systems. Identifier keys form the approach mostly encountered within relational database systems, as mentioned earlier. The major disadvantages of this technique, that violate the principles by which OID should be constructed are that [87]:

- the modification of the identifier keys that participate, results in modification of identity.

- the combination of different type fields (like string, integer and Boolean), results into non-uniform identities across different objects.

In some OO environments (as in Smalltalk), object identity is based upon the attribute values of the object, but only for basic, built-in objects (like INTEGER). In the context of an OOPL in general, variables are referenced through pointers, a concept first introduced in ALGOL 68. Pointers are actually physical addresses, which correspond to memory locations. Moreover, pointers are part of a technique very often used to implement the notion of identity.

In the case of OO environments, where all variables are treated as objects, and as every object has a unique identity complex, objects will therefore be treated as collections of references to other objects. In this way the state of an object can formally be expressed as the set of tuples which consist of the variable name and the actual value, that is the referenced object identity.

The life cycle of an object depends on the environment in which it is manipulated. In OO languages the object life cycle is the duration of the program execution. In OO databases the users of the database define the object life cycle. In any case, the identity of the object should not change. Some programming languages support the notion of object persistence, that is storage of the object past the program execution. Most of them, however, do not allow the persistence of objects if they contain pointers to other objects, since identity is usually implemented as the physical address of the object, and upon program termination it ceases to exist along with the objects. An exception to the above is the PS-ALGOL language, an extension of ALGOL, which supports persistence of the dynamically allocatable memory. It must be noted that in the case of persistent complex objects, all aggregate objects must be stored as well.

Every object in the transient or persistent object space is stored, and therefore a memory address points to the memory location where the object is stored. Types of object spaces (in term of physical storage media) may be:

- the virtual memory (for transient object spaces);

- a secondary storage address. e.g. the hard disk (for persistent spaces); or
- a structured name (in the context of a distributed environment).

The memory address is a direct way to refer to the position where an object is stored within an object space. Indirect ways to memory locations use other object-related information to access memory addresses. It is the most convenient way to differentiate between object identity and address, and keep these two correlated. However, to adopt this strategy, the processing overhead required should be taken into account: objects keep unchanged identities that point to memory addresses, which in turn may be changed, if objects are to be moved (as in garbage collection). Given the object ID, this address is always known. Therefore, there should always be a function that retrieves the object address through its identity, and *vice versa*. Indirect ways are also called *indexes* (explained further below in this chapter). Implementation strategies can be [87]:

- Object-tables: these are tables with tuples, each of which is a combination of the object identifier, along with the starting object address within the object space.
- Address schemes: this is the simplest way to implement the identity of persistent objects through its physical address within an object space. As opposed to the previous approach, no differentiation is done between identity and address. When used by itself, it imposes all the problems related to treating identity and address as identical constructs (expanded earlier in the identity section of this chapter).

The two different approaches, mentioned above, can be used in conjunction, although this will result in a dual representation [87]. The first could be used in the case of disk-resident objects (disk object space), while the latter when the object is retrieved and read into the read-only memory of the system (RAM object space). However the overhead of transforming memory addresses from the one object space to the other is required, in the case of complex objects that have references to other objects, the later being their aggregate parts.

The most powerful technique so far, to implement the concept of object identity is through *surrogates* (Bancilhon *et al.*, 1987). Surrogates are values that are generated by the system and have the properties of being:

- Globally unique (even in distributed object spaces).
- Independent of the object state.
- Independent of the object location in memory (transient or persistent object space).

An important issue arising here is how should a surrogate be formed within a distributed environment and yet preserve all the properties prescribed earlier that apply to the notion of object identity. Leach *et al.* [98] propose a way to implement surrogates in distributed environments.

Another important issue is when an object's attributes change, should its identity change as well? Arctur [6] suggests that this is an application related answer where some rules need to be predefined. This is also related to the mechanism embedded for object versioning. Should a new version of the whole object be created, therefore requiring a new identity, or should the attributes be versioned themselves, therefore having the same object throughout the version life cycle?

Since the OID differentiates itself from the state of an object (namely any combination of the actual values of its attributes), it is imperative to discuss the operations that can be used upon the object identity. Khoshafian and Abnous [87] identifies three types of identity operations: equality, copy, and merge/swap. Differences among objects of the same class are based mainly upon their identity and secondly on their state (the values of their properties). Therefore, equality of objects is a concept that must be examined in terms of their identity, their contents as well as the level of the object structure. More specifically:

a) Two objects can be considered *equal*, either on the basis of their identity or in terms of their attribute values. The two cases are quite different and impose semantic and symbolic differentiations on the operators that should be used to

denote each type of equality. Many OO environments use the "=" operator to denote value equality and the "= ="operator to denote identity equality. Identity equality occurs whenever two objects can be used interchangeably within an operation, without any effect in the operation result. It is obvious that whenever the identity equality condition is true, the equality in the values that the object holds should be true as well, since otherwise the principle of identity uniqueness is violated. In other words, two objects with the same identity must hold the same attribute values. The term equal is therefore used to denote complete object equality.

b) In the case that two objects hold exactly the same values in their attributes, belong to the same class, but have a different OID, it is the case of *equivalent objects*. Khoshafian and Abnous [87] uses the term of *shallow equality* for this type of equality.

c) When two objects belong to the same class, but have different OIDs and share partially identical values in their attributes, it is the case if *partially equal objects*. Khoshafian and Abnous [87] uses the notion of *deep equality*. If the equality is upon the isomorphism of the graph structure of the objects (which obviously belong to different classes) it is the case of *isomorphic partial equality*.

All the cases of object equality mentioned earlier have the properties of being reflexive, symmetric, and transitive.

As a conclusion, it is considered to be a purely conceptual problem whether and when an object should become another object (e.g. change its identity) if it undergoes a transformation. It is suggested that this should be specified in the early stage of the analysis and later in the design documentation, namely through the notation language used.

2.3.1.5 Structure of classes - Behaviour of objects (Interface, Methods/ Operations, Messages)

Apart from the static, structural part, objects are also dynamic: they respond to messages that other objects, called the *clients*, send to them, according to the operation definitions found in their class. It is also called the behavioural part of the object. Usually, for a given class, there are two kinds of clients: classes that invoke operations defined within the class; or, sub-classes that inherit from that class.

Behavioural specifications describe:

a) what requests are applicable to objects, as well as b) how the processing of these requests is executed inside the object.

Objects can process these requests from other objects, known as *messages*: they either ask the target object to perform a computation and return a value or modify the object's content resulting in a change of its state, namely the values of its attributes. Booch [21] names the interaction among the objects as *mechanisms.* He also considers objects as small independent machines, which are parts of a larger software system. Therefore objects may be classified as either *active*, when they encompass their own thread of control and as *passive* when they do not.

Each object has a communication protocol, which is the set of messages to which the object can respond, returning a non-error status. This, in turn, is defined by the class to which the object belongs. The protocol of an object defines its interface, which is a collection of methods defined for the instances of its class. The actual implementation of an interface is done through *method* or *operation definition.* Operations denote the services that a class may offer to its clients. The specification of a) the arguments (input) and b) the returned (output) types of a method are the *signature* of the method. In typeless OO environments the argument number is sufficient to define the signature. Booch [21] endorses that as the protocol of an object is the implementation of its behaviour it denotes the role that an object can play, mainly between its class and the clients that access

the class.

Objects are individual components but also related to each other. As Parnas [*] states that "the connections between modules (objects) are the assumptions which the modules make about each other". This statement makes clear and obvious the necessity for *a priori* interface definition of what an object exposes to other objects. When this is standard, communication between objects is feasible. This is a prerequisite stage prior to any object-oriented design and implementation, so that, objects which have been instantiated from classes of the same module, can communicate among them. When this interface is not standard, objects must have an inherent mechanism to declare their interface functionality to other objects, upon any request. The CORBA standard has been developed specifically for this purpose, so that objects coming from different designs and applications can communicate, without the interface having to be in a standard form. The CORBA standard is explained in detail in the relevant appendix.

A method definition comprises the name, the number and the type (or class) of the arguments. The actual code that specifies what the method does is the *implementation of the method.* In most programming environments, the definition and the implementation of a method are treated separately, since the principles of encapsulation and object-to-object message communication through public interfaces are preserved. Invoking a method involves a target object, the name of the method, and the arguments of the operator. Methods are inherited from parent classes similar to attributes, through the principle of inheritance, in a class hierarchy.

Khoshafian and Abnous [87] distinguishes three categories of methods:

- Accessor methods, which can simply retrieve values of an object's attributes;
- Update methods, which modify the value of one or more attributes of the object they belong to; and
- General methods which perform complex computations probably involving other objects and methods as well. According to Booch [21] a special type

of a general method is the *iterator*, which includes operations that permit all parts of an object to be accessed in some well-defined order.

Accessor methods exist so as to preserve the principle of encapsulation: the attribute values are accessed only through an invocation of a method, namely a message to an object. They are very common, so that many programming environments generate them automatically (usually with the qualification prefix "getXXX", where XXX is the name of the attribute which the method is retrieving the value). Update methods provide considerable flexibility when it comes to actually implement methods without modification of the interface, thus enforcing encapsulation and message communication between objects. Another two important types of methods encountered almost in every object-oriented environment are the *constructor* and *destructor* methods: they are associated with the creation and the destruction of an object respectively within the object life cycle. Constructor methods usually contain instructions relative to the initialization of the object and its attribute values, while destructor methods deal with the allocation of the space that the object was occupying while it exists in an object space. Both methods are usually invoked automatically, whenever there is a client request to create or destroy, respectively, an object. For a new object to be instantiated, a constructor method is usually invoked via a message to the object class, (since classes are treated as objects in most OO environments, as mentioned earlier). The strategy of free space allocation upon object destruction is called *garbage collection.* The computing environment is mainly responsible for this procedure, i.e. the OO language or DBMS. There are many proposals for garbage-collection techniques, which deal with the problem that arises when an object is deleted: the references that other objects have as attributes must be deleted or updated. Khoshafian and Abnous [87] group these proposals into reference counting, mark and sweep and scavenging algorithms. Garbage collection is well beyond the thesis' scope. When an OO environment does not provide implicitly constructor or destructor methods, the user is responsible for the explicit definition and implementation of such methods, whenever necessary. It is very

often encountered within database systems, where objects are stored persistently.

Egenhofer and Frank [59] use the concept of *propagation* to denote that a property of a value in a class is derived from a property value found in another class. Usually, this value originates from a component object (or a collection of them), and is inserted into a property of the composite object. The reverse case is equally possible. Propagation, as a mechanism which works among objects belonging in a "IS-PART-OF" relationship is different from inheritance, which works among objects belonging in the "IS-A" relationship. Formal definitions of propagation have been given in terms of first-order predicate calculus [54]. Propagation usually involves the calculation of a value in the aggregate object out of a set of property values that belong to its components. The functions used for this calculation are called *aggregate functions*, and their purpose is to reduce the aggregate object's details found in the aggregate parts. Examples of aggregate functions are maximum, minimum, average, sum, weighted average etc. The main advantage of propagation is that it guarantees data consistency along with reduction in data redundancy, across a dataset, since data are only stored once, and therefore any values depending on a data subset are derived. It must be noted that elementary values (the ones found in the aggregate part) may only be changed explicitly, while the aggregate value can not be altered except through the aggregate function. Two main properties of propagation are identifiable: 1) the calculation of a value through an aggregate function might involve values found in different classes, and 2) the propagation mechanism is transitive since the derived values may be used to calculate further aggregate values. However convenient this mechanism might be, it introduces performance problems in query execution, since checks should be applied beforehand, in order to guarantee data consistency.

The choice of the actual code to be executed for a given message to an object depends on the object's class. The determination is usually done at run time, and this technique is known as *dynamic binding* (also known as *late binding*), which is very common in object-oriented languages, unlike with procedural pro-

gramming. It means that the system binds message selectors to the methods that implement them at run time, instead of at compile time. If the environment does not support dynamic binding then a large case statement would be necessary [87] so as to differentiate among the various implementations of a single method name. Whenever an object receives a message to execute a method, then a search is necessary prior to execution so as to determine which class holds the method that matches the given message. The usual algorithm involves searching, firstly, the receiver object class. If no appropriate method is found, then the searching continues upwards along the class hierarchy, until the method is found, and on upwards to the root class, where it has to terminate. An error should be issued if no method is found at all.

Similar to the equality operation possibly imposed on two objects, the copy operation can also be implemented as shallow and deep. In the first case, an object is created with all values but the identity, equal to an existing object. In the latter case, the new object that will be created will be deeply equal to an existing one. Deep copying also denotes the copying of the structure of the object along with a partial copy of the values in its attributes.

Finally, two objects can be *merged* if they are instances of the same class and are deeply equal. The result is that one of the objects ceases to exist, and all references from other objects point now to the one object. Similar to this operation is also swapping identities between two objects, as long as they are, once again, instances of the same class and deeply equal.

In pure object-oriented environments, operations should always be implemented as methods that belong to a specific class as a member, hence no stand alone functions or procedures can exist. In some OO languages, however, (like C++) non-member functions can exist, usually called *free subprograms.* Booch [21] defines them as "...procedures or functions that serve as non primitive operations upon an object or objects of the same or different classes". The latter are also called *class utilities* and they serve as the basis on which free subprograms are

grouped. They are simply declared as members of a class with no state (no attributes). C++ has adopted this approach. Booch [21] comments on class utilities and based upon this concept, differentiates the concepts of methods and operations: all methods are operations but not all operations are methods, since some operations can be expressed as free-subprograms.

In order that two objects can communicate and send messages, they ought to be visible to each other in some way. This is feasible in the four following ways [21]:

- The server object is global to the client.
- The server object is a parameter of some operation of the client.
- The server object is a part of the client object.
- The server object is a locally declared object in some operation of the client.

Not all objects are visible to each other. This is a policy for which the user is responsible during designing and implementation.

Synchronization between two objects is achieved whenever one object passes a message to another across a defined link [21]. In the presence of many simultaneous messages among the objects, special mechanisms are necessary for the sequence of message passing to be guaranteed. Since active objects, as mentioned earlier, embody their own thread of control, their semantics will be guaranteed in the presence of other active objects. In the case of message passing across a link between an active and a passive object, three approaches exist to guarantee synchronization [21]:

- Sequential, when the semantics of the passive object are guaranteed only in the presence of a single active object at a time.
- Guarded, when the semantics of the passive object are guaranteed in the presence of multiple threads of control, but the active clients must collaborate to achieve mutual exclusion.

- Synchronous, when the semantics of the passive object are guaranteed in the presence of multiple threads of control, and the supplier guarantees mutual exclusion.

2.3.1.6 Associations

An *association* is the way that two or more classes are related in a class hierarchy. The following four major types of associations among classes can be found in the literature:

1. *Generalization*, when a class represents a more abstract real world object than the set of its sub-classes.
2. *Specialization*, when classes represent a more specific real world object than their super-class. It is the opposite of generalization, and is sometimes used interchangeably, in terms of the association. It is only the direction that changes. Booch [21] names this type of association as inheritance, and it is of the same meaning as the one described earlier in this chapter (section 2.3.1.3).
3. *Extension* [87] or "instance-of" relationship [88], when a set of objects is instantiated from a class. The association is between the class and each one of the objects that can be instantiated from it. Objects that are instances of the same class are indirectly related through their common class. Booch [21] names this relationship as *instantiation*.
4. *Metaclass* when a class is used to model a class. This concept implies that classes are treated as objects.
5. *Using* [21] denoting that an object uses the available resources of another object, along *peer-to-peer* links (links between objects that belong to the same class, or between objects that belong to classes that are at the same level in the class schema). The supplier object is called the *server*, and the requester object is called the *client*. This kind of association is confining only

when the public interface of a server is available to be accessed by clients. In C++ the concept of *friend class* is employed, in order to allow access to private and/or protected members to otherwise unauthorized client objects.

6. *Aggregation*, whenever one class is made of a number (at least one or more) of other classes. The instances of these classes form complex (or aggregate) objects. [88] uses the term *composite reference* and distinguishes between two types: *exclusive*, where a class is a part only of another class and shared, where a class may be a part of more than one class. In the case of dependent references, the existence of the aggregate instances depend on the existence of the object that they are part of. This view leads to four types of aggregate (or composite) relationships:

   - Exclusive dependent composite references
   - Exclusive independent composite references
   - Shared dependent composite references
   - Shared independent composite references

Aggregation usually denotes the physical containment of an object within another. It is then that the aggregate object is called a *container*. When two classes are associated via an aggregation relationship, the objects that are instantiated will come in pairs: one of them will belong to the other one and sometimes the existence of the part object will depend on the existence of the aggregate object. This type of dependent aggregation is named *by-value* aggregation. In contrast with this, when the part's existence does not depend on the container object, then the aggregation type is *by-reference* [21]. In any case, it is definite that aggregation is not cyclic, since both objects cannot be parts of one another at the same time. This type of containment relationship, which is modeled via the "PART-OF" relationship, is transitive: if an object named $\{a\}$ found at level $\{i\}$ on an aggregate hierarchy contains an object $\{b\}$, then its parent aggregate object, named $\{c\}$ which is at level $\{(i-1)\}$ also contains object $\{b\}$.

In general, the above types of associations can be defined explicitly in any OO environment. It is not necessary, though, for a compiler to actually know if a relationship is any specific kind, with a few exceptions. Differentiation among relationships is purely for user convenience. In this manner, any visual modeling becomes clearer and more comprehensible.

The "IS-A" relationship in a GIS context, is often used to model non-spatial relationships among classes (spatial or not). Choi and Luk [35] use the term "non-spatially-associated" to characterize these object classes. The "PART-OF" relationship is usually used to model the spatial containment relationship, which may or may not include spatial overlapping among the objects. This approach can be found in [35].

Classes are related through associations. When objects are instantiated, they are related through *links*, which is the implementation of the concept of association. Associations are implemented by placing attributes in the classes that reference other classes, through their name, be it specialization/generalization, relationships, aggregate parts or other associations. Links between objects are implementable through placing values in the attributes of their structure that contain the identifiers of the linked objects.

If class associations are defined explicitly by the user (e.g. aggregation), most of the types mentioned earlier are implemented by the user as well in the same fashion, namely as attributes within the object structure. The differentiation between two types of associations is done by the naming convention that the user follows. It is usually the implicit associations that are implemented by the system in an internal way, usually hidden from the user.

Every association is between two classes. Each class plays a *role* in this association. Roles are not necessary concepts for the OO model to work. They only help the reader to understand the relationship between two classes. Regarding the built-in behaviour of a class, objects that are instantiated and related through links, can play one of the three following roles [21]:

- Actor object, when it can operate upon other objects but never be operated upon by other objects. The term active object can be used in the same way.
- Server object, when it never operates upon other objects but it is only operated upon by other objects. It is the opposite role of the actor object.
- Agent object, when it can play both roles mentioned above. It is usually used as a mediator between an actor and a server object.

The identification of associations among classes is done in the stage of analysis and early design.

Associations between two classes are implemented as links between two objects that are instances of these classes. However in the case of instances, more than two objects can actually be related, even if their classes are related through a single association. The concept of *cardinality*, denotes the number of objects belonging to a class B that can be linked to a single object belonging to a class A and *vice versa*. The two classes are related through a single association. Cardinality can be of any type, always declared explicitly by the user. Three common kinds of cardinality are mostly used:

- *One-to-one*, when a single instance of class A is linked to a single instance of class B. The instance of class B is linked also to the same instance of class A. This is a rather narrow type of association.
- *One-to-many*, when a single instance of class A is linked to a number of instances of class B. The set of instances of class B are all linked to the same instance of class A.
- *Many-to-many*, when an instance of class A is linked to a number of instances of class B. An instance of class B might also be linked to a number of instances of class A, but not necessarily to the same ones.

2.3.1.7 Encapsulation

Encapsulation is the principle of keeping the internal structure and behaviour of an object hidden from being viewed by unauthorized clients. It is one of the fundamental concepts that object orientation is based upon, and it is feasible through the mechanism of information hiding.

Booch [21] defines encapsulation as "...the process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction and its implementation". He considers that abstraction and encapsulation are complementary concepts since abstraction focuses upon the observable behaviour of an object, whereas encapsulation focuses upon the secret implementation of this behaviour.

Encapsulation enables the so-called "plug-and-play" software, which is based on the client-server architecture. A software server object consists of two distinct parts: its interface, always presentable to the outside world, namely any client, and its implementation, which is kept private and only known to the object itself. A client sends a message to the server object requesting a service, with the proper arguments and in a predefined format: the response message is sent back, and the exact way it was calculated is known only to the server object itself. If more than one implementations of the same object exist in the same database, written in two different languages (e.g. Java and C++) they could easily substitute each other without the client knowing anything neither care about this, since messages and responses remain the same.

Although encapsulation is one of the fundamental OO concepts, it poses many obstacles: for instance, a major issue is how to preserve the concept of encapsulation in the environment of an OO database where attributes must be indexed, which is expanded more in the section for object-oriented databases. Encapsulation can also be a barrier to quality management issues in software, since the way that algorithms are implemented is hidden, therefore preventing the examination

of the code itself.

2.3.1.8 Polymorphism

Webster's dictionary (1998, http://www.m-w.com) defines *polymorphism* as "...the quality or state of being able to assume different forms". The application of polymorphism onto programming languages indicates that the same language construct can be used to manipulate different types of objects. One form of polymorphism is the *method* or *operator overloading* or *overriding.* In method overloading, the same method name can be used more than once but with different semantics and implementation, thus involving different code. Overloading can be applied to operators as well, resulting in *operator overloading*, meaning that the same operator can accept different types of operands. In *parametric polymorphism*, types are used as parameters in generic type declarations or classes. OO languages that support parametric polymorphism allow the use of parameterized classes in both built-in and user-defined programming constructs. Some OO environments (like C++) use the term "virtual" to denote a method defined in a class but which is being overridden in a sub-class that inherits from this class. Distinction among overridden methods of a class is done on the basis of their signature, which has to remain unique. When a method in a sub-class is overridden, it usually invokes a method found in the super-class, with the addition of some other behaviour. Booch [21] claims that in this way polymorphism in subclass methods plays the role of augmenting the behaviour defined in super-classes. Alternatively, the overridden method will include code totally different than the one found in its super-class.

There is a strong connection between dynamic binding and overloading. In method overloading, where the same method name can exist within the same class more than once with different implementations, the actual code that will be executed is decided after the message request to the object, provided that the full message specifications have been defined (method name, number of and type

arguments). This late decision is imperative during run-time rather than compile time, and that is how dynamic binding and overloading come in pairs. The same strategy applies in the case of operator overloading. It must be noted that the late binding technique is not only obligatory but also best utilized by typeless languages.

Operator overloading although is considered to be a powerful feature, however, it poses user comprehensibility issues, since the programmer has to be aware of the definition of the operator, whenever this has been altered.

The advantages that dynamic binding in conjunction with overloading offers are [87]:

- Extensibility, since the same method name or operator may apply to instances of many classes without code modification.
- Compact code development: elimination of case statements to decide upon which code to execute.
- Clarity: the generated code is more readable and comprehensible by the programmer.

The main disadvantage of dynamic binding and overloaded operations is performance cost increase, since run-time binding and/or type checking is necessary for correctness reasons. Acceleration can be achieved through hash tables and indexes to decrease the run-time overhead required.

Overloading offers great flexibility in the use of structure and behaviour naming. A categorization of how overloading can be used is given by Khoshafian and Abnous [87]:

For attributes:

- No redefinition: In this case, the overloading of attributes is prohibited. This is a conservative strategy since it disallows the power of polymorphism, although it eliminates the problems arising from the implementation of polymorphism.

- Arbitrary redefinition: this is the opposite of the first approach since no constraints are applied to how attributes are overridden. It is a characteristic of typeless OO languages.
- Constrained redefinition: this occurs when the redefinition of attributes is done only by sub-typing, that is the redefined attribute is a subtype of the parent attribute (or a sub-class when the attribute is a reference to a class). This is an imperative strategy within strong typed OO languages to ensure correctness of arguments.
- Hidden redefinition: in this case, the definition of the attribute is hidden from the child classes that could (but do not) inherit it. It is similar to the concept of selective inheritance. It actually enforces the concept of encapsulation even between parent and child classes in a class hierarchy.

For methods:

- Arbitrary redefinition: no restrictions are applied to how the overloading is done.
- Constrained redefinition: in this case, the arguments of the overridden method must be subtypes (or sub-classes) of the method defined in the parent class. This brings up the concept of signature conformity (also known as covariant rule) between the parent and the child definition of the method [87]. Note that conformity also applies to the post and pre-conditions that have been applied within the implementation of the method. This kind of redefinition is likely to be found in strong-typed OO languages, since it guarantees the correctness. The run-time however is required in this case.
- Explicit/implicit exclusion of method: this is similar to the hidden redefinition of inherited attributes as mentioned earlier. This might be useful whenever the method references attributes or performs operations on the class that are not inherited or not wanted, respectively. In this case the method is totally hidden from the sub-class. Run-time errors might occur though, within OO environments that support dynamic binding: an

excluded method can still be accidentally referenced to an instance that originates from a class that does not support it, although it is syntactically correct. Therefore exclusion of methods should be used with great care in combination with dynamic binding.

The various ways allowing class members to be or not to be redefined from within the child class, gives rise to the concepts of *public*, *private* and *protected* members of a class. Public members are accessible from within any client. Private members are accessible by no client. Protected members are only accessible from within child clients, namely objects belonging to sub-classes of the original class. Protected members also implicitly declare that they are private to clients that do not belong to the specialization hierarchy and therefore not accessible by them. Both the C++ as well as the Java programming languages support explicit declarations of all three types of class members mentioned above.

It is often necessary from within a class, to invoke an overridden method found in one of the super-classes. Both the sub-class and the super-classes share the same name of the method, but the actual code of the two methods might be totally different. The invocation of the overridden method is feasible through qualifying it, usually with the super-class name as a prefix. Another, special, qualification way is to use a keyword that denotes the immediate super-class of the class that the method is defined in, like "super". It is obvious that a keyword like this can only be used within OO languages that support single inheritance, since a qualification keyword may point only to one super-class.

When a method can take many arguments, upon which the modification in the behaviour depends, the method is then called a *multi-method*. In this case, the kind of polymorphism that the OO environment offers is called *multiple polymorphism*, since multiple behaviours can be achieved not only in different sub-classes of the server object, but also in different sub-classes of the arguments themselves. In environments that only support single polymorphism such as C++, argument-based behaviour can be achieved by a technique called *double-dispatching* [21].

2.3.1.9 Objects as instances of classes, state and behaviour of objects

Classes and objects are definitely two different concepts, although related, since an object is an instance of its class. Classes should be static prior to their instantiation since objects can not be created otherwise. Objects, however, are behaving dynamically since they are created, destroyed and their state often undergoes changes.

Booch [21] views an *object*, both as an instance of a class and as a "...tangible entity that exhibits some well-defined behaviour". He points out that objects can be more than material and the following are a few examples:

- A tangible and/or visible thing.
- Something that is apprehended intellectually.
- Something towards which thought or action is directed.

The software concept of the object was first introduced in the Simula programming language. Egenhofer and Frank [59] in their compact, three-part definition of an object, state that it is "...any entity, independent of whatever complexity and structure, may be represented by exactly one object". This is a definition regarding the structural aspect of an object, and implies that any object may be composite, otherwise the complexity of the system can not be modeled adequately. The operational aspect of an object can also be formally defined as a set of operations which are performed on complex objects and "...are possible without having to decompose the objects into a number of simple objects". The behavioural aspect of the object is defined by the rule according to which " a system must allow its objects to be accessed and modified only through a set of operations specific to an object type".

An object besides being well defined exists throughout time, both in the real world and in the environment where it is manipulated. The total lifetime of an object is often called its *life cycle*. During the life cycle of an object, its structure and/or its contents may be altered. Booch [21] incorporates this fact

in his definition, by stating that an object "...has state, behaviour and identity; the structure and behaviour of similar objects are defined in their common class; the terms instance and object are interchangeable".

As mentioned earlier, objects are created from classes from their instantiation. When an object is created, every single attribute it contains must initialize its value, according to constraints found within the class definition. When an object need no longer exist, it is destroyed, and it is therefore subject to garbage collection, so as to allocate free space. The OO system is usually responsible for the free-space allocation. The object space that the object resides in can be either transient, in the case of RAM memory storage, or persistent in the case of permanent disk-storage. Transient object spaces are usually found in programming languages, within the scope of a program, while persistent object spaces are found in database management systems. However, the object space within a database transaction, can be transient as well, which means that data are temporarily processed in memory for validation rules to be applied and checked, before they are finally stored in the persistent object space of the database. Usually, persistent object spaces are larger than transient ones, because the reason to use a database system is to store data larger in size than a program will manipulate. Moreover, in terms of physical storage, RAM memory is smaller in size than permanent media (like hard disks). Objects stored within a persistent space are said to "survive" both in terms of behaviour and state, after the termination of a transaction and are also called *persistent objects* as opposed to *transient objects*, which are deleted after their usage.

Khoshafian and Abnous [87] uses the notion of *extension* or extent of a class, to denote a set of instances of a class, which have been created but not destroyed within an object space. In systems where a class can be an object as well, then access to the extension of a class is feasible through the class object itself. This is a convenient way to process objects that belong to the same class, especially in database management systems, where queries regarding the retrieval of objects that belong to the same class are often encountered. Many database systems

support implicit class extension (e.g. Java). In the case where class extension is not supported explicitly, the same goal can be achieved through *collection objects*, which are usually objects referencing to all objects belonging to the same class. Elements that belong to the collection of objects described by a class are called *instances* of the class.

Khoshafian and Abnous [87] identifies the three main properties of objects:

1. object type (or class),
2. object state and
3. object identity.

The above differentiation clearly shows the importance and independence of object identity.

Real world objects that are being modeled appear to have an inherent time and event dependent behaviour and structure: they very often modify their attributes and might change their behaviour as well. This fact must also be incorporated in the object model, via the concept of *object state*, which, according to Booch [21] "...encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties". It can therefore be viewed as the properties along with the specific values of the object's properties in a given moment of its life cycle.

As already mentioned in the previous chapter, the concept of state is a concept quite similar to that of the primary key of a table in a relational database: a primary key is composed of specific attributes of the record and characterizes the uniqueness of it. In OO environments, the specific value set characterizes the state of the object, but not necessarily its uniqueness, since the same values may be shared among objects with different identities. Therefore the concept of object identity is used instead.

Complementary to object state, is the *object behaviour*, which according to Booch [21] "...is how an object acts and reacts, in terms of its state changes and mes-

sage passing". It is the part of the object which is usable by its clients. The fact that an object shows behaviour, which is usable by other objects, makes obvious the requirement that objects can not exist by themselves, since operations and message passing is done among different objects. The values of an object's attributes may change due to its own or other object's behaviour. Hence, there has to be a relation between object state and behaviour. Booch [21] comments on this relation and views the state as "...the cumulative results of its behaviour".

The lifetime of an object, extends from the time it is first created, until it is destroyed, and therefore the space it used to allocate is freed. Creation and destruction of objects is done either explicitly or implicitly. The latter occurs in the case of aggregate objects, since its parts must be created/destroyed when necessary. Booch [21] characterizes the object creation/destruction as transitive. However, when this is not desired, the user can declare which parts of the aggregate object will be created/destroyed by overriding the semantics of the copy constructor and assignment operator or the destructor method, respectively. This is a policy encountered in C++. Some OO environments (like Smalltalk) automatically destroy aggregate objects whose parts have been destroyed as well. This procedure is part of the garbage collection feature. When this feature is not available, aggregate objects continue to exist, even if their parts have been destroyed. Nonetheless, the OO environment should provide the user with the choice whether or not to destroy aggregate objects with non-existing parts.

2.3.1.10 Metaclasses

A *metaclass* is a class whose instance is not an object but also a class. It can also be described as the class of a class. Metaclasses are mostly used to model the class hierarchy schema itself, as the class hierarchy is used to model the state and behaviour of instantiated objects. In some OO environments, there is a distinction between two types of objects:

- *Class objects*, which are templates and can instantiate other objects as well

as themselves. These could be classes or metaclasses. They usually comprise attribute definitions, interface specifications and code.

- *Terminal objects*, which can be instantiated but can not create other objects. They usually comprise values and code.

OO environments support the notion of metaclass either explicitly or implicitly: in the first case, appropriate language constructs are necessary for the manipulation of metaclasses. In the second case, metaclasses are hidden from the user and there is a one to one correspondence between a metaclass and the class it models, since metaclasses have only one instance, namely the class that they model.

Some of the advantages that make the notion of metaclasses useful are:

- Storage of group information: It is a convenient way to centrally and globally store information on object groups, that is objects belonging to the same class, should these objects be instances of classes or classes as instances of metaclasses. This is a quite useful construct especially in database management systems, where queries on objects of the same template (class/metaclass) are very often encountered.
- Storage of initialization methods: a metaclass can hold information on initializing instance variables for classes that they model, as in turn, classes, might hold information on initialization of instance variables for objects that they model.

The concept of stereotype found in the UML, is similar to the one of the metaclass as it not only tries to model any schema artifact that is encountered with the same structure frequently, but it can also introduce new ones.

2.4 Analysis and Design Techniques

2.4.1 Introduction

One of the most frequently encountered areas that use OO methodologies are the *analysis and design techniques.* The three main methods of OO analysis and design that are briefly examined within this chapter are:

- Grady Booch's Object-Oriented Analysis and Design (OOAD)
- Rumbaugh *et al.*, widely know as Object Modeling Technique (OMT)
- Jacobson's Object Oriented Software Engineering methodology (OOSE)
- Coad/Yourdon's Object Oriented Analysis and Design (OOAD)

Moreover, the very recent OMG's proposal, called Unified Modeling Language (UML), which is an amalgamation of the previous first three methodology proposals, is more thoroughly examined, since it has been incorporated in the final design. Because all three methods mentioned earlier are used within the UML proposal, little regard has been given to the variations among them. The focus of the section is upon the functionality that the UML offers.

What do OOAD techniques offer as benefits? According to Khoshafian and Abnous [87] "...with object-oriented methodology we eventually achieve a linear expansion in effort as a function of size or functionality".

Booch [21] distinguishes among three types of analysis and design methods:

- Top-down structured design, which implies algorithmic decomposition. This category does not include the notions of data abstraction, information hiding, or concurrency.
- Data-driven design, which does not address the issue of time-critical events.
- Object-oriented design, which models complex software systems as collections of cooperating objects, which in turn are instances of a class, out of a class hierarchy.

He also suggests that "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". It is obvious that the concept of system decomposition is inherent and necessary in the process of the OO design. Regarding object oriented analysis, he defines it as "...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 Analysis is the stage where objects and their members are abstracted from the application domain. According to Coad/Yourdon (1991) the proposed sequence of steps in the analysis stage are:

- Subject layer analysis
- Class and object layer analysis
- Structure layer analysis
- Attribute layer analysis
- Service layer analysis

Each of the above steps takes place in every viewpoint (or module) of the overall design. Therefore, the viewpoints must be defined before analysis initiates.

Following the analysis stage, next is the design phase. In *Object Oriented Design*, identified objects are grouped into classes, classes are grouped into packages (or modules) and classes are refined with details about their structure and behaviour. Their relationships with other classes are also defined here. The design requires an appropriate modeling language to act as an interface and tool for the user. According to the UML authors (Booch, Rumbaugh and Jacobson), a modeling language must include:

- Model elements: fundamental modeling concepts and semantics.
- Notation: visual rendering of model elements.

– Guidelines: idioms of usage within the trade.

It is widely accepted that no official OOAD technique or modeling language existed before the creation of the UML. The three individual approaches mentioned earlier (Booch's, Jacobson's and Rumbaugh's) share similarities, they do have differences, but when a choice has to be made regarding a language to work with, a decision must be made. However, comparing the three individual approaches and choosing the most suitable one (which will definitely be in need of amendments and additions probably taken from the other two which were rejected) has been avoided, since the answer to this issue came with the recent industry-standard development in OO analysis and design techniques, the UML, which is exactly the merging of the three approaches mentioned earlier. The Object Management Group's Analysis and Design Task Force (ADTF) adopted the UML version 1.1 in November 1997. Maintenance of the UML was taken over by the OMG Revision Task Force (RTF).

However flexible and powerful object oriented methodologies are, they still show inadequacies, mostly regarding the following ([122]):

– There is a tendency to focus on objects at the lower level without taking into account the "scaling-up" of related objects into groups.

– There is no comprehensive methodology which provides clear guidance from the requirements stage to implementation. However, this is being overcome by the recent proposal called the Unified Objectory Process.

– Notations are usually imprecise and they lack semantic detail. Even if additional detail is supported in the notational language, it is the designer's responsibility to add and organize it.

– Analysis and design output is not effectively reused. Ways of reusing such material is still under research.

– There are not enough guidelines and definitely no formal methodology as to how to identify classes and objects.

– Concurrency and orthogonal persistence are still not adequately supported.

– There is no standard object oriented data model.

– There is no standard object oriented query language.

There has been an effort to resolve many of the above issues by groups such as OMG and by standards such as CORBA.

2.4.2 The UML approach

The Unified Modeling Language fuses the concepts of Booch, OMT, and OOSE. The result is a single, common, and widely usable modeling language for users of these and other methods. UML was developed jointly by Grady Booch, Ivar Jacobson, and Jim Rumbaugh at Rational Software Corporation, with contributions from other methodologists, software vendors, and users. Based on extensive use of the Booch, OMT, and Jacobson methods, the UML is the evolution of these and other approaches to object and component modeling.

According to the authors of the UML [22], the UML "...is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems." The two major parts that the UML is composed of (which are common with any other OO analysis and design techniques) are

– the semantics, and

– the notation that is uses.

It is noteworthy that a special component found in the UML approach, not found in other OOADT is the Object Constraint Language Specification explained later in this chapter.

The semantics part of the UML consists of three views:

- The abstract syntax: UML class diagrams are used to present the UML metamodel, its concepts (metaclasses), relationships, and constraints. Definitions of the concepts are included.
- The "well-formedness rules": The rules and constraints on valid models are defined. The rules are expressed in English and in a precise Object Constraint Language (OCL). OCL is a specification language that uses simple logic for specifying invariant properties of systems comprising sets and relationships between sets.
- The semantics: The semantics of model usage are described in English prose.

The three component views of the UML make mathematical expressions, used to express the model, unnecessary.

The *UML Notation* component is about graphical notation and textual syntax, which are the most visible part of the UML, and are used by users. These are representations of a user-level model, which is semantically an instance of the UML meta-model.

User-defined extensions of the UML are enabled through the use of stereotypes, tagged values, and constraints. Two extensions are currently defined: 1) Objectory Process and 2) Business Engineering.

In order to reduce potential confusion between implementations, the following terms have been defined within the UML proposal:

- UML Variant - a language with well-defined semantics that is built on top of the UML metamodel, as a metamodel. It specializes the UML metamodel, without changing any of the UML semantics or redefining any of its terms.
- UML Extension - a predefined set of so-called Stereotypes, TaggedValues, Constraints, and notation icons that extend and customize the UML for a specific domain. It is envisaged that this component will be used to add spatio-temporal modeling capabilities to the UML proposal.

These diagrams, are the primary components that a developer is visually using, although the UML and supporting tools will provide for a number of derivative views. The notation, enables the creation and refinement of the aforementioned views, within an overall model representing a problem domain and software system. Moreover, it provides graphical icons to represent each kind of model element and relationship.

2.4.3 UML elements

To capture the products of object-oriented analysis and design, a logical and a physical model are necessary. The overall model that is used within UML has the elements shown in table 2.3:

Table 2.3: UML elements

Classes
Metaclasses
Interfaces
Use cases
Logical packages
Operations
Component packages
Components
Processors
Devices
Relationships
Messages

Each of these elements is described in more detail in the next paragraphs. It must be noted that the goal of this section is not to serve as a UML tutorial but to emphasize the major components that have been used in the design of the

spatio-temporal model and to demonstrate how they can be used to effectively model spatio-temporal information. Not all of these elements have been used throughout the analysis and design stage, since some of them are of no particular use.

2.4.3.1 Classes

A UML class captures the common structure and common behaviour of a set of objects. The instances of the class are referred to as objects. For each class that has significant temporal behaviour, a state diagram may be created to describe this behaviour. Classes are declared in class diagrams and used in most other diagrams. UML provides a graphical notation for declaring and using classes, as well as a textual notation for referencing classes within the descriptions of other model elements. A class represents a concept within the system being modeled. Classes have data structure, behaviour, and relationships to other elements. The name of a class has scope within the package in which it is declared and the name must be unique (among class names) within its package.

A class icon is drawn as a 3-part box, with the class name in the top part, a list of attributes (with optional types and values) in the middle part, and a list of operations (with optional argument lists and return types) in the bottom part.

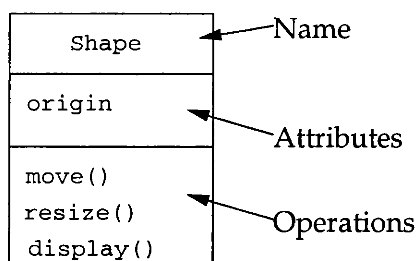


Figure 2.6: Class icon

The following UML features are associated with every class and must be set explicitly by the user:

- *Abstract*: Defines the class as a class with no instances. The abstract adjective identifies a class that serves as a base class, defining both operations and state that will be inherited by subclasses. An abstract class has no instances. A class that has one or more abstract operations is abstract as well. Abstract notation is indicated by displaying the class name in italics.

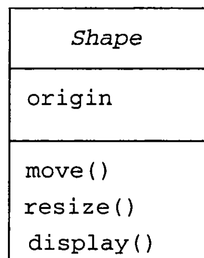


Figure 2.7: Abstract class icon

- *Attributes*: Specifies the parts in the case that the class is aggregate. Attributes may be characterized as:
  * *Public*: The attribute is visible outside of the enclosing class and can be imported into other portions of the design. Public visibility is marked using is a "+ " sign prefix on the attribute.
  * *Protected*: The attribute is visible only in the class in which it is defined. Protected visibility is marked using is a "#" sign prefix on the attribute.
  * *Private*: The class is visible only within itself and to any other elements declared as friends. Private visibility is marked using is a "- " sign prefix on the attribute. It must be noted that visibility specifications found in UML match the ones found in most popular programming languages as C++, Java, Ada and Eiffel. This type of adornment makes feasible the implementation of security policies when multiple developers are involved.
- *Operations*: Specifies the services provided by the class. An operation has a name and a list of arguments. Some of the information can also be displayed inside icons representing classes in the class diagrams. Operations may have

the same characterizations as the attributes (public, private and protected). An operation that does not modify the system state (one that has no side effects) is indicated by the keyword "{query}" [6]. A class-scope operation is shown by underlining the name and type expression string. An instance-scope operation is the default and is not marked. The set of operations found in a class actually defines its interface that is being exposed to other classes. This, however, should not be confused with the UML artifact interface, which is explained further below.

- *Cardinality*: Specifies the number of instances for the class, always in conjunction with another class through a class relationship. Cardinality specifies how many instances of one class may be associated with a single instance of another class. When a cardinality characterization is given to a class, the number of instances allowed for that class is indicated. When a cardinality characterization is given to a relationship, the number of links allowed between one instance of a class and the instances of the another class is indicated. Values are presented in "lower-bound .. upper bound" format.
- *Concurrency*: The concurrency of a class is a statement about its semantics in the presence of multiple threads of control. The concurrency of a class can be set to one of the following types:
  * *Sequential*: The semantics of the class are guaranteed only in the presence of a single thread of control. Only one thread of control can be executing in the method at any one time.
  * *Guarded*: The semantics of the class are guaranteed in the presence of multiple threads of control. A guarded class requires collaboration among client threads to achieve mutual exclusion.
  * *Active*: The class has its own thread of control. The method can be executing concurrently with other methods.

[6] This is quite similar to the query concept found in DBM systems, although queries are modelled as stereotypes, as it is discussed in chapter 3

    * *Synchronous*: The semantics of the class are guaranteed in the presence of multiple threads of control; mutual exclusion is supplied by the class.

  - *Visibility*: Specifies how the class is seen outside of the package in which it is defined. Class visibility specifies the export control and import status of the class. Export control specifies how this class is seen outside of the package in which it is defined.
  - *Components*: Specification of the software modules that realize the class.

A class utility is a grouping of global variables and procedures in the form of a class declaration. This is not a fundamental construct but a programming convenience. The attributes and operations of the utility become global variables and procedures. Therefore, it is inappropriate for a utility to declare class-scope attributes and operations because the instance-scope members are already interpreted as being at class scope. It is shown using the keyword "utility" and its cardinality has the value of zero, meaning it can be instantiated only once. Additionally, classes have responsibilities, which is a contract or an obligation for a class. UML models responsibilities in a free textual form within the class icon in a different compartment. The class element is the mostly used UML element throughout the MPOOST model, as it is the fundamental OO artifact. Abstract classes will mostly be used to model general categories that apparently can not be instantiated to objects.

2.4.3.2 Metaclasses

A metaclass is a class whose instances are classes rather than objects. Metaclasses provide operations for initializing class variables and serve as repositories to hold class variables where a single value will be required by all objects of a class (constant values). A metaclass is displayed as a 3-part box, with the class name in the top part, a list of attributes (with optional types and values) in the middle part, and a list of operations (with optional argument lists and return types) in the bottom part.

Regarding relationships:

- An "inherits" relationship may be defined between a metaclass and an instantiated class or another metaclass
- An "association" relationship may be defined between a metaclass and a class, a parameterized class, an instantiated class, another metaclass, a class utility, a parameterized class utility, an instantiated class utility, or an interface.
- A "dependency" relationship may be defined between a metaclass and a class, a parameterized class, an instantiated class, another metaclass, a class utility, a parameterized class utility, an instantiated class utility, or an interface.

The following metaclass features may be defined explicitly by the user:

- Attributes: Definition of the parts of the aggregate object.
- Cardinality: The number of instances for the class.
- Concurrency: The semantics in the presence of multiple threads of control.
- Operations: The services provided by the class.
- Persistence: The definition of the lifetime of the instances of a class.
- Visibility: Specification of how the class is seen outside of the package in which it is defined.

2.4.3.3 Interfaces

An interface specifies the externally-visible operations of a class and/or component, and has no implementation of its own. An interface typically specifies only a limited part of the behaviour of a class or component. Interfaces belong to the logical view but can occur in both class and component diagrams.

In component diagrams, an interface is displayed as a small circle with a line to the component (explained further below) that realizes the interface.

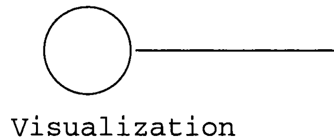


Figure 2.8: Interface icon in component diagrams

In class diagrams, an interface is represented by a class icon with the stereotype "interface." Thus, it is a 3-part box, with the interface name in the top part, a list of attributes (usually empty) in the middle part, and a list of operations (with optional argument lists and return types) in the bottom part.

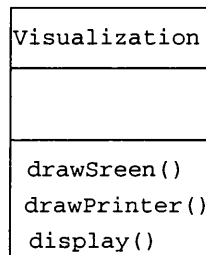


Figure 2.9: Interface icon in class diagrams

The attribute and operation sections of the interface class box can be suppressed to reduce detail when shown in a generic overview of the schema. Suppressing a section makes no statement about the absence of attributes or operations, but drawing an empty section explicitly states that there are no elements in that part. A "generalization" relationship may be drawn from one interface to another interface.

The following features may be defined for an interface:

- Abstract: Defines the interface as a base class with no instances.
- Cardinality: The number of instances for the interface.
- Concurrency: Definition of the semantics in the presence of multiple threads of control.
- Operations: The services specified by the interface.

– Visibility: Specification of how the interface is seen outside the package in which it is defined.

Interfaces are the means by which behaviour that is common to many classes is modelled, without including the actual implementation. For instance, a circle and a rectangle class may both have associated a display method. This method name can be part of the "visualization" interface, so that both classes are ensured to realize it. It may be that later in the design stage more classes have to realize this set of methods offered by this specific interface.

2.4.3.4 Use cases

A use case is a sequence of transactions performed by a system in response to a triggering event initiated by an actor to the system. In other words, they provide the users the ability to interact with the system. Users are modelled by actors in the context of UML (explained further below in this chapter). A full use case should provide a measurable value to an actor when the actor is performing a certain task. A use case contains all the events that can occur between an actor-use case pair, not necessarily the ones that will occur in any particular scenario. A use case contains a set of scenarios that explains various sequences of interaction within the transaction. A use case can also describe the behaviour of a set of objects, such as an organization. The basic shape of a use case is an ellipse:

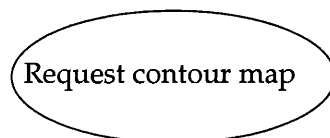


Figure 2.10: Use case icon

Use cases as they are part of the behavioural component of the system, will be used to model the possible interactions that users is anticipated to have with the model. Any kind of queries will be initially modeled by use cases.

2.4.3.5 Objects

An object is an instance of a class, and has state, behaviour, and identity. The structure and behaviour of similar objects are defined in their common class. Each object in a diagram indicates some instance of a class. An object that is not named is referred to as a class instance. If the same name is used for several object icons appearing in the same collaboration diagram, they are assumed to represent the same object, otherwise each object icon represents a distinct object. Object icons appearing in different diagrams denote different objects, even if their names are identical. If the name of the object's class is specified in the Object Specification, the name must identify a class defined in the model. The object icon is similar to a class icon except that the name is underlined. If multiple objects exist that are instances of the same class, the icon used is one with three staggered objects. Optionally, a name can also be given to the object, so that it can be uniquely identified throughout the object diagram where it is usually shown.

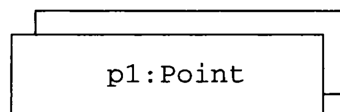


Figure 2.11: Single (left) and Multiple (right) Object icons

.

Whether the object is persistent or transient, it is also written in the object box. Objects will be used in specific object diagrams of the model, so as to illustrate specific cases of interest.

2.4.3.6 Packages

A package is a grouping of model elements. Packages themselves may be nested within other packages. A package may contain both sub-packages and model elements. The entire system description can be thought of as a single high-level

subsystem package with everything else in it. All kinds of UML model elements and diagrams can be organized into packages. It is worth noting that packages own model elements and model fragments and are the basis for configuration control, storage, and access control. Each element can be directly owned by a single package, so the package hierarchy is a strict tree. However, packages can reference other packages, so the usage network is a graph.

2.4.3.6.1 Logical packages Logical packages serve as means to partition the logical model of a system. They are clusters of highly related classes that are themselves cohesive, but are loosely coupled relative to other such clusters. Packages can be used to group classes, interfaces, and other packages as well. While many OO programming languages do not yet support this concept, using packages in a class diagram allows the expression and preservation of important architectural elements of the system design. A high-level design of the system may be captured simply by creating a class diagram that consists only of packages. The logical package is a folder shaped icon.

2.4.3.6.2 Component packages Component packages represent clusters of logically related components. Component packages parallel the role played by logical packages for class diagrams. They allow the partitioning of the physical model of the system. Typically, a component package name is the name of a file system directory. A component package can have dependencies with other component packages, components, and interfaces. The component package is a folder shaped icon as well.

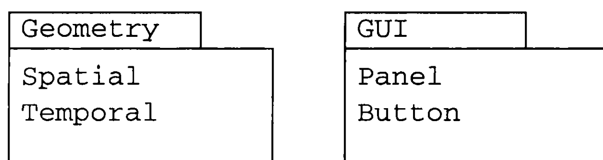


Figure 2.12: Logical (left) and Component (right) package icons

Packages are the means by which classes will be grouped. General data categories

can also be implemented using packages, instead of using abstract classes, since packages offers a greater degree of both encapsulation and grouping.

2.4.3.7 Operations

The information in the Operation Specification is presented textually. Some of the information can also be displayed inside icons representing classes in the class diagrams. One operation specification should be completed, for each operation that is a member of a class and for all free subprograms.

2.4.3.8 Components

A component represents a software module (e.g. source code, binary code, executable, dynamically linked library, etc.) with a well-defined interface. The interface of a component is represented by one or several interface elements that the component provides. Components are used to show compiler and run-time dependencies, as well as interface and calling dependencies among software modules. They also show which components implement a specific class.

A component icon is drawn as a large rectangle with two smaller rectangles attached to its left side. An interface circle attached to the component icon means that the component supports that particular interface. That is, there is no explicit relationship arrow between a component and its interfaces.

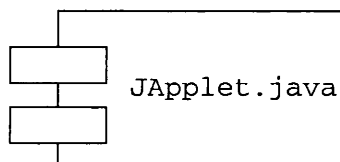


Figure 2.13: Component icon

Components may be used in the model to denote that services from existing programs that are already in binary format are being used. This however will be avoided as much as possible since often components that denote binary format are

strongly connected to specific software architectures, mainly operating systems. This of course does not apply in the case where the component refers to code that is in pre-compiled ASCII format, e.g. like Java files, or when the compiled code is platform independent, e.g. Java bytecode [7]. Moreover, components are the appropriate elements to model the physical implementation of the system in terms of files (executables, binaries, data etc.)

2.4.3.9 Processors

A processor is a hardware component capable of executing programs. Each processor must have a name. There are no constraints on the processor name because processors denote hardware rather than software entities. The icon for a processor is a shaded box:

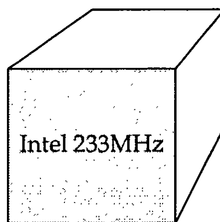


Figure 2.14: Processor icon

A processor is used within the model at a specific node to denote a local machine capable of executing code, usually a computer connected to the network that forms the distributed environment.

2.4.3.10 Devices

A device is a hardware component with no computing power. Each device must have a name. Device names can be generic, such as "modem" or "terminal." The icon for a device is a box. Sometimes, an icon representing the actual appearance of the device may be used instead.

[7] An exception to this is whenever the binary-format component is built including the usage of an interoperability standard (like CORBA, COM+, or Java Beans).

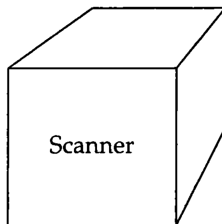


Figure 2.15: Device icon

Devices are excellent UML elements to model all the hardware peripherals of the system which will be used for data input and information output like digitizers, scanners, photogrammetric instrumentation, printers, plotters etc.

A processors may be linked to another processor as well as to a device, and the association is adorned with the type and characteristics of the connection employed, e.g. 10/100 Mbit Ethernet between two processors, or RS232 between a processor and a device.

2.4.3.11 Relationships

A relationship is the way that two model elements are connected and related between them. Different model elements may involve different kinds of relationship. In every relationship its name is defined and used to identify the type or purpose of the relationship. The following types of relationships are defined in the UML:

- "Generalize/Inherits": A "generalize" relationship between classes shows that the subclass shares the structure or behaviour defined in one or more super-classes. Usage of a "generalize" relationship is to show an "IS-A" relationship between classes. A "generalize" relationship is a solid line with an arrowhead pointing to the super-class (figure 2.16).

The definition of a "generalize" relationship may be done by specifying its access, identifying whether the class grants rights to another class, and identifying the super-class as a base class. The following items may be set:

- Access: public, private, protected, or implementation.

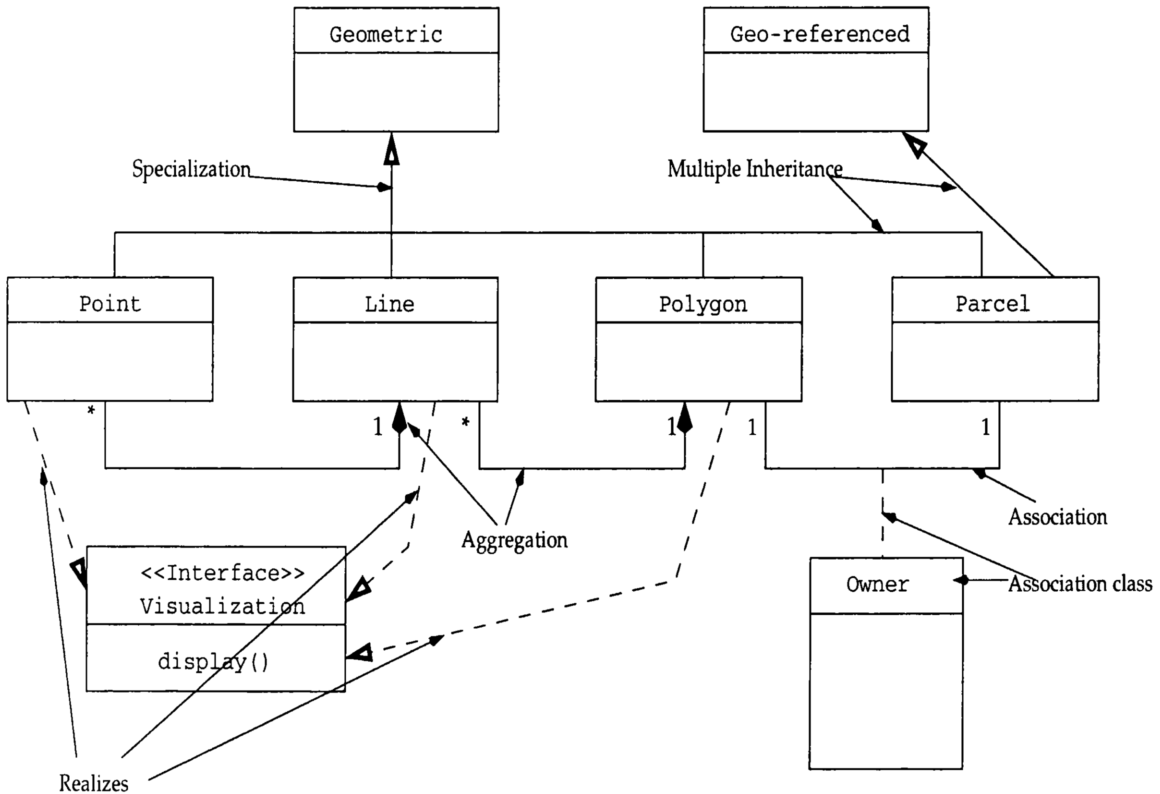


Figure 2.16: Notation used for Relationships

- Friendship required: the friend characterization is used to designate that the supplier class has granted rights to the client class to access its non-public parts.
- Virtual Inheritance: in a multiple inheritance situation, the virtual characterization may be applied to ensure that only one copy of the base class will be inherited by descendants of the subclasses.

An inheritance tree is useful when a number of subclasses share the behaviour and structure of a super-class. Rather than creating a multitude of inheritance relationships pointing directly from the subclasses to the super-class, the relationships may be tied together to form one link to the super-class, thereby creating a tree structure.

- "Aggregate" Relationship: The "aggregate" relationship shows a whole and part relationship between two classes. The class at the client end of the

aggregate relationship is called the aggregate class. An instance of the aggregate class is an aggregate object. The class at the supplier end of the "aggregate" relationship is the part whose instances are contained or owned by the aggregate object. The aggregate object is physically constructed from other objects or it logically contains another object. The aggregate object has ownership of its parts. An "aggregate" relationship is a solid line with a diamond at one end. The diamond end designates the client class (figure 2.16).

.

The following features may be defined for an "aggregate" relationship by specifying its access, cardinality, containment, and whether the instance of the part class is owned by the aggregate class.

- Containment: The physical containment for every "aggregate" relationship containment is either by-reference or by-value.
  * Cardinality: this may be specified for the client class, supplier class, or both
  * Static: used to specify that the instance of the part class is owned by the class itself not by its individual instances.
  * Access: For each aggregate relationship, the type of access allowed for the relationship may be set, namely public, private, protected, or implementation.
- "Association" Relationship: An association represents a semantic connection between two classes, or between a class and an interface. Associations are bi-directional: they are the most general of all relationships and the most semantically weak. This type of relationship, although it is general, however it is useful mainly in the early stages of analysis and early design: initially the identification of general dependencies between classes may be done. As the model evolves, additions are possible to associations so as to make them

more precise. An "association" relationship is an oblique or orthogonal line (figure 2.16).

An "association" relationship may be defined between all types of classes and interfaces. A variety of characterization and properties may be defined to "association" relationships. These are: derived, name direction, documentation, roles, cardinality, navigability, aggregate, static, friend, access, containment, association and role constraints, link elements, and qualifiers.

- "Dependency" relationship: a dependency relationship between two classes, or between a class and an interface, is used to show that the client class depends on the supplier class/interface to provide certain services, such as:
  * The client class accesses a value (constant or variable) defined in the supplier class/interface.
  * Operations of the client class invoke operations of the supplier class/interface.
  * Operations of the client class have signatures whose return class or arguments are instances of the supplier class/interface.

A dependency relationship is a dotted line with an arrowhead at one end. The arrowhead points to the supplier class.

A dependency relationship may be defined only between logical packages.

- "Realize" Relationship: A "realize" relationship between classes and interfaces and between components and interfaces shows that the class realizes the operations offered by the interface. A "realize" relationship is a dashed line with an arrowhead pointing to the interface (figure 2.16).

A "realize" relationship may be defined between a class and an interface or a component and an interface. The relationship between a component and an interface cannot be defined explicitly. It is usually defined when an interface is assigned to a component.

Relationships between classes may also be modeled as classes themselves in terms of having structure. The association class is used to model properties of associations. The properties are stored in a class and linked to the association relationship. Association classes are also referred to as Link Attributes in OMT literature. Link attributes are degenerate association classes comprised only of attributes. An association class is a class linked to an association by a loop (figure 2.16). An association class may be defined between all types of classes. Furthermore, for an association class its access, cardinality, concurrency and persistence characteristics may be defined as well.

- Meta relationship: The meta relationship is used to show the relation between a class and its metaclass. The meta relationship is a gray line with an arrowhead pointing towards the metaclass.

A meta relationship may be defined between a class and its metaclass, as well as between a parameterized class and its metaclass.

- Links: When classes are instantiated to produce objects, associations become links between objects. Objects interact and pass messages through their links to other objects. A link is defined as an instance of an association. A link should exist between two objects, including class utilities, only if there is a relationship between their corresponding classes. The existence of an relationship between two classes symbolizes a path of communication between instances of the classes: one object may send messages to another. Links can support multiple messages in either direction. If a message is deleted, the link remains intact. The link is depicted as a straight line between objects or objects and class instances in a collaboration diagram. If an object links to itself, the loop version of the icon is used instead.

2.4.3.12 Messages

A message conveys the source object's invocation of an operation from the destination object. Messages are carried by links. A message is represented on collaboration diagrams and sequence diagrams by a message icon which visually indicates its synchronization. A message's synchronization can be modified via the message's specification. The following synchronization types are supported:

- Simple: For messages with a single thread of control, one object sends a message to a passive object.
- Synchronous: In synchronous messages, the operation proceeds only when the client sends a message to the supplier and the supplier accepts the message. The client runs until it sends the message; it then waits for the supplier to accept it. The client continues to wait until the message is accepted.
- Balking: In balking synchronization, the client can pass a message only if the supplier is immediately ready to accept the message. The client abandons the message if the supplier is not ready.
- Timeout: In timeout synchronization, the client abandons a message if the supplier cannot handle the message within a specified amount of time.
- Asynchronous: Asynchronous communication occurs when the client sends a message to the supplier for processing and continues to execute its code without waiting for or relying on the supplier's receipt of the message.

2.4.3.13 General extensibility mechanisms

The elements in this category are general-purpose mechanisms that may be applied to any modeling element. The semantics of a particular use depends on a convention of the user or an interpretation by a particular constraint language or programming language, hence they constitute an extensibility device for UML.

2.4.3.14 Constraints

A constraint is a semantic relationship among model elements that specifies conditions and propositions that must be maintained as true, otherwise the system described by the model is invalid. Certain kinds of constraints (such as an association "or" constraint) are predefined in UML, others may be user-defined. A user-defined constraint is described in words in a given language, whose syntax and interpretation is a tool responsibility. A constraint represents semantic information attached to a model element, not just to a view of it. A constraint is shown as a text string in braces ({}).

A comment is a text string attached directly to a model element. This is syntactically equivalent to a constraint written in a non-programming language whose meaning is significant to humans but which is not "executable", as long as humans are regarded as the instruments of interpretation. A comment can therefore attach arbitrary textual information to any model element of general importance.

Constraints will be widely used throughout the MPOOST model so as to enhance the integrity requirements of the data that are being modelled. Comments will mostly be used for diagram comprehensibility purposes, although sometimes for modelling rules that are too difficult or not straightforward to be modelled using constraints.

2.4.3.15 Stereotypes

A stereotype represents a subclass of an existing modeling element with the same form (attributes and relationships) but with a different intent. Generally, a stereotype represents a usage distinction. A stereotyped element may have additional constraints on it from the base class. The general presentation of a stereotype is to use the symbol for the base element but to place a keyword string above the name of the element; the keyword string is the name of the stereotype within matched guillemets ("").

Stereotypes are the means by which the UML can be adjusted to model spatial and temporal information. Most of the UML elements will be stereotyped, before they will be used within the MPOOST model. For example, association relationships between classes can be e.g. «temporal», classes can be e.g. «geometric», operations can be «query», «modifier»etc.

2.4.4 UML diagrams

Each of the model elements examined earlier has properties that identify and characterize it. A model also contains diagrams and specifications, which provide a means of grouping, visualizing and manipulating the model's elements and their model properties. Since diagrams are used to illustrate multiple views of a model, icons representing a model element can appear in none, one, or several diagrams. In terms of the views of a model, the UML defines the following graphical diagrams:

1. use case diagram
2. class diagram
3. behaviour diagrams:
   (a) statechart diagram
   (b) activity diagram
   (c) interaction diagrams:
      i. sequence diagram
4. collaboration diagram
5. implementation diagrams:
   (a) component diagram
   (b) deployment diagram

2.4.4.1 Class and object diagrams

A class diagram is a graph of class elements connected by their various static relationships. It shows the static structure of the model, in particular, the elements that it consists of, such as classes and types, their internal structure, and their relationships to other elements. Class diagrams do not show temporal information, although they may contain occurrences of elements that have or describe temporal behaviour.

An object diagram is a graph of class instances, including objects and data values. It is an instance of a class diagram and shows a snapshot of the detailed state of a system at a point in time.

Additional notation that might be necessary is [122]:

- Class hierarchy collapse, so that a class schema can be viewed in any level, abstracting and hiding information in "higher" levels but showing more detail in "lower" levels.
- A part which is dependent on the existence of its assembly, as well for the assembly class itself.

Dependent assembly classes can be straightforwardly notated by using stereotypes. Hierarchy collapsing, in other words omitting unwanted classes and relationships can be done by adding additional notation to classes of higher level, so as to denote that these are specialized by other lower level ones.

2.5 Implementation - Languages and databases

The outcome from the analysis and design as discussed so far is a formal specification of how the system, or model should be structured and behave. This formal specification involves the usage of a visual language, like the UML, and it does not involve the implementation on any software platform, since according to the software engineering rules, it should be as independent as possible of the

implementation software. [8]. The software may be a *programming language* (PL) in combination with a *database management system* (DBMS). The programming language can be independent of the DBMS or a part of it. Moreover, both can either be object-oriented or not. This chapter does not examine how non object-oriented platforms can be used to implement an object-oriented model[9] since it is beyond the scope of the thesis.

2.5.1 Object Oriented Programming Languages

Object-oriented programming is based on a paradigm of objects responding to messages, rather than on one of operators performing actions on operands, as is the case with procedural languages [89]. Booch [21] defines object-oriented programming as "...a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.". It is the researcher's opinion however, that, a programming language can be characterized as object-oriented, if and only if, it incorporates the four major concepts of object-orientation, namely abstraction, encapsulation, modularity and hierarchy (see table 2.2 earlier in this chapter).

The three significant parts in the definition given by Booch [21] that contrast with procedural programming are that an OO language involves:

1. the usage of objects, in contrast with algorithms, in the case of the procedural programming,
2. objects, that are instantiated from classes which are related through the:
3. inheritance relationship.

If the concept of inheritance is absent from an OO language, then programming

---

[8] In practice however, this does not happen, since, choosing specific implementation software, effects the early stages of the design, due to the possible lack or idioms in its functionality.

[9] Exception is the object-relational databases, which is briefly discussed.

is said to be done with abstract data types [21], mentioned earlier in this chapter. Khoshafian and Abnous [87], and Booch [21] discuss the evolution of the programming languages from procedural to object-oriented.

It must be noted that programming languages are actually interfaces between the user and the computer. They are used by the programmers to develop applications. Therefore, the end user who is not involved in programming will not be affected by the incorporation of a OO language in the design and implementation stages of a GI system. However, this does not apply in the case where the end user will be using an object-oriented query language. Nonetheless, the user has to be familiar with the OO approach, since regardless of the interface involved, users are sometimes responsible for creating and maintaining the class hierarchy. The most widely used and popular object-oriented programming languages are C++, Smalltalk and Ada. Recently, and mainly due to the growing expansion and usage of the Internet, a C++ descendant, the Java programming language is slowly becoming the most popular network programming language. Appendix A includes a comparison of OO programming languages that shows the advantages and disadvantages of every language.

Traditional programming languages have provided facilities for the manipulation of data whose lifetime does not extend beyond the activation of the program. On the other hand, if data are assumed to survive a program activation, then some kind of environment that provides persistence is needed. Different persistent types are treated differently. The mapping between the two types of data is usually done in part by the file system or the computing environment and in part by explicit user transaction code which has to be written and included in each program. A consequence of this view of data is the need for a considerable amount of program code concerned with transferring data to and from files, or the DBMS, which leads to much space and time being taken up by the code required to perform translations. What is needed is to make the quality of persistence orthogonal to type and naming. A language which applies this rule is called a *persistent language*. A good example of a persistent language is the PS-ALGOL

(see appendix A), which is considered to be the first that introduced the concept of an orthogonally persistent language. An other similar but more recent effort is found in the PJama project, which has successfully attempted to incorporate orthogonal persistence in the Java OO programming language.

Java has been used for the implementation of the MPOOST data model. This implementation includes:

- the class schema definition with some basic algorithms for every class;
- the graphical user interface that the user can use to interact with the stored information.

2.5.2 Object Oriented Databases

A successful attempt has been made to incorporate the concept of persistence introduced to the object model, within a programming language (see previous section). Booch [21] defines *persistence* as "...the property of an object through which its existence transcends time (i.e. the object continues to exist after its creation ceases to exist) and/or space (i.e. the object's location moves from the address space in which it was created).". He also states that when persistent objects are involved, their state as well as their classes remain also persistent. The latter must transcend any individual program so that "...every program interprets this saved state in the same way." This issue is one more burden for the integrity of the system, since the class of the object requires a change. This is one of the issues when manipulation of persistent objects is required. Moreover, when simultaneous retrieval of a large amount of information by different users is involved, then the requirements can not be met within the functionality provided by a programming language, even if it is orthogonally persistent. These additional user requirements, namely the transactioning capability, give rise to *database management systems* (DBMS). In the context of the object model, an *object-oriented database* is a collection of objects whose behaviour and state, and the

relationships are defined in accordance with the object-oriented model. An *object-oriented database management system* is a database system which allows the definition and manipulation of an object-oriented database [88]. Object-oriented databases combine the object-oriented concepts with programming constructs and database management capabilities [87]. In the case that they incorporate object-oriented languages to define and manage the data that are stored within, it is then that they extenuate the "impedance mismatch" between the language and the database structure. This was not the case with older, relational databases, where the integration of a programming language within was cumbersome and sometimes impossible. In practice, as Booch [21] notes, such databases build upon proven technology (e.g. network, relational etc.) and offering the object-oriented interface through database operations are manipulated in terms of objects whose lifetime exceeds that of an individual program. A known example is the object-relational databases.

In OO database management systems, the structure of the object space is defined by the so called *class schema*, which is nothing more than the fully detailed definition of the class hierarchy. The schema, as well as any object instantiated from it are both persistently stored. Incorporation of object-oriented capabilities in databases is feasible in many ways [87], namely:

- Novel database data model/data language approach (e.g. POET or ONTOS).
- By extending an existing database language with object-oriented capabilities.
- By extending an existing object-oriented programming language with database capabilities. Example is the PJama project for an orthogonally persistent Java.
- By providing expansible object-oriented database management system libraries.
- By embedding object-oriented database language constructs within a host

language. This is a technique that many commercial products offer, even if they are not pure extensions of a specific language, by providing "bindings" for popular programming languages (e.g. POET DBMS offers bindings for C++ and Java).

Object oriented databases can also be defined as combining the object-oriented concepts plus database capabilities such as the ones listed and expanded below. The first two major features (namely the transaction management mechanism and support for integrity constraint) are those that differentiate database management systems from persistent languages.

2.5.2.1 Special characteristics of object-oriented databases

2.5.2.1.1 The transaction management mechanism This is a mechanism through which objects in a database are being manipulated. A transaction, as a group of database commands, has the property of being atomic, which means that either all of the code of the transaction can be executed (if commands after being processed result in a valid database state) or none (if some or all of the commands violate the validity of the database state). The constructs used to denote the abnormal termination of a transaction is called rollback, while upon the successful termination of the transaction it is then committed. Transaction can be nested ([107] from [87]). In this case, a *nested transaction* is composed of sub-transactions, each one of should be atomic. Nested transactions can be modeled through a tree structure, with the topmost transaction as the root, and all the rest of the transactions as children. Parent-child relationship can be recursive throughout the tree (children of the root can act as parents and have their own child transactions and so on). In order for the top-level transaction to be validated, all its children transactions must be validated first.

A transaction is also a scope, within which integrity constraints (explained below) may temporarily be violated, but only until the termination of the transaction, after which they should be amended or rejected, always resulting to a valid,

consistent state of the database according to the set of integrity constraints. Database consistency is guaranteed through the serialized order of transactions. This means that even if some transactions have the same starting time, they will be executed in a specific order. Khoshafian and Abnous [87] identifies three main categories of concurrency control algorithms, namely:

- Time-stamp ordering, where every transaction is associated with a time stamp, which is usually the transaction start time. The system is responsible for executing transactions based on their time-stamp order. In the case of conflicts (usually caused by update operations) a transaction may be aborted, if any operation is inconsistent with the database status.
- Optimistic algorithms, where transactions are allowed to execute, in private spaces, until finished. In the case of conflicts, transactions are aborted, therefore resulting into loss of transaction work, otherwise committed. They are mostly applicable in databases with minimum conflicts between transactions.
- Pessimistic algorithms, where locks on persistent objects are necessary before any transaction operation takes place. Locks can be either read or write. Upon transaction termination, locks are released, therefore allowing other transactions to operate upon them. This technique is imperative in the case of multi-user databases, where objects are shared among many users. In this case, transactions must follow the two-phase locking mechanism, where every operation must acquire a lock on the objects it accesses, and after the release of the lock, no more lock should be acquired.

For concurrency to work in conjunction with the locking strategy, locks must be minimized. *Multigranule locking* [88] allows different levels of the schema to be locked, depending on the requirements of the transaction, like total class and instances locking or partial instances locking. Pessimistic and optimistic algorithms can be used in combination. Locks can be either read or write. Read locks prevent write locks and vice-versa: when an object is read, it cannot be

updated until read access operation is through. If a write operation has locked an object, read operations cannot lock it until the write operation terminates. Deadlocks are transactions that cyclically wait for each other to terminate [87]. Although deadlocks are unlikely to occur since locking is sequential, still there has to be a mechanisms so as they can be identified and prevented by the system, through termination of one of the transactions in the cycle.

Management of long transactions in an OOGIS environment involves working with virtual copies of the dataset, without the need to use redundant physical copies. Arctur [6] uses the term "Local Integration" function to denote the potential need for conflict resolution among multiple users' changes to the database. The concept of *central data warehousing* may also be incorporated through the "Global Integration" function. Moreover, usage of multi-tiered long transactions may be used for large and complex datasets. [7]. It must be noted here that relational DBMS do not provide support for long transactions.

2.5.2.1.2 Support of integrity constraints *Integrity constraints* are rules and conditions to which the state of the database must conform, therefore speaking of a consistent database state. They usually apply on values of objects' attributes. Types of integrity constraints may be [87]:

- Unique key: these constraints ensure uniqueness of an attribute's value (whenever this applies for the attribute) within the database. For example, the object identity should be unique as a value throughout the database
- Referential constraints are rules which guarantee that when an attribute's value is the identity of another object, these values always point to existing ones, therefore avoiding dangling references. In OO systems where there is support for object identity, referential constraints are implied, and therefore not necessary.
- Non-null constraints which ensure that attributes cannot be null valued, wherever this applies. These constraints make necessary the definition and

inclusion of initialization methods, mentioned earlier in this chapter.

- Domain constraints, which limit the domain of the values that an attribute can receive. This is straightforwardly implementable in the OO databases by creating appropriate, specialized classes, which obey the constraints posed and incorporating them as aggregate parts in other objects, the attributes of which should follow the domain constraints.
- General constraints posed upon values that attributes should hold, usually in combination among them, and conjunction with functions such as average, sum or count.

It is quite often inevitable that integrity constraints result in redundant relationships among database objects [103].

2.5.2.1.3 Concurrency *Concurrency* is a technique through which many transactions can be running the same time in a database, having as access target the same objects. Concurrency control is the mechanism that a DBMS supports, and checks for conflicts among simultaneous transactions. Its purpose is to guarantee a valid database state, throughout the database life.

2.5.2.1.4 Recovery *Recovery* refers to the mechanisms available within a OODBMS that implements strategies to recover data lost by events such as:

- Transaction failures.
- System failure.
- Media failure.
- Any combination of the above three reasons.

Recovery, as a way to restore lost objects after a failure, is a capability that enforces the persistence of objects.

2.5.2.1.5 Versioning Kim [88] uses the notion of *version* as a relationship among instances of a class: a versioned class is a set of objects which are versions of the same conceptual object and it consists of a hierarchy of objects which capture the "version-of" relationship between an object and another object derived from the class. The notion of version introduces two types of relationships between versioned objects: "derived-from" relationship between an arbitrary version of an object and the initial, abstract object it came from and "version-of" relationship, between the newer and the older versions of an object in a version graph. It is obvious that the two relationships are related, since the "derived-from" can be extracted from the "version-of" by tracing upwards the version graph, from the version of interest up to the root object.

Versioning refers to the mechanism found in a database that allows previous states and state transitions of objects to be stored, and later retrieved at the user's will. Transitions from state to state are usually caused by events that change the values in one or more of the object's attributes. When a new version of an object must be created, its predecessor is selected and retrieved, and the new object inherits all values from the old object. This is where the value inheritance mechanism is necessary. The user, or a function can modify any of the values in the object and thus create the new version. Khoshafian and Abnous [87] name these two phases of version creation as check-in and check-out respectively. Versioning of an object throughout its life cycle can be modeled with a tree structure, where the initial state of the object is the root of the tree. Nodes represent versions and links represent transitions. Every version, including the root object, may have one or more newer versions. Additionally, any number of versions can be combined to create a newer version, that is a node (as a $\{n\}$ level version) can have many parents and children. It is obvious that a new version can be created out of any node of the tree. Problems arise if a version has to be deleted. Then a set of versions might be necessary to be merged.

Configuration objects are special system objects that hold information on all versions of a specific object [87]. Their purpose is similar to that of container

classes (or collection objects). Object versions can be grouped together, forming a dataset version of the database. Laser-Scan's Gothic ADE supports dataset versioning.

Versioning is a useful construct especially for applications that hold historical data in the persistent repository, but many issues arise with the introduction of the object versioning mechanism. One of them regards the maintenance of the uniqueness of the object identity. Should the object identity remain the same throughout all versions of the same object, or not? If a new object is to be created, as a successor of another object, then these two cannot share the same identity, since the principle of identity uniqueness is violated. If the identity of the newly created object is different than the previous one, then the two objects are different and therefore, semantically, the new one cannot be considered as a version of the previous. The researcher's suggestion on this problem includes the insertion of a self-relating association into the root class in the schema hierarchy. This association is inherited by every class in the schema. In this way every object which is instantiated as a new version of another object relates to a same class object, referred to as the predecessor object. The predecessor object in turn relates to the its next version via the same association (but through a different link). This class relationship could possibly be associated with an event class (as an association class) so that for every version of object the system or user event that caused the versioning can be identified. In this way, identity remains unique and versions of the same object are related along with the version causing event.

2.5.2.1.6 Performance enhancement Taking full advantage of an object-oriented database requires storage management strategies as well as very powerful hardware platforms, since performance issues arise, due to the complexity of the object space structure. The storage management includes the efficient organization, storage and access of the objects. Problems that need to be resolved include mapping of the object space that contains complex objects, to the linear physical space (e.g. disk pages). Query optimization is the set of techniques that

attempt to minimize the number of accesses and the computation time that the system performs to execute a query. This is done through proper usage of access methods such as clustering and indexing.

Indexes are sorted tables that relate the value of the object identifier to the values of one or more attributes, in the same object, therefore accelerating the access for queries based on these attributes. When a table relates the OID with an attribute, the attribute is characterized as indexed. Indexed attributes must be specified explicitly in most OO databases in the schema definition (as is the case in relational databases), because the creation and maintenance of the index table imposes processing overhead. Therefore the user must choose between the processing requirements of creating an index table and the requirements of processing queries without index tables, which is usually based upon the frequency that a query is relating to a specific attribute. Indexes do not add new information to the system, but the opposite: they contribute to the problem of data redundancy, only of course to aid the system to perform faster. An important issue arising, as already mentioned, is on how to combine the usage of indexes in a OO environment, with the concept of encapsulation: for an index to be created on an attribute, the values of it must be read, for all the set of objects that include this attribute in their structure. The principle of encapsulation, however, prohibits direct read accesses to a value. A solution to the problem might involve the permission of client objects and classes to directly access object values, for read only purposes. However, this strategy still violates the principle of encapsulation. Alternatively, values can be accessed indirectly, by a method written specificly for this purpose, which is usually called an accessor method (as already discussed). This does preserve encapsulation, however it may impose access time overhead, since for every value to be read, the associated method has to be executed as well.

Clustering is the strategy by which objects that are frequently accessed together, are stored on the same disk section. Clustering strategies can be either:

- Automatic, where the system is responsible for deciding when to cluster objects and create indexes of attributes, based on the frequency and the combination by which they are retrieved.
- Manual, where the user has built predefined indexes on specific attribute combinations, that he or she knows will be accessed together. The system will not create extra indexes unless instructed by the user to do so.

Some DBMS systems support breadth-first and depth-first clustering (e.g. GemStone), for the internal organization of a complex object. In breadth-first storage, the object's attributes are stored first in their entirety, as the OIDs of the aggregate objects and, next in sequence, the values of the attributes of each one of the aggregate parts. In depth-first order, each aggregate part is fully stored (both the ID and its attribute values) before the next one in sequence is stored the same way. The choice between the two ways is a matter of how queries are based upon an object's values: for example, if queries are often based on deep-level attributes of the object, then depth-first clustering is most suitable.

Other techniques for query optimization include caching of frequent accessed objects. A comparison on performance between a RDBMS and an OODBMS (namely Oracle and ONTOS) by Milne *et al.* [103] shows the advantages of object caching which RDBMS currently are lacking of. It also shows the advantage of the group storage manager model in database organization, where objects are clustered by their class: in this way, de-referencing of objects is accelerated and the number of disc accesses is reduced. However, this technique introduces the problem of maintaining an additional copy of the object space, which means data redundancy. This may not be a serious problem, especially when storage media are becoming lower in cost. Other issues include the performance and code cost of additional mapping of the persistent object space with the temporary memory, where objects are cached as well as the performance cost due to necessary update of cached objects from their original storage source, through a network in the case of a distributed environment. For the latter two reasons object caching might

prove insufficient and a burden on performance.

Egenhofer and Frank [57] report how storage structures for heterogeneously distributed spatial data can be improved so that access on less frequent objects performs well.

2.5.2.1.7 Query processing When a user has to retrieve some data out of the database, they first of all have to define the criteria that the required data must conform to. This usually initiates by the user specifying the criteria using their own native language, as e.g. English. Such an example sentence may be: "retrieve all administrative areas that are populated by more than 40000 people and are within a distance of 100 km around the capital of the country". This way of specifying criteria, is only meaningful to the user and not to the computer. Some sort of translation has to take place, so that the application that handles the data can process such criteria. This requires the formation and specification of a *query language*. Such a language must hold artifacts that will be both comprehensible by the user, without them having to know the query language in detail, or to necessarily have to look up a command so as to be aware of its functionality, but also readily processable by the database, in terms of formalized and strict definition. This, obviously, will result in a query language that is much less rigid than one used by humans. Again, some kind of mapping is necessary, based on what the user may retrieve from the database and how. As it is impossible to use a human spoken language as a query language, taking into account its idiomacies (e.g. English), it is inevitable that some kind of less complex and more plain language has to be specified. Additionally, this query language must support such features as spatial, aspatial and temporal searching, either in isolation or in combination. In other words, the query language is the mediator between the user and the database. Any query language is composed of queries. A *query* is a declarative and explicit user definition of what is needed to be accessed within a database and for what purpose. When queries are executed within a database, a subset of the whole dataset is produced, which conforms to

the specifications set within the query. In the case of OO databases, this data subset will be a set of objects. Kim [88] defines as *query graph* the subset of the overall class hierarchy which represents the results of a query: the selection operation on a class C retrieves instances of the class C which satisfies a Boolean combination of predicates on a sub-graph of the schema graph for C. Predicates can be distinguished by the attribute level that they seek: *simple predicates*, when the query retrieves instances and is based on the values of the attributes of these instances which are not other objects, or *complex predicates*, when the query is based on attribute values found in one or more aggregate object of the object that is being queried.

Cyclic branches [88] are branches of the class schema if it contains a class C_i and a class C_j, such that C_j is the (indirect) domain of an attribute of C_i, and C_i is the domain of an attribute of C_j; or C_j is the (indirect) domain of an attribute of C_i, and a super-class of a subclass of C_i is the domain of an attribute of C_j.

The standard query language in relational systems is the Structured Query Language (SQL). Some OO environments extent the SQL language with object-oriented constructs, called Object SQL (OSQL). In general, SQL is a convenient and popular language to use and extend, and users may become familiar fairly quickly, if they are not already. However, the additional constructs that are introduced pose questions regarding the standardization issue of an object-oriented SQL.

Language constructs should also be created and used to support querying of versioned objects, such as to retrieve objects for version creation and then version storage, as well as to retrieve predecessors and successors of a version.

Functions found in a query language that is to be used within a GIS, must be classified before any actual specification and implementation takes place. Choi and Luk [35] use the following categories of functions:

- Semantic spatial functions: the concept denotes a set of spatial operations that are invoked to manipulate spatial data and produce information. The

actual representation and implementation of these functions are both hidden from the user, therefore enforcing the concept of encapsulation. Examples of such functions are: adjacent, area, complement, contains, intersects, length, route and surrounds.

- Retrieval functions, used to retrieve properties or attributes.
- Set-oriented functions, like count, union, and common.
- Aggregate functions like minimum, maximum, sum, average and percentage. These functions initially retrieve data from the aggregate parts of an object. Therefore, the calculated values depend on the values found in the aggregate parts.

Custom, user defined functions may be utilized replacing a query when it is very complex (to reduce the degree of complexity) or one that is used very often (to reduce the amount of commands that must be invoked). Choi and Luk [35] use the concept of *superfunction* to denote a function which is composed of other, kernel based functions. A superfunction is defined using the query language. The superfunction construct requires an additional compiler to interpret the defined functions. Regardless of the overhead involved, it is considered to be one of the most suitable techniques (if not the best) in order to build custom, user-based applications on top of a database kernel.

Geometrical functions found in a query may be classified as Choi and Luk [35] suggests:

- "overlap", or "intersect", which are used to compute the geometrical commonality between objects;
- "containment", which calculate enclosure information;
- "component" or "aggregate", which retrieve the components of an aggregate geometrical object;
- geometric object computations, which create new geometrical objects;

- arithmetic computations on attributes, which compute derived information out of the object's geometry;

Usage of a query language within the GIS, requires the existence of a query processor module which parses the user defined queries and outputs information to the user. Query languages are supposed to benefit from the OO paradigm by possibly incorporating frequently used queries within the class specifications. Moreover, OO provides a user interface with operations on objects [57].

2.5.2.1.8 Schema Modifications In multi-user environments where the same persistent object space is accessed by a number of different users, it is sometimes necessary to modify the schema of the class hierarchy, as it might evolve through time. In this object space instances already exist, but the system must not be made unavailable due to the required modifications. This is more important for experimental purposes during the development phases of the database. Kim [88] identifies two ways that a schema may be modified: *single schema modification* is the direct modification of a single schema, without keeping track of the changes, and *schema versioning* when a single schema is versioned and therefore the changes are stored. The two ways of schema modification can co-exist, since a specific version can be modified without having to create a new version of the schema. Kim [88] classifies the changes that a schema may undergo, and distinguishes between two main types: class structure or behaviour changes and class hierarchy changes. He also separates changes in terms of required updates to the instances: *soft changes* do not require instance updates, while hard changes require updates to existing instances. In general, the impact of schema changes is not only on the class itself, but also on its sub-classes as well as on the instances that have already been produced and exist in the database. Nonetheless, changes incur updates to the instances in the databases, which may or may not be processed right after the schema changes take place.

Schema modifications may have a complex impact on both the class hierarchy as well as on the instances. It is therefore useful to define a set of rules and

principles that must be followed and preserved whenever a change in the schema is necessary. An example of these rules and principles is the ORION database [88]. In this paradigm, principles are called *invariants* and four types may be identified:

- Class-Hierarchy invariant, which defines the class hierarchy as a directed acyclic graph (DAG).
- Name Invariant by which names of classes, methods and attributes are unique throughout the class hierarchy.
- Origin Invariant by which all class members have distinct origin.
- Full-Inheritance Invariant, by which members are inherited to sub-classes, unless this violates the name and origin invariants.

Rules for single schema modification can be based upon the above invariants, therefore providing control for the impact that these will have on the schema itself.

Schema modifications are very often involved in an OO DBMS. There is usually no particular problem when temporality is not supported. Usually, the schema is stored differently than the objects themselves, even in the case where classes are treated as objects. However, in some cases, comparisons might become inexpressible and impossible to implement [7]. Dynamic schema support is essential in any OO DBMS kernel.

2.5.2.1.9 Security and authorization In multi-user database environments, where the object space is accessed by many users, simultaneously, the issue of data security and authorized access arises, that is what can be accessed and modified by which user. It is obvious that every user not only has different requirements when it comes to access the database, but not all data should be accessible or modified by all users.

In the case of relational databases, there exists an authorization model, through which a user may be granted access to invoke a query on the data which may or

may not alter the values stored. In OO databases, this model exists in a somehow different implementation.

2.5.2.2 UML in database modelling.

As mentioned in the beginning of this section, the major characteristic of DBMS is that they provide persistence for objects. The UML is well-suited to support persistence in the notational part [22], as one can adorn a class as persistent. Moreover, the UML is appropriate to model not only logical database schemas but physical databases as well, as the UML's class diagrams are a superset of entity-relationship diagrams (E-R), which is a common modelling tool for database design, mostly for the static part. Additionally, UML models the behavioural part of the schema, as it employs an OO methodology.

2.5.2.3 Usability of OO databases in GIS

Prototypes that have been implemented in OO databases include commercial products such as Smallworld GIS [85, 124], GeO_2 [43, 96], GeoStore [11]. Relevant research work can be found in Choi and Luk [35], Egenhofer and Frank [53], Egenhofer and Frank [57], Frank and Egenhofer [64], Milne *et al.* [103], Oxborrow and Kemp [113], Worboys *et al.* [139], Worboys *et al.* [140], Worboys *et al.* [141].

The lack of a theory regarding geographical databases, is considered to be a serious obstacle for research in this field. Formalization, if provided, may be used in many levels across the system development, such as the underlying data model. Efforts to provide such formal theories can be found in the work of Worboys [131], Egenhofer *et al.* [55], Egenhofer [56], Frank [65], Pullar and Egenhofer [117], Yang *et al.* [142].

An object-oriented data model can alternatively be implemented using a relational DBMS [19, 140]. It has been shown that many of the constructs of the OO data model can be mapped to the to the relational model although losing some of their natural meaning which is useful to the user. The efficiency of the process

is also decreased by using a relational database, since object oriented databases, unlike with relational ones, eliminate the need for frequent joins of tables when query execution is involved.

It has been shown [126] that SQL by itself has shortcomings and cannot be used in its current form in the context of an object-oriented database. Therefore, the need for a standard object oriented query language is imposed that can be used to model *spatio-temporal queries.* The major parts of a such a query sentence would be : who, what, where, when, why? Respectively, these keywords respond to parts of a physical language sentence as Subject, Action, Object, Place, Time, and Reason. Questions posed here are:

- Which ones are the fundamental to form a basic query?
- What if a part is omitted? Then assumptions for the missing parts are necessary.
- What if multiple parts exist?

Most commonly, object oriented databases offer a custom or standard object query language as part of their core functionality for the user to interact with the data.

2.6 Conclusions

The object orientated mechanisms of abstraction and inheritance are necessary to model the complex situations, such as geometric objects, which can change over a period of time. Moreover, OO programming languages will be necessary to implement an OO GIS design, which is more flexible and better-suited to describe complex data structures. By using OO databases as well, the required property of persistence is being best implemented since the architecture of a GIS will become clearer, and the maintenance will be easier and the life cycle longer. Moreover the physical location of data are not a user's or programmer's concern,

since a unified set of commands are used to retrieve them. Performance problems encountered by using process time demanding OO database management systems are considered to be eliminated by using powerful (but not necessarily expensive) hardware.

Chapter 3

Analysis

3.1 Introduction

Chapters one and two elaborate on the major issues encountered when spatio-temporal information is being modelled following an object oriented methodology. Some of these issues have already been addressed through proposals found in the literature, however some remain still under research. Both previous chapters aimed to provide input for the current, which eventually narrows down the problems, adopting or providing a solution whenever necessary, and using conventions, which are considered to be of major importance.

As it has been discussed in chapter two, the initial process that leads to development is first requirements capture and later analysis. It has been already stressed (chapter two) that **analysis**, as a conceptual process, is considered to be a prerequisite to synthesis. Hence, the material presented here is considered to be the first of four stages that result in the development of any software product, in this case the MPOOST (**m**ulti-**p**urpose **o**bject-**o**riented **s**patio-**t**emporal) model. It focuses on the identification of spatio-temporal entities in the real world and attempts to analyze their structure and behaviour. It must be noted that the part of the chapter that deals with the real world domain is to a great degree independent of any computerized context, since it is (and should be) totally independent

of the stages of design and implementation that follow. However, as a whole, analysis is considered to be part of a broader development process, hence there are references to the stages of design and implementation whenever necessary.

3.2 Organization of terminology

It is imperative that from the early stage of analysis, a set of basic terms is specified and used for consistency and comprehensibility reasons. This is called the *analysis domain vocabulary.* The main source for the concepts and terms of this vocabulary is the real world and the phenomena and objects that are being modelled by various applications that employ some kind of view. Later, this vocabulary may expand, as concepts are analyzed and new terms are being introduced, despite the fact that it is considered to be a finite set of terms for the purposes of this research. However, in real world projects, practice has shown that future expansions will occur, and the whole process of analysis-design-implementation-testing should undergo as many iterations as possible in order for the developed product to reflect the additions and/or modifications that have occurred. When exactly an iteration should be invoked is a decision primarily depending on the practical needs that the final product serves, as well as on the magnitude of modifications themselves. It must be noted that there are no theoretical strict rules to be followed for both issues, but it is a collaborative decision based primarily on empiricism.

Since a computerized environment is employed, in order that the MPOOST data model is implemented, a *system domain vocabulary* must also be established. It is anticipated that all terms found in the analysis domain vocabulary will have an associated term in the system domain vocabulary. We shall call this one-to-one association of terms as the *analysis to system mapping.*

3.3 Basic requirements

One of the basic requirements that the MPOOST model must satisfy regards its effective use by multiple applications (hence the adjective multi-purpose). But, what happens when multiple application domains are involved in the analysis? In this case, every application holds its own vocabulary domain with a set of terms, *and* a conceptual model. It is considered that applications are not independent of each other, as they share concepts and information, and possibly conceptual models. Specifically, in the context of a GIS, one of the default components that applications share is the core spatial conceptual model [1], which is separated into geometry and topology, as already mentioned in chapter one. This, of course, is not addressed and used in the same manner by all applications, but the way that geometry and topology are treated differs significantly from two-dimensional to three-dimensional geometry, from vector to raster models, and from spaghetti to node-link-arc topology. We shall call the spatial conceptual model that an application employs as a *spatial representation* [25]. In order that the MPOOST data model can stand up to its name, it is inevitable that it must support heterogeneous spatial representations. Regarding the support for any non-spatial data and models that applications may involve, it is considered that it will be achievable by the underlying computerized conceptual model, namely the object-oriented model, as has already been discussed in chapter two. Hence, it will not be of major concern in this stage of analysis. It must be noted that the comparison of different spatial representations is beyond the scope of this thesis[2]. In brief, spatial representations can be of four major types [25]:

1. vector;
2. raster;
3. hybrid; and

[1] In some cases it could be the spatio-temporal model, but not all application models consider time as an inherent part of the information

[2] Research work regarding comparison of spatial data models is from Peuquet [116], Breunig [25]

4. analytical or parametric.

There is a further categorization of spatial representations in each of the above categories. All of these representations use spatial objects as fundamental building blocks, which may be uniquely identified. Complex blocks and their internal structure and external relationships with other blocks are derived and stored. Any of these blocks is considered to be an independent chunk of data which may hold references to other blocks. All of these features are an inherent part of the object oriented modelling methodology. Therefore, it is considered that the realization of different spatial representations can coexist in an object oriented model. Hence, the requirement that model implementation should be spatially multipurpose is served by its object-oriented realization. However, at the higher conceptual level (spatial data model), this is achievable by using *extended complexes* (*e-complexes*) [25]. In brief, objects embedded in the discretized geometric space have geometry, topology and metrics. Geometry can be described by means of the e-complexes, topology is described by the internal and external relationships of the e-complexes and metrics as operations which may or may not be part of the e-complexes. One of the inherent key features of e-complexes is that they support the realization of heterogeneous spatial representations. The implementation of such a spatial data model is extremely straightforward using the object-oriented approach.

3.4 Requirements and terminology in the real world domain

Objects that exist in real world are called *real world entities.* If this entity involves or references some kind of geometry that corresponds to the real world, then this entity is called a *real world geo-entity.* Similar geo-entities can be grouped in *categories* (explained further below). The four main components of a geo-entity category are:

1. its *geometry*, which is modelled by temporal e-complexes as discussed later in section 3.8.1;
2. a set of *aspatial attributes*;
3. a set of *cartographic representation entities*; and
4. a set of *behavioural interfaces.*

Each one of the above aspects is discussed in more detail in the following sections of this chapter.

Both types of entities (real world or system) can have a number of *properties* associated as well as a set of well defined *actions* as part of their *behaviour*, which can be either *passive* or *active*, with regard to whether the subject of the action is the container geo-entity itself or another geo-entity, respectively. Real world entities that are similar in structure and/or behaviour can be grouped together into *categories.* Every real world entity has a unique *identity* which does not depend on the internal structure and behaviour of the entity. Real world entities may be *created*, *altered* in their structure *destroyed*, and *reincarnated* as a result of a real world event. A *real-world event* is anything that occurs in the real world, has a cause and an effect, may involve a set of entities which in turn may change in some way. Any real world entity may have associated a number of other real world entities (aggregate entity). At any moment, when the real world entity is not being altered, and for the whole time duration that the entity remains unaltered, we may say that the entity has a specific *status*. If something occurs that affects the structure of the entity in some way, then the entity transcends to a different status. Therefore the status of a real-world entity is strongly related to the snapshot in time of its internal structure. When a real world entity alters significantly because of a real world event, its status may alter significantly as a result, so that a new entity may be produced altogether, which has a different identity than the old one. The new entity may refer to its previous as the *predecessor entity.* If the entity's identity remains the same, and no new entity is created, then the entity has a series of statuses related, in an ordered

and linear fashion. We may call this the *status set.*

Both real world entity and event categories are quite general, and may contain a set of existing real world entities or events, in a similar manner that an object can be instantiated from an object class, in object-orientation (see chapter two, section 2.3.1).

Observations in the real world domain are always made in the context of a specific science or application, such as physical geography, topography, geology *etc.*. Every single such domain, has a different viewpoint, both quantitative and qualitative, as to how the real world is perceived, observed and modelled. This application context is called the *application domain.* Every single application domain involves a model (see chapter one section 1.4.1 for the definition of a model), called the *application domain model* [3]. In this way, the set of definable application domains is a subset of the real world domain.

As real world entities are viewed and modelled in the context of a scientific field or an engineering application, some kind of mathematical or empirical reasoning is employed that involves a set of constraints and rules to which entities and their parts must conform, so that the data integrity is guaranteed. We will call these *application integrity constraints*, and it is obvious that they have meaning only in the context of an application in the presence of geo-entities.

3.5 Terminology and requirements in the system domain

Real world entities (regardless of whether they may be geo-entities), when modelled and stored in a computerized environment system may be called *system entities* [4]. In this way, every real world entity should have a corresponding sys-

[3] also known as the conceptual application model

[4] This is similar but not identical to the concept of object found in object-orientation which is instantiated from a class, since a system entity may or may not be an object

tem entity used for modelling. System entities however, may not reflect directly back to real world entities, especially when purely software artifacts are used as system entities. Hence, the relationship between a real world entity and a system entity is *not* reflective. Nonetheless, system entities are being modelled as objects [5] in the database, therefore having both attribute values and associated behaviour. System entities when modelled as objects can be grouped into *classes* and have a unique *object identifier* (OID) that can be used when other system entities refer to them. System entities may be *instantiated*, *modified*, *deleted* and *restored*. A system *class* is used as a specification that holds information on what an object can contain as actual values. Objects produced by system classes are called *system instances*. A *system event* is anything that occurs within the context of the system, has a cause which has only meaning within the system and it ceases to be meaningful out-with, has an effect only within the system and may involve a set of system objects that are being affected. System events may change the state of these system objects, and as a result a series of new system objects may be produced. It is considered that different application domains that employ specific application domain models may use different system domain models. However, herein, all possible definable application models map vertically to the same system model, namely the object model.

3.6 Terminology and requirements in the map domain

Real world entities with an inherent geometric nature or with a relation to a geometry, map to system entities in the system domain. There has to be some way of visualizing the system objects produced from system entities, using specific methodology, which is none other than the *cartographic visualization* process. The output of such a process is considered to be the *map composition*, either on

[5]The basic concept of object-orientation, see section 2.3

Table 3.1: Domains and concepts

the computer screen or in a hardcopy format. System entities that appear on a map composition, are called *map features* when referred within the context of the map [6].

Real World Domain	System Domain	Map Domain
Application model	System (object) model	Cartographic model
Real world entity	System entity	Map feature
Property	Attribute	Visual attribute
Action/behaviour	Method/behaviour	-
Category	Class	Layer
Identity	Identifier	Identifier
Create	Instantiate	Project
Alter	Modify	Edit
Destroy	Delete	Erase
Reincarnate	Restore	Reinsert
Status	State	Version
Real world event	System event	-
Real world time	System time	-

3.7 Data input

A GIS as it is defined in chapter one (see page 28) amongst others, involves procedures to collect data that will be the input to the system. This is always done in the context of specific requirements for which the data will be used. Regardless of the data collection techniques involved, the output from a data collection phase will be called a *dataset*, which contains data that refer to the

[6] This term is not the same to the concept of a feature which denotes a characteristic of the map itself e.g. scale, size, legend etc.

same data collection session. As is obvious, a dataset comprises various types of data, both spatial and aspatial. The most important part of a dataset is the metadata information, that describes the content of the dataset, in the way that is explained later on in this chapter. An *input* dataset is the one that is used by the system to create information. It may contain information only for real world entities, for both real world entities/map features or only map features, always according to the data collection technique involved. It must be noted that real world entities will be transformed into system entities once the dataset is processed by the system. An *output* dataset is the one that contains information created by the system, and may contain system entities, map entities or both.

3.8 Spatial information

3.8.1 Geometry

As mentioned in the introductory section about fundamental requirements (chapter one), the data model should be able to support heterogeneous spatial representations.

Discretization of the space is based on first-degree polynomial approximations to 3-, 2- and 1- dimensional extents [135]. The simplicity of the polynomial degree contributes towards efficient and fast algorithms whenever calculations are involved in a computerized environment. However, higher degree polynomials may be used, as in the case of smoothed contour lines, or in the case of spatial regression analysis.

Regarding the geometry of primitive geometric objects, these are considered not to cross themselves. In this way, non-aggregate 1-extent objects cannot contain the same point more than once, non-aggregate 2-extent objects cannot contain the same polyline more than once, and non-aggregate 3-extent objects (solids) cannot contain the same surface more than once. This is done according to the principle of *maximum decomposition.* A complex information system, in

order to simplify its structure, has to be first analyzed and decomposed into the smallest possible units. This unit is strongly related to any resolution that the system will be capable of supporting, not only in the geometry but also in its structure. Advantages are that these units can serve as the basis upon which more complex aggregate objects can be build. Any operations on objects will eventually result in fundamental operations found in the units. Moreover, duplication of information is avoided since any complex object made of points can simply reference the specific point-object. Hence, structures that share the same points (with disregard to the rest of the structure) can be quite easily traced, e.g. within a user query. Thus, such an object structuring results in manipulation of references among objects. Disadvantages are that a large number of references must be maintained. In the case of spatial information and since the embedding space is discretized, these units are considered to be the class of 0-extents and more specifically the sub-class of points with co-ordinates x,y,z.

The fundamental spatial building blocks are the *e-complexes.* An e-complex is defined by means of a simplicial complex (see chapter one, 1.1), its geometry and its topology as (Breunig [25]):

Definition 3.1 (e-complex) *a triplet (C, T, G), where C is a simplicial complex of dimension d, T is the set of the d-simplices of C (for $d > 1$:with neighborhood), T is called the d-dimensional topology of C, and G is the Euclidean geometry of C (the set of the Euclidean coordinates of the 0-simplices of C)*

The geometry of e-complexes is considered to be embedded in the Euclidean $\mathcal{R}_3$ space.

Since spatial indexing of geometric entities will be involved, then bounding rectangles in all planes (xy,xz,yz) must also be computed. This will enable both two- as well as three-dimensional indexing and enhance query processing.

3.8.2 Coordinate systems

It is mentioned already that real world entities are assumed to be embedded in the Euclidean three-dimensional space. The geometry of real world entities is recorded using coordinates. For this to be feasible, a coordinate system must be defined. Coordinate systems may be either *geocentric* using *Cartesian coordinates* (x, y or x, y, z) or *geographical* that uses *geographical coordinates* (φ, λ, h). Both kinds of systems employ a number of variables that make possible their definition. Such variables are:

1. The reference surface, e.g. spheroid or ellipsoid of revolution.
2. The geometry of the reference surface (e.g. radius of sphere, axes length of ellipsoid).
3. The orientation of the reference surface.
4. The location of the reference surface.

In the case of geocentric Cartesian coordinates, a *geodetic datum* must be also defined.

It must be noted that reference systems employ a lot of variations, but not all of them are discussed herein.

Map features as depictions of real world entities, are considered to be embedded in a two-dimensional space. A *map projection* is a mathematical transformation that relates a point on the surface of the earth to a point on a developable surface via an ellipsoidal model of the earth. It is used to transform the 3D real world entities to 2D map features.

Map projections, according to the geometrical shape that they employ may be grouped in:

1. Cylindrical
2. Conical
3. Azimuthal

4. Other

According to the type of distortion they cause on the metrics of the real world entities they depict, they can be grouped in:

1. Equidistant when specific distances are being preserved.
2. Equal area, when areal properties are being preserved.
3. Conformal, when specific directions are being preserved.

Hence, every projection has to be defined as a combination of the above two categories. Moreover, mathematical formulae that transform the coordinates from the surface of the earth on the projection plane and *vice versa* must also be provided. These formulae operate upon the coordinates in the geometry of a geo-entity, and may affect at least one or more of the metrics properties (such as distance, azimuth, area, volume). Topology is not always affected by the projection of the geometry. This depends on the type of surface that the projection is using and on the extent of the earth surface that is being projected on the plane. Hence, whether the topology is required to be recomputed every time that the geo-entity is projected should be known *a priori*. For example, some of the azimuthal projections can only display half of the earth's surface, hence geo-entities that belong to the other half will not be shown, and if their references found in map features on the map are maintained then these will not be valid. Interrupted projections (such as the interrupted parabolic or the interrupted eumorphic [114]) are also well known not to preserve the topology of geo-entities. We shall call the map projections that preserve the topological relationships of the geo-entities as *homeomorphic* and those that do not as *non-homeomorphic*. The model should be aware of the whether topological relationships are valid after a non-homeomorphic map projection.

3.9 Spatial topology

Spatial topology is considered one of the three components (T) of an e-complex. Topology is divided into *internal* and *external*. Internal topology refers to the internal structuring of geometrical entities, namely the simplicial complexes, while external refers to the connectedness of an entity to its neighbors. If topology information for a specific geo-entity in a dataset is not present, then it can be derived from its geometry always in combination with the geometry of the geo-entities of the same dataset. The reverse, however, does not apply, since geometry cannot be constructed from topology. Topology, either internal or external usually refers to real world entities of the same real world time. It is not incorrect to correlate entities that have different real world timestamps, only if this is in the context of snapshot overlaying, i.e. to combine (so as to possibly compare) entities of different timestamps.

Not all topological relations among geo-entities should be computed *a priori*, since this would result in extremely large datasets, that would make query execution impractical. Moreover, not all geo-entity categories are relating to each other topologically. Depending on the query itself, topological relationships may be calculated on demand. This obviously imposes a serious delay in getting the result of the information, which mainly depends on the speed of the computer hardware involved, but it can be tackled by using proper topology optimization algorithms.

It must be noted that external topological relationships are stored in the geometrical component of a geo-entity, namely the e-complex, so that they remain independent of the application domain that the geo-entity belongs to. In this way the same geometry along with its external topological relationships can be used by different geo-entities. This can not be feasible in every case though, especially when the geometry and its wrapper geo-entity are strongly coupled together.

3.10 Aspatial data related to geometry

A plethora of aspatial information may be "attached" to a geo-entity, always according to the identifiable properties it holds. Data quality classes are considered to be the major aspatial component of a geo-entity, and are defined within an individual package. If this component is design-independent, it may be related to any object-oriented model rather easily, thus enforcing the multi-purpose design of the model. The latter can fit various other object-oriented spatial data models. For this to be achieved, some of the aspects of data quality which may be found as built-in OO model functionality are separated from the inherent mechanisms of the object-oriented model (such as lineage which should be linked to the event/state transition mechanism). This only results into additional design required, and avoiding using existing model features.

3.10.1 Attributes

An attribute can be defined as a fact about some location, set of locations or feature on the surface of the earth [68], and it usually a result of a measurement, of an interpretation or the outcome of historical/political consensus (e.g. place names). Attributes are part of any entity (real world or system) and they contain *values*.

Formally, an attribute A is defined explicitly by eight parameters: A (name n, unit u, static/non static, constant/variable, source/derived, function f, type Y, domain D, timestamp TS, constraints C).

The *name* of an attribute serves as the unique identifier within the entity that the attribute belongs to. The *unit* of an attribute is the name of the measurement unit used (*e.g.* "meter", "degrees", "pH", "pound", *etc.*). When an attribute value is the same for all entities, the attribute may be *static*, otherwise it is called *non static*. Common static attributes are the counters. Usually an attribute may receive any value, and then it is called *variable* attribute. If this value does not

change then the attribute is a *constant* attribute.

Attributes may contain values that originate from an input dataset as direct observations, called *original* attributes, or attributes that are calculated by some well-defined operation of the system from other attribute values, called *derived* attributes. Derived attributes are accompanied by the *function* that calculates the values.

Every attribute has a *type* Y, as well as a *domain* D from where its values are taken from. Regarding the numericity of the value types, attributes may be either *quantitative*, when numbers are used or *qualitative*, when non-numeric values are used. Attribute types can be:

1. ordinal, when values have a predefined ordering, usually the one found in the domain;
2. nominal, which may a name from any domain, which does not necessarily imply any relative order or priority among values;
3. ratio, when the zero value means a zero quantity; and
4. interval scales, when quantifying differences between particular values.

A domain is a real world or system entity. The fundamental entities (usually found as predefined by the system) that serve as attribute domains are:

1. Integer;
2. Real;
3. String; and
4. Boolean

All other attribute domains can be constructed using a specialization or a combination of the above. An attribute may be *timestamped*, therefore a bi-temporal interval TS will be associated.

Any attribute may have a set of associated *constraints* that will serve as a restriction to receive values from a specific subset of a domain. Constraints are given in the form of an equation (or most frequently an inequation) between the attribute value and acceptable values, either static or referring to other attributes.

3.10.2 Data quality

Brassel *et al.* [24] when discussing data quality emphasize the role it plays. The definition of data quality that they propose is based on the definition of the ISO Standard 8402:

> The totality of features and characteristics of a data set that bear on its ability to satisfy a stated set of requirements.

This definition emphasizes the very close connection of the quality of a dataset to the scope of its uses. If there are no requirements then no characterization can or should be given as to whether a dataset is of good or poor quality. And the quality of the same dataset can vary significantly when different requirements are to be met. Moreover, in the context of a multipurpose model, extended data quality information is crucial, and if missing, then the assessment of the data for particular uses by different applications is not feasible. It is apparent that the more there are applications accessing the dataset then the more detailed and extended the data quality information should be. Hence, a dataset may prove to have a different kind of quality, by applications that set their requirements *a posteriori*. This type of assessed quality is called *fitness for use* and Brassel *et al.* [24] give the following definition:

> The totality of features and characteristics of a data set that bear on its ability to satisfy a set of requirements given by the application.

Hence, while data quality is an input to the system, fitness for use is something which is assessed by the system, based on the data quality information and the set of a specific application requirements. This assessment may be done with or

without the aid of human interpretation. This human intervention may prove to be minimal, if data quality and requirements information are both structured properly, so that a deterministic model can be built. Therefore, a well defined set of procedures that match and assess the two inputs must also exist in the system.

In this manner, data quality information should be stored with the metadata part of the dataset. Requirements that were defined prior to data collection (and possibly re-processing) might be relevant for storing along with the metadata. However, in the context of the MPOOST model this is not considered necessary, since such requirements should be part of the application domain that will possibly access the data in the future. These requirements posed by different applications after the collection of the data will help to assess the degree of "fitness for use" against the data quality information stored within the dataset.

The major components of metadata are explained in detail in the next sections.

3.10.3 Positional accuracy

As has been discussed earlier in this chapter, real world entities that employ some kind of geometry are composed of points. Hence, positional accuracy in the MPOOST model is based on the positional accuracy of points. This is achievable by storing the standard deviation for every point in the database. Positional accuracy of higher dimension entities is derived from the standard deviation of the points of which they are composed, either directly or indirectly. It must be noted, that standard deviation remains unaltered only for system points that originate from an input dataset. If any kind of processing is involved, then a derived point is produced and the standard deviation should be calculated. The technique to be used for calculating the standard deviation propagation is that of variance propagation [48].

3.10.4 Attribute accuracy

Attributes may have an associated standard deviation or certainty statistic which is input to the system along with the attribute value. If this standard deviation or certainty statistic is unknown then the value is set to null.

Attributes that are derived from other attributes through a process, have their accuracy calculated as well. For this purpose, variance propagation is used to calculate the standard deviation for attributes such as length, area and volume, and set theory is used to calculate certainty statistics for attributes such as species, land use, and agricultural suitability.

3.10.5 Completeness

We shall use the concept of completeness, as discussed in chapter 1 (see section 1.7.2.5, page 76). It describes whether the objects within a dataset represent all entities of the "entities universe". It is noteworthy to mention that this "entities universe" may be relevant to a specific application, since a data collection phase is always done in this context. In order that this definition be functional, the application domain's "entities universe" must be well defined. Currently there are no formal way to define such a concept. In the case of the MPOOST model, this is a set of either tangible or conceptual real world entities that an application domain expert can identify. This facilitates the one-to-one comparison between the contents of the "entities universe", which are all possible observable or conceivable entities and the contents of the dataset. Since a) multiple views of the same application may exist, and b) there is no consensus on standardizations of "entities universes", the data model that is to be built (namely the MPOOST model) must provide the facility to map similar entities, relationships, attributes and behaviour of different viewpoints of the "entities universe" to common ones. For this to be effective, the MPOOST data model must, at its best, either include an expandable, predefined set of application domain entities universes, or at least provide the capability for such mappings with the existing system objects found

in the database. In this manner, two aspects of completeness assessment are feasible: whether each entity exists in a dataset or whether an entity is represented adequately within a dataset. The completeness of a dataset is anticipated to decrease as a result of the data age. If this rate of change in the real world is somehow known, another desired model capability would be the calculation of estimated completeness, expressed as the percentage of existing real world entities that correspond to existing system entities. Finally, for the MPOOST model itself to be complete [24] and flexible, it should provide the functionality to incorporate different real world representations, both spatial and aspatial. Modelling completeness information is considered to be of crucial importance, should the MPOOST model serve as a multipurpose one (See section 3.14 at the end of this chapter).

3.10.6 Complete list of required metadata

The metadata (including data quality information) that are considered to be of use for the development of the MPOOST model are:

1. **Data summary**
   (a) Source, classes of data, areal coverage, date, scale. This is in descriptive form and it is used for a quick overview of the metadata contents.
2. **Lineage**
   (a) Agency of origin. This is a paired list of the names of the company/institution responsible for the data collection along with the type of method of data collection.
   (b) Method of data collection. This involves all data collection techniques used.
      i. Primary survey techniques. A list of all techniques used. This might be topographic surveys, photogrammetric acquisition, sattelite image classification *etc.*

ii. Secondary data sources

A. Digitizing method, e.g.using a vector digitizer or a raster to vector format translation.

(c) Source scale denominator. This is the denominator of the map scale the data were digitized.

(d) Type of source media. This is textual description of the medium that the source data set originate from.

(e) Date of source media. This is the date that the source media refer to.

(f) Dates updated. This is a set of dates with updates on the data contents along with the description of the update.

(g) Source contribution. This is a textual description of the type of possible contribution to the data set from the source.

(h) Processing history:

i. Coordinate transformation. This a paired list of timestamps along with the type of coordinate transformations involved.

ii. Data model transformations. This is a paired list of timestamps along with the transformation involved. (e.g. raster to vector)

iii. Attribute transformations. This is a paired list of timestamps with the type of attribute transformation involved.

iv. Data transformations. A paired list of timestamps with the function responsible for transforming data.

3. **Coordinate system**

(a) Type of coordinate system. This can be either Cartesian or geographic.

(b) Map projection name.

(c) Map projection parameters. This is the full list of all parameters used to define the projection.

(d) Unit of measurement for coordinates.

4. **Spatial data model**

   (a) Specification of primitive spatial objects. This is a paired list of the name of the primitive spatial objects used along with their description and explanation of their geometry.

   (b) Topological data stored. This is the description of the topological model used as well as the relationships among geometric objects.

5. **Feature coding system**

   (a) Definition of feature codes or classes. This is a list of codes (numeric/alphanumeric for codes or textual for classes) along with a textual description of what these codes represent.

   (b) Definition of the classification system. This is a series of tables, possibly for every attribute, that correlate the values stored within the dataset to the textual description which is comprehensible to the users.

6. **Completeness**

   (a) Model completeness. The degree of the inherent conceptual model functionality which will be used in the context of an application domain.

   (b) Data completeness. Similar to the model completeness but refers to the actual volume of data and the degree to which they correspond with the dataset.

   (c) Formal completeness. This specifies whether [24]: a) all mandatory meta-information is available, b) the format corresponds to the standard or data format used, respectively, and c) the data is syntactically correct.

   (d) Entity object and attribute completeness, which is the degree to which entities that should be part of the dataset actually exist. For this to be assessed then a set of requirements, a dataset and a set of real world entities are necessary. Attribute completeness measures the degree to which a system object is including attribute type and values according to specifications. For this to be assessed, a set of system objects is necessary. In brief, when comparing a dataset to the "entities universe",

a) a value of an attribute may be missing, b) an attribute type may be missing or be of different type or c) the object itself may be missing from the dataset.

(e) Documentation on the usage extent of the classification system. This is a detailed textual description on the usage of the employed classification system.

7. **Geographical coverage**

   (a) Overall extent. A set of coordinates of the polygon that encloses the area surveyed.

   (b) Detailed specification of coverage if not complete. This is an additional set of polygons, enclosed by the overall extent polygon, that show which areas have been left out of the survey and why.

8. **Positional accuracy**

   (a) Statistics on coordinate error.

   (b) Standard deviation of point features, referring to the dataset as a whole [7].

9. **Attribute accuracy**

   (a) Statistics on attribute error. This is a table that correlates the attribute fields in the dataset along with statistical information about this attribute.

10. **Topological accuracy**

    (a) Methods of topology validation employed. This is a textual description of what methodologies were followed to validate the topology stored within the dataset.

11. **Graphical representation**

[7] Standard deviation for individual points is stored along with their geometry. All other geometric features (e.g. 1- and 2-extent) base their positional accuracy on this quantity.

(a) Graphical symbolism for each feature class. This is a table of two columns which correlates the symbol used to depict any information to the feature class.

(b) Text fonts for annotation. This is a series of text font names used to depict textual information in the dataset. The actual metric file should not be included, due to the diversity of software and font formats involved.

3.10.7 Spatial resolution

The geometry of real world entities is recorded using either Cartesian or geographic coordinates. *Spatial resolution* refers to the smallest distinguishable distance over which it is possible to identify and record change in the geometry. As it is obvious, spatial resolution in the real world depends strongly on the instrumentation used to measure geometry, and it can be quite small. Spatial resolution is also strongly connected to the application domain under which the collection of geometric data took place.

3.11 Temporality

In a chapter one, a detailed discussion is presented on philosophical issues regarding time and how it is being philosophically perceived throughout history. One, however, cannot rest at simply discussing and bringing up questions, but give answers as well. This section is a more pragmatic approach of the issue of temporality and it is based purely on how it is being used in the context of geographical information. However, it must be said that the basic notion used to initiate the analysis on this component is *time based* and not change based, since it is the researcher's opinion that change is an effect of various self-existing events, that have an existence in time, and a result in space. Hence space is embedded in time, in a hierarchical way. This hierarchy denotes that the approach of space

within time is definitely closer to how the real world functions, rather than the timeless space paradigm which is infinitely static, and rather more theoretical than pragmatic. Moreover, the result of an event is considered not to have any effect in the order time, as time's existence is independent of changes in space. This applies for real world time, as the system time can be manually set and therefore events do have an effect on its order[8].

As already discussed, time is inherent in both the real world (real world time) and system (system time) entities. However, this is not an explicit feature, but done implicitly through *timestamping*, which is the process of attaching temporal information to any real world or system entities. The basic temporal construct used for time stamping is the *timepoint*, which is identical to the chronon concept (see chapter one, page 65). An ordered pair of timepoints may define a period in time, called a *simple time interval.* A time interval may or may not include the starting and/or ending timepoint, therefore being an *open* or *closed* time interval. If two or more disjoint simple time intervals are combined, then a *complex time interval* may be composed, which involves aggregation of its parts. Complex time intervals are equivalent to time periods, which may contain void intervals.

For any entity properties for which we record a change in values, we may associate a *bi-temporal interval* as a two dimensional time interval. This is the Cartesian product of the real time r and the system time s.

Timepoints may be of different resolution, called the *temporal resolution.* This refers to smallest distinguishable difference of interest between two measurable time values. It can be said that temporal resolution may be derived from the value that a timepoint contains. E.g. the fact that "the property A was registered on Wednesday 26th of May", implies that the temporal resolution involved is that of one day. It is also implied that the exact time within the day that the event occurred is not of concern. However, if the registration time is known, then the temporal resolution increases.

[8]This however does not imply that it happens in a functional system. It is only emphasized to clearly differentiate between the two types on the concept of time.

In the real world, different temporal resolutions result in different units of measurement. As in the previous example, the knowledge of the exact time that the event occurred, is based on two different units (day and hour). Temporal units of different resolution show a relationship which is of hierarchical, generalizational and aggregational nature at the same time. The time units used are:

- 1 year (consists of 12 months)
- 1 month (consists of 31, 30, 29 or 28 days)
- 1 week (consists of 7 days)
- 1 day (consists of 24 hours)
- 1 hour (consists of 60 minutes)
- 1 minute (consists of 60 seconds)
- 1 second (consists of 100 mseconds)
- 1 millisecond (basic smallest temporal unit)

It is considered that there is no meaning to time below the value of one millisecond and this is already too detailed for events that occur in the real world and are being modelled by a GIS. However, this resolution should be maintained for system time, where it does have meaning (e.g. processor time, transaction execution time *etc.*).

Obviously it is possible to replace a timepoint unit using one of greater detail, e.g. 24 hours instead of one day), or using one of less detail (one week instead of seven days). However, such a replacement should not always happen, since it results in either a loss of information or a useless increase of resolution. In the first example there is no increase in accuracy, although a larger arithmetic value is used in replacement. In the second example the value replaced might result in a timepoint being treated as an interval, where it should not. Hence, the time unit should be pre-defined and events should be timestamped accordingly and in a consistent manner. If both real world and system time is measured in

milliseconds, it can be treated as an integer number, where simple arithmetic operations apply. As real world time has one direction, most recent events have greater time values than older ones.

3.12 Representation

Every geo-entity that is modelled via a system entity and stored in the system, according to its geometry, may be visualized as part of a *map composition* using a set of *cartographic symbols*. The map composition herein is considered to be embedded in a two-dimensional geometric space, regardless of whether the composition appears on screen or on hardcopy, since both result in a two dimensional view [9]. Hence, cartographic symbols are of four main types:

1. Point
2. Linear
3. Areal
4. Raster

In all three cases listed above, the *graphic variables* in a cartographic symbol are (enriched from Jones [82]):

1. Hue
2. Saturation
3. Brightness (or Luminosity)
4. Size
5. Shape
6. Texture or pattern
7. Orientation

[9] this applies in the case of using VRML as a means of map composition

An additional, not quite a graphic one, variable the colormap, also known as Look Up Table (LUT), which applies only in the case of visualizing raster entities, is linked to the above list of graphic variables. It associates one of the graphic variables (or a combination of those) to the value stored in a raster dataset (cell value).

Not all cartographic symbols employ all graphic variables. Some graphic variables may have meaning to not only the symbols but to other variables as well (e.g. orientation, which may apply to the shape of a point symbol to the pattern of an areal symbol).

As it is quite obvious that the relationships among the graphic variables are quite complex, it would be quite difficult (if not impossible) to model using any other data structuring methodology apart from an object-oriented one.

A *map feature* is a cartographic representation of a geo-entity that employs a set of cartographic symbols. The property values found in the geo-entity are being mapped to the graphic variables in a manner specified by the user (or more properly the cartographer).

Geo-entities may have a set of cartographic symbol objects associated, always according to their geometry. The model should be able to assign values from other components of a geo-entity to the visual variables.

Regarding the display, cartographic symbols should know how to "display " on any media (hardcopy or screen), given specific values to the graphic variables that a symbol contains. Hence, we need to construct methods that perform on the geometry of every visual variable, and be a part of the symbol behaviour.

The classification for the cartographic schema proposed in this thesis is based upon the proposal of Worboys [131] for a generic model for planar geographical objects. All cartographic objects are assumed to be embedded in a two-dimensional space (a plane). Moreover, cartographic objects have a relationship with real world entities that have been modeled, which although it may seem an one-to-one relationship, however it is not. This is because any cartographic

object has a corresponding real world entity that it models. On the other hand, a real world entity may have more than one cartographic object associated with it, since factors that play a significant role here include the time that the real world entity existed, as well as any attributes upon which the cartographic object was rendered on the map (e.g. its color, pattern, line width etc.)

The notion of map layer which is commonly found and used in conventional manual and computerized map production methodologies is being replaced by the notion of *object class*, as it is defined in the OO paradigm: a single class may be instantiated to produce a set of similar objects, e.g. a road class may produce a set of road objects, therefore the class itself acts as a representation of a layer. However, not all classes can act as cartographic layers, since not all of them model geometric objects. In the case where a layer has to consist of objects from different classes then a "layer" class is created only for the purpose of associating all relevant objects from different classes into one group. E.g. A road class may produce road objects, a railway class may produce railway objects. These two sets of objects may belong to a `TransportationMapLayer` object (through an association relationship) instantiated from a transportation class.

As maps are considered to be two-dimensional views of the three-dimensional world, the Euclidean space that objects are embedded within is considered to be the two-dimensional (2D) [10].

3.12.1 Behaviour

Behaviour, as part of a geo-entity, is called the *real world behaviour*, as opposed to the *system behaviour* that a system entity shows. The main type of system behaviour is *persistence*, denoting that the system entity will reside in a permanent storage media. Behaviour is modelled by identifying specific actions or methods that an entity can perform, either to itself (passive) or in collaboration with other

[10]This is mainly because even when cartographic objects are considered to be three-dimensional, they are always projected onto a plane, where only two dimensions are involved.

entities.

A *method* is a form of action that an entity shows as part of its behaviour, and it employs a *name* as well as a set of *arguments* that have a *type*. An *interface* is a set of methods as part of a behaviour common to many entities, regardless of the category to which they belong. Interface is a convenient concept to group entity methods. It is important to mention that regarding the semantics of the method naming, the same method found in similar geo-entities may be addressed differently by different application domains. Hence, there is no universal set of application-independent elementary GIS functions, but each application has its own view into the semantics, that may be expressed as a web of relations [4]. It is considered that semantic variety in method naming is supported by the concept of polymorphism as an inherent feature of the object model.

Methods may be either *entity methods* which are part of the geo-entities as an inherent behaviour, or *utility methods* which are independent of the geo-entities and more task oriented. Albrecht [4] categorizes such tasks into fifteen functional groups. Utility methods are part of the system behaviour.

3.13 Enabling action in space: spatio-temporal entities

Entity attribute types that may change must be identified. These may be either spatial attributes (the geometry-topology-metrics triplet) or any other aspatial attributes. Spatial attributes, are modelled by the e-complexes, which are multi-dimensional structures. One could consider that time is the fourth dimension, beyond the three dimensions of space. This would enable the e-complexes to serve as spatio-temporal units. However, the current approach differentiates space from time, hence we consider that the dimensions of an e-complex are purely spatial and the concept of the *temporal e-complex* is introduced, as a temporal extension to the triplet found in its definition.

Since geo-entities involve complex geometries, e-complexes are considered to be a straightforward way of such geometry modelling. This complex geometry is usually changed by events, and it may involve specific and different changes that descend to its aggregate parts, i.e. the simplices of the simplicial complex that is one of the 4 parts of an e-complex. Therefore, the simplices should be timestamped, so that information on the event, and the real world and system time can be derived. It is considered that changes in an e-complex that affect its whole geometry/topology/metrics triplet are less likely to occur than individual changes. We introduce the concept of the temporal simplex and the temporal e-complex. The concept of spatio-temporal simplices and complexes have already been proposed by Worboys [136]. We use the same building blocks but the way that higher level entities are composed is different, since we keep the notion of time and space separate, by timestamping spatial entities. The concepts of simplices and complexes remain the same, referring to the geometry/topology/metrics trinity. More specifically:

1. A *temporal simplex* is a timestamped simplex. A bi-temporal attribute T is being added to its geometry.
2. A *temporal e-complex*, is defined as a timestamped e-complex. A two dimensional timestamp T and an ordered set of e-complexes (excluding the container e-complex) called its *life cycle*, and represent the different statuses that the temporal e-complex has undergone.

More formally:

Definition 3.2 (temporal simplex) *a temporal simplex (t-simplex) s of spatial dimension d is a triplet $s^d(G, BT, L)$, where G is the geometry, BT is the bi-temporal timestamp, and L is the set of d-simplices that are different* states *of the t-simplex.*

Definition 3.3 (temporal e-complex) *A temporal e-complex of spatial dimension d (te_d-complex) is defined as a fivefold entity $e_d(C, T, BT, G, L)$, with the*

simplicial complex (C), topology (T) and geometry (G) as defined by Breunig [25] in section 3.8.1 of this chapter, with the addition of the bi-temporal interval BT as the timestamp, and the life cycle *L, as an ordered set of e_d-complexes (of the same dimension d). The bi-temporal interval and a pair of timestamps called the* real-world *(r) and* system *(s) timestamps. This set of e_d-complexes describes the various forms of the complex during its life cycle, form its creation to its destruction and its possible reincarnation and so forth. Members of the L set are called* states. *Every state relates to a* previous *state and a* next *state.*

An important observation (that applies for both simplices and e-complexes) is that the e-complexes (simplices) that are part of the life cycle set L of a te_d-complex (simplex) with spatial dimension d, constitute an e-complex (simplex) of $(d+1)$ spatial dimensions. Hence, the life cycle set of a e_0-complex is a polyline (e_1-complex), the life cycle of a e_1-complex is a surface (e_2-complex) and the life cycle of an e_2-complex is a solid (e_3-complex). The life cycle of a te_d-complex may also be called the *spatial projection operator* [136]. A similar kind of projection, the *temporal projection operator* may be defined to be the bi-temporal interval which is the union of all the individual, possibly disjoint, real world and system temporal intervals of the e-complex or simplex.

Spatial and temporal information does not necessarily have to be combined, in order to model spatio-temporal data, since through aggregation we can temporally enable a geo-entity. However, it is considered that combining temporal and spatial structures into a single, we can achieve support for applications that require purely spatio-temporal information, which may not be related to a geo-entity in the real world. In this way, the requirement of supporting different spatio-temporal representations is achieved, on top of supporting different spatial representations.

3.13.1 Application domains

Every application, either scientific or engineering (as distinguished in chapter one) involves an *application domain* which is nothing more than the container space of concepts, knowledge, information and processes involved in the specific application.

In order that different application domains can be modelled using a single data model, two fundamental steps are prerequisite:

1. Identification of the application domains and the classes they should contain.
2. Modelling of the classes in such a manner that they are not contained in more than one domain.

The debate at this point is mostly about which classes should belong to which application domain. In some cases this may be straightforward to decide, however there are cases that categorizing a class into a specific application domain might be ambiguous. For the sake of simplicity, we will suggest a way of organizing application domains, always according to the object oriented methodology adopted.

3.13.1.1 Case study: the Hellenic Cadastre

The project for the Hellenic National Cadastral System that is still undergoing by the time this thesis was written, was initiated back in 1993. Amongst many of the new technical sub-projects that are part of this novel organizational strategy, is first the *base topographic mapping*, covering the whole of Greece and second *the two-way linkage between the land owners and their land rights to their ownerships.* Both projects produce an extremely voluminous amount of data that is to be manipulated in the context of a national land information system, which will act as a virtual container for all kinds of information involved. This system, in order to be not only effective but to conform to the Greek law as well, has to be developed according to the following six principles:

1. The principle of parcel-based cadastral information organization, which primarily requires the creation, maintenance and continuous update of the information.
2. The principle of validity check of the titles, before a request about a title interest can be stored in the system.
3. The principle of public faith, so that any person in good faith, with interest on a title can be protected.
4. The principle of temporal priority of submitted requests.
5. The principle of the publicity of records, that requires the cadastral records be open to the public.
6. The principle of the multipurpose cadastre, which requires that the system should be open to any kinds of relevant data and information (e.g. technical and legislative).

It is considered that principles 1, 4, 5 and 6 are to be satisfied and guaranteed by the functionality of the conceptual data model. Principles 2 and 3 are considered to be coupled with the internal organization of the cadastral institution, and therefore are not examined in this thesis as they are well beyond its scope. Regarding principle 1, it is not clear whether update should be parcel change-based or based on a predefined frequency. It is considered that both types of information update will be necessary, hence the temporal component of the model should anticipate this.

Additional to the six principles mentioned above, the information system that will provide computing support, must be networked and based on a distributed and interoperable technology, since data input, pre- and post-processing as well as storage of the output information will take place at the local cadastral offices, equipped with the various appropriate hardware and software. Regarding both hardware and software, it must be anticipated that since neither will be from standard vendors, software must conform to some kind of interoperability and

platform independence standard. The cadastral offices will be scattered across the country, with one office for every local self-administrative area.

The focus of this research is on the specific view of the geo-entities that are to be manipulated by a cadastral system. The outcome will be used to identify the degree to which requirements posed here are satisfied.

Definition 3.4 (National Cadastre) *The National Cadastre constitutes a system of parcel-based legislative, technical and other additional information regarding the ownerships for the whole of the State and must conform to the six principles mentioned.*

3.13.1.1.1 Entities in the cadastre A *cadastral parcel* may be defined as:

Definition 3.5 (cadastral parcel) *A uniquely numbered and geometrically defined spatial unit of land, within which, homogeneous and unique legal land interests may be recognized.*

The means by which a parcel is identified is through the *unique parcel identification number* (UPID). This number is subject to change whenever the geometry of the parcel changes, but it will remain the same regardless of any administrative boundaries change.

The *cadastral sheet* contains all the land transactions relevant to the parcel, the owner(s) and the parties that have any type of legal right on the parcel. All information on land parcels is held in the *cadastral book*, which is defined as the collection of cadastral sheets. Both cadastral sheets and books are assumed to exist in digital form.

The content of the cadastral records are the *legal rights* according to the legislation valid at the time the record is stored. The fundamental reference unit for the overall parcel-based cadastral information is the individual ownership unit, i.e. the parcel, as a general geometric entity on or under the surface of the earth,

which may be subject to legal rights, according to the legislation at the time the unit was created.

Regarding historical data, the system must be able to hold at least 7 years of parcel and parcel related data. This only gives an indication of the importance of maintaining historical data, but it cannot be directly translated into the required amount of historical data that must be kept, unless there is information on the average amount of changes that occur within the period of one year.

Legal interests on land parcel may be expressed either by an individual person or a legally formed group of persons.

The relationship among the three major aspects of the cadastre, may be summarized in the following sentence: "A legal person may have a series of legal rights to one or more land parcels" The above sentence signifies the tertiary nature of the relationship. Any legal right exists only if there is a legal person that can show an interest on a land parcel. Any of the above parts can be specialized to be more specific and aggregate to include more specific attributes.

Parcel-based information stored in the cadastre can only be altered via a *legal process.* The type and content of the process is defined by the legislation at the time that this process initiates. Possible changes that may occur are of six types:

1. Geometrical changes of the land parcel.
2. Aggregational changes.
3. Changes in the relationships between the legal persons and the land parcel.
4. Changes in the relationship between the types of legal rights that a legal person has on a land parcel.
5. Changes in the relationship between the land parcel and the legal rights that a legal person has.
6. Any combination of the above

3.14 Analysis of the required multi-purpose model characteristics

The aspects of multipurpose data model have been discussed in chapters one and two. It can be seen clearly now that the main requirement of the data model is that many and different applications will request to manipulate datasets, modelled under the same data model and stored in the same database. These applications in turn have their own specific set of requirements upon the data contents and the model itself. It might be the case that these datasets have overlapping contents, mainly regarding their geometry, which as a major component is of paramount interest to all applications.

A successful multipurpose data model is the one that provides the functionality to assess the degree of fitness for use of the data manipulated by the data model, against requirements that are posed by a specific application domain. When multiple application domains are involved, then multiple sets of requirements should be defined. But how do these requirements differ with regard to each application?

The main categories of differences in class structuring found in various applications that can be identified are:

1. with regard to the class schema, applications that do not have a pre-defined object-oriented schema may map existing geo-entities to predefined classes in the model with difficulty, because of:
    (a) lack of specific class names;
    (b) different types of classes;
    (c) different semantics of the same class, similar but not identical classes (different class structuring of the same real world entity);
    (d) different number and types of thematic (or aspatial) data attached to semantically identical classes;

(e) different number and types of behaviour attached to semantically identical classes;

(f) different types and number of relationships between the same pair of classes.

(g) different way of class generalization and specialization.

2. with regard to the data stored within objects:

   (a) different demands on accuracy, precision and resolution of the properties;

   (b) different dimensionality in the geometry (e.g. 2D/3D, different coordinate systems, different map projections or map projection parameters);

   (c) different ways of classifying aspatial data;

   (d) different demands on topology (e.g. no topology vs. specific topological relations vs. full topological model);

   (e) different types of algorithms for similar operations used to calculate derived attribute values.

All the above deficiencies and differences are viewed as a result of different designs. From a user's point of view, Bishr *et al.* [18] group the various types of heterogeneity into three major groups: *syntactic*, *semantic* and *schematic.*

Semantic heterogeneity regards the classes found in a single schema or between two (or more) class schemata. It is considered that this is an issue closely related to answering the question of the degree of "fitness-of-use" for a particular dataset, from a completeness point of view. In the case of a single schema, and assuming that classes exist in this schema that are semantically similar at a high degree but not identical, and additionally assuming that all output information is feasible only through a query, then all query results that involve such classes, are likely to be of poor quality due to a lack of information completeness, since a (possibly large) number of objects will not have participated in the formulation of the result. It is noteworthy to mention that currently no object oriented system supports semantic similarity checks on their class schema, hence it is allowed to exist

in a database, sometimes without the user being aware. MPOOST classes have to be generalized and flexible enough so that various application domains can map their classes in such manner that there is no data structure and behaviour loss. Different applications that do not have a strictly defined class schema, can map their entities, data and behaviour (if any) onto a common central model, e.g. the MPOOST model, by defining mappings between classes (e.g. "Road Type 1" is "Motorway") and possibly member mappings (e.g. "Polyline LeftBoundary" is "Polyline PavementLine"). In the case that the core model class lacks a class member, then it should be possible that a new member can be added. Such mappings can be defined by the user, provided that the appropriate user interface exists. A GUI is considered more likely to be used, since no formal language exists that can be used to specify class mappings between different schemata. An example is Laser Scan's Translate module, which provides a GUI which helps the user to map entities found in various, non object-oriented, external file formats (e.g. DXF) into classes found in a pre-designed class schema. The experienced user can use the customization language Lull to write such specifications. However, this approach requires that some code is written every time a new format has to be imported. Nonetheless, it is considered that the best way to achieve semantic interoperability is to design generalized and flexible classes, that can be specialized and customized to suit the needs of the application, without allowing the existence of semantically similar classes.

Regarding the differences in class schemata found in different application domains (schematic heterogeneity), the answer lies in how these schemata can map into a single one, rather than how these schemata can interrelate among them, since the possible combinations may be rather numerous and hence too complicated to examine. In this way, a central object model must be built, which has at its core the set of definable geo-referenced classes. Then, any other schema can map directly on this common classification, which acts like a superset of all applications that may use it. Classes in this model must be generalized and decomposed enough, with a flexible way of adding or removing members. Moreover, geo-referenced

classes are not part of any specific application domain. This is considered to enhance the multi-purpose nature of the model.

The answer to the problem regarding the behaviour and algorithms employed by different applications (syntactic heterogeneity) that demand different types of both methods and algorithms is reusability via information and behaviour sharing. If a set of fundamental algorithms and operations common to all application domains can be identified, then through object oriented development these can be implemented as reusable software components. More specialized, refined and customized algorithms can be based on top of this set. Software reusability refers to the specific implementation programming language used. If implementation is totally independent of this language, then the degree of reusability is the highest possible. This is achievable by incorporating interoperability standards in the implementation, so that any kind of object-oriented programming environment can reuse these fundamental components. In the case of the MPOOST data model CORBA is the chosen standard (however not incorporated within the implementation).

Another approach to tackle the issue of different semantics by different applications on the geo-entity level is to present reusable information to the application native model as if the data model had been developed in the first place with the specific application in mind. This is very similar to the approach that Bishr *et al.* [18] have proposed. It involves the enrichment of metadata information with a detailed and formal specification about the meaning of the geo-entities in the real world [11]. In order that this type of information is functional, it must be specified in a formal way.

[11]Bishr *et al.* [18] call this type of information as *context information.*

Chapter 4

Design of the MPOOST model

4.1 Introduction

This chapter describes the design phase of the MPOOST model, specifically the way that classes are formed based on the analysis stage (discussed in the second chapter), as well the internal structure and their grouping into packages. Diagrams in this chapter that document the model are according to the notation of the Unified Modeling Language, as it is described in the previous chapter (see section 2.4.2). The description begins from the packages of the model, followed by a detailed description of the classes within the packages. This is the reverse order by which the MPOOST model was designed, since building up initiated from classes and then moved on to group classes into packages. `Typewriter text` (Courier font) has been used in this chapter for the naming of the packages and classes. For packages, the full path is used along with the qualifier "package" (e.g. package `MPOOST.Spatial.Cartographic`). Class names are usually given without the package name that they belong to, when they are described in a section named after their package. Otherwise, the full name of the package is given with the class (e.g. class `MPOOST.Spatial.Cartographic.OpenPolyline`).

Not all class members are described in detail, only the ones that are of major importance. This is mainly for comprehensibility reasons. Detailed class member

descriptions are included in chapter five, where Java code may also be found. Persistence-wise, all modelled classes will be stored in the database involved, hence they have not been adorned explicitly as «persistent», but it is, rather, implied for all classes. Moreover, when multiplicity is not shown explicitly on the upper-right corner of the class icon, the class is implied to have n instances, unless it is an abstract class, which is not allowed to have any instances, or a utility class which may have only one instance.

For a full explanation of the UML terms and notation used, the reader should refer to chapter two, section 2.4.2.

4.2 Overall structure of the MPOOST model

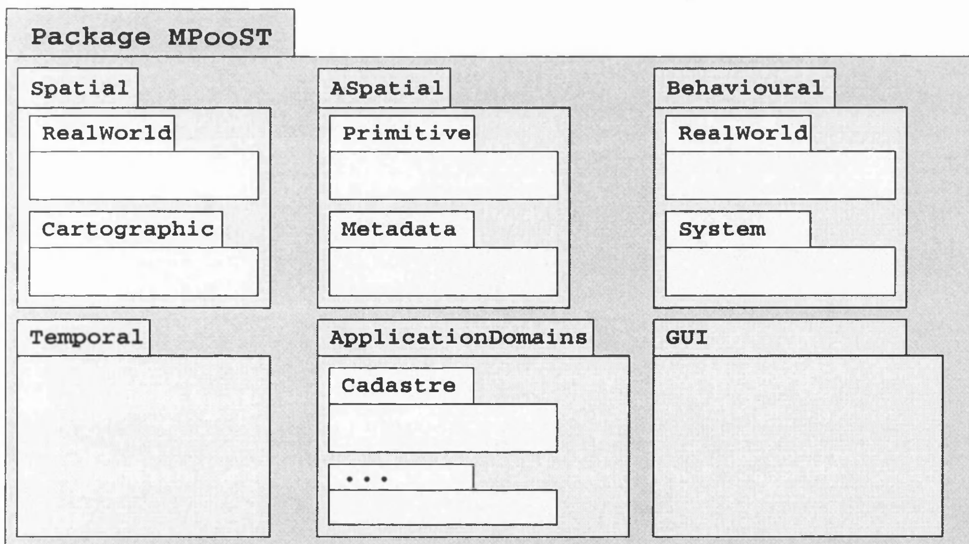


Figure 4.1: The overall structuring of the data model into packages

The three major component categories in the model are:

1. *Objects*: They are instantiated from object classes, which in turn belong to a package. Objects can model *any* type of entity. These entities are grouped according to their contents in six major packages (see figure 4.1 for the UML documentation):

(a) *Spatial.* Such classes are part of the package `Spatial`. Sub-packages in this package are `RealWorld` (which contains any spatially referenced objects as they exist in the real world) and `Cartographic` (which contains the cartographic representation of the real world object). The embedding space for objects belonging to the first sub-package is considered to be three-dimensional, while for the latter two-dimensional. The `Cartographic` package contains classes that model the geometry of the cartographic objects. The selection of the classes used to build both geometric modules is based on their homeomorphicity to formal geometric objects, which has been described in chapter three.

(b) *Aspatial.* Such classes are part of the package `ASpatial`. Classes belonging in this package do not involve any kind of geometry in their members. Metadata and coordinate systems information is also included in this package.

(c) *Temporal.* Such classes are part of the package `Temporal`. This package contains classes that model the temporal component of geo-referenced objects. Basic temporal constructs are *time points* (class `TimePoint`) which models a concept similar to that of chronons (see chapter one, page 65) and *time finite intervals* (class `TimeInterval`. Intervals are defined as sets of time points. An interval may have definite duration, when it is lower and upper bounded by a time point, or indefinite duration, where one of the two bounds is not defined. Temporal classes, as they do not employ any kind of geometry, could be part of the `ASpatial` package. However, temporality is considered to be one of the orthogonal aspects of geo-entities, and as such, should be treated as a peer to their aspatial component. Moreover, in this way, the timestamping module of the design may be used as an add-on to an existing class schema.

(d) *Behavioural.* This is the `Behavioural` package that contains mostly interfaces and utility classes which model the behaviour common to many entities. Behaviour is divided into two major categories, namely

the system behaviour (sub-package `System` and the real-world behaviour (sub-package `RealWorld`) which includes behaviour observed in the real world as part of specific application domain objects.

(e) *Application domains.* This group (package `ApplicationDomains`) contains sub-packages that respond to specific application domains and contain classes that model objects out of that domain. One of the sub-packages that is of interest for the development of the MPOOST data model is the Cadastre, which contains all the classes that model concepts related to a cadastral information system (sub-package `Cadastre`). It is considered that if more than one application will be manipulating data through the MPOOST model, then the relevant application domain specific declarations should be part of this package.

(f) *Graphical User Interface.* This is the `GUI` package. It contains all classes which are used to model the graphical interface that the user will interact with. Classes of this package are somehow separated from the rest of the structure, since they refer to classes of the implementation programming language (Java).

2. *Relationships.* It must be noted that most relationships involved in the design of the model are binary, that is they involve two model elements. These may be:

   (a) *spatial relationships* (or topological). They involve two spatial objects (objects that belong to the package `Spatial`), and are refering strictly to their topology, that is the external connectedness of an object (simple or complex) to another neighbourhing object. The usage of stereotypes separate spatial relationships into external and internal topological.

   (b) *aspatial relationships*

   (c) *temporal relationships.* The most important type of relationship in this group is the `VersionOf` relationship. It denotes an ordered sequence between the two objects involved, and signifies which object is the newer

version. Of course, this involves some kind of change in the state of the older object.

There has been an effort to avoid tertiary relationships as much as possible, as they are considered to be complicated implementation-wise. However they have been used in a few cases. To avoid such cases, whenever a tertiary relationship was initially observed among three classes, it has been turned into a binary, with the third class attached to the relationship itself as an association class.

Moreover each one of the above relationship categories is implicitly subdivided into the predefined stereotypes found in the UML specifications, namely specialization, aggregation, and association. Additionally, every relationship has a name (not necessarily unique) which makes more comprehensible to the reader the UML diagram. In some cases, relationship categories are treated as classes, provided that they have associated attributes. Such relationship classes are:

3. *Events.* All events are treated as objects in the OO model, therefore modelled by classes. They have both attributes and methods associated. Events may be separated, according to the context that they occured within, into:
   (a) *Real world events* and
   (b) *System events*

An event is usually instantiated whenever an object must undergo a change in one or more of its attribute values. Some methods also create event objects. System events are mostly associated with the user interaction process and the internal transaction mechanism.

4.3 Stereotypes defined in the UML model

Stereotypes (as discussed in chapter 2, section 2.4.3.15) are UML mechanisms that extend the notational elements of the language, and result in the visual

enhancement of the UML diagrams. The MPOOST development involves the definition and usage of stereotypes for classes, relationships and interfaces.

4.3.1 Stereotyped classes

Classes have been stereotyped into:

1. «Entity» for any kind of tangible object.
2. «Concept» for any kind of abstract idea or concept.
3. «RealWorldEntity», when the object modeled exists in the real-world as a tangible object. Examples are a building, a road, human being, a parcel etc.
4. «RealWorldConcept», when it models any non tangible, abstract but definable concept or idea, and refers to the application domains. An example is ownership rights.
5. «SystemConcept», when it models any non tangible, abstract but definable concept or idea, and refers to some computer (hardware or software) mechanism. An example is a transaction.
6. «SystemEntity», when the class models a purely system component that has no counterpart in the real world.

It is obvious that class stereotypes are specified using the specialization relationship. The first two stereotypes are likely to be used for abstract classes high up in the class hierarchy. Their sub-classes may specify whether an entity or a concept is a real world or a system. However, if an abstract class has as sub-classes both «Entity» and «Concept» classes, then it is not adorned with the appropriate stereotype information.

4.3.2 Stereotyped relationships

Relationships have been stereotyped into:

1. «ExternalTopological», for any relationships between different geometrical objects (e.g. the left and right polygons of a line).
2. «InternalTopological», for any relationships among any complex geometric object and its aggregate parts (e.g. the points comprising a line).
3. «Temporal», for the relationships between classes that enable the manipulation of historical information (e.g the previous and the next version of an application domain object.)
4. «Aspatial», for any other type of relationship that does not belong in any of the previous categories.

4.4 Detailed package structure and class description

This section describes the structure of the packages and classes in detail. Not all attributes and methods are mentioned here, since it is considered that not all of these are part of the design. The ones omitted are either trivial, or they are part of the implementation mechanism. A detailed class and class member documentation is included in the Java code and it is browsable in HTML format (see the MPOOST documentation on the accompanying CD).

4.4.1 The `MPOOST` package

The root package, which is a container for all packages, is called `MPOOST`. It contains the abstract class `MPOOSTRoot`, which is the root class for all classes across the model.

Abstract class `MPOOSTRoot` sub-class of `java.lang.Object`: The root class of all classes.

4.4.2 Package `MPOOST.Spatial`

The package `MPOOST.Spatial` contains packages with regard to the geometric modelling of the geo-referenced objects.

4.4.2.1 Package `MPOOST.Spatial.RealWorld`

The package `MPOOST.Spatial.RealWorld` contains classes that model the geometry of real-world objects. The fundamental class is the `Point` class, out of which all other classes are constructed. As the space that embeds all objects is the Euclidean R_3, the Point class contains a triplet of coordinates. These can be in many formats, according to the ReferenceSystem object that the Point is related to. The overall hierarchy of this package is shown in figure 4.2, on page 269. [1]

1. Abstract class `RRoot` sub-class of `MPOOST.Root`: this is the root class for all classes in this package.
2. Abstract class `ZeroExtent` sub-class of `RRoot`: The parent class for all zero dimensional classes. Attributes in this class are common to all of its sub-classes, hence their presence in this class.

   Attributes:

   (a) `Coordinate` X,Y,Z: Coordinate triplet for the point. This might be either Cartesian or geographical, depending on the coordinate system in which the point object is embedded.

   (b) `Double` SX,SY,SZ: Standard Deviation of the coordinate triplet respectively.

   (c) `Double` RMSE: The Root Mean Square Error associated with the point.

[1] It must be noted that the implementation of this package has been omitted, mostly due to the limited time schedule. However, the model is considered to be flexible enough to be functional only with the `MPOOST.Spatial.Cartographic` classes as the geometric component of the application domain entities.

(d) `CoordinateSystem` CSystem: The coordinate system object that the zero extent belongs to.

3. Concrete class `ZeroDSimplex` sub-class of `ZeroExtent`: A zero-dimension simplex.

   Attributes:

   (a) Set of `ZeroDComplex` aComplex: References to the zero-complexes that it belongs to.

4. Concrete class `ZeroDComplex` sub-class of `ZeroExtent`: A zero-dimension complex (zero-complex). It is contains a non-empty set of `ZeroDSimplex` objects.

   Attributes:

   (a) Set of `ZeroDSimplex`: The non-empty set of zero-simplex objects that it is composed of.

5. Concrete class `Point` sub-class of `ZeroDSimplex`: This the fundamental zero-dimensional geometrical object. Point objects do not connect topologically with other geometrical objects.

6. Concrete class `Centroid` sub-class of `ZeroExtent`: The centroid of a polygon.

   Attributes

   (a) `SimplePolygon` Polygon: The polygon object that the centroid belongs to.

7. Concrete class `Vertex` sub-class of `Point`: A vertex is a point that polylines are composed of. `Vertex` objects may belong to a single `PolyLine`. Hence, they have two references, one to the previous vertex in the sequence, and one for the next vertex in sequence. These references are not necessary, since the can be accessed through the `PolyLine` class, however it is considered that it enhances query output on the vertex sequence, since no reference is necessary to the `PolyLine` that it belongs to.

   Attributes:

(a) LineSegment Segment: The `LineSegment` object that the vertex belongs to.

(b) Vertex PreviousVertex: The previous vertex in order.

(c) Vertex NextVertex: The next vertex in order.

8. Concrete class `Node` sub-class of `Vertex`: A node is where a `PolyLine` begins or ends.

   Attributes:

   (a) Set of `OpenPolyline` Polylines: The non-empty set of `OpenPolyline` objects that begin/end at this node object.

9. Concrete class `Cell` sub-class of `Point`: A cell is used to construct raster objects (class `Raster`. It may contain a value of any type. Details about cell size and structuring are kept in the `Raster` class.

   Attributes:

   (a) `Object` Value: The value that the cell contains. Any object for which its class is a sub-class of `Object` can be a potential value for the cell.

   (b) `Raster` aRaster: The raster object it belongs to.

10. Abstract class `OneExtent` sub-class of `RRoot`: This is any linear, 1-dimensional geometrical object. One dimensional system entities are composed of points instead of line segments, since line segment information is not always necessary, and if used between any polyline and a point set it only adds complexity to the model. However, if it is desired, line segments can be used to model any one-extent.

    Attributes:

    (a) `3DLength` TotalLength: The total length of the object.

    (b) `2DLength` XYPlaneLength: The length of the projected plane on the X, Y plane.

    Methods:

(a) `Length` CalculateLength(): Calculates the total length of the object. Empty declaration that will be overridden by the sub-classes to fit specific needs, taking into account the dimensions and the type of coordinates available.

11. Concrete class `OneDSimplex` sub-class of `OneExtent`: This class models a 1-simplex which is equivalent to a line segment. It is composed of two `ZeroDSimplex` objects.

    Attributes:

    (a) `ZeroDSimplex` Start: The start point of the simplex.

    (b) `ZeroDSimplex` End: The end point of the simplex.

    (c) `OneDComplex` Complex: The one-dimensional object that this object may belong to.

12. Concrete class `OneDComplex` sub-class of `OneExtent`: A one-dimensional simplicial complex, consisting of a non-empty set of one-dimensional simplices.

    Attributes:

    (a) Set of `OneDSimplex` SimplexSet: The non-empty set of one-dimensional simplices.

    (b) `Integer` SimplexNumber: The total number of simplices comprising the simplicial complex.

13. Concrete class `Network` sub-class of `OneDComplex`: A network structure, which is a non-directed, cyclic graph, embedded in 3D space.

14. Concrete class `LineSegment` sub-class of `OneDSimplex` A straight line which starts and ends on a node.

    Attributes:

    (a) `Node` Start: The start node of the segment.

    (b) `Node` End: The end node of the segment.

15. Concrete class `PolyLine` sub-class of `OneExtent`: A polyline is an ordered set of vertices. It may be either open or closed, self crossing or not.

    Attributes:

    (a) `Integer` NumberOfVertex: The total number of points that define the polyline. Minimum vertex number must be three.

    (b) Ordered set of `Vertex` VertexSet: The set with the polyline vertices, with at least three vertices.

    Methods: `Calculate3DLength`: This operation is used to calculate the total length of the polyline. The formula used is:

$$L = \sum_{i=1}^{n} \sqrt{(X_{i+1} - X_i)^2 + (Y_{i+1} - Y_i)^2 + (Z_{i+1} - Z_i)^2} \tag{4.1}$$

16. Concrete class `ClosedPolyline` sub-class of `PolyLine`: A closed polyline is a polyline whose first and last nodes are identical. It is formed out of open polylines (at least two). A closed polyline is used to form simple polygons.

    Attributes:

    (a) `SimplePolygon` EnclosedArea: The polygon to which the polyline is a boundary.

17. Concrete class `OpenPolyline` sub-class of `PolyLine`: An open polyline is a polyline whose first and end node are *not* identical.

    Attributes:

    (a) `Node` StartNode: The start node of the open polyline. This is used to associate the `OneExtent` object with the set of `LineSegment` objects it is composed of.

    (b) `Node` EndNode: The end node of the open polyline. The same applies as for the previous attribute.

18. Abstract class `TwoExtent` sub-class of `RRoot`: This is any 2-dimensional geometrical object.

    Attributes:

(a) `Area` TotalArea: The total area of the polygon.

Methods:

(a) `Area` CalculateArea: Calculates the total area of the polygon, depending on the type. It is overridden by sub-classes to take into account the possible number of outer and inner polygons, or the number of triangles, in the case of the surface.

19. Concrete class `TwoDSimplex` sub-class of `TwoExtent`: This class models a two dimensional simplex, which is equivalent to a triangle, but it has been included for semantic consistency.

    Attributes:

    (a) `TwoDComplex` Complex: The two-dimensional simplicial complex that this object belongs to.

20. Concrete class `TwoDComplex` sub-class of `TwoExtent`: This class models a two dimensional simplicial complex, which is a set of two dimensional simplices, equivalent to a set of triangles. It has been included for semantic consistency. Attributes:

    (a) Set of `TwoDSimplex` TwoDSimplices: The non-empty set of two dimensional simplices.

21. Concrete class `SimplePolygon` sub-class of `TwoExtent`: A simple polygon is the area that is enclosed by a closed polyline. It's geometry cannot have any inner island polygons. However, it can be linked to other polygons which are inner to its geometry in the case where the earlier is referenced through a `ComplexPolygon`, where it is used to form complex polygons, either as an inner, an outer or both (to different complex polygons). Special types of simple polygons are the triangle and oblong (square) shapes. A simple polygon is associated with two `OneExtent` objects.

    Attributes:

    (a) `ClosedPolyline` Boundary: The closed polyline that is the boundary of the polygon.

   (b) `SimplePolygon` Outer: The polygon that encloses this polygon. Values may be null. When it is not null, it is used to associate outer and inner polygons that form `ComplexPolygon` objects.

   (c) Set of `SimplePolygon` Inner: The polygons that this polygon encloses. May be null.

22. Concrete class `ComplexPolygon` sub-class of `TwoExtent`: A complex polygon is the area enclosed by a set of simple polygons. There is always one outer simple polygon and a set of at least one simple inner polygons. Every inner polygon may have a number of inner polygons and so on. This structure is a hierarchy of `SimplePolygon` objects. Inner polygons are always topologically enclosed by the outer polygon.

    Attributes:

    (a) `Integer` Level: The maximum level in the hierarchy of island polygons.

    (b) `SimplePolygon` OuterPolygon: The outermost polygon of the object.

    (c) `Integer` Level: The level number of the polygon in a `ComplexPolygon` hierarchy.

23. Concrete class `Triangle` sub-class of `TwoDSimplex`: This is a simple polygon which is defined by three node objects.

    Attributes:

    (a) Ordered set of `Node` Nodes: The nodes of the triangle.

    (b) Ordered set of `LineSegment`: The line segments of the triangle.

24. Concrete class `Region` sub-class of `TwoExtent`: A region is the area defined by a set of complex polygons, which may be spatially disjoint. There is a set of outer simple polygons, each of which may have a set of enclosed inner simple polygons (as described in the simple polygon class).

    Attributes:

    (a) Set of `ComplexPolygon` Polygons: The set of complex polygons that belong to the region.

25. Concrete class `Surface` sub-class of `TwoExtent`: This class models any two-dimensional surface, defined by a Triangulated Irregular Network (TIN), embedded in three dimensions. It is composed of a number of `Triangle` objects.

    Attributes:

    (a) Set of `Triangles` SurfaceSegments: The set of triangles that the TIN surface is composed of.

    (b) `Region` Boundary: The set of polygons that are the boundaries of the surface, as modeled by the `Region`. Inner polygons may be null, in the case where there are no holes in the TIN.

26. Abstract class `ThreeExtent` sub-class of `RRoot`: This is any 3-dimensional geometrical object.

    Attributes:

    (a) `Volume` Volume: The volume of the object.

    (b) `Area` SurfaceArea: The area of the bounding object surface.

    Methods:

    (a) `Volume` CalculateVolume(): Calculates the volume of the object. It is overridden by sub-classes for customized calculation of the object, depending on the geometry.

27. Concrete class `SimpleSolid` sub-class of `ThreeExtent`: This class models any three dimensional solid, as it is composed of its bounding surface, without any internal solids.

    Attributes:

    (a) Set of `Surface` Boundary: The bounding surface of the object. Must not be null.

    (b) `SimpleSolid` OuterSolid: The simple solid object that this object is surrounded by. May be null if the object is not part of a complex solid.

    (c) Set of `SimpleSolid` InnerSolids: The set of simple solid objects that this object includes. May be null if it is not part of a complex solid.

28. Concrete class `ComplexSolid` sub-class of `ThreeExtent`: An object with an external solid object, and a hierarchy of inner simple solid objects, with any degree of recursiveness. Inner solids are referenced indirectly through the outermost solid object.

    Attributes:

    (a) `Integer` Level: The highest level of inner solids.

    (b) `SimpleSolid` OuterSolid: The outermost solid object.

29. Concrete class `ThreeDSimplex` sub-class of `ThreeExtent`: This is a 3-simplex class, which is equivalent to a tetrahedron. It is composed of four two-dimensional simplices.

    Attributes:

    (a) Set of `TwoDSimplex` Faces: The set of two-dimensional simplices that the objects consists of.

    (b) `ThreeDComplex` Complex: The three dimensional complex that this object belongs to.

30. Concrete class `ThreeDComplex` sub-class of `ThreeExtent`: This is a three dimensional simplicial complex, defined as a set of three dimensional simplex objects. Attributes:

    (a) Set of `ThreeDSimplex` Simplices: The set of three dimensional simplices that this object contains.

31. Concrete class `Tetrahedron` sub-class of `ThreeDSimplex`: This is the fundamental building block for three dimensional solids. Although it is similar to the `ThreeDSimplex`, it has been included for consistency reasons.

    Attributes:

    (a) Set of `Triangles` Faces: The faces of the tetrahedron.

4.4.2.2 Package `MPOOST.Spatial.Cartographic.Geometry`

The package `MPOOST.Spatial.Cartographic.Geometry` includes classes that model the cartographic geometry of the geo-referenced objects. Its purpose is to provide a minimal two-dimensional model which will serve as the basis upon which map products will be created and stored. Cartographic objects are considered to be derived from real-world geo-referenced objects through projection of their geometry onto a mathematical surface, according to the cartographic projection used. The basic unit of aggregation is the class `CPoint`. The rest of the classes in this package are considered to be either a specialization (e.g. class `Node`) or an aggregation (e.g. class `Polyline`) of this class. Classes in this package are related to the `ProjectionSystem` class (from the `MPOOST.ASpatial.Metadata.CoordinateSystems` package) used to create them. Every class in the geometry package is associated with an ordered set of classes from the `MPOOST.Spatial.Cartographic.Representation` package, always according to their dimensionality (e.g. a point object is associated with a set of point symbols, a linear geometric object is associated with a set of linear symbols, etc.). The ordered set denotes the aggregational layering of the representation of a single geometric feature. In this way, complex symbols may be assigned to a geo-entity and be drawn on a map composition. [2] Classes in the package are:

1. Abstract class `CartographicRoot`, which is the parent class of this package. This class contains members that are common to all of its sub-classes.
2. Abstract class `ZeroExtent` sub-class of `CartographicRoot`: This class models zero dimensional cartographic objects. It is associated with an `SPoint` class which is used to visualize the geometry of the sub-classes.
3. Concrete class `CPoint`: This class models a point as a zero dimensional geometric feature. It has no topological associations with other cartographic objects. It is associated with one real world geometric point (class

[2] As this package has been fully implemented in the Java programming language, and therefore thoroughly documented within the code, hence it is not documented in detail in this chapter.

MPOOST.Spatial.RealWorld. Point) that it models.

4. Concrete class `Centroid` sub-class of `ZeroExtent`: A centroid models the centroid point of a polygon, and as such, it is associated with a `SimplePolygon` class.

5. Concrete class `Vertex` sub-class of `CPoint`: This class is used to construct 1-extent classes. It is associated with one 1-extent class, where it belongs, and with two other `Vertex` objects, a PreviousVertex and a NextVertex.

6. Concrete class `Node` sub-class of `Vertex`: This class models a node as a point which either starts or ends any linear (one-extent) object. Intersections of 1-extent classes are supposed to occur on nodes. It is associated with a set of 1-extent classes.

7. Abstract class `OneExtent` sub-class of `CartographicRoot`: It is an ordered set of (at least two) `Vertex` classes. Ordering of the set is for storing the information on the direction of the linear object. It is associated with exactly two `Node` classes: one as a StartNode and one as an EndNode.

8. Concrete class `OpenPolyline` sub-class of `OneExtent`: This class models an open polyline, which is composed of at least two `Vertex` objects. The first and the last Vertex objects are a `Node` class (StartNode and EndNode), which must not be identical[3].

9. Concrete class `ClosedPolyline` sub-class of `OneExtent`: This is a closed polyline. The main difference with the `OpenPolyline` class is that the start and the end nodes are identical objects [4]. This must be verified prior to creation of the object.

10. Concrete class `Network` sub-class of `OneExtent`: This class models the structure of a DCG-like network (Directed Cyclic Graph).

---

[3]coordinate inequality (which implies object identity inequality)

[4]object identity equality since two nodes may have the same values for coordinates but may be different objects.

11. Abstract class `TwoExtent` sub-class of `CartographicRoot`: This is an abstract class that has as children any two-extent geometrical object.

12. Concrete class `SimplePolygon` sub-class of `TwoExtent`: This is the class that models the area enclosed by a polygon, along with its surrounding border. It is not allowed to have any "island" polygons (holes). It consists of exactly one `ClosedPolyline` class and of exactly one `Centroid` class.

13. Concrete class `Triangle` sub-class of `SimplePolygon`: This class models a triangle. It is a `SimplePolygon` consisting of a `ClosePolyline` with exactly three `Vertex` objects.

14. Concrete class `ComplexPolygon` sub-class of `TwoExtent`: This class models any polygon with islands, that has an outer polygon and a number of internal polygons. It consists of exactly one Outer `SimplePolygon` and of a non-zero set of inner `SimplePolygon` (at least one such object).

15. Concrete class `Region` sub-class of `TwoExtent`: This models any two-extent areal object, regardless of how complex it might be. It consists of a set of `ComplexPolygon` objects (at least one). These in turn may be composed of a set of `SimplePolygon` objects (at least one) as mentioned already.

4.4.2.3 Package ...`Cartographic.MapComposition`

The package `MPOOST.Spatial.Cartographic.MapComposition` holds information on classes that model the structure of a map composition either on-screen or as a digital file that may be printed to produce a hardcopy. Classes of the package are:

1. Concrete class `Legend`: The legend of the map. It contains explanatory information on the cartographic symbols shown.

2. Concrete class `FeatureSet`: A container for all cartographic objects displayed. Attributes:

(a) Set of `CRoot`: The set of cartographic objects that are displayed on the map composition.

3. Concrete class `Composition`: A map composition as it is being displayed on screen or printed on paper, or written to a file for exporting.

4.4.3 Package `MPOOST.Representational`

4.4.3.1 Package `MPOOST.Representational.Symbol`

The package `MPOOST.Representational.Symbol` contains classes that model the cartographic representation of a spatial object, either on screen or on a hard-copy. Symbols have been divided into three major categories for point, linear and areal cartographic symbols (see figure 4.3 on page 270).

1. class `Color`: Models the color information of a cartographic symbol. This class supports 16777216 (2^{24}) unique colors. Attributes:

   (a) `Integer` R,G,B: The red, green and blue components of the color, expressed in the range from 0 to 255.

   (b) `Integer` C,Y,M: The cyan, yellow, and magenta components of the color, expressed in the range from 0 to 255.

   (c) `Integer` H,S,I: The hue, saturation and intensity of the color.

   Methods:

   (a) `Color` toRGB(): Returns the R,G,B triplet of the color.

   (b) `Color` toHSI(): Returns the H,S,I triplet of the color

   (c) `Color` toCYM(): Returns the C,Y,M triplet of the color.

2. Concrete class `Shape`: This class models the geometry of a shape. Attributes:

   (a) `Polyline` Outline: The outline of the shape.

   (b) `Angle` Orientation: The orientation of the shape.

(c) `Color` FillColor: The color used to fill the interior of the shape.

(d) `Color` LineColor: The color used for the outline of the shape.

(e) `Bitmap` Icon: An external bitmap icon (raster) used instead of using a picture or a vector shape.

(f) `Picture` Pic: An external vector picture used instead of using a bitmap icon or a vector shape

(g) `Colormap` Lookup: The colormap that associates values from the map feature to colors of the shape. Can be null.

3. Concrete class `Pattern`: Attributes:

   (a) `Angle` Orientation: The orientation of the pattern.

   (b) `Color` FillColor: The color used to fill 2-extent objects.

   (c) `String` PatternType: The type of the pattern, specifying its definition.

   (d) `Colormap` Lookup: The colormap that associates values from the map feature to colors of the pattern. Can be null.

4. Abstract class `RootSymbol`: The parent class for cartographic symbols. Methods:

   (a) Draw(): Method that displays the geometry of the cartographic symbol according to its specifications and the geometry of the map feature. It is overridden by sub-classes for customized display, according to the geometry of the map feature, the symbol and the type of display (screen or file).

5. Concrete class `PointSymbol` sub-class of `RootSymbol`: A point symbol, with a location, a shape and a size. Attributes:

   (a) `GraphicCoordinate` LocationX, LocationY: The location of the symbol on the map composition.

   (b) `Shape` Geometry: The geometry of the point symbol.

   (c) `Integer` SizeX, SizeY: The size of the symbol.

6. Concrete class `LinearSymbol` sub-class of `RootSymbol`: A class with specifications on drawing linear map features.
   (a) Paired list of `GraphicCoordinate` Outline: The geometry of the map feature to be drawn, expressed in graphic coordinates.
   (b) `Shape`: The geometry of the symbol.
   (c) `Integer` Width: The width of the line symbol.
   (d) `Pattern` LinePattern: The pattern used to draw the line.
7. Concrete class `ArealSymbol` sub-class of `RootSymbol`: Attributes:
   (a) Paired list of `GraphicCoordinate` Outline: The polyline that encloses the area to be drawn.
   (b) `Pattern` FillStyle: The specification of the interior filling of the polygon.
8. Concrete class `Colormap`: This class maps the values of a `Raster` object to displayable colors (be it on screen or on a hardcopy). Attributes:
   (a) Paired list of (`Color`, `Object`): The mapping of raster values to colors.

4.4.4 Package `MPOOST.Behavioural`

This package contains interfaces and utility classes. Interfaces are being implemented by various classes. Utility classes contain methods that cannot semantically be part of any other class.

4.4.4.1 Package `MPOOST.Behavioural.Utility`

This package contains classes that include mainly operations that are not (or should not) be part of another class across the class schema. More specifically, the following methods have been specified:

1. Concrete class `DataImportUtility`: Contains classes and operations used to import data from external sources, mainly various GIS format files. For

the implementation purposes, the DXF format has been used to import data from external files.

2. Concrete class `Math`: This class contains any mathematical formulas defined as methods that are useful across the model.

4.4.5 Package `MPOOST.ASpatial`

The package `MPOOST.ASpatial` contains all aspatial classes, that are part of a geo-entity.

4.4.5.1 Package `MPOOST.ASpatial.Primitive`

The package `MPOOST.ASpatial.Primitive` contains all the classes which are fundamental for the model. Most of them are originating from the Java programming language, as it has been chosen as the implementation language and they are called Java Native Classes (JNC). Classes that are not coming from the Java PL, are either sub-classes or they employ a JNC through a relationship.

1. class `Constant`: This class stores data on constant values.
2. class Integer (JNC)
   (a) class `Year` sub-class of `Integer`
   (b) class `Hour` sub-class of `Integer`
   (c) class `Minute` sub-class of `Integer`
   (d) class `Second` sub-class of `Integer`
   (e) class `Millisecond` sub-class of `Integer`
3. class String (JNC)
   (a) class `UnitOfMeasure` sub-class of `String`. This is a collection of units of measurement. Example values are: "meter", "feet", "degrees", "rad", "gon"
   (b) class `Day` sub-class of `String`

(c) class `Month` sub-class of `String`

(d) class `Model` sub-class of `String`: The data model of a dataset. Values can be "raster", "vector" *etc.*

4. class Long (JNC)

   (a) Concrete class `Currency` sub-class of `Long`

5. class `Float` (JNC)

6. class `Double` (JNC)

   (a) Abstract class `Coordinate`

      i. Concrete class `LinearCoordinate` sub-class of `Coordinate`: Models cartesian coordinates (x, y, z)

      ii. Concrete class `AngularCoordinate` sub-class of `Coordinate`: Models geographical coordinates as expressed in angles (φ, λ)

      iii. Concrete class `GraphicCoordinate` sub-class of `Coordinate`: Models coordinates used on a map composition (x, y)

7. class `Area` sub-class of `Double`

8. class `Distance` sub-class of `Double`

9. class `Angle` sub-class of `Double`

10. class `Volume` sub-class of `Double`

11. class `Time`: Models time within a day. Attributes:

    (a) `Hour`

    (b) `Minute`

    (c) `Second`

    (d) `Millisecond`

12. class `Date`: Models a date. Attributes:

    (a) `Day`

    (b) `Month`

    (c) `Year`

13. class `Boolean` (JNC): The boolean value (true or false).

4.4.5.2 Package `MPOOST.ASpatial.Metadata`

The package `MPOOST.ASpatial.Metadata` contains metadata information of a specific dataset. A dataset is a container for system entities, that undergo changes due to a system event.

All application domains have in common a group of system entities modelled by the `Dataset` class. This class includes references to all metadata classes presented below. In this way, metadata information can easily plug-in to a group of objects that model geo-entities. Some metadata information is stored within geo-entities individually, and some is implicitly denoted through the dataset that the geo-entity belongs to.

1. Abstract class `Dataset`. This class is used as a generic container for either input or output datasets. It is a way of grouping system classes or map features.
2. Final class `InputDataset` sub-class of `Dataset`. This is a class used to model a group of data that were part of a single survey session.
3. Final class `OutputDataset` sub-class of `Dataset`. Used to group objects that are exported out of the system at some specific timestamp.

Metadata classes in this package are:

1. Concrete class `SpatialModel`: Description of the spatial data model employed in a dataset.

   Attributes:

   (a) `String` Name: The name of the model.

   (b) List of `Class` PrimitivesList: Specification of primitive spatial objects. This is a list of the classes of the primitive spatial objects. Their description and explanation of their geometry and topology is contained within the classes.

2. Concrete class `DataSummary`: Summary about the data contents, and it is used for a quick overview.

   Attributes:

   (a) Paired list of (`String`, `Source`) Origin: The source of the dataset. This is a paired list of the names of the company/institution responsible the data collection along with the type of method of data collection.

   (b) List of `Class`: The classes of objects included in the dataset.

   (c) `Region` Coverage: The area that geo-entities cover.

   (d) `RWInterval` DatePeriod: The real world time period that the data are valid.

   (e) `DBInterval` SystemPeriod: The total time that the data existed in the system.

   (f) `String` Media: Type of source media. This is textual description of the medium from which the source data set originates.

   (g) `Date` MediaDate: This is the date that the source media refer to.

   (h) Paired list of (`String`, `Date`): Updates: This is a set of dates with updates on the data contents along with the description of the update.

   (i) `String` SourceContribution: This is a textual description of what the type of possible contribution to the data set from the source.

   (j) `String` Scale: The scale used to collect the data, in the case of an `InputDataset`.

   (k) `CoordinateSystem` GeodeticSystem: The coordinate system of the dataset.

   (l) `Model` DataModel: The data model of the dataset.

   (m) Paired list of (`String`, `String`) CodeDefinition: The definition of codes used by object members. This is a list of codes (numeric/alphanumeric for codes or textual for classes) along with a textual description of what these codes represent.

   (n) `Double` DeviationX, DeviationY, DeviationZ: Average standard deviation of point features, for every coordinate, referring to the dataset as a

whole. It is calculated from the standard deviation for individual points that is stored along with their geometry.

(o) Paired list of (`Class`,`String`) `AttributeAccuracy`: Statistics on attribute error. This is a list that correlates the attribute fields in the dataset along with statistical information about this attribute.

3. Concrete class `Source` sub-class of `String`: This class models the method of data collection. This involves all data collection techniques used. Attributes:

   (a) `String` Name: The survey techniques name. A list of all techniques used. This might be "topographic survey", "photogrammetric acquisition", "sattelite image classification" , "vector digitizing", "scanning". More values may be added here.

   (b) `String` Info: Information on the instrumentation involved in the collection technique.

4. Abstract class `Transformation`: It models any kind of transformation that the data may undergo.

5. Concrete class `CoordinateTransformation` sub-class of `Transformation`: Any type of coordinate system transformation.

   Attributes:

   (a) `CoordinateSystem` OldSystem: The coordinate system that the data are refering to before the transformation.

   (b) `CoordinateSystem` NewSystem: The coordinate system that the data refer to after the transformation.

6. Concrete class `ModelTransformation` sub-class of `Transformation`: This involves a data model transformation involved, namely from raster to vector and vice versa.

   Attributes:

   (a) `Model` OldModel: The model that the data are refering to before the transformation.

   (b) `Model` NewModel: The model that the data refer to after the transformation.

7. Concrete class `AttributeTransformation` sub-class of `Transformation`: The type of attribute transformation involved in a class.

   Attributes:

   (a) `Class` Member: The member whose values undergo the transformation.

   (b) `Class` OldType: The attribute type that the data are refering to before the transformation.

   (c) `Class` NewType: The attribute type that the data refer to after the transformation.

8. Concrete class `Modification`: This class models a single modification of a dataset, that took place for a speficic period in the system. An effort has been made to include all possible variations of modifications.

   Attributes:

   (a) `DBInterval` TimeStamp: The period within which the modification took place, expressed in system time.

   (b) `Transformation` DataTransformation: The possible transformation of the data.

9. Concrete class `Lineage`: The history about a dataset. It contains a list of timestamped `Modification` objects.

   Attributes:

   (a) List of `Modification`: The list containing all the timestamped modifications of the dataset. Information in this class is added by `SystemEvent` objects.

4.4.5.2.1 Package ...`Metadata.ReferenceSystems` All the requirements discussed in chapter three (See section 3.8.2) dictate the need to classify hierarchically the set of coordinate systems, thus producing the package ReferenceSystems. Every geometric 0-extent class is associated with a class of a reference

system, therefore (e.g.) every point object will be linked to a reference system, e.g. WGS84. 1-, 2-, and 3-extent objects are implicitly linked to a reference system through their aggregation relationship (of 0-extent) to them.

4.4.5.2.2 Package `...Metadata.CoordinateSystems` The package `MPOOST.Aspatial.M` contains classes that model the reference systems used by the geometrical objects.

1. Abstract class `RootSystem` sub-class of `MPOOSTRoot`: The root class for this package.
2. Abstract class `Geodetic` sub-class of `RootSystem`: A coordinate system that refers to unprojected coordinates, geocentric or geographic (e.g. WGS84). Attributes:
   (a) `String` Name: The name of the coordinate system.
3. Abstract class `Projection` sub-class of `RootSystem`: A coordinate system that involves a map projection (e.g. Transverse Mercator). Attributes:
   (a) `String` Name: The name of the projection.
4. Abstract class `Shape` sub-class of `Projection`: Root class for categorizing projections according to the geometrical surface they employ.
5. Abstract class `Cylindrical` sub-class of `Shape`: A map projection that uses a cylinder as a projection surface.
6. Abstract class `Conical` sub-class of `Shape`: A map projection that uses a cone as a projection surface.
7. Abstract class `Azimuthal` sub-class of `Shape`: A map projection that uses a plane as a projection surface.
8. Abstract class `Distortion` sub-class of `Projection`: Root class for categorizing projections according to the distortion they cause to the metrics of geo-entities.
9. Abstract class `Equidistant` sub-class of `Distortion`: Any projection that preserves distances.

10. Abstract class `EqualArea` sub-class of `Distortion`: Any projection that preserves areas.

11. Abstract class `Conformal` sub-class of `Distortion`: Any projection that preserves directions.

12. Abstract class `Cartesian` sub-class of `Geodetic`: Any coordinate system that uses cartesian coordinates.

13. Abstract class `Geocentric` sub-class of `Cartesian`: A real world system that uses cartesian coordinates and considers the centroid of the earth as the point (0,0,0).

14. Abstract class `Geographical` sub-class of `Geodetic`: A real world coordinate system that uses geographical coordinates (φ, λ, h)

15. Abstract class `EarthShapes` sub-class of `RootSystem`: Root class for categorizing reference surfaces according to their geometry.

16. Abstract class `EllipsoidSystem` sub-class of `Geographical`, `EarthShapes`: A geographical coordinate system that employs an ellipsoid of revolution. Attributes:

    (a) `Length` MajorSAxis: The length of the major ellipsoid semi-axis.

    (b) `Length` MinorSAxis: The length of the minor ellipsoid semi-axis.

    (c) `Double` E: The eccentricity of the ellipsoid.

    (d) `Double` Ed: The second eccentricity of the ellipsoid.

    (e) `Double` F: The flattening of the ellipsoid.

    (f) `GeographicCoordinate` F,L: The latitude and longitude (φ, λ) of a point on the ellipsoid.

17. Abstract class `SphereSystem` sub-class of `Geographical`, `EarthShapes`: A geographical coordinate system that employs a sphere.

4.4.6 Package `MPOOST.Temporal`

The package `MPOOST.Temporal` contains all classes that model the temporal nature of the information.

1. Abstract class `Event` sub-class of `MPOOST.MPOOSTAppRoot` Attributes:
   (a) `String` Description: The description of the event
2. Concrete class `SystemEvent` sub-class of `Event`. Any event that regards system activity and has no counterpart in the real world. Attributes:
   (a) `RealWorldEvent` RealWEvent: A reference to the real world event that triggered this system event.
3. Concrete class `RealWorldEvent` sub-class of `Event`. This class models any real world event. It is usually accompanied by a system event, which reflects the system update that must take place. Attributes:
   (a) `SystemEvent` SysEvent: The system event that is triggered so that changes are reflected in the system entities.
4. Abstract class `TimeStamp` sub-class of `MPOOST.MPOOSTRoot`. This is a timestamp that refers to a single moment in the time domain.
5. Abstract class `TimePoint` sub-class of `TimeStamp`. This is a timestamp that refers to a single moment in the time domain.

   Attributes:
   (a) `Time` TimeData.
   (b) `Date` DateData.
6. Concrete class `DBTimePoint` sub-class of `TimePoint`. A momentum timestamp which is used to mark system events and entity life cycles.
7. Concrete class `RWTimePoint` sub-class of `TimePoint`. A momentum timestamp which is used to mark real world events and entity life cycles.
8. Abstract class `Interval` sub-class of `TimeStamp`. Any time period that spans more than one time point.

Attributes:

(a) `TimePoint` Start: The start timepoint of the period.

(b) `TimePoint` Stop: The end timepoint of the period.

9. Concrete class DBInterval sub-class of Interval. A time interval that is associated with system events and entities.

10. Concrete class RWInterval sub-class of Interval A time interval that is associated with real world events and entities.

4.4.7 Package `MPOOST.GUI`

The package `MPOOST.GUI` contains all the classes that are written for the graphical user interface of the model. Class members specific and detailed specification can be found in the relevant HTML files of the package.

4.4.8 Package `MPOOST.ApplicationDomains`

The parent class for all classes in this package is the **`AppRoot`** class. A self-referencing «temporal» relationship relates any object to its previous and its next state. Every object that is instantiated from a sub-class of this class belongs to a dataset group (**`Dataset`** class). A system event is associated whenever a new version is being produced due to a value change in one or more attributes. Classes belonging in this package are composed of a geometry, a temporal stamp, a series of interfaces specifying behaviour, as well as the cartographic representation, which uses the geometry of the class and any attribute value for the visual variables. If a new version is being produced, then the object has a different identity. Not all attribute values may change, and the ones that remain unchanged refer to values found in the immediately previous version through the reference to the object identity. In this way we avoid duplicating data when a new version is produced. Moreover nullable attributes that contain null values denote a lack of data that could be traced back to first version of the object. Application

domain classes are considered to be created on demand, that is the user will be able to specify the members of the class, according to the geo-entity that is being modelled. However, a few core classes have been included. Figure 4.4 on page 271 shows the UML diagram of the `AppRoot` class).

Attributes:

1. `Integer` VersionLevel: The sequential number of the object version.
2. `AppRoot` PreviousVersion: The previous version of the entity. Can be null if no value changes have occured regarding this object.
3. `AppRoot` NextVersion: The next version of the entity. Can be null if this object has not changed in any way.
4. `SystemEvent` ThisVersionReason: The system event that caused the object to transcend to the current version.

4.4.8.1 Package `MPOOST.ApplicationDomains.Cadastre`

The package `MPOOST.ApplicationDomains.Cadastre` contains sample classes that have been defined for the cadastral case study.

4.4.8.2 Package `MPOOST.AppilicationDomains.Topography`

The package `MPOOST.AppilicationDomains.Topography` contains real world entities that are part of a topographic survey. It is considered to be a base package, since other application domains will reference entities in this package (*e.g.* Package MPOOST.ApplicationDomains.Cadastre).

The next chapter (Implementation) discusses the partial implementation of the MPOOST data model using the Java programming language. It also contains a brief description of the implemented Java classes, and a short description of how the MPOOST Graphical User Interface functions.

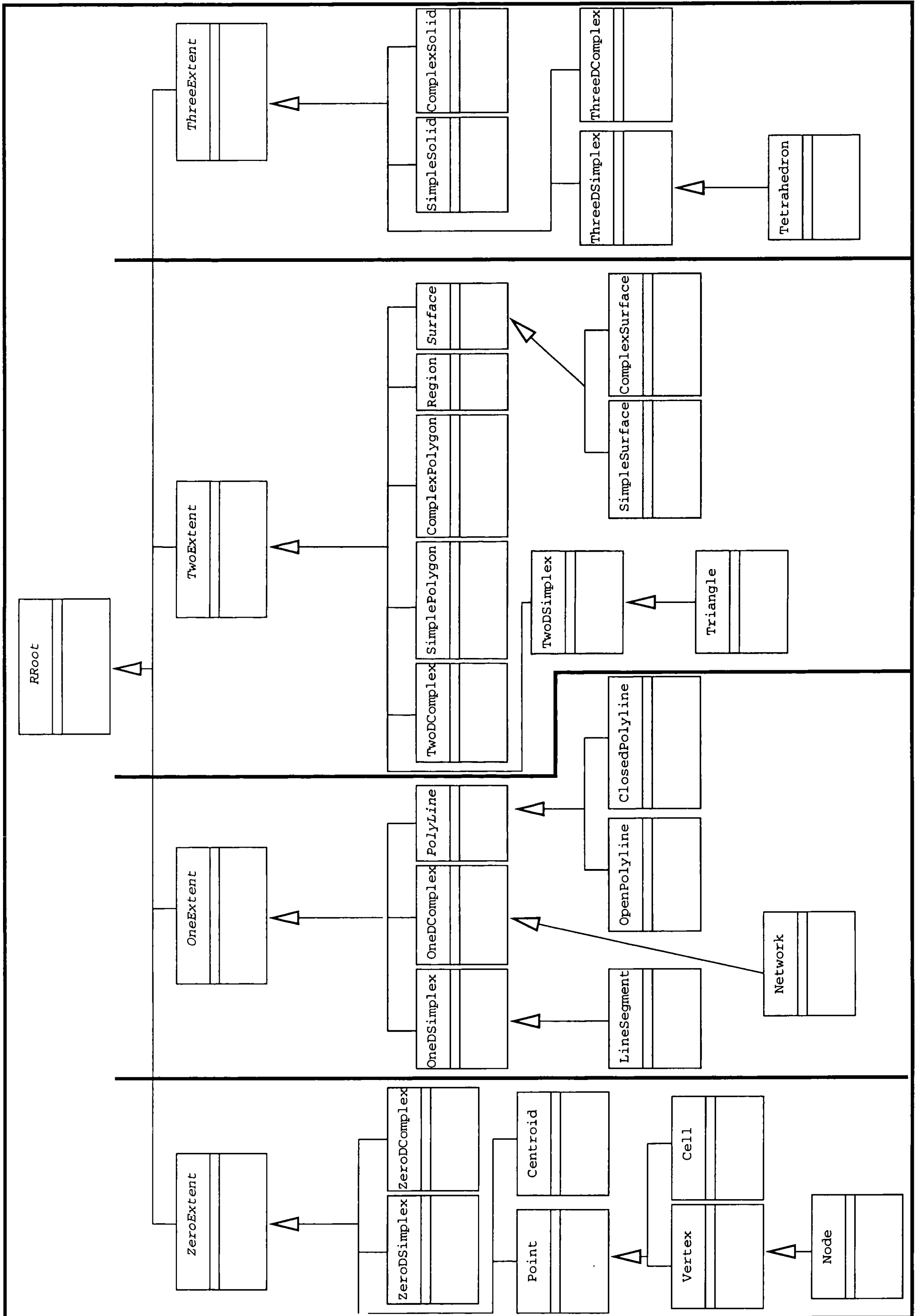


Figure 4.2: Package MPOOST.Spatial.RealWorld

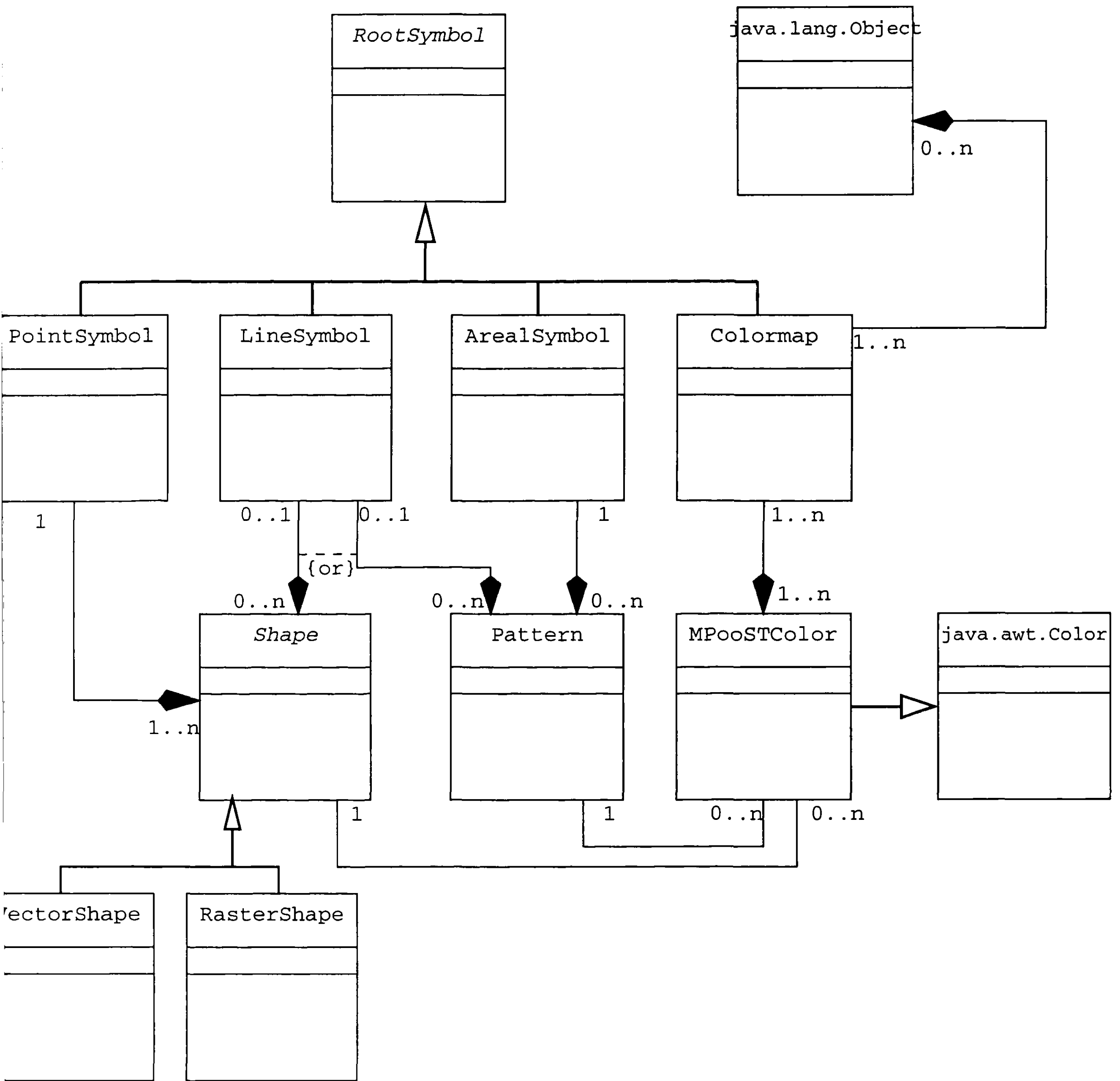


Figure 4.3: Package MPOOST.Representational.Symbol. For visual clarity, thick lines represent specialization and thin lines represent association.

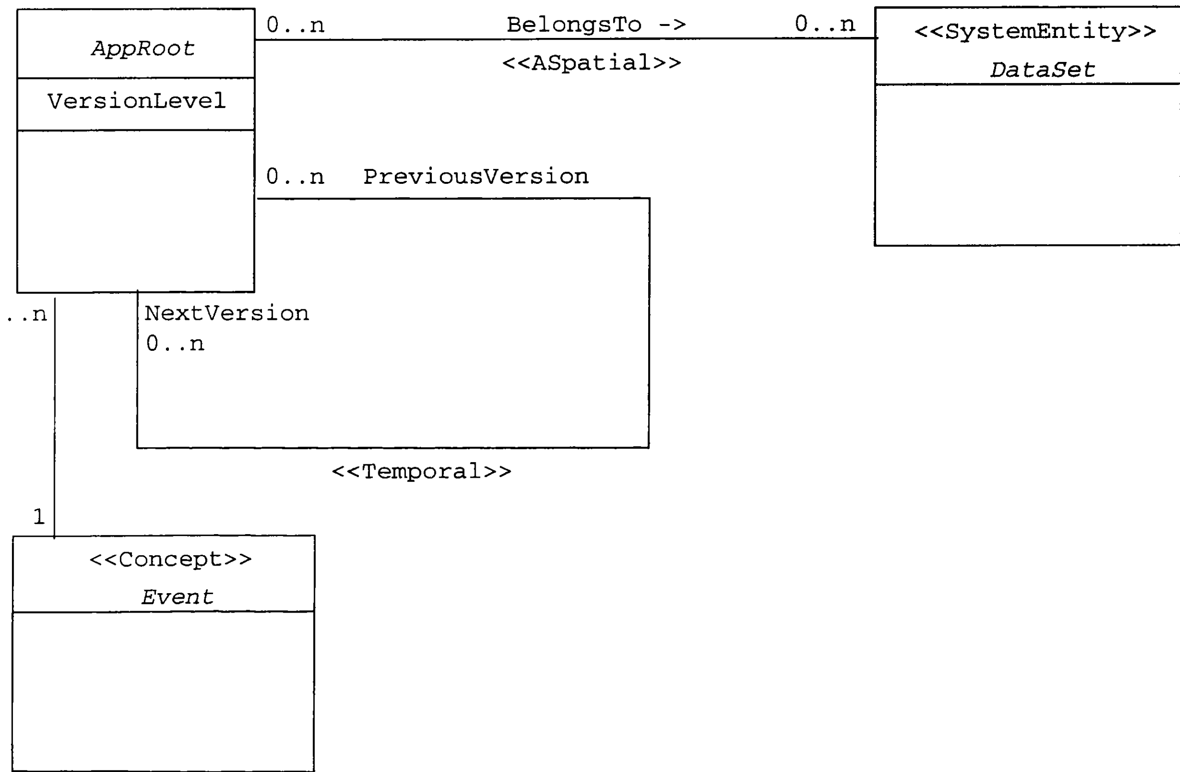


Figure 4.4: The parent class for all application domain classes.

Chapter 5

Implementation of the MPOOST model

5.1 Java as the implementation language

The implementation of the MPOOST model involves the usage of the Java programming language. The main reasons for choosing this specific environment are:

1. It is a pure object oriented programming language, where everything is treated like an object.
2. It implements the principle of "WORE" ("Write Once Run Everywhere"), since a Java compiler produces byte code that is platform independent.
3. The requirement of software interoperability is satisfied by the built-in functionality of Remote Method Invocation (RMI) among Java programs or by incorporating the CORBA standard so that it can co-exist with software written in other languages (*e.g.* C++).
4. Java applets, applications, servlets and components (beans) can be used effectively to deploy a distributed environment.

5. Awkward features found in other programming languages that are bug-prone have been eliminated (like operator overloading in C++), without amputating functionality to a great degree.

6. The Java development kit, as well as any client software used to execute Java applets locally, can be acquired free of charge, minimizing the financial cost of development.

7. A large amount of resources are available on the Internet where a developer can get valuable information and feedback regarding Java development.

8. The design of the model requires that associations found in high levels of the class schema must be inherited by sub-classes. This is perfectly handled by Java. Moreover, operations can receive as arguments not only the specified classes but their sub-classes as well. For example, a `Colormap` class relates a `Color` instance with the class `Object`, which is any type of instance. This means that any instance belonging to a class which is a sub-class of `Object` can exist in a `Colormap` instance.

However, problems that arise by implementing the model in Java are:

1. Multiple inheritance of classes is not supported. Although a class can implement more than one interface, it can only have a single parent class. This is considered to be a serious obstacle when implementing an object oriented data model that involves cases of multiple class inheritance. In this research, it has been addressed by eliminating multiple inheritance cases from the design stage[1].

2. Java applets although they are theoretically designed to work on any platform, practically this does not always happen, since new versions of Java require new versions of platform dependent software (*e.g.* browsers) that run the Java Virtual Machine (VM). This problem can be tackled if client platforms keep their Java VM software up-to-date.

[1]this is one of the possible solutions, see also section A.1 on page 308

3. Java does not have a built-in DBMS functionality, apart from object serialization, hence some external DBMS software is required. In this research, the problem of orthogonally persistent Java has been partially addressed by using either the experimental PJama classes, developed by the Computing Science Department of Glasgow University, as well as by using the ObjectStore PSE classes that provide the additional functionality of transactioning and data querying. The final persistence mechanism used is through Java's Object Serialization. It is considered to be the most appropriate solution since no additional server software is required.

4. Java does not provide functionality for GIS operations, hence a lot of development is required for including such capabilities. This would not be the case if other pure GIS software was used instead (e.g. LaserScan Gothic ADE). However, existing GIS software can be used if it implements any kind of interoperablity standard (e.g. CORBA).

Problems of code modifications due to continuous alterations and additions in requirements and design caused a delay and it proved to be rather time consuming, especially when modifications involved class interfaces (name, number of arguments or type of any argument), since this is considered to be the linkage mechanism that enables object communication.

It is worth mentioning that one of the most useful tools used in the development (which comes with the Java Development Kit) is the `javadoc` utility, that enables the integration of code and documentation in a single file. In this way, the developer can write code and document it simultaneously. Later, code documentation can be produced automatically on demand. This tool was used to produce the documentation of the code found in this chapter, as well as on the accompanying CD.

5.2 Code conventions

Package names, class names and class fields always begin with a capital letter (*e.g.* package `MPOOST.Spatial.RealWorld`, `Polyline`, `Polyline.Length`. Class methods begin always with a lower letter (*e.g.* `SimplePolygon.draw()`). Apart from the first letter, any combination of lower and upper case letters is used.

The detailed documentation of the packages and classes and the Java code files are included on the accompanying CD. A small part of the documentation has been printed on the following pages, regarding the implemented package index and the class hierarchy. Finally, as already mentioned, not all packages specified in the design stage (chapter four) have been implemented.

5.3 MPOOST Package Index

- package MPooST
- package MPooST.ASpatial
- package MPooST.ASpatial.Metadata
- package MPooST.ASpatial.Primitive
- package MPooST.Behavioural
- package MPooST.Behavioural.Utility
- package MPooST.GUI
- package MPooST.Representational
- package MPooST.Spatial
- package MPooST.Spatial.Cartographic
- package MPooST.Spatial.Cartographic.Geometry
- package MPooST.Spatial.Cartographic.MapComposition
- package MPooST.Spatial.Cartographic.Symbols
- package MPooST.Spatial.RealWorld
- package Spatial.Metadata.CoordinateSystems

5.4 Class Hierarchy

- class java.lang.Object
  - class java.awt.Component

- · class java.awt.Container
- · class com.sun.java.swing.JComponent
- · class com.sun.java.swing.JLabel
- · class MPooST.GUI.SampleTreeCellRenderer
- · class com.sun.java.swing.JPanel
- · class MPooST.GUI.DrawPanel
- · class java.awt.Panel
- · class java.applet.Applet
- · class com.sun.java.swing.JApplet
- · class MPooST.GUI.GUI
- · class java.awt.Window
- · class java.awt.Dialog
- · class com.sun.java.swing.JDialog
- · class MPooST.GUI.ListDialog

* class com.sun.java.swing.tree.DefaultMutableTreeNode
  - · class MPooST.GUI.DynamicTreeNode
* class com.sun.java.swing.tree.DefaultTreeModel
  - · class MPooST.GUI.SampleTreeModel
* class MPooST.GUI.DisplayManager
* class MPooST.MPooSTRoot
  - · class MPooST.ASpatial.ASpatialRoot
  - · class MPooST.ASpatial.Primitive.PrimitiveRoot
  - · class MPooST.ASpatial.Primitive.Constant
  - · class MPooST.ASpatial.Primitive.Rectangle
  - · class MPooST.Spatial.Cartographic.CartographicRoot
  - · class MPooST.Spatial.Cartographic.Geometry.OneExtent
  - · class MPooST.Spatial.Cartographic.Geometry.ClosedPolyline
  - · class MPooST.Spatial.Cartographic.Geometry.OpenPolyline
  - · class MPooST.Spatial.Cartographic.Geometry.TwoExtent
  - · class MPooST.Spatial.Cartographic.Geometry.ComplexPolygon
  - · class MPooST.Spatial.Cartographic.Geometry.Raster
  - · class MPooST.Spatial.Cartographic.Geometry.Region
  - · class MPooST.Spatial.Cartographic.Geometry.SimplePolygon
  - · class MPooST.Spatial.Cartographic.Geometry.Triangle
  - · class MPooST.Spatial.Cartographic.Geometry.ZeroExtent
  - · class MPooST.Spatial.Cartographic.Geometry.CPoint
  - · class MPooST.Spatial.Cartographic.Geometry.Cell
  - · class MPooST.Spatial.Cartographic.Geometry.Centroid
  - · class MPooST.Spatial.Cartographic.Geometry.LabelPoint
  - · class MPooST.Spatial.Cartographic.Geometry.Node
  - · class MPooST.Spatial.Cartographic.Geometry.Vertex

  · class MPooST.Behavioural.Utility.ClassSchema
  · class MPooST.Behavioural.Utility.Dataset
  · class MPooST.Behavioural.Utility.Debuger
  · class MPooST.Spatial.Cartographic.Symbols.SymbolRoot
  · class MPooST.Behavioural.Utility.Utility
- class MPooST.Behavioural.Utility.Math
- class MPooST.Behavioural.Utility.ObjectManager
- class MPooST.GUI.PanelDebug
- class PutClasses
- class system.utility.Query
- class MPooST.GUI.SampleData
- class MPooST.GUI.SampleTree
- class MPooST.Behavioural.Utility.SpatialObjectSet
- class MPooST.Behavioural.Utility.Tree
- class MPooST.Behavioural.Utility.TreeNode

5.5 The MPOOST Graphical User Interface

Besides the implementation of the data model itself, a graphical user interface (MPOOST GUI) has been deployed in order that the user can interact with the data model. The basic functionality of the interface includes:

- DXF import.
- Database creation.
- Database loading.
- Pan, zoom operations.
- Class selection for display.
- Class schema browser.

The MPOOST GUI is based on the Java Foundation Classes (JFC or mostly known as Swing) version 1.1 beta 2, which provide a rich library of visual gadgets (*e.g.* windows, panels, text areas, tables, lists*etc.*).

The full Java executables of the GUI as well as the Java Virtual Machine for Windows 95/98 can be found on the accompanying CD. Instructions on how to run the application may be also found.

5.6 MPOOST GUI Snapshots

A few snapshots of the application are included. It must be noted that the application looks different (and sometimes behaves in a slightly different manner) depending on the operating system being used. The following screen snapshots were obtained running MPOOST GUI in RedHat Linux environment (version 5.2, kernel version 2.0.37) with the KDE window manager (v1.1). These snapshots show the main functions of the application.

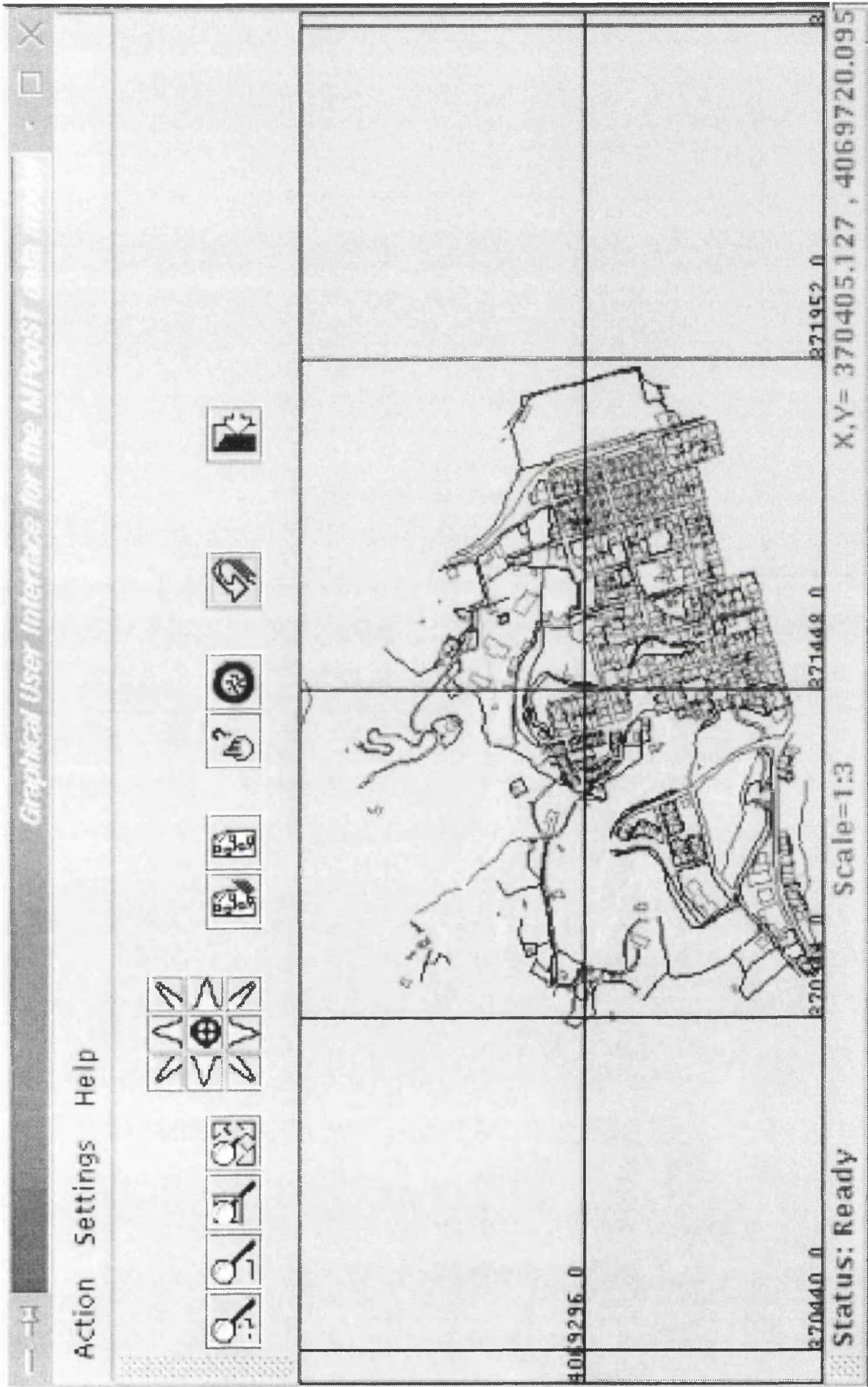


Figure 5.1: The main part of the MPOOST GUI.

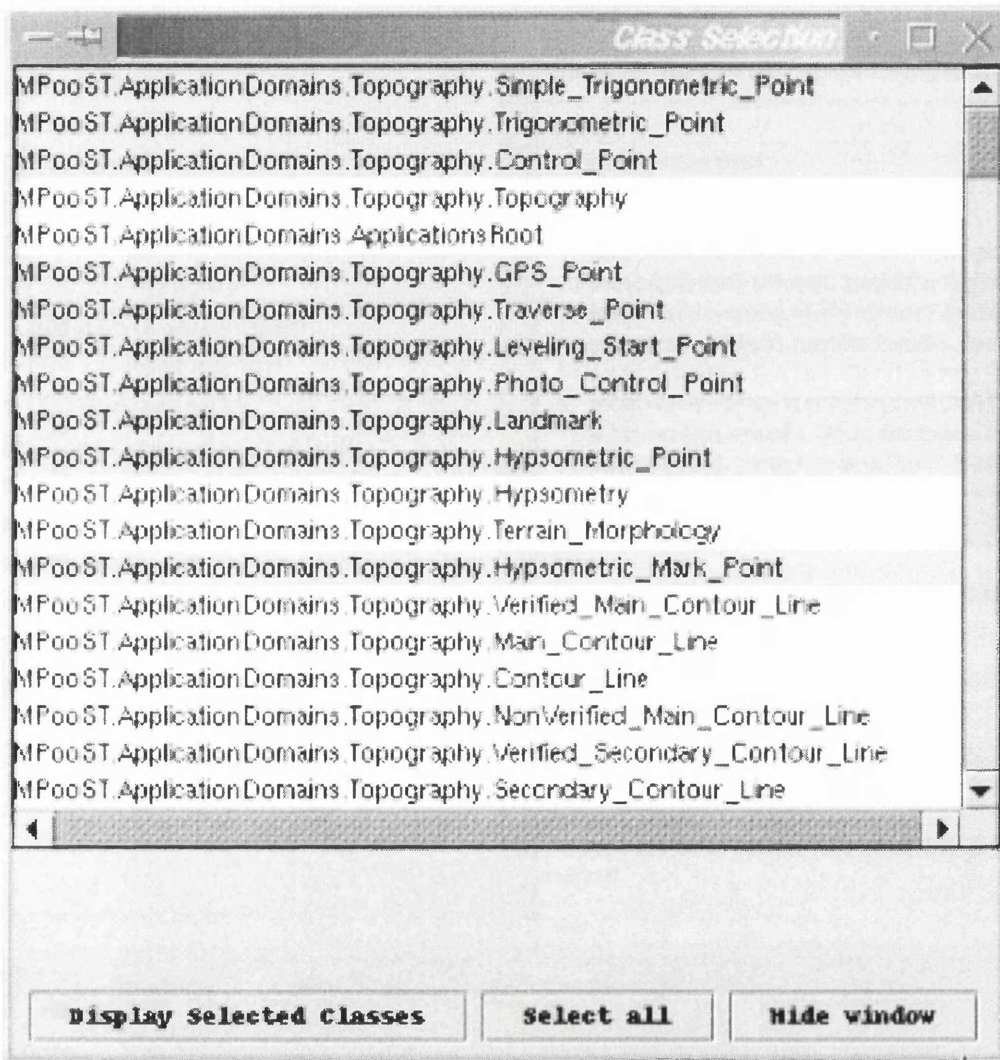


Figure 5.2: Class selection window

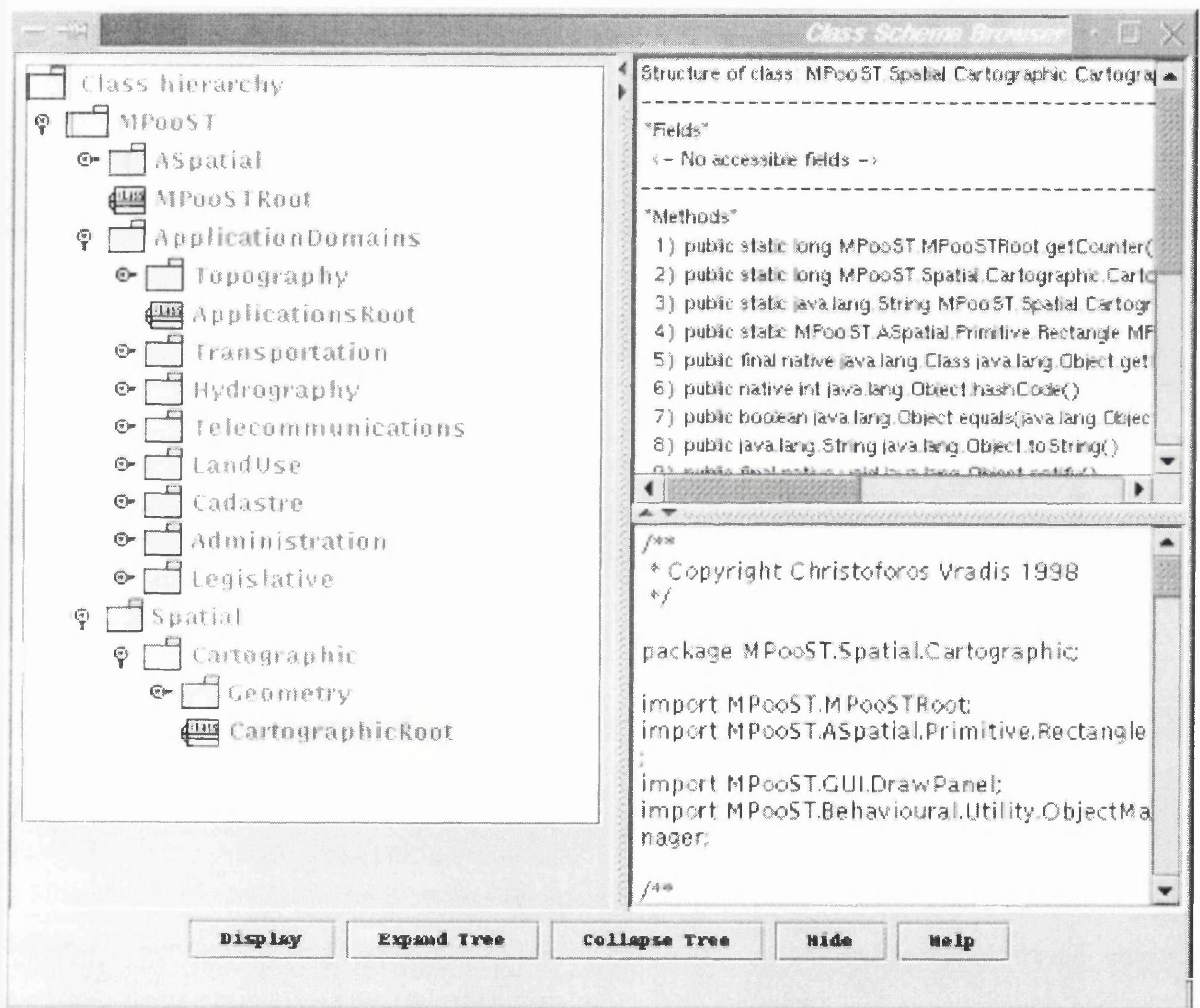


Figure 5.3: Class schema browser window

Chapter 6

Conclusions and further research

6.1 Data model assessment

The early stages of the research, and mainly the analysis phase, posed the requirements that had to be satisfied by the final outcome of the implementation. Since the implementation was not fully deployed, the design of the MPOOST data model is the key aspect that may determine the satisfaction degree of the requirements. The following list contains those requirements and an explanation on how they were satisfied by the design of the MPOOST data model and/or any other aspects involved.

1. **Effective manipulation of spatial, temporal, aspatial and composite information as well as relevant behavioural aspects by a single data model, in a single database.** In chapter one it has been proved that the relational model fails to efficiently address this issue. As it has been thoroughly discussed in chapter two, the only appropriate approach guaranteed to meet these requirements is the object-oriented approach (with features such as inheritance, polymorphism *etc.*). As it may be seen in chapter four, the design of the MPOOST data model is such that incorporates all the above categories of information (2D and 3D geometries,

miscellaneous temporal references, attribute data, metadata *etc.*). Each type of information is contained in autonomous objects and higher level composite real world entities are formed by aggregating the parts that are of interest. All objects include behavioural aspects involved and are stored in a single storage space in the conceptual level but may actually reside in different physical locations enabling the distributed environment. This holistic type of information embedding is essentially enabled by the conceptual application of the object as it is defined by the object-oriented methodology.

2. **Support for different spatial representations**. Different types of spatial representations may be incorporated in the model, and this was achievable because of:

   (a) The usage of *e*-complexes as the core spatial conceptual model, which were explicitly designed to handle a wide range of spatial complexity and spatial modelling approach, from "spaghetti"-type linework to 3D topologically enabled geometry with temporal references.

   (b) The direct implementation using the object-oriented computational model. The main features of this approach that enables the conceptual implementation of various spatial representations is modularity and aggregation. Objects may form autonomous modules depending on their conceptual classification while complex objects may be easily formed without having to alter any of the model's primary spatial components (*e.g.* points, lines and surfaces). Should a new object be modeled that does not belong to an already existing module, a new one may be created and implemented as a Java package.

Although the above features are valid, the MPOOST data model is not readily capable of such a support due to the lack of a model definition language to facilitate the composition of a custom spatial representation.

However, this is achievable through Java implementation of any custom spatial data model, meaning that a developer can conceptually construct an extension to the MPOOST model and implement this by using the Java programming language. For this to take effect, only the new code has to be compiled leaving intact the base MPOOST code, in most cases. An exception might be the case where two-way object references have to be implemented which requires the compilation of the initial code as well.

3. **Straightforward modelling of structure and behaviour of geo-entities that is not obstructed by the conceptual computational model.** In the MPOOST design this is considered to be supported very straightforwardly, since most concepts found in application domain models are inherently supported by the object-oriented methodology, *e.g.* uniqueness, hierarchy, specialization, generalization, aggregation, inheritance, polymorphism and behaviour. Moreover, the implementation was also extremely straightforward due to the object-oriented nature of the chosen programming language (Java).

4. **Support for a distributed and networked environment.** The final computational environment can be networked and distributed, since the implementation language can support both requirements without sacrificing the central management to the end-user, since objects appear to be stored in a single virtual space. Moreover, as discussed in chapter two, the distributional aspect of the database is relied not only on the object-oriented approach, according to which objects may reside on different network nodes, but also on the physical computational model, namely the interoperability standards involved (*e.g.* RMI) which enable the communication among objects on a physical level. Another less significant implementation feature that fosters the network environment is the TCP/IP [1] protocol support by the programming language (Java).

[1]Transmission Control Protocol/Internet Protocol

5. **Support for heterogeneous computational environments.** Regarding the variety in hardware platforms this has been effectively addressed by using Java as the implementation language. Java is the only programming language that claims (and proves) to be of the "Write-Once-Run-Everywhere" (WORM) type. In this way, it is not of concern what hardware platform the applet (or application) is executed since the underlying byte-code interpreter has been developed to run in a wide variety of processors and electronics (including portable devices such as mobile phones). Regarding the different software environments which can be used either to implement the design (*e.g.* C++) or simply act as the operating system wrappers, any kind of executable code can be part of the overall networked environment as a distributed object, which can be interoperable among them as long as they employ an interoperability standard such as RMI or CORBA. If no interoperability standards are involved, the Java code developed is guaranteed to execute in all software platforms supported by its manufacturer (*e.g.* Unix, Linux, Windows, BeOS, MacOS to name but a few). However, and as discussed in chapter five (Implementation) some problems might be experience due to various "bugs" found in some software versions of the Java bytecode compiler. It is only a matter of time (and of commercial benefits involved) until new bugfree versions will appear, which will fully justify the WORM property of Java.

6. **Support for expandability, code reuse and minimal structure changes necessary.** The object-oriented approach in the design of the MPOOST data model along with the selected interoperability standard reassures that the code written can be straightforwadly reused for other purposes, either within or outwith the MPOOST application. Moreover, the above combination guarantess the minimal changes in the structure of the model should any code changes occur in the code level. Additionally, the architecture of the model is both open and expandable because of its modular structure and behaviour.

It is the author's opinion that all of the satisfied requirements mentioned above suggest that the object oriented methodology, as an evolution of the relational approach, is of paramount importance for similar data modelling purposes, to the point that no other modeling approach can be used to satisfy all requirements at the same degree.

One of the current drawbacks in the data model implementation is the rather poor performance in terms of execution speed. As discussed in chapter two, object-oriented software platforms require robust and fast hardware components (processors, storage media *etc.*) in order that the overall performace can be satisfactional. It is mainly apparent when a voluminous amount of information is involved. This can only be tackled by using appropriate hardware equipment. However, the Java code itself may be optimized so as to minimize the time necessary to execute certain functions, such as queries. When indexing methods are involved they can effectively address this issue.

6.2 Future work

Although that the MPOOST data model has been partially implemented (due to the limited time available) however it is considered to be quite flexible since implemented classes act as the spatial component to the application domain classes. In order that the MPOOST data model is completely functional, future work should mainly involve the full implementation of the MPOOST model design, with priority to the following features:

1. Specialized algorithms to support the building and the maintenance of topological and temporal relationships, spatial analysis *etc.*

2. Modelling and optimization of complex user queries.

3. Spatial indexing of geometrical objects.

4. Creation of a CASE-like[2] tool and an appropriate graphical user interface to facilitate user interaction.

   Priority should be given to the following functions:

   (a) Design of customized class hierarchies, using either icons and diagrams or a textual interface.

   (b) Building of queries using visual components as well as an object query language.

5. Alternative exploitation of the functionality found in existing GIS platforms, like LasrScan's Gothic ADE.

Additionally, future research work might employ the design and development of a textual language that will be used to capture requirements from various application domains that are in need of geographical information support by the MPOOST data model. This language should be primarily textual as well as descriptive, and as close to a natural language (*e.g.* English) as possible so that it will be comprehensible by humans to the higher possible degree. This language should be well-defined, rule-based, formal and structured to be easily parsed and executed by the computer.

[2]Computer-Aided Software Engineering. It denotes the usage of a visual language to construct software components.

Bibliography

[1] A. A. Abdallah. *The design and implementation of a prototype geographic information system.* PhD thesis, University of Glasgow, 1990.

[2] K. K. Al-Taha. *Temporal reasoning in Cadastral systems.* PhD thesis, University of Maine, Orono, ME,USA, 1992.

[3] K. K. Al-Taha, R. T. Snodgrass, and M. D. Soo. Bibliography on spatio-temporal databases. *International Journal of Geographical Information Systems*, 8(1):95–103, 1993.

[4] J. Albrecht. Semantic Net of Universal Elementary GIS Functions. In *AUTOCARTO - 12th International Symposium on Computer Assisted Cartography*, pages 235–244, Charlotte, North Carolina, 1995. AUTOCARTO.

[5] J. Allen. Maintaining knowledge about temporal intervals. In *Communications of the ACM*, volume 26, pages 832–843, 1983.

[6] D. Arctur. Introduction to Object-Oriented GIS Technology. Technical paper, Laser-Scan Ltd, 1998.

[7] D. Arctur and P. Sargent. The future of Object-Oriented GIS Technology. Technical paper, Laser-Scan Ltd, Cambdridge, 1998.

[8] M. P. Armstrong, P. J. Densham, and D. A. Bennett. Object oriented locational analysis. In *Proceedings of the GIS/LIS Annual conference*, volume 2, pages 717–726, Orlando, 1989. ASPRS/ACSM.

[9] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. PS-ALGOL: A Language for Persistent Programming. In *Proceedings of the 10th Australian National Computer Conference, Melbourne, Australia*, pages 70–79, 1983.

[10] J. Aybet. The object-oriented approach: what does it mean to GIS users? *GIS Europe*, 3(3):38–41, 1994.

[11] O. Balovnev, M. Breunig, and A. B. Cremers. From GeoStore to GeoToolKit: The Second Step. In M. Scholl and A. Voisard, editors, *Proceedings of the 5th International symposium in Spatial databases, 1997 Jul, Berlin*, volume 1262 of *Lecture Notes In Computer Science*, pages 223–237, 1997.

[12] R. Barrera, A. Frank, and K. K. Al-Taha. Temporal Relations in Geographic Information Systems: A Workshop at the University of Maine. Technical Report 91-4, National Center for Geographic Information and Analysis (NCGIA), Santa Barbara, CA, 1991.

[13] L. G. Batten. National Capital urban planning project: Development of a three-dimensional GIS model. In *Auto-carto 9,Baltimore, MD*, pages 336–340, 1989.

[14] P. Batty. Smallworld GIS: Object-Orientation - some objectivity please!, http://www.smallworld.co.uk/, Accessed: Spring 1998. Smallworld Technical Paper No 7.

[15] L. Becker, A. Voigtmann, and K. H. Hinrichs. Developing applications with the object-oriented GIS-kernel GOODAC. In M. J. Kraak, M. Molenaar, and E. M. Fendel, editors, *Proceedings of the 7th International symposium in Spatial Data Handling (SDH 1996)*, Advances in GIS Research, no 2, pages 227–244, Delft, the Netherlands, 1996. Taylor and Francis, London.

[16] M. Bertrand. *Object-Oriented Software Construction.* Englewood Cliffs - Prentice Hall, 2nd edition, 1994.

[17] G. Birkhoff and J. Lipson. Heterogeneous Algebras. *Journal of combinatorial theory*, 8:115–133, 1970.

[18] Y. Bishr, M. Molenaar, and M. Radwan. A context sensitive model for sharing distributed geospatial information. In *Proceedings of the ISPRS Conference*, volume 32, pages 65–70, Stuttgart, 1998.

[19] R. B. Blaha, W. J. Premerlani, and J. E. Rumbaugh. Relational database design using an object oriented methodology. *Communications of the Association for Computing Machinery*, 31(4):414–427, 1988.

[20] P. Bofakos. *An object-oriented approach to geo-referenced data modelling.* PhD thesis, Keele University, 1994.

[21] G. Booch. *Object Oriented Analysis and design.* Benjamin/Cummings, Reading Mass. Harlow, 2nd edition, 1994.

[22] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide.* Object Technology Series. Addison Wesley, 1999.

[23] I. Bracken and C. Webster. Towards a typology of geographical information systems. *International Journal of Geographic Information Systems*, 3(2): 137–152, 1989.

[24] K. Brassel, F. Bucher, E. M. Stephan, and A. Vckovski. *Completeness*, chapter 5, pages 81–108. Volume 1 of , Guptill and Morisson [70], 1995.

[25] M. Breunig. *Integration of Spatial Information for Geo-Information Systems*, volume 61 of *Lecture Notes in Earth Sciences.* Springer-Verlag, Berlin, 1996.

[26] M. Breunig, T. Bode, and A. B. Cremers. Implementation of elementary geometric database operations for a 3D-GIS. In T. Waugh and R. Healey,

editors, *Proceedings of the 6th symposium in Spatial Data Handling, Edingburgh*, volume 1 of *Advances in GIS research*, pages 604–617, London, 1994. Taylor and Francis.

[27] M. L. Brodie. On the development of data models. In M. L. Broie *et al.*, editors, *On Conceptual Modeling, Perspectives from Artificial Intelligence, Databases and Programming Languages.* Springer Verlag, New York, 1984.

[28] M. Bundock. Integration of Object Oriented CASE with GIS. In *Environmental, Urban and Social planning: the winning vision. 21st Annual Conference, Adelaide, Australia*, pages 537–543, Sydney, 1993. Australasian Urban and Regional Information Systems Association., AURISA.

[29] P. A. Burrough and A. U. Frank, editors. *Geographical Objects with Indeterminate boundaries.* Taylor and Francis, London, 1996.

[30] P. A. Burrough and R. A. McDonell. *Principles of Geographical Information Systems.* Spatial Information Systems and Geostatistics series. Oxford University Press, Oxford, 2nd edition, 1998.

[31] H. W. Calkins and D. F. Marble. The transition to automated production cartography: design of the master cartographic database. *American Cartographer: Journal of American Congress on Surveying and Mapping*, 14 (2):105–119, 1987.

[32] L. Cardelli and P. Wegner. On Understanding Type, Data Abstraction and Polymorphism. In *ACM Computing Surveys*, volume 17, pages 471–522. Baltimore MD, 1985.

[33] A. Chance, R. G. Newell, and D. G. Theriault. Smallworld GIS: An overview of Smallworld Magik. http://www.smallworld.co.uk, http://www.smallworld.co.uk/ Accessed: 1998. Smallworld Technical Paper No 5.

[34] A. Chance, R. G. Newell, and D. G. Theriault. Smallworld GIS: An Object-Oriented GIS - Issues and Solutions, http://www.smallworld.co.uk/ Accessed: Spring 1998. Smallworld Technical Paper No 3.

[35] A. Choi and W. S. Luk. Using an object-oriented database system to construct a spatial database kernel for GIS applications. *Computer System Science and Engineering*, 7:100–121, 1992.

[36] N. R. Chrisman. Topological information systems for geographic representation. In *Proceedings of Second International Symposium on Computer Assisted Cartography (Auto Carto 2)*, pages 346–351, Falls Church, 1975. ASPRS/ACSM.

[37] N. R. Chrisman. The role of quality information in the long term functioning of a geographic information system. *Cartographica*, 21(2):79–87, 1984.

[38] E. Codd. Data Models in Database Management. *SIGMOD RECORD*, 11 (2):112–114, 1981.

[39] H. Couclelis. Beyond the raster-vector debate in GIS. In A. U. Frank, I. Campari, and U. Formentini, editors, *Theories of Spatio-Temporal Reasoning in Geographic Space*, volume 639 of *Lecture Notes in Computer Science*, pages 65–77. Springer-Verlag, Berlin, 1992.

[40] B. J. Cox. *Object Oriented Programming.* Addison-Wesley, Reading Mass. Wokingham, 1986.

[41] Dahlberg, J. D. McLaughlin, and Niemann, editors. *Developments in Land Information Management.* Institute for Land Information USA, 1989.

[42] P. F. Dale and J. D. McLaughlin. *Land Information Management. An introduction with special references to cadastral problems in Third World countries.* Clarendon Press, Oxford, 1988.

[43] B. David, L. Raynal, G. Schorter, and V. Mansart. *GeO*$_2$: Why objects in a geographical DBMS? In D. Abel and B. Ooi, editors, *Advances in Spatial Databases. Proceedings of SSD 1993*, volume 692 of *Singapore Lecture notes in Computer Science*, pages 264–276, Berlin, 1993. Springer-Verlag.

[44] L. De Floriani, P. Marzano, and E. Puppo. Spatial queries and data models. In A. U. Frank and I. Campari, editors, *Spatial Information Theory*, volume 716 of *Lecture Notes in Computer Science*, pages 113–138. Springer-Verlag, Berlin, 1993.

[45] R. DeCosta. Object database technology in GIS. *Mapping Awareness and GIS in Europe*, 7(3):44–55, 1993.

[46] H. M. Deitel and P. J. Deitel. *C++: How To Program.* Prentice Hall International Editions, Englewood Cliffs, N.J., 1st edition, 1995.

[47] H. M. Deitel and P. J. Deitel. *Java: How To Program.* Prentice Hall editions, Upper Saddle River, N.J., 2nd edition, 1998.

[48] J. E. Drummond. *Positional Accuracy*, chapter 3, pages 31–58. Volume 1 of , Guptill and Morisson [70], 1995.

[49] M. E. Easterfield, N. R. G., and D. G. Theriault. Smallworld GIS: Version Management in GIS - Applications and Techniques, http://www.smallworld.co.uk/ Accessed: Spring 1998. Smallworld Technical Paper No 4.

[50] J. Egenhofer and R. Colledge. *Spatial and Temporal Reasoning in Geographic Information Systems.* Oxford University Press, New York Oxford, 1998.

[51] J. M. Egenhofer and R. Golledge. Time in Geographic Space. Report on the Specialist Meeting of Research Initiative 10 94-9, National Center for Geographic Information and Analysis, Santa Barbara, CA, 1994.

[52] M. Egenhofer and K. K. Al-Taha. Reasoning about Gradual Changes of Topological Relationships. In A. Frank, I. Campari, and U. Formentini, editors, *Theory and Methods of Spatio-Temporal Reasoning in Geographic Space*, volume 639 of *Lecture Notes in Computer Science*, pages 196–219. Springer-Verlag, Pisa, Italy, 1992.

[53] M. Egenhofer and A. Frank. PANDA: An Object-Oriented Database Based On User-Defined Abstract Data Types. Technical Report 67, University of Maine, Department of Civil Engineering, Surveying Engineering Program, Orono, ME, 1986.

[54] M. Egenhofer and A. Frank. Object-Oriented Software Engineering Considerations for Future GIS. In *International Geographic Information Systems (IGIS) Symposium, Baltimore*, 1989.

[55] M. Egenhofer, D. Mark, and J. Herring. The 9-Intersection: Formalism and Its Use for Natural-Language Spatial Predicates. Technical Paper 94-1, National Center for Geographic Information and Analysis (NCGIA), Santa Barbara, CA, 1994.

[56] M. J. Egenhofer. A formal definition of binary topological relationships. In W. Litwin and H. Schek, editors, *Proceedings of the Third International Conference on Foundations of Data Organization and Algorithms (FODO), in Paris*, volume 367 of *Lecture Notes in Computer Science*, pages 457–472, Berlin, 1989. Springer-Verlag.

[57] M. J. Egenhofer and A. Frank. Object-oriented databases: Database requirements for GIS. In *Proceedings of the International GIS symposium: The Research Agenda*, volume 2, pages 189–211, Washington DC, 1987. US Government Printing Office.

[58] M. J. Egenhofer and A. Frank. Object oriented modeling in GIS: Inheritance and propagation. In *Proceedings of Auto Carto 9*, pages 588–598, Baltimore, Maryland, U.S, 1989.

[59] M. J. Egenhofer and A. Frank. Object-oriented modeling for GIS. *Journal of the Urban and Regional Information Systems Association*, 4:3–19, 1992.

[60] M. J. Egenhofer, A. Frank, and J. Jackson. A topological data model for spatial databases. In O. Gunther and T. Smith, editors, *Proceedings of SSD'89: Design and Implementation of Large Spatial Databases*, volume 409 of *Lecture Notes in Computer Science*, pages 271–286, Santa Barbara, CA, 1989. Springer-Verlag.

[61] M. J. Egenhofer and A. U. Frank. LOBSTER: combining AI and database techniques for GIS. *Photogrammetric Engineering and Remote Sensing*, 56 (6):919–926, 1990.

[62] M. J. Egenhofer and R. G. Golledge, editors. *Spatial and temporal reasoning in geographic information systems.* Spatial Information Systems. Oxford University Press, 1998.

[63] R. A. Elmasri and S. B. Navathe. *Fundamentals of database systems.* World Student Series. Benjamin/Cummings Publishing Company Inc., Redwood City, Calif. Wokingham, 2nd edition, 1994.

[64] A. Frank and M. Egenhofer. Object-Oriented Databases for GIS. In *4th International Symposium on Spatial Data Handling*, GIS/LIS, San Antonio, TX, 1988.

[65] A. U. Frank. An object-oriented, formal approach to the design of cadastral systems. In M. J. Kraak, M. Molenaar, and E. M. Fendel, editors, *Proceedings of the 7th International symposium on Spatial Data Handling (SDH'96)*, volume 2 of *Advances in GIS Research*, pages 245–262, Delft, the Netherlands, 1996. Taylor and Francis, London.

[66] A. U. Frank and M. J. Egenhofer. Computer cartography for GIS: an object-oriented view on the display transformation. *Computers and Geosciences*, 8(8):975–987, 1992.

[67] G. Golod and J. Shochat. Theoretical and Practical Issues in Developing a Nationwide Cadastral GIS. In *Proceedings of the Thirteenth Annual ESRI User Conference*, volume 2, pages 171–177, Palm Springs, CA, 1993.

[68] M. Goodchild. *Attribute accuracy*, chapter 4. Volume 1 of , Guptill and Morisson [70], 1995.

[69] S. Guptill and M. Stonebraker. The Sequoia 2000 approach to managing large spatial object databases. In E. Corwin and D. Cowen, editors, *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 642–651, Columbus, OH, 1992. International Geographical Union.

[70] S. C. Guptill and J. L. Morisson, editors. *Elements of Spatial Data Quality.* Pergamon Press, Oxford, 1995.

[71] T. Hamre. An object-oriented conceptual model for measured and derived data varying in 3D space and time. In T. Waugh and R. Healey, editors, *Proceedings of the 6th symposium in Spatial Data Handling, Edinburgh*, volume 2 of *Advances in GIS research*, pages 868–881, London, 1994. Taylor and Francis.

[72] P. Hardy and P. Woodsford. Mapping with live features: Object-oriented representation. White paper, Laser-Scan ltd., http://www.laserscan.com/papers/livefeatures.htm Accessed: 2nd December 1998.

[73] F. Harvey. Improving Multi-Purpose GIS Design: Participative Design. In S. Hirtle and A. Frank, editors, *Spatial Information Theory: a theoretical basis for GIS, International Conference COSIT'97, Laurel Highlands, Pensylvania, USA*, volume 1329 of *Lecture Notes in Computer Science*, pages 314–328, Berlin London, 1997. Springer-Verlag editions.

[74] N. W. J. Hazelton. *Integrating time, dynamic modelling and geographical*

information systems: Development of four-dimensional GIS. PhD thesis, University of Melbourne, Australia, 1991.

[75] J. Herring. TIGRIS: A data model for an object-oriented geographic information system. *Computers and Geosciences*, 18:443–452, 1991.

[76] C. M. Hoffman. *Geometric and solid modelling - An introduction.* Morgan-Kaufmann, San Mateo, 1989.

[77] K. Hornsby and M. J. Egenhofer. Qualitative Representation of Change. In S. Hirtle and A. Frank, editors, *Spatial Information Theory: a theoretical basis for GIS, Proceedings of the International Conference COSIT'97, Laurel Highlands, Pensylvania, USA*, Lecture Notes In Computer Science, Berlin London, 1997. Springer-Verlag editions.

[78] G. Hunter and I. Williamson. The development of a historical digital cadastral database. *International Journal of Geographical Information Systems*, 4(2):169–179, 1990.

[79] A. N. S. Institute. Object Oriented Database Task Group Final Report. Technical Report X3/SPARC/DBSSG OODBTG, American National Standards Institute, 1991.

[80] I. Jacobson, M. Christenson, and G. Overgaard. *Object Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley Publishing Company, Wokingham, England, 1992.

[81] C. Jensen. A consensus glossary of temporal database concepts. *Association for Computing Machinery SIGMOD Record*, 23(1):52–64, 1994.

[82] C. Jones. *Geographical Information Systems and Computer Cartography.* Longman editions, Harlow, 1997.

[83] W. Kainz, M. J. Egenhofer, and I. Greasley. Modelling Spatial relations and operations with partially ordered sets. *International Journal of Geographical Information Systems*, 7(3):215–229, 1993.

[84] A. Kay. Computer Software. *Scientific American*, 251(3):53–59, 1984.

[85] G. Kendrick and P. Batty. Smallworld GIS: Use of an integrated CASE tool for GIS customisation. Technical Report 11, Smallworld, http://www.smallworld.co.uk/ Accessed: Spring 1998. Smallword Technical Paper No 11.

[86] S. Khoshafian. *Object Oriented Databases.* John Wiley editions, New York Chichester, 1993.

[87] S. Khoshafian and R. Abnous. *Object Orientation: Concepts, Languages, Databases and Interfaces.* Wiley editions, New York, 1990.

[88] W. Kim. *Introduction to Object-Oriented Databases.* The MIT Press, Cambridge, Mass. London, 1990.

[89] D. Kjerne. Modelling location for cadastral maps using an object oriented computer language. In *Papers from the 1986 Annual Conference of the Urban and Regional Information Systems Association*, volume 1, pages 174–189, Denver, Colorado, 1986. URISA.

[90] G. Kosh and K. Loney. *Oracle, The complete reference.* Oracle Press, 1995.

[91] G. Kosters, B. Pagel, and H. Six. GIS-application development with *GeoOOA*. *International Journal of Geographical Information Science*, 11 (4):307–335, 1997.

[92] G. Langran. Accessing spatiotemporal data in a temporal GIS. In *Auto-Carto 9 Symposium*, pages 191–198, Baltimore, MD, 1989.

[93] G. Langran. Dilemmas of implementing a temporal GIS. In *Proceedings of the International Cartographic Association*, pages 547–555, Bournemouth,UK, 1991.

[94] G. Langran. *Time in Geographic Information Systems.* Technical issues in Geographic Information Systems. Taylor and Francis, London, 1992.

[95] G. Langran. Issues of implementing a spatiotemporal system. *International Journal of Geographical Information Systems*, 7(4):305–314, 1993.

[96] R. Laurent, D. Benoit, and S. Guylaine. Building an OOGIS prototype: Some experiments with GeO_2. In *AUTOCARTO 12-Twelfth International Symposium on Computer-Assisted Cartography*, volume 4, pages 137–146, Charlotte, North Carolina, 1995.

[97] R. Laurini and D. Thompson. *Fundamentals of Spatial Information Systems.* The A.P.I.C. series, number 37. Academic Press Ltd, London, 1992.

[98] P. Leach, B. Stumpf, J. Hamilton, and P. Levine. UIDS as internal names in a distributed file system. In *Proceedings of the First Symposium On Principles Of Distributed Computing*, Ottawa, Canada, 1982.

[99] B. Liskov. Data Abstraction and Hierarchy. *SIGPLAN Notices*, 23(5), 1988.

[100] H. J. Lucas. *The Analysis, Design, and Implementation of Information Systems.* McGraw-Hill, 4th edition, 1990.

[101] N. Mattos, K. Meyer-Wegener, and B. Mitschang. Grand tour of concepts for object-orientation from a database point of view. *Data and Knowledge Engineering*, 9:321–352, 1993.

[102] J. D. McLaughlin and S. Nichols. Parcel based land information systems. *Surveying and Mapping*, 47(1):11–30, 1987.

[103] P. Milne, S. Milton, and J. Smith. Geographical object-oriented databases: A case study. *International Journal of Geographical Information Systems*, 7(1):39–56, 1993.

[104] H. Moellering. A draft proposed standard for digital cartographic data. Technical report, National Committee for Digital Cartographic Standards, 1987.

[105] E. Moise. *Geometric Topology in Dimension 2 and 3.* Springer, New York, 1977.

[106] M. Molenaar. Object hierarchies and uncertainty in GIS or why is standardisation so difficult? *Geo-Informations-System*, 6(4):22–28, 1993.

[107] E. Moss. *Nested Transactions: An approach to Reliable Distributed Computing.* PhD thesis, M.I.T., Cambridge, MA, 1981.

[108] National Research Council. *Need for a multipurpose cadastre.* National Academy Press, Washington D.C., USA, 1980.

[109] NCDCDS, FICCDC-SWG, and DCDSTF. The proposed standard for digital cartographic data. *The American Cartographer*, 15(1):11–140, 1988.

[110] R. Newell and D. Theriault. Smallworld GIS: Ten difficult problems in building a GIS . Technical Report 1, Smallworld, http://www.smallworld.co.uk/ Accessed: Spring 1998.

[111] V. Oliver. Digital Cadastral Mapping: Design and development considerations. Master's thesis, Department of Surveying Engineering, University of New Brunswick, Fredericton, N.B., Canada, 1985.

[112] P. v. Oosterom. *Reactive Data Structures for Geographic Information Systems.* Spatial Information Systems. Oxford University Press, Oxford, New York, 1993.

[113] E. Oxborrow and Z. Kemp. An Object-Oriented Approach to the Management of Geographical Data. In *Managing Geographical Information Systems and Databases.* Lancaster University, 1989.

[114] F. I. Pearson. *Map projection methods.* Sigma Scientific, Inc., Blacksburg, Virginia, USA, 1984.

[115] J. Peckham and F. Marianski. Semantic data models. *ACM Transactions on Database Systems*, 20(153), 1988.

[116] D. Peuquet. A conceptual framework and comparison of spatial data models. *Cartographica*, 21:66–113, 1984.

[117] D. Pullar and M. Egenhofer. Towards formal definitions of spatial relationships among spatial objects. In *Proceedings of the 3rd International Symposium on Spatial Data Handling*, pages 225–242, Sydney, Columbus, OH, 1988. International Geographical Union.

[118] L. Raynal, B. David, and G. Schorter. Building an OOGIS Prototype: Some Experiments with GeO_2. In *12th International symposium in Computer-assisted cartography, Charlotte, NC*, pages 137–146, 1995.

[119] C. Roussilhe and J. Peloux. OGQL: Object Geographic Query Language for Object GIS. In M. Rumor, R. McMillan, and H. F. L. Ottens, editors, *Proceedings of the JEC-GI 1996, 2nd Joint European conference, Barcelona, Spain*, pages 53–62, 1996.

[120] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modelling and Design.* Prentice Hall editions., London, 1991.

[121] R. Snodgrass. Temporal databases. In A. Frank, I. Campari, and U. Formentini, editors, *Theories of Spatio-temporal reasoning in Geographic Space*, volume 639 of *Lecture Notes in Computer Science*, pages 22–64. Springer-Verlag, Berlin, 1992.

[122] P. Story. *Designing spatio-temporal information systems: an object-oriented approach.* PhD thesis, Keele University, 1996.

[123] L. Valet. *An object-oriented approach to the modelling of time-varying spatial data.* PhD thesis, Leeds University, 1996.

[124] M. Wachowicz. *Integrating space and time in an object-based GIS: a case study of public boundary evolution.* PhD thesis, University of Edinburgh, 1996.

[125] M. J. White. Technical requirements and standards for a multipurpose geographic data system. *American Cartographer*, 11(1):15–26, 1984.

[126] R. Whittington. *Database Systems Engineering.* Oxford University Press, Oxford, 1988.

[127] P. Woodsford. Object Orientation, Cartographic Generalisation and Multi Product Databases. White paper, Laser-Scan Ltd., http://www.lsl.co.uk/papers/cartogen.htm Accessed: 2nd December 1998.

[128] P. Woodsford. The significance of Object Orientation for GIS, http://www.lsl.co.uk/papers/ooforgis.htm Accessed: 2nd December 1998.

[129] P. Woodsford and D. Arctur. Data conversion and update in the Object Paradigm. Technical report, Laser-Scan Ltd, http://www.lsl.co.uk/papers/datacon.htm Accessed: 2nd December 1998.

[130] M. Worboys, H. Hearnshaw, and D. Maguire. The IFO Object-Oriented Data Model. In *Managing Geographical Information Systems and Databases.* Lancaster University, 1989.

[131] M. F. Worboys. A generic model for planar geographic objects. *International Journal of Geographical Information Systems*, 6(5):353–372, 1992.

[132] M. F. Worboys. Object oriented models of spatio-temporal information. In *Proceedings LIS/GIS 92*, pages 824–835, San Jose California, 1992.

[133] M. F. Worboys. A unified model of spatial and temporal information. *Computer Journal*, 37(1):26–34, 1994.

[134] M. F. Worboys. Object oriented approaches to geo-referenced information. *International Journal of Geographical Information Systems*, 8(4):385–399, 1994.

[135] M. F. Worboys. *GIS. A computing perspective.* Taylor and Francis, London, 1995.

[136] M. F. Worboys. *A generic model for spatio-bitemporal geographic information*, chapter 2. Volume 1 of , Egenhofer and Golledge [62], 1998.

[137] M. F. Worboys. Modelling changes and events in dynamic spatial systems with reference to socio-economic units. In *ESF GISDATA Conference on Modelling Change in Socio-Economic Units*, 1998.

[138] M. F. Worboys and P. Bofakos. A canonical model for a class of areal spatial objects. In D. Abel and B. Ooi, editors, *Advances in Spatial Databases, Proceedings of SSD 1993, Singapore*, volume 692 of *Lecture Notes in Computer Science*, pages 36–52, Berlin, 1993. Springer-Verlag.

[139] M. F. Worboys, H. Hearnshaw, and D. Maguire. Object-oriented data and query modeling for geographical information systems. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, pages 679–689, Zurich, 1990.

[140] M. F. Worboys, H. M. Hearnshaw, and D. J. Maguire. Object-oriented data modelling for spatial databases. *International Journal of Geographical Information Systems*, 4(4):369–383, 1990.

[141] M. F. Worboys, K. Mason, and B. R. P. Dawson. The object-based paradigm for a geographical database system: modelling, design and implementation issues. In P. Mather, editor, *Geographical Information Handling - Research and Applications*, pages 91–102. John Wiley editions, Chichester, 1993.

[142] H. Yang *et al.* Implementation of Object Oriented GIS Using Formal Data Structure with Planar Topology-Some Considerations. *International archives of photogrammetry and remote sensing*, 29(3):151–154, 1992.

Appendix A

Object Oriented Programming Languages

A.1 Smalltalk

Smalltalk is an object oriented programming language that was first created in 1972 by the members of the Xerox Palo Alto Research Center Learning Research Group as a software component of the Dynabook project. Smalltalk's main influence was Simula. It is a pure object oriented language and everything is viewed as an object. Smalltalk is responsible for influencing many other successors OO languages. There are five identifiable releases of Smalltalk, namely Smalltalk-72, -74, -76, -78, and the most current version Smalltalk-80, which has been ported to a variety of machine architectures. Smalltalk is built upon the idea that everything is an object and objects communicate via message passing.

A.2 PS-ALGOL

PS-ALGOL is the first language in a family that introduces the concept of persistence as a property of data. It originates from the procedural language ALGOL. The idea that it is based on is that all data with no regard to their type should

have equal rights for persistence or transience. It has been developed jointly at the University of Glasgow and St. Andrews and it is based on three principles: data type completeness, abstraction and correspondence.

A.3 C++

C++ is an object oriented programming language, which was first designed by Bjarne Stroustrup of AT&T Laboratories in the early 1980's. It originates from a language called C with Classes, which in turn was influenced by the languages C and Simula. The major characteristic of C++ is that it adds features like type-checking, overloaded functions, and mainly object-oriented concepts not found in its ancestor, the C language. No matter how many different development environments have been released so far (like Visual C++ for Microsoft Windows), language artifacts, features and mechanisms remain the same. It is only the provided class libraries that differ, which are used to support specific computing environments (mostly operating systems) wherein the final application will be executed. Moreover to operating systems, C++ may be used as a development language in many software packages. C++ has gained popularity for the past two decades, and it is being used in most large-scale computing projects, because of the rich functionality it offers.

A.4 Java

Java is the most recent pure object oriented programming language available, known to introduce the concept of "write once, run everywhere". It originally came from a OO language called Oak, part the Green research project, of Sun Microsystems, a language based on C and C++. Java was first introduced in 1993, when the World Wide Web gained significant popularity and ever since it is being used vastly within the WWW, since it is tool that introduced real portable programming as well as dynamic contents in pages and sites. Unlike

with the other languages examined that were developed within an academic environment, Java was created mainly for commercial reasons. Java is based on C++, but more compact as many of the complex and awkward features of C++ (like pointers and operator overloading)[1] have been eliminated, mainly for simplification reasons. Although the unique features it offers, many authors believe that it still remains in a developing stage and it cannot be used in the same way that C++ has been for the past two decades. However, it remains perhaps the only language whose code can run in almost all operating systems available, without any alteration necessary. Java is used to create either *applets*, that is programs that are runnable only through a web browser, or *applications*, which are executed in the operating system, without the context of a browser. In the first case many security constraints apply, therefore restrictring the degree up to which Java applets can access local resources (like hard disks). This is not the case with applications, where full-blown file system functionality is present. A Java compiler creates bytecodes (.class files) as a mediator bewteen the actual code written in plain ASCII format(.java files) and the machine code eventually executed by a computer. This is mainly because the executable code remains independent of the target computing environment (namely the operating system) and it is being produced "on-the-fly', every time that a class is loaded. It is obvious though that this imposes extra computation time, due to the required class loading. The most recent Java version is 1.2, which has a built-in advanced set of GUI classes called the Java Foundation Classes (JFC or Swing). One of the major shortcomings in Java is that it does not support multiple class inheritance, meaning that a class can have only one parent class. This is beacuse Java was designed to be a simple and easy to use OO programming language. However, a class can realize more than one interfaces, and interfaces can have more than one parent interfaces. design A few workarounds can be suggested so as to overcome the problem occured when the object-oriented design involves multiple inheritance (we assume for simplicity reasons that the higher level design is

[1] although these features make C++ the most powerful programming language

documented in UML):

- Modeling of multiple parent classes with Java interfaces instead of using Java classes. This approach denotes that parent classes which are modelled via interfaces, will lack of variable attributes and code implementation, since interfaces include constant attribute values and method declarations only.

- Linear re-ordering of the multiple parent classes so that multiple inheritance is eliminated. This solution cannot be applied in all cases, especially when parent classes belong to different packages (figure A.1-III).

- Modification of the higher level OO design so that it does not involve multiple inheritance situations. This denotes that across the UML class diagrams, all classes should have one at the most parent classes. If there are cases where multiple parents are necessary, then one specialization relationship can be kept, while the remaining can be turned into simple association or aggregation relationships. This solution can be used only if it has been anticipated in the early stages of OO design. Access to ex-parent class members that are no longer inherited is still feasible via the association or aggregation relationship. A visual example is given in figure A.1-II.

- A more extreme and complicated solution, but very effective, since it does not require any higher level design alteration, is when another programming language is used, additionally to Java, and an interoperability standard has been incorporated (such as CORBA). Then a Java class can inherit from a parent class, which in turn is written e.g. in C++, where multiple inheritance is a built-in feature. This C++ class may be inheriting from more than one super-classes, regardless of the language that these might be declared and compiled in, as long as they can provide their services through an interoperability standard. However, as it is obvious, this requires the usage of a language that supports multiple inheritance in addition to Java.

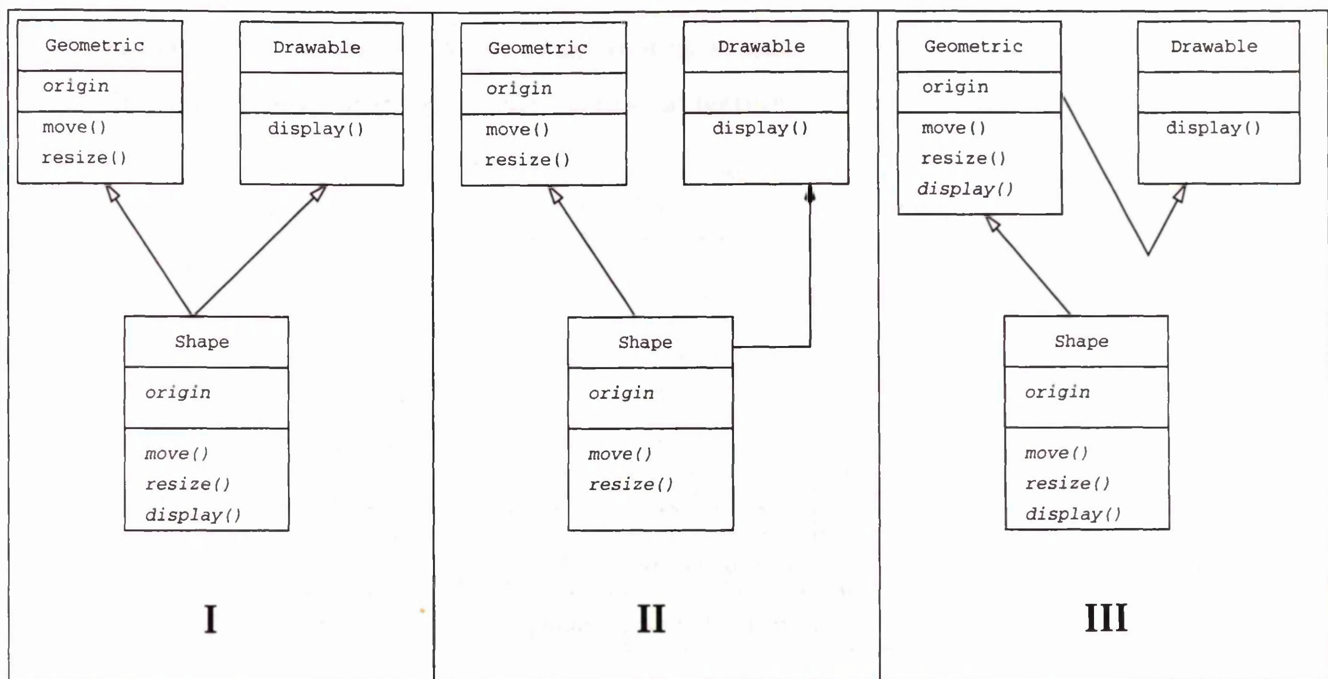


Figure A.1: Workaround for multiple inheritance in the UML class diagram when not supported by the implementation language. Text in italics denotes class members inherited from the superclass(es). See text for explanation

Regardless of the approach chosen, it must be noted that multiple inheritance situations should be avoided as much as possible, when using Java as the implementation language, since the functionality provided by any of the above solutions is considered not to be equivalent to that of multiple inheritance. The latest Java release offers a core application programming interface (API) to display 2D and 3D data. Third party attempts to incorporate Java within the GIS field are the OpenMap project, the JShape project.(* add more) These attempts address the problem of the GUI part mostly as a mediator between the user and the dataset, and do not examine exhaustively how Java can be used to build a data model for a GIS.

Laser Scan's Gothic environment and applications built on top of this core package,

Table A.1: Comparison of OO programming languages. Based on comparison found in Booch (1994), augmented with additional features and facts about Java.

Language-> Concept	**C++**	**Smalltalk**	**Java**
Abstraction			
Instance variables	yes	yes	yes
Instance methods	yes	yes	yes
Class variables	yes	yes	yes
Class methods	yes	yes	yes
Encapsulation of variables	public/protected/private	private	public/protected/private
Encapsulation of methods	public/protected/private	private	public/protected/private
Kinds of modularity	file	none	file
Inheritance	multiple	single	single
Generic Units	yes	no	yes
Metaclasses	no	yes	no
Strong typing	yes	no	yes
Polymorphism	single	single	single
Concurrency			
Multitasking	indirectly by class	indirectly by classes	built in multithreading
Persistence	no	no	yes
Pure Object-Oriented	no	yes	yes

Appendix B

Interoperability Standards In Object Orientation

B.1 CORBA

Objects in an object-oriented environment communicate via messages which have a well and pre-defined format. These messages are sent through the class interfaces. Usually[1], interfaces are only known to the classes themselves as well as to the client classes that send messages according to the interface specification. The designer, of course, is also aware of the interface format. In this way, for two classes to communicate, they ought to know each other's interface. If this format has to be changed for any reason, objects fail to communicate. Then, every message has to be re-formatted, which means extensive code alteration. As a result, components will work together only if they have been designed and built on standard interfaces which are independent of platforms, operating systems, programming languages and network protocols (Siegel 1996). This is the status of class communication without any interoperability standards. (Peckham and Marianski 1988)

If somehow, client classes request the format of the messages from the server

[1] If no interoperability standards have been embedded in the implementation of the classes.

classes before they actually send the message, then even if the interface of a server class is altered, the client class will always be able to communicate properly with any other class: in any case, each node in a distributed environment is an object with a well-defined interface, identified by a unique handle. Messages pass between a sending object and a target object; the target object is identified by its handle, and the message format is defined in an interface known to the system. This information enables the communications infrastructure to take care of all of the details. In this way, interoperability results because clients on one platform know how to invoke standard operations on objects on any other platform. This is how standards like CORBA work.

The Common Object Request Broker Architecture (CORBA), introduced by the Object Management Group (OMG) back in 1989, is an architecture that allows applications to be broken up into components. Those components communicate via a mediator software called Object Request Broker (ORB); it provides language and platform independence, transparently converting client requests in language X on platform A, to language Y and platform B at the server end. The power of CORBA is that, rather than code to a communications API, a developer simply defines an abstract interface and the compiler generates the entire communications routine.

Encapsulation enables CORBA to provide location transparency (Siegel 1996): clients send the invocation to their local ORB, not to the target object itself; the ORB routes the message to its destination through stored object references. In this way, ORBs know about object's physical addresses or where an application resides so they automatically route messages to their proper targets. Inheritance and polymorphism let CORBA work with object-oriented tools and languages.

In CORBA, an object's interface is defined in OMG Interface Definition Language (IDL). The interface definition specifies the operations the object is prepared to perform, the input and output parameters they require and any exceptions that may be generated along the way. Client and object implementation are then

isolated by at least threee components: an IDL stub on the client end, one or more ORBs and a corresponding IDL skeleton on the object implementation end. In this way, clients can only access an object as defined by its IDL interface, as there is no way in the architecture to access the implementation directly. The CORBA architecture separates the interface written in OMG IDL from the implementation, which must be written in some programming language. The object implementor must implement in some programming language all of the operations specified in the interface, so that writing the interface in IDL and implementing in a language are two different steps.

For every major programming language, an OMG standard language mapping specifies how IDL types, method invocations, and other constructs convert into language functions calls. This is how the IDL skeleton and the object implementation come together.

The ORB is responsible for storing interface definitions from server objects in an interface repository (IR). In it, interface definitions can be added, modified, deleted or retrieved. Its contents may be searched, and inheritance trees may be traced to determine the exact type of an object.

Objects that conform to CORBA standard can be dynamically added within a network. The Dynamic Invocation Interface (DII) is responsible for identifying newly installed objects, during run-time stage. Moreover, a DII may:

- discover objects' interfaces;
- retrieve their interface definitions;
- construct and dispatch invocations and
- receive the resulting response or exception information

Interoperability in CORBA is based on ORB-to-ORB communication. ORBs form a network, where each one of them is responsible for sending requests to local objects or forwarding requests to other ORB's.

GLASGOW
UNIVERSITY
LIBRARY