COMPARATIVE EVALUATION OF THE MODEL-CENTRED AND THE APPLICATION-CENTRED DESIGN APPROACH IN CIVIL ENGINEERING SOFTWARE

By
Alexander Sinske

DISSERTATION PRESENTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY (CIVIL ENGINEERING)
AT THE UNIVERSITY OF STELLENBOSCH

PROMOTORS: PROF. P.E. DUNAISKI, UNIVERSITY OF STELLENBOSCH,
O.PROF.DR.DR.H.C.MULT. P. J. PAHL, TECHNICAL UNIVERSITY OF BERLIN

December 2002

Declaration

I, the undersigned, hereby declare that the work contained in this dissertation is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Abstract

In this dissertation the traditional model-centred (MC) design approach for the development of software in the civil engineering field is compared to a newly developed application-centred (AC) design approach.

In the MC design software models play the central role. A software model maps part of the world, for example its visualization or analysis onto the memory space of the computer. Characteristic of the MC design is that the identifiers of objects are unique and persistent only within the name scope of a model, and that classes which define the objects are components of the model.

In the AC design all objects of the engineering task are collected in an application. The identifiers of the objects are unique and persistent within the name scope of the application and classes are no longer components of a model, but components of the software platform. This means that an object can be a part of several models.

It is investigated whether the demands on the information and communication in modern civil engineering processes can be satisfied using the MC design approach. The investigation is based on the evaluation of existing software for the analysis and design of a sewer reticulation system of realistic dimensions and complexity. Structural, quantitative, as well as engineering complexity criteria are used to evaluate the design. For the evaluation of the quantitative criteria, in addition to the actual *Duration of Execution*, a *User Interaction Count*, the *Persistent Data Size*, and a *Basic Instruction Count* based on a source code complexity analysis, are introduced.

The analysis of the MC design shows that the solution of an engineering task requires several models. The interaction between the models proves to be complicated and inflexible due to the limitation of object identifier scope: The engineer is restricted to the concepts of the software developer, who must provide static bridges between models in the form of data files or software

transformers.

The concept of the AC design approach is then presented and implemented in a new software application written in Java. This application is also extended for the distributed computing scenario. New basic classes are defined to manage the static and dynamic behaviour of objects, and to ensure the consistent and persistent state of objects in the application. The same structural and quantitative analyses are performed using the same test data sets as for the MC application.

It is shown that the AC design approach is superior to the MC design approach with respect to structural, quantitative and engineering complexity criteria. With respect to the design structure the limitation of object identifier scope, and thus the requirement for bridges between models, falls away, which is in particular of value for the distributed computing scenario. Although the new object management routines introduce an overhead in the duration of execution for the AC design compared to a hypothetical MC design with only one model and no software bridges, the advantages of the design structure outweigh this potential disadvantage.

Opsomming

In hierdie proefskrif word die tradisionele modelgesentreerde (MC) ontwerpbenadering vir die ontwikkeling van sagteware vir die siviele ingenieursveld vergelyk met 'n nuut ontwikkelde applikasiegesentreerde (AC) ontwerpbenadering.

In die MC ontwerp speel sagtewaremodelle 'n sentrale rol. 'n Sagtewaremodel beeld 'n deel van die wêreld, byvoorbeeld die visualisering of analise op die geheueruimte van die rekenaar af. Eienskappe van die MC ontwerp is dat die identifiseerders van objekte slegs binne die naamruimte van 'n model uniek en persistent is, en dat klasse wat die objekte definieer komponente van die model is.

In die AC ontwerp is alle objekte van die ingenieurstaak saamgevat in 'n applikasie. Die identifisieerders van die objekte is uniek en persistent binne die naamruimte van die applikasie en klasse is nie meer komponente van die model nie, maar komponente van die sagtewareplatform. Dit beteken dat 'n objek deel van 'n aantal modelle kan vorm.

Dit word ondersoek of daar by die MC ontwerpbenadering aan die vereistes wat by moderne siviele ingenieursprosesse ten opsigte van inligting en kommunikasie gestel word, voldoen kan word. Die ondersoek is gebaseer op die evaluering van bestaande sagteware vir die analise en ontwerp van 'n rioolversamelingstelsel met realistiese dimensies en kompleksiteit. Strukturele, kwantitatiewe, sowel as ingenieurskompleksiteitskriteria word gebruik om die ontwerp te evalueer. Vir die evaluering van die kwantitatiewe kriteria word addisioneel tot die *uitvoerduurte* 'n *gebruikersinteraksie-telling*, die *persistente datagrootte*, en 'n *basiese instruksietelling* gebaseer op 'n bronkode kompleksiteitsanalise, ingevoer.

Die analise van die MC ontwerp toon dat die oplossing van ingenieurstake 'n aantal modelle benodig. Die interaksie tussen die modelle bewys dat

dit kompleks en onbuigsaam is, as gevolg van die beperking op objek-identifiseerderruimte: Die ingenieur is beperk tot die konsepte van die sagteware ontwikkelaar wat statiese brûe tussen modelle in die vorm van lêers of sagteware transformators moet verskaf.

Die AC ontwerpbenadering word dan voorgestel en geïmplementeer in 'n nuwe sagteware-applikasie, geskryf in Java. Die applikasie word ook uitgebrei vir die verdeelde bewerking in die rekenaarnetwerk. Nuwe basisklasse word gedefinieer om die statiese en dinamiese gedrag van objekte te bestuur, en om die konsistente en persistente status van objekte in die applikasie te verseker. Dieselfde strukturele en kwantitatiewe analises word uitgevoer met dieselfde toetsdatastelle soos vir die MC ontwerp.

Daar word getoon dat die AC ontwerpbenadering die MC ontwerpbenadering oortref met betrekking tot die strukturele, kwantitatiewe en ingenieurskompleksiteitskriteria. Met betrekking tot die ontwerpstruktuur val die beperking van die objek-identfiseerderruimte en dus die vereiste van brûe tussen modelle weg, wat besonder voordelig is vir die verdeelde bewerking in die rekenaarnetwerk. Alhoewel die nuwe objekbestuurroetines in die AC ontwerp in vergelyking met 'n hipotetiese MC ontwerp, wat slegs een model en geen sagteware brûe bevat, langer uitvoerduurtes tot gevolg het, is die voordele van die ontwerpstruktuur groter as die potensiële nadele.

Zusammenfassung

In dieser Dissertation wird die traditionell Modell-zentrierte (MC) Methode für den Entwurf von Software im Bauingenieurwesen verglichen mit einer neu entwickelten Applikation-zentrierten (AC) Methode.

Bei Software, die mit der MC-Methode entworfen ist, spielen Modelle die zentrale Rolle. Ein Softwaremodell bildet einen Teil der Welt, zum Beispiel die Visualisierung oder Analyse, auf den Speicher eines Computers ab. Charakteristisch für den MC-Entwurf ist, dass die Identifikatoren der Objekte nur innerhalb des Namenraums des Modells eindeutig und persistent sind, und dass Klassen, die Objekte definieren, Komponente des Modells sind.

Beim AC-Entwurf werden alle Objekte einer Ingenieursaufgabe in einer Applikation zusammengefasst. Die Identifikatoren der Objekte sind eindeutig und persistent innerhalb des Namenraums der Applikation, und Klassen sind nicht mehr Komponente eines Modells, sondern Komponente der Softwareplatform. Das bedeutet, dass ein Objekt ein Teil von mehreren Modellen sein kann.

Es wird untersucht, ob den Anforderungen, die bei modernen Prozessen im Bauingenieurwesen an Information und Kommunikation gestellt werden, bei der MC-Methode Genüge geleistet wird. Die Untersuchung stützt sich auf eine Bewertung vorhandener Software für Analyse und Entwurf von Abwasser-Kanalnetzen mit realistischem Umfang und Komplexität. Strukturelle, quantitative, und Komplexitätskriterien werden zur Bewertung herangezogen. Zur Bewertung der quantitativen Kriterien werden zusätzlich zur tatsächlichen *Ausführungsdauer*, ein *Benutzer-Interaktionszähler*, die *persistente Datengröße*, und ein *Grund-Anweisungszähler*, der sich auf eine Komplexitätsanalyse des Quellcodes stützt, eingeführt.

Die Analyse des MC-Entwurfs zeigt, dass für die Lösung einer Ingenieursaufgabe mehrere Modelle benötigt werden und dass die Wechselwirkung zwischen den Modellen wegen der Beschränkung des Identifikatorenraums kompliziert und starr ist. Die Bewegungsfreiheit des Ingenieurs ist eingeschränkt auf die Konzepte des Softwareentwicklers, der statische Brücken zwischen den Modellen in Form von Dateien oder Softwaretransformatoren anbringen muss.

Die Grundlagen der AC-Methode werden dann dargestellt und in eine neue Applikation, geschrieben in Java, umgesetzt. Diese Applikation ist auch für das verteilte Rechnen im Computernetzwerk erweitert. Neue Grundklassen sind definiert, um das statische und dynamische Verhalten der Objekte zu verwalten und den konsistenten und persistenten Status in der Applikation zu sichern. Dieselben strukturellen und quantitativen Analysen wurden durchgeführt, wobei dieselben vier Datensätze, wie bei der MC-Methode, gebraucht wurden.

Es wird gezeigt, dass die AC-Methode der MC-Methode hinsichtlich der strukturellen, quantitativen und Komplexitätskriterien überlegen ist. Bei der Entwurfstruktur entfällt die Beschränkung auf den Identifikatorenraum und damit die Forderung nach Brücken, was besonders für das verteilte Rechnen im Computernetzwerk von Vorteil ist. Obwohl durch die neuen Objektverwaltungsroutinen beim AC-Entwurf, verglichen mit einem hypothetischen MC-Entwurf mit nur einem Modell und keinen Softwarebrücken, sich eine längere Ausführungsdauer ergibt, überwiegen die Vorteile der Entwurfstruktur diesen potenziellen Nachteil.

Contents

Declaration ii

Abstract iii

Opsomming v

Zusammenfassung vii

Acknowledgements xvi

1 Introduction 1

1.1 Purpose of the research 1
1.2 Computer-oriented concepts for civil engineering tasks 2
1.3 The OO paradigm 8
1.4 Definition of Quantitative Comparison Criteria and Test Projects . 16
1.4.1 Definition of Quantitative Criteria 16
1.4.2 Definition of test projects 18
1.5 Conclusion 18

2 Analysis of the MC design approach 26

2.1 Introduction 26
2.2 Characteristics of the MC approach 27
2.2.1 Definition of general concepts 27
2.2.2 The concept of software bridges 29
2.2.3 Evaluation of the MC approach 32
2.3 Description of the investigated engineering process 33
2.4 Functionality of the existing MC software system 36
2.5 Algorithmic background 38
2.6 Decomposition of existing software system into models 39
2.6.1 Hydraulic model 40

2.6.2 Visualization model 42
2.6.3 Topology model 42
2.6.4 Elevation model 46
2.6.5 Topography model 47
2.6.6 Geographical model 47
2.7 Bridges between models in the existing software system 48
2.7.1 Use of data files 48
2.7.2 Use of memory as a bridge 53
2.7.3 Transformer module 54
2.8 Analysis of the MC design structure 55
2.8.1 Limitation of object identifier scope 59
2.8.2 A priori implementation by software developer 60
2.8.3 Object duplication 60
2.8.4 Program maintenance 61
2.8.5 Program extensibility 61
2.8.6 Suitability for distributed computing 62
2.9 Quantitative analysis of the MC approach 63
2.9.1 Evaluation of the BIC and UIC 64
2.9.2 Result of the quantitative analysis 66
2.10 Conclusion 72

3 Concept of the AC design approach **73**
3.1 Introduction 73
3.2 Concept of an object identifier management 75
3.3 Concept of an object set management 80
3.4 Concept of an object relation management 83
3.5 The model-object 86
3.6 Extending the design to a distributed environment 88
3.7 Conclusion 89

4 Implementation of the AC design approach **90**
4.1 Introduction 90
4.2 Basic engineering objects 90
4.3 Implementation of object identifier management 93
4.4 Implementation of object set management 98
4.5 Implementation of the object relation management 99
4.6 Implementation of the engineering process 112

4.7 Functionality of the AC software system . . . 115
4.8 Algorithmic background . . . 115
4.9 Classes of the Product-data model . . . 117
4.10 Implementation of the engineering models . . . 122
4.11 Extending the design for a distributed environment . . . 129
4.12 Conclusion . . . 130

5 Analysis of the AC design approach **132**
5.1 Introduction . . . 132
5.2 Analysis of the AC design structure . . . 132
5.2.1 Limitation of object name scope . . . 136
5.2.2 A priori implementation by the software developer . . . 137
5.2.3 Object duplication and reduction . . . 137
5.2.4 Program maintenance . . . 138
5.2.5 Program extensibility . . . 138
5.2.6 Suitability for distributed computing . . . 138
5.3 Quantitative analysis . . . 139
5.3.1 Evaluation of BIC and UIC . . . 140
5.3.2 Result of the quantitative analysis . . . 141
5.4 Conclusion . . . 150

6 Comparison of designs and conclusions **151**
6.1 Introduction . . . 151
6.2 Comparison of the design structure . . . 151
6.3 Quantitative comparison . . . 153
6.3.1 Duration of Execution (DoE) comparison . . . 153
6.3.2 User Instruction Count (UIC) comparison . . . 155
6.3.3 Persistent data size (PDS) . . . 156
6.3.4 Modified BIC . . . 157
6.4 Comparison of complexity . . . 159
6.5 Summary and Conclusions . . . 161
6.6 Recommendations . . . 162

Bibliography **163**

A Abbreviations and trademarks **168**
A.1 Abbreviations . . . 168
A.2 Trademarks . . . 169

B Evaluation of Basic Instruction Count **170**

C Source code for typical MC Application **176**
C.1 Implementation of a typical *Hydraulic model* 176
C.2 Implementation of a typical *Visualization model* 182
C.3 Transformer module 186
C.4 Implementation of the traversal algorithm 189

D MC Instruction Count Evaluation **190**

E MC Analysis Results **199**

F Source code samples from AC system **201**
F.1 TopoClasses.POTreeTraversal Class 201
F.2 TopoClasses.DFTreeTraversal Class 204
F.3 TopoClasses.BFTreeTraversal Class 206
F.4 TopoClasses.AgeComparator Class 208
F.5 TopoClasses.Util Class 209

G AC Instruction Count Evaluation **211**

H AC Analysis Results **223**

I Tutorial for Sewsan AC **225**
I.1 Introduction 225
I.2 Installing the program 225
I.2.1 Stand-alone scenario 225
I.2.2 Single PC client-server scenario - Java client 225
I.2.3 Single PC client-server scenario - Applet client 226
I.3 Description of user interface 227
I.4 Loading the CAD Drawing 228
I.5 Building the Data Model 229
I.6 Using the Data Model 231

List of Tables

1.1 Size of test projects 18
1.2 Basic Instruction Count Summary 23

2.1 Typical rows of the .YXZ structured text file 49
2.2 Abstract data type for the .ELV structured binary file 50
2.3 Structure of the database files. 51
2.4 Typical lines of the .TRI unstructured text file 52
2.5 Typical rows of the .SDF unstructured text file 53
2.6 Comparison of key MC operations for DoE and BIC 67
2.7 Key MC operations for UIC 70
2.8 PDS requirement in Mbyte 71

4.1 Traversal algorithm 124

5.1 Comparison of key AC operations for DoE and BIC 142
5.2 Key AC operations for UIC 144
5.3 Analysis of BIC for internal vs external relationships 145
5.4 Analysis of PDS for internal vs external relationships 147
5.5 Comparison of key AC operations for DoE in distributed scenario 148

6.1 Comparison of key AC and MC operations for DoE 153
6.2 Comparison of total AC and MC operations for UIC 155
6.3 Comparison of total persistent data size for MC and AC systems 156
6.4 Comparison of key AC and MC operations for modified BIC 157

E.1 Spreadsheet of MC performance evaluation 200

H.1 Spreadsheet of AC performance evaluation 224

List of Figures

1.1 Topology model for sewer network of Test Project 1 19
1.2 Topology model for sewer network of Test Project 2 20
1.3 Topology model for sewer network of Test Project 3 21
1.4 Topology model for sewer network of Test Project 4 22

2.1 Relationship between model (M), partial models (A,B) and sub-model (S_i) 28
2.2 Relationship between two models and a software bridge 29
2.3 Five models connected via ten bridges 31
2.4 Collaboration diagram showing the engineering process in *SEWSAN MC* 34
2.5 Applications comprising the *SEWSAN MC* package 36
2.6 Components of the *SEWSAN MC* software package 40
2.7 Diagram showing the Hydraulic and Visualization models 41
2.8 Visualization of Test Project 2 43
2.9 Topology of sewer network stored in a CAD drawing 44
2.10 Topography for Test Project 1 48
2.11 Class Diagram of typical MC system 56
2.12 Collaboration diagram of typical MC System 57
2.13 Graph of key MC operations for DoE and BIC 69
2.14 Graph of UIC performance 70
2.15 Graph of PDS performance 71

3.1 Definition of AC approach 74
3.2 Mapping of persistent identifier to temporary reference 78
3.3 Inheritance vs interface design for implementing application identifiers 79
3.4 Definition of uniqueness in a relation 84
3.5 Definition of completeness in a relation 84

3.6 Definition of model-objects in an application 87

4.1 The basic engineering objects in a sewer network. 91
4.2 The classes of the Application package. 94
4.3 Class diagram of AppObject and App 95
4.4 Class diagram of AppSetObject and AppSet 98
4.5 Class and overview diagrams to illustrate relation complexity . . . 101
4.6 Edge class for reference-identifier design 104
4.7 Edge class for relation-object design 105
4.8 Vertex class for reference-identifier design 106
4.9 Vertex class for relation-object design 107
4.10 Class diagram of RelObject, Relation and Rel 108
4.11 Collaboration diagram showing the process flow in the AC approach 112
4.12 Class-diagram showing models and structure of the AC approach 113
4.13 Hierarchy of classes for the AC approach. 119
4.14 The DataModel class 120
4.15 The HydraulicModel class 123
4.16 The *Visualization model* and associated classes 125
4.17 The *Elevation model* and associated classes 126
4.18 Classes of the *CAD model* 128
4.19 The user-interface of the *CAD model* with Project 2 loaded. 131

5.1 Class Diagram of typical AC system 134
5.2 Graph of key AC operations for DoE and BIC 143
5.3 Graph of UIC performance 144
5.4 BIC for Internal vs External relationships 146
5.5 Comparison of internal vs external relationships of PDS 147
5.6 Graph of distributed AC performance 149

6.1 Comparison of key operations for MC and AC systems 154
6.2 Comparison of total UIC between MC and AC systems 155
6.3 Variation of file size with the number of pipes for MC and AC systems 156
6.4 Comparison of modified BIC for the MC and AC system 158

Acknowledgements

This work was completed with the support of a DAAD Scholarship. The author was also supported by the Charl van der Merwe Foundation.

Chapter 1

Introduction

1.1 Purpose of the research

Engineering models : The use of a computer in civil engineering is traditionally based on the concept of a software model. Such a model is a mapping of a part of the world to the memory and the processors of a computer. The model is used to simulate a part of the world in various ways, such as by its visualization, analysis of behaviour and optimization of properties. A model, if designed by the Object-Oriented paradigm (OO paradigm) (see Hartmann [16]), is composed of objects, which in turn possess attributes and methods. Information processing and communication for engineering tasks are supported by classes of objects, which are treated as referenced data types of the programming language. Engineers are particularly interested in the prediction of the performance of planned constructed facilities, designed products and natural systems using models.

State of the art : Information and communication in the civil engineering process are at present based on models which implement the OO paradigm. This state of the art is called model-centred design approach. The advantages and limitations of the state of the art will be investigated. Examples will be drawn from the field of hydraulic engineering, where the state of the art is implemented with existing software.

Model-centred design approach : The traditional model-centred design approach (MC design approach) is analysed in this dissertation. The design is

characterized by the following key properties: Objects are uniquely identifiable only within the scope of a model and classes of objects are regarded as components of a model. The analysis of the MC design shows that the solution of an engineering task requires several models. The interaction between the models in the engineering process proves to be complicated and inflexible: The engineer is restricted to the concepts of the software developer.

Application-centred design approach : A new concept, the application-centred design approach (AC design approach) for computer-aided engineering, is investigated in this dissertation. All objects of the engineering task are collected in an application. Their identifiers are unique and persistent within the scope of the application. The classes are no longer treated as components of a model, but as components of the software platform. Each object is unique in the application, but may be a component of several models. Furthermore, special data structures are introduced at an abstract level, to model the relationship between objects and the collection of objects independent of complexity of the engineering problem.

Objective : The objective of this dissertation is the investigation of the AC design approach for information processing and communication in civil engineering tasks. In particular, the complexity of the structure of an application based on the AC design and the consequences of the assignment of the same object to more than one model will be investigated. The complexity is strongly dependent on the number and the diversity of the objects and models in the application. The AC design approach will be studied for the same task, for which the MC design approach was analysed, namely a computer application for the analysis and design of a sewer reticulation system of realistic dimension and complexity. The design will also be extended for a distributed computing scenario. Finally an objective comparison can be drawn between the MC and AC designs based on quantitative and qualitative criteria.

1.2 Computer-oriented concepts for civil engineering tasks

In this section, a sequence of concepts are introduced which describe methods of computer-aided civil engineering which preceded the OO paradigm.

The concepts are listed in the sequence in which they were introduced historically. The shortcomings of these concepts justified the introduction of the OO paradigm, and thus the MC design and the AC design, which are studied in this dissertation.

Procedural approach : Computers were used initially in civil engineering to solve numerical tasks. The theoretical background of the engineering task was used to formulate an algorithm, for example for the computation of the forces in a structural frame or of the flows in a network of pipes. The algorithms were implemented in a series of statements of a programming language. The data of the engineering task was stored in such a way that it could only be used in conjunction with the program in which the algorithm was implemented. The significant improvement of computer-supported analysis over hand calculation was the speed of calculation (Pennington [36]). Later the top-down design was introduced using subroutines or procedures, which collected sets of statements together into small functional units.

Modules : The software component for the collection of common algorithms is called a module, a unit or a package, depending on the programming language which is used. A module is a sequence of statements and data storage areas, which are treated as a whole when they are converted to machine operations, or when they are loaded from a file to core storage. Due to the severe limitations of memory space in older computing systems, often only a kernel module and one optional module could be loaded into core memory at the same time. Usually, the kernel module for data storage and retrieval of files was retained in system storage for the full runtime, whereas separate modules for model definition, analysis and data presentation were loaded sequentially for pre-processing, main task and post processing.

Abstract data types : In the course of the application of modules to engineering practice it became apparent that the procedural approach suffered from serious drawbacks. Information was generated by one module, stored in a file or common global memory block and then reused by other modules, which read the data. This chain-concept requires that each of the modules knows the complete structure of each data file or memory block. The storage format of the data is therefore dependent on the algorithms in the modules.

The dependence of program code on file structure led to such a high complexity of software systems that their adaptability to new hardware environments and their extensibility for new areas of application proved inadequate. This deficiency was counteracted by the concept of self-describing data, whose syntax and semantics are independent of the algorithms in which they are needed and of the particular medium on which they are stored.

This concept was implemented with the creation of abstract data types: each type consists of structural data and attribute values. This is also known as `record` structures in Pascal [40] or `struct` components in the C programming language. The language processor registers the structural data when the abstract data type is defined; it exists only once for each module. The attribute values vary for every use of the data type as program variables.

Bridges : In order to structure the communication between models, the concept of a *software bridge* was introduced. A bridge functions as a gateway between models. It can transfer data between models passively via a file format (for example based on an abstract data type), or provide an active program logic to map data from one model to the other. The latter requires the bridge to have either direct access to the memory spaces of the models, in order to map the data types and structures, or to channel data types via a declared interface using standardized inter-process application calls. This is the case when partial models are packaged as Dynamic Link Libraries, for example as commonly introduced in the large C applications under Microsoft Windows (Petzold [37]). This also allows for the interaction between models which have been written in different programming languages and for which the source code is not available.

Partial models and files : If a model represents only part of an engineering product or system, it is defined as a partial model. The separation of procedures and data into independent components led to an information technology whose primary components were partial models (implemented as modules) and software bridges (implemented as data files). For example, the software package which is analysed in Chapter 2 is decomposed into partial models for data capturing, hydraulic analysis, graphical visualization and geographical presentation. The links between the partial models are implemented using data files, which are based on abstract data types or active software bridges.

Complexity of engineering process : The structure of civil engineering software must reflect the complexity of the engineering process if it is to be useful in practice. This complexity has several characteristics (Pahl [32]):

- **Dynamics :** Civil engineering projects are designed in stages. As a consequence, the information base is established in steps over a period of time. The data structure must therefore be flexible and adaptable. The user must be able to query the current state of the project information base with reasonable effort and in a manner which is consistent with his work style. This demands that the information must remain well structured and consistent at all stages of the project, even though it is modified and extended continuously.
- **Specialization :** Civil engineering projects are handled by specialists belonging to several professions. For each professional, the skills that are required to use the computer efficiently, should be restricted to his or her field. At the same time, each professional must be able to obtain the information which he or she requires from partners working on other parts of the project. This demands appropriate tools of communication.
- **Volume :** The volume of information which is associated with an engineering project is large. This leads to a problem of scaling. Methods and techniques which are adequate for small amounts of information prove inadequate for larger information bases. In spite of the high capacity of today's computers, significant effort is required to find efficient algorithms, data structures and user interfaces.
- **Teamwork :** Engineering projects are typically designed and executed by a large number of persons working in different organizations and at different locations. Traditionally documents such as contracts, specifications, calculations, drawings, bills, reports, etc. are ordered by organization in the first place, and by project in the second place. Although a significant portion of the information is passed on orally, for example by persons working in the same office or on the same construction site, there are established formats for documentation and communication. The software platform must support this teamwork.
- **Documentation :** The traditional documentation for teamwork originated before computers became widely used. In many aspects, it is not suited for computer-aided engineering in the distributed environment of modern computer networks. For example, many drawings are too large

and part of their information is not required by the recipient. Since engineering practice is an ongoing process, modifications of the documentation and the modes of communication can only be made gradually. The software platform must support this gradual transition.

- **Modification :** The design and the construction of civil engineering projects are characterized by frequent modifications, which tend to be small in volume. This must be reflected in the communication strategy of the software platform. It must be possible to form adequately small information packages for data transfer in the computer network, in order to avoid unnecessary transfer and review of unchanged data. The coherence and the consistency of the distributed database must be preserved.

- **Technological advances :** The capacity of computers and of networks has grown at a fast rate over several decades and is continuing to grow. The doubling of the capacity every 18 months leads to an increase by a factor of 100 every 10 years. As a result, the software platform is in a constant state of change and adaptation. The reusability of the software components and the databases in this growth process must be very high to guarantee the stability of the work processes and to protect the documentation in the projects and in the organizations.

  Additional modifications in the computer environment are required due to technological change in civil engineering itself. Codes and standards are in a constant state of review and modification. New techniques and products influence engineering practice. The interaction between civil engineering and other parts of society changes. The computer environment must be adapted to this change.

Computer-aided engineering with partial models and files is not suited to handle the complexity of the engineering process, as discussed above: This can be quantified in terms of the complexity criteria:

- **Dynamics :** The rigid file structure is not suited for a flexible and adaptable information base. The resulting system proves to be inflexible, since modification of an abstract data type in one module results in a change in the data file structure. This makes an old data file incompatible with the new abstract data type. All other modules must be recompiled to be able to use the new abstract data. At worst, large parts of the source code must be updated to account for the new data format.

- **Specialization :** The rigid file structure does not provide flexibility in the exchange of data between the professionals working on the project. The exchange format is defined by the developer of the software and does not allow for customization by the specialists, for example for the addition of new types of relevant data. Models and software bridges between models should be extensible to handle changes and additions introduced by connected models.

- **Volume :** The transfer of data between models using files is adequate for small to medium amounts of data. However, as soon as large volumes of data are involved the time required to export data from a model to secondary storage (hard drive) and subsequent input of data to a different model by using a file, grows proportionally with file size. Especially if data is to be transferred between models located on remotely located platforms the simple data file structure does not present an efficient means of transport in a distributed computing system.

- **Teamwork :** The exchange of data between models using files is suitable when one person is working on the project, since this person knows the data structure for the different models and the execution order of the different models. However, as soon as more than one person works with a specific model, e.g. several people who enter the network topology, the rigid file structure does not support concurrent management. A management system based on the concepts of distributed computing is required. The ordering of documents should not be rigid, for example by organization and project as is the case with a data file system in an operating system directory. A flexible management system is required which allows for indexing of documents on various keywords. The documents should be stored without redundancy using unique identifiers.

- **Documentation :** The exchange of drawing data between models via files has been the primary mode of transfer for graphical information. The structured drawing interchange file does not support the management and tracking of changes. However, the CAD files are mostly just computerized versions of paper plans and are not optimized for transferring model information. Furthermore, the structured drawing interchange file normally does not support the management and tracking of changes. A solution could be that the documentation is always extractable from the model. The model should also contain all documentation relevant to the design process accessible to all parties in the design process.

- **Modification :** Traditionally CAD or database files are used to exchange data between models. Normally only the complete data set or file is transferred. However, whenever a change is made in one model, the new updated data must be passed to all other models. It is very inefficient for large data sets to be transferred in distributed systems. Each modification requires an update of the file structure and triggers a chain of changes. If n models are linked to each other, each change triggers up to $(n-1)$ changes to connected models. Because of indirect independence up to $O(n^2)$ connections may need to be updated.
- **Technological advances :** The data file structure to interface with specific models is set out at the time it is designed by the developer. Using the same structure in a new and different scenario is not simple, as enhancements to the models that implement the file structure for data exchange will require changes to the file structure itself and thus programming of the affected models and interface routines. New standards in engineering are not easy to implement as they also lead to changes in the data file structure.

1.3 The OO paradigm

Both the MC design and the AC design, which are studied in this dissertation, are based on the OO paradigm, which has found extensive usage in civil engineering and software development during the past decade. The *Deutsche Forschungsgemeinschaft*, DFG (German Research Council) [11] has supported research by 16 universities with a substantial financial investment over a period of six years ending 1998. Hartmann [16] and the research teams have summarized the results of the research in the documentation *Objektorientierte Modellierung in Planung und Konstruktion*. In this section, the general aspects of their findings, which are relevant to this dissertation, are summarized and commented. All references are to the team leader(s) as well as the chapter number.

Focus points of research : In the application to the DFG it was made clear that the benefit of object-orientation was to be introduced to civil engineering without inhibiting the problem solving and thought processes of the engineer. For the purpose of the research the following key points were investigated

in the fields of computer science, architecture and building planning, finite element modelling in civil engineering, construction engineering, construction management and geotechnical engineering:

- OO modelling of building infrastructure, facility management and environmental interaction.
- OO modelling of norms and codes used in civil engineering.
- OO modelling of the planning and construction process with emphasis on the formal transfer of information between parties.
- Design of a functional user interface for software in design, planning and construction.

Comments by Hartmann : In the introduction Hartmann introduces the development of the OO paradigm and details the principles, concepts and methods of object-orientation. Then the three classical steps of OO software technology are outlined: OO analysis, OO design and OO programming. The different modelling techniques developed over the years are discussed in detail and compared. The different diagramming tools available for the modelling techniques are shown by example. The construction of models with the OO paradigm was the focus of the DFG research together with the software implementation using OO programming. The principles of object-orientation which have made it so successful as design and programming paradigm are then detailed:

Principles of object-orientation : The following principles of the object-orientation are identified in the introduction by Hartmann:

- **Principle of secrecy :** Objects provide a public interface to the outside world (to the user of the object) yet the inner source code is hidden and available only to the developer.
- **Encapsulation of data :** Object attributes, methods and relations between objects are contained in one inseparable structure, which models the real-world scenario much better than conventional paradigms.
- **Communication between objects :** Objects can communicate with each other and transfer relevant information, similar to the human communication process.

- **Sets and classes :** Objects which have common properties, can be collected in *sets* so that the mathematical set theory can be applied. *Classes* group objects with common attributes and methods and provide a template for the construction (instantiation) of objects. The use of *inheritance* allows methods and attributes to be passed on to other objects in a tree structure.

- **Variations of the standard approach :** Most projects of the DFG as described in the report were based on the standard OO modelling concepts. Some researchers have introduced special techniques and derivations of the standard technique. Pahl and Damrath [35] introduced an object-centred design where objects in the application are defined on system level and not on model level. A different object-centred design, similar to the work of Garret [14], was used in the work of Scherer [39], where objects are no longer class-bound: conditions are embedded in a class definition which determine if an object is to be classified as an instance of the class. In this design, an object can therefore be an instance of several independent classes.

Finding an OO solution : The process of finding an OO solution can be divided into the following steps:

- **OO analysis :** From a detailed description of the real-world problem, an adequate abstraction of the objects required to model the system can be created. It is very important that the mapping of the real-world system leads directly to the objects. Generally this step requires more effort than the classical analysis concepts. It can however save time during the implementation phase, as a full understanding of the problem has already been reached.

- **OO design :** In this step the software architecture is determined. Partitioning into partial systems as well as the formation of classes and sets are considered during this phase. Other issues such as choice of storage system, use of parallel processing and the user interface design are also treated.

- **OO programming :** This is a direct extension of the preceding two steps. Ideally, the generation of source code should now follow automatically. It is important to have an OO programming language which supports most or all of the features and functionalities introduced during the design

phase. However, as the reports show, none of the OO programming languages today fully supports all of the OO modelling concepts.

- **Modelling methods :** Since 1990 many modelling methods have been developed for the OO paradigm. For the study period of the DFG projects, three modelling methods were recommended: OO analysis (OOA) by Coad and Yourdon [5], OO modelling technique (OMT) by Rumbaugh e.a. [38], and Unified Modelling Language (UML) [3].

- **UML :** Of these methods, today only the UML method is still widely used, since it has become the basis for standard modelling methods. However, in the civil engineering community, the implementation has not been that successful. The dynamic behaviour of the complex civil engineering problems can generally not be adequately defined using the diagramming tools and techniques available in UML. The structured diagramming tools of UML are used selectively, when appropriate:

  The *class diagrams* have found common use to show the static class structure of models. UML highlights the interaction between the user and the application, with special *use case diagrams*, showing this interaction for different scenarios. *Sequence diagrams* try to show the dynamic behaviour of a system by detailing the exchange of messages between objects. In order to show the interaction between objects within their class framework, the *collaboration diagram* can be used. Another useful diagram is the *component diagram*, which is used to show the interaction between program modules or packages. Hartmann [15] used many of these diagrams successfully in a sample system. In another work of Hartmann [17], the *state diagram* was found inadequate to model parallel threads and had to be improved.

Fundamental research in OO modelling and programming : It has been found that the modelling concepts which are available in the OO analysis as well as the OO design phases of a project can not all be implemented by the software technologies during the OO programming phase. Researchers have therefore introduced several new concepts to existing programming languages, such as C, C^{++}, and database systems. Some of these features encountered in the report include:

- **Views :** Cremers [7] introduces the extension of OO databases with the concept of perspectives. This allows the customization of global database

schemas in such a way that views, which map the individual partial models into the global database, can be created.

- **Relations :** The connection of objects with each other is defined using a concept of relations. However in a standard OO programming language, such as Java, no implementation for this is found. A new family of types, similar to classes, have been developed by Cremers [7] to manage relationships between objects. Similar research has also been done by Pahl and Damrath [35] where relations are modelled as stand-alone objects which manage the connection between linked objects. Independently, Kolender [22] compared the modelling of relationships between objects internally within the classes versus modelling the relationships as independent classes. He concluded that the external implementation of relationships using classes decisively increases the quality of an application. Olbrich [30] makes use of set, relation and graph theory to model the relationships between objects external to the class definition, in tables within special relationship-objects with their own properties and behaviour.

- **Sets :** Basing object collections on mathematical set theory, Pahl and Damrath [35] provided a new way of managing groups of objects independently of their class structure. Sets are introduced as a stand-alone generic object type. Operations on sets such as the execution of a method on all elements of the set have been developed.

- **Listeners :** New ways of ensuring consistency of the state of objects in different models as well as the propagation of changes led to the design of systems which model the real-world much closer. (Pahl and Damrath [35]). A listener concept based on user interaction has been introduced. It was found that automatic treatment of inconsistencies would require too many calculations and iterations. By flagging the inconsistent state, the user can detect the inconsistency and trigger an update when required, e.g. click the refresh button in a graphical visualization.

- **Programming language choice:** In most reports the C^{++} language was used. The Java programming language has been available only for a few years and few researchers have commented on its usability. Pahl and Damrath [35] recommended the research of the Java Interface technology to manage relations and the use of the Java *Reflection API* to create dynamically adaptable software systems.

Choice of modelling strategy: MC or AC design : Another interesting aspect is the choice of the modelling strategy. Though detailed reasons as to why the strategy was chosen are not always stated, the following could be observed:

- Wassermann [45] considered both alternatives when designing a planning system for buildings. For the MC design a system of transformers between the partial models is considered. However it was decided to extract the shared functionality from both models and add to a common model to the system. This is still basically a MC design, yet with a practical solution to integrate the models in a closer way.
- Pahl and Damrath [35] argued that it is better to separate objects from models, and manage them as elements on a system level. A MC design is inappropriate when models are to be altered often, as the functional interfaces (ports) between models would have to be changed every time.
- Hartmann [17], Meißner and Möller [27] created separate object models for the partial models and then consider the integration into a holistic object model.
- Hartmann [15] presented a MC solution. A concept of variable transformers between partial models is introduced. On top of the partial models, a super-ordinate interface component manages the control of the software system.

Interaction between partial models : In many cases, existing software modules were used to implement partial models. A common topic then was to find the best way to transfer data and commands between these partial models.

- Wassermann [45] solved the problem by storing the topology and geometry of the elements using mathematical graph theory and common administrative data in a central model. The two partial models both have access to the central model and present different views of the data. Complex transformations between partial models can therefore be avoided.
- Hartmann [17] used a persistent OO database as medium to transfer data between partial models.

- Hartmann [15] used transformers (software bridges) for communication between partial models. For the interaction with program-external components, wrapper components are created which then interface with the program-internal components. An advantage of this solution is that all partial models can function on their own as stand-alone applications. A similar approach to integrate separate models is presented by Diaz [10] for the geotechnical engineering field.
- For every partial process, Meißner and Möller [27] designed a partial model whose creation, storage and management is sub-ordinate to a central OO model management system, which uses an OO database as persistent storage medium.
- Wörner [47] argued that the scope of data plays a critical role in modelling of partial models. Global data, which is used by more than one partial model, such as topology and geometry data should be stored in a central OO database. Local model data is characterized by direct coupling with a single partial model. It is only used in special cases within other partial product models. This data is also stored in the central OO database, but accessed via interface functions of the owning partial model. Specific partial model data is not stored persistently in the central database, as other models do not use it. Furthermore he argues that the use of *STEP2DBS* or other standards for the exchange of civil engineering data should be considered in the future.
- Werner [46] argued that component software is seen as the solution to the integration question. Component wrappers can easily provide wrappers for legacy systems and then bind them into existing models.

Distributed computing concepts : Some of the reports consider the potential of OO modelling in a distributed computing environment. The following presents a summary of what has been researched:

- Hübler [19] introduced the concept of a distributed object server in the environment of civil engineering. Using standardized interfaces for the CORBA Object server, clients are ensured of the functionality available on the server, even if the server software is exchanged.
- Hartmann [17] also considered the use of CORBA as medium for communication between models in order to allow parallel co-operative work.

This involves the creation of a Product, Project, Schedule and Team Server using *COOPERATE*, a CORBA based solution.

- Werner [46] used Microsoft *ActiveX* as Object Request Broker and used the *DCOM* system to embed components from remote locations into the geotechnical application.

Unique object identification : Many reports have focussed on this important topic.

- Hübler [19] recommends the unique differentiation of an object from all other objects, regardless of its storage place, time and object state.
- Pahl and Damrath [35] recommend a system where objects are uniquely identifiable on a level outside of models.

Concept of views : Many researchers have addressed the issue of views on a model.

- Cremers [7] developed a new way of mapping views of models onto objects. A discussion on why existing techniques from OO programming (such as using single inheritance, interfaces or multiple inheritance) do not provide an adequate solution is presented. However the solution requires the use of his proprietary compiler preprocessor.
- Pahl and Damrath [35] investigated two options of mapping a real-world object to a software object: A real-world object is mapped exactly to one single system object. Then the partial models each have a view on the object and only "see" the attributes which are relevant to the view. Alternatively and recommended, a real-world object is mapped onto a set of objects, which are connected to each other via structural relations. One object is typically the static core object located on the system level, while the other objects present the role objects which belong to the partial models. A system is extended by the addition of new role objects for an existing core object. This should allow for a flexible extension of the system with more views.
- Hartmann [17] stored all the attributes of all models in a common OO database and then ignored the attributes which are not relevant to a specific model, as OO databases do not yet provide a view concept.

Conclusion : The foundation for the research has now been formulated. The next section proceeds with a discussion of the quantitative comparison criteria to be used for the analysis of the two design approaches and a definition of the test projects used in all evaluations.

1.4 Definition of Quantitative Comparison Criteria and Test Projects

Four quantitative measures are introduced to capture the performance of a software application: the duration of execution, the basic instruction count, a user interaction count and the persistent data size. The criteria are now defined in detail followed by a definition of the test projects used in all comparisons.

1.4.1 Definition of Quantitative Criteria

Duration of Execution (DoE) : The duration of execution is an absolute measurement of the time elapsed during the execution of program code. It is only of interest where the execution time on the test bench computer results in noticeable delays for test projects of typical size. As many computer hardware and software related factors influence the numerical values for this criterion, it is vital that a static test bench computer system is chosen for all evaluations for both the MC design and the AC design.

Of particular interest is the ratio between the time required to execute the test projects of different size. This can be plotted and evaluated to compare results for the MC design and the AC design.

Basic Instruction Count (BIC) : The basic instruction count is used to evaluate the time complexity of source code in particular for the interaction between models.

Complexity analysis attempts to eliminate the platform dependence of DoE measurements. The number of basic processor instructions is counted for each common high-level operation. An alternative approach suggested by Li [24] for Java code, is to assume that an operation in Java takes one unit of time if the operation is implemented with a simple Java Virtual Machine

instruction. This approach cannot be used to evaluate the complexity of non-Java applications, and requires understanding of the Java byte-code.

The BIC can be used to compare the time complexity of source code written for different software programming environments. This allows for example a comparison of execution duration of source code written for a compiler language (such as Object Pascal or C++) and source code written for an interpreter language (such as Basic or Java), if it is assumed that the same software and hardware environments are available.

The BIC can give an indication of how efficiently algorithms, written in different programming languages, are designed. The BIC can also be compared to the DoE and should follow the same trend provided that the execution takes place on the same hardware processor.

The BIC is based on using a set of codes for each type of basic operations in the source code. For each unique category code the equivalent number of Intel Pentium processor instructions is evaluated. This is done by debugging typical programming code instructions to the assembly language level (see Intel [21]), and counting the number of instructions. In order to count the number of instructions during the interactive debugging process, a keystroke recorder application was used to count the number of times the *step-into* debugger command was activated. Appendix B shows a screen-shot of the test application written for this purpose and the evaluation of the BIC in a spreadsheet.

The basic operations are grouped into categories, which are designated by the first letter of the category code: Initialization *(I)*, Assignment *(A)*, Comparison *(C)*, Flow of execution *(F)* and Operation *(O)*. The codes for the basic operations, their descriptions and the BIC are listed in Table 1.2 at the end of the chapter.

As an alternative, the number of basic operations rather than the actual BIC can also be used as a measure. This measure is similar to the Java byte-code evaluation by Li [24].

User Interaction Count (UIC) : Some steps of the interaction between the user and the software application require considerable user interaction time. This is especially the case during the topology model construction and verification phase. Since an absolute measure is not useful in comparing the interaction times between differently skilled users and between differently fast computers, a user interaction count is proposed. This measure counts the number of keystrokes or input-device clicks required for a typical cycle of

operations.

Persistent Data size (PDS) : Although storage complexity can be evaluated for all basic operations during program execution, it has been found that this factor is generally of secondary importance in modern computer systems with large and inexpensive storage capacities. Of primary importance however is the size of data to be transferred between networked computer systems. This size is best characterized by the total size of the persistent data required by the application, defined as the persistent data size.

1.4.2 Definition of test projects

In order to obtain a range of typical situations, four different scenarios are investigated. All data define real-world problems to be analysed and designed in a consulting bureau. In all cases the complete engineering process of analysing and designing a sewer network from captured plans to final reports is investigated.

The projects represent data sets of increasing size and are used to illustrate and evaluate the performance of the software system for a typical civil engineering bureau environment. The sizes of the data sets are illustrated in Table 1.1. Graphical views of the test projects are presented in Figures 1.1 to 1.4.

Table 1.1: Size of test projects

Description	Number of pipes
Test Project 1 (TP1)	104
Test Project 2 (TP2)	682
Test Project 3 (TP3)	2934
Test Project 4 (TP4)	6713

1.5 Conclusion

The aim and scope of the research have now been formulated. The next chapter will proceed with the analysis of the MC design approach of a typical civil engineering application.

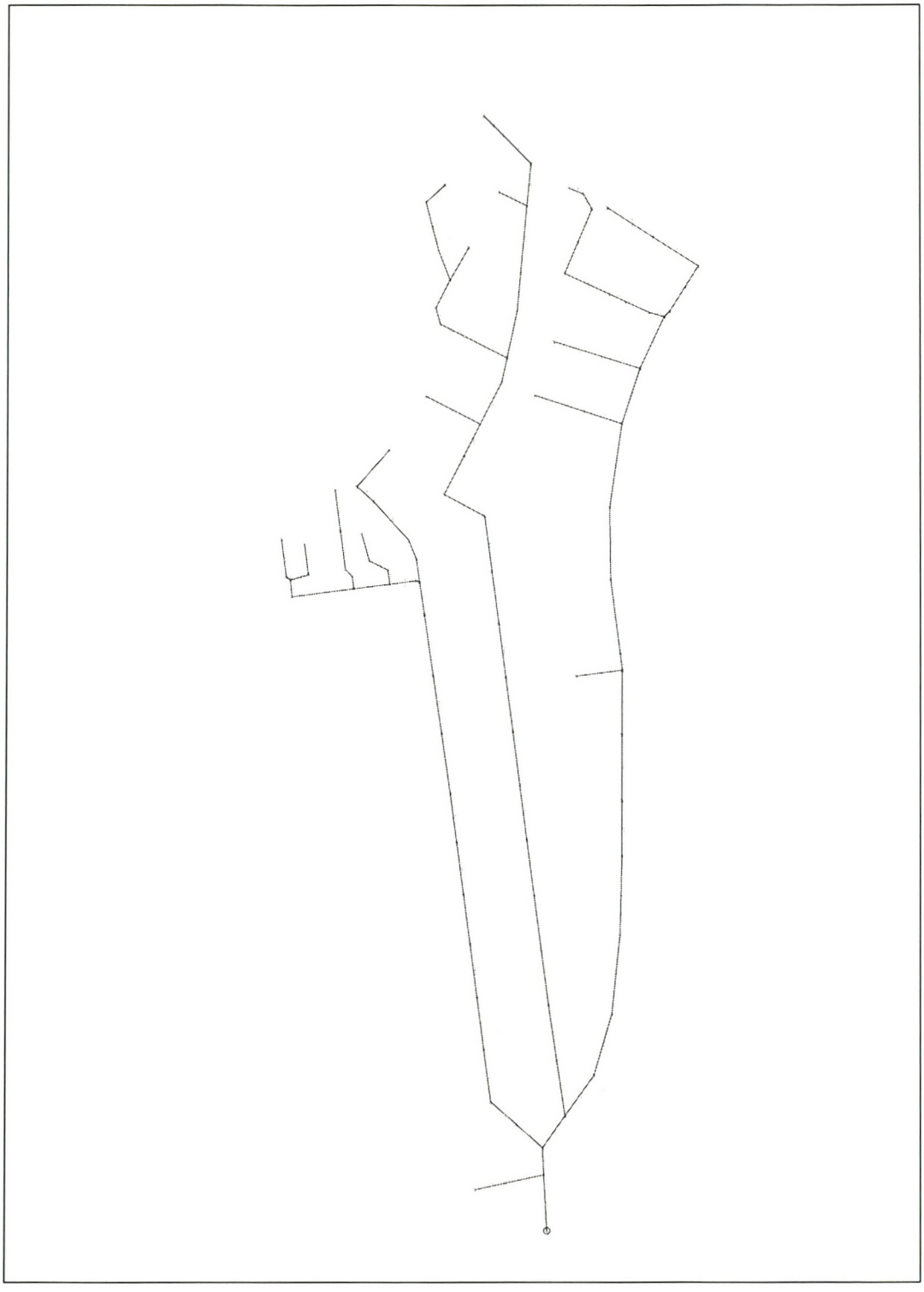


Figure 1.1: Topology model for sewer network of Test Project 1

Figure 1.2: Topology model for sewer network of Test Project 2

Figure 1.3: Topology model for sewer network of Test Project 3

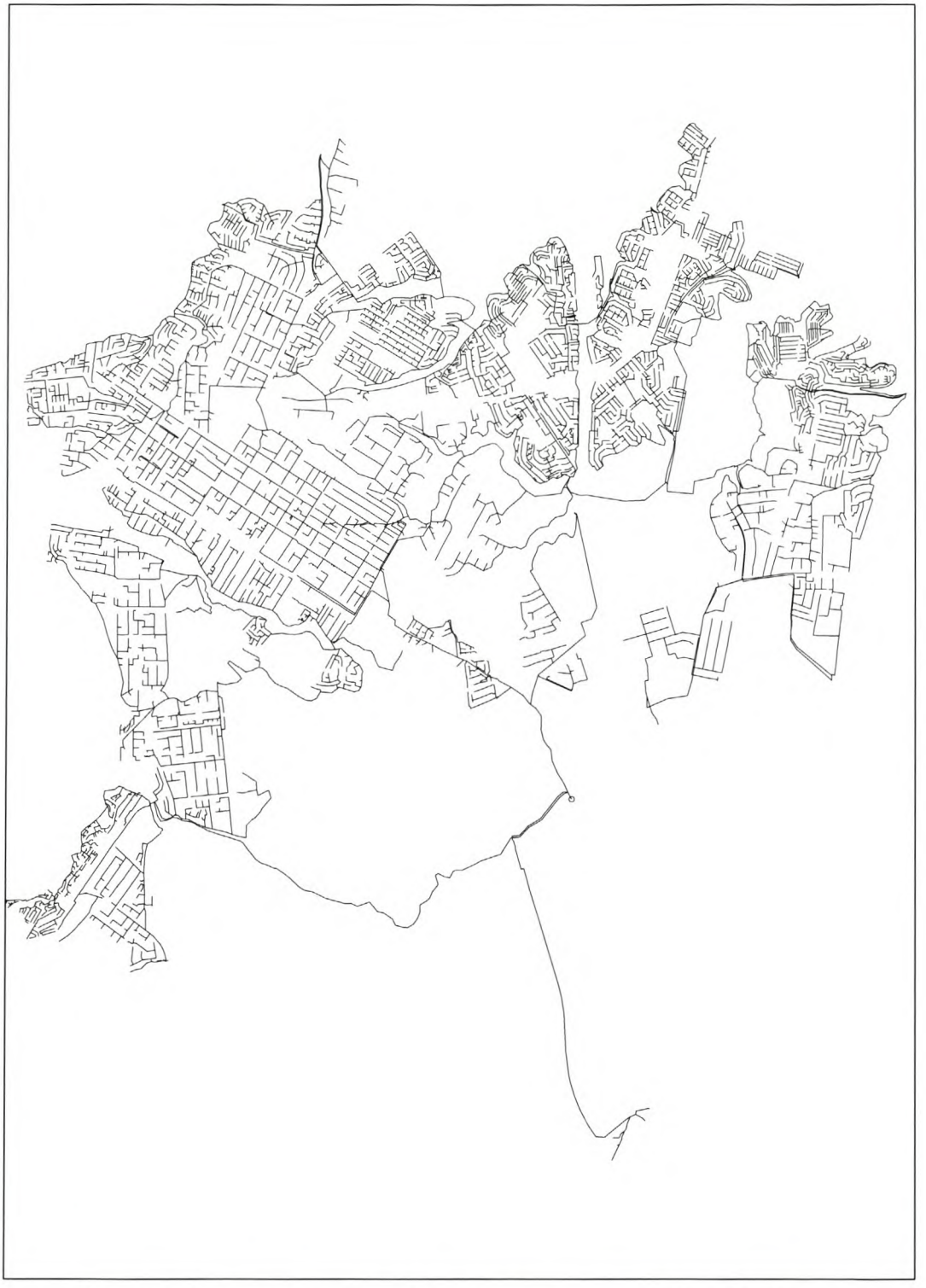


Figure 1.4: Topology model for sewer network of Test Project 4

Table 1.2: Basic Instruction Count Summary

Code	Description	Count
(I)	**Initialization**	
(IC)	*Create and initialize local*	
(ICLI)	integer	1
(ICLD)	double	3
(ICLO)	object reference	2
(ICLS)	string	5
(A)	**Assignment**	
(ACL)	*Assign constant to local x*	
(ACLI)	integer	1
(ACLD)	double	2
(ACLS)	string	222
(ALL)	*Assign local x to local x*	
(ALLI)	integer	2
(ALLD)	double	4
(ALLO)	object reference	22
(ALLS)	string	12
(ACF)	*Assign constant to far x*	
(ACFI)	integer	14
(ACFD)	double	535
(ACFS)	string	418
(ALF)	*Assign local x to far x; far x to local x*	
(ALFI)	integer	19
(ALFD)	double	354
(ALFO)	object reference	300
(ALFS)	string	665
(C)	**Comparison**	
(C2C)	*Compare local x to constant*	
(C2CI)	integer	2
(C2CD)	double	4
(C2CO)	object reference (zero)	2
(C2CS)	string	50
(C2L)	*Compare local x to local x*	
(C2LI)	integer	3
(C2LD)	double	5
(C2LO)	object reference	3
(C2LS)	string	15
(CFC)	*Compare far x to constant*	
(CFCI)	integer	29
(CFCD)	double	240
(CFCO	object reference (zero)	2
(CFCS)	string	94
(C2F)	*Compare local x to far x; far x to local x*	
(C2FI)	integer	174
(C2FD)	double	356
(C2FO)	object reference	256
(C2FS)	string	389

Basic Instruction Count Summary (Continued)

Code	Description	Count
(F)	**Flow of execution**	
(FN)	*Passing flow to near location*	
(FNG)	goto location	1
(FNLF)	loop: for statement	1
(FNLN)	loop: next statement	2
(FNPI)	passing integer value parameter	2
(FNPD)	passing double value parameter	3
(FNPR)	passing parameter by reference	2
(FNPF)	function return by reference	2
(FNC)	make call to function	7
(FF)	*Passing flow to far location*	
(FFPI)	passing integer value parameter	2
(FFPD)	passing double value parameter	3
(FFPR)	passing parameter by reference	2
(FFPF)	function return by reference	6
(FFC)	Make call to function	212
(O)	**Operation**	
(OA)	*Local Addition of*	
(OAIC)	integer with constant	1
(OAII)	integer with integer	2
(OADC)	double with constant	4
(OADD)	double with double	4
(OS)	*Local Subtraction of*	
(OSIC)	constant from integer	1
(OSII)	integer from integer	2
(OSDC)	constant from double	4
(OSDD)	double from double	4
(OM)	*Local Multiplication of*	
(OMIC)	integer with constant	3
(OMII)	integer with integer	3
(OMDC)	double with constant	4
(OMDD)	double with double	4
(OD)	*Local Division of*	
(ODIC)	integer by constant	4
(ODII)	integer by integer	4
(ODDC)	double by constant	4
(ODDD)	double by double	4
(OC)	*Local Casting of*	
(OCIS)	integer to string	613
(OCDS)	double to string	719
(OCSI)	string to integer	69
(OCSD)	string to double	118

Basic Instruction Count Summary (Continued)

Code	Description	Count
(O)	**Operation (continued)**	
(OS)	*String handling functions*	
(OSA)	add string to string	524
(OSM)	middle substring	310
(OSI)	search for in string	44
(OSL)	get length of string	7
(OST)	*Text File handling functions*	
(OTWI)	write integer	106
(OTWD)	write double	294
(OTWS)	write string	54
(OTWL)	write line end	30
(OTRI)	read integer	120
(OTRD)	read double	840
(OTRS)	read string	370
(OTRL)	read line end	24
(OB)	*Binary Record File handling functions*	
(OBW)	write record	27
(OBR)	read record	27
(OR)	*Referencing functions*	
(ORDA)	de-referencing local array	1
(ORDL)	de-referencing local object	2
(ORDE)	de-referencing external object	200
(ORDI)	de-referencing var index objects	380
(OM)	*Mathematical functions*	
(OMS)	square root of double	4
(O)	*Other functions*	
(OCAD)	external CAD function from VB	2000
(OJH)	external Hash table function from Java	50
(OCO)	create Java Object overhead	10

Chapter 2

Analysis of the MC design approach

2.1 Introduction

In this section the features of the model-centred design approach (MC design approach) are structured on the basis of a software package, *SEWSAN* [25] (defined as *SEWSAN MC* in the dissertation). The software is used in the engineering practice in South Africa for the analysis and design of sewer reticulation systems, and is typical for packages of the MC design type. The chapter starts with a discussion of the characteristics of a MC approach to form the basis for the analysis. A description of the engineering process for the analysis and design of a sewer system follows. Then the functionality of the applications in the engineering process is discussed. It includes a brief review of the theoretical background and identifies the corresponding algorithms which have been implemented in the package. The decomposition of the existing system into partial models and the implementation in modules are investigated. Then an analysis of the MC approach is presented. The suitability of the package for the engineering process is investigated for both a local environment and a distributed computing environment. The structure of the connections between the models in the form of data files and active software bridges is presented in detail. Detailed quantitative analyses follow for the interaction between models. The main properties, the efficiency and the flexibility of the data transfer methods are investigated. Typical volumes of data files for four test projects, which have been handled with the software package, are considered. Finally, an overall evaluation of the model-centred

software system (MC system) is presented. The results are summarized in such a way that the features of the MC approach can be compared with the features of the application-centred design approach (AC design approach) in Chapter 5 of the dissertation.

2.2 Characteristics of the MC approach

In this section, the general characteristics of the MC approach are presented. First some terms relevant to modelling software systems in engineering and especially the MC design are defined in a logical order. Then the name scope limitation of object identification is illustrated. This is followed by a description of communication structures between models, such as data files and active software bridges. The flexibility and efficiency of the MC approach is then discussed. Finally, the potential usage of a software system based on the MC design in a distributed computing environment is presented.

2.2.1 Definition of general concepts

Engineering process : This is the application of activities from engineering technology to a real-world system which is of interest. An example is the process of analysing and designing a sewer reticulation system.

Engineering software model : An engineering software model is the mapping of an engineering process to the memory and the processors of a computer. This model is used to simulate part of the real-world in various ways, such as by its visualization, analysis of behaviour, optimization of properties, etc. The model consists of components and describes state. The generic term *model*, refers to the engineering software model in this dissertation.

Partial engineering model : If an engineering software model represents only part of an engineering process, it is defined as a partial engineering software model (or short partial model). Such a partial engineering software model could be the visualization partial model or the analysis partial model. Partial models are uniquely identifiable within the scope of the engineering software model. The term partial engineering software model is often reduced to the term *model* if the partial property is not emphasized.

Module : The software component for the implementation of a partial model is called a module. A module is a sequence of statements and data storage areas, which are treated as a whole when they are converted to machine operations or when they are loaded from a file to core storage.

Engineering software sub-model : In this dissertation the additional term sub-model is introduced. Whenever partial models are implemented as modules in such a way that the modules share objects and classes in common memory structures, they are referred to as sub-models. Sub-models do not follow the classical MC approach of scope limitation on objects and classes of objects. Sub-models are uniquely identifiable within the scope of the containing partial model. Figure 2.1 shows this relationship.

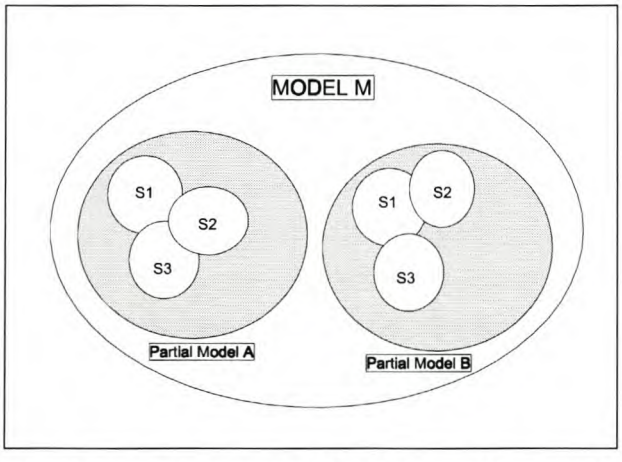


Figure 2.1: Relationship between model (M), partial models (A,B) and sub-model (S_i)

Scope of objects : In the MC approach, objects are uniquely identifiable only within the scope of a partial model. The same object identifier can therefore be used to reference different objects in separate partial models.

Scope of classes of objects : The scope of classes should be clarified. From a classical modelling standpoint, classes are seen as the blueprints for objects. It follows that a model is considered as consisting of classes, which, when instantiated, provide the objects which are bound by the scope of the model. One could therefore argue that classes must also be bound by the scope of models. However, modern programming languages follow a different approach. Classes which contain abstract data type definitions are considered to be independent of any model. Rather they are part of the software and

hardware platform on which the application is executing.

2.2.2 The concept of software bridges

Since the scope of the name of an object is restricted to the partial model which contains the object, a communication structure is required to connect objects which interact with one another across the model boundaries. This communication structure is called a *software bridge*. Figure 2.2 shows a software bridge. The bridge transfers data and commands across the boundaries of models. The software bridge is a special software which provides a one-to-one relationship between models. The bridge is defined by specifying which information can be transferred (syntactically and semantically) and which commands can be accepted for execution.

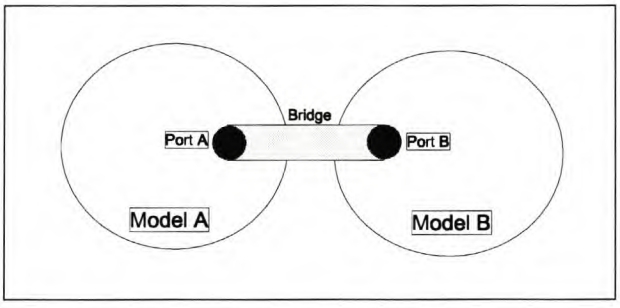


Figure 2.2: Relationship between two models and a software bridge

Fundamentally, three different types of software bridge are used in the MC design. It is shown how *data files* of different type, *shared memory access* and software *transformers* are implemented in the *SEWSAN MC* package. All of these are typical of a MC system. Regardless of the technique used, *ports* are required on both ends of a bridge and are to be considered as part of the bridge structure.

Requirements of a port on the model boundary : The port provides an interface from the model to the bridge. It is located within the model and has full access to the data structure of the model. It prepares the data for the bridge, i.e. writes or reads it to disk or shared memory structure. It can also route commands over the bridge.

A good bridge should be simple, transparent, understandable to users and powerful. A bridge is powerful if few instructions are required to issue commands or to obtain information from the connected model.

Use of a data file as a bridge : This is the most common way of exchanging data between models. The structure of the data file is based on a schema set out in the two connecting ports. This schema can be defined by an abstract data type which maps the data between the computer memory and the data file. However, an unstructured data file can also be used, where the structure of the file is not a direct image of the abstract data type. An example of this is a free format alphanumeric data file. Extensive parsing is typically required to read this file into the computer memory. Often the format allows for human readability. This property compensates an inefficient storage format.

Other structured data formats include binary files with record structure, where each record is an image of the abstract data type. Another technique is to use a database format. Database systems are optimized for random multi-user access of large volumes of data. Databases provide a structured and compact file format which can be viewed by many standard software programs.

The latest development in using a common data structure is the use of markup languages. The extensible markup language (XML) allows both the description of the data and the data itself to be stored in a human readable alphanumeric format.

In the evaluation of typical MC systems, alphanumeric text, binary data and database files are considered.

Use of memory as a bridge : The bridge can be implemented via the direct access of common (shared) memory resources or a communication system (such as piping or inter-application data communication). Examples of using memory as a bridge are evaluated in the MC system.

The use of common blocks was popular in older programming languages, such as FORTRAN, where it was the fastest way of transferring data between modules of implemented models. This was especially important as access to secondary storage during file transfer was generally an order of magnitude slower than the in-core operations. This technique is however only applicable where small volumes of data are required to be transferred via the bridge, as in-core memory is limited and therefore expensive.

Modern techniques allow for sophisticated software communication systems to be used to marshal the transfer of data between different models located on the same computer. These communication systems are very much dependent on the operating system and its underlying functionality. Communication

systems for the in-core transfer of data on the Microsoft platforms include the Dynamic Link Library (DLL), Dynamic Data Exchange (DDE), Object linking and embedding (OLE), Component Object Model (COM) and the enhanced version COM+ technologies.

Many of the modern standards also provide for the transfer of data between objects or applications located on different computing systems and running on different operating systems. This includes the Distributed Component Object Model (DCOM) and the Common Object Request Broker (CORBA) design. Data streams based on a communication protocol such as TCP/IP are used as bridges in the distributed scenario.

Use of a bridge as a transformer : The bridge can function as a transformer. Programming code in part of the bridge converts data from the input port to the output port. The conversion can be complex. An example of such a transformer is investigated for the MC approach.

Bridge complexity : Let there be n models and let every pair of models be connected with a bridge. Then the number of bridges to construct and maintain is $n(n-1)/2$. The number of bridges required to interconnect models is thus of the order $O(n^2)$. This is illustrated for a system with five models in Figure 2.3. Since the bridge manages all object communication between the connected models, a fairly complex communication system must be maintained. As soon as many models are involved, the overall complexity becomes a large problem for maintenance, extensibility and performance of the application. This should be avoided in a modern and complex software application.

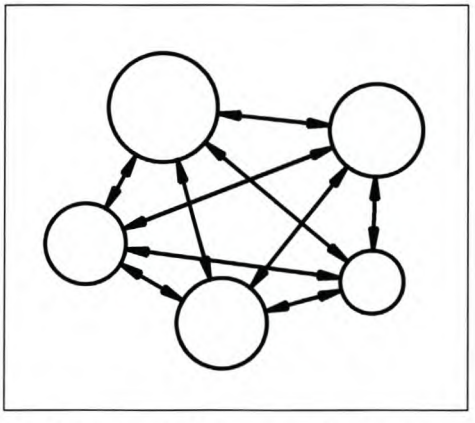


Figure 2.3: Five models connected via ten bridges

2.2.3 Evaluation of the MC approach

Inflexibility of design : Models as well as bridges are implemented by the software developer during the development phase. This is called an a priori implementation. Normally the user has no influence on processes which build connections between models. A few modern applications in the Civil Engineering field, e.g. *AutoDesk World* and *AutoCAD Map*, which can be regarded as host applications for graphical models of an engineering problem, provide some means of customizing the structure of the bridges. The user can develop a custom driver which is linked into the application to import data from a model, such as an external engineering analysis model. However, the complexity limits this functionality only to specialist users. The MC approach does not expose the object structure of the model to the user, and therefore limits the flexibility of design. This is also observed in the investigated MC system.

Stand-alone bridges of the transformer type transport data from one model to another via a transformer. This approach can be followed as long as the data file formats for both models are available. An example of such a bridge is investigated in the MC system.

Inefficiency of design : Object duplication due to the limitation of object name scope to models leads to inefficient systems in the MC approach. This has a negative influence on program maintainability, program extensibility, source code size and program execution time. Often objects of similar type (i.e. with similar class structure) are required in more than one model. As the communication between models most often does not directly permit transfer of objects, or the invocation of methods over model boundaries, duplication of objects and class structures occurs frequently in a MC system.

Existing models from diverse sources are frequently connected with bridges. Especially if the source code or the object structure of the model is not available, duplication of functionality cannot be avoided.

The investigated example of a MC system is analysed for an inefficient design due to object duplication.

Support for distributed computing : The MC approach provides only limited support for distributed computing. A distributed design is characterized by more than one process executing on at least two platforms. Often more

than one user is involved and concurrent usage can be required. When models are hosted on separate platforms, the system of communication between the models becomes a critical issue. The concept of the bridge must be extended to connect two platforms. It is clear that the exchange of data files alone is not sufficient. A transformer or special conveyor is required to facilitate and manage the communication between the models. The ports are now required to function as gateways between the platforms.

Inherently bridges are designed to transfer large volumes of data, such as the entire geographical or hydraulic model of an engineering system. The geographical and hydraulic model can be located on different platforms. An interactive system which permits small and continuous changes in the hydraulic model to propagate to the geographical model over the platform boundaries cannot be easily implemented.

The management of change propagation in a MC system over platform boundaries is very difficult if only data files are used. The inherent complexity of relationships between models in a MC system is drastically increased when platform boundaries have to be crossed.

2.3 Description of the investigated engineering process

The engineering process for the analysis and design of a sewer system using the *SEWSAN MC* software package as observed in a civil engineering consulting bureau is discussed in this section. It consists of the construction of the *Topology model* from the *CAD model*, the incorporation of the *Elevation model*, the construction of the *Topography model*, the construction of the *Hydraulic model*, the use of the *Visualization model* and the construction of the *Geographical model*. Figure 2.4 shows this model structure. The numbers are referenced in the discussion.

Construction of *Topology model* : The engineering process starts with the collection, evaluation and processing of data defining the topology of the existing sewer system as contained on paper or in digital plans of a *CAD model* hosted in a Computer Aided Design (CAD) system. The topology model contains a collection of connecting lines representing the sewer pipes and a circle representing the outfall manhole. It can be extended with text attributes

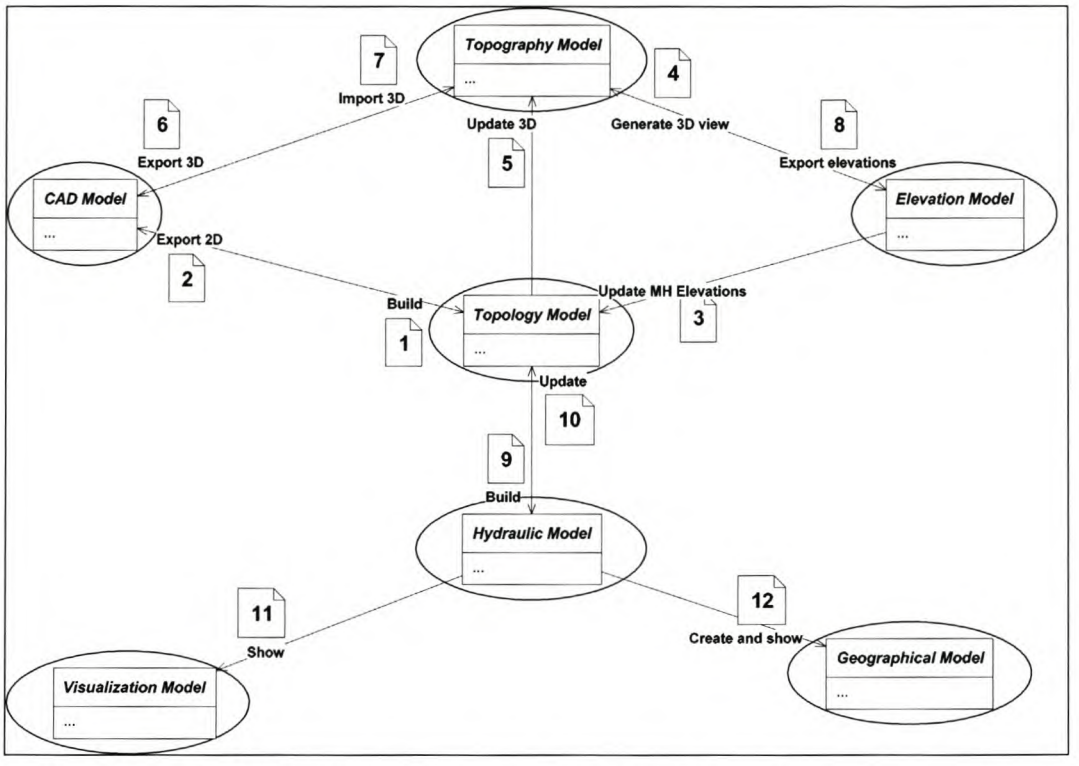


Figure 2.4: Collaboration diagram showing the engineering process in *SEWSAN MC*

such as manhole numbers as well as other key hydraulic information, e.g pipe diameters and the number of connected land parcels. This leads to the construction of the extended *Topology model* of the sewer network (1). The addition of new hydraulic structures to the *Topology model* can be specified graphically with the software package. After a cycle of topology building, the resulting *Topology model* can be exported to the *CAD model* (2).

Incorporation of the *Elevation model* : The *Topology model* must be augmented with an elevation model of the sewer system (3). The ground elevation at the nodes (manholes) of the sewer network can be obtained from an elevation model hosted by an external Digital Terrain Modelling (DTM) application. The elevations are required for the verification of captured lid levels, for calculating invert levels of new manholes and to calculate the pipe slope between connecting manholes. The calculated slopes can be edited interactively by the user.

Construction of the *Topography model* : From the *Elevation model* a 3D visualization of the terrain can be generated within the software package (4).

The *Topography model* can also be updated with the topology model, resulting in the 3D visualization of the sewer system (5). The model can be exported to external CAD Systems as 3D CAD drawing files (6).

Alternatively, a 3D CAD drawing file representing a topography model can be loaded into the *Topography model* (7). From this model an *Elevation model* can be exported (8), which can again be used to update the *Topology model* (3).

Construction of *Hydraulic model* : Alphanumeric input data, such as flow measurement data for the calibration of frictional coefficients of pipes and municipal water sales records, which are used to verify the domestic sewer production, can be collected and evaluated parallel to the construction of the *Topology model.* Then the *Hydraulic model* is constructed and the initial hydraulic analysis performed (9). Should errors in the topology or attribute data be found during the analysis, a correction is required. Small errors can be corrected directly in the *Hydraulic model* since it extends data contained in the *Topology model.* Larger corrections such as the addition of new suburbs initially not captured require that the *Topology model* be updated by the *CAD model* (1), which in turn provides an update to the *Hydraulic model* (9). An interaction between the *Hydraulic model* and the *Topology model* is therefore required. It is also possible to extract the *Topology model* from an existing *Hydraulic model* (10). Similarly an interaction between the *Topology model* and the *Elevation model* is required to update new elevations. For a new system with unknown pipe diameters, the design algorithm in the hydraulic software allows for the optimum diameter to be calculated from the pipe slope information. As a final step, the engineering results are shown in tables and XY-graphs, such as outflow hydrographs, as a function of time, or longitudinal sections and elevations as a function of pipe distance.

Use of the *Visualization model* : The *Hydraulic model* is visualized graphically in the *Visualization model.* At any stage after the topology of the sewer system has been entered, a 2D graphical visualization of the system can be obtained (11). Specific input or result variables are visualized by colouring pipes and manholes (nodes) according to colour legends. A zoom functionality permits the interactive enlargement of areas of interest. The *Visualization model* must be updated as soon as any changes are made to the *Topology model,* to the *Elevation model* and the possibly affected *Topography model* or to the *Hydraulic model.* The *Visualization model* does not allow the user to make any changes which influence other models.

Construction of the *Geographical model* : The presentation of results is also extended to a Geographical Information System (GIS) with the construction of the *Geographical model* (12). Here the final topology is exported together with associated databases containing all the input variables of the sewer system as well as results. Should any changes be made to the *Topology model*, *Elevation model* (and *Topographic model*) or *Hydraulic model*, the *Geographical model* must be updated to reflect the current state.

2.4 Functionality of the existing MC software system

This section describes the functionality of the *SEWSAN MC* package and explains the choice of programming languages. Figure 2.5 shows how the different applications are connected in the software package.

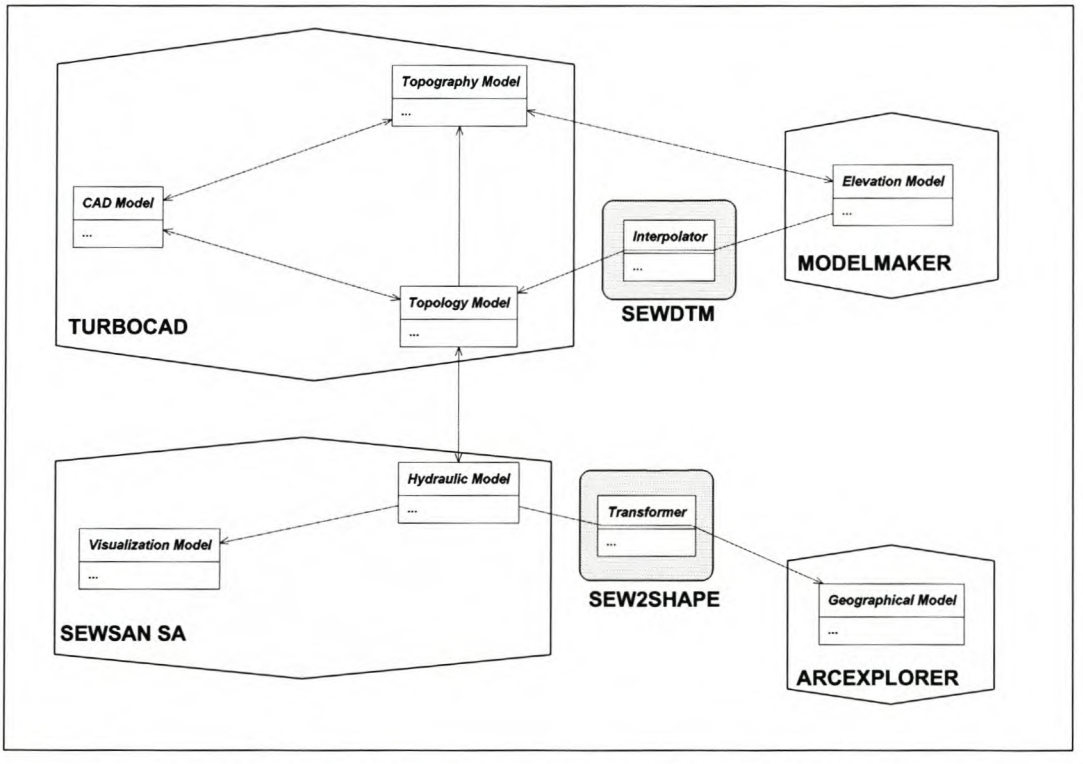


Figure 2.5: Applications comprising the *SEWSAN MC* package

Choice of existing MC software system : In order to show the shortcomings of the MC approach it was decided to analyse the functionality of a software package from the civil engineering field which is well known to the author. This package clearly exhibits the MC design, and the software itself

is not too complex, so that it does not distract the reader from the software technology issues.

It was decided to analyse an existing software package for the analysis and simplified design of urban sewer reticulation systems. As there is an interest to develop a modern sewer system analysis and design program based on the AC approach, the software example falls in the same engineering field as the pilot application-centred software system (AC system).

The original software package contained only the *Hydraulic model* and the *Visualization model* hosted in the *SEWSAN SA* program. The other models were implemented ad hoc using various third-party software products. For the purpose of this research and in order to be able later to compare results from the MC approach with the AC approach, it was decided to rewrite several of the interfacing applications and provide a common framework for the MC system, collectively known as the *SEWSAN MC* package.

The *TurboCAD* [20] 3D-CAD system was chosen to provide the common framework within which the *CAD model*, *Topology model* and *Topography model* are hosted. This is a modern low-cost CAD system which provides a similar programming interface as other high-end CAD systems. The port structures to the *CAD model*, the *Topography model*, the *Hydraulic model*, and the *Elevation model* are implemented within the programming environment of the CAD system. Control can be passed between the *Hydraulic model* and the *Geographical model*.

The *Elevation model* is hosted in an external software application dedicated to the construction and management of digital elevation models. The third-party product used in the bureau environment was the *Modelmaker* [26] program.

For the *Geographical model* any product capable of reading the GIS Shape File format from *ESRI* can be used. The freeware Java viewer *ArcExplorer* [13] was chosen.

Programming language used for model hosting : The core application, *SEWSAN SA*, which hosts the *Hydraulic model* and the *Visualization model* is a Microsoft Windows based program, written in Visual Basic for Windows [28], a programming language typically used by many engineering bureaus developing their own software. The visual programming environment simplifies the task of creating a functional and professional application. As all visual components in the programming language are based on OO technology, the program logic contained in the Visual Basic forms of the application

follows the OO principle. However, the program modules in the application also allow the use of traditional procedural code. Most algorithms of *SEWSAN MC* are implemented in the procedural style, as they have been inherited directly from code previously written for non-OO BASIC compilers. A sensible mixture of the styles results in an effective hybrid programming paradigm. The programming language used within the CAD system is Visual Basic for Applications. This is used by many applications which require integration on the Microsoft Windows platform.

Programming language used for transformer bridges : The stand-alone bridges with transformer functionality have been implemented in Object Pascal (*Delphi*). This includes the bridge between the *Topology model* and the *Elevation model* which interpolates elevations and the transformer between the *Hydraulic model* and the *Geographical model*.

2.5 Algorithmic background

This section describes the algorithms used for the hydraulic analysis and design in the *SEWSAN MC* program.

Contributor hydrograph method : The hydraulic simulation in the *SEWSAN MC Hydraulic model* is based on the *contributor hydrograph* method for sewer systems originally developed by Shaw [42]. A 24-hour contributor hydrograph for each pipe is generated by the number of stands (or unit parcels) served by the pipe and the type of area (e.g. higher income residential) which they represent. Other flow components of the contributor hydrographs are storm water ingress (rain water that enters the system illegally through manhole covers and at house connections), ground water infiltration (which enters the system through joints and connections at manholes) and leakage (due to leakage of the domestic plumbing systems). The hydrograph generated at each pipe is routed and accumulated downstream towards the outfall (most downstream) manhole using a time lag routing. Flow peaks in the system are attenuated due to the out-of-phase arrival of hydrograph peaks at a specific point. Bottlenecks in the system are identified at manholes where overflow occurs. Overflow occurs when accumulated flow at a specific manhole becomes larger than the capacity of the immediate downstream pipe.

Design option : The hydraulic software also has a design option to estimate the required pipe diameter so that no bottlenecks occur in the system. Bottlenecks in the system occur due to pipes of inadequate size. The required diameters of all inadequate pipes in the system are computed iteratively. The diameters of all inadequate pipes are increased stepwise until there are no more bottlenecks in the system. The minimum diameter is governed by three factors:

- Pipe capacity: The diameter must be such that the peak flow can be accommodated.
- Minimum velocity: The diameter must be such that full bore velocity is more than a user-defined minimum (typically 0,5 m/s).
- Upstream pipes: The diameter must be equal to or larger than the largest upstream pipe.

Interpolation algorithm : This algorithm is a software bridge located between the *Topology model* and the *Elevation model*. It interpolates the unknown elevation at a manhole from known elevations in a grid pattern around the manhole.

First a file is read which contains the known coordinates and elevations of a set of points. The coordinates of a set of points are specified where the elevation is to be calculated. For each point in this set, the algorithm finds the three closest points from the set of points with known elevations which form 3D triangle containing the unknown point.

2.6 Decomposition of existing software system into models

This section describes the components of the *SEWSAN MC* software package and their aggregation in models, partial models, sub-models and ports. The components are illustrated graphically in Figure 2.6. The functionality of the software bridges is discussed in the next section.

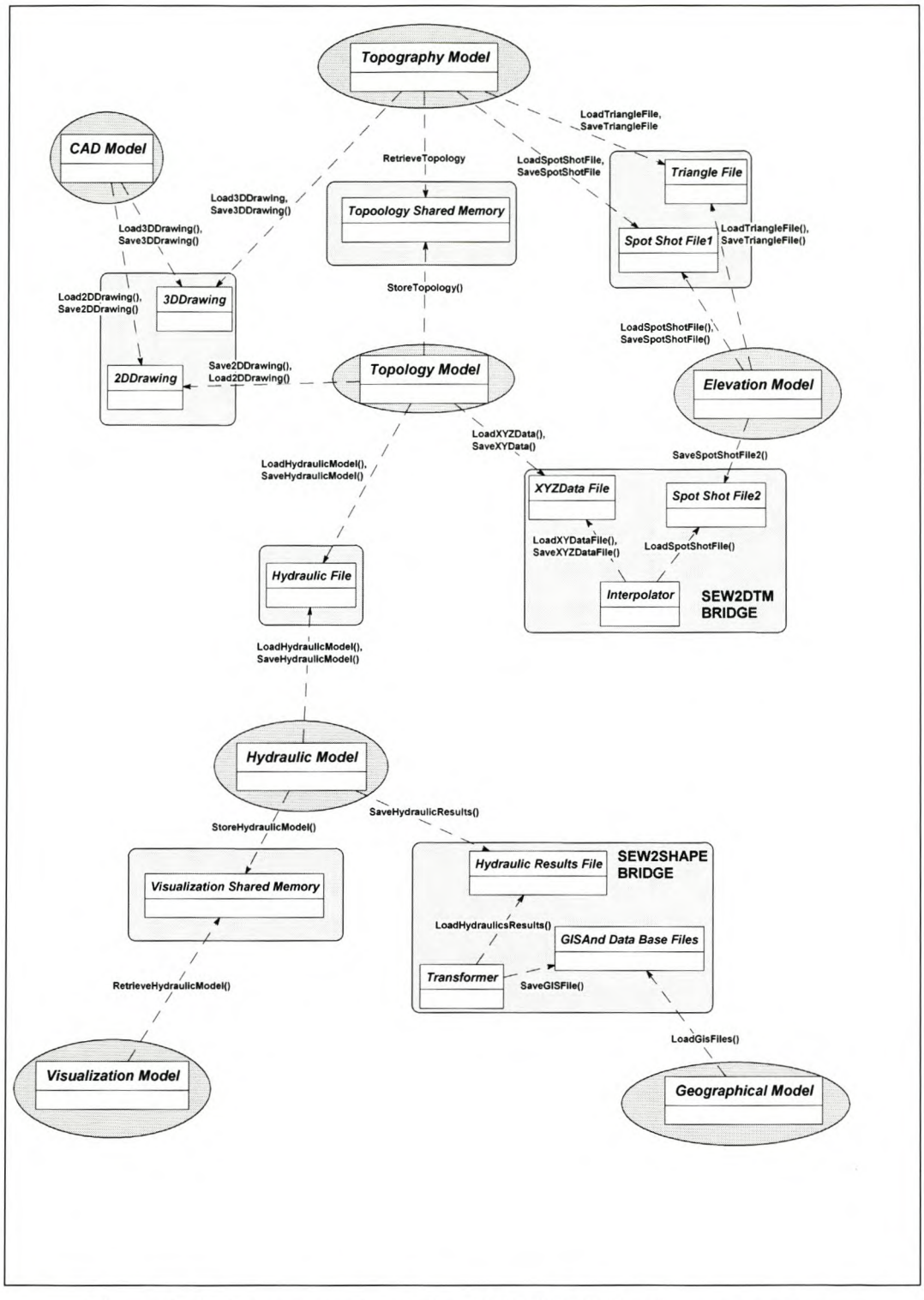


Figure 2.6: Components of the *SEWSAN MC* software package

2.6.1 Hydraulic model

The *Hydraulic model* can function as a stand-alone application and is responsible for the input, analysis, design and output of the hydraulic model. In

most cases where engineering problems of significant size are involved, the additional models, such as the *Topology model* and the *Elevation model*, are also required.

The *Hydraulic model* forms the core of the *SEWSAN MC* program. Figure 2.7 shows an overview of the classes. The *Hydraulic model* is sub-divided into several sub-models:

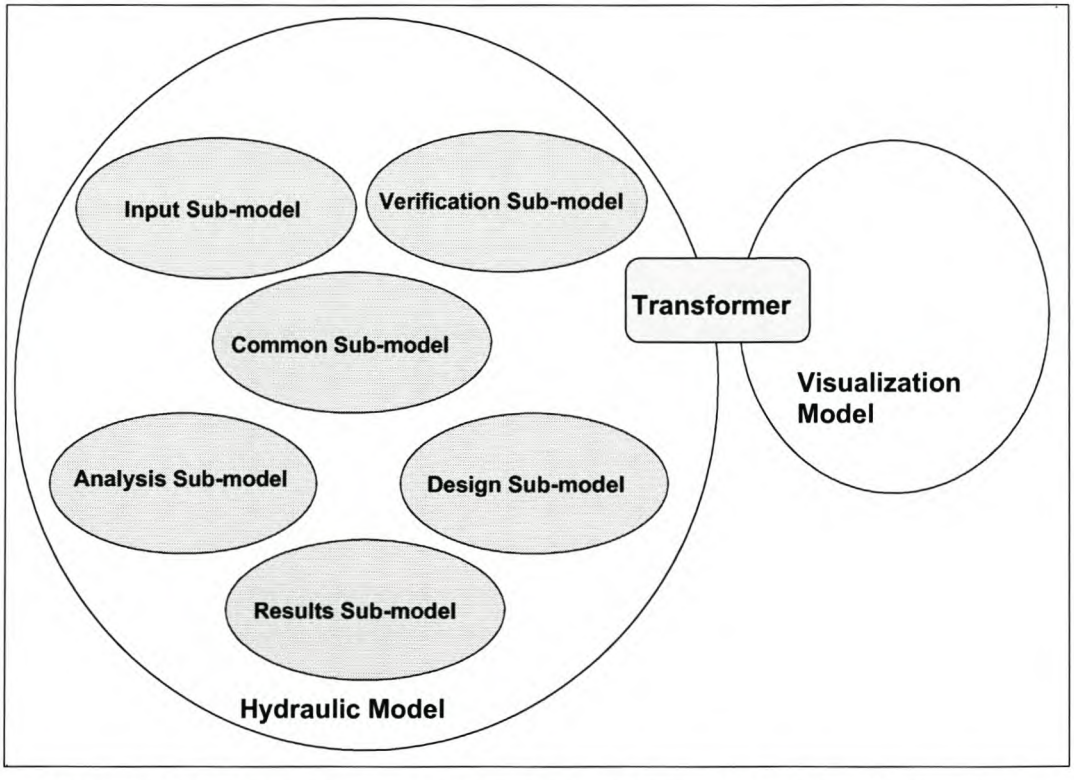


Figure 2.7: Diagram showing the Hydraulic and Visualization models

Input sub-model : The input sub-model permits the import of sewer networks from an external model (such as the *Topology model* via the SDF data file port) or the manual alphanumeric definition of a new model in data editors. It also presents time dependent input data in the form of XY-Graphs. The input sub-model allows the visual verification of the sewer system. As the graphical presentation is reused for the result presentation, it has been developed as a separate model, the *Visualization model*.

Verification sub-model : Another sub-model in the logical process of the Sewer Analysis and Design System is the verification sub-model. In this model the input is checked for errors before analysis or design. This model allows

interaction with the input model so that errors can be corrected after they have been identified in the verification sub-model.

Analysis sub-model : This sub-model executes the sewer analysis for a given input model after the model has been verified.

Results sub-model : This model represents the results of the analysis in the form of tables on the computer screen and on printer output devices, as well as in the form of XY-Graphs.

Design sub-model : This model manages the input of design criteria, presents XY-graphs of time-dependent design input data (for example calibration data) and executes the design algorithm.

Common sub-model : This model provides common functionality such as the persistent storage of the sewer system. It also contains the simulation system for routing flow through the sewer system, which is used by both the analysis and design models and the main menu, which controls the program execution of the application.

2.6.2 Visualization model

The graphical presentation for input verification of the sewer system as well as the display of analysis and design results, are implemented in a stand-alone visualization model. This model maintains its own name scope for classes and object variables which are independent of all other models. Figure 2.8 shows the visualization of Test Project 2.

2.6.3 Topology model

The *Topology model* contains topology data of the sewer network as captured from existing paper or digital plans. The topology model is stored persistently as CAD-based data contained in an industry standard 2D CAD drawing. The graphical data structure of this model, as well as the functionality available within the model is discussed in this section. Although this storage format is not very efficient, it does provide compatibility between most CAD systems.

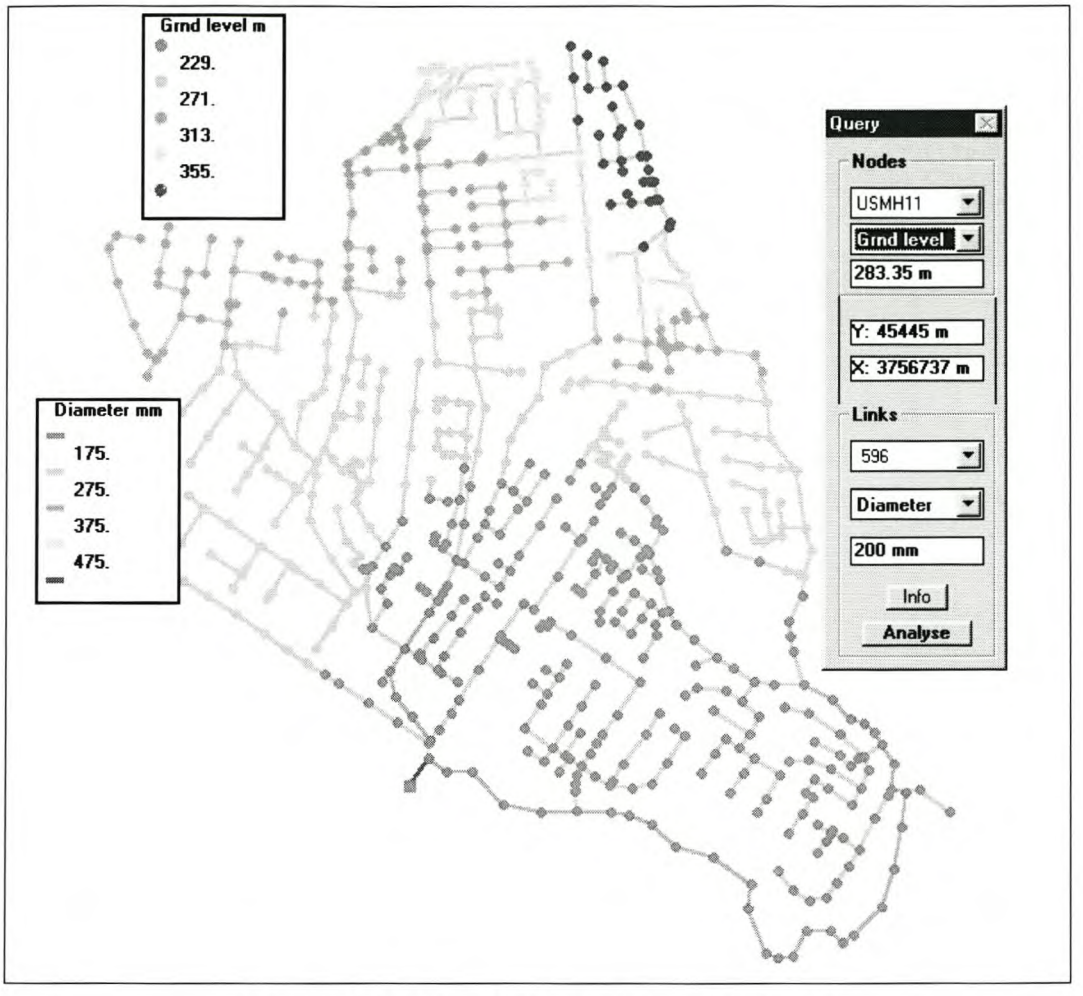


Figure 2.8: Visualization of Test Project 2

Data structure : The topology model is stored in the form of CAD entities on specific layers in an industry standard 2D CAD drawing. The basic CAD entities representing topological data are lines, text and circle entities. Figure 2.9 shows the data associated with the topology model as stored in a CAD drawing. The following structure details the entity structure:

- **Lines :** Lines are used to represent pipes of the sewer network. A connectivity between two pipes is implied when the endpoints of the two lines representing the two pipes have the same coordinates. Only lines which exist on a specific layer are considered to be part of the sewer model. This allows other lines to be used to describe parcel information or street layout. The definition of the pipe direction is of importance: a pipe is represented by a line whose first endpoint is the upstream manhole, and whose second endpoint is the downstream manhole. In order

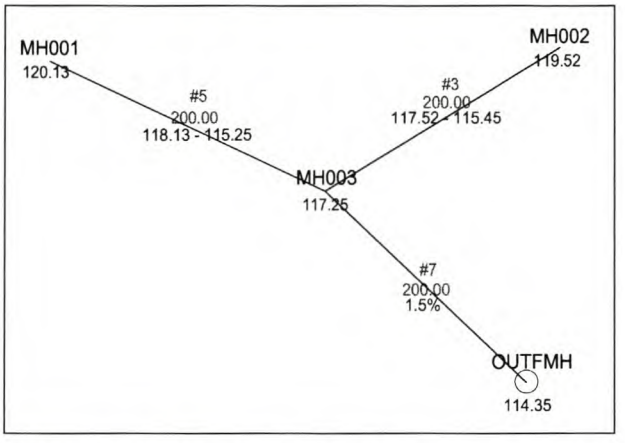


Figure 2.9: Topology of sewer network stored in a CAD drawing

to reduce the complexity of the drawing, no specific manhole or nodal entities such as circles are stored in the drawing. All manhole related information can be stored as text entities associated with the endpoints of the lines representing pipes.

- **Text :** Text entities are used extensively to provide additional information to the topology model. None of the text entities are required initially to build a topology model from a CAD model, since default and automatic settings are defined. For example if manholes (nodal points or end points of line entities) are not identified by a text entity, default identifiers are generated. Normally the user prefers to provide unique identifiers to manholes in the sewer network. These identifiers are strings contained in text entities. A system of layering and proximity is used. For example manhole text will only be captured from a specific layer, and the entity closest to a line endpoint (nodal point) defines the manhole identifier.

  Other information which is considered part of the extended topology model, such as the specification of the diameter of the pipes or the elevation at nodes is also stored or captured on separate layers in proximity of the referenced entity, either close to the endpoint of a line, or near the centre of the line.

  Special attention should be paid to the storage of invert levels. Every pipe in the sewer system has an upstream and a downstream invert level, i.e. the level at which the pipe connects to the manhole. Initially it was considered to store this information in close proximity to the manhole, i.e. at the end points of lines. A confusion could arise as to whether a specific text entity refers to the upstream or the downstream side of

a manhole, it was decided to write both the upstream and downstream invert levels at the centre of the referenced pipe, using a hyphen ("-") to separate the numeric values. The convention has been introduced that the text entity on the left side then always refers to the upstream invert level; the text entity on the right side, to the downstream invert level.

Finally the allocation of the number of standard land parcels producing sewage which should be allocated to a specific pipe can also be represented by adding a text entity on the correct layer, near the centre of the referenced pipe. The text is prefixed with a hash symbol.

- **Circle :** The circle is used exclusively to represent the outfall manhole. Normally only one circle can exist in the representation of the topology of the sewer system, as only one outfall manhole is allowed during analysis.

Functionality : Several functions are available to update or verify the consistency of the topology model. New pipes are specified in the sewer system with the draw line command on the correct CAD layer. The bottom manhole can be defined by drawing a CAD circle at the location of the manhole. By using the CAD Add Text command, additional information can be added to the topology model, as detailed in the preceding paragraphs.

In addition to the CAD commands, five functions are available. One of them provides a customized dialog box for the input of invert levels or slopes of a pipe. The option of transferring invert levels to a downstream pipe facilitates the construction of the extended topology model. Another function swaps the pipe direction, i.e. the definition of the first and the second point of a line representing a pipe. The third function allows the display of invert level error. An asterisk is drawn on the line where the pipe has an upstream invert level which is lower than the downstream invert level. It is also possible to draw arrows on the lines to show the direction of definition of the pipe. Finally, a dialog box permits editing of the number of parcels associated with a pipe.

These functions have been written as subroutines in the embedded Visual Basic for Applications (VBA) programming language. The additional functions are triggered with a menu bar, which appears on top of the CAD environment.

Construction procedure: The topology model of the sewer system within the *Topology model* is constructed as follows:

- **Load existing drawing :** An industry standard drawing file can be

loaded via the bridge to the *CAD model* by using the file input port. This drawing generally requires editing to correct errors in the topology.

- **Edit the drawing :** The editor is used to move drawing entities to the correct layers, and correct general drawing errors. Additional text or line entities can be added, such as known invert levels or manhole top elevations. Manholes can be moved by selecting the end point of a line and dragging it to a new position. All connected pipes must be moved accordingly. Incorrect pipes can be deleted. Elevations can be updated using the port to the elevation interpolation bridge. This port is accessible from the custom menu bar.
- **Verify the topology model :** The pipe orientation definition can be checked against the invert levels. Corrections to the pipe orientation can be made. Further verification of the model will take place once it has been exported via the port to the bridge connection with the *Hydraulic model*. The *Hydraulic model* will check for errors in the sewer system, such as duplicate identifiers, invalid invert levels etc. Once these errors are reported in the *Hydraulic model* they can be corrected in the *Topology model*.
- **Export the topology model :** When the model has been verified, it can be exported (using the file output port to the industry standard drawing file) via the bridge to the *CAD model*. The final model is exported to the *Hydraulic model* via the dedicated port and bridge for analysis and design operations.

Connectivity with other models: The CAD environment hosting the *Topology model* provides the custom menu bar from which many other models can be accessed. The *Topology model* is directly connected via ports and bridges to the *Topographical model*, the *Elevation model* the *CAD model* and the *Hydraulic model*. The ports to all these models and the bridge to the *Topographical model* are written in the VBA programming language and contained within the CAD environment.

2.6.4 Elevation model

The *Elevation model* contains the elevation (z ordinate levels) of the terrain within which the sewer system is located. Typically, the professional surveyor includes data from survey books, total stations, stereograph photos, aerial

photography and data obtained from satellite measurements [6]. A regular grid of elevation points is created and then exported using the surveyor application via a port to an ASCII text file. This file contains a list of XYZ elevation points. The SEW2DTM bridge has a port to this file to read it and then interpolate the elevations at the manholes obtained from a port to the *Topology model*.

A triangular mesh representing a surface model of the terrain can also be exported via a port to an ASCII text file bridge. The *Topography* model provides a port to read the data again, and present it in the CAD environment.

2.6.5 Topography model

This model contains the 3D visualization of the terrain containing the sewer system, as well as the sewer system itself. The 3D rendering is done by the CAD environment. Standard options include specifying parameters for the texture, colour and thickness of elements to be rendered. Additionally draft and final quality is available. After completion, the direction and magnification for the window to the 3D view can be set interactively by the user.

Export via a bridge from the *Topology model* provides the data of the sewer system. An import function via a port to the *Elevation model* or via a port to the *CAD model* allows the input of data defining the 3D terrain model to this model. The *Topography model* can also function as an intermediary model for the extraction of elevation points from a topography model ported from the *CAD model* to the *Elevation model*.

Figure 2.10 shows the topography for Test Project 1.

2.6.6 Geographical model

The *Geographical model* is used to present results from the sewer system analysis with spatial queries. Third-party applications based on GIS standards can host the *Geographical model*. The model has its own port which connects with the SEW2SHAPE bridge, situated between the *Hydraulic model* and the *Geographical model*.

The port must be able to write standard GIS files based on the ESRI Shape file standard and dBase IV database format. This format is commonly used by GIS applications. As the GIS application is a third-party product, the structure of the application is not relevant to this research. In the current

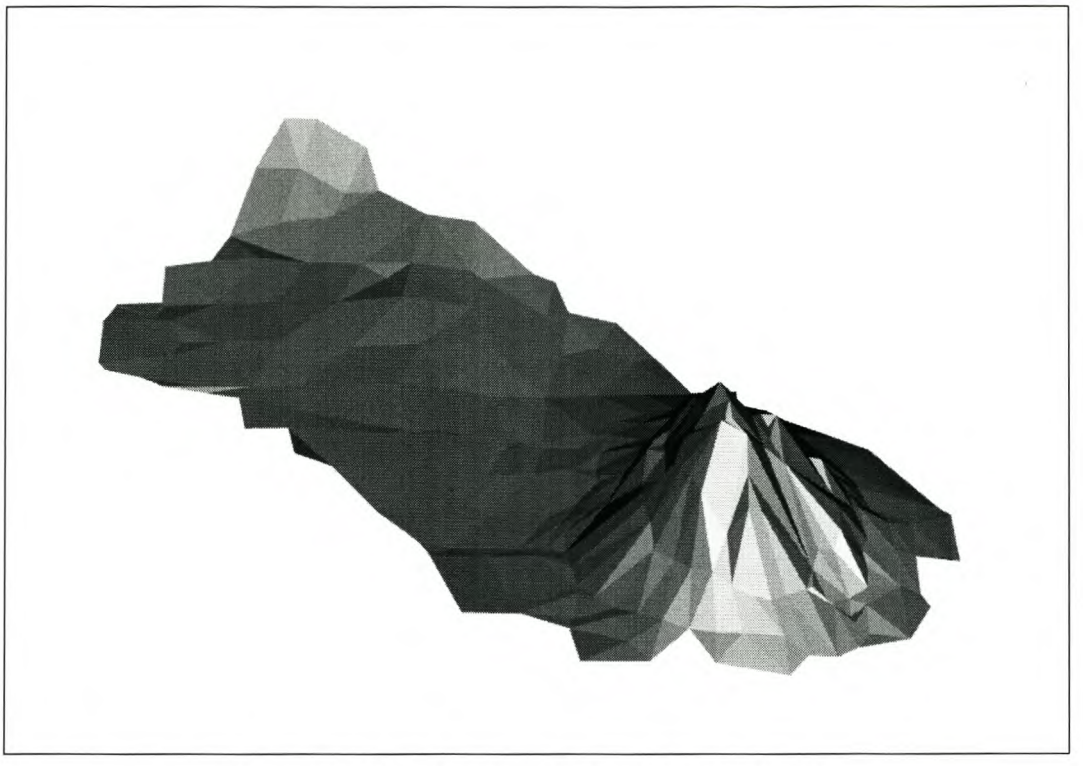


Figure 2.10: Topography for Test Project 1

implementation of this model, it is not possible to make changes to the *Geographical model* as the bridge structure responsible for migrating the changes back to the hydraulic model would be too complex. Support for remote connections (for example via the Internet to the GIS data source) is not provided in this implementation.

2.7 Bridges between models in the existing software system

The interaction between models within the scope of the *SEWSAN MC* application environment through the use of different types of data files or the use of active software bridges is detailed in this section. The structure of the interfacing systems (files, shared memory and software transformers) is presented.

2.7.1 Use of data files

The files in which data are transferred between the model of a MC application can be structured or unstructured. All records of a structured file consist

of the same sequence of data types: only the values of the variables differ between records. Text files, which are composed of readable characters, are distinguished from binary files, which contain the values of the variables in internal computer format. A special variant of the binary data file is the database file. The database file has a fixed record structure, i.e. each record is based on a data type and is designed for fast random access through the use of additional index files. Furthermore the database file is compatible with many industry standard applications. The following examples are taken from the *SEWSAN MC* application environment.

Structured text files : The *.YXZ* file is an example of a structured text file in the *SEWSAN MC* application environment. This file is used by the *SEW2DTM* transformer bridge (located between the *Topology model* and the *Elevation model*) to obtain data from the *Elevation model*. The file contains a textual representation of the coordinates and the elevation at the points located on a regular grid containing the sewer system.

Typical rows of the file are shown in Table 2.1. The first row is ignored during input.

Table 2.1: Typical rows of the .YXZ structured text file

Y	X	Z
43600	3755400	303.1
43600	3755600	295.2
43600	3755800	282.5
43600	3756000	276.3
Size: 25 bytes per line with up to 2 000 000 lines		

The size of the file is not directly related to the size of the sewer system. It is a function of the mesh density and the size of the terrain described in the *Elevation model*. The *SEW2DTM* transformer supports the storage of up to 2 000 000 rows.

Structured binary files : The *.ELV* file is used between the *Topology model* and the *SEW2DTM* transformer bridge (located between the *Topology model* and the *Elevation model*). The file is exported from the *Topology model* which contains only the X and Y coordinates of the manholes in the sewer model. Then data is read by the transformer, updated with the elevation (Z value) and written to the same file. The *Topology model* then imports the updated file.

Since the file is binary, it is not human readable. The file is based on an abstract data type, which is presented in Table 2.2 as it appears in the *Topology model* (Visual Basic) and as it is defined in the *SEW2DTM* transformer (Object Pascal). Different programming languages can be used to access the structured binary file if they have the same underlying data types.

Table 2.2: Abstract data type for the .ELV structured binary file

Visual Basic :
`Type CoordType`
`  X As Double`
`  Y As Double`
`  Z As Double`
`End Type`

Object Pascal :
`CoordType=packed record`
`  X,Y,Z:Double;`
`end;`

Size: 24 bytes per record x total number of manholes

Structured database files : The *.DBF* files are used for data transfer between the *SEW2SHAPE* transformer bridge (located between the *Hydraulic model* and the *Geographical model*) and the *Geographical model*. Three files are exported from the transformer bridge. They contain the pipe related input data, manhole related input data and pipe result related data of the *Hydraulic model*. The database files have key fields which are referenced by corresponding Shapes files (*.SHP*) with the same name.

The structure of the three files is outlined in Table 2.3.

Table 2.3: Structure of the database files.

Field Name	Type	Size	Decimals
Manhole input table :			
MH_NAME	Char	16	-
Y_COORD	Num	20	4
X_COORD	Num	20	4
GR_LEVEL	Num	20	4
S_INVERT	Num	20	4
TOTAL: m Manholes	x	96	bytes
Pipe input table :			
S_MANHOLE	Char	16	-
Y_COORD	Num	20	4
X_COORD	Num	20	4
GR_LEVEL	Num	20	4
S_INVERT	Num	20	4
E_MANHOLE	Char	16	-
E_INVERT	Num	20	4
SLOPE	Num	20	4
I_S	Char	4	-
DIAM	Num	20	4
LENGTH	Num	20	4
C_L	Char	4	-
MANNING	Num	20	4
E_N	Char	4	-
INFIL	Num	20	4
ADD_LENGTH	Num	20	4
S_TYPE	Char	6	-
UH1 .. UH10 10x:	Num	20	4
TOTAL: n Pipes	x	470	bytes
Pipe results table :			
S_MANHOLE	Char	16	-
E_MANHOLE	Char	16	-
DIAM	Num	20	4
LENGTH	Num	20	4
SLOPE	Num	20	4
MANNING	Num	20	4
LAG	Num	20	4
FULL_FLOW	Num	20	4
FULL_VEL	Num	20	4
MAX_FLOW	Num	20	4
MAX_VEL	Num	20	4
SPARE_CAP	Num	20	4
OVER_FLOW	Num	20	4
E_N	Char	4	-
TOTAL: n Pipes	x	256	bytes

Unstructured text files : Several unstructured text (ASCII) files are used for the exchange of data in the *SEWSAN MC* application environment. The ASCII equivalent of the *.DWG* drawing file, the *.DXF* file, can be used as an alternative for the import and export to the *CAD model*. The *.DXF* file is an example of an unstructured text file. The *.DXF* file is less economical and file sizes as well as loading/saving times are longer than for the binary *.DWG* format. However, because of its text structure the .DXF file is accessible to third-party developers. The structure of the *.DXF* file is complex, and falls outside of the scope of this research.

Another unstructured text file, which will not be considered further, is the *.RES* file. This file is exported from the *Hydraulic model* for import by the *SEW2SHAPE* transformer bridge. It contains the results of a hydraulic analysis.

The unstructured text based *.TRI* file is used to exchange data between the *Elevation model* and the *Topography model*. Its structure originates in the application hosting the *Elevation model* and can vary. One variation of the file contains only the definition of triangles forming the topographical model without additional data. Typical lines of the file are shown in Table 2.4. Note that the structure requires that the first node of a triangle is to be repeated.

Table 2.4: Typical lines of the .TRI unstructured text file

```
No Topo
10039.8763551521,56072.7202846054,195.48929988888
10060.7607640921,56078.8253992284,193.30119619699
10058.1715315063,56105.5103002358,203.77688433845
10039.8763551521,56072.7202846054,195.48929988888

9413.79371416861,56269.7822844367,157.38455114057
9447.82584306483,56290.6728766598,165.46806374323
9391.90152009054,56269.8044848554,157.03309407088
9413.79371416861,56269.7822844367,157.38455114057
```

Size: typical 200 bytes per triangle
x unlimited triangles

The unstructured text base *.SDF* file forms the core bridge between the *Hydraulic model* and the *Topology model* as well as between the *Hydraulic model* and the *SEW2SHAPE* bridge to the *Geographical model*. It is also used for the persistent storage of data from the *Hydraulic model*.

The file is a free format text space delimited (ASCII) file. Except for the prologue and epilog, the structure is mostly based on a fixed data type. A typical line of the file is shown in Table 2.5. One line from the prologue is also shown. The lines wrap over four rows (indicated by ..). The files contains approximately 180 bytes per pipe in the sewer model.

Table 2.5: Typical rows of the .SDF unstructured text file

Type Code Y-Coord X-Coord Gr.Lev Invert .. Code Invert Diam Length C/L Man E/N Infil S.Typ .. UH1 UH2 UH3 UH4 UH5 UH6 UH7 UH8 UH9 UH10 .. Slope S/I Addl
PIPE USMH0 45289.808 3757087.012 239.590 0.000 .. DSMH0 0.000 200 17.137 C 0.012 E 0.050 1 .. 1.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 .. 0.500 S 0.000
Size: typical 180 bytes per row x total pipes in sewer system

Unstructured binary files : An example of an unstructured binary file in the *SEWSAN MC* application environment is the industry standard *.DWG* file. This file is generally used to transfer data from the *CAD model* to the *Topography model* as well as the *Topology model*. The format is propriety to AutoDesk, but the OpenDWG alliance is actively involved to publicize (reverse engineer) this format. It can be considered as the de facto standard for exchanging 2D and 3D drawings. However, as it is not regulated by a standard committee, incompatibilities exist between the different vendors and between the different versions of the drawing file format. The details of this file structure will not be investigated further as part of this research.

The *.SHP* format is used together with the database file to transport the input and output data of the *Hydraulic model* from the *SEW2SHAPE* transformer bridge to the *Geographical model*. This unstructured binary file format is propriety to ESRI. The file format is documented by ESRI [12], and can therefore be used to write input and output ports.

2.7.2 Use of memory as a bridge

If two application are developed by the same company and the source files for both applications are available, the transfer of data via data files is frequently avoided. The advantage of using in-core memory for frequent transfer of data

between models is apparent. In-core memory is several orders of magnitude faster than out-of-core memory (hard disk etc). However, the shared memory access is not as open to future development. Any changes made to the data structure of the one model normally require that changes must be made to the other models connected via the shared memory system. Usually this requires changes at source code level and thus the recompilation of executable code. Two examples of the use of memory as a bridge exist in the *SEWSAN MC* application environment. Such a bridge exists between the *Topology model* and the *Topography model*. Here the *Topology model* can directly access the data structures of the *Topography model* and use the data to draw the sewer network in 3D space. The interaction between the *Hydraulic model* and the *Visualization model* located in a special *Transformer module* between the two models, is investigated next.

2.7.3 Transformer module

The transformer module in *SEWSAN MC* supplies the following functionality: In the *Hydraulic model* classes and objects are available which define the sewer topology and pipe attribute data. In the *Visualization model*, several of these classes and objects are also required, but in a different format. For example in the *Hydraulic model*, the data is stored in a format suitable for the hydraulic analysis, whereas in the *Visualization model*, the vector format is favoured.

Functionality : The *Transformer module* in the *SEWSAN MC* application functions as a one-way adapter in the direction from the *Hydraulic model* to the *Visualization model*. It propagates changes over the model boundary and overcomes name scope limitation (for class names and object identifiers) by mapping the classes and object identifiers between the models. Another function of the bridge is the transfer of commands between the models.

Synchronization : Objects located in different models, which represent the same real-world object, must be synchronized in order to ensure that consistency is maintained. In the *SEWSAN MC* application a simple approach is followed: Since the *Visualization model* only presents results (i.e. no changes can be made in the *Visualization model*, which would influence the other models) the *Visualization model* is only updated whenever changes are made to the *Hydraulic model*.

The transformer will be analysed in depth in the next section.

2.8 Analysis of the MC design structure

In this section an analysis of the MC approach will show the deficits in the design. The previous section described different types of bridges in a MC design. However, the *Transformer module* as implemented between the *Hydraulic model* and the *Visualization model* is chosen since it exhibits the most complex design.

The analysis does not focus on the actual implementation in *SEWSAN MC* but evaluates the general aspects of the *MC* design, which is also applicable to MC software systems for other hydraulic engineering problems. In a typical MC system, a *Hydraulic model* for the hydraulic analysis and a *Visualization model* for the graphical presentation is connected via a *Transformer module*. The models have independent name scopes, characteristic of a MC approach. The communication of data and commands between the models takes place via a *Transformer module*.

The two engineering models are implemented in separate modules (or packages), namely the `HydraulicModel` and `VisualizationModel`. The transformer is contained in the `TransformerModule` package. Figure 2.11 shows the class diagram for the typical MC system. The Java implementation for the system is found in Appendix C.

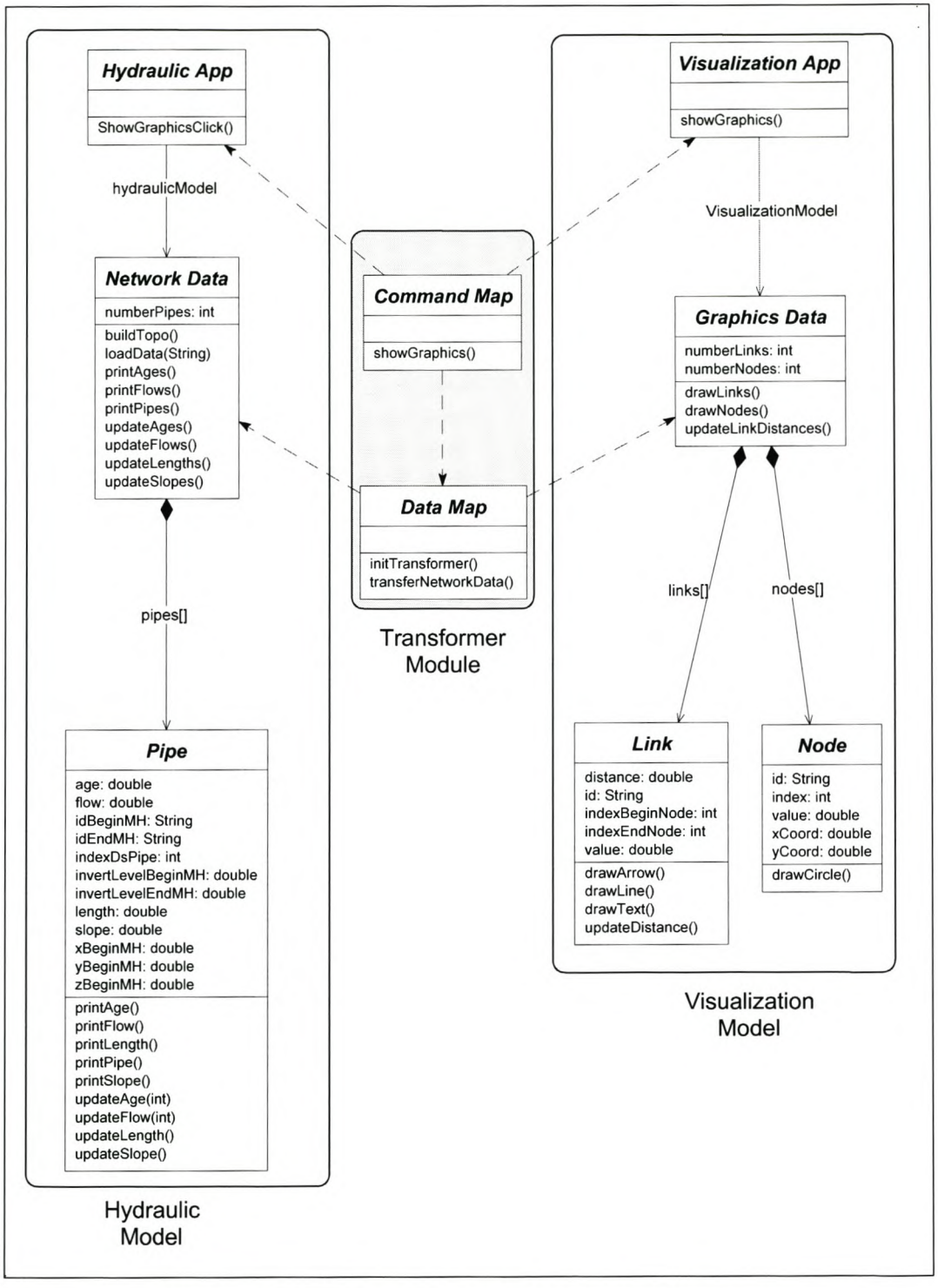


Figure 2.11: Class Diagram of typical MC system

Dynamics of bridge design : In order to analyse the dynamics of the transformer bridge, a UML based Collaboration diagram shown in Figure 2.12 is used in the following discussion. The flow of process is sequentially numbered and objects are identified with their class names.

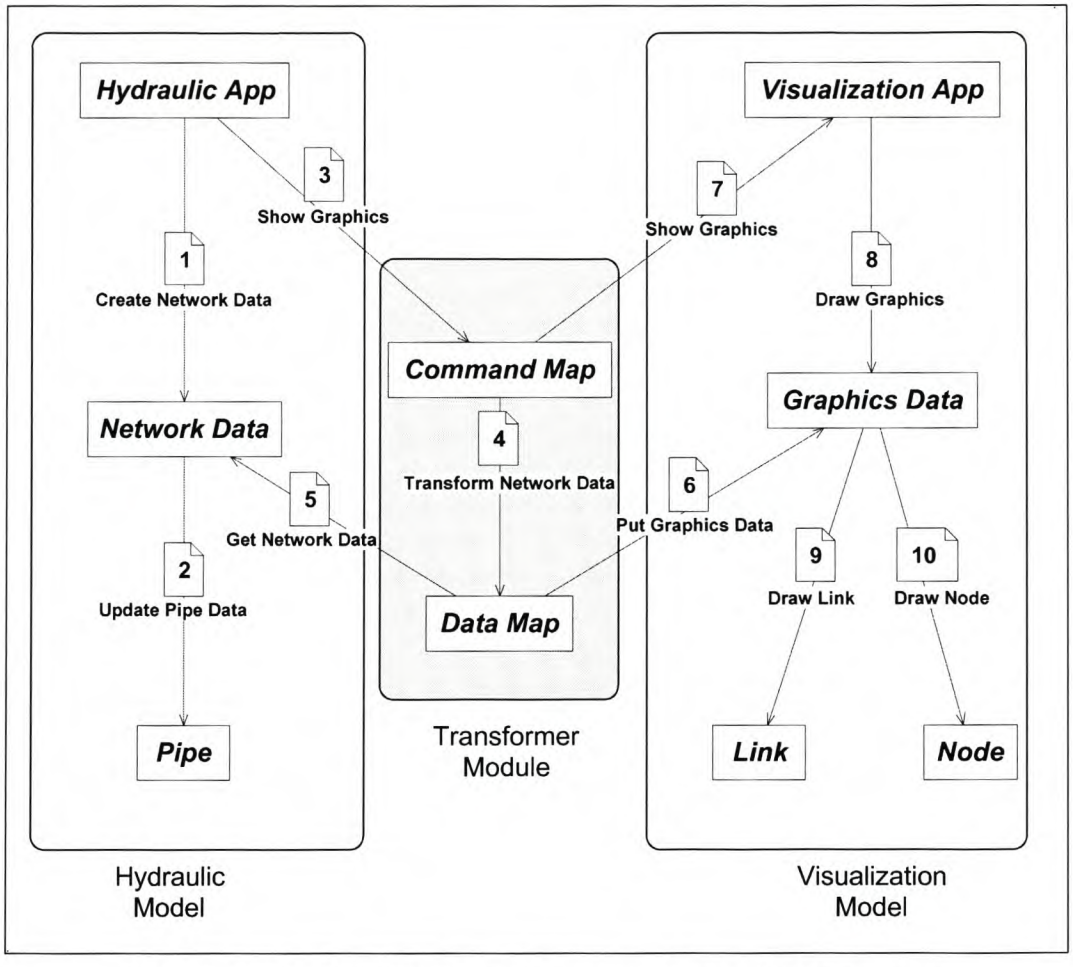


Figure 2.12: Collaboration diagram of typical MC System

Execution starts at the `HydraulicApp` object in the `HydraulicModel` package. During initialization it creates the `NetworkData` object (1) as well as subordinate `Pipe` objects (2). The `NetworkData` object contains an array `pipes[]` of all instantiated pipe objects, as well as methods which operate on the array of `Pipe` objects such as `updateLengths()` to calculate the lengths of all pipes in the model and `printFlow()` to print the resulting flow in all pipes. The `showGraphicsClick()` method in `HydraulicApp` (3) is executed when the user requests a graphical visualization. This method transfers execution to the `CommandMap` object in the `TransformerModule` package, which firstly calls `initTransformer()` to initialize the transformer, and then calls

`transferNetworkData()` (4) in the `DataMap` object. This method maps the relevant data from the `NetworkData` object in the `HydraulicModel` package (5) to the `GraphicsData` object in the `VisualizationModel` package (6).

In order to achieve this, the `DataMap` object is dependent on the `NetworkData` object in the `HydraulicModel` and on the `GraphicsData` object in the `VisualizationModel`. Then the `showGraphics()` method in the `VisualizationApp` object of the `VisualizationModel` (7) is called, which in turn calls `updateLinkDistances()` (to update the lengths of the links in the `VisualizationModel`) and then `drawLinks()` and `drawNodes()` in the `GraphicsData` object (8). These methods in turn call the `drawLine()`, `drawArrow()` and `drawText()` methods (9) for the links and `drawCircle()` for the nodes in the `VisualizationModel`. From the class diagram it can be seen that the `links[]` and the `nodes[]` arrays are contained in the `GraphicsData` class.

Topological structure : From the two figures it can be seen that the *Hydraulic model* and the *Visualization model* use different techniques to store the topology and geometry. This is typical where models are developed for different purposes.

In the *Hydraulic model* the topology and geometry data are stored in `Pipe` objects. Each `Pipe` object has a reference to the index of its downstream pipe in the `pipes[]` array. This linked-list data structure is adequate to model a tree data structure and only the geometric data (coordinates) of each beginning node is stored in the pipe object. The last pipe in the tree (terminating pipe) defines the end of the pipe system. This pipe has a null `idEndMH` and no associated length.

In the *Visualization model* the topology and geometry data are stored in `Link` and `Nodes` objects. This allows the generic display of a network of links to include for example loops of links. The `Link` object contains an `id` and the index to the begin node object (`indexBeginNode`) and end node object (`indexEndNode`) of the link. The `Node` object then contains an `id`, its `index` in the array of `nodes[]`, and the X and Y coordinates of the node.

Tree traversals : The `updateAge()` method performs a typical pre-order tree traversal and calculates the age of all pipes (represented by their length) from the terminating node upstream in the pipe system. The `updateFlow()` effectively performs a post-order tree traversal, i.e. downstream towards the

terminating pipe. The two operations are performed using nested loop operations as shown in Java source code traversal algorithm in Appendix C.4. The two typical operations for data trees will also be analysed for the AC design.

With the structure of the typical MC approach now set out, the next part discusses the limitation of the design based on the above discussion of the typical MC system and the fully implemented *SEWSAN MC* system.

2.8.1 Limitation of object identifier scope

An object is an instance of a class. Objects require an identifier which distinguishes one instance from another within a specified name scope. The name scope in the MC design is limited to the scope of the model. Therefore objects can only be identified and accessed uniquely within this name scope.

Objects can be identified uniquely within a name scope using different techniques. A single object can be assigned to a unique string, such as `outFallManhole`. A set of objects is typically identified by an identifier for the collection (such as `pipes[]`) and an index to the specific element. This identifier can be a positive integer (such as in the case of an array data structure) or another string identifier in the case of a hash table (dictionary) data structure. A hash table data structure maps string identifiers to object references in an effective way.

The typical MC system makes use of arrays as storage structure for sets of objects, namely `pipes[]` in the *Hydraulic model* and `links[]` and `nodes[]` in the *Visualization model*. This is common in older programming languages where extended data structures are not available.

The `pipe[]` array is bound by the name scope of the *Hydraulic model* package and can therefore not be accessed directly outside its name scope. The same applies to the `link[]` and `node[]` arrays in the *Visualization model* package.

The access of objects over the model boundaries functions only in the typical MC system if marshalling via a bridge structure takes place. The *Transformer module* provides the functionality to map commands via methods in the `CommandMap` class and data via the `DataMap` class between the two models.

In an alternative design using ports and data files, such as between the *Hydraulic model* and the *Topology model* in *SEWSAN MC*, the transformer is moved to the port methods in each of the models, and the data file represents

a neutral format for exchanging data via secondary storage.

Regardless of the techniques used to interconnect models, the data structures must be maintained throughout the lifetime of the application.

2.8.2 A priori implementation by software developer

The MC approach does not allow the user of the application to influence the composition or connection of models. As the application is designed a priori and coded by the software developer using rigid bridges to interconnect models, it is for example not possible in the *SEWSAN MC* application environment to exchange the existing *Visualization model* with either an alternative model (say a 3D visualization system) written by the original software developer at a later stage, or even by a third-party developer, without access to the full source code. The transformer bridge as implemented by the Transformer module cannot be designed transparent enough as it is too tightly interwoven with the global data structures of the two models.

Other forms of bridges, such as data files, offer greater flexibility. If an existing bridge data structure provides all the information required for the alternative model, it would be possible to develop an isolated model which uses an existing data bridge. However, should the data provided by the existing bridge not be sufficient, there is no way for the developer of the new model to implement a new bridge to the existing model. The existing models do not allow the customization of their ports in order to provide new bridges to new models.

2.8.3 Object duplication

Each model and sub-model in the MC approach focuses on the engineering task at hand and implements a solution for that task. For example, the *Hydraulic model* accesses data in its name scope and operates analysis and design algorithms on the data. The *Visualization model* uses data available only in its names scope and presents it graphically.

As the models contain data of the same engineering problem it is evident that a duplication of classes and objects will take place. This can be seen in the duplication of attributes between the objects representing the basic hydraulic elements (pipes and manholes) between the *Hydraulic model* and *Visualization model*. For example in the typical MC system, the endpoint coordinates of

pipes (or the coinciding manhole coordinates) are effectively duplicated in the `xBeginMH` and `yBeginMH` attributes of the `Pipe` class in the *Hydraulic model* and the `xCoord` and `yCoord` attributes of the `Node` class in the *Visualization model*.

The duplication of classes also results in a duplication of functionality. For example the method to compute the length of pipe from the coordinates of its endpoints should only be implemented once in a class Pipe. In the typical MC system it is implemented once with the name `updateLength()` in the `Pipe` class of the *Hydraulic model* for the hydraulic calculations and once with the name `updateDistance()` in the `Link` class of the *Visualization model* for the placement of text and symbols.

The resulting redundancy influences the performance and maintainability of the application.

2.8.4 Program maintenance

The periodic elimination of programming errors is called program maintenance. If objects are duplicated, i.e. the same real-world object is contained in more than one model, it is likely that a part of the object functionality is replicated. The length calculation routine in the *Hydraulic model* and the *Visualization model* are an example. Should an error be found in the calculation logic of this routine in one model the same error may exist in the other model. The other models must be verified. This duplication of object functionality leads to additional program maintenance.

Debugging (i.e. the process of using special software to trace the flow of execution) is limited in a MC approach. As the debugging is limited to the executable application, only one model can be debugged at a time. This means data flow between models cannot be followed interactively. For example, in *SEWSAN MC* the flow of execution from the *Topology model* contained within the CAD Environment cannot be traced from the CAD Environment to the *Hydraulic model* contained in the *SEWSAN SA* application, since the *.SDF* data file bridge prohibits this.

2.8.5 Program extensibility

Should the topological definition of sewer pipes be changed to include intermediary points between manholes, a new algorithm is required to calculate

the pipe length. This change in the attribute data structure of a pipe object requires a change in the method which calculates the length of pipes, now including the intermediary points.

Should this functionality also be required in other models, the equivalent methods in pipe or topology classes need to be re-implemented. In the case of the *SEWSAN MC* application environment this means that the *Hydraulic model*, *Visualization model* and the *Topology model* must be updated at source code level.

2.8.6 Suitability for distributed computing

As the *SEWSAN MC* application is not designed to be used in a distributed computing scenario, the following discussion is based on a hypothetical situation. The ability to spread an application environment over several platforms located at different physical locations has become a very important design consideration, especially in light of the fast-paced development of the Internet. Several factors make a MC approach not suited for a distributed scenario:

Name scope limitation : As discussed previously, the MC approach limits the name scope of objects to the containing model. In a distributed computing scenario this means that models are located on different remote platforms. Objects cannot be transferred directly from one model to the other as the name scope of the object is limited to its model and platform. However the use of bridges, such as data files which must now be transferred between platforms, can be used.

Requirement for bridges : The functionality of software bridges must be extended so that they also interconnect models located on different platforms. Issues such as the marshalling of remote procedure calls over the network and security aspects must be handled in the extended bridge. If several models are to be connected over the distributed network, the complexity rises quickly. It is again of the order $O(n^2)$, where n is the number of models. The classical use of bulky data files as bridges cannot be a recommended for the distributed design, as the communication channel is still a premium resource and bandwidth is expensive and limited. Only a new design, where small volumes of data (and possibly code) encapsuled in objects which are exchanged

between remote models can be considered. This is not possible in the MC approach.

Object duplication : In a distributed computing scenario, the name scope of an object identifier is limited to the model in which the object is defined. This leads to duplication of objects and to issues involving program maintenance, extensibility, source code size and execution time, as discussed in the previous section. The negative effects are, however, compounded by the fact that models are located remotely relative to each other, and that the bridge system is now much more complex.

2.9 Quantitative analysis of the MC approach

The construction of the *Hydraulic model* by making use of the *Topology model* and *Elevation model* has been identified as a suitable benchmark operation which would be ideal for the quantitative analysis. This operation includes time-intensive user interaction as well as numerous data file operations. Wherever a data operation is perceived as taking noticeably long, the Duration of Execution (DoE) is quantified. Wherever user input requires considerable time, the User Interaction Count (UIC) is used to quantify the total time. For specific key operations, the Basic Instruction Count (BIC) and the Persistent Data Size (PDS) is used as additional measure. By using the same suite of operations for the AC approach, a comparison can be made. The different quantitative measures are defined in the Introduction (see Section 1.4). The spreadsheet summarizing all quantitative results is found in Appendix B.

Duration of Execution : The execution duration is measured in seconds and depends on the benchmark computer. The computer has an Intel Pentium II 333 MHz processor, with 160 Mb RAM running Microsoft Windows 98.

Basic Instruction Count (BIC) : The BIC has been evaluated for the port operations on both sides of the bridge, between the *Topology model* and the *Hydraulic model* as well as between the *Topology model* and the *SEW2DTM* bridge. Thus the two most common time consuming data file operations (for the unstructured text file and the structured binary file) are evaluated with

the DoE and the BIC. Only operations which are performed during each iteration of a loop are evaluated as they determine the perceived execution time. The BIC is then multiplied by the number of passes of the execution loop. Appendix D shows a spreadsheet where the BIC is calculated.

User Interaction Count (UIC) : The number of operations for a single basic operation is counted and multiplied by the number of entities, e.g. pipes in the sewer system. This represents the ideal situation, where a user does not make input mistakes.

Persistent data file size (PDS) : The size of data files plays an important role in the performance of an application in a distributed computing scenario. Although the MC approach is not designed to directly support the distributed scenario, it is possible to execute the different applications comprising the external models independently at remote locations. For example, the elevation interpolation can be separated from the main entry of data and the analysis. The data files which are used in the bridge structures must be transferred to the remote locations via e-mail or file transfer protocol (FTP). Only the size of the persistent data is evaluated.

The basic operations are briefly described and categorized in the next section. Then the results of the quantitative analysis for the test projects are reported for each category of operations.

2.9.1 Evaluation of the BIC and UIC

This section describes the basic operations which are considered in the BIC and UIC for a sewer model. It is assumed that a layout drawing file of the sewer system has been imported from the *CAD model* to the *Topology model*. This layout drawing file contains lines representing the sewer pipes as well as a circle indicating the outfall manhole. The entities in the drawing are already located geographically. No additional information such as manhole names or pipe attributes such as diameter are given. On a separate layer, the layout of the area is available. This typically consists of the street layout as well as parcel (plot) boundaries. A topography model may already exist in the CAD environment. If an *Elevation model* is available, the elevations can be interpolated.

- **Topology flow direction (TFD) :** In the first step the orientation of the

pipes in the sewer system is verified. As only single tree structured sewer networks are evaluated, the following procedure is followed: A function *Show Direction Definition* (referenced here as TFD1) draws arrows in the direction from start node to end node on all lines representing pipes. Corrections are then made interactively, using the *Swap Direction Definition* command. The procedure for correcting flow direction is referenced here as TFD2. For the purpose of the quantitative analysis, it is defined that 20% of the pipes in the sewer network require correction.

- **Topology model export (TME) :** During this step, unique names are automatically generated for manholes. Manholes are defined at the endpoints of lines representing sewer pipes. This is an automatic process. This is the first export of the *Topology model* to file. During this export, default values are assigned for pipe diameter, number of parcels per pipe, manhole elevation and invert levels. The quantitative evaluation is performed at a later stage when the fully populated model is exported from CAD (see TME2).

- **Topology model import (TMI) :** During this stage the topology model is re-imported. The model now contains standard sized text for all attributes as well as default values. Any errors in the topology become visible, most notably by lines running across the system to the outfall manhole. These errors in topology now require correction. The quantitative evaluation is postponed until the fully populated system is imported (see TMI2).

- **Topology model correction (TMC) :** In this interactive phase errors in the topology are corrected which were missed during the first iteration. Only the most common error is evaluated here, namely the drawing connection error. This occurs during the construction of the *CAD model* if connecting pipes do not meet at one point. A typical value of 10% of all pipes in the sewer system is regarded as modelled incorrectly.

- **Topology model coordinate export (TCE) :** During this process, the coordinates of the manholes of the sewer system are exported to a binary structured file. All operations are quantified using the DoE and the BIC.

- **Elevation model coordinate import (ECI) :** During this process, the coordinates of the manholes of the sewer system are imported to the *SEW2DTM* bridge. In the bridge the elevations are updated using the interpolation algorithm.

- **Elevation model elevation export (EEI) :** In this process, the updated elevations are exported back to the structured binary file.
- **Topology model elevation import (TEI) :** The updated elevations are imported from the structured binary file, and the elevations are generated in *Topology model* as part of the CAD environment.
- **Topology slope update (TSU) :** The initial slope of the pipes is updated from the elevations at the endpoint manholes. This is an automatic process.
- **Topology parcel update (TPU) :** This is an interactive operation whereby the number of parcels (plots) associated with each pipe is counted. Using the elevation update dialog box, the text located at the centre of each pipe can be quickly updated. A typical scenario where 20% of the parcel count is different from the default (1) is assumed. The interaction index for this operation is determined. The model should is now ready for export to the *Hydraulic model.*
- **Second topology model export (TME2) :** The topology model is exported to the *.SDF* data file. In this stage the DoE and the BIC is determined.
- **Hydraulic model import (HMI) :** The duration for importing the topology model is determined using DoE and BIC. If errors are found, another iteration of TMC, and TME2 is required. It is however assumed that the system no longer contains topology errors at this stage. The optimum diameters are calculated given the slopes now available for each pipe. The system is then analysed.
- **Hydraulic model export (HME) :** The analysed hydraulic model is exported to the *.SDF* file. This export is quantified.
- **Second topology model import (TMI2) :** The updated *.SDF* data file containing the optimum diameters is re-imported to the *Topology model.* Both the DoE and the BIC are determined.

2.9.2 Result of the quantitative analysis

Three key operation categories are defined to group the basic operations. They are again used for the comparison with the AC approach, and are defined as follows:

- **Total Update Elevation (UE) :** This is the sum of the TCE, ECI, EEI and TEI operations and represents the total duration for the update of elevations.
- **Total Build Model (BM) :** This is the sum of the TME2 and HMI operations and represents the total duration for the construction and export of a populated model from the *Topology model* and the import into the *Hydraulic model.*
- **Total Draw Model (DM) :** This is the sum of the HME and TMI2 operations and represents the total duration for the export of a populated model from the *Hydraulic model* and the import into the *Topology model* and the display thereof.

Duration of Execution and Basic Instruction Count : The performance for the key operations (UE, BM and DM) is shown in Table 2.6 and shown in graphical form in Figure 2.13.

Table 2.6: Comparison of key MC operations for DoE and BIC

Project	**TP1**	**TP2**	**TP3**	**TP4**
Number of pipes	**104**	**682**	**2934**	**6721**
Update Elevations (UE)				
(1)	2.0	47.0	350.4	2608.4
(2)	2.0	27.5	353.0	1715.0
(3)	2.0	27.5	353.0	1715.0
Build Hydraulic Model (BM)				
(1)	9.8	191.8	1717.9	10290.8
(2)	16.3	191.3	2233.0	10552.0
(3)	16.3	191.3	2233.0	10552.0
Draw Hydraulic Model (DM)				
(1)	9.0	63.3	222.8	951.0
(2)	8.9	62.2	327.0	978.2
(3)	8.9	62.2	327.0	978.2

Note: (1) Duration of Execution (DoE) in s
(2) Basic Instruction Count (BIC) in million operations
(3) Equivalent duration for Basic Instruction Count (BIC) in s

The following quantitative results are observed in the table: One second of equivalent BIC is approximately equal to 1.0E6 BIC operations.

DoE and BIC comparison : The BIC of the *Update Elevations (UE)* operation does not accurately represent the DoE for large sewer systems where a value of 2608 s is measured for the DoE but only a value of 1715 s (or 1715xE6 instructions) is expected. This is most likely a scenario where external factors such as insufficient system memory (which results in hard disk spooling to secondary storage) can influence the performance. However, it is the purpose of the complexity analysis used for the BIC to eliminate the platform dependence of DoE measurements.

It can be seen that the BIC is an absolute measure of execution performance since it correlates well with measured DoE values, especially for the *Build Hydraulic Model* (BM) evaluation. It will again be used in Chapter 5 for the analysis of the AC design and in Chapter 6 for the comparison between the MC approach and the AC approach.

The *Build Hydraulic Model (BM)* collective operation is a very time intensive operation. It takes up to 2.75 hours (over 10000 s) to build the hydraulic model for the largest network. Generally this operation is only required once to convert a complex *CAD model* to a *Topology model*. Factors which contribute to the poor performance are the interpreted programming langauge, the slow interprocess communication (COM) between the embedded Visual Basic environment and the CAD engine, as well as the `getClosestTextGraphic()` algorithm which must evaluate all text entities within the vicinity of each point of interest to find the closest text string. Although an improved algorithm could be implemented for the AC design, the same algorithm will be used in order not to influence the general comparison between the MC approach and the AC approach.

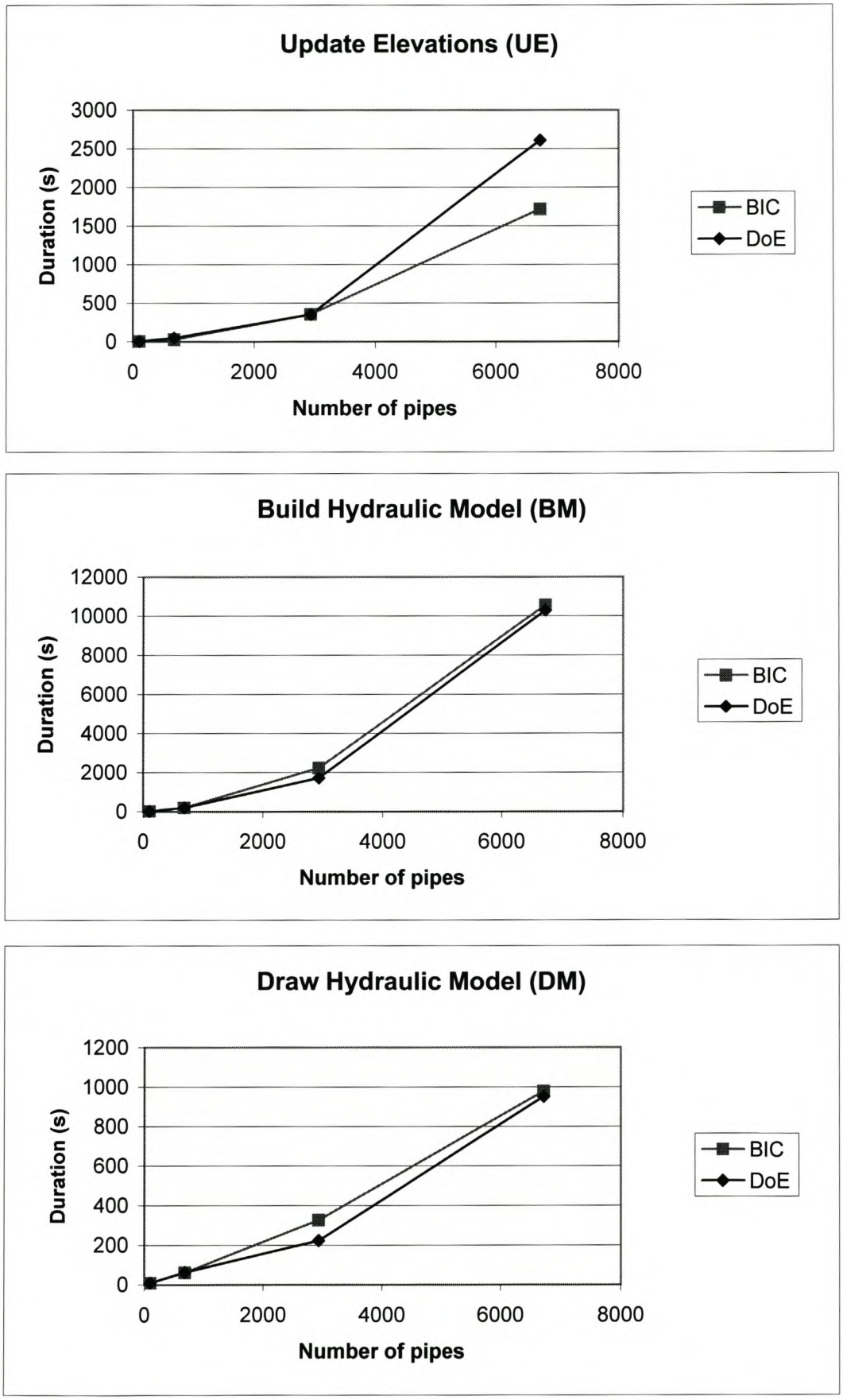


Figure 2.13: Graph of key MC operations for DoE and BIC

User Interaction Count : The performance for the operations (TFD2, TMC and TPU) with respect to the UIC is summarized in Table 2.7. Figure 2.14 plots the UIC against the number of pipes. It shows the interaction count for the three measured cases cumulatively, namely from bottom to top for *Swap Direction Definition* (TFD2), *Topology Manhole Correction* (TMC) and *Topology Parcel Update* (TPU). This graph is of use only when compared later to the AC design scenario.

Table 2.7: Key MC operations for UIC

Project **Number of pipes**	**TP1** **104**	**TP2** **682**	**TP3** **2934**	**TP4** **6721**
User Interaction Count (UIC)				
Swap Direction Definition (TFD2)	104	682	2934	6721
Topology Model Correction (TMC)	125	818	3521	8065
Topology Parcel Update (TPU)	63	409	1760	4031
Total	292	1909	8215	18817

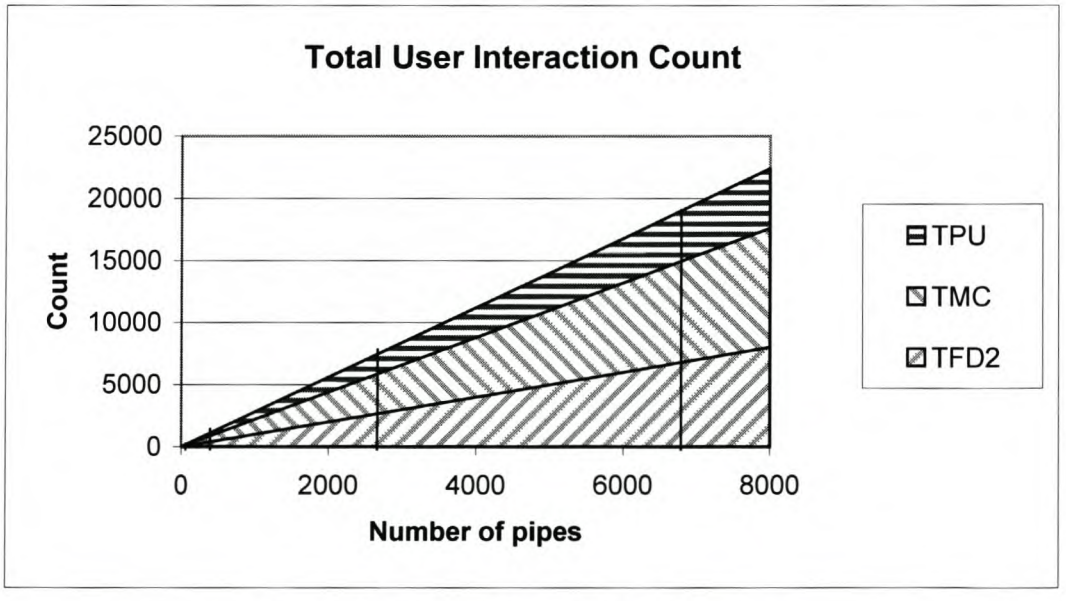


Figure 2.14: Graph of UIC performance

Size of persistent data : The size of all files which contribute to the persistent data state is totalled and presented in Table 2.8 for the different test projects. The first column describes the data file followed by the size in Mbyte for each test project. It can be seen that a total file size of over 10 Mbyte is required to store the persistent data for the largest test project.

Table 2.8: PDS requirement in Mbyte

Project	**TP1**	**TP2**	**TP3**	**TP4**
Number of pipes	**104**	**682**	**2934**	**6721**
File description				
DXF CAD file	0.091	0.722	2.295	5.054
SDF Model file	0.023	0.123	0.524	0.751
ELV DTM file	0.004	0.017	0.069	0.158
SHP GIS PIPE file	0.010	0.059	0.253	0.578
SHP GIS NODE file	0.003	0.019	0.081	0.187
DBF GIS PIPE file	0.051	0.326	1.397	3.198
DBF GIS NODE file	0.011	0.065	0.279	0.637
Total	0.193	1.331	4.898	10.563

A graph for the PDS is presented in Figure 2.15. It shows a near-linear relationship between persistent file size and number of pipes. This relation can be explained as the data files generally contain one row or record with constant length per pipe.

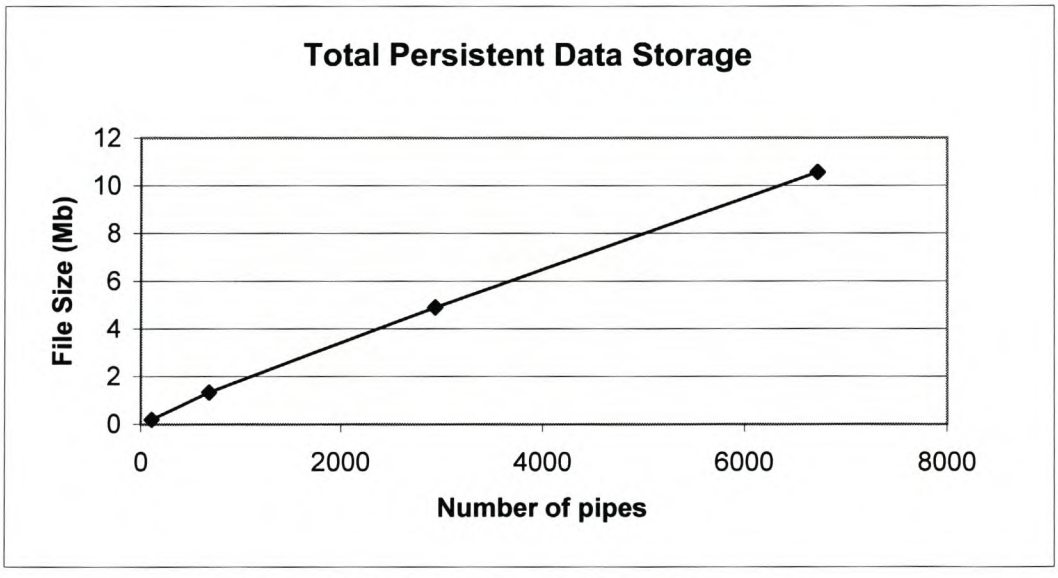


Figure 2.15: Graph of PDS performance

2.10 Conclusion

This chapter presented the characteristics of the MC approach, followed by an in depth evaluation of a typical MC system taken from the hydraulic engineering field. Great effort is taken to detail the typical engineering process as well as the design of the MC system with special emphasis on the transformer bridge. Many of the concepts introduced, such as management of the topology and geometry, are typical for other applications from the field, such as for Water Distribution Analysis Software [43] or Storm Water Analysis Software.

The key result from the analysis of the design structure is that the limitation of the name scope of object identifiers in the MC approach severely limits the flexibility of the design to allow for program maintenance and extension. The complex design and management of software bridges inhibits the natural development of the software design.

The quantitative analysis is primarily of use later for the comparison of results with the AC approach. However, the BIC criterion has been calibrated against the measured DoE times and allows the limited extrapolation of results for larger test projects for a platform independent state.

The analysis of the MC design approach has now been concluded. The next chapter will deal with the concept of the AC design approach, followed by its implementation.

Chapter 3

Concept of the AC design approach

3.1 Introduction

The analysis of the MC design approach in the preceding chapter shows that the data flow in the software package is hindered by the basic structure of the system. Especially the number and complexity of the ports between models, the effort for software maintenance and the detailed model documentation, which is required when adding new ports, have initiated the following study of an alternative design.

In order to overcome as many of the shortcomings as possible, the AC design approach was developed. The concepts of this design are presented in this chapter. The implementation of the design will then be presented in Chapter 4.

Definition : In the AC design the concept of separate models for different applications interconnected via ports is replaced with a concept where the application as a whole contains all the objects. Models are now formed by the collection of objects according to semantic views within the application. Furthermore, a model in the AC approach can contain sets of objects as well as data structures to define the relationship between objects. Figure 3.1 shows a diagram representing models, sets, relation-sets and objects in an AC approach and indicates that objects, sets and relations can be shared between models.

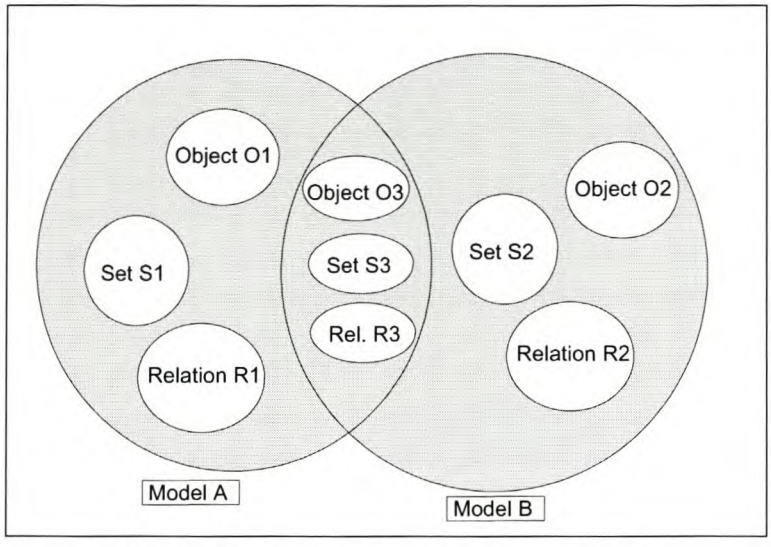


Figure 3.1: Definition of AC approach

Typical models in a civil engineering application are treated in this chapter. These models include a data model and a visualization model as well as other engineering models.

Outline : Several characteristics, which define the AC design, are discussed in depth in this chapter. The concept of application-wide unique object identifiers is introduced, which forms the backbone for the software design. This is followed by a discussion of the concept of object set management, which controls the aggregation of objects into sets. The formation of relationships between objects and the management of state consistency are then presented. The structure of the model-object is described. Finally the adaptation of an AC approach for a distributed computing scenario is investigated.

Definition of classes : Contrary to work done by Laabs [23] for an object-centred design, the class structure, which defines the blueprints for the creation of objects, is not considered part of the model or application structure in the AC design. The classes are stored as files and are components of the software platform. It will however be shown in the implementation (Chapter 4) that classes for basic engineering objects (such as pipes and manholes), which exist in typical engineering models, can be designed effectively in a hierarchal manner.

3.2 Concept of an object identifier management

Every instantiated object in an application occupies a unique space in the computer memory. The state address of this space is referred to as the data pointer to the object (in C++) in the memory space of the computer, or as a reference to a memory location in a virtual machine (in Java). However, the memory address or reference is not persistent and can be different each time the application is executed and the object is instantiated.

Furthermore, an object must be instantiated before it has a unique memory reference. This presents a problem where relationships are to be implemented between objects which have not yet been created. For example let an object reside in an object database but not in application memory. By using identifiers (such as strings) which are defined uniquely within the application for all objects of the application, it is possible to identify and reference an object persistently over its lifetime, even if the object is not instantiated.

Type of identifier : The early computer programs used consecutive natural numbers (0, 1, 2, .., n) as identifiers for objects due to memory space limitations. As larger integer data types and memory space became available, the requirement of consecutive numbers was lifted, which allowed the user to use any natural number within a given range. Then identifiers with a fixed character length were introduced. Today character sequences (strings) of dynamic length are used as identifiers.

The consecutive numbering of the objects may not be suitable for a given problem. In particular, the modification of lists of objects by the insertion of new objects is difficult. The identification with character sequences of dynamic length provides the user with the largest flexibility in the choice of mnemonics. The mapping of the persistent identifiers to object references requires special attention if this method is used.

Assignment of identifiers : The assignment of unique identifiers by the user is frequently useful in an engineering environment, for instance if the software objects represent real-world engineering objects to which the engineer can relate, such as manholes in a sewer network. The user can add meaning to the identifier which relates it to other parts of the engineering problem. However, the assignment of identifiers by the user should be optional throughout the application. It must be possible to create internal

identifiers for objects whose identifiers do not interest the user. This then necessitates a method for the automatic generation of object identifiers.

Scope of identifiers : Usually an engineering problem contains objects from several classes. Choosing the class as the identifier space for its objects can result in objects from different classes with equal names. Since an object can be contained in more than one set, sets are also not suitable as unique object identifier spaces. Therefore the object identifier must be unique at least within the scope of the application. This is the lowest level of uniqueness imposed by the AC approach.

The concept of the application can be limited to a single processor computer with one memory address space or it can be extended to multiple processors in a distributed network scenario. Even a passive object, that is an object which is not instantiated in the memory space of a processor but which resides in storage, can be part of the object name scope of the application.

Compelling reasons may frequently dictate a higher level of uniqueness. If a persistent object may be accessed at a later stage outside the context of the current application, then a uniqueness based on a project or geographical level can be introduced. For example the merging of the data models of two sewer systems, each originally created in its own object identifier space, can result in objects with colliding identifiers. This may be avoided by assigning object identifiers which are unique over all sewer system projects. For example all the identifiers can be prefixed with the relevant unique project name.

Ensuring uniqueness : When providing the flexibility of user-defined identifiers for objects, care must be taken to ensure that every new identifier is unique within the name scope of the application. This does not present a problem as long as the application has full control over the issuing of identifiers. This control can be achieved by querying an indexed list of all identifiers issued in local memory, or by requesting a unique identifier from a remote name server or database.

As an alternative, universally unique identifiers can be used. These identifiers can be generated by special algorithms which typically use the system time and other unique properties of the computer environment, such as processor id, network adapter address and IP address to generate a unique identifier. Java [44], for example, provides a function to return a unique identifier (UID)

which uses a random number, the system time in milliseconds, an integer counter and the network IP address for the generation. An example of the Java generated identifier is

62eec8:e98869e9b3:-7cf1:192.168.0.1.

In CORBA [31] two formats for specifying global repository IDs are defined. The one combines strings to construct a multi-scoped name consisting of a unique prefix, a list of identifers (separated by "/" characters) in a tree structure and a version number. The uniqueness for this string is not guaranteed. An example of this CORBA identifier is

wadiso.com/Water/Models/CapeTown/2005/:2.0.

The second format is known as a DCE (Distributed Computing Environment) Universal Unique Identifier (UUID). This identifier is generated using the current date and time, a network card ID and a high-frequency counter. Microsoft DCOM (Distributed Component Object Model) uses a similar concept to generate GUID (Global unique identifiers) for objects. An example of such a CORBA identifier is

700dc500-0111-22ce-aa9f.

Automatic generation of identifiers : If the user does not provide an identifier, a unique identifier must be generated automatically. This may be the case where no meaningful relation to a real-world engineering object exists, or where the generation of the identifier is automatically enforced when a user-defined identifier collides with an existing identifier. For example, the identifiers of pipes, which are typically not provided by the user, may be generated automatically.

Design of identified objects : Two alternatives will be considered for the introduction of objects with persistent identifiers, one using inheritance and the other using interfaces.

Using inheritance in a single inheritance programming language permits the introduction of the identifier logic at the top of the class hierarchy, so that all derived classes are provided with the functionality to support identifiers in instantiated objects. The logic that must be supported is the mapping of persistent object identifiers to valid references within the address space of the virtual machine or processor. In Figure 3.2, if object x references an object a using the persistent identifier s, then object x cannot address object a in

the system memory until the persistent identifier has been mapped onto the temporary reference to object *a*, denoted as <*Object a*>.

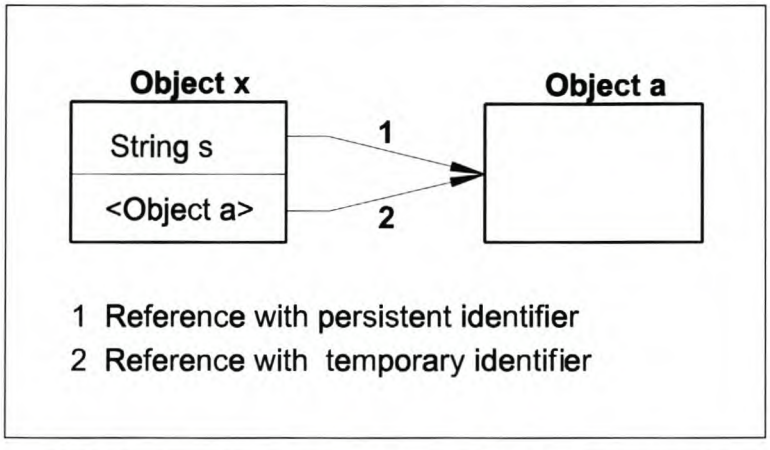


Figure 3.2: Mapping of persistent identifier to temporary reference

This mapping in the application id-manager causes an overhead in the execution of the application. It can be expected that for time sensitive operations, the object references must be cached for performance reasons, as objects can only be accessed directly with an object reference. Alternatively, the program logic to determine the object reference from the identifier should be optimized for speed using for example efficient hashing algorithms. Hashing is a method for the management of object pairs, which will be discussed in the implementation (Chapter 4).

One disadvantage of the inheritance design, especially in single inheritance programming languages, is that the developer is forced to derive custom classes from the top-level identifier management classes and therefore cannot derive new classes from existing classes in a graphical class library.

An alternative is to make use of interfaces to assign unique identifiers for the basic engineering objects. The interface defines a contract with methods implemented typically in classes common to all engineering objects, to assign, generate and maintain identifiers. This results in an associative relationship between the basic engineering objects and the common objects. This has the disadvantage that for every object in the application such a relation must be maintained throughout the lifetime of the application.

Figure 3.3 shows the two design options as UML class diagrams.

Interaction between objects : The AC design approach does not require ports between models. Since all objects contained in the application are

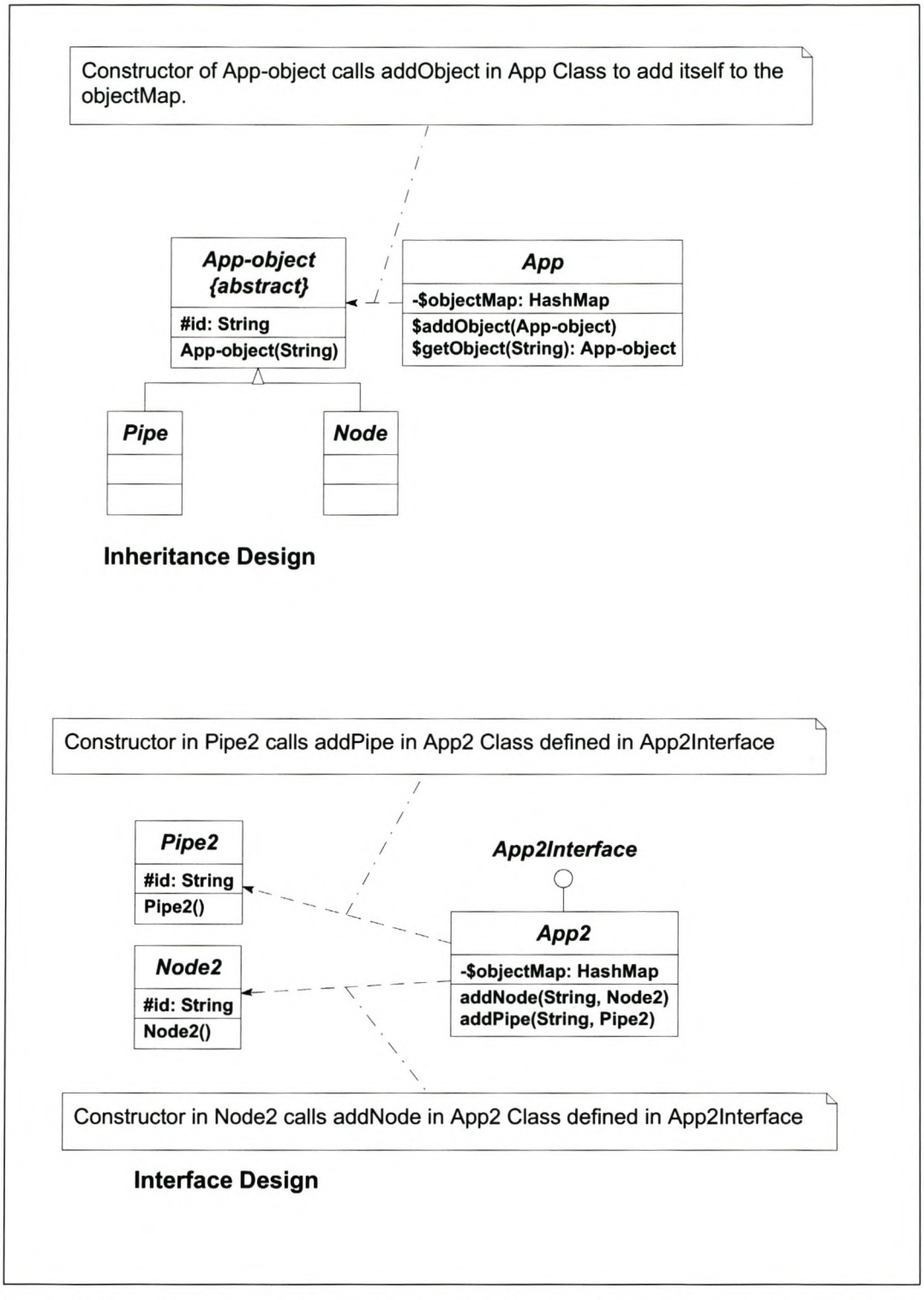


Figure 3.3: Inheritance vs interface design for implementing application identifiers

persistently identified, and since a central identifier manager exists to map persistent object identifiers to object references, any object can now directly access any other object in its model, as well as in other models in the application, for which an identifier has been stored.

By passing messages between objects using structured method calls, the state of the called object can be changed by updating its attributes. The synchronization of state and version management between objects is important to ensure that the creation, deletion and update of objects located in different models are handled in a consistent manner over the application. This topic will not be investigated in depth in this dissertation.

Updating of objects : In order to evaluate the use of identifiers in the software system, the following three basic operations are considered and evaluated: The addition of an identifier (as part of the object constructor operation and in object factory methods), the deletion of an identifier from the list of issued identifiers (required when an object is to be deleted from the application), and the change of an identifier (should an object identifier be renamed). The option to rename objects is important as soon as user-defined identifiers are allowed, as the user may make errors during input. Furthermore, if objects from models with distinct identifier spaces are combined in a common identifier space, their identifiers must be checked for uniqueness and modified if necessary.

3.3 Concept of an object set management

Objects of the real-world, which are separate identifiable elements, are collected in sets. An example is the set of all pipe objects in a sewer model. A set by itself is unstructured. The set is structured by specifying relations on the set.

A set-object is the simplest form of a set implemented in the AC design. Here a collection of similar objects is treated as a set, and operations can be performed on all elements of the set. A relation-object contains a set of objects which describes a relation: each element of the set is an n-tupel. The largest identifiable unit in the AC approach is a model-object, which can consist of objects, set-objects and relation-objects.

In the AC design all objects representing the engineering task are collected

and managed in one application. The persistent identifiers of objects and not their temporary memory references are collected in the sets.

In contrast to a MC design approach, the same object can be part of more than one model in the application. This is achieved by storing the same persistent identifier in one or more set-objects which may be contained in more than one model-object. An object representing a pipe, for example, can be part of the *Hydraulic model* and of the *Visualization model*. In the *Hydraulic model*, only the methods and attributes required for the hydraulic analysis are stored. In the *Visualization model*, the methods and attributes for a graphical presentation are implemented.

Mathematical background : A set is a collection of separate objects which have some common property which is used to identify the objects. These objects are called the elements of the set. Elements are considered to be identical if they have equal identifiers. If an element e is contained in a set S, this is expressed as $e \in S$. An empty set which does not contain elements is denoted by $\emptyset$. Sets are equal if they have exactly the same elements. A subset A of S is denoted by $A \subset S$. Sets can also be elements of other sets. This leads to a system of sets. Furthermore, algebra can be performed on sets to create new sets (see Pahl, Damrath [34]).

Operations on sets and models : Operations which are common to all objects of a set are executed by issuing a message to the set-object, which in turn passes the message to all elements of the set. This concept can be extended to allow the issuing of a message to a model (a collection of sets), which in turn forwards the message to all sets of the model.

An example is the aggregation of the identifiers of all erven (parcel) objects related to a pipe in a set. A command such as *calcArea* can be executed on the set-object which is passed to the individual objects referenced by the set. Another example is the command *drawModel* for the Geometry model-object, which will be passed to the sets containing the model-object, and then to the objects referenced in the sets.

Design of object set management : The objects in a set can be instantiated from the same class, but this is not a requirement. Sets can be formed of objects from different classes. Standard methods are available to add objects to the set, to delete objects from the set and to maintain a count of the number

of objects contained in the set.

Set membership : The membership of an object in a set can be defined in two different ways: (a) An object can define its relation to a set by an identifier attribute in its class structure, or (b) the object references can be listed in a dedicated set object.

However, using (a) as the only mechanism to define set containment does not provide a practical solution. The collection of object references using (b) in set-objects provides the flexibility that operations can be performed on all elements of the set by issuing one operation. To determine whether an object is a member of a set requires a sequential search over the set and demands considerable processing time in large sets.

The combination of the two techniques can be considered where it is required to have fast access to the set to which an object belongs, but still have the flexibility of set operations.

Use of identifiers : Either unique object identifiers or object references can be used as elements in the set-objects. However, in order to create a persistent software system it is recommended that only unique object identifiers be used. This allows set-objects to be stored persistently, for example all the pipe identifiers can be stored in a set.

The validity of objects in the set is not influenced by the lifetime of the objects. Also, the debugging of program code containing sets during development is simplified, as most integrated development environments or error-trapping code can present the list of user-readable identifiers in the set to the programmer.

The set-object like all other objects should be uniquely identifiable within the application. This allows for set-objects to be referenced in the same way as application-objects in a relation.

Updating of sets : Sets can form part of a relation and should therefore be treated in the same way as a single object in a binary relation. The addition, removal and change of elements in sets can have far-reaching effects on the contained objects. A dedicated management structure must therefore be provided for every set-object. Methods in the set-class must exist to notify all contained objects of changes or pending deletion. The implementation will be

shown in Chapter 4.

Dynamic sets : A set can be created and maintained dynamically. The elements in the set are only added when required. In Java this can be done by implementing the *Iterator* interface. In the pilot implementation, all network tree traversals are done using these dynamic sets.

3.4 Concept of an object relation management

Objects are not independent in a software application. Objects of the real-world usually stand in some relationship with other objects. For example a sewer pipe must always have an upstream and a downstream manhole. This dependency of the pipe on the two manholes can be expressed in an engineering model, for example a sewer network topology model, by the relationship between the sewer pipe object and the two manhole objects.

Mathematical background : A relation is a subset of the cartesian product of two sets A and B. Elements $a \in A$ and $b \in B$ are related by a connective rule R. The value aRb of the connective rule for a pair (a, b) of the cartesian product $A \times B$ is either true or false. If the value of aRb is true, then the pair (a, b) is contained in the relation. If the value of aRb is false, then the pair (a, b) is not contained in the relation. The order of the elements a and b can be of importance for the connective rule. The relation R is therefore a set of ordered pairs (a,b) for which the connective rule yields the value true. The relation is formally defined as follows:

$$R := \{(a, b) \in A \times B \mid aRb\} \tag{3.1}$$

In software applications a relation is often not specified in the form of a connective rule and a cartesian product set. Instead the ordered pairs of the relation are given explicitly. If the elements of the relation are object references then the term *relation-set* is introduced to describe the set of ordered pairs. An object pair in the relation-set which defines the relationship between two objects is subsequently referred to as a *relation-object*.

Types of relations : Different types of relations can be formed. The type can be expressed in terms of multiplicity or uniqueness. For example a 1:1 object

relation expresses a unique relation, whereas a 1:n object relation represents a left-unique and a m:1 object relation a right-unique relation. This concept is shown in Figure 3.4. Also, optionality or completeness can be attributes of the relation. Relations can be left-total, right-total and bi-total relations. This is shown in Figure 3.5.

A left-total, right-unique relation is called a mapping. A bijective mapping is a special case of a mapping where the relation is right-total and left-unique.

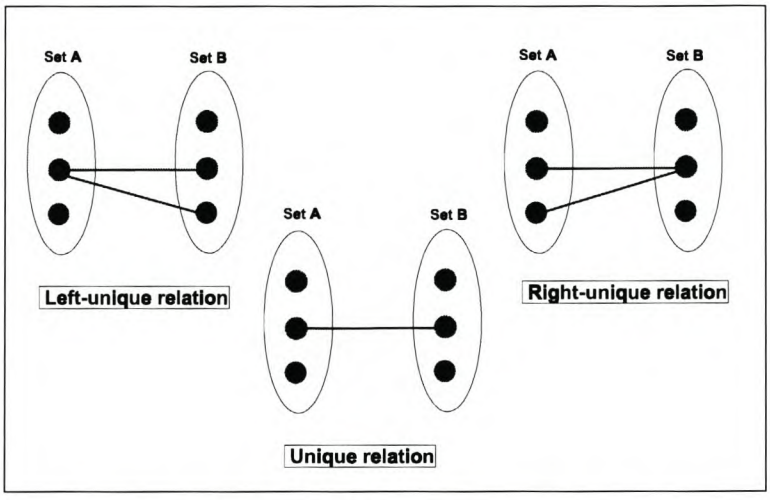


Figure 3.4: Definition of uniqueness in a relation

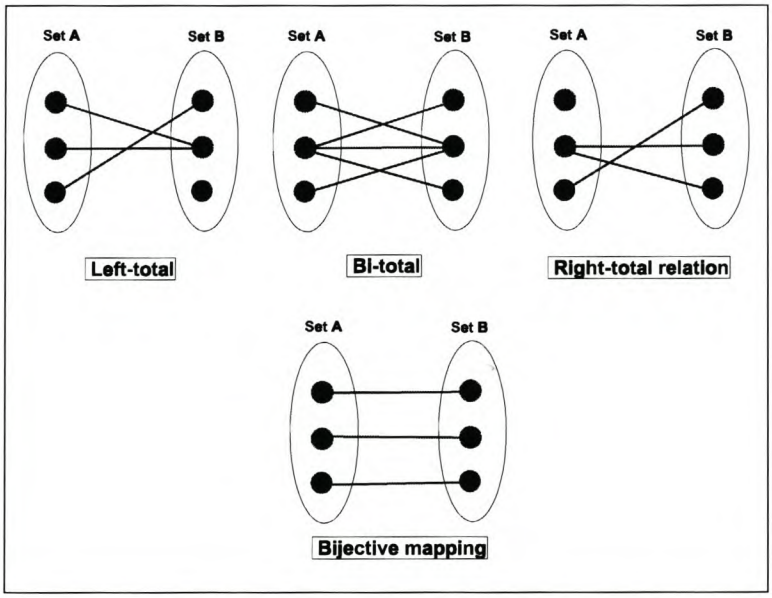


Figure 3.5: Definition of completeness in a relation

Implementation of relations : Most OO programming platforms do not provide separate data structures to define relationships between objects. One way to implement relations is to incorporate the reference to the destination object as an attribute within the class structure of the source object. This results in an a priori design, as the relationship between instantiated objects is defined in the class structure and cannot be changed at run-time.

Alternative design : An alternative design is presented in this dissertation. Relation-objects are collected in one or more relation-sets. A relation-object is instantiated from the relation-class. This class defines the structure and operations so that an ordered pair of object references and additional data on the multiplicity of the relation can be maintained.

The relation-objects as well as relation-sets are uniquely identified by application-wide persistent identifiers. An object pair is uniquely identified within the application if the identifier of the containing relation-set and the identifiers of the two objects, which stand in relation, are combined. Usually, however only the left-paired object (frequently called the key) is known a priori, so that the identifier cannot be formed in this manner and passed to the application id-manager to obtain the reference. A different semantic rule for the formation of the identifier is therefore used.

A solution to this problem, which is presented in Chapter 4, makes use of the class name of the right-paired object, which is known a priori. However, the prerequisite for this design is that only mappings are allowed between the left-paired and the right-paired objects, which implies a right-uniqueness as well as left-totality in the relation. This ensures that only one right-paired object can exist. Multiple right-paired objects can exist only in the relation if they are first collected in a set-object, and then the set-object is referenced as the only right-paired object.

Alternatively, if no semantic is assigned to the identifier of the relation-object, then the attributes of the relation-objects must be searched for the left-paired object identifier and for the matching right-paired identifier. Spacial data structures can be used to control the search effort, for instance trees.

Operations on relations : Several basic set operations can be performed on the relation-set. This includes the addition and removal of relation-objects. The addition of a relation-object to a relation-set is typically performed during the construction of the relation-object. The relation-set can also be searched

for a specific relation-object. Furthermore operations to navigate between the related objects are provided.

Dynamic nature of relations : The definition of relationships between objects in relation-objects, which are aggregated in relation-sets, is advantageous if the relations vary with time. Creating, modifying and deleting relationships between objects is greatly simplified because the management is centralized outside of the objects involved.

The difference in structure between the two designs as implemented for relations is shown in the Chapter 4. It will be seen that several complexities in the application, such as ensuring a consistent system state between objects which stand in relation, can be incorporated generically into the external relation-set and set-object design.

Issues such as the prevention of cycles in state notification are also addressed. Updating of state can either occur automatically when the program logic requires action from the user, or manually when the user triggers an update chain. Both scenarios are considered in the implementation.

3.5 The model-object

A model is implemented as a complex object in the AC design. The model-object contains objects, set-objects and relation-sets as attributes.

Figure 3.6 shows a diagram combining model-objects (M_i), set-objects (S_i), relation-objects (R_{ij}), relation-sets (R_i) and objects (O_i) in an application. The diagram shows that different objects can be contained in more than one model.

Operations which are applicable only to a model are defined in the model-class. These operations are executed by operating on the sets and relations in the model-object.

Some of the typical models in a civil engineering application, other than the engineering models, are:

Data model : This model contains all the sets and objects defining the engineering data. It is the purpose of this model to ensure persistent storage to and retrieval from permanent media for all the persistent objects. Data

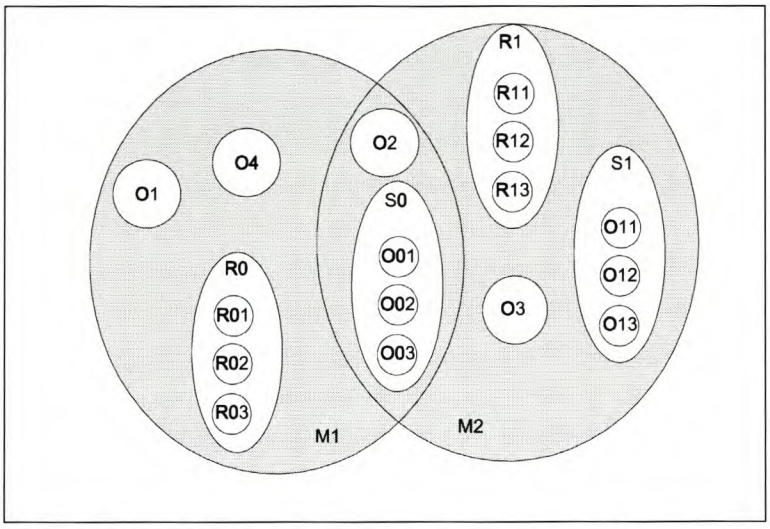


Figure 3.6: Definition of model-objects in an application

models are frequently called product models.

Visualization model : The visualization model typically makes use of extensive graphical resources and is frequently based on an external graphics library. A typical scenario is therefore investigated in this dissertation, where a visualization model is dependent on external source code. As the programmer cannot change the third-party code, a problem might arise where external non-engineering objects contained in the visualization model must interact with objects which are application-objects, thus derived from the application-object or vice versa.

This problem can be overcome be using well defined interfaces for both application and non-application parts. This will then allow object communication in both ways. It is, however, important to note that the engineering objects are to be directly referenced in the data model via the interface, and duplication of engineering related objects does not take place in the non-application parts.

An example of this interaction using interfaces and a modified relation manager is shown in Chapter 4.

3.6 Extending the design to a distributed environment

Modern software systems must be functional in a distributed computing environment. As discussed in the introduction of the dissertation, several reasons prevail for the introduction of distributed computing to a software project. This includes the centralized storage of project object data, the flexibility to share project information with co-workers and the definition of different levels of user access to the same project.

In this dissertation the implementation of the centralized data model and engineering model on the server side is considered. On the client side, different visualization user-interfaces are considered. The effort required to implement an application for the distributed computing environment for the key components of the AC approach will be considered.

Persistent identifiers : In principle persistent identifiers are also applicable in a distributed scenario, as long as the software platform provides the infrastructure to extend the name scope over the network. The mapping of identifiers to references in the virtual memory space of the distributed application is simplified if a distributed software platform (such as Java Remote Method Invocation, RMI or CORBA) is used.

Sets and relations : As long as references in sets remain valid over the name scope of the distributed application, sets also function in this scenario. The use of unique identifiers instead of object references is recommended for this scenario as well, as references are particulary difficult to maintain in a distributed scenario.

Model-objects : A practical scenario is considered where all engineering models are located on the server side. This ensures that the overhead in communication between objects located in different models is limited. The visualization model is located on the client-side. The interaction between the visualization model and the engineering models will be evaluated.

3.7 Conclusion

The concepts of the AC design approach have been outlined. Several options have been discussed for each concept, of which only a few will be selected and implemented in the next chapter. The choices can be summarized as follows:

- The application has been chosen as the space for the object identifiers.
- User-definable identifiers are proposed.
- An inheritance design has been chosen to implement the application identifiers.
- The mapping between persistent identifiers and object references is to be optimized in the application id-manager.
- The containment of an object in a set is defined by aggregating the object identifier in an external set-object.
- Set-objects provide logic to distribute addition, removal and update notifications to member objects.
- Dynamic sets are used.
- If relations between objects vary over time their definitions are stored in external relation-objects and not internally as attributes in the objects.
- Relation-set objects provide a logic to distribute addition, removal and update notification to member relation-objects and in turn to the referenced objects.
- Both manual and automatic consistency updating are evaluated.
- Models are implemented as complex objects, containing sets and objects.
- The integration of library code or other non-application objects with application-objects will be shown in the implementation of the visualization model.
- A simple extension for the distributed environment will be implemented.

Chapter 4 now deals with the implementation of these concepts in a practical software application, capable of handling typically sized civil engineering problems.

Chapter 4

Implementation of the AC design approach

4.1 Introduction

A prototype of the application following the concepts of the AC design approach described in the preceding chapter is implemented and studied in this chapter. The basic engineering objects are described first. Then the generalized implementation of object identifier management, the object set management and the object relations management are discussed. The revised engineering process is then presented as well as a description of the functionality and algorithmic background required for the AC software system. The classes for the Product-data model are detailed, followed by the implementation of the engineering models as model objects. The chapter is concluded with a section on the implementation changes required for a distributed computing environment.

4.2 Basic engineering objects

The real-world objects which are modelled as software objects are defined in this section. These real-world objects are called the basic engineering objects in the software application. In order to define the objects, the primary data and functionality associated with each object is outlined below. Figure 4.1 shows the basic engineering objects in a sewer network.

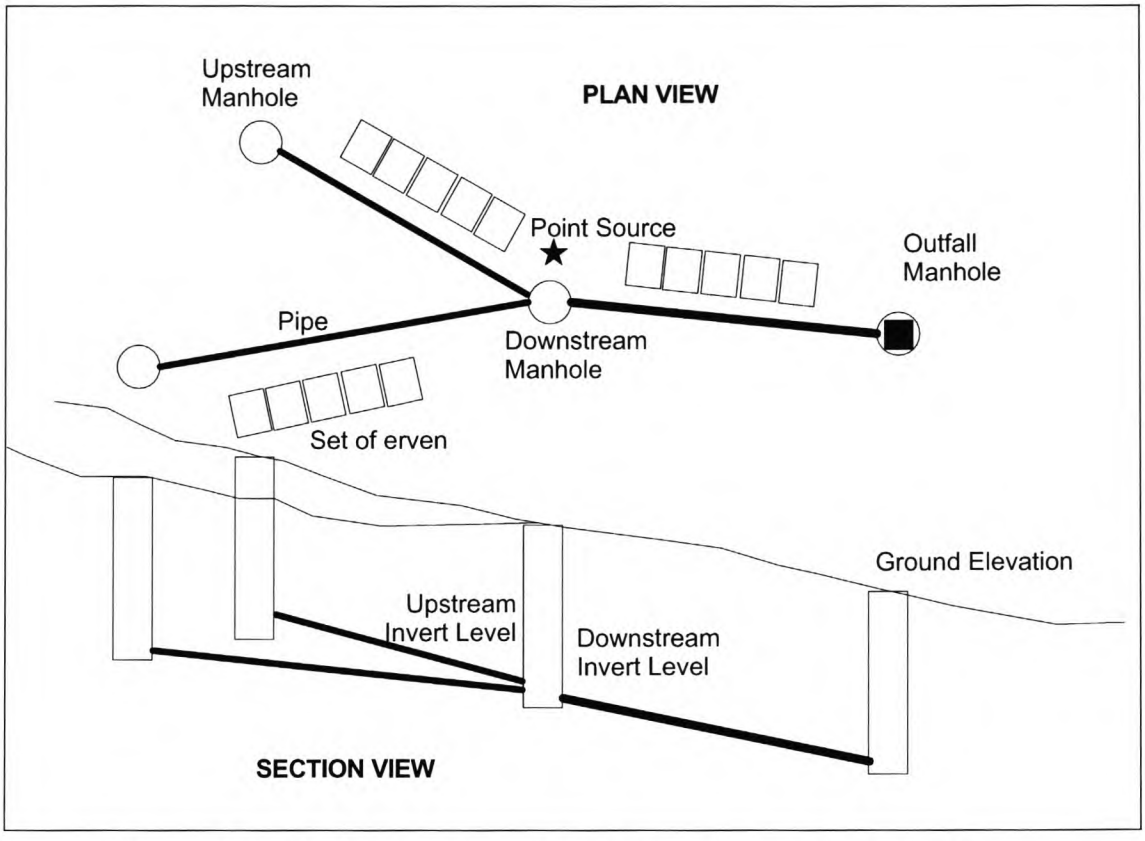


Figure 4.1: The basic engineering objects in a sewer network.

Manhole : A manhole-object represents a physical junction or node in a sewer network. The manhole-object contains property data structured in several classes for its persistent user-definable identifier, its topology ("visited" state for graph traversals), its geometry (x,y,z coordinates), its topography (elevation interpolated state) and its hydraulic properties (calculated inflow and outflow). Operations include the hydraulic calculation of outflow, given inflow, as well as the interpolation of elevations.

Pipe : A pipe-object represents a sewer conduit. Typically the identifier of a pipe is not specified by the user but generated automatically. A pipe is a directional element which is defined from a from-manhole to a to-manhole, which is defined by the topological input data. For example, pipe P703 is defined from manhole N653 to manhole N234.

The direction can be defined automatically using the direction of slope of a sewer pipe (the drop in height over its length). This is because all sewer pipes modelled are gravity flow elements. However, the direction might initially be undefined if only the slope value has been entered by the user. The

from-manhole therefore implicitly defines the upstream manhole, and the to-manhole the downstream-manhole. Then the slope property can be verified against the defined direction, and a warning can be shown for mismatches.

The topology data defines the relationship between the pipe-objects and node-objects. This topology data can either be stored in the pipe-object (upstream manhole ID and downstream manhole ID) and optionally also in the manhole objects (list of upstream pipes IDs and list of downstream pipes IDs), or it can be stored externally in relation-objects. Both scenarios are evaluated for the sewer system topology.

The pipe object also stores the hydraulic pipe characteristics (length and slope, upstream and downstream pipe invert levels, diameter and roughness) and hydraulic result variables (flow magnitude, flow velocity, and flow depth for different scenarios), design result variables and presentation result variables. The reference to the erven-objects (land parcels) associated with each pipe can be stored either within the pipe object, or externally in a relation-object. The reference is required to calculate the total sewage production contributed to a pipe by its associated erven.

Several operations are performed on the pipe object, such as calculating the outflow hydrographs given the inflow hydrographs, calculating the flow velocity and flow depth, as well as checking results against analysis criteria.

Erf : An erf is defined as a surface defining a parcel of land. Its persistent identifier can be user-defined (for example unique Surveyor General Number) or auto-generated. Typically erven are collected in an erven-set, which is then associated with a pipe. Therefore an individual erf is not directly linked to a pipe-object. Again the storage of the relations can either be in the respective objects or externally in relation objects.

Additionally, the four coordinates defining the simplified geometry of the erf can be stored. Also the population of the erf, as well as the land usage category associated with the erf are stored as attributes. This information is used to calculate sewage production values for different scenarios.

Point-source : A point-source is defined as a node with persistent identifier where a sewage production hydrograph, for example as it is the case for a large factory or a hotel, is present. A point source stores its coordinates (x,y,z) and is associated with a manhole.

Bottom manhole : This is the manhole identifier which defines the only outfall or bottom manhole in the sewer system.

Graphs and trees : The pipes in the sewer network normally form a special graph, called a tree, since every pipe can only have one downstream pipe. Although the topological structure used in the implementation does not prevent the formation of more complex graphs in the pipe network, for example a pipe with more than one downstream pipe or loops in the pipe system, the hydraulic model currently does not support diversion structures at the nodes, which would be required if more than one downstream pipe is introduced.

As the tree of pipe-objects must be traversed, special tree-traversal algorithms have been implemented to support post-order, depth-first and breadth-first tree traversal with a generic age comparator, which compares either the accumulated pipe length at any pipe from the outfall manhole, to start with the longest water course for example, or the number of pipes upstream from the outfall manhole.

The next three sections discuss the abstract classes which define the core of the AC design approach. These classes are located in the `Application` package. Figure 4.2 shows the relationship between these classes and highlights the important methods.

4.3 Implementation of object identifier management

Two different concepts for the implementation of an object identifier management have been introduced in Chapter 3, namely using inheritance or interfaces/association. It will be argued that only the inheritance design permits the use of reflection technology. Using reflection, an object can be queried during run-time to determine if it is instantiated from a specific class.

In the inheritance design, the operator `instanceof` can be used to determine if an object is instantiated from the Application Object (`AppObject`) class. Such objects will be called *application-objects*. Only application-objects are regarded as persistent and are stored to secondary storage.

A design based on association or the use of interfaces does not permit the natural classification of persistent and non-persistent objects. An additional attribute must be stored in each Application Object, which indicates whether it should be stored persistently.

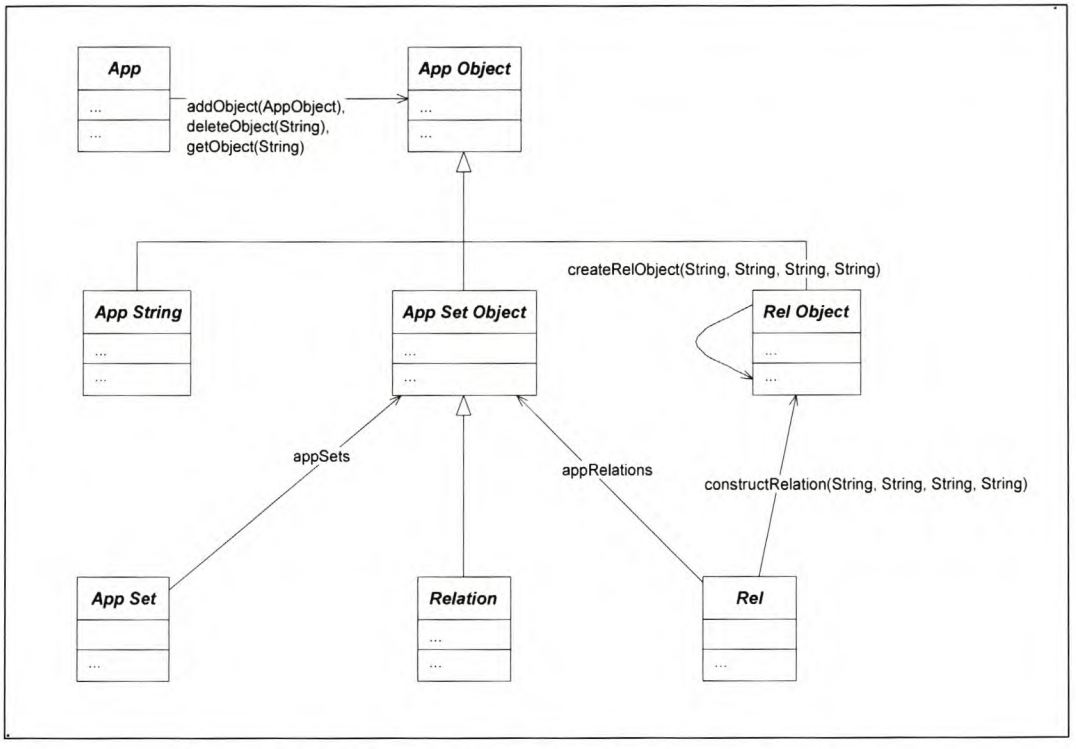


Figure 4.2: The classes of the Application package.

In the next paragraphs, the inheritance based object identifier management as implemented will be discussed. However, two concepts from the Java programming language, which are used extensively, will be explained first, namely the HashCode and the HashMap:

HashCode : The HashCode is a unique integer generated by Java for each of its objects. Whenever it is invoked on the same object during the execution of a Java application, the HashCode method returns the same integer. The HashCode is not persistent and has different values for different executions of the application. Typically the HashCode is implemented by converting the internal address of an object into an integer. Furthermore Java HashTables and HashMaps make use of the HashCode to index their objects.

HashMap as data structure : The HashMap provides an ideal storage system for application-identifier/application-object pairs in civil engineering software applications (Schutte [41]). The HashMap implements the Map interface. In this interface, a mapping consists of ordered object pairs (key object, value object). The key object is mapped to the value object. A key object can be contained in a Map only once, whereas a value object can be contained more

than once with different keys. No guarantee regarding the order of the map is given.

The HashMap implementation provides constant-time performance for basic operations (`get()` and `put()`) assuming the hash function disperses the elements properly among so-called buckets. Iteration over collection views requires time proportional to the number of buckets or capacity of the HashMap instance plus its size, the number of key-value mappings. The use of basic operations is therefore favoured over the use of sequential iterators.

The HashMap can grow automatically. Whenever its capacity reaches the product of *load factor* (which is typically set at 0.75) and the current capacity, the capacity is roughly doubled automatically by calling the `rehash()` method. If many mappings are to be stored it is advisable to create the HashMap with a sufficiently large initial capacity, as this will reduce the number of automatic rehash operations.

AppObject : This class is located at the root of all applications in the AC approach and is defined in the Application package. Its primary function is to store the unique and persistent object identifiers, as well as provide the basic functions to access the object identifier. Figure 4.3 shows a class diagram with the key properties and methods of the AppObject and the associated App class.

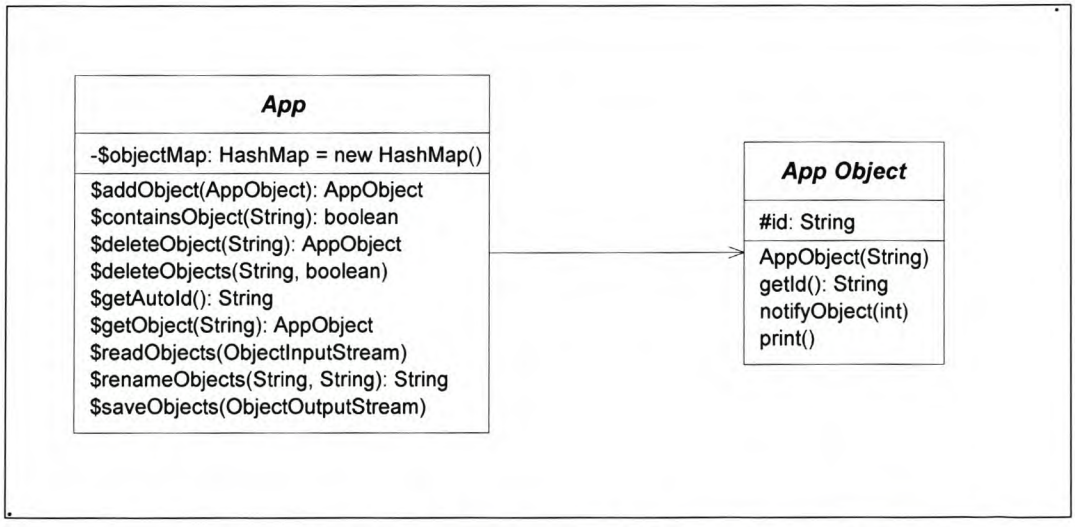


Figure 4.3: Class diagram of AppObject and App

App : This class is defined as a utility class for all application objects. Its primary function is to store the `objectMap`, a cross-reference table of all the application-object identifiers with their object reference. This table is implemented as a Java HashMap.

The class itself only contains static methods and attributes. This means that only one instance of the class is created at runtime at the system level automatically, and no further instances are created. This allows optimization in the Java runtime system.

Typical functions included in this class are `addObject()` to add a new object to the application HashMap and `containsObject()` to check if an object identifier is contained in the application HashMap. Another primary function is the `getObject()` method, which finds an object in the application HashMap, given the string identifier.

In order to automatically generate identifiers, the `getAutoId()` method creates unique identifiers with 8 characters in length and effectively maps a number system with base 62, as it cycles through the digits 0..9, then the characters A..Z and finally the characters a..z. The total number of unique identifiers that can therefore be represented by this simple system is $62^8 = 2.183E14$. An error will result should this number ever be exceeded.

Consistency : To ensure the consistency of the application object, the renaming and the deletion of application-objects must be handled specially to avoid compromising the integrity of data. Two designs are supported.

In the first design, an application-object can only safely be renamed if no other objects (relation-objects or set-objects) are referencing it. Then the old object is removed from the HashMap, given a new proposed identifier, and then again added to the HashMap. Should the identifier be in use, another identifier will be generated automatically, similarly for the deletion of application-objects. With the `deleteObject()` method, the object is only removed from the HashMap after a check has confirmed that the object is not contained in a set-object or relation-object. These checks run deeply: The search is started at the topmost application object and is performed recursively deeper into sub-sets.

In the second design, an application-object is renamed using the `renameObjects()` method. This is in effect a deep rename function. After a new unique identifier is obtained and added to the HashMap, all relation-objects in the application are searched for the old identifier and renamed to

the cached new identifier. Then all set-objects are searched deeply for reference to the old identifier, and renamed accordingly. Finally the old object-pair is removed from the HashMap. The method `deleteObjects()` functions in a similar manner. First all relation-objects are searched for the object to be deleted. For so-called `requiredRelations`, not only are the relations, but also the associated object itself, deleted. All set-objects are searched recursively, starting from the topmost set-object. Then the entries are deleted in the set-tables. Finally the application-object is also removed from the application. This approach ensures an automatic consistent state after the operation, but can result in substantial processing time.

Persistent storage : The persistent storage of the static HashMap cannot be implemented through the Java serialization of the `App` class, as a static class cannot be serialized with the standard methods. However, the custom method `saveObjects()` is used to serialize the HashMap, and the method `readObject()` to de-serialize it. What makes the Java serialization so powerful is that, by exporting the HashMap to the serialization stream, not only are the object-identifiers and the associated application-objects exported, but also the whole graph of other objects which are associated with the application-object. Java manages this graph automatically, and ensures that an object is not recursively written to the stream. This ensures that the resulting data stream (or file when transferred to file) does not grow larger than required.

As all the other objects for the AC approach, such as set-object, relation-sets and relation-objects, are derived from the application-object, they too are serialized automatically. However by using relation-objects, as will be seen later, which define the association between objects using only string identifiers, the serialization process cannot take place over the relation-objects. For application-objects this is not a problem, since application objects are serialized via the serialization of the application HashMap.

In order to accommodate non-application objects in relations, the Java HashCode can be used as identifier for non-application objects. This object can then take part in the relation, but it will not be found during Java's automatic search for the graph of objects to serialize. This means that only application objects are serialized, and for example external graphical objects not derived from the application-object will not be stored persistently to secondary storage.

4.4 Implementation of object set management

As seen in Chapter 3, sets play an important role in the AC approach. Sets are used to group application-identifiers together, which are associated with application-objects. It will be seen later that models are formed primarily by the formation of sets of object identifiers. Sets are also used to group application-objects of the same class, for example the identifiers of `erf` objects are grouped in an erven-set.

Set-objects themselves can be referenced in relations, as to-objects, for example the pipe/erven-set association. Sets-objects are also used for the definition of relation-sets. Sets can be nested, so that one topmost set can be provided, from which recursive (deep) operations can start.

A special set class known as `AppSetObject` is introduced as part of the Application package.

AppSetObject : This class defines the application set-object. It is derived from AppObject, so that it shares the object identifer management functions. It implements the Java Set interface, so that the familiar set operations known to Java programmers are fully available (Pahl [33]). This class stores an instance of the Java HashSet as variable `set`. Figure 4.4 shows a class diagram with the key properties and methods of the AppSetObject and the associated AppSet class.

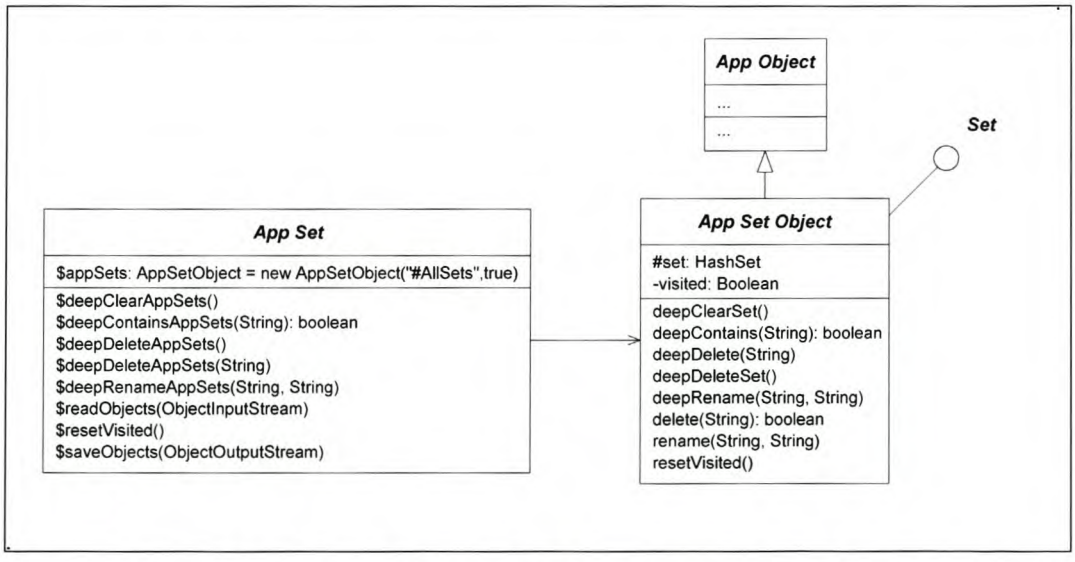


Figure 4.4: Class diagram of AppSetObject and AppSet

The constructors of the Java HashSet class are extended to support the passing of the proposed identifier, as well as add the identifier of the constructed set to the `AppSet.appSets` object. In addition to the the standard methods implemented in the HashSet class (which are replicated), several deep (recursive functions) are implemented.

Recursive functions : To ensure the consistency of the set-object, the renaming and the deletion of an application-object must be handled specially to avoid compromising the data integrity of the application. For the implementation of both designs, recursive functions are provided which operate on the current set.

Method `deepContains()` searches this set and all subsets contained in this set by means of their set identifier, until the object identifier is found. The `rename()` method renames an identifier in a set, whereas the `deepRename()` recursively searches through subsets.

Method `delete()` deletes the object-identifer from the set, and also deletes the associated object from the application, if it is unreferenced, whereas `deepDelete()` searches sets deeply. Method `deepClearSet()` does the same, except that it attempts to delete all objects in the set from the application.

Recursive functions must guard against loops in sets. This occurs when a sequence of set identifiers form a closed chain. The `AppSetObject` objects contain a private attribute `visited` which is set to *true* whenever a set has been visited. Should execution pass the set again, it is not entered again. The method `resetVisited()` in the `AppSet` class is called before every recursive operation to set the visited state of all sets in the application to *false*.

AppSet : This class is defined as a utility class for all application set-objects. Its primary function is to store and maintain the `appSets` variable. This is a static instance of the appSetObject, used as parent set for all sets in the application. Useful recursive functions are provided as static functions.

4.5 Implementation of the object relation management

Two concepts for the implementation of object relations are considered, namely the storage of the relation data within the object, or externally within

special relation-objects.

Object relations stored in objects : The object references are stored within the objects. Most programming languages directly support only this concept. For example, let a gravity sewer pipe be defined from its upstream from-manhole to its downstream to-manhole. In a more abstract view where only the topology is considered, the pipe is represented by an object `edge`, which is an instance of the class `Edge`. The manholes are represented by two object instances of the `Vertex` class.

In this concept, as implemented in Java, the `edge` object contains two attributes, namely `fromVertex` and `toVertex` which are of the type `Vertex`. However, the use of object references alone is not recommended in the AC approach. This is because object references are temporary and remain only valid during the lifetime of the application. In order to ensure a persistent storage of objects as well as their relations to other objects, the use of persistent object identifiers, which are unique within the scope of the application, has been introduced in Chapter 3.

The two object reference attributes are therefore replaced by two string identifiers, namely `fromId` and `toId`. The edge object can then be stored persistently. In order to access or operate on the object, the application identifier manager is called to resolve the actual object reference.

However, the major drawback of this design is that it is static. Especially in the example of the topology definition, new `edge` objects may be added during the life-time of the topology model, old objects deleted, renamed or redefined to reference different `vertex` objects. The logic for these operations must be contained in the topology objects. This makes class structures, from which the objects are instantiated, large and difficult to maintain.

Object relations stored in external relation-objects : An alternative implementation stores all information regarding relationships between objects not as attributes of the object but in special relation-objects. Relation-objects are contained in different relation-sets, which represent the mathematical relation which defines the connection between sets of objects. This permits the consistent management of relations and allows for easy class maintenance. Another relation can be defined by a single method.

In order to illustrate the difference between the two concepts, as well as the complexity involved in the simple example, the `Edge` and `Vertex` classes are

now presented and briefly discussed for the reference-identifier and relation-object designs. Figure 4.5 shows a class diagram for the objects involved in the design, as well an overview diagram. Only the relevant methods have been detailed.

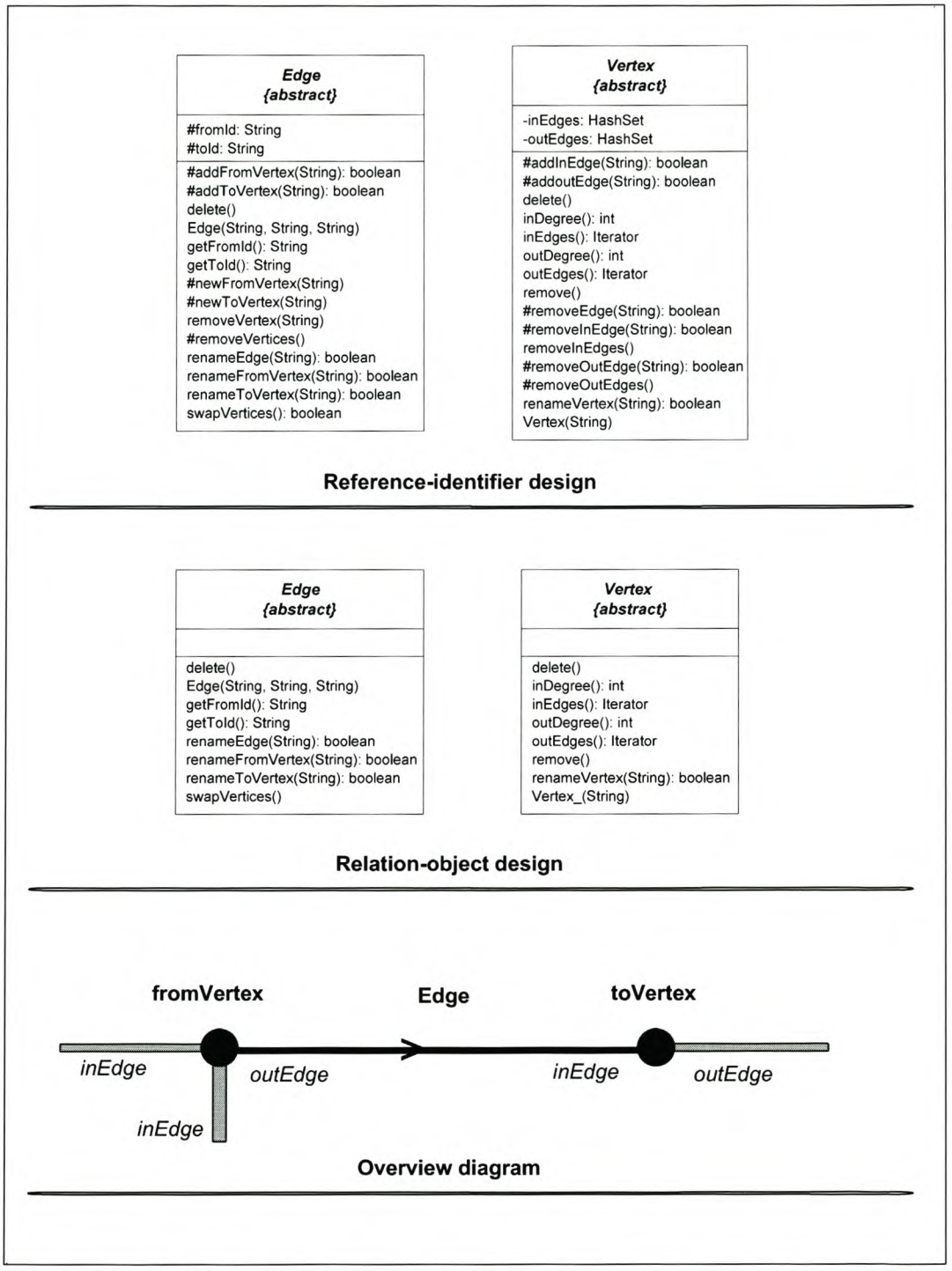


Figure 4.5: Class and overview diagrams to illustrate relation complexity

Figures 4.6 and 4.7 show the Java source code of the `Edge` class, for the reference-identifier and relation-object designs respectively. Figures 4.8 and 4.9 show the Java source code of the `Vertex` class, for the reference-identifier and relation-object designs respectively. The line numbers match between the two designs, and are used to reference the source code. The source code for the central relation-manager is not presented here, but will be discussed in the next paragraph.

The basic functionality supported by theses classes is:

- create new topology elements via the constructor
- provide direct access to the referenced objects using accessors
- provide data consistent functions to swap vertices for links and rename edge or vertices
- delete edges or vertices in a consistent way

Discussion of the relation complexity example : In the reference-identifier design (Figure 4.6) the object references appear as String attributes in lines 15 and 16. This is clearly absent in the relation-object design, where the data are stored centrally in relation objects. Similarly two private HashSet variables `inEdges` and `outEdges` are used in Figure 4.8, lines 18 and 19 to store the String references to the incoming and outgoing edges of a vertex.

The constructor of the reference-identifier (lines 21 to 25) calls utility methods (lines 122 and 132) which assign the `fromId` and `toId` if they are unassigned and also call the `addOutEdge()` (add outgoing edge) and `addInEdge()` (add incoming edge) methods of the newly connected vertex objects (lines 101 and 107 of Figure 4.8).

It can already be seen that the `Edge` class is tightly interwoven with the `Vertex` class and vice versa.

Other examples are the `renameEdge()` or `renameVertex()` methods (lines 62 and 51 in the respective files), where methods in the related objects are called to remove and add identifiers. In line 71 and 68 respectively, it can be seen that objects in external models must be updated, for example in the `DataModel`. Therefore the object contains a graph of relations with other objects, which can be very difficult to maintain.

Another problem is the final removal of an object from the application. Line 71 (Figures 4.6 and 4.8) assumes that no other object in the application has a reference to the renamed object (and thus the deleted application identifier). The more objects there are which stand in relation to each other, the bigger the chance that the programmer oversees a relation to be renamed or deleted. The same applies to the `delete()` method.

The equivalent methods in the relation-object design are uncluttered, as all relation operations are managed centrally from the `Rel` class, whereas application-object rename and delete operations are managed from the `App` class.

The next paragraphs describe the relation-manager functionality in detail.

```
package TopoClasses;

import java.util.*;
import Application.*;
import CoreClasses.DataModel;

/**
 * Abstract class for Edges in the Topology
 */
public abstract class Edge extends AppObject{

/**
 * Instance variables
 */
protected String fromId;
protected String toId;

/**
 * Constructor
 */
public Edge(String fromId, String toId, String id) {
   super(id);
   newFromVertex(fromId);
   newToVertex(toId);
}



/**
 * Accessors
 */
public String getFromId(){return fromId;}
public String getToId(){return toId;}



/**
 * Utility functions
 */
 public boolean swapVertices(){
 String tmpId;

  if ((fromId!=null)&&(toId!=null)) {
    Util.vertex(fromId).removeOutEdge(this.id);
    Util.vertex(toId).removeInEdge(this.id);

    Util.vertex(toId).addOutEdge(this.id);
    Util.vertex(fromId).addInEdge(this.id);

    tmpId=fromId;
    fromId=toId;
    toId=tmpId;
    return (true);
  }else
    return (false);
 }

     public boolean renameEdge(String newId){
            if (!App.containsObject(newId)) {

               Util.vertex(fromId).removeOutEdge(id);
               Util.vertex(fromId).addOutEdge(newId);

               Util.vertex(toId).removeInEdge(id);
               Util.vertex(toId).addInEdge(newId);

               DataModel.renameEdge(id,newId);

               App.removeObject(id);
               id=newId;
               App.addObject(this);

               return (true);
              }else
              return (false);
      }


  public boolean renameFromVertex(String newId){
          if ( App.containsObject(newId)) {
          Util.vertex(fromId).removeOutEdge(id);
               fromId=newId;
             Util.vertex(fromId).addOutEdge(id);
               return (true);
              }else
                 return (false);
  }

  public boolean renameToVertex(String newId){
          if (App.containsObject(newId)) {
               Util.vertex(toId).removeInEdge(id);
                 toId=newId;
               Util.vertex(toId).addInEdge(id);
               return (true);
              }else
                 return (false);
  }



/**
 * Removal function
 */


public void delete(){
  removeVertices();
  DataModel.removeEdge(id);
  App.removeObject(id);
 }


/**
 * Package level utility functions
 */
  void newFromVertex(String vertexId){
   if (addFromVertex(vertexId))
     Util.vertex(fromId).addOutEdge(this.id);
  }
  boolean addFromVertex(String vertexId){
    boolean b = (fromId==null);
    if (b) fromId=vertexId;
    return b;
  }

  void newToVertex(String vertexId){
   if (addToVertex(vertexId))
     Util.vertex(toId).addInEdge(this.id);
  }
  boolean addToVertex(String vertexId){
    boolean b = (toId==null);
    if (b) toId=vertexId;
    return b;
  }

  void removeVertices(){
        Util.vertex(fromId).removeOutEdge(id);
        Util.vertex(toId).removeInEdge(id);
        fromId=null;
        toId=null;
      }


public void removeVertex(String excludeVertex){
 if (excludeVertex==fromId)
   Util.vertex(toId).removeEdge(id);
 else
   Util.vertex(fromId).removeEdge(id);

 fromId=null;
 toId=null;

 DataModel.removeEdge(id);
 App.removeObject(id);
}
}
```

Figure 4.6: Edge class for reference-identifier design

```
package TopoClasses;

import java.util.*;
import Application.*;
import CoreClasses.TopologyModel;

/**
 * Abstract class for Edges in the Topology
 */
public abstract class Edge extends AppObject{

/**
 * No Instance variables
 */



/**
 * Constructor
 */
public Edge(String fromId, String toId, String edgeId) {
   super(edgeId);
   Rel.constructReverseRelations(this.id,fromId,
     Rel._RelSuffixes[Rel._OUT_EDGES],
     Rel._RelSuffixes[Rel._FROM_VERTEX]);
   Rel.constructReverseRelations(this.id,toId,
     Rel._RelSuffixes[Rel._IN_EDGES],
     Rel._RelSuffixes[Rel._TO_VERTEX]);
}


/**
 * Accessors
 */
public String getFromId(){return Rel.getToId(id,
                    Rel._RelSuffixes[Rel._FROM_VERTEX]);}
public String getToId(){return Rel.getToId(id,
                    Rel._RelSuffixes[Rel._TO_VERTEX]);}


/**
 * Utility functions
 */
public void swapVertices(){
   Rel.swapRelations(id,
     Rel._RelSuffixes[Rel._FROM_VERTEX],
     Rel._RelSuffixes[Rel._TO_VERTEX]);
}









public boolean renameEdge(String newId){
  String newRetId=App.renameObjects(id,newId);
  if (newRetId==null )
    return false ;
  else
    return true ;
}









public boolean renameFromVertex(String newId){
  String newRetId=App.renameObjects(getFromId(),newId);
  if (newRetId==null )
    return false ;
  else
    return true ;
}


public boolean renameToVertex(String newId){
  String newRetId=App.renameObjects(getToId(),newId);
  if (newRetId==null )
    return false ;
  else
    return true ;
}



/**
 * Removal function
 */

public void delete(){
  App.deleteObjects(id,true );
}



/**
 * No Package level utility functions
 */

}
```

Figure 4.7: Edge class for relation-object design

```
package TopoClasses;

import java.util.*;
import UtilClasses.*;
import Application.*;
import CoreClasses.DataModel;


  /**
   * Abstract class for Vertices in the Topology
   */
public abstract class Vertex extends AppObject{

  /**
   * Instance variables
   */

    private HashSet inEdges;
    private HashSet outEdges;

  /**
   * Constructor
   */

    public Vertex(String id){
      super(id);
      inEdges = new HashSet();
      outEdges = new HashSet();
    }


  /**
   * Accessors
   */

    public Iterator inEdges() {return inEdges.iterator();}

    public Iterator outEdges() {return outEdges.iterator();}


    public int inDegree() {return inEdges.size();}

    public int outDegree() {return outEdges.size();}



  /**
   * Utility functions
   */

  public boolean renameVertex(String newId){
      if (!App.containsObject(newId)) {


      Iterator i = inEdges();
      while (i.hasNext()){
        Edge e = Util.edge(i.next());
        e.toId=newId;
      }

       i = outEdges();
      while (i.hasNext()){
        Edge e = Util.edge(i.next());
        e.fromId=newId;
      }


      DataModel.renameVertex(id,newId);


      App.removeObject(id);
      this.id=newId;
      App.addObject(this);


      return true;
    }else
    return false;
  }



    public void remove() {
     removeInEdges();
     removeOutEdges();
    }

    public void delete() {
      remove();
      DataModel.removeVertex(id);
      App.removeObject(id);
    }


  /**
   * Package level utility functions
   */


    boolean addoutEdge(String edgeId){
    boolean b= !outEdges.contains(edgeId);
    if (b) outEdges.add(edgeId);
    return b;
    }

    boolean addInEdge(String edgeId){
    boolean b= !inEdges.contains(edgeId);
    if (b) inEdges.add(edgeId);
    return b;
    }


        boolean removeInEdge(String edgeId) {
         return inEdges.remove(edgeId);
        }

    boolean removeOutEdge(String edgeId) {
     return outEdges.remove(edgeId);
    }

    boolean removeEdge(String edgeId){
     return (outEdges.remove(edgeId) &&
             inEdges.remove(edgeId));
    }

    void removeInEdges(){
    Iterator i = inEdges();
    while (i.hasNext()) {
      Util.edge(i.next()).removeVertex(id);
    }
    inEdges.clear();
    }

    void removeOutEdges(){
    Iterator i = outEdges();
    while (i.hasNext()) {
      Util.edge(i.next()).removeVertex(id);
    }
    outEdges.clear();
    }
  }
```

Figure 4.8: Vertex class for reference-identifier design

```
package TopoClasses;

import java.util.*;
import UtilClasses.*;
import Application.*;
import CoreClasses.TopologyModel;


/**
 * Abstract class for Vertices in the Topology
 */
public abstract class Vertex extends AppObject{

/**
 * No Instance variables
 */



/**
 * Constructor
 */

  public Vertex(String vertexId) {
    super(vertexId);
  }



/**
 * Accessors
 */

  public Iterator inEdges() {return Rel.getToIdsIterator(id,
                    Rel._RelSuffixes[Rel._IN_EDGES]);}
  public Iterator outEdges() {return Rel.getToIdsIterator(id,
                    Rel._RelSuffixes[Rel._OUT_EDGES]);}

  public int inDegree() {return Rel.getToIdCount(id,
                    Rel._RelSuffixes[Rel._IN_EDGES]);}
  public int outDegree() {return Rel.getToIdCount(id,
                    Rel._RelSuffixes[Rel._OUT_EDGES]);}


/**
 * Utility functions
 */

public boolean renameVertex(String newId){
  String newRetId=App.renameObjects(id,newId);
  if (newRetId==null)
    return false;
  else
    return true;
}









public void remove(){
  App.deleteObjects(id,false);
}

public void delete(){
  App.deleteObjects(id,true);
}

}
```

Figure 4.9: Vertex class for relation-object design

RelObject : This class defines the relation-objects in the AC approach and is located in the Application package. It is derived from the `AppObject` class, so that it shares the object identifier management functions and `id` field. It contains four private fields which are used to store the from- and to-identifiers as well as the group-code of the from-object and the to-object. The group code is used to structure relation-objects into groups for the search operations. These groups correspond to mathematical relations and are therefore also implemented as relation-set objects. Figure 4.10 shows a class diagram with the key properties and methods of the RelObject and the associated Relation and Rel classes.

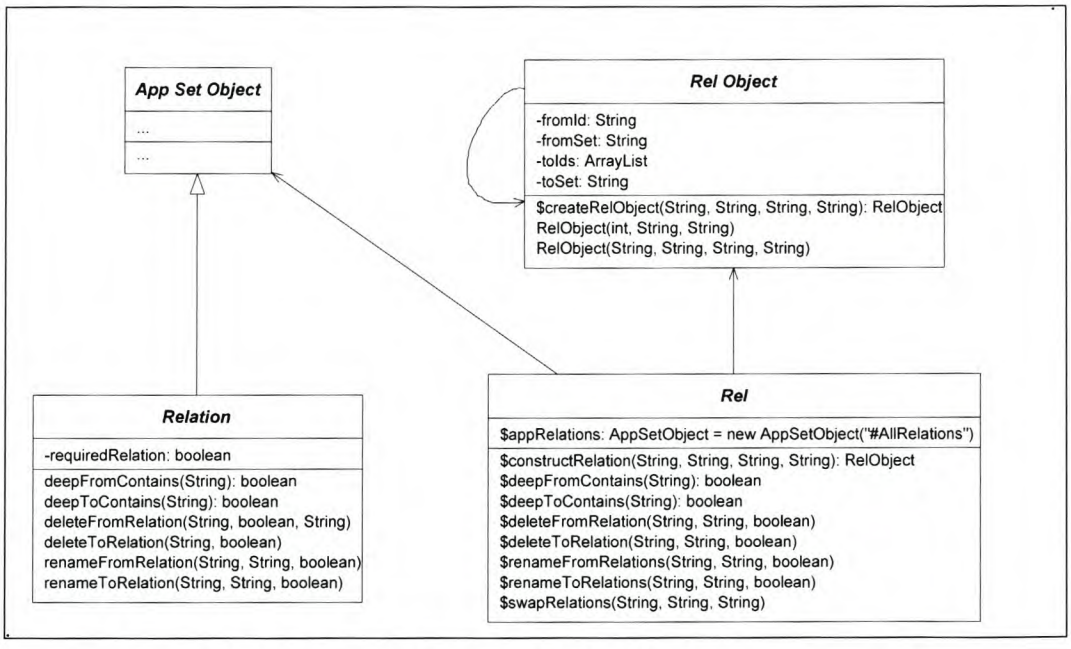


Figure 4.10: Class diagram of RelObject, Relation and Rel

For example, let the topological association between an edge (pipe) with identifier say EDGE701 and its from vertex (upstream manhole), say VERTEX234 be stored as a relation. A `RelObject` object is created from a factory method in the `Rel` class, with the combined application `id` @EDGE701.FVERTEX, and the field data:

- `fromId` : EDGE701
- `fromSet` :.OEDGES (outgoing edges)
- `toIds` : VERTEX234 (set of identifiers)
- `toSet` : .FVERTEX (from vertex)

The '@'-sign is used as a prefix to identify a relation-object in the application and the '.'-sign is used as a prefix to identify a set-object. The `toIds` field can contain more than one identifier in a Java ArrayList. This accommodates one-to-many associations. The object is created preferably via the `createRelObject()` factory method, which checks and verifies if the relation-object already exists in the application. If so, the new to-identifier is added to the `toIds` field, otherwise the normal constructor is called to create the new relation-object.

Non-application objects, such as Java objects, can also be accommodated in directed mappings from the foreign object to an application-object. The `fromId` field thus contains the HashCode of the object as String, and the `fromSet` field is null.

Furthermore utility functions are included in the class to access and change the four fields, as well as the objects associated with the identifiers.

Relation-set : The identifier of relation-objects is added to an instance of the `Relation` relation-set class, which has the identifier .FVERTEX in the above example. This relation-set contains all the relation-objects which adhere to the *to-from-vertex* relation, for example the above EDGE701 *to-from-vertex* relation-object.

The `Relation` class extends the `AppSetObject` class discussed previously. Therefore methods of the application-set class are inherited. It defines the field `requiredRelation` which is a flag for the entire relation-set, and thus applies to all relation-objects in the relation-set. This flag signals that the from-object in the relation cannot remain alive after the relation-object has been removed, and that the application-object must be completely removed from the application, together with any relation-objects that originate from it. For example, if a `Vertex` object is removed, then the connected `Edge` object must also be removed, as otherwise an inconsistent system would result. However, if an `Edge` object is removed, the linked `Vertex` objects are not removed automatically, as they can still define the topology of other `Edge` objects or could be used to define new `Edge` objects in future.

Search functions : Several search functions are provided in the `Relation` class to ensure that a consistent application-wide management of relations is guaranteed. The `deepFromContains()` method searches all objects in the relation-set if they contain `fromId` as from-object in the relation. In

this method, the from-object identifier is combined with the to-relation set-identifier (group) to check against the key of the relation-objects. This provides a fast constant time HashMap based search which makes use of the Java HashSet `contains` operator. On the other hand the `deepToContains()` method searches all the objects in the relation-set if they contain `toId` as to-object in the relation. Normally this function should not be used, as it is substantially slower than `deepFromContains()`. This is because the to-object identifier is not stored in the key of relations, and therefore a sequential search over all relations must be performed using the Java `Iterator` class. As soon as an object has been found, the search is terminated. This results in a search time which is proportional to the total number of elements in the set and the backing HashMap.

Relation consistency - deletion : Two methods have been implemented for this purpose: the method `deleteFromRelation()` which searches on from-objects in the relation set, and `deleteToRelation()` which searches on to-objects in the relations set. Again as discussed for the search functions, the first method can result in a constant time order $O(0)$ for the initial search operation and should be preferred over the second method which requires a time of $O(n)$.

The program logic behind the `deleteFromRelation()` method is of interest and discussed in more detail : Firstly it removes the unique relation from this relation-set with `fromId` as from-object. Then it attempts to remove the relation-object from the application with the inherited `AppSetObject.delete()` method. By specifying an additional `toId` parameter it can be directed to delete only one specific relationship in the case of a one-to-many relation. This is done by only removing one to-object identifier from the `toSet` of the relation-object.

Optionally the reverse relationship (the relation-object which is defined from `toId` to `fromId` and contained in the `fromSet` relation-set), can be deleted, without the requirement of another search, as the identifiers of the reverse object and relation-set are stored in the fields of this relation-object.

Finally if the `requiredRelation` property is set in the parent relation of a relation-object, then the from-object is removed completely from the application, using the `App.deleteObjects()` method.

Relation consistency - renaming : Two methods are implemented for this purpose: the `renameFromRelation()` method searches over the from-objects in the relation-set, and the `renameToRelation()` method searches over to-objects in the relation-set. The method is similar to the delete method, except that a new relation-object is created using the `RelObject()` constructor with the proposed new identifier for the from-object. This ensures that if the system allocates an alternative identifier, the correct new identifier is used. The inherited `rename()` method then renames the unique old identifier to the unique new identifier. Finally the old relation-object is completely removed from the application.

The renaming of reverse relations is handled in a similar way for the deletion of relation-objects.

Utility class - Rel : This class is defined as a utility class for all relation-set objects. Its primary function is to store and maintain the `appRelations` variable. This variable is a static instance of the `AppSetObject` and is the parent set of all relation-sets in the application. By grouping all relation-sets in the application together in one set, common operations can be performed over all sets, using methods from the `AppSetObject` class and this class.

This class provides two factory methods. The one constructs a directed map, the other an invertible map relationship. Should the relation or relation-objects not exist, new objects are created automatically.

The other methods discussed for the relation-set object are now extended to iterate over the `appRelations` variable, and then into each contained relation-set.

The method `swapRelations()` exchanges the from-objects in two binary relations. For example for edge EDGE203, the from-vertex VERTEX12 is exchanged with the to-vertex VERTEX17. This means that the two relation-objects, EDGE203.TVERTEX(VERTEX17) and EDGE203.FVERTEX(VERTEX12) must be changed to EDGE203.FVERTEX(VERTEX17) and EDGE203.TVERTEX(VERTEX12). The reverse relations, VERTEX17.INEDGE(EDGE203)) and VERTEX12.OUTEDGE(EDGE203) must also be exchanged. The normal relations are easy to change, as the parent relation-set does not change. However the reverse relations have to be deleted from the application as well as from their parent relation-sets respectively. Two new relation-objects are then created using the `constructRelation()` method.

4.6 Implementation of the engineering process

The engineering process as discussed in paragraph 2.3 is adapted slightly for the AC approach. Although the models are now no longer stand-alone applications, the models are still instances of model-objects. The engineering process now consists of the construction and population of the *Product-data model*, from the CAD drawing contained in the *CAD model*, the incorporation of elevation data using the *Elevation model*, the execution of hydraulic analyses and design operations from the *Hydraulic model*, the use of the *Visualization model* to visualize the engineering model in the CAD environment and the use of the *Presentation model* to display analysis and design results. Figure 4.11 shows this relation. The numbers are referenced in the discussion. Figure 4.12 shows the class diagram.

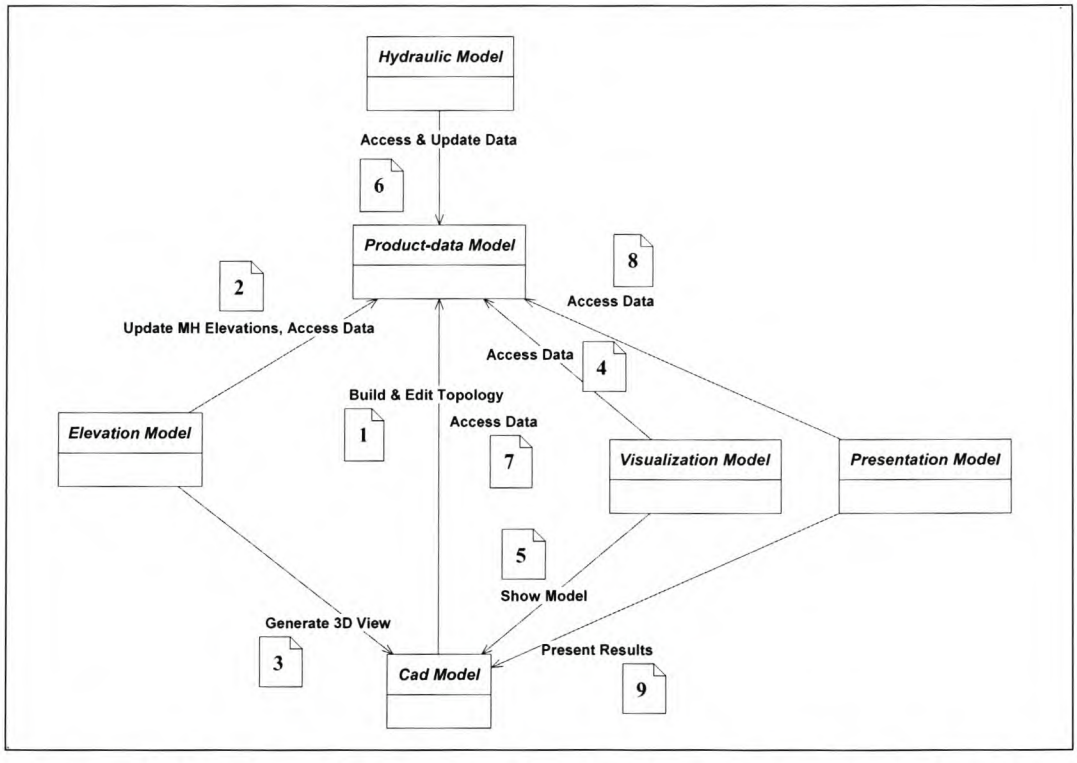


Figure 4.11: Collaboration diagram showing the process flow in the AC approach

Construction of the *Product-data model* : The engineering process starts with the collection, evaluation and processing of data defining the topology of the existing sewer system such as contained on paper or digital plans in a CAD drawing. As the implementation of a fully functional CAD system lies

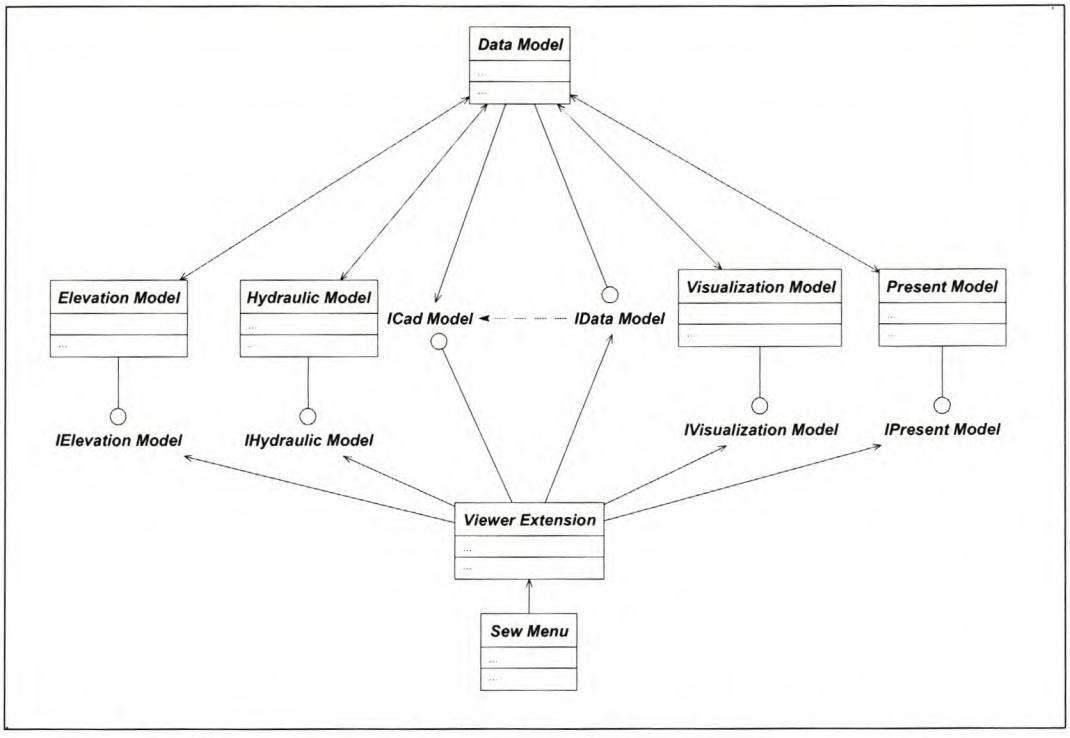


Figure 4.12: Class-diagram showing models and structure of the AC approach

outside the scope of this research, only the input of DXF file format into the *CAD model* is supported. This drawing contains a collection of connecting lines representing the sewer pipes and a circle representing the outfall manhole. It can then be extended externally with text attributes such as manhole numbers as well as other key hydraulic information, e.g pipe diameters and the number of connected land parcels. During the automatic construction of the topology, the relevant CAD entities are extracted from the CAD drawing and added one by one to the persistent data structure in the *Product-data model* method (1). The engineering data is then stored in the *Product-data model* as application-objects. Within the *CAD model* the topology of hydraulic structures can be added, removed and changed graphically and interactively.

Incorporation of the *Elevation model* : The *Product-data model* must be updated with an elevation model of the sewer system (2). The ground elevation at the nodes (manholes) of the sewer network can be interpolated from data in a surveyor data-point measurement file (xyz-point file) using the methods available in the *Elevation model*. The elevations are required to calculate the ground slope between connecting manhole ground elevations along which

sewer pipes are to be designed, as well as for the verification of captured existing manhole lid levels and invert levels. The calculated slopes can be edited interactively by the user.

The xyz-points from the *Elevation model* file can be visualized as a 3D surface model in the *CAD model* (3). The need for a separate *Topography model* therefore falls away. The export of data for an *Elevation model* from a 3D CAD drawing falls outside the scope of this research.

Analysis and design using the *Hydraulic model* : The sewer system is presented by the *Visualization model* which accesses the data in the *Product-data model* (4) and draws a visualization into the *CAD model* (5). Then the hydraulic analysis and design operations can be performed.

The implemented hydraulic model is a modified version of the one implemented for the MC design approach. The method accumulates all flows in the sewer system and then performs a hydraulic analysis. Should errors in the topology or attribute data be found during the analysis, a correction is required. All errors can be corrected directly from the user-interface provided in the *CAD model.* It might be necessary to update the elevations again using functions from the *Elevation model.* For a new system with unknown pipe diameters, the design algorithm in the *Hydraulic model* allows for the optimum diameter to be calculated given the pipe slope information. At this stage, the extensive production of XY-graphs and tables for the presentation of hydraulic results has not been implemented in the pilot application.

Use of the *Visualization model* : During the interactive use of the *Hydraulic model* the model must be viewed graphically. This is accomplished in the *Visualization model.* At any stage after the topology of the sewer system has been entered in the *Product-data model* (4), a 2D graphical visualization of the system can be generated into the *CAD model* (5). The graphical interactive environment of the *CAD model* allows the user to query any hydraulic structure for data from the *Product-data model* (7).

Use of the *Presentation model* : Results of hydraulic calculations can be shown by drawing pipes and manholes (nodes) onto separate layers in the *CAD model* (9) using colour legends for a specific input or result variable. In this process methods in the *Presentation model* must first access data from the *Product-data model* (8).

The view generated by the *Presentation model* can be updated manually at any time should changes be made to the *Product-data model*.

The functionality to query interactively any hydraulic element, which was supplied by the *Geographical model* in the MC design, has been incorporated into the *Presentation model* and *CAD model*.

4.7 Functionality of the AC software system

In order to illustrate all aspects of the AC approach for a civil engineering software application, it was decided to write a completely new application, SEWSAN AC. This section describes the choice of the programming language selected for the implementation.

Programming language used : As this application is written coherently in one programming language, a modern object-oriented programming language was selected. Although *Object Pascal* as implemented in Borland Delphi [4] is used in many engineering offices as rapid application development language, the choice fell on *Java* as implemented by Sun Microsystems. Not only is the language available freely; it also earns a growing world-wide acceptance and supports distributed computing concepts. As alternative the *C++* programming language could have been used.

The implementation of the AC approach has therefore been completed in the *Java* Platform, Standard Edition, version 1.3. [44].

4.8 Algorithmic background

This section describes the algorithms used for the hydraulic analysis and design in the SEWSAN AC Java application. A different approach to the contributor hydrograph method was chosen for the AC application. The approach adopted is based on the European practice of designing the sanitary sewer network of the separate sewer system.

Method : See ATV [2], Hosang/Bischof [18], Novotny [29], Department of Housing [8] and [9]. Sanitary sewers carry a mixture of sewage, industrial wastewater and clean water (extraneous flow). Clean water inputs into sanitary sewers are mainly inputs from cross connections between stormwater

drainage (broken manhole covers, illegal yard and roof drainage connections) and to a lesser degree inputs from ground water infiltration, and from leaking taps and valves. Excessive clean water inputs into sanitary sewers should be avoided wherever possible, as they result in treatment plant overloads.

Clean water inputs into sanitary sewers may be of the same magnitude or even greater than the sewage and industrial inputs. Therefore two design flows are generated in the network, a dry weather peak flow and a wet weather peak flow. The dry weather peak flow is used to check whether minimum flow velocities (0,5 m/s) for self-cleaning are maintained. The wet weather peak flow is used to check weather the pipe capacities (depending on pipe diameter and slope) are sufficient.

Dry weather peak flow for each pipe is computed by determining the average daily sewage production from all stands served by the pipe and multiplying it with a peak factor. The average daily sewage production follows from information regarding the type of residential zone (housing category, density, income class). The peak factor takes into account the variation in sewage production during the 24 hours of a day. Typical values are between 1,5 and 3,0 depending on the type of residential zone. Industrial wastewater flow follows from information regarding the type of industry (factory, hospital, hotel, etc.) and is added to the network as a point source inflow at a specific manhole.

Wet weather peak flow at each pipe is computed by multiplying the dry weather peak flow with an extraneous flow factor. The extraneous flow factor takes into account the inputs of clean water. Typical values are between 1,5 and 2,0 depending on the condition (old/new, maintenance) of the network.

Dry weather and wet weather peak flows are routed and accumulated downstream towards the outfall. (In this connection it must be noted that the effect of time lag attenuation is ignored, as this is sufficiently accurate for sanitary sewers: Whereas for stormwater sewers and combined sewers the runoff hydrograph of a short duration rainstorm is used as input into the network, the wet weather peak flow in a sanitary sewer is approximately constant for one or more hours. Assuming a typical flow velocity of 1,0 m/s, the sewage flow will travel 3 to 4 km within one hour. Therefore it is not necessary at junctions to lag the peaks of combining flows.)

Bottlenecks in the system are identified where the wet weather peak flow exceeds the flow capacity of a pipe. Pipes where the dry weather peak flow velocity falls short of the minimum velocity (0,5 m/s) are flagged.

Point sources : Provision is made to include user point sources in the analysis. It may be required to input a fixed flow rate at a specific manhole in the network. This may happen where industrial flow is discharged into the network (as discussed above), where pumps discharge into the network, where other networks join the network being analysed, or where a large network has to be fragmentized because of computer memory or computer time limitations.

Design option : The similar algorithm as for the MC design approach is implemented, but adapted for the new analysis method.

Interpolation algorithm : The same algorithm as for the MC design approach is used, but now fully implemented in Java.

4.9 Classes of the Product-data model

The Product-data model consists of the `DataModel` object as well as objects instantiated from several packages.

Use of packages : The following packages are defined for the Product-data model:

- Application : The application manager classes are located here. This defines an abstract top level for all the engineering classes.
- TopoClasses : The topology of the interconnected basic engineering objects are defined in this package.
- GeomClasses : The coordinate related information for the basic engineering objects is defined in this package.
- DTMClasses : The classes required for the update of elevations and the visualization of the topography are defined in this package.
- HydroClasses : This package contains the hydraulic data for the basic engineering objects.
- AnalClasses : The hydraulic analysis algorithms are defined in this package.

- DesignClasses : The design (optimization) algorithms are defined in this package.
- PresentClasses : The results are presented by classes in this package.
- FinalClasses : This package contains classes which are defined as final as well factory classes which create objects.

These classes are structured in a hierarchical way to ensure optimal sharing of common properties and methods. Figure 4.13 shows the hierarchical structure of classes and packages for the AC approach.

The detail of the classes is discussed in connection with the implementation of the engineering models in Section 4.10.

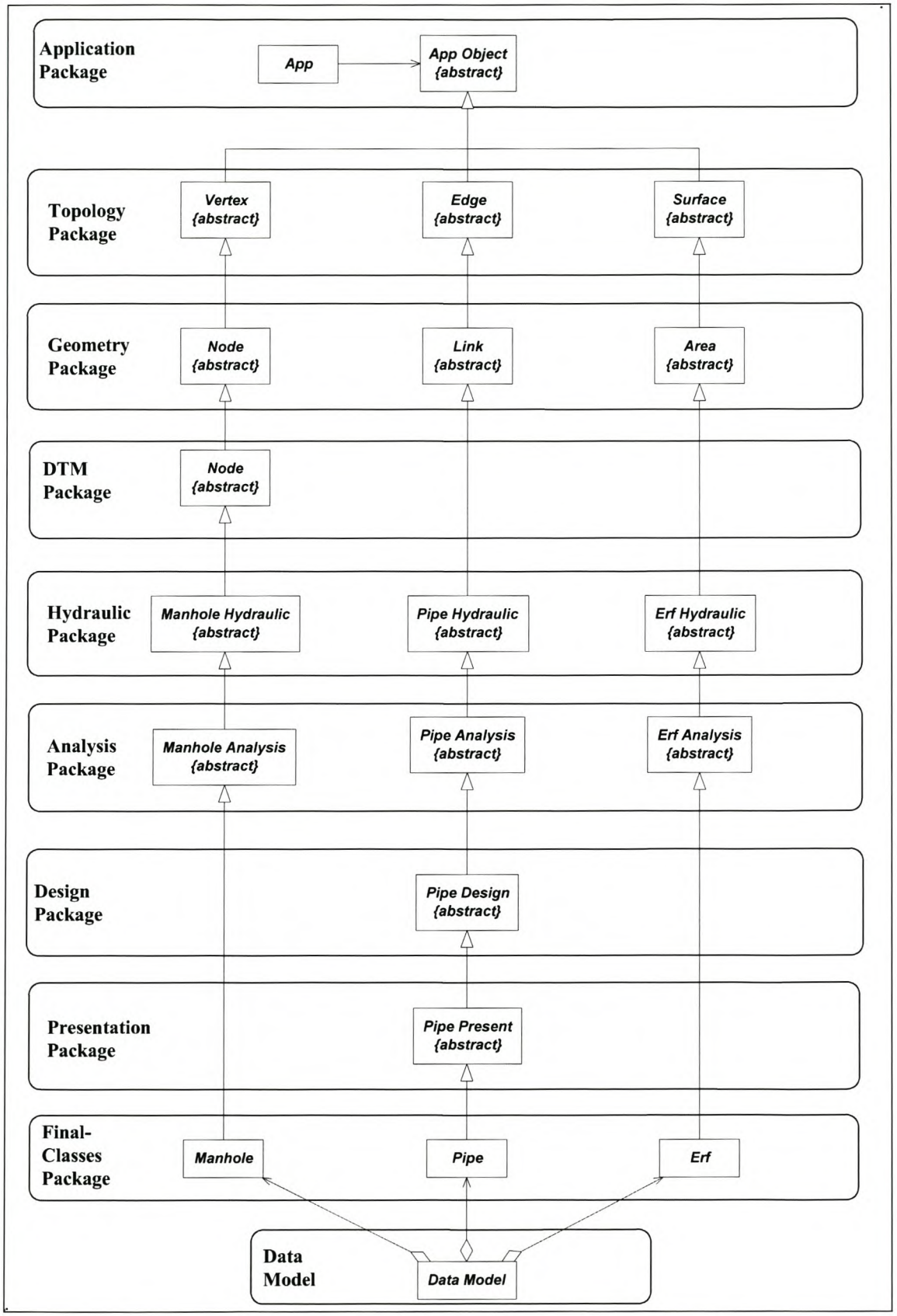


Figure 4.13: Hierarchy of classes for the AC approach.

The DataModel object: The class *DataModel* is located in the CoreClasses package. It extends the class `UnicastRemoteObject` to inherit functionality required for distributed computing scenario. The common operations which affect the central storage and management of the engineering data are collected in this class. It implements the `IDataModel` interface, which is accessed from the `CAD model`. Figure 4.14 shows this class.

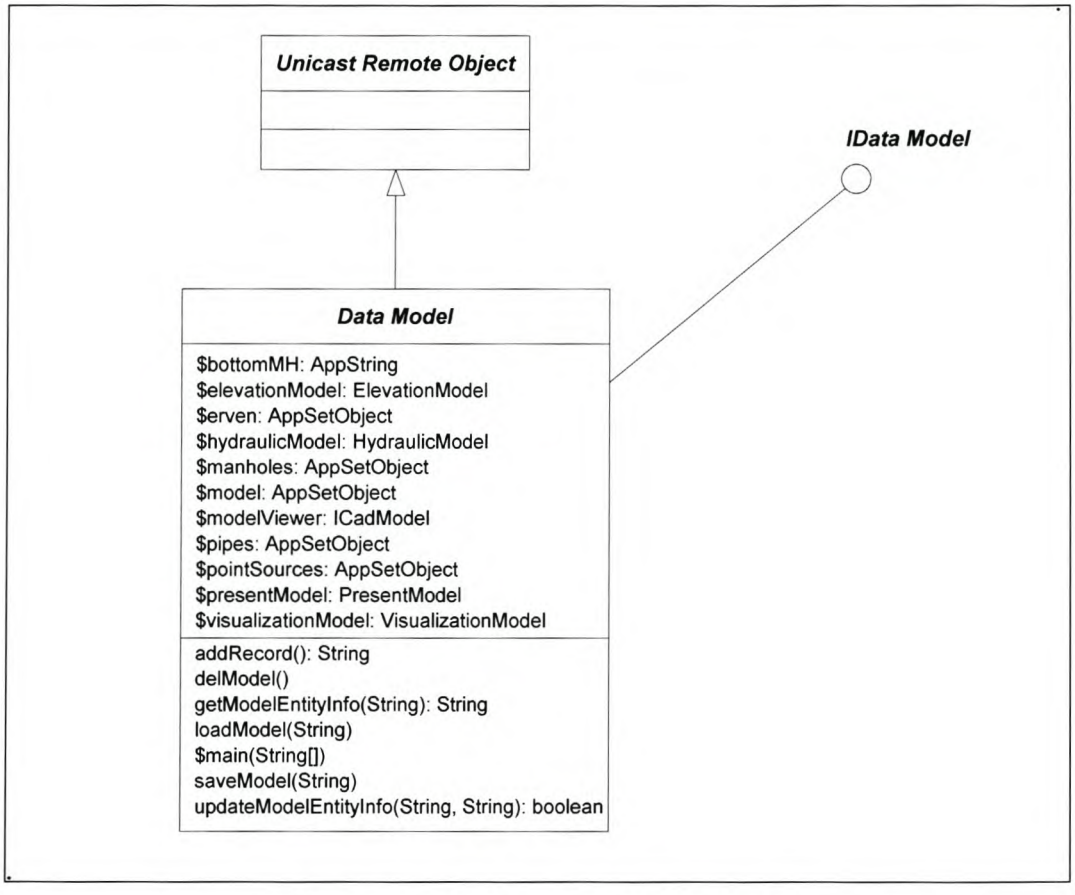


Figure 4.14: The DataModel class

The model-object contains instances of the set-objects which reference the basic engineering objects. As the AC approach currently only supports one active model in memory space, all instance variables are declared static. This improves the run-time performance, as the Java virtual machine can optimize the access to the data structure.

The following `AppSetObjects` are collected:

- model : This set contains the identifiers of the sets defined below. It provides direct access to the objects in the engineering model.

- manholes : This is the collection of the manhole identifiers in the application.
- pipes : This is the collection of the pipe identifiers in the application.
- erven : This is the collection of the erven (stands) in the application.
- pointSources : This is the collection of the point sources in the application.
- bottomMH : This *AppString* object defines the identifier of the bottom (outfall) manhole.

Furthermore instances of the visualization, elevation, hydraulic and present model-objects are stored in the *Data model*, as well as a reference to the *CAD model*. These variables provide direct access from the *Data model* to the model-objects.

The constructor of the *CAD model* object calls its inherited constructor to create an object which is compatible with the JAVA RMI system. It creates the model-objects.

Other primary functions are to provide iterators to the application sets and to implement methods which operate on the data of the application. This includes the `delModel()` method. In order to delete all objects in the application, the `deepClearSet()` method of the `appSets` object (which collects all set-objects defined in the application) is called. This method does a recursive (deep) operation to remove the object from all sets, but does not remove the object from the application. The `App.clearObjects()` method removes all application-objects from the application. As this is the only direct Java reference which refers to the application-objects, the Java Garbage Collection system, which removes unreferenced objects, can now remove these object from the memory space of the computer.

The `addRecord()` method is used by the *CAD model* during the `buildModelfromCAD()` operation or by the local `readFile()` method to add a data record to the Data model. A "record" contains the information required to define one pipe object, its upstream and downstream manhole objects as well as associated point sources and erven objects. This method is the primary way to construct the basic engineering objects, together with the required relation and set-objects.

The methods `getModelEntityInfo()` and `updateModelEntityInfo()` allow the retrieval and update of data for a specific pipe or manhole.

Persistent storage of data : The *Data model* is made persistent by storing it to disk with the `saveModel()` method. This methods makes use of the standard Java serialization technology to convert the application-objects to a data stream. This data stream is then written to disk. The method first stores the application-object HashMap, the application-set HashSet and then the relation-set HashSet. As static variables cannot be found automatically during the Java serialization process, they have to be written explicitly to the stream. This includes the set-objects defined in the *Data model*.

The application objects are automatically written to the stream because the application-object HashMap contains object references to all objects of the application in its value field. The corresponding `loadModel()` loads the data model into memory space.

For the distributed computing scenario, the `main()` method is performed when the *Data model* is executed as a stand-alone application on the server-side. This method then creates a `RMISecurityManager` object, and binds the name `DM` to an instance of `sewerModel`. This is a requirement for the JAVA RMI to function.

4.10 Implementation of the engineering models

This section briefly describes the functionality of the model-objects. The model-objects are analogous to the traditional engineering models in the MC approach. However, each model-class implements a model-interface, which defines the methods which must be implemented. A model-object does not contain any product data. All product data are located centrally in the *Product-data model*. However, the model-objects have full access to the *Product-data model* and its classes.

Hydraulic model : The *HydraulicModel* class extends the class `UnicastRemoteObject` of the `rmi.server` package of Java so that this class can be accessed using RMI from the User Interface located in the *CAD model* in the distributed computing scenario. Furthermore, this class implements the `IHydraulic` interface, which defines the public methods to be exported by this class. The use of an interface is not required for the stand-alone execution, but defines which methods are to be exposed publicly. The Java RMI distributed system, however, requires that all objects which are to be accessed remotely must implement an Interface. Figure 4.15 shows

this class.

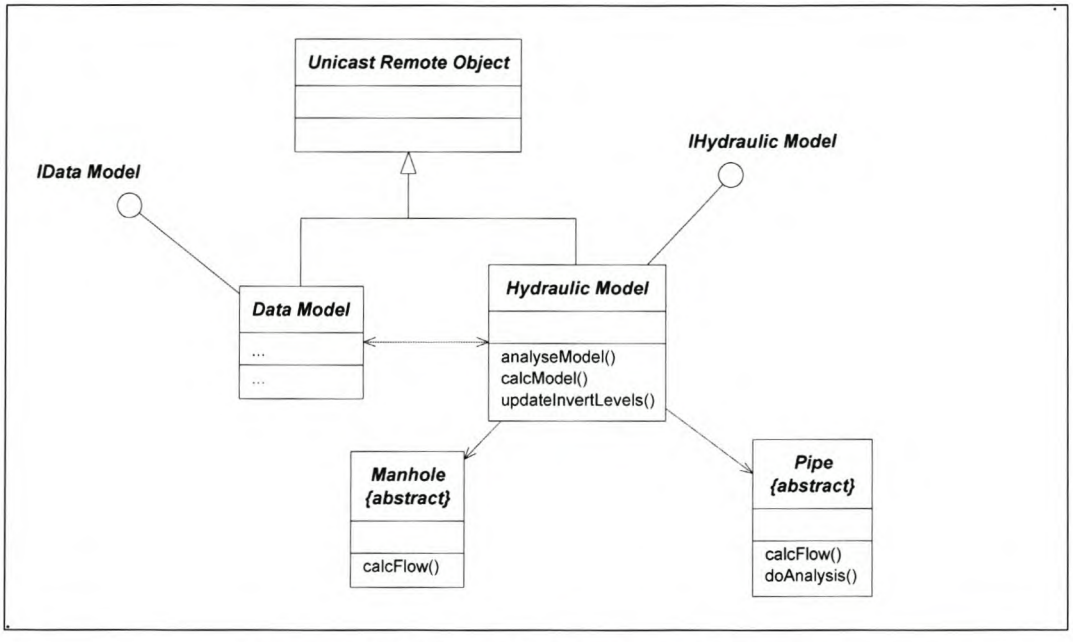


Figure 4.15: The HydraulicModel class

The *Hydraulic model* exposes three methods for the analysis and design of the sewer system, namely:

The method `updateInvertLevels()` calculates the invert levels of all pipes at the manholes from the elevations at the manholes by following a post-order tree traversal (from leaves to root) over the manholes and pipes. This methods ensures that the minimum slope requirement is adhered to.

Method `calcModel()` accumulates the flow in the sewer system by traversing the tree of pipes and manholes in post-order and accounts for the contribution from erven and point sources. Two flow conditions are calculated, namely a dry-weather flow condition and a wet-weather flow condition.

Method `analyseModel()` then performs the velocity calculation and the analysis by iterating sequentially over all pipes. The `Pipe.doAnalysis()` method for each pipe object is called. This is followed by the execution of the design algorithm.

Traversals : Tree traversals form a central part of the *Hydraulic model* and can be applied more generally to other applications. Firstly an in-order traversal (from root to leaves) is performed using the `calcTreeAge()` method (located in the TopoClasses.Util class), which assigns an "age" to

each pipe in the model, based on the number of pipes or distance from the root outfall manhole. Post-order tree traversals are carried out by creating a `POTreeTraversal()` object and using it as an iterator. An extract from the source code of the `updateInvertLevels()` method is shown in Table 4.1. The full source code for the traversal algorithms can be found in Appendix F.

Table 4.1: Traversal algorithm

```
public void updateInvertLevels()   throws RemoteException{
   Manhole usManhole;                                  //reference to us manhole
   Pipe pipe;                                          //reference to pipe

                                                       //do in-order traversal
   TopoClasses.Util.calcTreeAge(DataModel.bottomMH.getAppString(),this.dm);
                                                       //get post-order iterator,
                                                       //starting at bottom manhole
   Iterator nodeIds =  new POTreeTraversal(DataModel.bottomMH.getAppString(),dm);
                                                       //now iterate over nodes
   while (nodeIds.hasNext()) {
     usManhole =Util.manhole(nodeIds.next());          //get reference to manhole
                                                       //skip bottom manhole
     if (!usManhole.getId().equals(DataModel.bottomMH.getAppString())) {
       usManhole.calcOutInvert();                      //calculate the us invert for
                                                       //all pipes on ds side of usManhole
                                                       //now transfer level to ds
       Iterator pipeIds = usManhole.outEdges();        // manhole. Get iterator over all
       while (pipeIds.hasNext()) {                     //outgoing edges
         pipe = Util.pipe(pipeIds.next());             //get reference to pipe object
         pipe.calcDsInvert();                          //calc the new ds invert level
         pipe.getSlope().update(pipe);                 //now update the slope
       }
     }
   }
}
```

Visualization model : This class contains the operations for the visualization of the model. It implements the `IVisualizationModel` interface. Figure 4.16 shows the classes used for the visualization.

The model draws pipes and manholes given their identifiers. First an object reference must be obtained from the application id manager and then the `pipe.draw()` and `manhole.draw()` methods in the GeomClasses package are called.

The `draw()` method accesses the `modelViewer` object (an instance variable to the `ViewerExtension` class in the *CAD model*) via the `ICadModel` interface and executes the `drawLine()` method. The return value of this method is the Java HashCode of the created CAD entity. This HashCode is used in the `Rel.constructRelation()` method to construct a directed mapping between a non-application object (the CAD entity) and an application-object. This

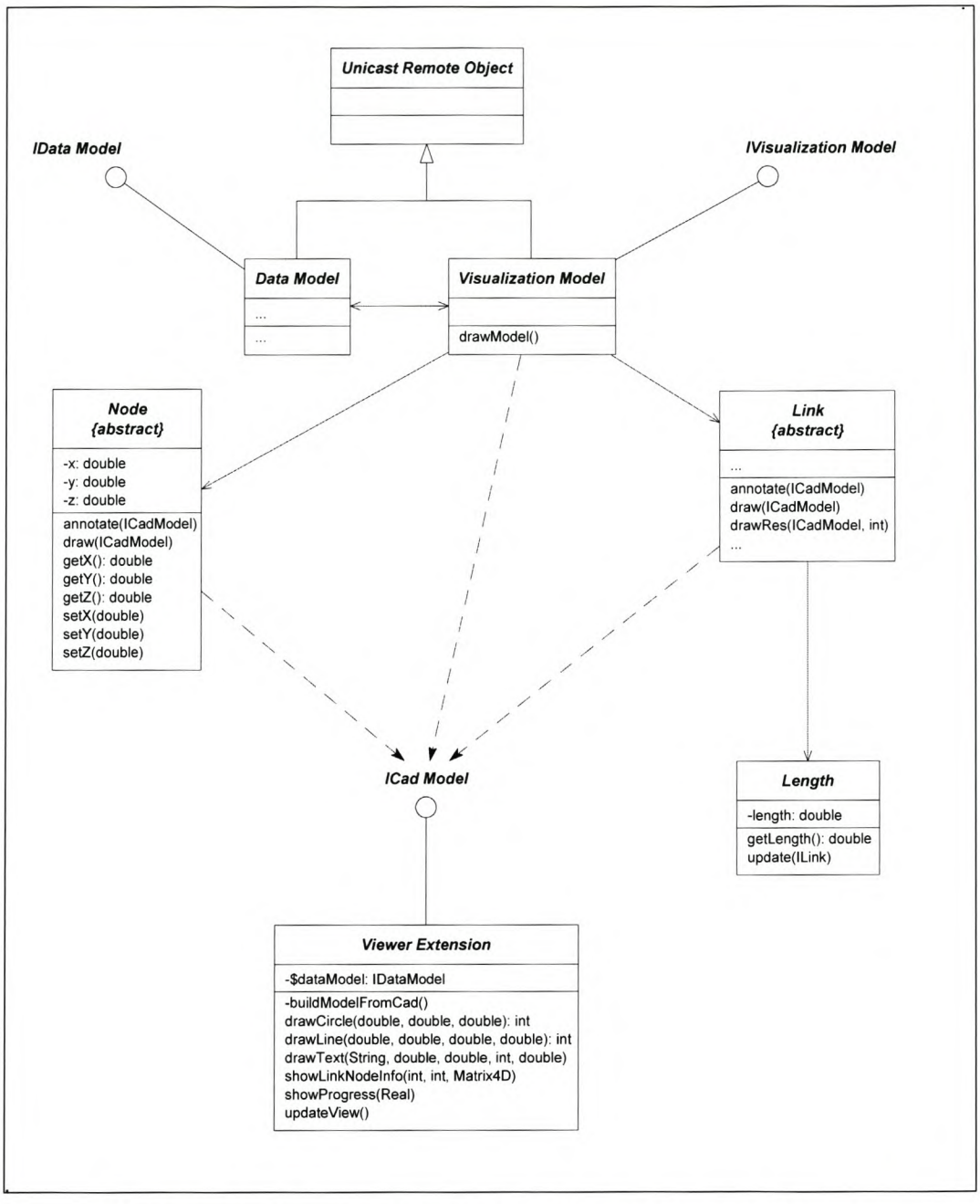


Figure 4.16: The *Visualization model* and associated classes

relation-object is used for example to obtain a reference to the application-object (pipe or manhole) which is visualized when the user clicks on a CAD entity.

The `drawModel()` method redraws the entire *Product-data model*. This method deletes all existing relation-objects between CAD entities and

application-objects using `Rel.deleteRelations()`. Then all pipes are traversed to call `pipe.draw()` and `pipe.annotate()` for each pipe.

The same procedure is repeated for the manholes. Finally the `modelViewer.updateView()` method is called to refresh the CAD display.

Presentation model : This model extends the class `UnicastRemoteObject` for the distributed computing scenario. This class contains the operations for the presentation of results. It implements the `IPresentModel` interface.

The `drawResModel()` method iterates over all pipes and calls the `pipe.drawResult()` in the `PresentClasses` package and draws pipe objects as colour-coded vectors and annotates result variables. Communication to the *CAD model* takes place via the defined `ICADModel` interface.

Elevation model : This model also supports the distributed computing scenario. This class contains all the operations which affect the elevation and topography models. It implements the `IElevationModel` interface. Figure 4.17 shows the classes used for the visualization.

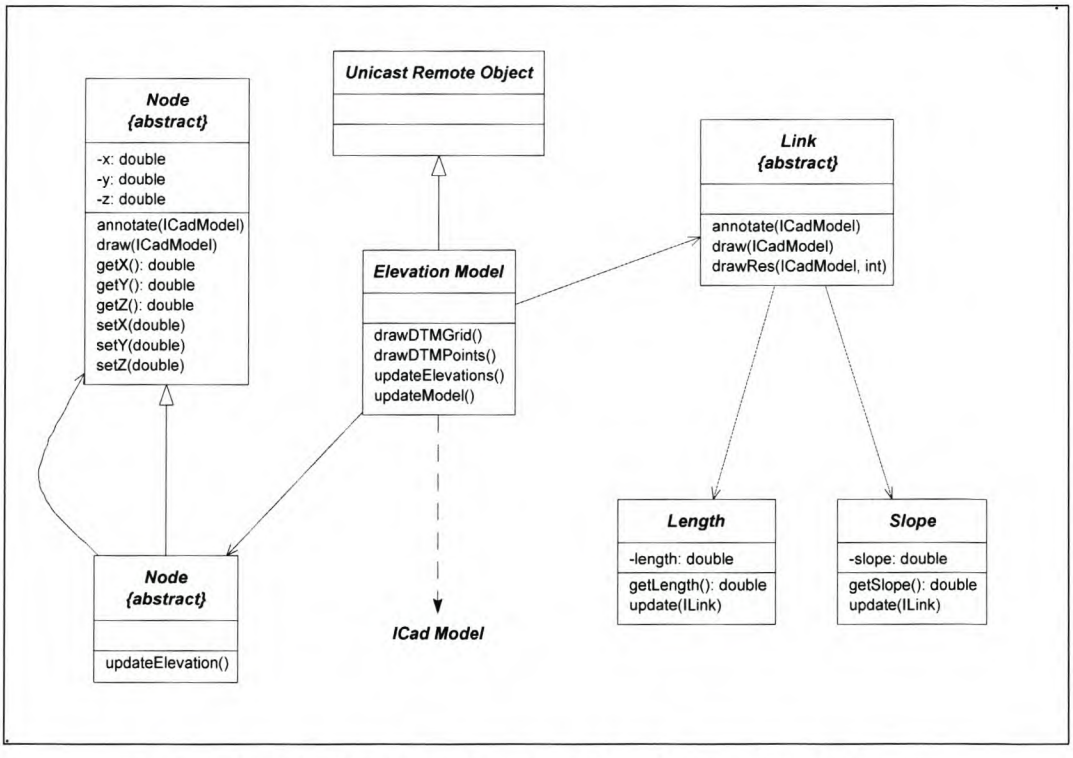


Figure 4.17: The *Elevation model* and associated classes

The method `drawDTMGrid()` visualizes the xyz-points of the elevation model

as 3D-mesh, provided the points are stored in a rectangular array. The `drawDTMPoints()` method is used to annotate the xyz-points with elevation labels. These methods call functions from the `DTMClasses` package. None of these classes operate on the application objects, as the points are only temporarily stored in memory.

The method `updateElevations()` interpolates the elevation for each manhole-object in the sewer system. Information regarding the quality of the interpolation (for example whether the elevation was interpolated or extrapolated) is stored in the manhole-object.

The method `updateModel()` is used to update the lengths and slopes of all pipe-objects in the *Product-data model*, given the new elevation. It traverses the pipe and calls the methods `pipe.getLength().update()` and `pipe.getSlope().update()` in the `Geometry` package.

CAD model : The `ViewerClasses` packages define a CAD visualization system in Java, which can read an industry standard 3D DXF file and visualize the geometry by projecting it on a plane. A full description of the *CAD model* falls outside the scope of this dissertation but it is written fully using the Java AWT API [1]. The interaction with the `Product-data model` is however outlined below. Figure 4.18 shows the classes used for the *CAD model*.

The CAD visualization system has its own object data structure for managing and visualizing its drawing space. It has therefore been decided to detach it from the engineering application. This ensures that the visualization system can be used for other applications as well and that it can be replaced easily with another product. In order to illustrate this concept, the interfaces between the engineering classes and the CAD visualization classes are described.

When detaching the CAD classes from the engineering classes, the MC approach where engineering objects are replicated in the *CAD model* is avoided. This is achieved by exposing only primitive methods for drawing, annotating and querying in the interface of the *CAD model*. However, as the *CAD model* requires information on user interaction (such as clicking on pipes) from the *Product-data model*, it must be able to access the engineering objects directly. This is accomplished by an interface to the *Product-data model*.

The user-interface in the *CAD model* is extended with a custom class, `SewMenu`. This is typical of commercial CAD systems. The *CAD model* provides the complete user-interface for the SEWSAN AC application.

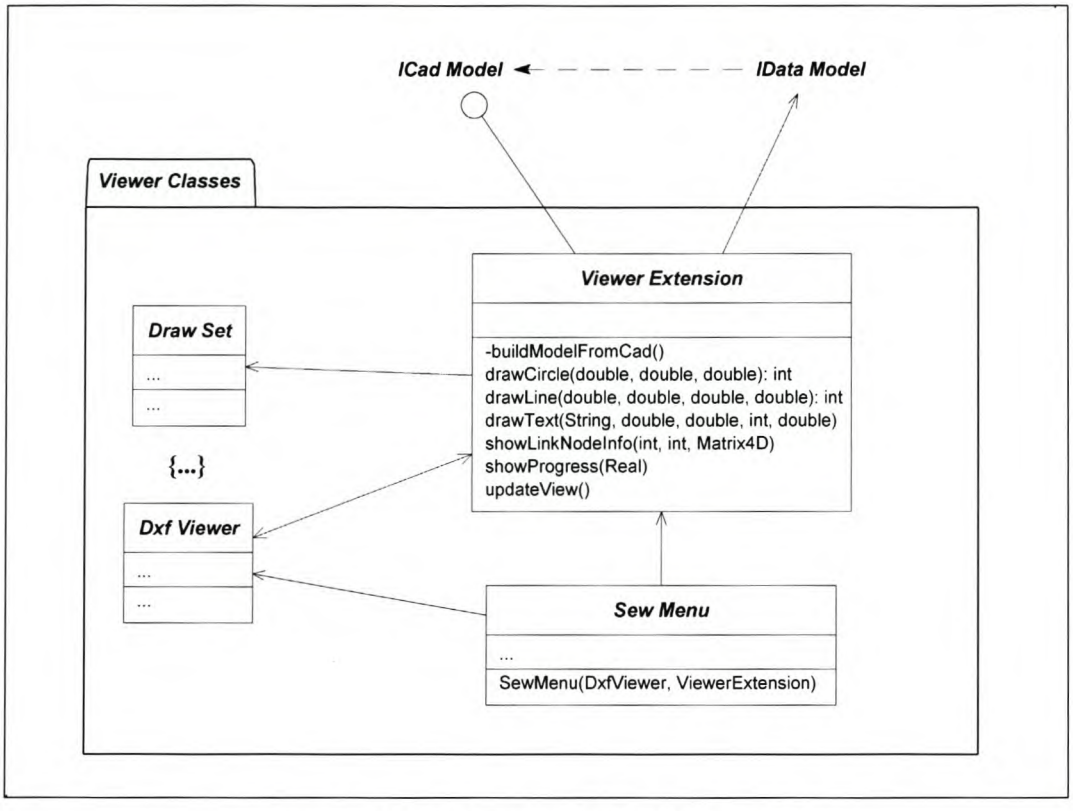


Figure 4.18: Classes of the *CAD model*

The implementation of the `ICadModel` is of interest, as it is accessed from the engineering classes. The *CAD model* is independent of the engineering models. This allows the implementation of a distributed computing scenario, where the *CAD model* with user-interface is available at the client side, and the engineering models are located at a central server location.

The `ViewerExtension` class is located in the *CAD model* and implements the primitives line, circle, symbol and text drawing functions, a progress bar and a status window. It exposes an `updateView()` method to force a refresh of the CAD system. The `buildModelfromCAD()` function interacts extensively with the CAD drawing to extract entities. It calls the `dataModel.addRecord()` method via the `IDataModel` interface to pass the extracted raw topology to the *Product-data model*.

The `showLinkNodeInfo()` method is called from the *CAD model* whenever the user clicks on a cad entity. It calls the `dataModel.getRelationToID()` method in the *Product-data model* to determine if an application-identifier is related to the CAD-object and then presents a formatted string using the `getModelEntityInfo()` method. This is an example where the relationship

between a non-application object (CAD entity) and an application-object (for example the pipe object) is queried.

This kind of interaction is typical for the communication between a *CAD model* and a *Product-data model.*

Figure 4.19 shows the user-interface of the *CAD model*

4.11 Extending the design for a distributed environment

In order to illustrate the conversion of the AC design pilot project for distributed computing, a scenario is considered where the *CAD model* is located locally and the *Product-data model* is located remotely at a server.

This concept follows the traditional two-tiered client-server application design. As the user-interface is graphically intensive, it is placed on the client side. A user-interface, which only requires textual input like an input form hosted by a web-server and accessed through a web-client, can also be considered as a three-tier alternative. However, the development of a web-server falls outside of the scope of this research.

The *Product-data model* does not require a user-interface but high processing power. The central off-site storage of persistent model data is advantageous.

Java RMI : Several client-server communication standards have been developed. Three choices are available to the Java programmer (Orfali [31]).

- Sockets : The developer can use low-level socket communication. The programmer must implement own methods to transfer object data over the wire and to manage the communication.
- CORBA : This is a standard for inter-operability between systems written in different programming languages. It requires software which is not included in Java.
- JAVA RMI : This technique provides a Java-native communication system which can be used for powerful client-server applications. The software subsystem is included in the Java distribution.

Web-browser integration : An alternative to convert the client code from a Java application to an Applet (which runs within a web-browser) is also evaluated for the SEWSAN AC application. This option provides an interesting alternative in the client-server scenario as the client software can now be loaded on the web-server. When the user accesses the web-page, the client-applet is downloaded onto the local machine, and then executed.

The advantage of this design is that the user can always use the most updated version of the client software. Furthermore, the graphical-interactive client interfaces still execute locally. The product-model as well as all data are however stored persistently on the server.

Java applets are limited by the security restriction of the web-browser container. For example an applet-application cannot normally write to the local hard drive or display a menu. In SEWSAN AC the DXF file is for example only accessible from the remote server location (i.e. the base directory on the web-server).

4.12 Conclusion

This chapter covered the implementation of the AC design approach. Key aspects of the implementation included the design and implementation of the object identifier management, the object set management and the object relation management. These subsystems support the AC design and are required when a software system must manage a large number of objects across several models in one application. Special attention was given to object relation management. A practical comparison between the standard practice to include object references as attributes using a relation manager has shown that the new design can substantially reduce the complexity of source code. Model objects are introduced which replace the concept of isolated applications in the MC design. A special *Data model* object manages the product data of the application and ensures that application objects can be stored persistently. Non-application objects can also be included in the application-wide relation management. This is illustrated with the relationship between non-persistent CAD entities and basic engineering objects.

The engineering process as adapted for the AC design, as well as the functionality and background of algorithms are discussed in this chapter. It is important to show that the pilot implementation can represent a typical software system for civil engineers with its associated complexity. Generalized

topology functions such as data-tree traversals algorithms and typical geometric methods such as the calculation and updating of the length and slope of link elements are discussed. These functions are also applicable to other civil engineering software applications.

The pilot application has been extended to support a basic client/server scenario. It has been shown that a stand-alone application can be converted to a client/server application with a modern programming language such as Java. A system is presented where the application is delivered via the internet as Java applet and server-side processing and persistent object management is supported. This allows the user to always use the latest version of the application and offload numeric processing to a fast server. An implementation of multi-user access to the server and threaded processing on the server should form part of a design for distributed computing. However this would have exceeded the scope of this research.

The interaction between a Hydraulic model and a Visualization model in the AC design is substantially different from the AC design. The next chapter will focus on analysing the new structure, as well as quantifying the performance of the AC design.

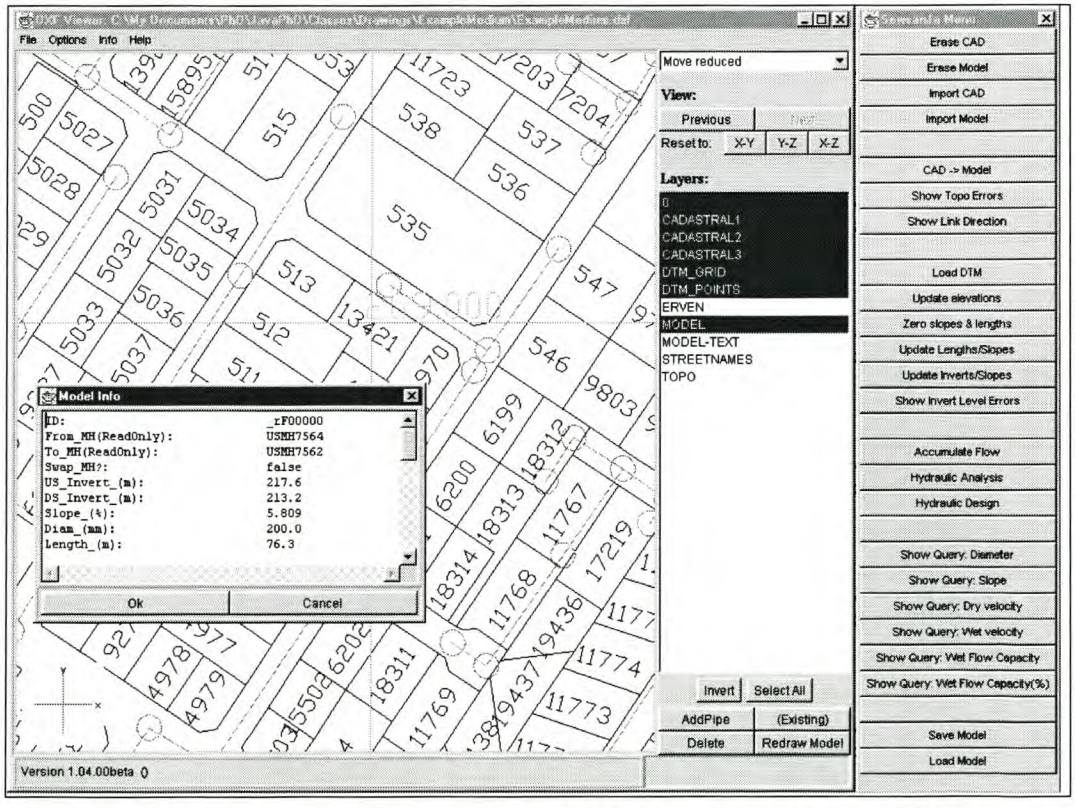


Figure 4.19: The user-interface of the *CAD model* with Project 2 loaded.

Chapter 5

Analysis of the AC design approach

5.1 Introduction

The pilot implementation as introduced in the previous chapter is applied to projects of realistic practical size. The performance of the pilot implementation is studied from two points of view: Analysis of structure and quantitative evaluation.

5.2 Analysis of the AC design structure

In this section an analysis of the structure of the AC design approach will show the characteristics of the design. In order to compare the structure of the design with the MC approach the interaction between the *Hydraulic model* and the *Visualization model* will again be chosen as typical example. A subset of the full implementation of the AC software system will be used to highlight the structural aspects.

The analysis evaluates the general aspects of the *AC* design which are also applicable to other hydraulic engineering software systems. A *Hydraulic model* for the hydraulic analysis and a *Visualization model* for the graphical presentation share the common identifier scope of the application.

The two engineering models are implemented in objects, namely the `HydraulicModel` and `VisualizationModel` object. Figure 5.1 shows the class diagram for the typical AC system. The full system is implemented

in Java and was discussed in Chapter 4.

Introduction of basic engineering objects: The AC design allows basic engineering objects to have an application-wide identifier scope. Different designs for the implementation of the objects can be considered. For example, the functionality associated with different models, such as the topology, geometry and hydraulic analysis can either be incorporated directly into the basic object, or included in several smaller view objects and then collected into one larger parent object.

The first approach was chosen in *SEWSAN AC* in order to reduce the total number of objects which must be maintained in the application. By introducing additional relations between a `TopoLink`, a `GeomLink` and a `HydroPipe` object, the formation of relationships between these objects at program runtime is supported. However, static design-time relations were chosen for the *SEWSAN AC*. Class inheritance is used to reduce the total number of attributes and methods in the objects and results in the `FinalPipe` and `FinalNode` classes to be instantiated only once.

Collection of object identifiers : In the AC design objects are identified by their unique application-wide identifier. Furthermore, any collection of objects does not store object references, but consistently only stores object identifiers. An example are the `nodes` and `pipes` attributes in the `DataModel`. Object references can be resolved at any time using methods from the `App` class. The creation of new application objects automatically manages the storage of identifier/reference pairs in the `objectMap` in the `App` class.

Function of the *Data model* : The *Data model* functions as central repository for sets of basic engineering objects, as well as for methods such as `saveModel()` and `loadModel()` which manage the persistent state. Common methods which are related to the topology or geometry and need to operate over sets of objects can be located in this model. An example is `updateLengths()` which is of use for both the *Hydraulic model* and the *Visualization model.*

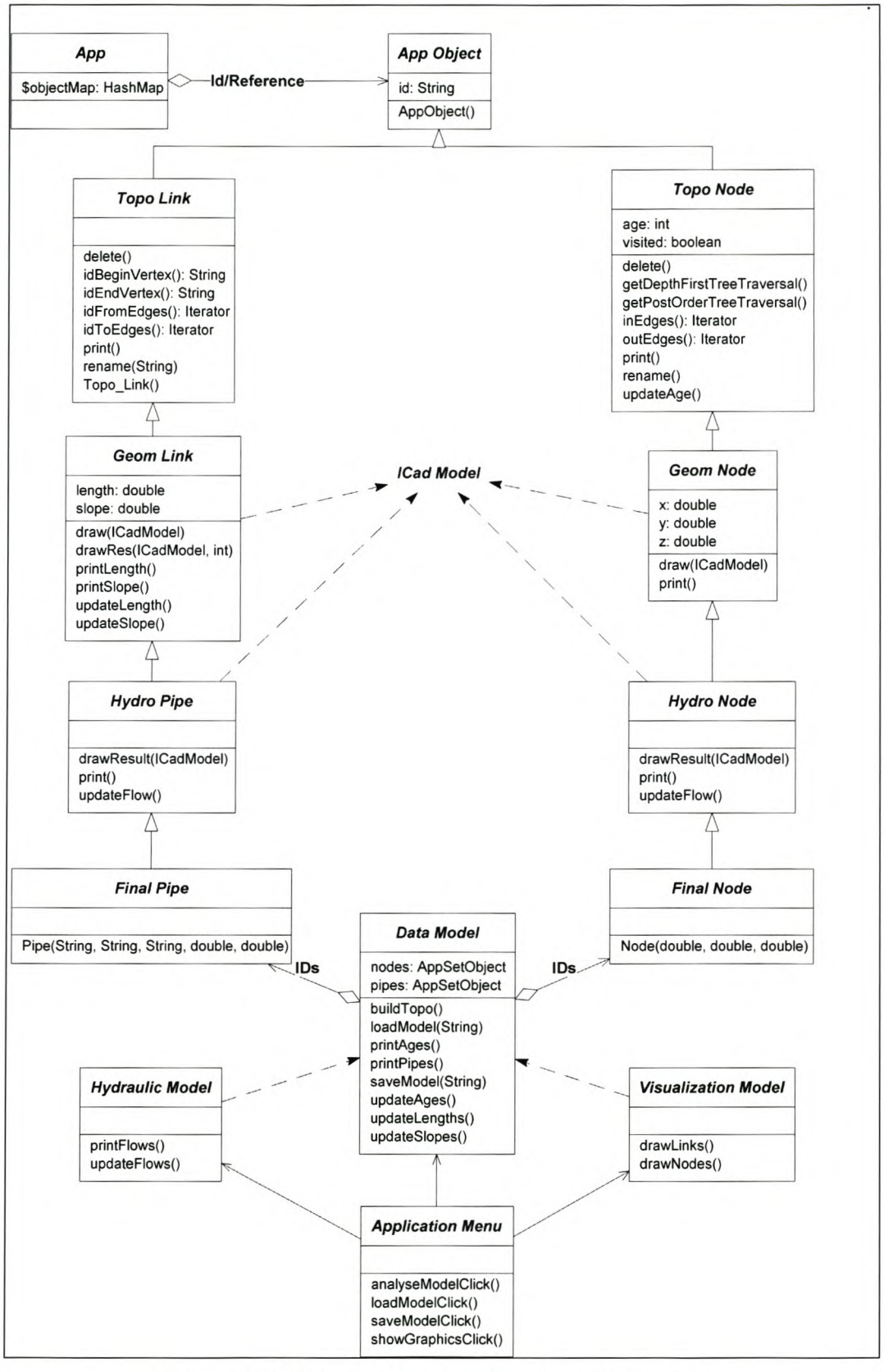


Figure 5.1: Class Diagram of typical AC system

5.2.1 Limitation of object name scope

The AC design approach includes the new application identifier manager (paragraph 3.2). This identifier manager ensures that for each object which is created under its management, a unique string identifier is used or created and stored together with the object reference in the central HashMap.

This identifier manager can be used from any point in the application to query or retrieve the object reference for a valid identifier. All application objects (objects which extend the `AppObject` directly or indirectly) are not bound to a specific model object. This allows the developer the freedom to access any object in the application from any point in the source code. Objects are therefore no longer bound to the name scope of a model-object.

A software model of the real-world is mapped to a model-object which contains or references sets of class `AppSetObject`. These sets collect the object-identifiers of the object to be related to this model. For example, the application set-object `DataModel.pipes` is used in the `drawModel()` method of the *Visualization model* to obtain an iterator over all the pipe identifiers, and then obtain an object reference from the identifier manager for each object identifier. The method `draw()` of the pipe objects is called with this reference.

A problem for the central management of object identifiers arises where objects are to be accessed which are not instantiated from an `AppObject`. These objects are not stored in the central HashMap, and their references can therefore not be located. It is the responsibility of the developer to ensure that all objects which are to be accessible throughout the application are instantiated from the `AppObject` or a descendant. However, objects which are not product-data can still be instantiated from other classes. For example, all objects in the *CAD model* are temporary and do not require persistent storage as part of the product-data. By using a special version of the relation-object these external objects can be used in directed mappings.

As discussed in Chapter 3, the application has been chosen as the object name scope which only supports one project at a time. The name scope can be extended to include more than one project. The identifier manager can then manage objects with the same identifier located in different projects.

A direct consequence of new application-wide name scope of objects is that software bridges are no longer required for a communication between models.

Model objects : Model objects now contain operations which are performed over all basic engineering objects in sets. For example the `updateFlows()` method can loop over the `pipes` object obtained from the `DataModel` in order to execute a method such as `Pipe.updateFlow()` on each object. Similarly the `drawLinks()` method in the `Visualization model` executes over all `pipes` and calls the `Pipe.draw(ICadModel)` method. Only the functionality which is relevant for the model is extracted from the object.

Interface to *CAD model* : In the AC design, the visualization model performs only high level commands on the basic objects. Within the basic object, the `draw()` method is then executed and requires interaction to a low level graphics system. By using well defined interfaces for communication in both directions, it can be ensured that the CAD subsystem is totally generic and independent, and does not form a stand-alone model as in the case of the MC design.

Topology and tree traversals : A separate topology model is often not required. The topology logic is stored in a layer of the basic engineering objects. This includes the creation of traversal iterators, such as the `getDepthFirstTreeTraversal()` method and associated methods such as `updateAge()`. Traversals are used extensively in *SEWSAN AC*.

Flow of command : In the AC design the flow of execution can typically start with the user action from a user interface, such as in an `ApplicationMenu` class. For example `loadModelClick()` is selected, which delegates action to the `DataModel`, to `loadModel()` and `updateLength()`. The last method is performed sequentially over all `pipe` objects, but `updateAges()` involves an in-order tree traversal. Execution then returns to the main menu, from where `updateFlows()` in the `HydraulicModel` can be called. This performs a post-order traversal over nodes and links to accumulate flows (in a sewer network).

Then `showGraphicsClick()` is selected, which performs `drawLinks()` and `drawNodes()`. The corresponding methods in the objects are called, and the interface to the CAD-model is used to display the information graphically.

The following issues found in the MC design are now addressed with reference to the typical AC example.

5.2.2 A priori implementation by the software developer

The use of interfaces in Java permits the development of isolated software components, which can be replaced by other implementations which adhere to the same interface specifications. This is illustrated by decoupling of the *CAD model* from the *Product-data model* with interfaces. It is possible to replace all classes of the *CAD model* with a commercial product which implements the interface *ICadModel*.

The design of the *Product-data model* and especially the hierarchical class structure introduced as an extension to the AC approach permits the addition of packages with new functionality directly above the `FinalClasses` package. An example could be the extension of the *Product-data model* to present results in tabular form. This change can be implemented even if the source code for the existing *Product-data model* is not available to the developer. A new model-object class, for example `TablePresentModel`, is added to the application. This class uses the *Product-data model* to traverse the set of model-objects which are to be presented in a table.

5.2.3 Object duplication and reduction

The traditional MC design approach includes classes (i.e. the definition of object functionality) as part of a model. Objects must therefore be duplicated when the same functionality is required in two models, since the two objects belong to separate classes.

Classes for an AC application are part of the computing platform. As one common platform is used for all classes (the Java platform), common functionality between the models is factored and stored in common classes on the platform.

The AC design consolidates all product-data in one centrally managed location, the *Product-data model*. This model manages the product-data defined by objects from the basic engineering objects classes as well as additional utility classes. The functionality for the *Product data* is defined in a hierarchical class structure, as common functionality is factored in the higher hierarchy. To reduce the number of objects to be managed in the application by the application manager, each basic engineering object is only instantiated once in the *Product-data model*, instead of once for each engineering model. This results in objects which are instances of the classes from the `FinalClasses` package. These objects contain the functionality required for all engineering

models in their class structure.

5.2.4 Program maintenance

The effort for program maintenance is defined by the size and complexity of the source code, and thus the number and size of the classes in the application. As class duplication can be reduced in the AC approach and classes can be consolidated, an AC application requires less maintenance than the equivalent MC application. The `updateLength()` method can now be located in one place, namely in the topology layer of the application, and is accessible from all models.

Debugging an AC application is simplified as all source code is located in one application.

5.2.5 Program extensibility

The introduction of the hierarchical class structure as well as the use of interfaces as discussed in Section 5.2.2 on an a priori implementation enhance the extensibility of the AC application.

As class duplication is reduced and classes are consolidated, an AC design can be extended more easily than the equivalent MC application.

5.2.6 Suitability for distributed computing

Name scope limitation : The name scope of an AC design can be extended to a computer network as the distributed computing environment manages remote object references and the marshalling of data over the network. This is the case with the implemented JAVA RMI.

Requirement for bridges : Software bridges are not required in the AC design approach. The transfer of large volumes of data between models located at different locations on the network is therefore avoided. Only the object references (either the persistent identifiers or if necessary, special remote object references) must be passed.

Object duplication : In a distributed computing scenario, classes must be installed on the platform at all locations where the object is instantiated. This

leads to duplication which can be avoided in a non-distributed scenario. Of more importance is the time and cost required to distribute classes to remote locations. However the distribution of classes takes place infrequently relative to normal data transfer, since classes are cached on the local computer and are updated only when out of date.

5.3 Quantitative analysis

The construction of the *Product-data model* from the *CAD drawing data* and the *Digital elevation data* is a typical benchmark operation and is suitable for the quantitative analysis as in the case of the MC design approach. Wherever the duration of an operation is noticeable long, Duration of Execution (DoE) was measured. Wherever user input requires considerable time, the User Interaction Count (UIC) is used to quantify the effort. For specific key operations, the Basic Instruction Count (BIC) is used as additional measure. The spreadsheet summarizing all quantitative results is found in Appendix H.

Duration of Execution (DoE) : All DoE measurements are measured in seconds, and are on the benchmark computer. The same computer is used as for the MC design approach, a Pentium II, 333 MHz machine with 160 Mb RAM running Microsoft Windows 98.

Basic Instruction Count (BIC) : The BIC has been evaluated for operations equivalent to the port operations required in the MC design approach: the process of building a *Product-data model* from the *CAD drawing data* and the update of the *Product-data model* with the *Digital elevation data*. These are the two most common time consuming data operations. Additionally, the visualization of the *Product-data model* is evaluated since it automatically is part of the MC evaluation. Only operations which are performed during each iteration of a loop are counted, as they contribute primarily to the execution time. Appendix G shows a spreadsheet where the instructions are counted for the AC design approach.

User Interaction Count (UIC) and Persistent Data Size (PDS) : The same assumptions as for the MC design are made. See Section 2.9.

Next the basic operations are described and categorized. Then the results

for the quantitative analysis are reported for each category over the four test projects.

5.3.1 Evaluation of BIC and UIC

This section describes the basic operations used for the evaluation of the BIC in the AC design. As for the MC design, it is assumed that a layout drawing file of the sewer system exists which contains lines representing the sewer pipes, as well as a circle indicating the outfall manhole. The entities in the drawing are located geographically correct. The *Digital elevation data* of the area is also available. The steps of the evaluation procedure are outlined below, with differences to the MC design (Section 2.9.1) outlined:

- **Topology flow direction (TFD) :** The same assumptions as for the MC design are made and a similar evaluation of *Show direction definition* TFD1 and *Swap direction definition* TFD2 is performed.
- **Build Model (BM) :** This step combines the steps for the building of the hydraulic model, which where considered individually for the MC design. Unique names are automatically generated for manholes. Manholes are defined at the endpoints of lines representing sewer pipes. Default values are assigned for pipe diameter, number of parcels per pipe, manhole elevation and invert levels. This step combines the MC codes *TME2* and *HMI*.
- **Draw Model (DM) :** During this stage the *Product-data model* is visualized. The model display contains standard sized text for all attributes as well as default values. This step combines the MC codes *HME* and *TMI2*.
- **Topology model correction (TMC) :** Errors in the topology which were missed during the first iteration, are corrected interactively. See Section 2.9.1.
- **Update Elevations (UE) :** The coordinates of the manholes of the sewer system are passed to the interpolation algorithm and the elevations are updated in the *Product-data model*. Only the transfer of data is evaluated as in the MC evaluation. All operations are also quantified using the *BIC*. This step combines the MC codes *TCE*, *ECI*, *EEI* and *TEI*.
- **Topology slope update (TSU) :** The initial slope of the pipes is automatically updated from the elevations at the endpoint manholes.

- **Topology parcel update (TPU) :** During this interactive operation the number of parcels associated with each pipe are counted and the operation is evaluated with the UIC. As for the MC design, it is assumed that 20% of the parcel count is different from the default (1).
- **Hydraulic Analysis and Design (HAD) :** The flow is accumulated, followed by an initial analysis. The optimum diameters are designed using the slopes now available for each pipe. These operations do not form part of the comparison between the MC design and the AC design since different algorithms are implemented. They are evaluated in order to compare the performance of AC implementation for stand-alone, single PC client/server and dual PC client/server operation.

5.3.2 Result of the quantitative analysis

In order to compare results against the three key operations from the MC system, the Update Elevations (UE), Build Model (BM) and Draw Model (DM) operations have been evaluated.

Duration of Execution and Basic Instruction Count : The performance for the key operations (UE, BM and DM) for execution on the stand-alone PC is shown in Table 5.1 and shown in graphical form in Figure 5.2.

Table 5.1: Comparison of key AC operations for DoE and BIC

Project	**TP1**	**TP2**	**TP3**	**TP4**
Number of pipes	**104**	**682**	**2934**	**6721**
Update Elevations (UE)				
(1)	0.10	0.15	0.4	1.0
(2)	0.04	0.26	1.1	2.5
(3)	0.01	0.05	0.2	0.5
Build Hydraulic Model (BM)				
(1)	0.8	18.7	156.3	1267
(2)	3.1	75.8	1253.0	6442
(3)	0.6	15.2	250.6	1288
Draw Hydraulic Model (DM)				
(1)	0.8	2.5	5.2	10.3
(2)	1.0	6.4	27.5	62.9
(3)	0.2	1.3	5.5	12.6

Note: (1) Duration of Execution (DoE) in s
(2) Basic Instruction Count (BIC) in million instructions
(3) Equivalent duration for Basic Instruction Count (BIC) in s

The following quantitative results are observed in the table: One second of equivalent BIC is approximately equal to 5.0E6 BIC operations.

DoE and BIC comparison : The equivalent BIC for the *Update Elevations (UE)* operation does not correlate with the single PC DoE. This is partly due to the difficulty in measuring the short DoE duration of the UE operation which is under 1 s in duration. It seems that a near constant overhead duration is not accounted for in the BIC.

The equivalent BIC for the *Build Hydraulic Model (BM)* correlates closely with the DoE. The BIC for the *Draw Hydraulic Model (DM)* also accurately represents the DoE values.

The BIC has again been verified as an absolute measure of execution performance since it correlates well with measured DoE values. It will be used later in this section for the comparison between internal and external relationships, and in Chapter 6 to compare results from the MC design with the AC design.

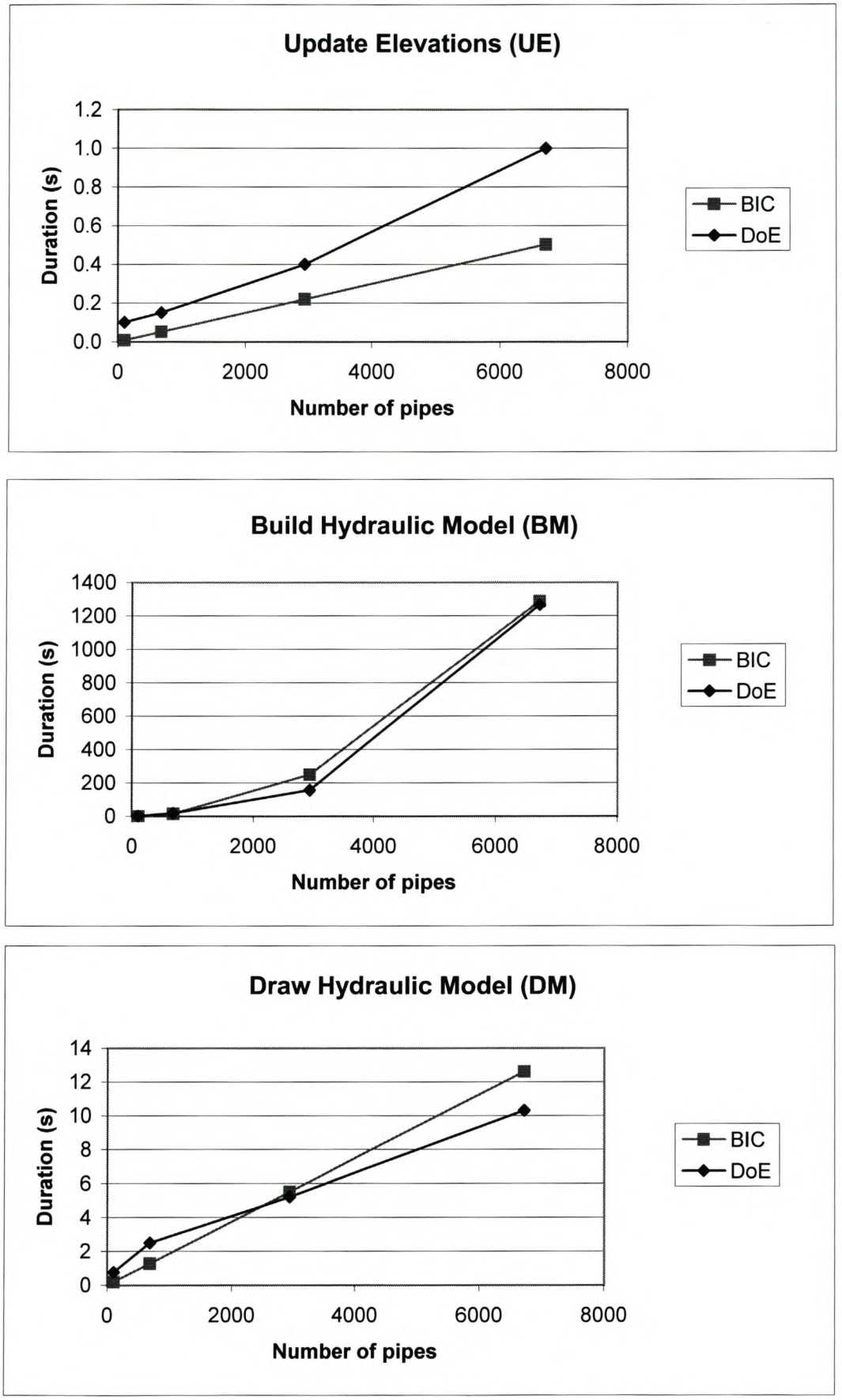


Figure 5.2: Graph of key AC operations for DoE and BIC

User Instruction Count : The performance for the operations (TFD2, TMC and TPU) with respect to the UIC is summarized in Table 5.2. Figure 5.3 plots the UIC against the number of pipes. It shows the interaction count for the three measured cases cumulatively, namely from bottom to top for *Swap Direction Definition* (TFD2), *Topology Manhole Correction* (TMC) and *Topology Parcel Update* (TPU). This data is used in the comparison between the AC and DC design in Chapter 6.

Table 5.2: Key AC operations for UIC

Project **Number of pipes**	**TP1** **104**	**TP2** **682**	**TP3** **2934**	**TP4** **6721**
User Interaction Count (UIC)				
Swap Direction Definition (TFD2)	104	682	2934	6721
Topology Model Correction (TMC)	94	614	2641	6049
Topology Parcel Update (TPU)	83	546	2347	5376
Total	281	1842	7922	18146

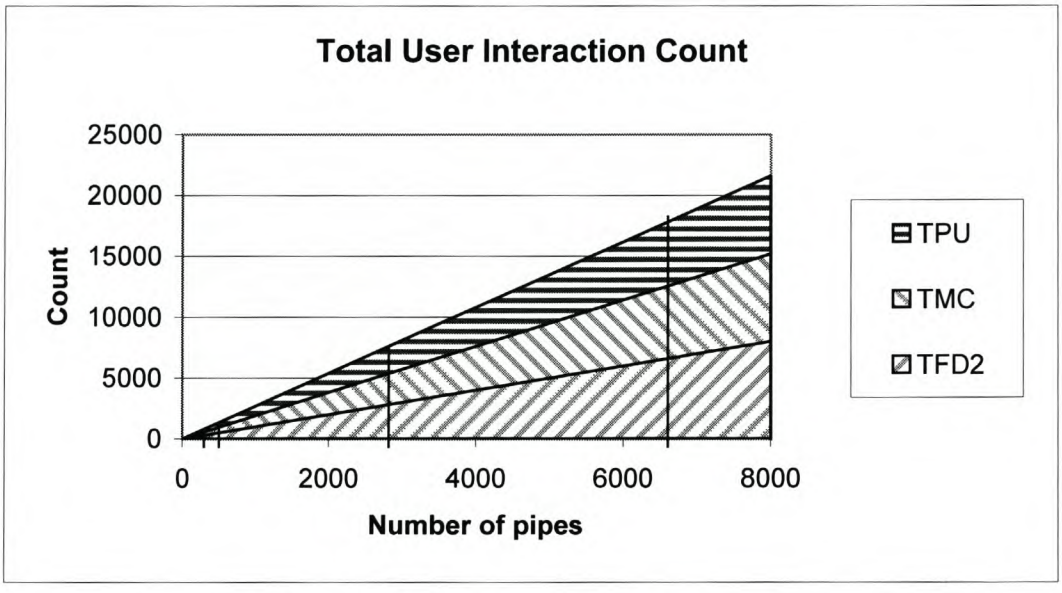


Figure 5.3: Graph of UIC performance

Internal vs External relationships : Two different implementations of the AC design are evaluated using the BIC. Relationships between objects are stored as persistent identifiers within the objects or an external relation manager system is used. Table 5.3 compares the BIC for the time-consuming *Build Hydraulic Model* operation and the *Draw Hydraulic Model* operation. Figure 5.4 shows the comparison in graphical form. As the *Update Elevations* operation does not make use of relationships between pipes, manholes or erven objects, it is not included in the comparison.

Table 5.3: Analysis of BIC for internal vs external relationships

Project **Number of pipes**	**TP1** **104**	**TP2** **682**	**TP3** **2934**	**TP4** **6721**
Build Hydraulic Model (BM)				
External (count x1E6)	3.12	75.8	1253	6442
Internal (count x1E6)	2.47	71.5	1235	6400
Ratio: External / Internal	1.26	1.06	1.01	1.01
Draw Hydraulic Model (DM)				
External (count x1E6)	0.97	6.39	27.5	63.0
Internal (count x1E6)	0.83	5.45	23.5	53.7
Ratio: External / Internal	1.17	1.17	1.17	1.17

It can be seen that the external relation manager system results in almost no performance overhead for the *Build Hydraulic Model* operation. The *Draw Hydraulic Model* exhibits only a 17% performance overhead when storing relationships external to objects. The marginal cost for using the relation manager must be weighed against the improved design structure as discussed in Section 5.2.

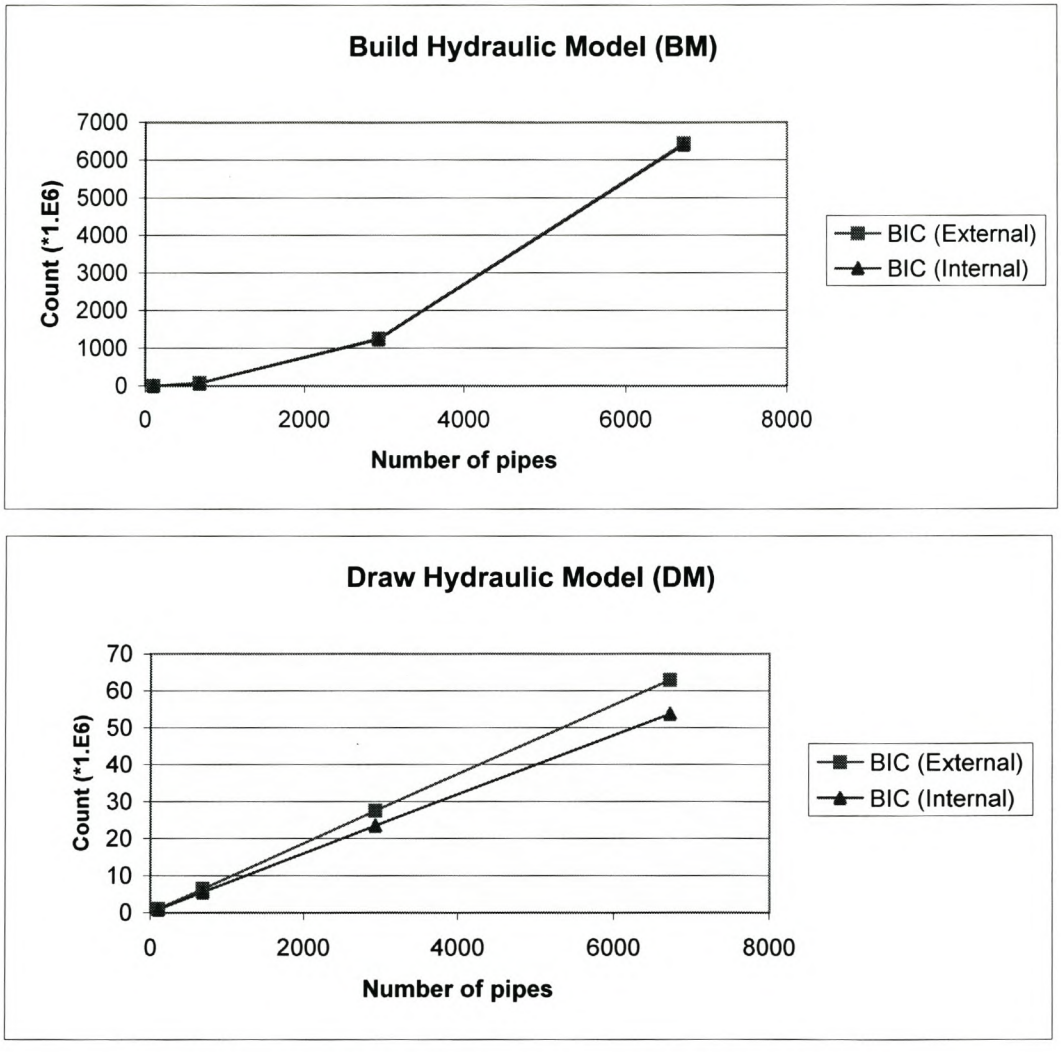


Figure 5.4: BIC for Internal vs External relationships

Size of persistent data : The file size of the *SDFJ* file, which contains all persistent objects of the AC system, is presented in Table 5.4 for the different test projects. File sizes are presented in Mbyte for the internal and external storage of relationships.

Table 5.4: Analysis of PDS for internal vs external relationships

Project **Number of pipes**	**TP1** **104**	**TP2** **682**	**TP3** **2934**	**TP4** **6721**
Key file: **SDFJ File**				
External (size in Mb)	0.16	0.96	4.06	9.73
Internal (size in Mb)	0.11	0.61	2.51	6.19
Ratio: External / Internal	1.48	1.58	1.62	1.57

The graph presented in Figure 5.5 shows a near-linear relationship between file size and number of pipes, since a fixed number of basic engineering objects can be associated with each pipe in the system.

The external storage of relationships requires 50 to 60% more persistent disk space than the internal storage. This is due to overhead required to store the relation objects.

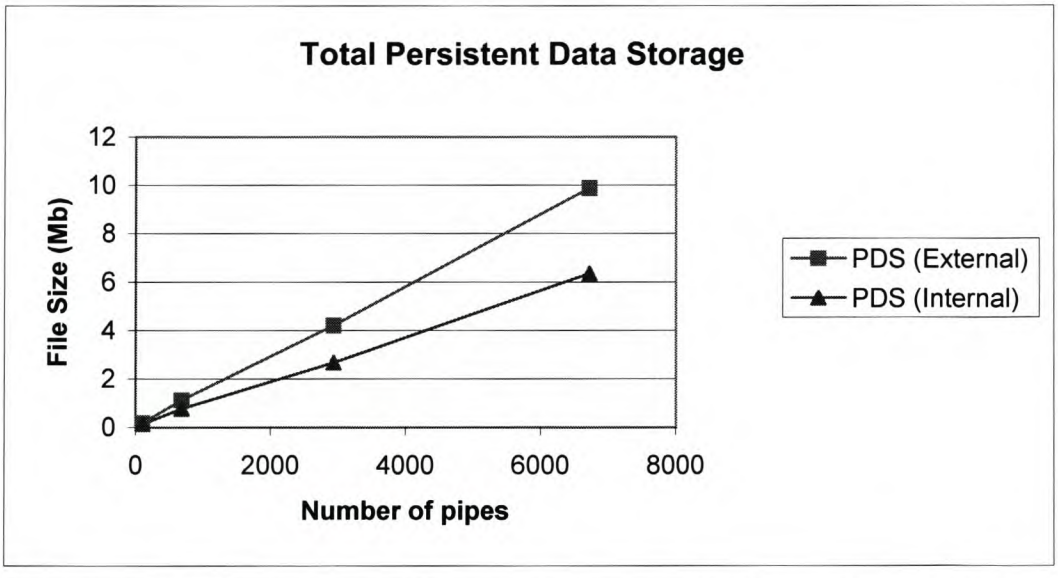


Figure 5.5: Comparison of internal vs external relationships of PDS

Distributed scenario : Two distributed scenario cases, namely a single PC client/server setup, as well as a two PC client/server configuration are investigated. The server PC has an Intel Pentium II 633 MHz processor, and

is thus roughly twice as fast as the client PC (the Intel Pentium II 333 MHz based computer) used for all previous analyses.

Three operations (Topology Slope Update (TSU), Draw Hydraulic Model (DM) and Hydraulic Analysis and Design (HAD) are selected to illustrate the effect of distributed computing on the performance.

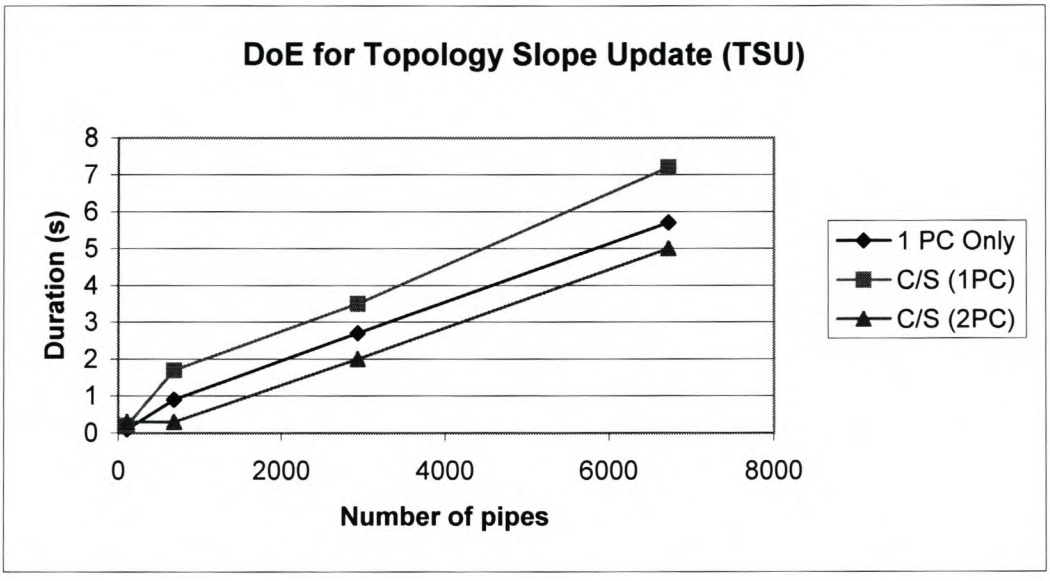
The results are presented in Table 5.5 and shown in graphical form in Figure 5.6.

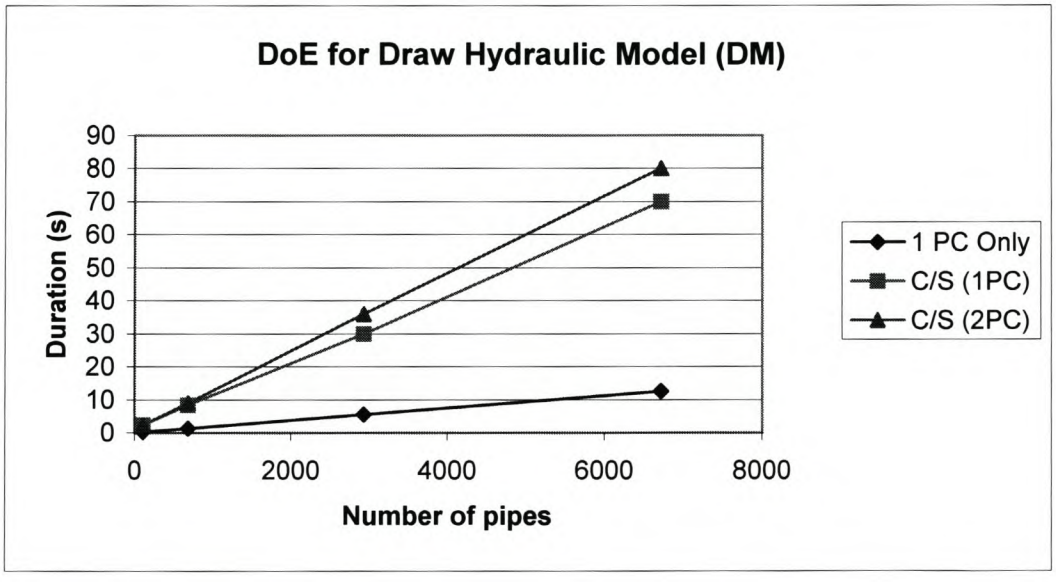
Table 5.5: Comparison of key AC operations for DoE in distributed scenario

Project **Number of pipes**	**TP1** **104**	**TP2** **682**	**TP3** **2934**	**TP4** **6721**
Topology Slope Update (TSU)				
(1)	0.1	0.9	2.7	5.7
(2)	0.2	1.7	3.5	7.2
(3)	0.3	0.3	2.0	5.0
Draw Hydraulic Model (DM)				
(1)	0.8	2.5	5.2	10.3
(2)	2.3	8.4	30.0	70.0
(3)	2.3	9.0	36.0	80.0
Hydraulic Analysis and Design (HAD)				
(1)	1.3	6.1	22.9	47.5
(2)	1.8	7.6	27.1	48.7
(3)	1.0	3.8	12.8	26.6

Note: (1) Duration for 1 PC in s
(2) Duration for C/S (1 PC) in s
(3) Duration for C/S (2 PCs) in s

In all cases the single PC client/server setup performs worse than the stand-alone PC scenario. This is due to the overhead of the Java RMI networking subsystem. For the two PC client/server configuration the performance depends on the operation: For the TSU operation, the distributed operation is marginally better than a stand-alone operation. For the DM operation, both distributed scenarios result in a significant overhead. This is due to the individual handling of line entity data for each pipe object by the communication system. On the other hand, when operations can be performed primarily on the server side (and especially if the server PC is faster than the client PC), significant time savings can be achieved. This is illustrated for the HAD operation. The halving in execution time correlates with the ratio of server to the client CPU speeds.





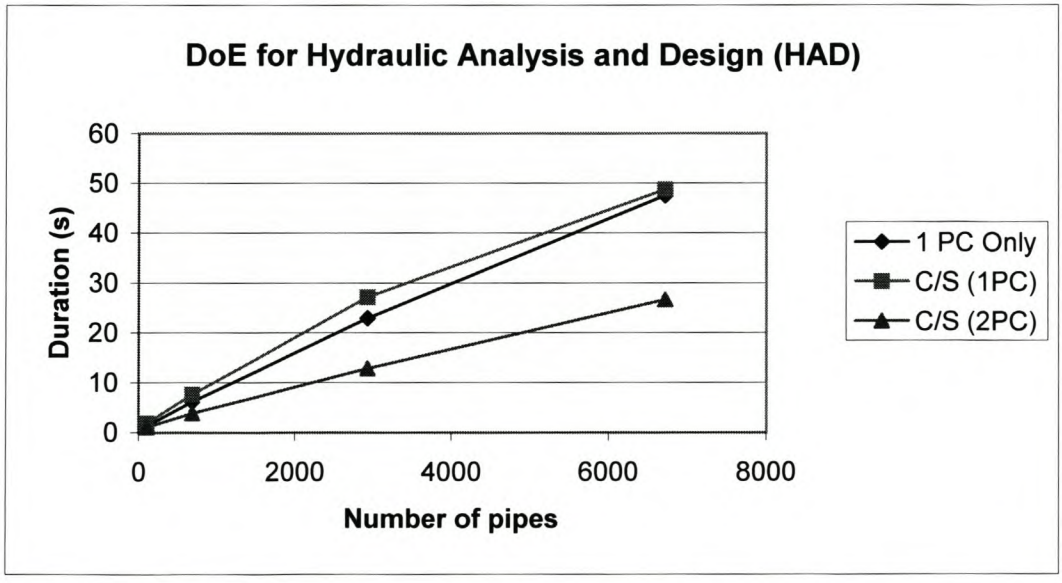


Figure 5.6: Graph of distributed AC performance

5.4 Conclusion

In this chapter an analysis of the design structure and a quantitative analysis of the AC design have been conducted.

The structure of the AC design has been analysed using a part of the fully implemented AC pilot system. This allows elements of a typical AC to be highlighted in the analysis such as the management of object identifiers and the storage of topology. Only two models, a *Hydraulic model* and a *Visualization model*, together with a *Data model* were selected to clarify the discussion.

The analysis of structure has shown that the AC design is superior to the MC design. This will be emphasized in Chapter 6.

The quantitative analysis confirmed that the BIC is also validated for the AC design. The results for the key operations are summarized for later comparison with the MC design for the four different quantitative criteria, namely BIC, DoE, UIC and PDS.

A comparison between the use of internal and external relationships between objects has shown that up to 17% more time is spend to execute code using the external relation manager, and an increase in persistent data size of up to 60% is recorded. The cost for using the relation manager must be weighed against the improved design structure.

The distributed scenario was also analysed quantitatively. The performance of operations varies with the type of operation. The performance increase due to the additional (and faster) processor is weighed against the time cost of transferring data across the network.

The analysis of the AC design approach has now been concluded. In the next chapter the final comparison between the MC design and the AC design will be presented.

Chapter 6

Comparison of designs and conclusions

6.1 Introduction

A comparison of the MC design with the AC design shows the impact of the two approaches on a practical engineering problem. It highlights the structural and quantitative shortcomings in the MC design and shows that the AC design approach resolves or reduces the problems. The criteria for the comparison of complexity follow from the previous introductory discussion for civil engineering software: dynamics of engineering data, involvement of different professionals, volume of data, support for teamwork, documentation requirements, allowance for modification and technical advances.

6.2 Comparison of the design structure

The same qualitative criteria have been used to evaluate the MC design and the AC design in Chapters 2 and 4. The following conclusions can be drawn with respect to the structure of the AC approach, which demonstrates the advantages of the design. A detailed analysis was performed in Section 5.2.

Limitation of object name scope : This primary limitation of the MC design is removed in the AC design. All application-objects are readily accessible using the identifier manager for the application. Software models of the

real-world are created as special model-objects which contain application set-objects. Non-application objects can also be handled using special relation-object classes. Software bridges between models are no longer required.

A priori implementation by the software developer : The use of interface technology as well as class inheritance introduces flexibility to the design of the AC system. The AC implementation shows that model-objects are readily formed by the collection of basic objects into application-sets and that relations of objects can be formed dynamically using relation-objects and relation-sets.

Using these new concepts, an application can be developed which allows easy user-interaction to the manipulation of the basic objects of the application. For example the addition, renaming and deletion of topology objects of typical engineering software systems can now be managed in a consistent and simplified way.

Object duplication, program maintenance and extensibility : The traditional MC design treats classes as part of a model. This results in duplication of functionality. Classes for an AC application are part of the computing platform so that class duplication is reduced, as common functionality of models can be factored and stored in common classes on the platform.

Using hierarchical class structures, engineering functionality can be structured efficiently into classes to ensure optimum storage and minimum program maintenance. The extensibility of the design can further be improved by using interfaces.

Suitability for distributed computing: The name scope of an AC design approach is not limited to a model. Applications can be extended to computer networks provided a distributed computing environment is available. As data can be located centrally, and in most cases only the references need to be transferred, the AC approach is well suited for the distributed scenario.

The pilot implementation also shows performance advantage of a distributed civil engineering application when numeric processing can be uploaded to a fast server.

6.3 Quantitative comparison

6.3.1 Duration of Execution (DoE) comparison

Figure 6.1 shows a quantitative comparison of three key operations (Update Elevations (UE), Build Hydraulic Model (BM) and Draw Hydraulic Model (DM)) for the MC approach and the AC approach.

Table 6.1: Comparison of key AC and MC operations for DoE

Project	**TP1**	**TP2**	**TP3**	**TP4**
Number of pipes	**104**	**682**	**2934**	**6721**
Update Elevations (UE)				
(1)	2.0	47.0	350.4	2608.4
(2)	0.1	0.2	0.4	1.0
Build Hydraulic Model (BM)				
(1)	9.8	191.8	1717.9	10290.8
(2)	0.8	18.7	156.3	1267.1
Draw Hydraulic Model (DM)				
(1)	9.0	63.3	222.8	951.0
(2)	0.8	2.5	5.2	10.3

Note: (1) MC duration of execution (DoE) in s
(2) AC duration of execution (DoE) in s

Evaluation : The duration of execution for the AC design is significantly less than the duration for the MC design. The DoE for the MC design and especially the BM operation has been explained in Section 2.9.2.

The large difference in the performance of the UE and DM operations can be explained by the requirement of the MC system to export and import data files. The two operations each require to traverse two ports and a data file bridge. This involves a total of four porting operations for each basic operation. In the AC system the requirement for the ports as well as the bridges falls away. For the BM operation in the MC design, the port on the *CAD model* is responsible for the poor performance.

In Section 6.3.4 the effect of this port will be excluded from the comparison.

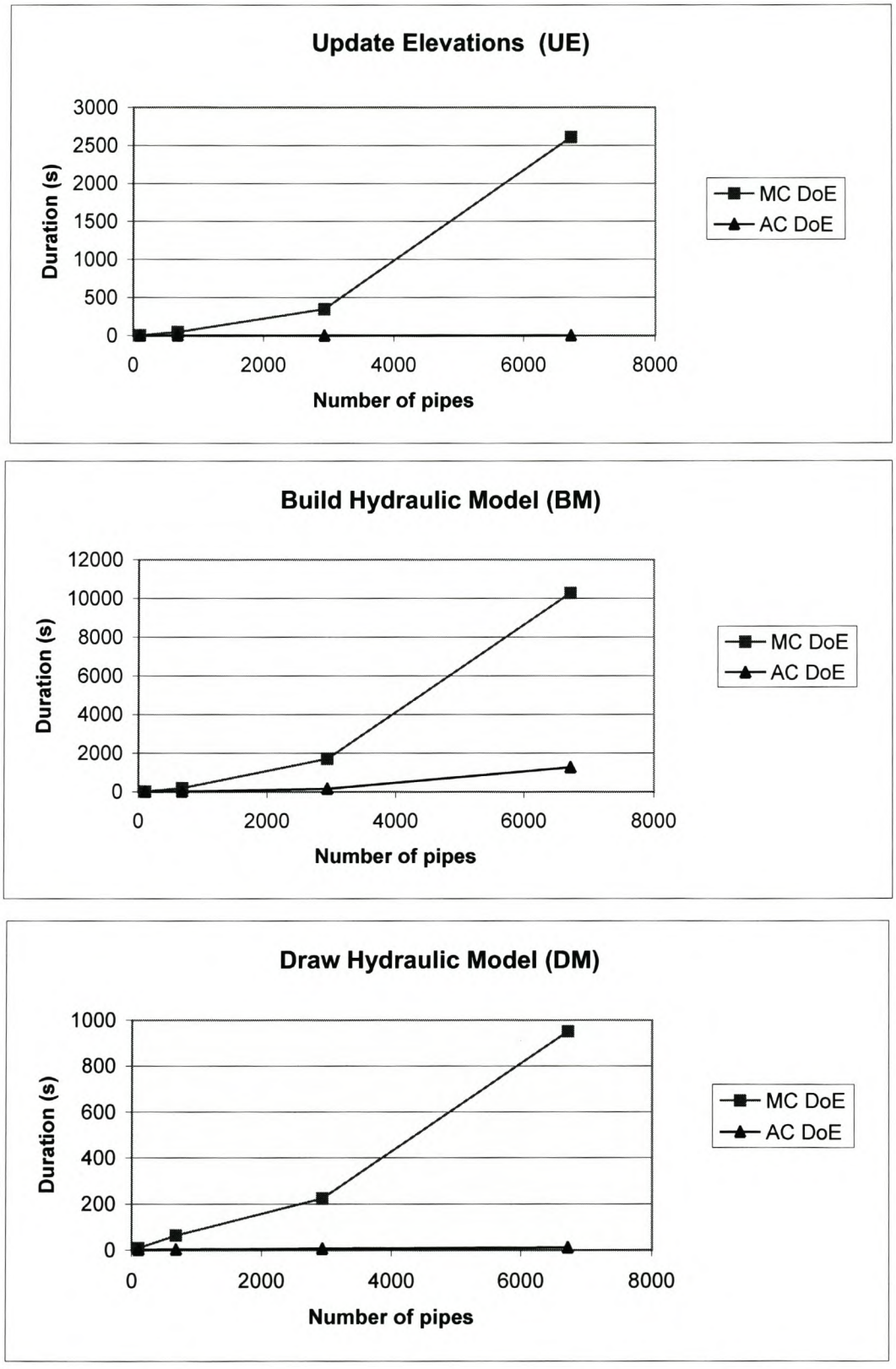


Figure 6.1: Comparison of key operations for MC and AC systems

6.3.2 User Instruction Count (UIC) comparison

The total effort due to UIC is compared for the MC system and for the AC system with different number of pipes in Table 6.2 and Figure 6.2.

Table 6.2: Comparison of total AC and MC operations for UIC

Project **Number of pipes**	**TP1** **104**	**TP2** **682**	**TP3** **2934**	**TP4** **6721**
Total count for MC	292	1909	8215	18817
Total count for AC	281	1842	7922	18146

Evaluation : The effort associated with user-interaction as measured with the UIC in the AC system is practically the same as for the MC system. This indicates that the new AC system can provide the same degree of user-friendliness as the well established MC system.

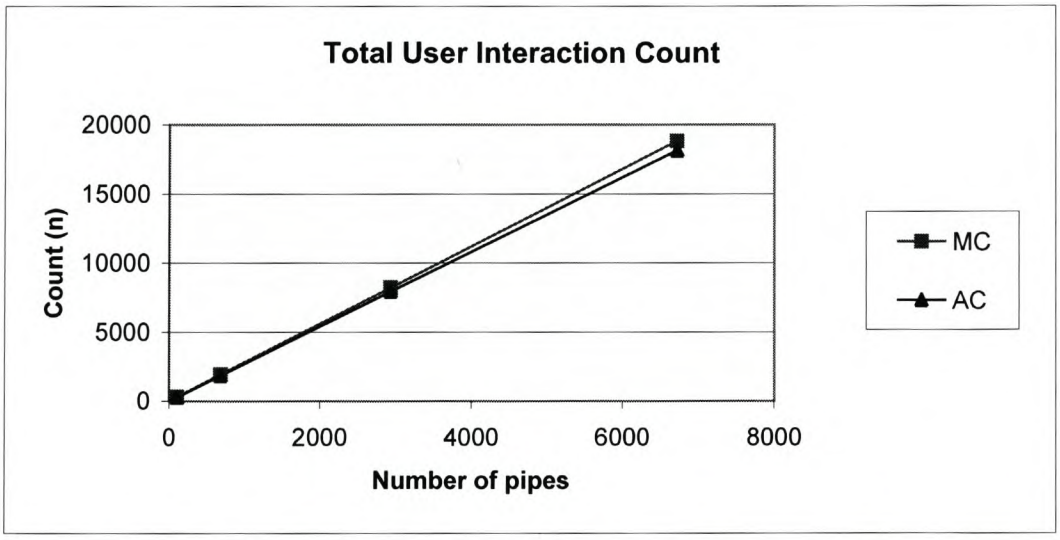


Figure 6.2: Comparison of total UIC between MC and AC systems

6.3.3 Persistent data size (PDS)

The file size of persistent data of the MC system and the AC system are compared as function of the number of pipes in Table 6.3 and in Figure 6.3.

Table 6.3: Comparison of total persistent data size for MC and AC systems

Project **Number of pipes**	**TP1** **104**	**TP2** **682**	**TP3** **2934**	**TP4** **6721**
Total file size for MC (Mbyte)	0.193	1.331	4.898	10.563
Total file size for AC (Mbyte)	0.180	1.117	4.217	9.885

Evaluation : The files size of the AC system is less than the file size for the MC system. Through the use of file compression techniques, the size can be reduced further. This indicates that the additional requirement for storing relation objects and set objects in the AC design does not necessarily lead to a larger overall persistent data size requirement.

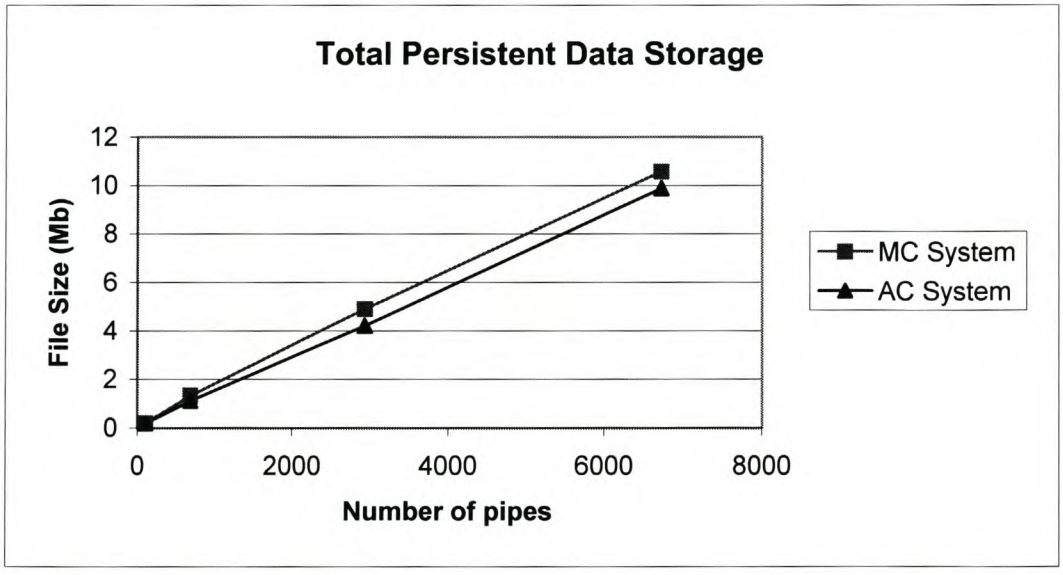


Figure 6.3: Variation of file size with the number of pipes for MC and AC systems

6.3.4 Modified BIC

Several factors contribute to the difference in performance between the MC system and the AC system. One key factor is the slow communication between models over bridges, for instance between the *CAD model* and the *Visualization model* in the MC design.

The BIC technique is used to evaluate the hypothetical scenario of a MC design where no time is lost in the communication over software bridges or where an MC design consists only of one model with no software bridges. This is achieved by replacing all *far* calls with *near* calls for the evaluated key methods. Equal platforms and environments are assumed.

Table 6.4 shows the results for key operations for the test projects, and Figure 6.4 show a graphical presentation of the variation in BIC with the number of pipes for the MC system and the AC system.

Table 6.4: Comparison of key AC and MC operations for modified BIC

Project **Number of pipes**	**TP1** **104**	**TP2** **682**	**TP3** **2934**	**TP4** **6721**
Update Elevations (UE)				
(1)	1.2	14.3	171.6	818.1
(2)	0.0	0.3	1.1	2.5
Build Hydraulic Model (BM)				
(1)	11.1	112.6	1146.0	5175.5
(2)	3.1	75.6	1252.0	6440.4
Draw Hydraulic Model (DM)				
(1)	7.8	54.6	294.4	903.5
(2)	1.0	6.4	27.3	62.6

Note: (1) MC Modified Basic Instruction Count (BIC) (x1E6)
(2) AC Modified Basic Instruction Count (BIC) (x1E6)

Evaluation : For the UE and the DM operation the modified BIC evaluation shows the clear advantage of the AC design. The requirement to traverse the data bridge (and its two ports) for each of the two directions of the basic operations falls away, and results in a significant time saving.

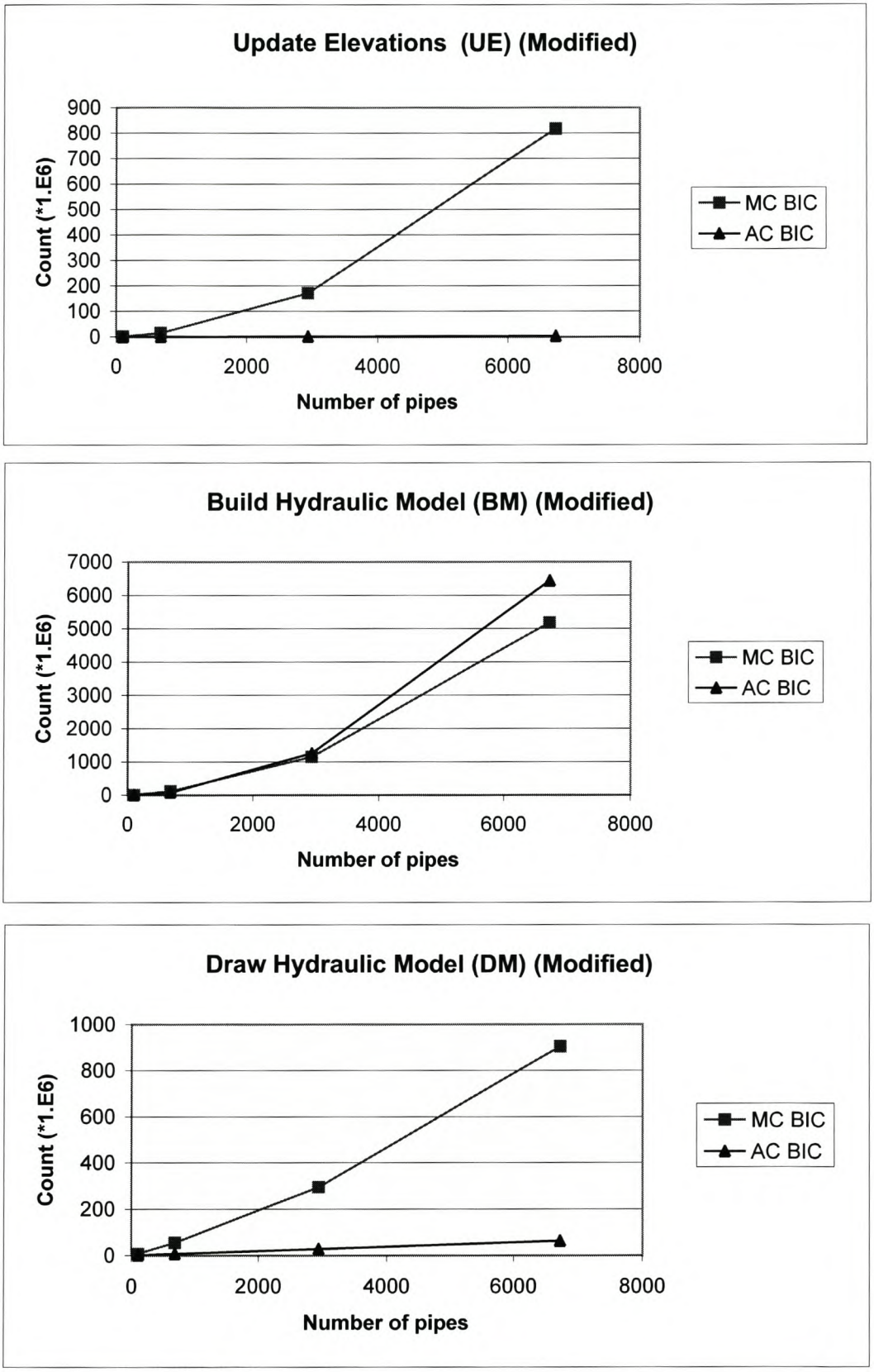


Figure 6.4: Comparison of modified BIC for the MC and AC system

However, the BM operation performs a maximum of 25% poorer for the AC implementation than for the MC implementation, with respect to the modified BIC. Here in the hypothetical case where no time is lost due to the communication with the CAD system, the MC system performs better because the execution overhead contributed by the application-, set- and relation-managers are not present.

It is thus clear that there is room for improvement in the implementation of the AC design. However, the potential loss in execution performance must be set against the improvement in the design structure introduced by the AC approach.

6.4 Comparison of complexity

An AC system is compared with a MC system with respect to civil engineering complexity criteria.

Dynamics : The AC design is suitable for applications with dynamic behaviour. The data structures defined by classes are flexible so that they can be extended by later developers. The graphical and dynamic formation of relations and the construction of sets of basic objects are supported by the structure of the AC system.

The AC design ensures that the software for an application remains dynamic.

Specialization : Specialist professionals influence the design of an engineering package during its development. The incorporation of captured data from professional CAD technicians and digital terrain data from surveying specialists is evaluated for both designs as part of the input to the system. On the output side, the town-planning specialist requires information on the updated sewer model in the form of a geographical information system. The access of this system via the Internet is evaluated for both designs.

The knowledge of professionals is incorporated into the AC design with a consistent and persistent object management system.

Volume : The efficient management of large data sets, the reduction in duplication, the transfer of data only when required to reflect changes, as well as the assurance of consistency between different data sets are evaluated.

The management of data between the *Topology / Data model*, the *Elevation model*, the *Hydraulic model*, the *Visualization model* and the *Geographical / Presentation model* is evaluated for both designs.

By evaluating the AC system also for large data sets, the effect of data volume on the suitability of the design is taken into account. It is shown that the AC system can handle large data sets successfully.

Teamwork : The modern software platform must support teamwork. In the MC design team work is hampered by the limitation of object name scope. An object is only uniquely identifiable within a model in the application of one team member and is not valid in the name scope of the application of another team member. Complex software bridges must therefore exchange data between applications of team members.

In the AC design the name scope of objects can be extended beyond the model space to be valid in the applications of all team members. This facilitates teamwork.

Documentation : The transfer of large CAD files is the standard form of data exchange for the MC system. For AC systems a central data storage is introduced: The data model is stored persistently in object form. This classical documentation (a CAD drawing or paper plan) can be derived from this model at any time. This is illustrated in the AC system where a drawing of the hydraulic model or a geographical report can be generated at any time.

Modification : Frequent small modifications in the engineering design must be managed within the application. The degree to which data can be consistently updated between the *Topology / Data model*, the *Elevation model*, the *Hydraulic model*, the *Visualization model* and the *Geographical / Present model* was investigated for both designs. The AC design permits consistent and persistent management of the propagation of changes, for example in the topology using the application, relation and set-managers.

6.5 Summary and Conclusions

In this dissertation the concept of a model as implemented in engineering software is considered in depth. It is shown that traditional models have been implemented in the model-centred (MC) design approach as stand-alone software components, and have resulted in complex unmanageable applications.

One such MC application is considered in depth and evaluated according to structural and quantitative criteria. A special quantitative criterion, the Basic Instruction Count (BIC) is introduced in addition to the Duration of Execution (DoE) in order to perform measurements which are independent of the computing platform. The disadvantages of the MC design are discussed in detail, especially the limitation of object name scope and resulting requirement for ports between models. The limitations are aggravated in a distributed computing scenario.

An alternative design, the application-centred (AC) design approach is introduced in which not the engineering model, but the basic engineering objects play the central role. Models are now formed by the collection of special application-objects which are created by an application-manager. Mechanisms such as set- and relation-managers are developed to support the static and dynamic structure between objects and to ensure the consistent state of objects in the application. The persistent storage of application-objects is considered, as well as the handling of non-application objects.

This AC design is then implemented as a fully functional pilot implementation written in the Java programming language. The system is also extended to a two-tier distributed computing scenario. The application is tested and evaluated with respect to structural and quantitative criteria with real-world examples of typical dimension. The same structural and quantitative criteria as used for the MC system are applied to measure the performance of the AC system. In addition a comparison between two different implementations for the management of object relationships are evaluated quantitatively. It can be seen that almost no performance penalty is paid for the external management of object relations.

A comparison between the two designs shows that the AC design is superior to the MC design with respect to structural, quantitative and engineering complexity criteria. The BIC has proven to be a good measure to evaluate the performance of implemented algorithms in an application and platform independent scenario and allows the comparison of source code written in

different programming languages between the MC application and the AC application. The criterion shows that in a hypothetical scenario, where the negative effect of port operations in the MC design is ignored, for example when only one model and no software bridges exist, the execution overhead as a result of the application-, set- and relation-managers in the AC design can result in a marginal poorer relative quantitative performance. However the advantages of the design structure outweighs this potential disadvantage.

6.6 Recommendations

Additional work can be done to improve the AC approach. The introduction of the concepts of application, set and relation management at a lower level of programming can be considered. This would make these functions almost an extension of the Java programming language.

The AC system can also be optimized for speed. Although consideration was given to the optimal use of data structures, more efficient algorithms for tree traversal and topological searches can improve the performance of the application. Identifiers have been used throughout the application. This requires lookup of object references in a hash table. Although the performance penalty for the DoE criterion is acceptable compared to the MC design, an optimization of source code using advanced caching algorithms could improve overall performance.

The distributed scenario has only been implemented using the Java RMI methods for a basic pilot application. The extension of the AC design to a full distributed environment, which includes user access management for multiple users and threaded execution of multiple server processes, could be considered.

Bibliography

[1] D. Adair, J. Ball, and M. Pawlan. *The Java Tutorial Continued*, chapter Trial: 2D Graphics. Sun Microsystems, http://www.java.sun.com/-docs/books/tutorial/2d/index.html, 2000.

[2] ATV. *Planung der Kanalisation Abwassertechnische Vereinigung.* Ernst und Sohn, 4th edition, 1994.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[4] Borland. *Borland Delphi 5.0.* http://www.borland.com, 1999.

[5] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, Prentice Hall, Englewood, New Jersey, 1990.

[6] Durban Corporation. *Survey Handbook.* Institute of topographical and engineering surveyors of South Africa, P.O. Box 699, Durban, 4000, South Africa, 5th edition, 1987.

[7] A. Cremers. Definition und Implementierung eines Bauwerkmodelkerns. In *Deutsche Forschungsgemeinschaft Abschlussbericht: Objektorientierte Modellierung in Planung und Konstruktion*, chapter 1. Wiley, 2000.

[8] CSIR. *Guidelines for the provision of engineering services and amenities in residential township development, Chapter 9: Sanitation.* CSIR, Department of Housing, Pretoria, 1994.

[9] CSIR. *Guidelines for human settlement planning and design, Chapter 6: Stormwater Management, and Chapter 9: Water supply.* CSIR, Department of Housing, Pretoria, 2000.

[10] J. Díaz. *Objektorientierte Modellierung geotechnischer Systeme.* PhD thesis, Technical University of Darmstadt, 1998.

[11] DFG. Deutsche Forschungsgemeinschaft, http://www.dfg.de.

[12] ESRI. *ShapeFile format.* Environmental Systems Research Institute (ESRI), http://www.esri.com, 2000.

[13] ESRI. *ArcExplorer.* Environmental Systems Research Institute (ESRI), http://www.esri.com/arcexplorer, 2002.

[14] J. Garret and M. Hakim. Modeling engineering design information: An object-centered approach. *Computing in Civil Engineering: proceedings*, 1994.

[15] D. Hartmann. Einsatz objektorientierter Paradigmen für den interaktiven Entwurf optimal ausgelegter Tragwerke. In *Deutsche Forschungsgemeinschaft Abschlussbericht: Objektorientierte Modellierung in Planung und Konstruktion*, chapter 4.2. Wiley, 2000.

[16] D. Hartmann. Grundlegende Betrachtungen zur Anwendung der Objektorientierung in der Planung und Konstruktion des Bauwesens. In *Deutsche Forschungsgemeinschaft Abschlussbericht: Objektorientierte Modellierung in Planung und Konstruktion.* Wiley, 2000.

[17] D. Hartmann. Objekorientierte Strukturanalyse, Bemessung und konstruktive Durchbildung von Industriebauten unter besonderer Berücksichtigung parallel ablaufender Prozesse. In *Deutsche Forschungsgemeinschaft Abschlussbericht: Objektorientierte Modellierung in Planung und Konstruktion*, chapter 4.1. Wiley, 2000.

[18] Hosang and Bischof. *Abwassertechnik.* B.G. Teubner, 10th edition, 1993.

[19] R. Huebler. Intelligente CAAD-Systeme in objektorientierte Umgebung (iCAAD). In *Deutsche Forschungsgemeinschaft Abschlussbericht: Objektorientierte Modellierung in Planung und Konstruktion*, chapter 2.1. Wiley, 2000.

[20] IMSI. *TurboCAD Professional.* IMSI, http://www.imsi.com/turbocad, 2002.

[21] Intel. *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference.* Intel Corporation, http://www.intel.com/-design/pentium4/manuals/, 2002.

[22] U. Kolender. *Qualitätsteigernde Maßnahmen bei der Entwicklung von Software-Systemen im Bauwesen durch Objektrelationen.* PhD thesis, Lehrstuhl für Ingenieurinformatik in Bauwesen, Ruhr-University Bochum, 1997.

[23] A. Laabs. *Methoden für die Modellierung mit Objekten im Bauingenieurswesen.* PhD thesis, Technical University of Berlin, 1998.

[24] L. Li. *Java: data structures and programming.* Springer, 1998.

[25] B. Loubser and S. Sinske. *SEWSAN 2.3 User's Guide for Sewer System Analysis.* http://www.sewsan.com, 2000.

[26] Model Maker. *Model Maker.* Model Maker Systems, http://www.modelmaker.co.za, 2002.

[27] U. Meißner. Objektorientierte Tragwerksmodelle für die Systemintegration von Planungs- und Konstruktionsvorgängen im Bauwesen. In *Deutsche Forschungsgemeinschaft Abschlussbericht: Objektorientierte Modellierung in Planung und Konstruktion*, chapter 4.3. Wiley, 2000.

[28] Microsoft. *Visual Basic 3.0 for Windows.* http://www.microsoft.com, 1993.

[29] V. Novotny, K. Imhoff, M. Olthof, and P. Krenkel. *Karl Imhoff's Handbook of Urban Drainage and Wastewater Disposal.* J. Wiley and Sons, 1989.

[30] M. Olbrich. *Relationsorientiertes Modellieren mit Objekten in der Bauinformatik.* PhD thesis, Institut für Bauinformatik, University of Hannover, 1998.

[31] R. Orfali and D. Harkey. *Client/Server Programming with Java and Corba.* Wiley, 1998.

[32] P.J. Pahl. Complexity of civil engineering software. Personal comm., Dec 1999.

[33] P.J. Pahl. *Data Structures, Lecture Notes.* http://www.ifb.bv.tu-berlin.de, 2000.

[34] P.J. Pahl and R. Damrath. *Mathematische Grundlagen der Ingenieurinformatik.* Springer, 2000.

[35] P.J. Pahl and R. Damrath. Objektorientierte Analyse und Visualisierung zeitabhängiger physikalischer Zustände dreidimensionaler Körper. In *Deutsche Forschungsgemeinschaft Abschlussbericht: Objektorientierte Modellierung in Planung und Konstruktion*, chapter 3.1. Wiley, 2000.

[36] R. Pennington. *Introductory Computer Methods and Numerical Analysis*. Collier-Macmillan, 1965.

[37] C. Petzold. *Windows Programming*. Osborne McGraw-Hill, 1986.

[38] J. Rumbaugh et al. *Objektorientiertes Modellieren und Entwerfen*. Hanser Verlag, München, 1994.

[39] U. Scherer. Produktinformationssysteme unterstützt durch dynamische Klassifikation und ähnlichkeitsbasierte Suche. In *Deutsche Forschungsgemeinschaft Abschlussbericht: Objektorientierte Modellierung in Planung und Konstruktion*, chapter 4.4. Wiley, 2000.

[40] H. Schildt. *Advanced Turbo Pascal Programming and Techniques*. Osborne McGraw-Hill, 1986.

[41] G. Schutte. *Eine Erweiterung des Prinzips der virtuellen Verschiebungen zur Ermittlung von Spannungen*. PhD thesis, Institut fuer Bauingenierswesen, Technical University of Berlin, 2000.

[42] V. Shaw. The development of contributory hydrographs for sanitary sewers and their use in sewer design. *South African Civil Engineer*, sep 1963.

[43] A. Sinske. Development of a modern Water Distribution Analysis Program for the Windows Environment. In *Twenty-first Annual Symposium on Information Technology in Engineering*, chapter 2. South African Institution of Civil Engineering, Sep 1999.

[44] Sun. *Java, version 1.3*. http://www.java.sun.com, 2001.

[45] K. Wassermann. Integration raum- und bauteilorientierter Daten in der Gebäudeplanung in einem zentralen Objektmodell. In *Deutsche Forschungsgemeinschaft Abschlussbericht: Objektorientierte Modellierung in Planung und Konstruktion*, chapter 2.2. Wiley, 2000.

[46] H. Werner. Objektorientierte Modelle und Methoden in der Geotechnik. In *Deutsche Forschungsgemeinschaft Abschlussbericht: Objektorientierte Modellierung in Planung und Konstruktion*, chapter 6.2. Wiley, 2000.

[47] J. Wörner. Objektorientierte Integration von Teilprozessen im Bauwesen mit Hilfe einer objektorientierten Datenbank für den Bereich der Bemessung und Konstruktion von Hochbauteilen im Massivbau. In *Deutsche Forschungsgemeinschaft Abschlussbericht: Objektorientierte Modellierung in Planung und Konstruktion*, chapter 4.6. Wiley, 2000.

Appendix A

Abbreviations and trademarks

A.1 Abbreviations

The following abbreviations are used throughout the text.

General:

OO: object-oriented

UML: Unified Modeling Language

RMI: Remote Method Invocation

Models:

AC design: Application-Centred design

AC approach: Application-Centred approach

AC system: Application-Centred software system

MC design: Model-Centred design

MC approach: Model-Centred approach

MC system: Model-Centred system

Evaluation criteria:

BIC: basic instruction count

DoE: duration of execution

PDS: persistent data size

UIC: user interaction count

A.2 Trademarks

The following trademarks are used throughout the text.

AutoDesk, AutoCAD and AutoDesk World are registered trademarks or trademarks of AutoDesk, Inc in the USA and/or other countries.

Borland, Delphi and dBASE IV are registered trademarks or trademarks of Borland Software

dBASE is a registered trademark of dBase, Inc in the U.S. and other countries

ESRI and ArcExplorer are registered trademarks or trademarks of Environmental Systems Research Institute (ESRI), in the USA and/or other countries.

IMSI and TurboCAD are registered trademarks or trademarks of IMSI, in the USA and/or other countries.

Intel and Pentium are are registered trademarks or trademarks of Intel Corporation in the U.S. and other countries

Java and all Java-based marks are registered trademarks or trademarks of Sun Microsystems, Inc. in the U.S. and other countries

Modelmaker Systems, and Modelmaker are registered trademark or trademarks of Modelmaker Systems, in South Africa and/or other countries.

OpenDWG is a trademark of OpenDWG Alliance in the United States and/or other countries. Corporation, in the USA and/or other countries.

OMG and CORBA are registered trademarks or trademarks of Object Management Group, Inc

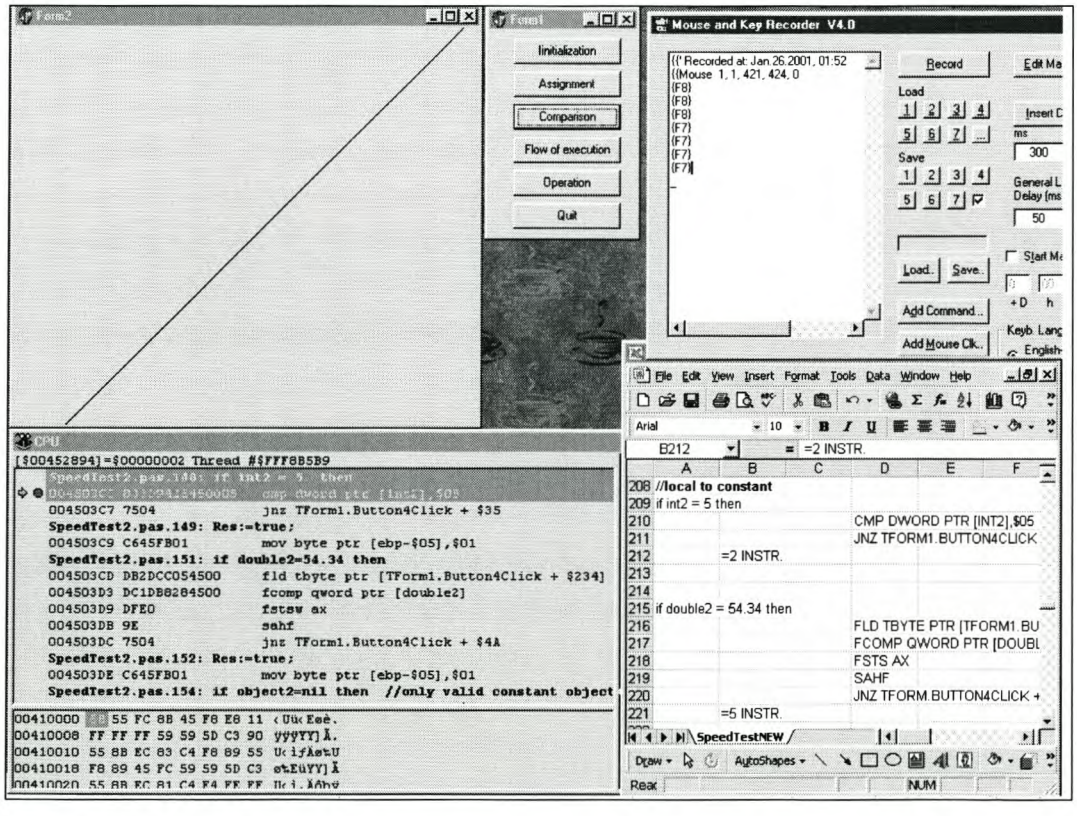
SEWSAN, SEW2SHAPE and SEW2DTM are registered trademarks or trademarks of GLS Engineering Software (Pty) Ltd, in South Africa and/or other countries.

Microsoft, ActiveX, Visual Basic, Visual C++, Visual C# and Microsoft Windows are registered trademarks or trademarks of Microsoft Corporation, in the USA and/or other countries.

All other brand and product names may be trademarks or registered trademarks of their respective holders.

Appendix B

Evaluation of Basic Instruction Count



Screen shot from test bench computer

Spreadsheet evaluation

COUNT SHEET FOR BASIC OPERATION COUNT

INITIALIZATION:

```
//local

int1:=0;
                        MOV [INT1].EAX
        =1 INSTR.

LocalTest:=nil;
                        XOR EAX.EAX
                        MOV [LOCALTEST],EAX

        =2 INSTR.

double1:=0.0;
                        XOR EAX.EAX
                        MOV [DOUBLE1].EAX
                        MOV [DOUBLE1+$4].EAX
        =3 INSTR.


LocalTest:=TLocalTest.Create; (not used)
                        MOV DL,$01
                        MOV EAX,[$0044e460]
                        CALL TOBJECT.CREATE
        +182  INSTR.
                        MOV [LocalTest].EAX
        =186  INSTR.

LocalTest.Free;
                        MOV EAX,[LOCALTEST]
                        CALL TOBJECT.FREE
        +180 INSTR.
        =182 INSTR.


string1:='';
                        CALL @LSTRCLR
                        lstrclr:
                        MOV EDX,[EAX]
                        TEST EDX.EDX
                        JZ @LSTRCLR + $22
                        RET
        = 5 INSTR.
```

ASSIGNMENT:

```
//constant
int2:=5;
                        MOV [INT2].$0000005
        = 1 INSTR.

double2:=54.34;
                        MOV [DOUBLE2],$1EB851EC
                        MOV [DOUBLE2 + $4],$404B2B85
        = 2 INSTR.

N/A

string2:='AABBCC';
                        MOV EAX,$004528C8
                        CALL @LSTRASG
                        TEST EDX.EDX
                        JZ @LSTRASG + $28
                        MOV ECX,[EDX-$08]
                        INC ECX
                        JNLE @LSTRASG + $24
                        PUSH EAX
                        PUSH EDX
                        MOV EAX,[EDX-$04]
                        CALL @NEWANSISTRING
        +177 INSTR.
                        MOV EDX,EAX
                        POP EAX
                        PUSH EDX
                        MOV ECX,[EAX-$04]
                        CALL MOVE
        +22 INSTR.
                        POP EDX
                        POP EAX
                        JMP @LSTRASG + $28
                        XCHG [EAX].EDX
                        TEST EDX.EDX
                        JZ @LSTRASG + $42
                        RET
```

FLOW OF EXECUTION

```
//passing flow to near location
for i:=1 to 10 do
begin
 ..
end
                        MOV WORD PTR [EBP-$06],$0001
                        ..
                        INC WORD PTR [EBP-$06],$0B
                        JNZ TFORM1.BUTTON5CLICK + $2E
        =3 INSTR

resx:=testcall(i,x1,x2)
                        PUSH DWORD PTR [EBP-$14] ; x-low push val on stack
                        PUSH DWORD PTR [EBP-$18] ; x-high

                        LEA EDX,[EBP-$20]       ; x2
                        MOVZX EAX,[EBP-$06]     ; i
                        CALL TESTCALL
                        FSTP QWORD PTR [EBP-$10] ; resx=RESULT
                        WAIT
        =7 INSTR


function testcall(i:integer;x:double.var x2:double):double;
begin

end;
                        PUSH EBP            ;prefix
                        MOV EBP.ESP
                        ADD ESP,-$10

                        MOV [EBP-$08].EDX     ;x2 copy onto stack
                        MOV [EBP-$04].EAX     ;i copy onto stack
        =5 INSTR

                        FILL DWORD PTR [EBP-$04] ; work with stack values
                        MOV EAX,[EBP-$08]       ; update var parameter
        = 2 INSTR PER PARAM
                        ..
                        ..
        = 2 INSTR PER RESULT

                        FLD QWORD PTR [EBP-$10]  ;update result

                        MOV ESP.EBP           ;suffix
                        POP EBP
                        RET $0008
        =3 INSTR

//passing flow to far location

Circle:=TheDrawing.Graphics.AddCircle(10.0,10.0,0.0);

                        PUSH $00
                        PUSH $00 ; 2 PER DOUBLE
        = 2 INSTR

                        ;derefence and call
                        PUSH EAX
                        MOV EAX,[THEDRAWING]
                        PUSH EAX
                        MOV EAX,[EAX]
        = 4 INSTR
                        CALL DWORD PTR [EAX+$3C]        ;Graphics
        = 697 INSTR
                        ;setting new far object
                        MOV EAX,[EBP-$34]
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$0000000110]  ;AddCircle
        = >5000 INSTR
```

OPERATIONS:

```
//Addition
 int4:=int4+1;
                        INC DWORD PTR [INT4]
        = 1 INSTR
 int3:=int3+int4;
                        MOV EAX,[INT4]
                        ADD [INT3].EAX
        = 2 INSTR
```

```
        =222 INSTR.

//local to local

int3:=int2;
                        MOV EAX,[INT2]
                        MOV [INT3],EAX
        =2 INSTR.

double3:=double2;
                        MOV EAX,[DOUBLE2]
                        MOV [DOUBLE3],EAX
                        MOV EAX,[DOUBLE2 + $4]
                        MOV [DOUBLE3 + $4],EAX
        =4 INSTR.

object3:=object2;
                        MOV EAX,$0045289C
                        MOV EDX,[OBJECT2]
                        CALL @INTFCOPY
        +19 INSTR
        =22 INSTR.


string3:=string2;
                        MOV EAX,$004528CC
                        MOVE EDX,[STRING2]
                        CALL @LSTRASG
        +10 INSTR.
        =12 INSTR.

//Constant to far

TheView.AutoRedraw:=true;
                        PUSH $FF
                        MOV EAX,[THEVIEW]
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$00000110]
        +10 INSTR.
        =14 INSTR.


TheVertex.X:=5.7;
                        PUSH $4016CCCC
                        PUSH $CCCCCCCD
                        MOV EAX,[THEVERTEX]
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$5C]
        +530 INSTR
        =535 INSTR

TheView.Name:='TEST';
                        PUSH $0044F948
                        MOV EAX,[THEVIEW]
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$64]
        +414 INSTR
        =418 INSTR.

//local to far

b:=TheView.AutoRedraw;
                        LEA EAX,[EBP-$10]
                        PUSH EAX
                        MOV EAX,[THEVIEW]
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$0000010C]
        +17 INSTR.
                        MOV EAX,[EBP-$0C]
                        MOV [INT4],EAX
        =19 INSTR.

double4:=TheVertex.X;
                        LEA EAX,[EBP-$1C]
                        PUSH EAX
                        MOV EAX,[THEVIEW]
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$58]
        +345 INSTR.
                        FLD QWORD PTR [EBP-$1C]
                        FSTP QWORD PTR [DOUBLE4]
                        WAIT
        =354 INSTR.

object4:=TheApplication.ActiveDrawing;
                        LEA EDX,[EBP-$10]
                        MOV EAX,[THEAPPLICATION]
                        CALL THXAPPLICATION.GET_ACTIVEDRAWING
        +274 INSTR.

double4:=double4+1.25;
                        FLD QWORD PTR [DOUBLE4]
                        FADD DWORD PTR [TFORM1.BUTTON6CLICK + $364]
                        FSTP QWORD PTR [DOUBLE4]
                        WAIT
        = 4 INSTR
double3:=double3+double3;
                        FLD QWORD PTR [DOUBLE3]
                        FADD DWORD PTR [DOUBLE3]
                        FSTP QWORD PTR [DOUBLE3]
                        WAIT
        = 4 INSTR

//Subtraction
int4:=int4-1;
                        DEC DWORD PTR [INT4]
        = 1 INSTR
int4:=int4-int3;
                        MOV EAX,[INT3]
                        SUB [INT4],EAX
        = 2 INSTR
double4:=double4-1.25;
                        FLD QWORD PTR [DOUBLE4]
                        FSUB QWORD PTR [DOUBLE3]
                        FSTP QWORD PTR [DOUBLE4]
                        WAIT
        = 4 INSTR
double4:=double4-double3;
                        FLD QWORD PTR [DOUBLE4]
                        FSUB QWORD PTR [DOUBLE3]
                        FSTP QWORD PTR [DOUBLE4]
                        WAIT

        = 4 INSTR

//Multiplication
int4:=int4*4;
                        MOV EAX,[INT4]
                        SHL EAX,$02
                        MOV [INT4],EAX
        = 3 INSTR
int4:=int3*int4;
                        MOV EAX,[INT3]
                        IMUL DWORD PTR [INT4]
                        MOV [INT4],EAX
        = 3 INSTR
double4:=double4*1.25;
                        FLD QWORD PTR [DOUBLE4]
                        FMUL DWORD PTR [TFORM1.BUTTON6CLICK + $364]
                        FSTP QWORD PTR [DOUBLE4]
                        WAIT
        = 4 INSTR
double4:=double4*double3;
                        FLD QWORD PTR [DOUBLE4]
                        FMUL QWORD PTR [DOUBLE3]
                        FSTP QWORD PTR [DOUBLE4]
                        WAIT
        = 4 INSTR

//Division
int4:=int4 div 2;
                        MOV EAX,[INT4]
                        SAR EAX,1
                        JNS TFORM1.BUTTON6CLICK + $18d
                        MOV [INT4],EAX
        = 4 INSTR
int3:=int3 div int4;
                        MOV EAX,[INT3]
                        CDQ
                        IDIV DWORD PTR [INT4]
                        MOV [INT3],EAX
        = 4 INSTR
double4:=double4 / 1.25;
                        FLD QWORD PTR [DOUBLE4]
                        FDIV DWORD PTR [TFORM1.BUTTON6CLICK + $364]
                        FSTP QWORD PTR [DOUBLE4]
                        WAIT
        = 4 INSTR
double4:=double4 /double3;
                        FLD QWORD PTR [DOUBLE4]
                        FDIV QWORD PTR [DOUBLE3]
                        FSTP QWORD PTR [DOUBLE4]
                        WAIT
        = 4 INSTR
//Casting
string1:=inttostr(int1);
                        LEA EDX,[EBP-$1C]
                        MOV EAX,[INT1]
                        CALL INTTOSTR
        +417 INSTR

                        MOV EDX,[EBP-$1C]
                        MOV EAX,$004528CC
                        CALL @LSTRASG
        +190 INSTR
```

```
                        MOV EDX,[EBP-$10]
                        MOV EAX,$004528A8
                        CALL @INTFCOPY
        +20 INSTR.
        =300 INSTR.


string4:=TheApplication.Name
                        LEA EDX,[EBP-$20]
                        MOV EAX,[THEAPPLICATION]
                        CALL TXAPPLICATION.GET_NAME
        +383 INSTR.
                        MOV EDX,[EBP-$20]
                        MOV EAX,$00528D8
                        CALL @LSTRFROMWSTR
        +276 INSTR.
        =665 INSTR.

        COMPARISON:

//local to constant
if int2 = 5 then
                        CMP DWORD PTR [INT2],$05
                        JNZ TFORM1.BUTTON4CLICK + $35
        =2 INSTR.


if double2 = 54.34 then
                        FLD TBYTE PTR [TFORM1.BUTTON4CLICK + $22]
                        FCOMP QWORD PTR [DOUBLE2]
                        FSTS AX
                        SAHF
                        JNZ TFORM.BUTTON4CLICK + $5B
        =5 INSTR.

if object2 = nil then
                        CMP DWORD PTR [OBJECT2].$00
                        JNZ TFORM1.BUTTON4CLICK + $57
        =2 INSTR.


if string2 = 'AABBCC' then
                        MOV EAX,[STRING2]
                        MOV EDX,$0044FC60
                        CALL @LSTRCMP
        +46 INSTR.
                        JNZ TFORM1.BUTTON4CLICK + $70
        =50 INSTR.

//local to local
if int3 = int2 then
                        MOV EAX,[INT3]
                        MOV EAX,[INT2[
                        JNZ TFORM1.BUTTON4CLICK + $81
        =3 INSTR.

if double3 = double2 then
                        FLD QWORD PTR [DOUBLE3]
                        FCOMP QWORD PTR [DOUBLE2]
                        FSTSW AX
                        SAHF
                        JNZ TFORM1.BUTTON4CLICK + $BD
        =5 INSTR.


if object3 = object2 then
                        MOV EAX,[OBJECT3]
                        CMP EAX,[OBJECT2]
                        JNZ TFORM1.BUTTON4LICK + $92
        =3 INSTR.


if string3 = string2 then
                        MOV EAX,[STRING3]
                        MOV EDX,[STRING2]
                        CALL @LSTRCMP
        +11 INSTR.
                        JNZ TFORM1.BUTTON4CLICK + $BD
        =15 INSTR.

//far to constant
if TheView.AutoRedraw=true then
                        LEA EAX,[EBP-$10
                        PUSH EAX
                        MOV EAX,[THEVIEW]
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$0000010C]
        +17 INSTR.
                        MOV AX,[EBP-$10]
                        NEG AX
                        SBB EAX,EAX
                        NEG EAX
                        CMP AL,$01
        =613 INSTR
string2:=floattostr(double3);
                        FLD QWORD PTR [DOUBLE3]
                        ADD ESP,-$0C
                        FSTP TBYTE PRE [ESP]
                        WAIT
                        LEA EAX,[EBP-$20]
                        CALL FLOATTOSTR
        +526 INSTR
                        MOV EDX,PEBO-$20]
                        MOV EAX,$004528D0
                        CALL @LSTRASG
        +184 INSTR
        =719 INSTR
int4:=strtoint(string1);
                        MOV EAX,[STRING1]
                        CALL STRTOINT
        + 66 INSTR
                        MOV [INT4],EAX
        = 69 INSTR
double4:=strtofloat(string3);
                        MOV EAX,[STRING3]
                        CALL STRTOFLOAT
        +114 INSTR
                        FSTP QWORD PTR [DOUBLE4]
                        WAIT
        =118 INSTR

//Dereferencing

string1:=Theapplication.ActiveDrawing.Views[ind].Name:
                        LEA EAX,[EBP-$24]  ;ACTIVEDRAWING
                        CALL @WSTRCLR
        +4 INSTR

                        PUSH EAX
                        LEA EAX,[EBP-$28]  ;VIEWS
                        CALL @INTFCLEAR
        +4 INSTR

                        PUSH EAX
                        LEA EAX,[EBP-$18]  ;IND
                        PUSH EAX
                        LEA EAX,[EBP-$2C]  ;NAME
                        CALL @INTFCLEAR
        +4 INSTR

                        PUSH EAX
                        LEA EDX,[EBP-$30]
                        MOV EAX,[THEAPPLICATION]
                        CALL TXAPPLICTION.GET_ACTIVEDRAWING ;IN TLB
        +253 INSTR

                        MOV EAX,[EBP-$30]
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$00000094] ; VIEWS
        +44 INSTR

                        MOV EAX,[EBP-$2C]
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$24] ; IND
        +375 INSTR

                        MOV EAX,[EBP-$28]
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$60] ; NAME
        +99 INSTR

                        MOV EDX,[EBP-$24]
                        MOV EAX,$004528CC       ;STRING1=RES
                        CALL @LSTRFROMWSTR
        +284 INSTR
        =1096 INSTR

string2:=TheView.Name;
                LEA EAX,[EBP-$34]
                CALL @WSTRCLR
    +4 INSTR
                PUSH EAX
                MOV EAX,[THEVIEW]
                PUSH EAX
                MOV EAX,[EAX]
                CALL DWORD PTR [EAX+$60]
    +119 INSTR

                MOV EDX,[EBP-$34]
                MOV EAX,$004528D0
                CALL @LSTRFROMWSTR
    +304 INSTR
    =437 INSTR
```

```
                        JNZ TFORM1.BUTTON4CLICK + $E7
        =29 INSTR.

if TheVertex.x=5.7 then
                        LEA EAX,[EBP-$18]
                        PUSH EAX
                        MOV EAX,[THEVERTEX
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$58]
        +228 INSTR.
                        FLD QWORD PTR [EBP-$18]
                        FLD TBYTE PTR [TFORM1.BUTTON4CLICK + $248]
                        FCOMPP
                        FSTSW AX
                        SAHF
                        JNZ TFORM1.BUTTON4CLICK + $10F
        =240 INSTR.

if TheVertex=nil then
                        CMP DWORD [PTR] [THEVERTEX],$00
                        JNZ TFORM1.BUTTON4CLICK + $118
        =2 INSTR.

if TheView.Name='TEST' then
                        LEA EAX,[EBP-$1C]
                        CALL @WSTRCLR
                        PUSH EAX
                        MOV EAX,[THEVIEW
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$60]
        +83 INSTR.
                        MOV EAX,[EBP-$1XC
                        MOV EDX,$0044FC78
                        CALL @WSTRCMP
                        JNZ TFORM1.BUTTON4CLICK + $13B
        =94 INSTR.

//local to far

if int4=TheDrawing.Index then
                        LEA EAX,[EBP-$20]
                        PUSH EAX
                        MOV EAX,[THEDRAWING]
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$4C]
        +165 INSTR.
                        MOV EAX,[EBP-$20]
                        MOV EAX,[INT4]
                        JNZ TFORM1.BUTTON4CLICK + $15E
        =174 INSTR.


if double4=TheVertex.X then
                        LEA EAX,[EBP-$18]
                        PUSH EAX
                        MOV EAX,[THEVERTEX]
                        PUSH EAX
                        MOV EAX,[EAX]
                        CALL DWORD PTR [EAX+$58]
        +345 INSTR.
                        FLD QWORD PTR [EBP-$18]
                        FCOMP QWORD PTR [DOUBLE4]
                        FSTSW AX
                        SAHF
                        JNZ TFORM1.BUTTON4CLICK + $1A0
        =356 INSTR.


if object4=TheApplication.ActiveDrawing then
                        LEA EDX,[EBP-$24]
                        MOV EAX,[THEAPPLICATION]
                        CALL TXAPPLICATION.GET_ACTIVEDRAWING
        +250 INSTR.
                        MOV EAX,[EBP-$24]
                        CMP EAX,[OBJECT4]
                        JNZ TFORM.BUTTON4CLICK + $17A
        =256 INSTR.

if string4=TheView.Name then
                        LEA EAX,[EBP-$28]
                        MOV EDX,[STRING4]
                        CALL @WSTRFROMLSTR
        +235 INSTR.
                        MOV EAX,[EBP-$28]
                        PUSH EAX
                        LEA EAX,[EBP-$2C]
                        CALL @WSTRCLR
        +5 INSTR.
                        PUSH EAX
                        MOV EAX,[THEVIEW]
                        PUSH EAX
```

```
//String Handling
 string2:=copy(String1,1,5);
                        PUSH $004528d4
                        MOV ECX,$00000005
                        MOV EDX,$00000001
                        MOV EAX,[STRING1]
                        CALL @LSTRCOPY
        + 310 INSTR
        = 315 INSTR
int1:=pos('D',String1);
                        MOV EDX,[STRING1]
                        MOV EAX,$004507FC
                        CALL @LSTRPOS
        + 40 INSTR
                        MOV [INT1],EAX
        = 44 INSTR

int4:=length(string1);
                        MOV EAX,[STRING1]
                        CALL @LSTRLEN
        + 4 INSTR
                        MOV [INT4],EAX
        = 7 INSTR

// Text File handling
write(filvar,int4);
                        MOV EDX,[INT4]
                        LEA EAX,[EBP-$0000001E4]
                        CALL @WRITE0LONG
        + 94 INSTR
                        CALL @FLUSH
        +  8 INSTR

        =106 INSTR

write(filvar,double4);
                        FLD QWORD PTR [DOUBLE4]
                        ADD ESP,-$0C
                        FSTP TBYTE PRE [ESP]
                        WAIT
                        LEA EAX,[EBP-$000001E4]
                        CALL @WRITE0EXT
        +280 INSTR
                        CALL @FLUSH
        +  8 INSTR
        =294 INSTR

write(filvar.string4);
                        MOV EDX,[STRING1]
                        LEA EAX,[EBP-$00000E4]
                        CALL @WRITE0LSTRING
        +42 INSTR
                        CALL @FLUSH
        + 8 INSTR
        =54 INSTR


writeln(filvar);
                        CALL @WRITELN
        =30 INSTR


read(filvar.int4);
                        LEA EAX,[EBP-$000001E4]
                        CALL @READLONG
        +117 INSTR
                        MOV [INT4].EAX
        =+120 INSTR

read(filvar.double4);
                        LEA,[EBP-$000001E4]
                        CALL @READEXT
        +836 INSTR
                        FSTP QWORD PTR [DOUBLE4]
                        WAIT
        =840 INSTR

read(filvar,string4);
                        MOV EDX,$004528D0
                        LEA EAX,[EBP-$000001E4]
                        CALL @READLSTRING
        +367 INSTR
                        LEA EAX,[EBP-$0000001E4]
        =370 INSTR

readln(filvar)
                        CALL @READLN
        =24 INSTR


//Binary file handling: testrec.x,testrec.y,testrec.z

 write(binfile,testrec);
```

```
                    MOV EAX,[EAX]
                    CALL DWORD PTR [EAX+$60]
+101 INSTR.
                    MOV EDX,[EBP-$2C]
                    POP EAX
                    CALL @WSTRCMP
+32 INSTR.
                    JNZ TFROM1.BUTTON4CLICK + $1DA
=389 INSTR.
```

```
                                LEA EDX,[EBP-$00000348]
                                LEA EDX,[EBP-$00000330]
                                CALL @WRITEREC
                        +24 INSTR
                        =27 INSTR

 read(binfile,testrec)
                                LEA EDX,[EBP-$00000348]
                                LEA EAX,[EBP-$00000330]
                                CALL @READREC
                        +24 INSTR
                        =27 INSTR


//Mathematical
  double4:=sqrt(double3);
                                FLD QWORD PTR [DOUBLE3]
                                FSQRT
                                FSTP QORD PTR [DOUBLE4]
                                WAIT
                         =4 INSTR

 typical TCAD function call
                        =2000 INSTR
typical external  JAVA hash function
                        = 500 INSTR
typical JAVA create object
                        = 10 INSTR
```

Appendix C

Source code for typical MC Application

C.1 Implementation of a typical *Hydraulic model*

```
package HydraulicModel;

import TransformerModule.CommandMap;

public class HydraulicApp {

/**
 * Static package level storage of hydraulic model = network data
 */
  static  NetworkData hydraulicModel = new NetworkData();

/**
 * The event handler responding to the user action to see a visualiza-
tion of the model;
 * Shows graphics for result variable 2 by call the CommandMap and pass-
ing a reference
 * of the hydraulic model.
 */
  public static void showGraphicsClick(){
     CommandMap.showGraphics(hydraulicModel, 2);
     System.out.println("Graphics clicked");
  }

/**
 * The main method; loads the data, calculates ages and flows, show graphics.
 */
  public static void main(String args[]){
      System.out.println("Welcome to Sewsan MC");

      //load the model
      hydraulicModel.loadSDFFile("dummy.sdf");
```

```
      //build topo and update lengths
      hydraulicModel.buildTopo(); hydraulicModel.printPipes();
      hydraulicModel.updateLengths(); hydraulicModel.printLengths();

      //update elevations and slopes
      hydraulicModel.updateSlopes(); hydraulicModel.printSlopes();

      //update ages and flows
      hydraulicModel.calcAges(); hydraulicModel.printAges();
      hydraulicModel.calcFlows(); hydraulicModel.printFlows();

      //show results graphically
      showGraphicsClick();
      System.out.println("End of program");
  }
}

package HydraulicModel;

/**
 * Implementation of the Hydraulic Mode; Collects all Pipe objects;
 * must access CommandMap Class in the TransformerModule to issue commands to the
 * GraphicsModel; Cannot access GraphicsModel directly
 */
public class NetworkData {

  //The pipes in the hydraulic model. No nodes are used.
  public int numberPipes;
  public Pipe pipes[];

 //Constructor
  public NetworkData() {
    numberPipes=0;
  }

/**
 * Builds the topology, i.e. populate the dsPipeIndex attribute by nested looping
 */
  public void buildTopo(){
     for (int i=0;i<numberPipes;i++)
        if (pipes[i].idBeginMH.equals("OUTFALL"))
          pipes[i].indexDsPipe=-1;
        else
         for (int j=0;j<numberPipes;j++)
           if (pipes[i].idEndMH.equals(pipes[j].idBeginMH))
                pipes[i].indexDsPipe=j;

      System.out.println("Topology build");
  }

/**
 * Update the lengths of all pipes
 */
  public void updateLengths(){
```

```
    for (int i=0;i<numberPipes;i++)
       pipes[i].updateLength();
     System.out.println("Lengths updated");
  }


/**
 * Update the slopes of all pipes
 */
  public void updateSlopes(){
    for (int i=0;i<numberPipes;i++)
       pipes[i].updateSlope();
     System.out.println("Slopes updated");
  }

/**
 * Update the age of all pipes
 */
  public void calcAges(){
    for (int i=0;i<numberPipes;i++)
        pipes[i].updateAge(i);
     System.out.println("Ages calculated");
  }

/**
 * Update the flows of all pipes
 */
  public void calcFlows(){
    for (int i=0;i<numberPipes;i++)
        pipes[i].updateFlow(i);
     System.out.println("Flows calculated");
  }


/**
 * Print the ids of all pipes
 */
  public void printPipes(){
    for (int i=0;i<numberPipes;i++)
      pipes[i].printPipe();
     System.out.println("Pipes printed");
  }

/**
 * Print the lengths of all pipes
 */
 public void printLengths(){
    for (int i=0;i<numberPipes;i++)
        pipes[i].printLength();
     System.out.println("Lengths printed");
  }

/**
 * Print the slopes of all pipes
 */
```

```
 public void printSlopes(){
     for (int i=0;i<numberPipes;i++)
         pipes[i].printSlope();
      System.out.println("Slopes printed");
  }


/**
 * Print the ages of all pipes
 */
 public void printAges(){
     for (int i=0;i<numberPipes;i++)
         pipes[i].printAge();
      System.out.println("Ages printed");
 }

/**
 * Print the flows of all pipes
 */
  public void printFlows(){
     for (int i=0;i<numberPipes;i++)
         pipes[i].printFlow();
      System.out.println("Flows printed");
  }

/**
 * Resembles the loading of an SDF file by populating the model; also
 * builds the topology, update lengths and slopes and prints results
 */
  public void loadSDFFile(String filename){
      numberPipes=5;
      pipes = new Pipe[numberPipes];
      for (int i=0;i<numberPipes;i++){
        pipes[i]=new Pipe(this);
      }

      pipes[0].idBeginMH="A1";
      pipes[0].idEndMH="B";
      pipes[0].xBeginMH=100.0;
      pipes[0].yBeginMH=150.0;
      pipes[0].zBeginMH=150.0;

      pipes[1].idBeginMH="A2";
      pipes[1].idEndMH="B";
      pipes[1].xBeginMH=100.0;
      pipes[1].yBeginMH=50.0;
      pipes[1].zBeginMH=145.0;

      pipes[2].idBeginMH="B";
      pipes[2].idEndMH="C";
      pipes[2].xBeginMH=100.0;
      pipes[2].yBeginMH=100.0;
      pipes[2].zBeginMH=143.0;

      pipes[3].idBeginMH="C";
```

```
        pipes[3].idEndMH="OUTFALL";
        pipes[3].xBeginMH=200.0;
        pipes[3].yBeginMH=100.0;
        pipes[3].zBeginMH=141.0;

        pipes[4].idBeginMH="OUTFALL";
        pipes[4].idEndMH="";
        pipes[4].xBeginMH=300.0;
        pipes[4].yBeginMH=100.0;
        pipes[4].zBeginMH=140.0;

        System.out.println("Model loaded");
    }
}

package HydraulicModel;
/**
 * Basic Pipe Class containing Topology, Geometry and Hydraulic attributes.
 */
public class Pipe {
    NetworkData nwdata;

    //Constants for hydraulics
    public static final double _beginMHInvertDrop=1.5;
    public static final double _endMHInvertDrop=1.7;
    public static final double _infiltration=1.0; //l/s/m

    //Topology
    public String idBeginMH, idEndMH;
    public int indexDsPipe;
    public double age;

    //Geometry
    public double xBeginMH, yBeginMH, zBeginMH;
    public double invertLevelBeginMH,invertLevelEndMH;
    public double length,slope;

    //Hydraulics
    public double flow;

    //Constructor
    public Pipe(NetworkData nwdata) {
        this.nwdata = nwdata;
        indexDsPipe=0;
        zBeginMH=0.0;
        length=0.0;
        slope=0.0;
        invertLevelBeginMH=0.0;
        invertLevelEndMH=0.0;
    }

/**
 * Updates the age of pipe, by doing the equavalent of a traversal from
 * the outfall manhole and accumulating pipe lengths;
 * Performs a nested loop to accumulate all ages. Exits  when outfall manhole
```

```
 * is found.
 */
  public void updateAge(int thisPipeIndex){
       Pipe dsPipe;
       age = 0;
       int indexDsPipe = thisPipeIndex;
       for (int i=0;i<nwdata.numberPipes;i++){
          dsPipe = nwdata.pipes[indexDsPipe];
          age = age + dsPipe.length;
          indexDsPipe = dsPipe.indexDsPipe;
          if (indexDsPipe==-1) break;
        }
  }

/**
 * Updates the flows of all downstream pipes with the contribution due to this pipe;
 * this is the equavalent of a post-order traversal from
 * the outfall manhole and accumulating pipe flow contributions.
 * Performs a nested loop to accumulate all ages. Exits  when outfall manhole
 * is found.
 */
  public void updateFlow(int thisPipeIndex){
       Pipe dsPipe;
       int indexDsPipe = thisPipeIndex;
       for (int i=0;i<nwdata.numberPipes;i++){
          dsPipe = nwdata.pipes[indexDsPipe];
          dsPipe.flow=dsPipe.flow + _infiltration * length;
          indexDsPipe = dsPipe.indexDsPipe;
          if (indexDsPipe==-1) break;
        }
  }


/**
 * Calculatates the length of a pipe by finding the coordinates of
 * begin manhole of downstream pipe. Is = 0 for outfall manhole/pipe.
 */
  public void updateLength(){
    if (indexDsPipe!=-1) {
      Pipe dsPipe = nwdata.pipes[indexDsPipe];
      double xEndMH = dsPipe.xBeginMH;
      double yEndMH = dsPipe.yBeginMH;
      if (length==0.0) length= Math.sqrt( (xEndMH-xBeginMH)*(xEndMH-xBeginMH) +
                                          (yEndMH-yBeginMH)*(yEndMH-yBeginMH) );
    }
  }


/**
 * Calculatates the slope of a pipe by finding the ground level of
 * begin manhole of downstream pipe. Is = 0 for outfall manhole/pipe,
 * assumes lengths have been calculated.
 */
    public void updateSlope(){
     if (indexDsPipe!=-1) {
```

```
      Pipe dsPipe = nwdata.pipes[indexDsPipe];
      double glEndMH = dsPipe.zBeginMH;
      if (slope==0.0)
        if (length!=0) {
          slope= (zBeginMH - _beginMHInvertDrop - (glEndMH - _endMHInvertDrop))/
                                        length;
        }
    }
  }


/**
 * Print the age of the pipe.
 */
 public void printAge(){
   System.out.println(idBeginMH+": "+String.valueOf(age));
 }

/**
 * Print the flow of the pipe.
 */
 public void printFlow(){
   System.out.println(idBeginMH+": "+String.valueOf(flow));
 }

/**
 * Print the length of the pipe.
 */
 public void printLength(){
   System.out.println(idBeginMH+": "+String.valueOf(length));
 }

/**
 * Print the ids of the pipe.
 */
 public void printPipe(){
   System.out.println(idBeginMH+"->"+idEndMH);
 }

/**
 * Print the slope of the pipe.
 */
 public void printSlope(){
   System.out.println(idBeginMH+": "+String.valueOf(slope));
 }


}
```

C.2 Implementation of a typical *Visualization model*

```
package VisualizationModel;
```

```
public class VisualizationApp {

/**
 * Static package level storage of hydraulic model = network data
 */
  static GraphicsData visualizationModel = new GraphicsData();

/**
 * Public access to a reference of GraphicsData
 */
  public static GraphicsData getVisualizationModel() {return visualizationModel;}

/**
 * The final showGraphics method invoked from the CommandMap class; update link
 * distances and draw the links and nodes.
 */
  public static void showGraphics(int varIndex){
    System.out.println("Showing Graphics Now for variable "+String.valueOf(varIndex));
    visualizationModel.updateLinkDistances();
    visualizationModel.drawLinks();
    visualizationModel.drawNodes();
    System.out.println("Graphics shown");
  }
}

package VisualizationModel;

/**
 * Implementation of the Visualization Model; Collects all Link and Node objects;
 * must not access the TransformerModule as no return communication supported
 */
public class GraphicsData {

  //The links and nodes in the visualization model.
  public int numberLinks;
  public int numberNodes;

  public Link links[];
  public Node nodes[];

  public void createGraphicsData(int numberLinks, int numberNodes){
     this.numberLinks = numberLinks;
     this.numberNodes = numberNodes;

     links = new Link[numberLinks];
     nodes = new Node[numberNodes];

     for (int i=0;i<numberLinks;i++)
       links[i]=new Link(this);

     for (int i=0;i<numberNodes;i++)
       nodes[i]=new Node(this);
  }
```

```
/**
 * Updates all the distances (length) properties of links
 */
  public void updateLinkDistances(){
     for (int i=0;i<numberLinks;i++)
        links[i].updateDistance();
    System.out.println("Link distances updated");
  }

/**
 * Resembles drawing all links with line, text and arrow information
 */
  public void drawLinks(){
     for (int i=0;i<numberLinks;i++)  {
        links[i].drawLine();
        links[i].drawText();
        links[i].drawArrow();
     }
    System.out.println("Links drawn");
  }


/**
 * Resembles drawing all nodes with circles
 */
  public void drawNodes(){
     for (int i=0;i<numberNodes;i++)
        nodes[i].drawCircle();
    System.out.println("Nodes drawn");
  }


}

package VisualizationModel;

/**
 * The basic link class in the Visualization
 */

public class Link {

  private GraphicsData gdata;
  //Topology
  public String id;
  public int indexBeginNode;
  public int indexEndNode;

  //Geometry
  public double linkDist;

  //Hydraulic
  public double value;

  public Link(GraphicsData gdata){
```

```
    this.gdata=gdata;
  }

/**
 * Calculates the length of links from the nodal coordinates
 */
  public void updateDistance(){
      linkDist= Math.sqrt(
      (gdata.nodes[indexEndNode].xCoord-gdata.nodes[indexBeginNode].xCoord)*
      (gdata.nodes[indexEndNode].xCoord-gdata.nodes[indexBeginNode].xCoord) +
      (gdata.nodes[indexEndNode].yCoord-gdata.nodes[indexBeginNode].yCoord)*
      (gdata.nodes[indexEndNode].yCoord-gdata.nodes[indexBeginNode].yCoord) );
  }

/**
 * Resembles the drawing of a line in the visualization.
 */
 public void drawLine(){
     System.out.println("Drawing line from ["+
       String.valueOf(gdata.nodes[indexBeginNode].xCoord)+","+
       String.valueOf(gdata.nodes[indexBeginNode].yCoord)+"] to ["+
       String.valueOf(gdata.nodes[indexEndNode].xCoord)+","+
       String.valueOf(gdata.nodes[indexEndNode].yCoord)+"] for value :"+
       String.valueOf(value));
 }

/**
 * Resembles the drawing of text at the centre of the link
 */
 public void drawText(){
    double x = (gdata.nodes[indexBeginNode].xCoord + gdata.nodes[indexEndNode].xCoord) / 2;
    double y = (gdata.nodes[indexBeginNode].yCoord + gdata.nodes[indexEndNode].yCoord) / 2;

      System.out.println("Drawing text at ["+String.valueOf(x)+","+
                                          String.valueOf(y)+
                                             "] with value :"+
                                        String.valueOf(value));
 }

/**
 * Resembles the drawing of an arrow at the centre of the link; must calcu-
late sin and
 * required the length of a link in the calculation.
 */
 public void drawArrow(){
    double x = (gdata.nodes[indexBeginNode].xCoord + gdata.nodes[indexEndNode].xCoord) / 2;
    double y = (gdata.nodes[indexBeginNode].xCoord + gdata.nodes[indexEndNode].xCoord) / 2;
    double asin = (gdata.nodes[indexEndNode].xCoord -
                  gdata.nodes[indexBeginNode].xCoord) / linkDist;

      System.out.println("Drawing arrow at midpoint ["+String.valueOf(x)+","+
                                                    String.valueOf(y)+
                                               "] with sine value of "+
                                                    String.valueOf(asin));
 }
```

```
}

package VisualizationModel;

/**
 * The basic node class in the Visualization
 */
public class Node {

   private GraphicsData gdata;
   //Topology
   public int index;
   public String id;

   //Geometry
   public double xCoord;
   public double yCoord;
   public double value;

  public Node(GraphicsData gdata){
    this.gdata=gdata;
  }


/**
 * Resembles drawing a circle at the node
 */
 public void drawCircle(){
     System.out.println("Drawing circle at ["+
       String.valueOf(gdata.nodes[index].xCoord)+","+
       String.valueOf(gdata.nodes[index].yCoord)+"] for value :"+
       String.valueOf(value));
 }
}
```

C.3 Transformer module

```
package TransformerModule;

 import VisualizationModel.*;
 import HydraulicModel.NetworkData;

/**
 * Maps commands from to VisualizationModel; Has access to that model.
 */
public class CommandMap {

  /**
   * Transfers data using to the DataMap transformer, and finally
   * shows a graphic visualization for variable varIndex by calling
   * showGraphics in the VisualizationApp.
```

```
   *
   */
  public static void showGraphics(NetworkData nwData, int varIndex){
       GraphicsData gData = VisualizationApp.getVisualizationModel();

       DataMap.initTransformer(nwData,gData);
       DataMap.transferNetworkData(nwData,gData,varIndex);

       VisualizationApp.showGraphics(varIndex);
  }
}

package TransformerModule;

import HydraulicModel.*;
import VisualizationModel.*;

/**
 * Maps data from the Hydraulic Model to the Visualization Model; has ac-
cess to both models.
 */
public class DataMap {

/**
 * Initialize the transformer; calls createGraphicsData in GraphicsData
 */
   static void initTransformer(NetworkData nwdata, GraphicsData gdata){
     gdata.createGraphicsData(nwdata.numberPipes-1,nwdata.numberPipes);
   }

/**
 * Transfer the Network from NetworkData to GraphicsData
 */
   static void transferNetworkData(NetworkData nwdata, Graphics-
Data gdata, int varIndex){

      //links
      for (int i=0;i<gdata.numberLinks;i++){
        gdata.links[i].id = String.valueOf(i);
        gdata.links[i].indexBeginNode = i;

         switch (varIndex) {
         case 0: {gdata.links[i].value =nwdata.pipes[i].length;}
         case 1: {gdata.links[i].value =nwdata.pipes[i].slope;}
         case 2: {gdata.links[i].value =nwdata.pipes[i].age;}
         case 3: {gdata.links[i].value =nwdata.pipes[i].flow;}
         }
      }

      //nodes
      for (int i=0;i<gdata.numberNodes;i++){
        gdata.nodes[i].index = i;
        gdata.nodes[i].id = nwdata.pipes[i].idBeginMH;
        gdata.nodes[i].xCoord = -nwdata.pipes[i].yBeginMH;
        gdata.nodes[i].yCoord = -nwdata.pipes[i].xBeginMH;
```

```
            switch (varIndex) {
            case 0: {gdata.nodes[i].value =nwdata.pipes[i].invertLevelBeginMH;}
            case 1: {gdata.nodes[i].value =nwdata.pipes[i].invertLevelEndMH;}
            case 2: {gdata.nodes[i].value =nwdata.pipes[i].zBeginMH;}
            }

        }

       // Now must create new index for linkN2 from all nodes
        for (int i=0;i<gdata.numberLinks;i++){
          for (int j = 0;j<gdata.numberNodes;j++)
            if (nwdata.pipes[i].idEndMH.equals(gdata.nodes[j].id)) {
                gdata.links[i].indexEndNode = j;
                break;
             }
        }

    }

}
```

C.4 Implementation of the traversal algorithm

```
/**
 * Updates the age of pipe, by doing the equivalent of an
 * in-order traversal from the terminating pipe,
 * accumulating pipe lengths; Performs a nested loop to
 * accumulate all ages. Exits  when terminating pipe is
 * found (-1). Operation is of order O(n2)
 */
  public void updateAge(int thisPipeIndex){
       Pipe dsPipe;
       age = 0;
       int indexDsPipe = thisPipeIndex;
       for (int i=0;i<nwdata.numberPipes;i++){
          dsPipe = nwdata.pipes[indexDsPipe];
          age = age + dsPipe.length;
          indexDsPipe = dsPipe.indexDsPipe;
          if (indexDsPipe==-1) break;
        }
  }

/**
 * Updates the flows of all downstream pipes with the
 * contribution due to this pipe;
 * (its perLengthContribution x length); This is the
 * equivalent of a post-order traversal from the terminating
 * pipe and accumulating pipe flow contributions. Performs a
 * nested loop to accumulate all ages. Exits  when terminating
 * pipe is found (-1).
 */
  public void updateFlow(int thisPipeIndex){
       Pipe dsPipe;
       int indexDsPipe = thisPipeIndex;
       for (int i=0;i<nwdata.numberPipes;i++){
          dsPipe = nwdata.pipes[indexDsPipe];
          dsPipe.flow=dsPipe.flow +
           _perLengthContribution * length;
          indexDsPipe = dsPipe.indexDsPipe;
          if (indexDsPipe==-1) break;
        }
  }
```

Appendix D

MC Instruction Count Evaluation

CALCULATION SHEET FOR OPERATION COUNT: MODEL-CENTRED

TOTAL: 682 PIPES //104//682//2934//6721

	One Pipe	n Pipes	
	TOPOLOGY MODEL		
TCE	**3,143**	**2,143,526**	**SaveElevations()**
TEI	**36,965**	**25,209,955**	**LoadElevations()**
	ELEVATION MODEL		
ECI	**147**	**100,254**	**ReadNodes(Fname)**
EEI	**57**	**38,874**	**SaveNodes(Fname)**
	40,312	***27,492,609***	***UpdateElevations Total***
	TOPOLOGY MODEL		
TMI2	**55,211**	**37,653,902**	**LoadSystem()**
TME2	**201,759**	**137,599,954**	**SaveSystem()**
	HYDRAULIC MODEL		
HMI	**78,720**	**53,687,040**	**Read_Net()**
HME	**35,988**	**24,543,816**	**Save_Net()**
	280,479	***191,286,994***	***Build Model from CAD***
	91,199	***62,197,718***	***Draw Model***

#	Code	Count	Source Code
			Function SaveElevations() As Integer
			Export X & Y Coords from model to DTM File
1	(FNLF)	1	**For i = 0 To NumSel - 1**
0	()	0	'..
1	(ALFO)	300	Set Gr = ObjSel.Item(i) 'Returns part of a collection - graphic in the selection collection
1	(ORDI)	380	
1	(ALFI)	19	GrSelType = Gr.TypeByValue
1	(C2FS)	389	If (Gr.Layer.Name = ElevationTextLayer) And (GrSelType = imsiText) Then
1	(ORDE)	200	
1	(C2CI)	2	
1	(ALFI)	19	Gr.Vertices.UseWorldCS = True
1	(ORDE)	200	
1	(ALFO)	300	Set Ver = Gr.Vertices.Item(0) ' Return the Vertices collection for current graphic
1	(ORDE)	200	
1	(ORDI)	380	
1	(ALFD)	354	Coord.X = -(Ver.Y - TextOffSetConst)
2	(OSDC)	8	
1	(ALLD)	4	
1	(OSDC)	4	Coord.Y = -Ver.X
1	(ALFD)	354	
1	(OBW)	27	Put #1, , Coord
0	()	0	End If
1	(FNLN)	2	**Next i**
		..	
	TOTAL	**3143**	**End Function**
			Function LoadElevations() As Integer
			Import X & Y & Z Coords from DTM File to model
		..	
1	(CFCI)	29	**While Not (EOF(1))**
1	(OBR)	27	Get #1, , Coord
1	(C2CD)	4	If Coord.Z = -999999 Then GoTo exitpos
1	(OCIS)	613	ZStr = Trim(Format(Coord.Z, Fstr))
1	(FNPF)	2	Set Gr2 = PickEnt(-Coord.Y, -Coord.X, ElevationTextLayer, "0.00")
1	(ALLO)	22	
1	(FNC)	7	
4	(FNPR)	8	
2	(OSDC)	8	
1	**(REF)**	**29985.744**	

```
  1 (C2CO)          2    If isNothing(Gr2) = False Then
  1 (FFC)         212      Call Gr2.Delete
  1 (OCAD)       2000
  1 ()              0    End If
  1 (ICLO)          2    Set Gr2 = Nothing
  1 (FFPF)          6    Set Gr = ActDr.Graphics.AddText(ZStr, -Coord.Y, -Coord.X + TextOffSetConst, 0#, TextHeightConst, 0, 0, 0, 2)
  1 (ALLO)         22
  1 (FFC)         212
  1 (ORDE)        200
  9 (FFPR)         18
  2 (OSDC)          8
  1 (OADC)          4
  1 (OCAD)       2000
  1 (ACFS)         418    Gr.Properties("Layer") = ElevationTextLayer
  1 (ORDI)         380
  1 (ACFI)          14    Gr.Properties("PenColor") = -3
  1 (ORDI)         380
  1 (FNG)           1  Wend
  0 ()              0   expos:
                       ..
  TOTAL      36964.744 End Function


                       Function LoadSystem() As Integer
                        Load existing the sewer system model
                        '..
  1 (CFCI)         29   While (Not EOF(1))
  1 (OTRS)        370     Line Input #1, stg
  1 (REF)       19484     Call Parse_input(stg, result(), 27, ier)
  1 (C2CI)          2     If ier Then
  1 (FNG)           1     '.. Error
  0 ()              0     End If
  1 (ALFS)        665     typeStr = UCase$(CStr(result(1)))
  1 (OSM)         310     If Left$(typeStr, 5) = "BOTMH" Then
  1 (C2CS)         50     ' Only once
  1 (FNG)           1     Else
  1 (OAIC)          1      i = i + 1
 11 (ORDA)         11 'Deref
  1 (ALFS)        665      Smhstr(i) = CStr(result(2))
  2 (OCSD)        236      Xsm(i) = -val(result(3))  'SH
  2 (OSDC)          8      Ysm(i) = -val(result(4))  'SH
  2 (ALLD)          8
  1 (OCSD)        118      Glsm(i) = val(result(5))
  1 (ALLD)          4
  1 (ALFS)        665      Emhstr(i) = CStr(result(7))
  1 (C2CS)         50      If result(27) = "S" Then
  1 (OCSD)        118       Slope(i) = val(result(26))
  1 (ALLD)          4
  1 (ACLD)          2       Ism(i) = 0#
  1 (ACLD)          2       Iem(i) = 0#
  0 ()              0      Else
  0 ()              0       'seldom
  0 ()              0      End If
  0 ()              0
  3 (OCSD)        354      Diam(i) = val(result(9))
  3 (ALLD)         12      Man(i) = val(result(12))
  0 ()              0      EE1(i) = val(result(16))
  0 ()              0     End If
  1 (FNG)           1     Wend
  0 ()              0 '..
  0 ()              0 ' Now Draw the network
  0 ()              0  For i = 1 To TotalPipes
  0 ()              0    ' Find End Manhole Coords
  3 (ALLD)         12    X = BXsm: Y = BYsm: Z = BGlsm
  1 (FNLF)          1    For j = 1 To TotalPipes
500 (ORDA)        500    ' average = TotalPipes / 2
341 (C2LD)       1705     If Emhstr(i) = Smhstr(j) Then
341 (ALLD)       1364       X = Xsm(j)
341 (ALLD)       1364       Y = Ysm(j)
341 (ALLD)       1364       Z = Glsm(j)
341 (FNG)         341       Exit For
  0 ()              0     End If
  1 (FNLN)          2    Next j
  3 (ORDA)          3
  1 (FFPF)          6    Set Gr = ActDr.Graphics.AddLineSingle(Xsm(i), Ysm(i), Glsm(i) * zfact, X, Y, Z * zfact)
  1 (ALLO)         22
  1 (FFC)         212
  1 (ORDE)        200
  6 (FFPR)         12
  2 (OMDC)          8
  1 (OCAD)       2000
  1 (ACFS)         418    Gr.Properties("Layer") = TopoLayer
  1 (ORDI)         380
```

```
1 (ACFI)        14    Gr.Properties("PenColor") = -3
1 (ORDI)       380
3 (ORDA)         3    'MH Text
1 (FFPF)         6    Set Gr = ActDr.Graphics.AddText(Smhstr(i), Xsm(i), Ysm(i), 0#, TextHeightConst, 0, 0, 0, 2)
1 (ALLO)        22
1 (FFC)        212
1 (ORDE)       200
9 (FFPR)        18
1 (OCAD)      2000
1 (ACFS)       418    Gr.Properties("Layer") = MhTextLayer
1 (ORDI)       380
1 (ACFI)        14    Gr.Properties("PenColor") = -3
1 (ORDI)       380
3 (ORDA)         3
1 (ORDE)       200    'Elev Text
1 (FFPF)         6    Set Gr = ActDr.Graphics.AddText(Trim(Str(Glsm(i))), Xsm(i),
1 (ALLO)        22         Ysm(i) + TextOffSetConst, 0#, TextHeightConst, 0, 0, 0, 2)
1 (FFC)        212
1 (ORDE)       200
9 (FFPR)        18
1 (OCAD)      2000
1 (OCIS)       613
1 (OADC)         4
1 (ACFS)       418    Gr.Properties("Layer") = ElevationTextLayer
1 (ORDI)       380
1 (ORDI)       380    Gr.Properties("PenColor") = -3
1 (ORDA)         4
1 (C2CD)         4    If Slope(i) <> 0 Then
1 (FFPF)         6      Set Gr = ActDr.Graphics.AddText(Trim(Str(Slope(i))) + "%", (Xsm(i) + X) / 2#,
1 (ALLO)        22         (Ysm(i) + Y) / 2# - TextOffSetConst, 0#, TextHeightConst, 0, 0, 0, 2)
1 (FFC)        212
1 (ORDE)       200
9 (FFPR)        18
1 (OCAD)      2000
1 (OCIS)       613
2 (OADC)         8
1 (OSDC)         4
2 (ODDC)         8
0 ()             0    Else
0 ()             0      'seldom
0 ()             0    End If
0 ()             0
1 (ACFS)       418    Gr.Properties("Layer") = InvertTextLayer
1 (ORDI)       380
1 (ACFI)        14    Gr.Properties("PenColor") = -3
1 (ORDI)       380
3 (ORDA)         3    'Diam Text
2 (FFPF)        12    Set Gr = ActDr.Graphics.AddText(Trim(Str(Diam(i))),
0 ()             0         (Xsm(i) + X) / 2#, (Ysm(i) + Y) / 2#, 0#, TextHeightConst, 0, 0, 0, 2)
2 (ALLO)        44    Gr.Properties("Layer") = DiamTextLayer
2 (FFC)        424    Gr.Properties("PenColor") = -3
1 (ORDE)       200
2 (ORDI)       760
18 (FFPR)       36    'EE Text
3 (ORDA)         3
2 (OCAD)      4000    Set Gr = ActDr.Graphics.AddText(Trim(Str(EE1(i))), (Xsm(i) + X) / 2#,
0 ()             0      (Ysm(i) + Y) / 2# + TextOffSetConst, 0#, TextHeightConst, 0, 0, 0, 2)
2 (OCIS)      1226    Gr.Properties("Layer") = ErvenLayer
4 (OADC)        16    Gr.Properties("PenColor") = -3
4 (ODDC)        16
2 (ACFS)       836
2 (ORDI)       760
1 (ORDE)       200
2 (ACFI)        28
2 (ORDI)       760
2 (OADC)         8
                       Next i

TOTAL         55211 End Function


                    Function SaveSystem() As Integer
                     Save Sewer model
                    '
1 (FNLF)         1    For i = 0 To NumSel - 1
1 (ALFO)       300      Set Gr = ObjSel.Item(i) 'Returns part of a collection - graphic in the selection collection
1 (ORDI)       380
1 (ALFI)        19      GrSelType = Gr.TypeByValue
1 (C2FS)       389      If Gr.Layer.Name = TopoLayer Then
2 (ORDE)       400
1 (C2CI)         2        If (GrSelType = imsiPolyline) And (Gr.Vertices.Count = 2) Then
1 (C2FI)       174
1 (ORDE)       200
```

```
 1 (ACLI)          1     ErvenCount = 0
 1 (ALFI)         19     Gr.Vertices.UseWorldCS = True
 1 (ORDE)        200
 1 (ALFO)        300     Set Ver = Gr.Vertices.Item(0) ' Return the Vertices collection for current graphic
 1 (ORDE)        200
 1 (ORDI)        380     X1 = Ver.X
 2 (ALFD)        708     Y1 = Ver.Y
 1 (ALFO)        300     Set Ver = Gr.Vertices.Item(1)
 1 (ORDE)        200     X2 = Ver.X
 1 (ORDI)        380     Y2 = Ver.Y
 2 (ALFD)        708
 0 ()              0
 1 (FNPF)          2     Gr2 = PickEnt((X1 + X2) / 2#, (Y1 + Y2) / 2#, ErvenLayer, TempErvenStr)
 1 (ALLO)         22
 1 (FNC)           7
 4 (FNPR)          8
 2 (OADD)          8
 2 (ODDC)          8
 1 (REF)     29985.744
 1 (ALFS)        665     ErvenStr=Gr2.Name
 1 (OCSI)         69     ErvenCount = ErvenCount + val(Ervenstr)
 1 (OAII)          2
 1 (FNPF)          2     Gr2 = PickEnt((X1 + X2) / 2#, (Y1 + Y2) / 2#, InvertTextLayer, TempInvertStr)
 1 (ALLO)         22
 1 (FNC)           7
 4 (FNPR)          8
 2 (OADD)          8
 2 (ODDC)          8
 1 (REF)     29985.744
 1 (ALFS)        665     InvertString=Gr2.Name
 0 ()              0
 1 (OSI)          44     inpos = InStr(1, InvertString, "-")
 1 (C2CI)          2     If inpos = 0 Then
 1 (OSI)          44       inpos = InStr(1, InvertString, "%")
 1 (C2CI)          2       If inpos > 0 Then
 1 (FNG)           1 '
 0 ()              0       Else
 1 (C2LS)         15         SlopeString = InvertString
 0 ()              0       End If
 1 (C2CS)         50       SlopeFlagString = "S"
 1 (C2CS)         50       USInvertString = "0.00"
 1 (C2CS)         50       DSInvertString = "0.00"
 1 (FNG)           1     Else
 0 ()              0       '..
 0 ()              0     End If
 2 (ORDA)          2
18 (OTWS)        972     Print #1, "PIPE ";
 1 (OCIS)        613     Print #1, (PickEnt(X1, Y1, MhTextLayer, "USMH" + Str(i))).Name; " ";
 6 (OCDS)       4314     Print #1, Format(-X1, Fstr); " "; Format(-Y1, Fstr); " ";
 1 (FNPF)          2     Print #1, (PickEnt(X1, Y1, ElevationTextLayer, "0.00")).Name; " ";
 0 ()              0     Print #1, USInvertString; " ";
 4 (ALLO)         88     Print #1, (PickEnt((X2, Y2, MhTextLayer, "DSMH" + Str(i))).Name; " ";
 4 (FNC)          28     Print #1, DSInvertString; " ";
16 (FNPR)         32     Print #1, (PickEnt((X1 + X2) / 2#, (Y1 + Y2) / 2#, DiamTextLayer, TempDiamStr))).Name; " ";
 4 (ALFS)       2660     Print #1, Format(CalcLength(Gr), Fstr); " ";
 4 (REF)     119942.976  Print #1, "C ";
 0 ()              0     Print #1, ManConst; " ";
 2 (OCIS)       1226     Print #1, "E ";
 2 (OADD)          8     Print #1, InfilConst; " ";
 2 (ODDC)          8     Print #1, StormConst; " ";
 0 ()              0     Print #1, ErvenCount; " ";
 1 (REF)        4850     Print #1, "0 0 0 0 0 0 0 0 0 ";
 1 (FNC)           7     Print #1, SlopeString; " "; SlopeFlagString
 1 (FNPR)          2   End If
 1 (FNPF)          2 End If
                       Next i
                     ..
   TOTAL   201759.464 End Function

                       AUXILLARY FUNCTIONS

   125                 1/n
 5.456                 Assume number of pipes searched in standard radius:  -> more found in denser network

                       Function PickEnt(xClick#, yClick#, layName As String, defaultStr As String) As Graphic
                       Locate the nearest entity and return it as Graphic
 1 (ICLO)          2 Dim Vi As View
 1 (ICLO)          2 Dim PicRes As PickResult
 0 ()              0
 1 (ALFO)        300 Set Vi = IMSIGX.Application.ActiveDrawing.Views.Item(0)
 2 (ORDE)        400
 1 (ORDI)        380
 2 (ICLD)          6 Dim xView#, yView#
```

```
    1 (FFC)          212    Vi.WorldToView xClick#, yClick#, 0, xView#, yView#, 0
    6 (FFPR)          12
    1 (OCAD)        2000
    1 (FFPF)           6 Set PicRes = Vi.PickPoint(xView#, yView#, ApertureConst, False, False, True, False, False, False)
    1 (ALLO)          22
    1 (FFC)          212
    9 (FFPR)          18
    1 (OCAD)        2000
    1 (CFCI)          29 If PicRes.Count > 0 Then
    1 (FNPF)           2  Set PickEnt = GetClosestTextGraphic(layName, PicRes)
    1 (REF)    24343.744
    1 (ALLO)          22
    1 (FNC)            7
    1 (FNPF)           2
    2 (FNPR)           4
    0 ()               0 Else
    0 ()               0  Set PickEnt = Nothing
    0 ()               0 End If
    1 (ICLO)           2 Set PicRes = Nothing
    1 (ICLO)           2 Set Vi = Nothing

    TOTAL      29985.744 End Function


                         Function GetClosestTextGraphic(layName As String, PicRes As PickResult) As Graphic
                         Locate the closest text graphic
    1 (ICLO)           2  Dim ClosestGraphic As New Graphic
    1 (ICLD)           3  Dim ClosestDist As Double
    1 (ICLO)           2  Dim GrSel As Graphic
    1 (ICLD)           3  Dim Xp As Double
    1 (ICLD)           3  Dim Yp As Double
    1 (ICLD)           3  Dim xyDist As Double
    1 (ICLI)           1  Dim i As Integer
    0 ()               0
    1 (ICLD)           3  ClosestDist = 10000000000#
    0 ()               0
5.456 (FNLF)       5.456   For i = 0 To PicRes.Count - 1
5.456 (ACFI)      76.384
5.456 (ALFO)      1636.8     Set GrSel = PicRes.Item(i).Graphic ' gets selected graphic
5.456 (ORDE)      1091.2
5.456 (ORDI)      2073.28
5.456 (C2FS)      2122.384    If GrSel.Layer.Name = layName Then
5.456 (ORDE)      1091.2
5.456 (ALFD)      1931.424       xyDist = PicRes.Item(i).Distance
5.456 (ORDE)      1091.2
5.456 (ORDI)      2073.28
5.456 (OCAD)      10912
5.456 (C2LD)      27.28          If xyDist < ClosestDist Then
5.456 (ALLD)      21.824           ClosestDist = xyDist
5.456 (ALLO)      120.032          Set ClosestGraphic = GrSel
    0 ()               0         End If
    0 ()               0       End If
    1 (FNLN)           2     Next i
    0 ()               0
    2 (ALLO)          44  Set GetClosestTextGraphic = ClosestGraphic
    1 (ICLO)           2  Set GrSel = Nothing
    1 (ICLO)           2  Set ClosestGraphic = Nothing

    TOTAL      24343.744 End Function


   27 num:Avg              Sub Parse_input(stg As String, Res() As Variant, ByVal num As Integer, ier As Integer)
                         Parses one line of input from text file with 27 columns
    1 (ICLI)           1  Const maxcount = 42
    0 ()               0
    1 (ICLI)           1  Dim lenstr As Integer
    0 ()               0
   27 (FNLF)          27  For i% = 1 To num
   27 (ORDA)          27
   27 (ICLS)         135   Res(i%) = Empty
   27 (FNLN)          54  Next i%
    0 ()               0  'Too many fields
    1 (C2CI)           2  If num > maxcount Then
    0 ()               0  '..
    0 ()               0  End If
    1 (OSL)            7  lenstr = Len(stg)
    0 ()               0  'check for blank string
    1 (C2CI)           2  If lenstr = 0 Then
    0 ()               0  '..
    0 ()               0  End If
    0 ()               0  'skip initial blanks
    1 (ICLI)           1  icount% = 0
    1 (ICLI)           1  It% = 1
```

```
 0 ()                    0  'Main loop
 1 (FNLF)                1  Do
 0 ()                    0    'Skip over leading blanks - only once typical
27 (FNLF)               27    For j% = lt% To lenstr
27 (OSM)              8370      If (Mid$(stg, j%, 1) <> " ") Then
27 (C2LS)              405
27 (ALLI)               54        lt% = j%
27 (FNG)                27        Exit For
 0 ()                    0      End If
 0 (FNLN)                0    Next j%
27 (OAIC)               27    icount% = icount% + 1
27 (C2CI)               54    If icount% > num Then
 0 ()                    0    '  Exit Sub
 0 ()                    0    End If
27 (OSI)              1188    rt% = InStr(lt%, stg, " ") - 1
27 (OSIC)               27
27 (ACLI)               27
27 (C2CI)               54    If rt% < 0 Then rt% = lenstr
27 (ACLI)               27
27 (OSM)              8370    tmpstr$ = Mid$(stg, lt%, (rt% - lt% + 1))
27 (OAIC)               27
27 (OSII)               54
27 (ORDA)               27
27 (ALLS)              324    Res(icount%) = tmpstr$
27 (OAIC)               27    lt% = rt% + 1
27 (ALLI)               54
27 (FNLN)               54  Loop Until lt% >= lenstr
 1 (ICLI)                1  ier = 0

  TOTAL              19484 End Sub
```

Function CalcLength(Gr As Graphic) As Double

Calculates the length of a vertex

```
 1 (ICLD)                3 Dim a As Double
 1 (ICLO)                2 Dim Vertex1 As Vertex
 1 (ICLO)                2 Dim Vertex2 As Vertex
 1 (ALFI)               19   Gr.Vertices.UseWorldCS = True
 1 (ORDE)              200
 2 (ALFO)              600   Set Vertex1 = Gr.Vertices.Item(0)
 2 (ORDE)              400   Set Vertex2 = Gr.Vertices.Item(1)
 2 (ORDI)              760
 8 (ALFD)             2832   a = (Vertex1.X - Vertex2.X) * (Vertex1.X - Vertex2.X) + _
 4 (OSDD)               16      (Vertex1.Y - Vertex2.Y) * (Vertex1.Y - Vertex2.Y)
 2 (OMDD)                8   CalcLength = Math.Sqr(a)
 1 (OADD)                4
 1 (OMS)                 4

  TOTAL               4850 End Function
```

SEWSAN.EXE

Sub READNET()

Load Sewer model

```
..

 1 (CFCI)               29   While (Left$(Typestr(i), 5) <> "BOTMH") And (Not EOF(1))
 1 (OSM)               310 ..
 1 (C2CS)               50
 1 (OAIC)                1     i = i + 1
 1 (C2CI)                2     If i > (Max_Pipes + 1) Then
 1 (FNG)                 1 ..
 0 ()                    0     End If
 1 (OTRS)              370     Line Input #1, stg
 1 (REF)             20149     Call Parse_input(stg, result(), 28, ier)
 1 (C2CI)                2     If ier Then
 1 (FNG)                 1 ..
 1 (ORDA)                1     Else
 1 (ALFS)              665       typeStr(i) = UCase$(CStr(result(1)))
```

```
 1 (OSM)          310     If Left$(Typestr(i), 5) <> "BOTMH" Then
 1 (C2CS)          50 ...
16 (ORDA)          16
 8 (ALFS)        5320      Smhstr(i) = CStr(result(2))
22 (OCSD)        2596      Ysm(i) = Val(result(3))
22 (ALLD)          88      Xsm(i) = Val(result(4))
 4 (ODDC)          16      Glsm(i) = Val(result(5))
 0 ()               0      Ism(i) = Val(result(6))
 0 ()               0      Emhstr(i) = CStr(result(7))
 0 ()               0      Iem(i) = Val(result(8))
 0 ()               0      Diam(i) = Val(result(9)) / 1000
 0 ()               0      Lenn(i) = Val(result(10))
 0 ()               0      CLstr(i) = CStr(result(11))
 0 ()               0      CLstr(i) = UCase$(CLstr(i))
 0 ()               0      Man(i) = Val(result(12))
 0 ()               0      Exstr(i) = CStr(result(13))
 0 ()               0      Exstr(i) = UCase$(Exstr(i))
 0 ()               0      Infil(i) = Val(result(14)) / 1000 / 60
30 (ORDA)          30      Typ(i) = Val(result(15))
 1 (FNLF)           1      For j = 1 To 10
10 (OAIC)          10        Ee(i, j) = Val(result(15 + j))
 1 (FNLN)           2      Next j
 7 (ORDA)           7      Slope(i) = Val(result(26)) / 100
 0 ()               0      ISstr(i) = CStr(result(27))
 0 ()               0      ISstr(i) = UCase$(ISstr(i))
 1 (C2CS)          50      If ISstr(i) = "  " Then
 1 (ACLS)         222          ISstr(i) = "I"
 0 ()               0      End If
 0 ()               0      Addlenn(i) = Val(result(28))
 0 ()               0     End If
 0 ()               0    End If
 1 (FNG)            1    Wend
                       ..
   TOTAL        78720  End Sub


                       Sub SAVE_NET()
                        Save Sewer model
 1 (FNLF)           1  ..
 1 (ICLS)           5  For i = 1 To Npipes
 1 (ALLS)          12    fileline = ""
 1 (ORDA)           1    tmpstr = Typestr(i)
 1 (OSA)          524    fileline = fileline + tmpstr
 1 (ALLS)          12    tmpstr = Smhstr(i)
 1 (ORDA)           1
 8 (OSA)         4192    fileline = fileline + " " + tmpstr
 4 (OCDS)        2876    tmpstr = Format$(Ysm(i), "######0.000")
 4 (ORDA)           4
 4 (ALLS)          48    fileline = fileline + " " + tmpstr
 0 ()               0    tmpstr = Format$(Xsm(i), "#######0.000")
 0 ()               0    fileline = fileline + " " + tmpstr
 0 ()               0    tmpstr = Format$(Glsm(i), "###0.000")
 0 ()               0    fileline = fileline + " " + tmpstr
 0 ()               0    tmpstr = Format$(Ism(i), "###0.000")
 0 ()               0
 2 (OSA)         1048    fileline = fileline + " " + tmpstr
 0 ()               0
 1 (ALLS)          12    tmpstr = Emhstr(i)
 0 ()               0
10 (OSA)         5240    fileline = fileline + " " + tmpstr
 5 (OCDS)        3595    tmpstr = Format$(Iem(i), "###0.000")
 5 (ORDA)           5    fileline = fileline + " " + tmpstr
 5 (ALLS)          60    tmpstr = Format$(Diam(i) * 1000, "####")
 0 ()               0    fileline = fileline + " " + tmpstr
 0 ()               0    tmpstr = Format$(Lenn(i), "###0.000")
 0 ()               0    fileline = fileline + " " + tmpstr
 1 (ALLS)          12    tmpstr = CLstr(i)
 1 (ORDA)           1    fileline = fileline + " " + tmpstr
 0 ()               0    tmpstr = Format$(Man(i), "#0.000")
 0 ()               0    fileline = fileline + " " + tmpstr
 1 (ALLS)          12    tmpstr = Exstr(i)
 1 (ORDA)           1    fileline = fileline + " " + tmpstr
 0 ()               0    tmpstr = Format$(Infil(i) * 1000 * 60, "#0.000")
 0 ()               0    fileline = fileline + " " + tmpstr
 1 (ALLI)           2    tmpstr = Format$(Typ(i), "#0")
 1 (ORDA)           1    fileline = fileline + " " + tmpstr
 2 (OSA)         1048
10 (FNLF)          10    For j = 1 To 10
10 (OCDS)        7190      tmpstr = Format$(Ee(i, j), "##0.00")
10 (OSA)         5240      fileline = fileline + " " + tmpstr
20 (ORDA)          20
10 (ALLS)         120
```

```
10 (FNLN)          20  Next j
 0 ()               0  tmpstr = Format$(Slope(i) * 100, "##0.000")
 6 (OSA)         3144  fileline = fileline + " " + tmpstr
 2 (OCDS)        1438  tmpstr = lSstr(i)
 3 (ORDA)           3  fileline = fileline + " " + tmpstr
 3 (ALLS)          36  tmpstr = Format$(Addlenn(i), "###0.000")
 0 ()               0  fileline = fileline + " " + tmpstr
 0 ()               0
 1 (OTWS)          54 Print #1, fileline
 1 (FNLN)           2 Next i
                    ..
  TOTAL         35988 End Sub
```

GLSDTM.EXE

function Readnodes(FName:string):boolean;

Read the points to be interrmpolated

```
                    //Type
                    //  CoordType=packed record
                    //            X,Y,Z:Double;
                    //          end;
                    // first find number of valid lines to read.
                    //Data type of Nodes[i]

                     repeat
 1 (OAIC)            1   inc(count);
 1 (OBR)            27   BlockRead(filvar,coord,1);
 1 (C2CD)            4    if coord.z=-999999 then
 1 (ORDL)            2    begin
 1 (OSIC)            1      dec(count);
 1 (FNG)             1      break;
 0 ()                0    end;
 1 (FNG)             1  until ((EOF(Filvar)) or (count = maxlinks));
 1 (CFCI)           29
 1 (C2CI)            2
 0 ()                0    repeat
 1 (OAIC)            1      inc(count);
 1 (OBR)            27      BlockRead(filvar,coord,1);
 2 (ALLD)            8      Nodes[count].x:=Coord.x;
 4 (ORDL)            8      Nodes[count].y:=Coord.y;
 2 (ORDA)            2
 1 (FNG)             1    until (EOF(Filvar) or (count = TotalNodes));
 1 (CFCI)           29 //...
 1 (C2LI)            3

  TOTAL            147 end;
```

function SaveNodes(FName:string):boolean;

```
                    writes the updated nodes
                    //..
 1 (FNG)            1  for i := 1 to TotalNodes do
 0 ()               0  begin
 3 (ALLD)          12    Coord.X:=Nodes[i].X;
 6 (ORDL)          12    Coord.Y:=Nodes[i].Y;
 3 (ORDA)           3    Coord.Z:=Nodes[i].Z;
 1 (OBW)           27    BlockWrite(Filvar,Coord,1);
 1 (FNLN)           2  end;
                     //..
  TOTAL            57 end;
```

Appendix E

MC Analysis Results

MODEL-CENTRED DESIGN

			PR1	PR2	PR3	PR4
Type	*Code*	*Description*	**104**	**682**	**2934**	**6721**
DoE [s]	TFD1	Show Direction Definition	1.2	12.1	48.1	494.8
UIC	*TFD2*	*Swap Direction Definition*	*104*	*682*	*2934*	*6721*
UIC	*TMC*	*Topolology MH Correction*	*125*	*818*	*3521*	*8065*
			104	**682**	**2934**	**6721**
DoE [s]	TCE	Topology Model Coord Exp	0.1	4.4	55	283.6
Eq. BIC[s]	*TCE*	*Topology Model Coord Exp*	*0.327*	*2.143*	*9.221*	*21.124*
DoE [s]	ECI	Elevation Model Coord Imp	0.2	0.5.	1.0	1.5
Eq. BIC[s]	*ECI*	*Elevation Model Coord Imp*	*0.015*	*0.1*	*0.431*	*0.988*
DoE [s]	EEI	Elevation Model Elev Exp	0.2	0.5	1	1.5
Eq. BIC[s]	*EEI*	*Elevation Model Elev Exp*	*0.006*	*0.038*	*0.167*	*0.383*
DoE [s]	TEI	Topology Model Elev Imp	1.5	42.1	293.4	2321.8
Eq. BIC[s]	*TEI*	*Topology Model Elev Imp*	*1.7*	*25.2*	*344*	*1693*
DoE [s]	UE	**TOTAL: Update Elevations**	**2.0**	**47.0**	**350.4**	**2608.4**
Eq. BIC[s]	*UE*	***TOTAL: Update Elevations***	***2***	***27.5***	***353***	***1715***
			104	**682**	**2934**	**6721**
DoE [s]	TSU	Topology Slope Update	2.9	98.3	546	3550
UIC	*TPU*	*Topology Parcel Update*	*63*	*409*	*1760*	*4031*
			104	**682**	**2934**	**6721**
DoE [s]	TME2	Second Topology Model Exp	2.3	146.8	1507.9	9017.3
Eq. BIC[s]	*TME2*	*Topology Model Exp*	*8.2*	*137.6*	*2003.0*	*10024.0*
DoE [s]	HMI	Hydraulic Model Import	7.5	45.0	210.0	1273.5
Eq. BIC[s]	*HMI*	*Hydraulic Model Import*	*8.2*	*53.7*	*231.0*	*529.1*
DoE [s]	BM	**TOTAL: Build Model from CAD**	**9.8**	**191.8**	**1717.9**	**10290.8**
Eq. BIC[s]	*BM*	***TOTAL: Build Model from CAD***	***16.3***	***191.3***	***2233***	***10552***
			104	**682**	**2934**	**6721**
DoE [s]	HME	Hydraulic Model Export	3.8	22.5	90.0	210.0
Eq. BIC[s]	*HME*	*Hydraulic Model Export*	*3.7*	*24.5*	*105.6*	*241.9*
DoE [s]	TMI2	Second Topology Model Imp	5.2	40.8	132.8	741.0
Eq. BIC[s]	*TMI2*	*Second Topology Model Imp*	*5.2*	*37.7*	*221.5*	*736.4*
Time [s]	DM	**TOTAL: Draw Hydraulic Model**	**9.0**	**63.3**	**222.8**	**951.0**
Eq. BIC[s]	*DM*	***TOTAL: Draw Hydraulic Model***	***8.9***	***62.2***	***327.0***	***978.2***
			104	**682**	**2934**	**6721**
DoE [s]	ALL	**TOTAL: Build Topo+Model+Draw**	**20.8**	**302.1**	**2291.1**	**13850.2**
Eq. BIC[s]	*ALL*	***TOTAL: Build Topo+Model+Draw***	**27.2**	**281.0**	**2367.8**	**13245.2**
UIC	*ALL*	***TOTAL: Mix***	***292***	***1909***	***8215***	***18817***

Notes

1 HMI and HME times are scaled to bring on par with instruction count by multiplying with 15.

Table E.1: Spreadsheet of MC performance evaluation

Appendix F

Source code samples from AC system

F.1 TopoClasses.POTreeTraversal Class

```
package TopoClasses;

import java.util.*;
import Application.*;
import CoreClasses.*;

  /**
   * Implementation of a Post-order tree traversal iterator
   * Builds a stack of vertices, and provide an iterator to this stack
   */

  public class POTreeTraversal implements Iterator {
  /**
   * Maintain a stack of vertices
   */
     private Stack vertexStack;
  /**
   * The root identifier
   */
     private String rootId;

  /**
   * Public parameterless constructor not used
   */
     public POTreeTraversal(){}

  /**
   * Public constructor
   * Initializes the vertex stack
```

```
   * Reset the visited state on all vertices by calling Util#unVisitVertices(DataModel dm)
   * Add the rootid as first element of queue, and set the visited prop-
erty of this item.
   * @param rootId the start vertex for the traversal
   * @param dm a reference to the DataModel
   *
   */
     public POTreeTraversal(String rootId, DataModel dm) {
       this.rootId = rootId;
       vertexStack = new Stack();
       Util.unVisitVertices(dm);
       if (rootId != null) {
          vertexStack.push(rootId);
          Util.vertex(rootId).visit();
       }
     }

  /**
   * Public implementation of hasNext() method.
   * @return true if the vertex stack is not empty
   *
   */
     public boolean hasNext() {
       if (vertexStack.isEmpty()) return false;
       return true;
     }

  /**
   * Public implementation of next() method.
   * Add fromVertices() to stack, as long as it is unvisited.
   * Include push from stack for post-order processing.
   * @return the next vertex in the virtual iterator
   *
   */
     public Object next() {
       boolean continueClimb;
       String currentVertexId = (String) vertexStack.peek();

       do{
         continueClimb=false;
         Vertex currentVertex = Util.vertex(currentVertexId);
         Iterator e = currentVertex.fromVertices();
         while (e.hasNext()){
           Vertex adjacentVertex = Util.vertex(e.next());
           if (!adjacentVertex.isVisited()) {
             vertexStack.push(adjacentVertex.getId());
             adjacentVertex.visit();

             currentVertexId =adjacentVertex.getId();
             continueClimb=true;
             break;
           }
         }
       }while (continueClimb);
```

```
        vertexStack.pop();
        return currentVertexId;
    }

    public void remove(){

    }
}
```

F.2 TopoClasses.DFTreeTraversal Class

```
package TopoClasses;

import java.util.*;
import Application.*;
import CoreClasses.DataModel;

  /**
   * Implementation of a depth-first tree traversal iterator
   * Builds a stack of vertices, and provide an iterator to this stack
   */

  public class DFTreeTraversal implements Iterator {
  /**
   * Maintain a stack of vertices
   */
     private Stack vertexStack;
  /**
   * The root identifier
   */
     private String rootId;

  /**
   * Public parameterless constructor not used
   */
     public DFTreeTraversal(){}

  /**
   * Public constructor
   * Initializes the vertex stack
   * Reset the visited state on all vertices by calling Util#unVisitVertices(DataModel dm)
   * Add the rootid as first element of queue, and set the visited prop-
erty of this item.
   * @param rootId the start vertex for the traversal
   * @param dm a reference to the DataModel
   *
   */
     public DFTreeTraversal(String rootId, DataModel dm) {
       this.rootId = rootId;
       vertexStack = new Stack();
       Util.unVisitVertices(dm);
       if (rootId != null) {
          vertexStack.push(rootId);
          Util.vertex(rootId).visit();
       }
     }

  /**
   * Public implementation of hasNext() method.
   * @return true if the vertex stack is not empty
   *
   */
     public boolean hasNext() {
```

```
        if (vertexStack.isEmpty()) return false;
        return true;
      }

  /**
   * Public implementation of next() method.
   * Add fromVertices() to stack, as long as it is unvisited.
   * @return the next vertex in the virtual iterator
   *
   */
      public Object next() {
        String currentVertexId = (String) vertexStack.pop();
        Vertex currentVertex = Util.vertex(currentVertexId);

        Iterator e = currentVertex.fromVertices();
        while (e.hasNext()){
          Vertex adjacentVertex = Util.vertex((String) e.next());
          if (!adjacentVertex.isVisited()) {
            vertexStack.push(adjacentVertex.getId());
            adjacentVertex.visit();
          }
        }
        return currentVertexId;
      }

      public void remove(){

      }
}
```

F.3 TopoClasses.BFTreeTraversal Class

```
package TopoClasses;

import java.util.*;
import Application.*;
import CoreClasses.*;

  /**
   * Implementation of a breadth-first tree traversal iterator
   * Builds a vector queue of vertices, and provide an iterator to this vector
   */
  public class BFTreeTraversal implements Iterator {
  /**
   * A Vector queue build the list of vertices
   */
     private Vector vertexQueue;

  /**
   * The root identifier
   */
     private String rootId;

  /**
   * Public parameterless constructor not used
   */
     public BFTreeTraversal(){}

  /**
   * Public constructor
   * Initializes the vector queue
   * Reset the visited state on all vertices by calling Util#unVisitVertices(DataModel dm)
   * Add the rootid as first element of queue, and set the visited prop-
erty of this item.
   * @param rootId the start vertex for the traversal
   * @param dm a reference to the DataModel
   *
   */
     public BFTreeTraversal(String rootId, DataModel dm) {
       this.rootId = rootId;
       vertexQueue = new Vector();
       Util.unVisitVertices(dm);
       if (rootId != null) {
          vertexQueue.addElement(rootId);
          Util.vertex(rootId).visit();
       }
     }

  /**
   * Public implementation of hasNext() method.
   * @return true if the vertex queue is not empty
   *
   */
```

```
    public boolean hasNext() {
      if (vertexQueue.size() == 0) return false;
      return true;
    }


  /**
   * Public implementation of next() method.
   * Add fromVertices() to queue, as long as it is unvisited.
   * @return the next vertex in the virtual iterator
   *
   */
    public Object next() {
      String currentVertexId = (String) vertexQueue.elementAt(0);
      Vertex currentVertex = Util.vertex(currentVertexId);

      vertexQueue.removeElementAt(0);
      Iterator e = currentVertex.fromVertices();
      while (e.hasNext()){
        Vertex adjacentVertex = Util.vertex((String) e.next());
        if (!adjacentVertex.isVisited()) {
          vertexQueue.addElement(adjacentVertex.getId());
          adjacentVertex.visit();
        }
      }
      return currentVertexId;
    }

    public void remove(){

    }
}
```

F.4 TopoClasses.AgeComparator Class

```
package TopoClasses;

import java.util.*;

  /**
   * A special implementation of the Comparator Interface
   */
public class AgeComparator implements Comparator{

  /**
  * Parameterless constructor - not used.
  */
   public AgeComparator(){}

  /**
  * Implementation of the compare method
  * Results in sorting from small to large
  */
   public int compare(Object o1, Object o2){
     Integer age1,age2;
     age1=new Integer( ((Vertex) o1).age);
     age2=new Integer( ((Vertex) o2).age);
     return (age1.compareTo(age2));
   }

  /**
  * Implementation of the equals method
  */
  public boolean equals(Object obj){
      return super.equals(obj);
  }

}
```

F.5 TopoClasses.Util Class

```
package TopoClasses;
import java.util.*;
import CoreClasses.*;
import GeomClasses.*;
import Application.*;

  /**
   * Utility class for the Topology Package
   */
public class Util {


/**
   * Iterate over all manholes in the application, and reset visited state
   */
  static void unVisitVertices(DataModel dm){
    Iterator v = dm.getManholes();
    while (v.hasNext()){
      String tmpId = (String) v.next();
      Vertex tmpVertex = (Vertex) App.getObject(tmpId);
      tmpVertex.unvisit();
    }
  }

  /**
   * Iterate over all manholes in the application, and reset age to zero
   */
  private static void resetAge(DataModel dm){
    Iterator v = dm.getManholes();
    while (v.hasNext()){
      String tmpId = (String) v.next();
      Vertex tmpVertex = (Vertex) App.getObject(tmpId);
      tmpVertex.resetAge();
    }
  }


  /**
   * Calculate the age of all vertices in the tree.
   * Firstly reset the age to zero for all vertices.
   * Then switch Vertex#sorting off, since sorting requires age calcula-
tions, and age calculations requires POTreeTraversal (from leaves to root).
   * Then get a post-order traversal iterator (i.e. from leaves to root).
   * Then traverse, and increment age by one.
   * Untested for branching outEdges!
   * Finally, set Vertex#sorting on again.
   */
  public static void calcTreeAge(String rootId, DataModel dm){
    Vertex usVertex, dsVertex;
    Edge edge;
```

```
    resetAge(dm);
    Vertex.sorting=false;
    Iterator vertexIds = new POTreeTraversal(rootId,dm);
    while (vertexIds.hasNext()) {
       usVertex =(Vertex) App.getObject( (String) vertexIds.next());
//       if (usVertex.outDegree() > 0) {
         Iterator edgeIds = usVertex.outEdges();
         while (edgeIds.hasNext()) {
           edge = (Edge) App.getObject( (String) edgeIds.next());
           dsVertex = (Vertex) App.getObject(edge.getToId());
           dsVertex.incAge(usVertex.getAge());
         }
//     }
    }
    Vertex.sorting=true;
  }


  /**
   * Calculate the age in terms of distance (edge lengths) of all ver-
tices in the tree.
   * Firstly reset the age to zero for all vertices.
   * Then switch Vertex#sorting off, since sorting requires age calcula-
tions, and age calculations requires POTreeTraversal (from leaves to root).
   * Then get a post-order traversal iterator (i.e. from leaves to root).
   * Then traverse, and increment age by length of edge (in GeomClasses)
   * Untested for branching outEdges!
   * Finally, set Vertex#sorting on again.
   */
  public static void calcTreeDistance(String rootId, DataModel dm){
    Vertex usVertex, dsVertex;
    Link tmpLink;

    resetAge(dm);
    Vertex.sorting=false;
    Iterator vertexIds = new POTreeTraversal(rootId,dm);
    while (vertexIds.hasNext()) {
       usVertex =(Vertex) App.getObject( (String) vertexIds.next());
//       if (usVertex.outDegree() > 0) {
         Iterator edgeIds = usVertex.outEdges();
         while (edgeIds.hasNext()) {
           tmpLink = GeomClasses.Util.link((String) edgeIds.next());
           dsVertex = Util.vertex(tmpLink.getToId());
           dsVertex.incAge(usVertex.getAge(),
           (new Double(tmpLink.getLength().getLength())).intValue());
         }
//       }
    }
    Vertex.sorting=true;
  }

}
```

Appendix G

AC Instruction Count Evaluation

CALCULATION SHEET FOR OPERATION COUNT: APPLICATION-CENTRED
(Internal relation classes)

TOTAL: 682 PIPES //104//682//2934//6721

	One Pipe	n Pipes	
UE=	DATA MODEL / ELEVATION MODEL		
TCE/ECI/ EEI/TEI	374	255,068	*UpdateElevations()*
BMI=	CAD MODEL / DATA MODEL		
TME2/HMI	104,849	71,507,182	*buildModelFromCad()*
HME/TMI2	7,995	5,452,590	*drawModel()*
=DMI			

#	Code	Count	Source Code
			public void updateElevations() throws RemoteException{
			Transfer coordinates from Data Model to DTM Model
	1 ()	0	**Iterator mhIds = manholes.iterator();**
	1 (FNLF)	1	while (mhIds.hasNext()) {
	1 (ORDL)	2	
	1 (ALLO)	22	//> **boolean hasNext();**
	1 (FNC)	7	
	1 (ORDL)	2	
131	1 ()	0	**[UTILGE** Manhole mh = Util.manhole(mhIds.next());
	1 (ALLO)	22	//> **Object next();**
	1 (FNC)	7	
102	1 ()	0	**[MH]** //> **public static Manhole manhole(Object objectId) {**
	1 (FFPR)	2	return (Manhole) App.getObject((String) objectId);
	1 (FFPF)	6	
	1 (FNC)	7	
	1 (ORDL)	2	
	1 (FFPR)	2	
83	1 ()	0	**[GO]** //> **public static AppObject getObject(String objectId) {**
	1 (ALLO)	22	AppObject tmpObj = (AppObject)objectMap.get(objectId);
	1 (FNC)	7	
	1 (ORDL)	2	
	1 (FFPR)	2	
	1 (OJH)	50	//hash op
	1 (FNC)	7	mh.updateElevation();
	1 ()	0	//> **public void updateElevation(){**
	1 (FNC)	7	UtilDTM.UpdateElevation(id);
	1 (ORDL)	2	
	1 (FNPR)	2	//> **public static void UpdateElevation(String nodeID){**
	2 (FNC)	14	Node tmpNode = node(nodeID);
	1 (ORDL)	2	
	2 (FNPR)	4	//> **private static Node node(Object objectId) {**
	2 (ALLO)	44	return (Node) App.getObject((String) objectId);
	1 (FNPF)	2	
83	1 ()	83	**[GO]** //> **public static AppObject getObject(String objectId) {**
	2 (ALLD)	8	double tmpX = tmpNode.getX();
	2 (FNC)	14	//> **public double getX(){return x;}**
	2 (FNPF)	4	double tmpY = tmpNode.getY();
	1 ()	0	//> **public double getY(){return y;}**
	1 ()	0	/.....

```
1 (FNC)          7                                    tmpNode.setZ(tmpZ);
1 (FFPD)         3                                      //> public void setZ(double z){this.z=z;}
1 (ALLD)         4                                  }
1 (FNLN)         2                          }

  TOTAL        374

                                  public void buildModelFromCad() throws RemoteException{
                                  Build a hydraulic model from the CAD model
                                      //.....
3 (ORDL)         6                      if (ds.drawables[i].getDrawType()==DrawAble._drawLine){ //line
1 (ORDA)         1                          //>  public int getDrawType(){
1 (FNC)          7                              if (nrPoints==DrawableConverter.CIRCLE_SEGMENTS) return (_drawCircle);
1 (FNPF)         2                              if (nrPoints==2) return (_drawLine);
1 (C2LI)         3                              if (nrPoints==1) return (_drawPoint);
3 (C2LI)         9                              return (_drawUndef);
1 (ORDL)         2                          }
1 (ALLD)         4      58 [GFC]        tmpX1=ds.drawables[i].getFromCoord().x;
3 (ORDL)         6
1 (ORDA)         1
1 (FNC)          7                          //> public Point3D getFromCoord(){
1 (C2CI)         2                              if (nrPoints<=2)
1 (OCO)         10                                  return (new Point3D(line[0]));
3 (FNPD)         9      23 [P3D]                    //>  public Point3D(float xx, float yy, float zz) {
3 (ALLD)        12                                        x = xx;
1 (FNPF)         2                                        y = yy;
0 ()             0                                        z = zz;
0 ()             0                                   }
1 (C2LI)         3                              if (nrPoints==DrawableConverter.CIRCLE_SEGMENTS) //circle
1 (ORDL)         2                              else
0 ()             0                          }
1 ()            58 [GFC]                tmpY1=ds.drawables[i].getFromCoord().y;
0 ()            58                          //> public Point3D getFromCoord(){
0 ()             0                      //Get To Node Coords
1 ()            58 [GFC]                tmpX2=ds.drawables[i].getToCoord().x;
0 ()             0                          //> public Point3D getFromCoord(){
1 ()            58 [GFC]                tmpY2=ds.drawables[i].getToCoord().y;
2 (OADC)         8                          //> public Point3D getFromCoord(){
2 (ODDC)         8                      tmpXc=(tmpX1+tmpX2)/2.0;
2 (ALLD)         8                      tmpYc=(tmpY1+tmpY2)/2.0;
2 (FNPD)         6                       nco = new NearestCADObject(tmpXc,tmpYc,_ervenLayer,"0");
1 (FNPI)         2
1 (FNPR)         2
1 (OCO)         10
1 (ALLO)        22
1 ()      1553.048 [NCO]
1 (ALLI)         2      84 [GI]          ervenCount=nco.getInt();
1 (ORDL)         2
1 (FNC)          7                        int getInt(){
1 (ALLI)         2                            tmpInt= Integer.parseInt(nearestString);
1 (OCSI)        69                           return tmpInt;
1 (FNPF)         2                       }
2 (FNPD)         6                      nco = new NearestCADObject(tmpXc,tmpYc,_invertTextLayer,"0.00 - 0.00");
1 (FNPI)         2
1 (FNPR)         2
1 (OCO)         10
1 (ALLO)        22
0 ()      1553.048 [NCO]                                                                    //Invert Strings
1 (ALLD)         4     495 [GP]          slopeval=nco.getPercentage();//%
1 (ORDL)         2                        //>    double getPercentage(){
1 (FNC)          7                             int inpos = nearestString.indexOf("%");
1 (OSI)         44                             try {
1 (ALLI)         2                              if (inpos>-1) {// dash found
1 (C2CI)         2                                tmpDouble= Double.parseDouble(nearestString.substring(0,inpos));
1 (OSM)        310                               }else
1 (OCSD)       118
1 (ALLD)         4                              } catch (Exception x) {
1 (FNPF)         2                              return tmpDouble;
```

```
        2 (FNPD)          6                          nco = new NearestCADObject(tmpX1,tmpY1,_mhTextLayer,"USMH"+Integer.toString(i));
        1 (FNPI)          2
        1 (FNPR)          2
        1 (OCO)          10
        1 (ALLO)         22
        1 (OCIS)        613
        1 (OSA)         100      100 [JavaStringAdd]
        1 ()       1553.048 [NCO]
        1 (ALLS)         12       23 [GS]           usMhText=nco.getString();
        1 (ORDL)          2                            //>    String getString(){
        1 (FNC)           7                                   return nearestString;
        1 (FNPF)          2                                 }
        2 (FNPD)          6                          nco = new NearestCADObject(tmpX2,tmpY2,_mhTextLayer,"DSMH"+Integer.toString(i));
        1 (FNPI)          2
        1 (FNPR)          2
        1 (OCO)          10
        1 (ALLO)         22
        1 (OCIS)        613
        1 (OSA)         100      100 [JavaStringAdd]
        1 ()       1553.048 [NCO]
        1 (ALLS)         23 [GS]                    dsMhText=nco.getString();
        2 (FNPD)          6                          nco = new NearestCADObject(tmpX1,tmpY1,_elevTextLayer,"0.00");
        1 (FNPI)          2
        1 (FNPR)          2
        1 (OCO)          10
        1 (ALLO)         22
        0 ()              0 [NCO]    N/A
        1 (ALLD)          4      137 [GD]           elevText=nco.getDouble();
        1 (ORDL)          2                            //>    double getDouble(){
        1 (FNC)           7                                      tmpDouble= Double.parseDouble(nearestString);
        1 (OCSD)        118                                    return tmpDouble;
        1 (ALLD)          4
        1 (FNPF)          2                                 }
        2 (FNPD)          6                    [NCO] nco = new NearestCADObject(tmpXc,tmpYc,_diamTextLayer,"199.99");
        1 (FNPI)          2
        1 (FNPR)          2
        1 (OCO)          10
        1 (ALLO)         22
        1 ()       1553.048 [NCO]
        1 ()            137 [GD]                    diamText=nco.getDouble();
        1 (FFC)         212                         sewerModel.addLine(usMhText,tmpX1,tmpY1,tmpX2,tmpY2,elevText,usInvert,
        1 (FFPR)          2                               dsMhText,dsInvert,diamText,0.0,0.0,slopeval,1,
       14 (FFPD)         42                               ervenCount,0,0,0,0,0.0,0.0);
        6 (FFPI)         12
        1 ()            93895 [SMAL]               //->                    //Cross Platform Call
                              }
TOTAL              104849.24 [BM]

                                   LENGTHY AUXILLARY FUNCTIONS

      125                          1/n
    5.456                          Assume number of pipes searched in standard radius:  -> more found in denser network
                                   This is to ensure compatibility with the assumptions made for the model-centred design.

                                   class NearestObject {
        1 (FNC)           7        1 //Constru    NearestCADObject(double xPos, double yPos, String layerT, String defaultString){
        1 (ALLI)          2        1                DrawAble.resetSnap();                    //reset snap state
        1 (FNLF)      5.456    5.456                for (j = 0;  j < ViewerExtension.ds.nrDrawables;  j++) {
        1 (C2LI)     16.368    5.456
        2 (ORDL)     21.824    5.456
        1 (C2CI)     10.912    5.456                if (ViewerExtension.ds.drawables[j].getDrawType()==DrawAble._drawText)
        4 (ORDL)     43.648    5.456
        1 (ORDA)      5.456    5.456                   //> public int getDrawType(){
        1 (FNPF)     10.912    5.456                        if (nrPoints==2) return (_drawLine); //most cases
        1 (C2LI)     16.368    5.456                        if (nrPoints==DrawableConverter.CIRCLE_SEGMENTS) return (_drawCircle);
        0 (ORDL)          0    5.456                        if (nrPoints==1) return (_drawPoint);
        1 (FNC)      38.192    5.456                        return (_drawUndef);
```

```
        1 (C2CS)      272.8    5.456                 if (ViewerExtension.ds.drawables[j].layer.getName().equals(layerT))
        5 (ORDL)      54.56    5.456                '//most cases not:
        1 (ORDA)      5.456    5.456
        1 (FNC)      38.192    5.456
        1 (FNC)      38.192    5.456                   ViewerExtension.ds.drawables[j].getNearestEntity((new Float(xPos)).floatValue(),
        3 (ORDL)     32.736    5.456                                                    (new Float(yPos)).floatValue());
        1 (ORDA)      5.456    5.456
        2 (OCO)      109.12    5.456
        2 (FNC)      76.384    5.456
        2 (FNPF)     21.824    5.456
        2 (FNPD)     32.736    5.456                  //>  public DrawAble getNearestEntity(float xPos, float yPos) {
        2 (ALLO)     240.064   5.456                            p3=dxftextTmp.getPosition();
        1 (ORDL)     10.912    5.456
        1 (FNC)      38.192    5.456
        1 (ALLD)     21.824    5.456                            dist = (p3.x-xPos)*(p3.x-xPos)+
        1 (OADD)     21.824    5.456                                   (p3.y-yPos)*(p3.y-yPos);
        2 (OMDD)     43.648    5.456
        4 (OSDD)     87.296    5.456
        4 (ORDL)     43.648    5.456
        1 (C2LD)     27.28     5.456                            if (dist < snapMinFloatDistance) {
        1 (ALLD)     21.824    5.456                                snapMinFloatDistance=dist;
        1 (ALLO)     120.032   5.456                                snapEntity=this;
        1 (FNPF)     10.912    5.456                       return (this); }

TOTAL               1553.048 [NCO]

                                                  public String addLine(String usMhText, double xfC, double yfC, double xtC, double ytC,
                                                            double elev, double usInvert,
                                                            String dsMhText,  double dsInvert, double diam,
                                                            double leng,  double manning, double slope, int newpipe,
                                                            int ervenCountA, int ervenCountB, int ervenCountC, int ervenCountD,
                                                            int ervenCountE,
                                                            double PointSource1, double PointSource2)  throws RemoteException {

        1 (C2CO)          2                           if (dsMhText==null){
        0 ()              0                           else{
        1 (ALLO)         22    44908 [CM]               usManhole=Manhole.construct( usMhText,xfC, yfC, elev, this);
        1 (FNC)           7                                  //>  static public Manhole construct(String id, double x,double y,double z, DataModel dm) {
        2 (FNPR)          4
        3 (FNPD)          9
        1 (ORDL)          2
        1 ()             83 [GO]                                 if (App.containsObject(id)) {   //only for numbered systems
        0 ()              0                                      else{       //new: search based on coords for blank systems
        1 (ALLO)         22                                          tmpMHid=null;
        2 (C2CD)          8                                          if ((x!=0)&&(y!=0)){
        1 (ALLS)         12                                           tmpMHid = findNode(x,y,dm);
        2 (FNPD)          6
        1 (FNPF)          2
        1 (FNC)           7                                            //>    public static String findNode(double x, double y, DataModel dm){
        2 (ALLO)         44                                             Iterator v = dm.getManholes();
        1 (ORDL)          2
        1 (FNC)           7
        1 (FNPF)          2
        1 (ALLO)       7502      341 AVG 50%?                          while (v.hasNext()){
        1 (FNC)        2387      341
        1 (FNLF)        341      341
        1 (ALLS)       4092      341                                    tmpId = (String) v.next();
        1 (FNC)        2387      341
        1 (ORDL)        682      341
        1 (ALLO)       7502      341
        1 ()              0      341 [GO]                               Node tmpVertex = (Node) App.getObject(tmpId);
        2 (FNC)        4774      341                                    if ((Math.abs(tmpVertex.getX()-x) < Util._deltaX) &&
        2 (FNPF)       1364      341                                      (Math.abs(tmpVertex.getY()-y) < Util._deltaY)) {
        2 (ALLD)       2728      341                                       resId=tmpId;
        2 (OSDD)       2728      341                                       break;
        2 (C2CD)       2728      341                                    }
        6 (ORDL)       4092      341
```

```
1 (ALLS)      12      1
1 (FNG)        1      1
1 (FNLN)     682    341                                                                    }
1 (FNPF)       2      1                                                                    return (resId);
0 ()           0                                                                         }
1 (C2CO)       2                                                                    if (tmpMHid==null)
1 (ALLO)      22                                                                     tmpMH = new Manhole(id,x,y,z);
1 (FNPR)       2
3 (FNPD)       9
1 (OCO)       10
1 (FNPR)      12      6                                                              //>   private Manhole(String id,double x, double y, double z){super (id,x,y,z);}
3 (FNPD)      54      6                                                                    //>   public Manhole(String id,double x, double y, double z){super (id,x,y,z);}
1 (FNC)       42      6                                                                         //>   public Manhole(String id,double x, double y, double z){super (id,x,y,z);}
0 ()           0                                                                              //>   public Manhole(String id,double x, double y, double z){super (id,x,y,z);}
0 ()           0                                                                                   //>   public Manhole(String id,double x, double y, double z){super (id,x,y,z);}
0 ()           0                                                                                        //>   public Node(String id,double x,double y,double z){
0 ()           0                                                                                          super(id,x,y,z);}
0 ()           0                                                                                             //>   public Node(String id,double x,double y,double z){
1 (FNPR)       4      2                                                                                        super(id);
0 ()           0                                                                                              //->   public Vertex(String id) {  //can construct unconnected vertex
0 ()           0                                                                                                   super(id);
0 ()           0    237 [APPO]                                                                                      //->   public AppObject(String id) {
1 (CFCO)       2                                                                                                        if (id==null) {id=App.getAutoId();}
1 (FNC)        7
1 (ORDL)       2
1 (FNPF)       2
1 (ALLS)      12                                                                                                                  this.id = id;
1 ()         200          //Getting unused id
1 (ALLS)      12
3 (ALLD)      12                                                                                                  this.x=x;
0 ()           0                                                                                                  this.y=y;
0 ()           0                                                                                                  this.z=z;
2 (ALLO)      44                                                                                                  inFlow = new Hydrograph(null);
2 (OCO)       20                                                                                                  outFlow = new Hydrograph(null);
2 (ALLO)      44                                                                                                            toEdges = new HashSet();
2 (OCO)       20                                                                                                            fromEdges = new HashSet();
0 ()           0                                                                    else
1 ()         131 [UTILGET]                                                           tmpMH = Util.manhole(tmpMHid);
0 ()           0                                                                }
2 (ORDL)       4     75 [MA]                                                   manholes.add(usManhole.getId());
1 (FNC)        7
1 (ALLS)      12
1 (FNPF)       2
1 (OJH)       50
1 ()       44908 [CM]     //second     dsManhole = Manhole.construct(dsMhText,xtC,ytC, 0.0, this);
1 ()          75 [MA]                 manholes.add(dsManhole.getId());
1 (ALLO)      22                      pipe = new Pipe(usManhole.getId(),dsManhole.getId(),null);
3 (FNPR)       6
2 (ORDL)       4
1 (OCO)       10
2 (FNC)       14
2 (FNPF)       4
2 (ALLO)      44
3 (FNPR)      30      5              //>      public Pipe(String fromId, String toId, String id ) {
1 (FNC)       35      5                           super(fromId,toId,id);
0 ()           0                           //>      public Pipe(String fromId, String toId, String id ) {
0 ()           0                                      super(fromId,toId,id);
0 ()           0                                 //> public Pipe(String fromId, String toId, String id ) {
0 ()           0                                           super(fromId,toId,id);
1 (ACLD)       2                                           oldDiameter=0.0;
0 ()           0                                      //> public Pipe(String fromId, String toId, String id ) {
0 ()           0                                           super(fromId,toId,id);
2 (ALLO)      44                                           bs = new BitSet();
2 (OCO)       20                                           full = new Results();
0 ()           0      8 [RES]                               //> public Results() {
1 (ACLD)       2                                                 capRes=0.0;
1 (ACLD)       2                                                 flowRes=0.0;
```

```
1 (ACLD)      2                                                    velRes=0.0;
1 (ACLD)      2                                                    depthRes=0.0;
2 ()          8                                                wet = new Results();
2 (ALLO)     44                                                dry = new Results();
2 (OCO)      20
0 ()          0                                               //>  public Pipe(String fromId, String toId, String id ) {
0 ()          0                                                       super(fromId,toId,id);
0 ()          0                                                   }
0 ()          0                                                       public Link(String fromId, String toId, String id) {
0 ()          0                                                           super(fromId,toId,id);
0 ()          0                                                       }
0 ()          0                                                   //>  public Link(String fromId, String toId, String id) {
0 ()          0                                                           super(fromId,toId,id);
2 (ALLO)     44                                                           this.length = new Length(); // set defaults
2 (OCO)      20
0 ()          0                                                       //>  public Length() {
1 (ACLD)      2                                                           this.userLength = 0.0;
1 (ACLD)      2                                                           this.calcLength = 0.0;
1 (ACLD)      2                                                           this.length = 0.0;
0 ()          0                                               }       }
2 (ALLO)     44                                                       this.slope = new Slope();  // set defaults
2 (OCO)      20                                                   //>  public Slope() {
1 (ACLD)      2                                                       this.userSlope = 0.0;
1 (ACLD)      2                                                       this.calcSlope = 0.0;
1 (ACLD)      2                                                       this.slope = 0.0;
1 (ACLD)      2                                                       this.usInvert= 0.0;
1 (ACLD)      2                                                       this.dsInvert = 0.0;
0 ()          0                                                   }
2 (ALLO)     44                                                   inFlow = new Hydrograph(null);
2 (OCO)      20                                                   //>  public Hydrograph(String id) {
0 ()          0                                                       super(id);
1 (ACLD)      2                                                       minVal=0.0;
1 (ACLD)      2                                                       maxVal=0.0;
0 ()          0                                                   }
2 (ALLO)     44                                                   outFlow = new Hydrograph(null);
2 (OCO)      20                                                   //>      public Edge(String fromId, String toId, String id) {
3 (FNPR)      6                                                           super(id);
1 ()        237 [APPO]
3 (FNC)      21                                                           newFromVertex(fromId);
1 (ALLO)     22                                                           newToVertex(toId);
1 (OCO)      10                                                           surfaces = new HashSet();
1 (FNC)       7      305 [nFV]                                        //> public void newFromVertex(String vertexId){
1 (FNPR)      2                                                              if (addFromVertex(vertexId)) Util.vertex(fromId).addFromEdge(this.id);
1 (C2CI)      2                                                          }
1 ()        131 [UTILGET]
1 (FNC)       7
1 (FNPR)      2
1 (ORDL)      2                                                        //> public boolean addFromVertex(String vertexId){
0 ()          0                                                              boolean b = (fromId==null);
1 (C2CI)      2
1 (ALLI)      2                                                              if (b) fromId=vertexId;
1 (C2LI)      3                                                              return b;
1 (ALLS)     12                                                       }
1 (FNPF)      2                                                      //> public boolean addFromEdge(String edgeId){
0 ()          0                                                              boolean b= !fromEdges.contains(edgeId);  //redundant?
1 (OJH)      50
1 (ALLI)      2                                                              if (b) fromEdges.add(edgeId);
1 (C2CI)      2
1 ()         75 [MA]                                                        return b;
1 (FNPF)      2                                                       }
0 ()          0                                                  //> public void newToVertex(String vertexId){
1 ()        305 [nFV]                                                      if (addToVertex(vertexId)) Util.vertex(toId).addFromEdge(this.id);
0 ()          0                                                      }
0 ()          0
1 ()         75 [MA]          pipes.add(pipe.getId());
4 (FNC)      28               pipe.getSlope().setUsInvert(usInvert);
2 (FNPD)      6               pipe.getSlope().setDsInvert(dsInvert);
```

4 (ORDL)	8			//> public void setUsInvert(double usInvert){this.usInvert=usInvert;}
2 (ALLD)	8			//> public void setDsInvert(double dsInvert){this.dsInvert=dsInvert;}
3 (FNC)	21			pipe.setDiameter(diam / 1000.0); //m
3 (ALLD)	12			pipe.setLength(leng); //0-> from coord
3 (ORDL)	6			pipe.setSlope(slope/100.0); //o->from coords
3 (FNPD)	9			
2 (ODDC)	8			
1 (C2CD)	4			if (manning==0.0)
1 (ALLD)	4			pipe.setManning(Util._manConst);
1 (FNC)	7			else
1 (FNPD)	3			pipe.setManning(manning);
1 (ORDL)	2			
1 (C2CI)	2			if (newpipe==1) pipe.setExisting(false);
1 (FNC)	7			
1 (FNPI)	2			
1 (ALLI)	2			//A
1 (ALLO)	22	2191	[ERFCOI	erventmp=Erf.construct(pipe.getId(),"A",HydroClasses.Util._capitaConst,
1 (FNC)	7			ervenCountA);
2 (FNPR)	4			
1 (FNPI)	2			
1 (FNPD)	3			
4 (ORDL)	8			
1 (FNC)	7			
1 (ORDL)	2			
1 (FNPF)	2			
1 (ALLS)	12			
0 ()	0			//> **public static String[] construct(String edgeId, String productionId, double capita, int count){**
1 (FNLF)	2.5	2.5	Avg=2.5	for (int i=0;i<count;i++){
1 (C2LI)	7.5	2.5		
1 (OCO)	25	2.5		erf = new Erf(null,edgeId,productionId,capita); //id to be generated
3 (FNPR)	15	2.5		
1 (FNPD)	7.5	2.5		//> **public Erf(String id, String edgeId, String productionId, double capita) {**
12 (FNPR)	60	2.5		super(id,edgeId,productionId,capita);
4 (FNPD)	30	2.5		//> **public Erf(String id, String edgeId, String productionId, double capita) {**
4 (FNC)	70	2.5		super(id,edgeId,productionId,capita);
0 ()	0	2.5		//> **public Erf(String id, String edgeId, String productionId, double capita) {**
0 ()	0	2.5		super(id,edgeId,productionId,capita);
0 ()	0	2.5		//> **public Erf(String id, String edgeId, String productionId, double capita) {**
0 ()	0	2.5		super(id,edgeId,productionId,capita);
0 ()	0	2.5		}
6 (FNPR)	30	2.5		//> **public Erf(String id, String edgeId, String productionId, double capita) {**
3 (FNC)	52.5	2.5		super(id,edgeId);
1 (ALLS)	30	2.5		this.productionId = productionId;
1 (ALLD)	10	2.5		this.capita = capita;
0 ()	0	2.5		//> **public Area(String id, String edgeId) {**
0 ()	0	2.5		super(id,edgeId);
0 ()	0	2.5		//> **public Area(String id, String edgeId) {**
0 ()	0	2.5		super(id,edgeId);
1 (ALLO)	55	2.5		vertices = new double[3][];
1 (OCO)	25	2.5		//> **public Surface(String id, String edgeId) {**
1 ()	592.5	2.5	[APPO]	super(id);
1 (ALLO)	55	2.5		this.edgeId=edgeId;
1 ()	762.5	2.5	[nFV]	Util.edge(edgeId).addSurface(this.id);
1 (ALLS)	30	2.5		erven[i]=erf.getId();
1 (FNC)	17.5	2.5		
1 (ORDL)	5	2.5		
1 (ORDA)	2.5	2.5		
1 (ALLS)	30	2.5		
1 (FNPF)	5	2.5		
1 (FNLN)	5	2.5		}
1 (FNPF)	2	2.5		return erven;
0 ()	0	2.5		}
1 ()	187.5	2.5	[MA]	for (i=0;i<erventmp.length;i++) erven.add(erventmp[i]);
1 (ORDL)	5	2.5		
1 (ORDA)	2.5	2.5		
0 ()	0	End of erf add		//E Repeat for B to E
0 ()	0	0	[ERFCOI	erventmp=Erf.construct(pipe.getId(),"B",HydroClasses.Util._capitaConst,

```
TOTAL                61565 [SMAL]

                           public void drawModel() throws RemoteException{
                           Draws the hydraulic model in the CAD Environment

    1 (FNLF)         1                          while (pipeIds.hasNext()) {
    1 (ORDL)         2                            pipe = Util.pipe( (String) pipeIds.next());
    1 (ALLO)        22
    1 (FNC)          7
    1 (ORDL)         2
    1 ()           581 [UTILGET]
    1 (FNC)          7                          pipe.draw(modelViewer);
    1 (FNPR)         2                           //> public void draw(ICad modelViewer){
    1 (ORDL)         2
    1 (ALLI)         2                                int o = modelViewer.drawLine( ((Node) this.getUsNode()).getX(),
    5 (ORDL)        10                                                             ((Node) this.getUsNode()).getY(),
    4 (FFC)        848                                                             ((Node) this.getDsNode()).getX(),
    4 (ALLD)        16                                                             ((Node) this.getDsNode()).getY());
    4 (FFPF)        24
    4 (FFPD)        12
    0 ()             0                             //> public int drawLine(double x1, double y1, double x2, double y2) throws RemoteException;
    2 (C2CO)         4                                 if ((dxfViewer.inputFile!=null) & (dxfViewer.fileConverter!=null)) {
    2 (ORDL)         4
    1 (FNC)          7                                     o= dxfViewer.fileConverter.addLine(dxfViewer.inputFile, (float) x1, (float) y1, (float) x2, (float) y2, (short) 1, _modelLayer );
    3 (ORDL)         6                                      //> public Object addLine(DxfFile dxf, float x1, float y1, float x2, float y2, short col, String layer ) {
    4 (FNPD)        12
    1 (FNPI)         2
    2 (FNPR)         4
    1 (ALLO)        22                                                 DrawLines dline = new DrawLines(2);
    1 (OCO)         10
    1 (FNPI)         2
    0 ()             0                                                  //> public DrawLines(int nr) {
    1 (C2CI)         2                                                       if (nr > 0) {
    1 (ALLO)        22                                                         line = new Point3D[nr];
    1 ()            23 [P3D]
    1 (OCO)         10                                                      }
    1 (ORDL)         2
    1 (FNC)          7                                                 dline.addPoint(x1,y1,0.0f);
    3 (FNPD)         9
    1 (ORDL)         2
    0 ()             0        116 [AP]                                  //> public void addPoint(float x, float y, float z) {
    1 (C2CO)         2                                                       if (line == null) {
    1 (ALLO)        22                                                         line = new Point3D[2];
    1 ()            23 [P3D]
    1 (OCO)         10                                                      }
    1 (ORDL)         2                                                      else if (nrPoints == line.length) {        //NA
    1 (ALLO)        22                                                      line[nrPoints++] = new Point3D(x, y, z);
    1 (OAIC)         1
    1 (ORDA)         1
    1 (OCO)         10
    1 ()            23 [P3D]
    1 ()           116 [AP]                                          dline.addPoint(x2,y2,0.0f);
    1 (ALLI)         2                                             dline.setColor(col);
    1 (FNC)          7
    1 (ORDL)         2
    1 (FNPF)         2
    2 (FNC)         14                                             dline.setLayer(dxf.getLayer(layer));
    1 (FNPF)         2
    2 (ORDL)         4
    2 (ALLS)        24
    1 (FNPR)         2
    1 (FNC)          7        35 [AD]                               complete.addDrawable(dline);
    1 (ORDL)         2
    1 (FNPR)         2                                              //> public void addDrawable(DrawAble d) {
    0 ()             0                                                   drawables[nrDrawables++] = d;
    1 (ALLO)        22                                             return (dline);
    1 (OAIC)         1
```

```
2 (ALLS)      24
1 (FNPR)       2
1 (FNC)        7      35 [AD]                                    complete.addDrawable(dline);
1 (ORDL)       2
1 (FNPR)       2
0 ()           0                                                   //>  public void addDrawable(DrawAble d) {
1 (ALLO)      22                                                       drawables[nrDrawables++] = d;
1 (OAIC)       1                                                  return (dline);
1 (ORDA)       1                                              }
1 (FNPF)       2
1 (FFPF)       6                                          return o.hashCode();
1 (FNC)        7
1 (ORDL)       2
1 (FNPF)       2                                      }
1 (ALLI)       2
1 (ALLO)      22     2086 [REL]              Relation tmpRel = new Relation(o,id,Rel._linkRelSuffix); //construct new external binary relation
1 (OCO)       10                             //>      public Relation(int fromHashCode, String toString, String toClassName) {
3 (FNPR)       6
1 (ALLS)      12                                          tmpFromString=Integer.toString(fromHashCode);
1 (OCIS)     613
1 ()         102 [MH]                                    this.toObject=App.getObject(toString);          //store reference
2 (ALLS)      24                                         this.toId=toString;                             //store id
2 (OSA)     1048                                         id=tmpFromString+"-"+toClassName;               //create key string
1 (ALLS)      12                                         this.id = id;                                   //store id locally
1 ()         237 [APPO]                                  Rel.addRel(this);                               //file in hashtable
1 (FNC)        7                    pipe.annotate(modelViewer);}
1 (ORDL)       2                    //> public void annotate(ICad modelViewer){
1 (FNPR)       2
1 (FFC)      212                          modelViewer.drawText(id,
3 (FFPD)       9                                  (getUsNode().getX() + getDsNode().getX() ) / 2 + 3.0,
1 (FFPI)       2                                  (getUsNode().getY() + getDsNode().getY() ) / 2 + 3.0 ,3,2);
1 (FFPR)       2
4 (OADD)      16
4 (ORDL)       8
2 (ODDC)       8
8 (FNC)       56
8 (FNPF)      16
4 (ALLD)      16
4 (ALLO)      88
0 ()           0     773 [DT]          //->[Cross Boundary]
0 ()           0                              //>  public void drawText(String st, double x1, double y1,int pen, double size)  throws RemoteException{
2 (C2CO)       4                                  if ((dxfViewer.inputFile!=null) & (dxfViewer.fileConverter!=null)) {
2 (ORDL)       4
1 (FNC)        7                                     dxfViewer.fileConverter.addText(dxfViewer.inputFile, st, (float) x1, (float) y1, (short) pen, (float) size, _modelTextLayer );
3 (ORDL)       6                                      //>  public void addText(DxfFile dxf, String text, float posX, float posY, short col, float size, String layer  ) {
3 (FNPD)       9
1 (FNPI)       2
2 (FNPR)       4
1 (ALLO)      22                                        DxfTEXT dxfText = new DxfTEXT();
1 (OCO)       10
5 (FNC)       35                                        dxfText.setGroup((short)1,text);  //Text
5 (FNPI)      10                                        dxfText.setGroup((short) 10,posX); //X
4 (FNPD)      12                                        dxfText.setGroup((short) 20,posY); //Y
1 (FNPR)       2                                        dxfText.setGroup((short) 30,0.0f);  //Z
5 (ORDL)      10                                        dxfText.setGroup((short) 40,size);  //Size
1 (C2CI)       2                                        if (fontsAvailable()) {
1 (ALLO)      22                                          DrawText dtext = new DrawText(null, dxfText, dxf, this);
1 (OCO)       10
4 (FNPR)       8                                          //> Assumed
1 ()         500                                        dtext.setColor(col);
1 (ALLI)       2
1 (FNC)        7
1 (ORDL)       2
1 (FNPF)       2
2 (FNC)       14                                        dtext.setLayer(dxf.getLayer(layer));
1 (FNPF)       2
2 (ORDL)       4
```

```
2 (ALLS)    24
1 (FNPR)     2
1 ()        35 [AD]                                                    complete.addDrawable(dtext);
0 ()         0                                                      }
0 ()         0                                                  }
0 ()         0              }
1 (FNLF)     1          //NODES
1 (ORDL)     2          while (nodeIds.hasNext()) {
1 (ALLO)    22             manhole = Util.manhole((String) nodeIds.next());
1 (FNC)      7
1 (ORDL)     2
1 ()       131 [UTILGET]
1 (FNC)      7             manhole.draw(modelViewer);
1 (FNPR)     2              //> public void draw(ICad modelViewer ){
1 (ORDL)     2
1 (ALLI)     2                      int o = modelViewer.drawCircle(x,y, 5.0);
1 (FFC)    212                      //> public int drawCircle(double x1, double y1, double rad) throws RemoteException;
3 (FFPD)     9
0 ()         0                    //->[Cross Boundary]
0 ()         0                          public int drawCircle(double x1, double y1, double rad) throws RemoteException{
2 (C2CO)     4                          if ((dxfViewer.inputFile!=null) & (dxfViewer.fileConverter!=null)) {
2 (ORDL)     4
1 (ALLI)     2                            o= dxfViewer.fileConverter.addCircle(dxfViewer.inputFile, (float) x1, (float) y1, (float) rad, (short) 1, _modelLayer );
1 (FNC)      7
4 (ORDL)     8
3 (FNPD)     9
1 (FNPI)     2                            //->
2 (FNPR)     4
0 ()         0                            //> public Object addCircle(DxfFile dxf, float centerX, float centerY, float radius,  short col, String layer  ) {
1 (C2CI)     2                                  if (radius > 0) {
1 (ALLO)    22                                   line = new DrawLines(CIRCLE_SEGMENTS);
1 (OCO)     10
1 (FNPI)     2
0 ()         0                                     //> public DrawLines(int nr) {
1 (C2CI)     2                                           if (nr > 0) {
1 (ALLO)    22                                            line = new Point3D[nr];}
1 (OCO)     10
1 (ORDL)     2
1 ()        23 [P3D]                                  }
1 (ALLD)     4                                   float delta = (float)(2*Math.PI/CIRCLE_SEGMENTS);
1 (OMDC)     4
1 (ODDC)     4
1 (ACLD)     2                                   float angle = 0;
1 (FNLF)     1                                   for (int i = 0;  i < CIRCLE_SEGMENTS;  i++, angle+=delta) {
1 (C2CI)     2
1 (OAIC)     1
1 (OADD)     4
1 (FNC)      7                                    line.addPoint(centerX + (float)(radius*Math.cos(angle)),
1 (ORDL)     2                                                  centerY + (float)(radius*Math.sin(angle)),centerZ);
2 (OADD)     8
2 (OMDD)     8
2 ()        20 Cos,Cos
3 (FNPD)     9                                              //> public void addPoint(float x, float y, float z) {
0 ()         0                                                   if (line == null) {
1 (C2CO)     2                                                       line = new Point3D[2];
1 (ALLO)    22
1 ()        23 [P3D]
1 (OCO)     10                                                  else if (nrPoints == line.length) {
1 (C2CI)     2                                                  }
1 (ORDL)     2                                                  line[nrPoints++] = new Point3D(x, y, z);
1 (ALLO)    22                                                 }
1 (OAIC)     1
1 (ORDA)     1                                   }
1 (OCO)     10
1 ()        23 [P3D]
1 (FNC)      7                                   line.close();
1 (ORDL)     2
```

```
0 ()                0                                    //> public void close() {
1 (C2CI)            2                                        if (nrPoints > 0) {
1 (ACLI)            1                                          isClosed = true;
1 (ALLI)            2                                    line.setColor(col);
1 (FNC)             7                                    line.setLayer(dxf.getLayer(layer));
1 (ORDL)            2                                    complete.addDrawable(line);
1 (FNPF)            2
2 (FNC)            14
1 (FNPF)            2
2 (ORDL)            4
2 (ALLS)           24
1 (FNPR)            2
1 ()               35 [AD]
1 (FNPF)            2                                    return (line);
0 ()                0                                }
1 (FFPF)            6                            return o.hashCode();
1 ()             2086 [REL]                      Relation tmpRel = new Relation(o,id,Rel,_nodeRelSuffix); //construct new external binary relation
1 (FNC)             7                manhole.annotate(modelViewer);}
1 (ORDL)            2                    //> public void annotate(ICad modelViewer){
1 (FNPR)            2                        modelViewer.drawText(id,x,y+5.0,3,2);
1 (FFC)           212                    }
3 (FFPD)            9
1 (FFPI)            2
1 (FFPR)            2
1 (OADD)            4
1 (ORDL)            2
1 ()              773 [DT]
0 ()                0            } //END
0 ()                0
```

Appendix H

AC Analysis Results

Table H.1: Spreadsheet of AC performance evaluation

APPLICATION-CENTRED DESIGN

			Standalone				Client / Server (One PC)				Client / Server (2 PCs)			
			PR1	PR2	PR3	PR4	PR1	PR2	PR3	PR4	PR1	PR2	PR3	PR4
Type	*Code*	*Description*	104	682	2934	6721	104	682	2934	6721	104	682	2934	6721
DoE [s]	TFD1	Show Direction Definition	0.3	0.7	0.9	2.3	0.9	3.3	10.8	23.8	1.0	3.5	11.5	27.0
UIC	*TFD2*	*Swap Direction Definition*	*104*	*682*	*2934*	*6721*								
UIC	*TMC*	*Topolology MH Correction*	*94*	*614*	*2641*	*6049*								
			104	682	2934	6721	104	682	2934	6721	104	682	2934	6721
DoE [s]	UE	**TOTAL: Update Elevations**	**0.1**	**0.2**	**0.4**	**1.0**	**0.2**	**0.3**	**0.9**	**1.3**	**0.2**	**0.3**	**0.4**	**0.5**
Eq. BIC[s]	*UE*	***TOTAL: Update Elevations (scaled)***	***0.01***	***0.05***	***0.22***	***0.50***								
			104	682	2934	6721	104	682	2934	6721	104	682	2934	6721
DoE [s]	TSU	Topology Slope Update	0.1	0.9	2.7	5.7	0.2	1.7	3.5	7.2	0.3	0.3	2.0	5.0
UIC	*TPU*	*Topology Parcel Update*	*83*	*546*	*2347*	*5376*								
			104	682	2934	6721	104	682	2934	6721	104	682	2934	6721
DoE [s]	BM	**TOTAL: Build Model from CAD (All)**	**0.8**	**18.7**	**156.3**	**1267.1**	**1.2**	**22.2**	**175.8**	**1250.0**	**1.2**	**34.5**	**179.0**	**1300.0**
Eq. BIC[s]	*BM*	***TOTAL: Build Model from CAD (scaled)***	***0.62***	***15.15***	***250.60***	***1288.40***								
			104	682	2934	6721	104	682	2934	6721	104	682	2934	6721
DoE [s]	DM	**TOTAL: Draw Hydraulic Model**	**0.8**	**2.5**	**5.2**	**10.3**	**2.3**	**8.4**	**30.0**	**70.0**	**2.3**	**9.0**	**36.0**	**80.0**
Eq. BIC[s]	*DM*	***TOTAL: Draw Hydraulic Model (scaled)***	***0.19***	***1.28***	***5.50***	***12.60***								
			104	682	2934	6721	104	682	2934	6721	104	682	2934	6721
DoE [s]	ACCU	Accumulate all flows	0.3	1.4	4.9	9.5	0.5	1.8	9.0	12.4	0.3	1.0	3.1	6.0
DoE [s]	ANAL	*Do analysis*	0.4	2.0	8.5	15.5	0.6	2.5	8.1	17.0	0.3	1.3	4.3	7.8
DoE [s]	DESGN	Do design	0.6	2.7	9.5	22.5	0.7	3.3	10.0	19.3	0.4	1.5	5.4	12.8
DoE [s]	HAD	**TOTAL: Hydraulic Analysis & Design**	**1.3**	**6.1**	**22.9**	**47.5**	**1.8**	**7.6**	**27.1**	**48.7**	**1.0**	**3.8**	**12.8**	**26.6**
			104	682	2934	6721	104	682	2934	6721	104	682	2934	6721
DoE [s]	ALL	**TOTAL: Build Topo+Model+Draw**	**1.7**	**21.4**	**161.9**	**1278.4**	**3.7**	**30.9**	**206.7**	**1321.3**	**3.7**	**43.8**	**215.4**	**1380.5**
Eq. BIC[s]	*ALL*	***TOTAL: Build Topo+Model+Draw***	**0.8**	**16.5**	**256.3**	**1301.5**								
UIC	*ALL*	***TOTAL: Mix***	***281***	***1842***	***7922***	***18146***								

Notes

1 All BICs (where indicated) are scaled by **0.2**
to accommodate the difference in execution speed of
environments and provide the equivalent Basic Instruction Count

Appendix I

Tutorial for Sewsan AC

I.1 Introduction

The purpose of this appendix is to provide a guide to demonstrate or evaluate the functionality of the *SEWSAN AC* application. Contact alex@sinske.com to obtain the contents of the distribution CD.

I.2 Installing the program

I.2.1 Stand-alone scenario

For the stand-alone (1 PC) scenario, copy all files from the `Classes` directory on the distribution CD to a directory with the same name on the hard drive. Ensure that Java 1.3.0 or later is installed and that the Java program can be found in the search path. The installation has only been tested under Microsoft Windows 98 and Microsoft Windows 2000, but should also work under Linux, provided an equivalent batch file is used. The program is executed by typing `startStandalone` at the command prompt. The CAD files are located in the subdirectory `Drawings` and can be loaded from there. During execution the console window shows status messages for both the client and server components.

I.2.2 Single PC client-server scenario - Java client

For the 1 PC client-server scenario, both the client and server components are executed on the same PC. This setup requires a web server (http server)

running on the computer. The web server is used as codebase for the shared interface definition files. The webserver must accept requests on localhost or 127.0.0.1

Copy all files from the `Classes` directory to a directory on the hard drive. Ensure that Java 1.3.0 or later is installed and that the Java program can be found in the search path. Ensure the webserver is running and execute `CopyForCS.bat` to copy interface definition files to the webserver. Edit the batch file if required. It assumes a default Microsoft IIS Webserver setup and requires `javaclasses` as virtual directory.

Copy the `ClientClasses` directory to a directory with the same name on the hard drive. Now, firstly start the RMI server. This is done by executing `startRMI.bat` from the `Classes` directory. Then start the server by executing `startServer.bat` from the `Classes` directory. This console window will show all server side messages.

Finally execute `startClient.bat` from the `ClientClasses` to start the Java client. This console window will show any client side messages.

The installation has only been tested under Microsoft Windows 98 and Microsoft Windows 2000, but should work also under Linux, provided equivalent batch files are used.

The server can also be located on a different PC. All the above-mentioned batch files must then be edited to reference the IP address of the server PC instead of 127.0.0.1

I.2.3 Single PC client-server scenario - Applet client

For the 1 PC client-server scenario, both the client and server components are executed on the same PC. This setup requires a web server (http server) running on the computer. The web server is used as codebase for the shared interface definition files as well as for hosting the client side classes in compact JAR format. The web server must accept requests on localhost or 127.0.0.1

Ensure the web server is running and execute `CopyForWWW.bat` from the `Classes` directory to copy interface class files to the web server. Edit the batch file if required. It assumes a default Microsoft IIS web server setup and requires `javaclassesi` as virtual directory. Ensure that you have updated your browser with the Sun Plug-in for Java 1.3.0 or later. This ensures that the correct virtual machine is available for executing Java code in the

browser. Also ensure that the security settings of the browser allow the execution of Java code.

Firstly the RMI server must be started. This is done by executing `startRMI.bat` from the `Classes` directory. Then start the server part by executing `startServer.bat` from the `Classes` directory. This console window will show all server messages.

Finally point the internet browser to, `http://127.0.0.1/ javaclass-esi/ ExampleVerySmall.htm` to start the Java client. This optional Sun Plug-in console window will show any client side messages. The installation has only been tested under Microsoft Windows 98 and Microsoft Windows 2000 with Microsoft Internet Explorer 5.5 and 6.0, but should also work under Linux and Netscape, provided equivalent batch files are used and minor changes are made.

Alternatively, the server can be located on a different PC. All the above-mentioned batch files must then be edited to reference the IP address of the server PC instead of 127.0.0.1

Please note that in the client-server scenario, the CAD (DXF) files are located on the client machine. The data model as well as DTM are stored on the server. Within the browser the top menu is not visible, as well as the File Open Dialog Box for DXF and SDF cannot be shown due to security constraints. Only the DXF drawing located on the web server and specified in the web page file can be loaded.

I.3 Description of user interface

The user interface consists of two parts: The DXF Viewer and the SEWSAN AC Menu. The DXF Viewer shows the loaded DXF drawing, the list of layers and provides command buttons for interactive model editing. The SEWSAN AC Menu provides access to commands to load a drawing, build a data model, perform an analysis and design on the model, as well as present results. The persistent storage of the data model is also supported. The following steps are performed in general from top to bottom in the SEWSAN AC Menu.

I.4 Loading the CAD Drawing

The first step involves loading an existing CAD drawing representing the sewer layout as well as background street and plot layout. As an alternative an existing model stored as an SDF data file (compatible with SEWSAN MC) can be loaded using `Import Model`.

- **Select Erase CAD from the SEWSAN AC Menu.** The existing default drawing is erased. However, the model data (if present) is not erased. Certain default layers are created automatically.
- **Select Import CAD** and select `ExampleVerySmall/ExampleVerySmall.dxf` This will draw lines and text representing the plan view of a sewer network, as well as associated erven (plots).
- **Zoom, Pan and Layer functions:** The following functions are available to zoom and pan in the DXF Viewer and change visible layers. They are activated by keeping the LEFT mouse button depressed and moving the mouse in combination with the follow key on the keyboard:

  (1) Dynamic zoom: Keep the CTRL button on the keyboard depressed; move to the RIGHT with the mouse to zoom in; to the LEFT zoom out

  (2) Dynamic pan: Keep the SHIFT button depressed; move to the RIGHT to move drawing to the RIGHT; move to the LEFT to move drawing to the LEFT.

  (3) Dynamic viewpoint: Keep the ALT button depressed and move the mouse around the outside of the circle to rotate drawing around Z-axis; move the mouse inside of the circle in line with centre of circle to rotate drawing around an axis perpendicular to this line and the centre of the circle.

  (4) Zoom All: Select the `X-Y` button. Also useful are the `Previous` and `Next` buttons to navigate between saved zoom states. The `Y-Z` and `X-Z` are used to change views in 3D drawings.

  (5) Layer management: Layers can be switched on and off by clicking on the layer name in the `Layers` box.

I.5 Building the Data Model

- **CAD to Model conversion:** The next step is to convert the lines, text and one circle (representing the outfall manhole) to a hydraulic model. Deselect the `CADASTRAL` layer to see all the layers that are used in the conversion. Now select `CAD->Model` to start the process. The `MODEL` and `MODEL-TEXT` layers are created, representing the visualization of the hydraulic model. Deselect the `TOPO`, `DIAM_TXT`, `ELEV_TXT`, `ERVEN`, `INVERT_TXT` and `MH_TXT` layers, as they have all been processed, and zoom in to see the pipes, manholes and their identifiers.

- **Querying a hydraulic element:** The basic engineering objects, i.e. the manholes (represented by circles) and the pipes (represented by lines) can be queried by clicking with the RIGHT mouse button on the element. A pop-up dialog appears showing the information as contained in the data model for the entity. It can be seen that automatically generated string identifiers are assigned to the pipe entities, prefixed with _ and the manholes are assigned identifiers with prefix `US` and `DS`. Dismiss the dialog by selecting CANCEL, unless changes are made.

- **Verifying and correcting the topology :** The next step is to verify the topology. Zoom to all (`X-Y button`) and select `Show Topo Errors`. Two asterisks on the `SCRATCH2` layer show where pipes enter at a node with conflicting direction. By selecting `Show Link Direction` and ensuring that `SCRATCH` layer is visible, the defined direction for each link is shown with an arrow. The link with the arrow pointing in the wrong direction at an asterisk (top-right pipes) can now be corrected: Zoom in, RIGHT-click on the link and change the "false" to anything else, e.g. "t" for "true" and press `Ok`. Verify the change by selecting `Show Link Direction` again or the `Show Topo Errors` which should not show asterisks any more on the `SCRATCH2` layer.

- **Adding elements to the topology :** This step can also be skipped altogether for this tutorial. It illustrates how pipes and manholes can be deleted and how new pipes can be added between new or existing manholes. It ensures that a consistent state is maintained after each operation.

  Zoom all, and then ensure the `CADASTRAL` layer is switched on. It can be seen that plots 16414, 16415 and 16416 are not being served by a sewer. This will now be corrected by adding a line which terminates as manhole

D1070: Zoom/Pan in to see the three plots as well as the terminating manhole. The buttons in DXF Viewer, viz. `AddPipe`, `(Existing)/(New)` will now be used. Select `AddPipe`. The button changes to `Cancel` which will later be used to stop the definition process. Your are now prompted to select the upstream manhole by LEFT-clicking once at the correct location, for example under the text `16414`. A circle representing the new manhole is drawn with its identifier. The mode has changed to select an existing manhole. However, a new manhole is again to be defined, therefore firstly click on `(New)` to change the definition mode, then click under the text `16415` to define the downstream manhole of the pipe. The pipe as well as the downstream manhole is drawn, as well as the identifiers. The mode has change to existing, which is correct as the lastly defined manhole must be selected as start manhole of the next pipe. Therefore click near the newly added circle. It will flash briefly in blue. The mode is set to define a new node, which is correct. LEFT-click under the text `16416` to define the next downstream manhole. To draw the last pipe, click on the lastly added circle, change the mode to existing by clicking on the `(Existing)` button, and click on the circle representing manhole `D1070`, the terminating manhole. This process is now done, so click the `Cancel` button.

If we assume that the second manhole added is located incorrectly, the `Delete` button can be used to remove a manhole as well as dependent topology elements: Select `Delete`, and click on the second manhole added. Now redefine a new pipe from the start manhole to a new manhole located under the text `16415`, and a second one from the last manhole to the manhole under the text `16416`. In a similar way the first pipe can be deleted, and redefined. When a pipe is deleted, the associated manholes are not automatically deleted.

The number of erven associated with each pipe may be edited. The default of 1 class A erf can be edited by RIGHT-clicking on a pipe and editing the field `Number_of_erven`.

- **Updating elevations :** Now load the DTM model by selecting `Load DTM` in the SEWSAN AC Menu. The model will be loaded from the source (server) directory, and from a file with the same name as the DXF file, but the with the extension DTM. Two new layers are now added, `DTM_GRID` and `DTM_POINTS`. The `DTM_GRID` layer contains a 5x5 3D-mesh representing the surface model. The `DTM_POINTS` layer contains labels (plotted at zero elevation) with the known elevation heights. The dynamic

viewpoint can be changed to visualize the DTM mesh and the model as described under Zoom, Pan and Layer functions.

The elevation at the manholes can now be interpolated from the known elevations by selecting the `Update Elevations` option from the SEWSAN AC Menu. This can be verified by clicking on a node, and observing the new `Elevation` value. Also the `Interpolation_state` and `Number_block_points` variables provide information on the quality of the interpolation.

- **Updating slopes and lengths :** Now an initial slope for the pipes in the model can be calculated, based on the ground elevation, minimum slope along pipes, as well as minimum drop value at manholes. But first any existing slopes and lengths must be zeroed, by selecting `Zero slopes & lengths`, then `Update Lengths/Slopes` will calculate lengths and slopes based only on the ground elevations and constant manhole drops. The function `Show Invert Level Errors` can then be used. It draws asterisks on the `SCRATCH2` layer to indicate where pipes with zero or negative slopes exists. This can be followed by `Update Inverts/Slopes` which also considers minimum slopes and the transfer of levels in the direction of the sewer network. The `Show Invert Level Errors` function now displays no errors. The data model is now ready to be analysed. Should you want to take a break at this stage, the system can be stored persistently using the `Save Model` option, and later loaded again using the `Load Model` option, after loading the CAD background drawing.

I.6 Using the Data Model

- **Analysing and designing the model :** Now the flows in the sewer system can be accumulated for two scenarios, a dry weather condition without the ingress of rain water, and a wet weather condition with the ingress of rain water. This is performed by selecting `Accumulate Flow` from the menu. A results file is also written, namely `CALC.OUT` in the program directory. This file contains a post-order tree of the network, i.e. the last row contains the outfall manhole, and the longest water course can be followed from bottom to top of the file. The file also contains data on the age of each pipe (either length or index based) as well as input parameters.

Next the velocity and flow depth is to be calculated for the two scenarios for all pipes in the sewer network. This is performed by selecting `Hydraulic Analysis` from the menu. Again a file is written to the program file directory with the name `ANAL.OUT`. This file contains a list of all the pipes in no particular order, with Flow Capacity (%), Flow (l/s), Velocity (m/s) and Depth of flow (mm) for both the Wet and Dry scenarios as well as the Full Flow condition. A status field comments on possible problems with the pipe.

Finally the hydraulic design can be selected by choosing `Hydraulic Design` from the menu. This starts a calculation of the minimum diameter required for a pipe in order to accommodate the wet flow scenario. The result is written to a file, `OPTIM.OUT` which states what size of pipe has been assign to each pipe. This is automatically followed by another analysis with the new pipe sizes.

All the analysis and design results for a pipe can be seen by RIGHT-clicking on the pipe.

- **Show queries :** Finally the results can be queried graphically, similar to a GIS system. Predefine queries can be selected in the menu for input data, such as `Diameter`, `Slope`, and results such as `Dry weather velocity` and `Wet weather velocity`. The `Wet weather flow capacity` in l/s or % of full-flow capacity, can also be shown.

  The queries are generate on a layer called `RESULTS`.

- **Other menu options :** A few other menu options remain in the DXF Viewer: `File / Exit` terminates the program; `Options / Language` allows selecting the language for the program (at this stage only the DXF Viewer); `Options / Show CS` toggles the display of the coordinates axis on the screen; `Options / CS Position` defines where the coordinate axis should appear; `Swap Black & White` toggles the background colour of the drawing. `Info / Model` provides information on the CAD model, such as the number of lines created. `Info / Java` presents information on the Java Virtual Machine state.

  This concludes the tutorial. A similar procedure can be followed on the larger text data sets, which are stored in subdirectories `ExampleSmall`, `ExampleMedium`, `ExampleLarge` and `ExampleHuge`, and have been used for the comparison with the MC design.

Index

A priori implementation, 60, 136, 152
Abstract data type, 3
AC design approach, 2, 73
AC quantitative analysis, 139
ActiveX, 15
Algorithmic background, 115
App, 95
AppObject, 95
AppSet, 99
AppSetObject, 98

Basic engineering objects, 90, 133
Basic Instruction Count, BIC, 16, 139
Basic Operations, 17
BIC, 63, 64, 67, 139–141
BIC (modified), 156
Bottom manhole, 92
Bridges, 4, 29, 48
Bridges, datafile, 29
Bridges, memory, 30, 53

CAD model, 127, 135
Class diagram, 11
Collaboration diagram, 11
Comparison of complexity, 159
Comparison of design, 151
Complexity, 4
Component diagram, 11
Construction procedure, 45
Contributor hydrograph method, 38
CORBA, 14

Data model, 86
DataModel object, 120
DFG, 8
Distributed scenario, 14, 62, 87, 129, 138, 147, 152
DoE, 16, 63, 67, 139, 141, 153
Duration of Execution, 16

Elevation model, 46, 126
Engineering models, 1, 122
Engineering process, 111
Erf, 92

Geographical model, 47
Graphs, trees, 93

HashCode, 94
HashMap, 94
HashTable, 94
Hydraulic model, 40

Identifiers, 74
Interpolation algorithm, 39

Listener, 12

Manhole, 90
Mapping, 84
MC design approach, 1
MC evaluation, 32
MC quantitative analysis, 63
Model objects, 135
Modelling methods, 11

Modules, 3, 27

Object duplication, 60
Object identifier management, 74, 93
Object interaction, 78
Object name scope, 135, 151
Object relation management, 83, 99
Object set management, 80, 97
OO analysis, 10
OO design, 10
OO paradigm, 8
OO programming, 10

Partial model, 27
Partial models, 4
PDS, 18, 64, 70, 147, 156
Persistent Data size, 18
Pipe, 91
Point-source, 92
Ports, 29
Presentation model, 126
Principles of OO, 9
Procedural approach, 3
Process, engineering, 27
Product-data model, 117
Program extensibility, 61
Program maintenance, 61

Quantitative comparison, 152
Quantitative criteria, 16

Rel class, 111
Relation, 12
Relation class, 109
Relation complexity, 102
Relation-set, 109
Relations, 83
RelObject, 108

Sequence diagram, 11
Sets, 12, 81
SEWSAN AC, 133
SEWSAN MC, 36
Software model, 27
State diagram, 11
Structured binary files, 49
Structured database files, 50
Structured text files, 49
Sub-model, 28

Test projects, 18
Topography model, 47
Topology model, 42
Transformer, 13, 31, 54
Tree traversal, 58, 135

UIC, 17, 64, 70, 139, 140, 144, 155
UML, 11
Unstructured binary files, 53
Unstructured text files, 52
User Interaction Count, 17

ViewerExtension class, 128
Views, 11, 15
Visualization model, 42, 87, 124