Ørjan Hatland

Sketcher .NET –
A Drawing Tool for Generalized Sketches

Department of Informatics
University of Bergen
June 2006

PREFACE

So, finally reached the destination! It (almost) seems like yesterday I started my education at Bergen University College, not quite knowing what my path would be. The three years spent there earning my Bachelor degree, extended my knowledge extensively within a very exciting area of computer applications, gave me many new friends and good friends, and also a great deal of challenges, adventures and experiences.
Coming from a more practical application area of computers at Bergen University College, the last two years have required lots of blood, sweat and tears moving towards a much more abstract area of computer science at the University of Bergen. However, it has been worth every cry!
Through the work with the present thesis, not only have I been acquainted to this highly interesting subject, I have also learned new aspects of my own abilities. For this I want to express my gratitude to my supervisors, Yngve Lamo at Bergen University College and Uwe Wolter at University of Bergen. I owe great thanks to both for skilled guidance and for being there for me through these years.
A special memory is the trip to Germany in the fall of 2005, where I was allowed to present my studies at Mittweida University of Applied Science. I would like to thank Bergen University College for providing the necessary funds.
I would also like to use this opportunity to thank my fellow students during the last five years, contributing to a nice and social community. Especially, I would like to thank Joakim, Roy, Håvard, Tor, Ole, Jørgen and Trond for a fertile collaboration throughout these years and lots of fun that I could not have been without.
Special thanks go to my family; my mother and father for providing a safe home and always being there for me, my kind-hearted older brothers Kim and Cato for constantly looking out for me and being the best brothers I could ever dream of. Also, my thanks go to Evangeline and Francis for encouraging my work, and to my seven month old nephew Daniel who brings so much joy to my family.
At last – but not at least – I would like to thank my dear girlfriend Kathrine for your love and your patience with me these last months and for your special gift at making me forget all my troubles. You are the best!

Finishing my thesis means finishing my days as a student, from which I bring along good memories. The knowledge I have earned will hopefully make me able to meet new challenges. I feel privileged!

CONTENTS

Preface ... **iii**
Contents ... **v**
List of Figures ... **viii**
List of Tables ... **ix**
List of Definitions ... **x**
1 Introduction ... **1**
1.1 Topic ... 1
1.2 Problem description ... 1
1.3 Justification, motivations and benefits ... 1
1.4 Construction of thesis ... 2
2 Historical Background ... **3**
2.1 Software engineering and the rise of object-oriented methodologies ... 3
2.2 Graphical specification languages ... 6
2.2.1 Entity-Relationship Diagram (ERD) ... 7
2.2.2 Unified Modeling Language (UML) ... 8
2.2.3 Problems with modern modeling languages ... 10
2.3 Category theory ... 11
2.4 Generalized sketches ... 18
3 Problem Analysis ... **23**
3.1 Task ... 23
3.1.1 Functional requirements specification ... 24
3.1.2 Non-functional requirements specification ... 25
3.2 Existing solution ... 25
3.2.1 History ... 26
3.2.2 Description of solution ... 26
3.2.2.1 The signature domain ... 28
3.2.2.2 The sketch domain ... 30
3.2.3 Drawbacks in the solution ... 31
3.3 Challenges ... 32

4 Available Technology 33
4.1 Developing platforms and tools 33
4.1.1 Microsoft Windows 33
4.1.2 Linux 35
4.1.3 Qt 35
4.1.4 The Java 2 platform 37
4.1.5 The Microsoft .NET platform 39
4.2 Programming languages 41
4.2.1 C++ 42
4.2.2 The Java language 43
4.2.3 C# 43
4.3 Data persistence method 44
4.3.1 Database 44
4.3.2 File system 46
4.3.3 XML 46
4.4 Technologies of choice 48
4.5 A closer look at the .NET Framework 50
4.5.1 Common Language Runtime (CLR) 50
4.5.2 Windows Forms 53
4.5.3 GDI+ support 56
4.5.4 XML support 56
4.5.5 Assemblies 57
4.5.6 .NET Framework 1.1 vs. .NET Framework 2.0 57
5 Solution 59
5.1 Implementing the core of generalized sketches 60
5.2 Implementing drawing tool related functionality 61
5.2.1 Draw objects 62
5.2.2 The tool case 63
5.2.3 The drawing surface 65
5.2.4 Drawing issues 67
5.2.4.1 Drawing nodes 68

5.2.4.2 Drawing arrows 69
5.2.4.3 Drawing diagrams 71
5.2.4.4 Drawing sketches 72
5.3 Implementation of generalized sketches specific functionality 73
5.3.1 Visual representation of signatures 73
5.3.2 Visual representation of sketches 75
5.3.3 Signature operations 77
5.3.4 Sketch operations 78
5.4 Implementation of data persistence methods 79
5.5 Other issues 81
5.5.1 Resources 81
5.5.2 Use of external libraries 81
6 Conclusion 83
6.1 Prospective features 83
6.2 Status of the program 83
6.3 Further work 83
References 85

LIST OF FIGURES

Figure 2-1: Moore's Law and Intel's processors. 4
Figure 2-2: A simple ER Diagram. 8
Figure 2-3: Formal logic for a modeling language: ML – a language, L – a logic..... 11
Figure 2-4: A diagram with its corresponding shape graph. 14
Figure 2-5: A simple generalized sketch. 19
Figure 3-1: Sketcher 95 – A Multiple Document Interface (MDI) application. 27
Figure 3-2: Separating source as diagram constraint. 29
Figure 3-3: Dialog box for node constraints. 30
Figure 3-4: Dialog box for a node's properties. 31
Figure 4-1: Microsoft Windows versions timeline. 33
Figure 4-2: Qt supports cross-platform development. 36
Figure 4-3: Java as a platform independent technology. 38
Figure 4-4: An overview of the .NET Framework. 40
Figure 4-5: Simplified C# family tree. 44
Figure 4-6: The DBMS approach. 45
Figure 4-7: Common Language Infrastructure. 52
Figure 4-8: Windows Forms hierarchy. 54
Figure 5-1: Sketcher .NET – A Tabbed Document Interface (TDI) application. 59
Figure 5-2: Structure of the generalized sketches formalism. 61
Figure 5-3: Class diagram over draw objects. 63
Figure 5-4: Communication model of the drawing tool. 66
Figure 5-5: Decomposition of lines. 70
Figure 5-6: Communication model for drawing sketches. 73
Figure 5-7: Node constraint dialog. 78

LIST OF TABLES

Table 4-1: How to write a Windows application using a C-based language (Microsoft-centric view)..... 34
Table 4-2: An example of an XML structure..... 47
Table 4-3: Supported programming languages in the .NET Framework..... 50
Table 5-1: Interface of the abstract DrawObject class..... 63
Table 5-2: Common interface of tools..... 64
Table 5-3: Responding to the event raised by the mouse movement over the drawing surface..... 65
Table 5-4: Handling key events..... 67
Table 5-5: Drawing nodes..... 69
Table 5-6: Drawing node markers..... 69
Table 5-7: Drawing arrows markers..... 71
Table 5-8: Adding the missing OnPaint..... 75
Table 5-9: Extending the System.Windows.Forms.UserControl class..... 76
Table 5-10: Drawing a sketch..... 77
Table 5-11: Defining complex types in XML Schema..... 80
Table 5-12: XML attributes..... 80
Table 5-13: Serializing and de-serializing..... 81

LIST OF DEFINITIONS

Definition 2-1: Graphs 12
Definition 2-2: Graph Homomorphisms 13
Definition 2-3: Categories 13
Definition 2-4: Functors 14
Definition 2-5: Diagrams 14
Definition 2-6: Mono and jointly mono 15
Definition 2-7: Products 16
Definition 2-8: Sums (coproducts) 16

1 INTRODUCTION

1.1 Topic

The present thesis is based on a graph-based specification language, called generalized sketches. The primary objective is the development of a drawing tool software application adapted for this language. Key words are software engineering, graphical specification languages, category theory, generalized sketches, .NET Framework v1.1, Windows Forms, graphical 2-D programming, GDI+, C#, XML, XML Schema, and XML serialization.

1.2 Problem description

Most graphical notations used in software engineering are based on improper logics, leading to *ambiguous constructions* and *semantic relativism*. For this reason, one has seen on the opportunity of using constructions already developed in *category theory* to straighten the flaws of these languages. A proposal was made by M. Makkai, and independently by Z. Diskin of a universal graphical specification-formalism with profound roots in category theory, but more suitable for practical software engineering. Makkai called them *generalized sketches*.
To the current state of the art, there exist no drawing tool software applications for generalized sketches suitable for practical work. A program has been developed for this purpose, but it is rather old and has unfortunately not been finalized, a problem that is reflected through the use of this application as it suffers from some serious bugs. The opportunity for straightening and further development of the program has been examined. Unfortunately, neither source code nor documentation of the program is available due to the drawing tool's age and licensing issues. Thus, the emphasis of this thesis is the reimplementation of such a drawing tool program adapted for generalized sketches, and the problem that is treated may be stated as follows:

Develop an extensible framework of a drawing tool software application adapted for generalized sketches. Implement the core functionality so that it is possible to define signatures and draw sketches based on given signatures.

1.3 Justification, motivations and benefits

A novel formalized specification paradigm with profound roots in category theory has been developed that makes it possible to solve a lot of specification problems occurring in software engineering. It is a universal formalism with a nice amalgamation of logical rigor and graphical evidence, and can be used as a unifying framework of the great many graphical notation systems developed for specification purposes within software engineering.

To further explore the practical values of the formalism in question however, some program must be developed to provide the necessary framework for building such specifications and which supports the diversity of potential operations that can be applied in coordination with this universal formalism. This reflects the main focus of the present thesis.
The work of developing such a program is both extensive and challenging, and thus this thesis is only meant to be the initial effort of a bigger project with the goals of achieving a software tool capable of putting to use all advantages offered by this novel formalism, to find out if its claims hold and indeed is of practical interest for software engineering.

1.4 Construction of thesis

Chapter 2 outlines the historical background of the current subject to put things into perspective. It introduces software engineering as a very young and immature engineering discipline together with modern software development techniques used to deal with the seemingly inevitably complexity of software. Especially concerns are directed towards graphical specifications used in software engineering and their role as such, trying to illuminate some well known problems and their entailing consequences, and giving the reason why this is so. The chapter finishes off with an introduction to the mathematical background material of the thesis, that of category theory as a mathematical framework supporting formal specifications and models, and finally, generalized sketches emerging from category theory as a unified specification framework for the entire field of software engineering.
Chapter 3 starts with analyzing the actual task of the thesis and giving a rough requirement specification. Subsequently, it discusses the status of an existing program developed in 1995-1996 with the same purposes in mind as the one of the present thesis, and (more or less) describes the solution this program offers to the problems involved. The chapter finishes off with a summary of the major challenges of the thesis.
Chapter 4 for the most part discusses different available technologies that could be used to solve the task, and further on explains the choice of technologies. It also gives a more thoroughly introduction to essential parts of the chosen technologies that are heavily used in the solution.
Chapter 5 comprises the solution, in a systematic matter trying to explain the way of thinking when developing a drawing tool application specially adapted for generalized sketches. It gives an outline of the interconnections between important entities of the drawing tool machinery, but also goes further in details on subjects where it feels natural to do so.
Chapter 6 summarizes prospective features, while giving the current status of the program and looking on immediate further work.

2 HISTORICAL BACKGROUND

2.1 Software engineering and the rise of object-oriented methodologies

In history of computing, the most noticeable trend in hardware development is the size reduction, or shrinking, together with rapidly increase in performance.
The advent of the *integrated circuit* (or *microchip*) in the late 1950's allowed the development of much smaller machines. On November 15, 1971, Intel released the world's first commercial *microprocessor*, a more advanced variant of the integrated circuit invented by Ted Hoff. The microprocessor led to the development of fourth generation computers, *microcomputers*. Those were small, low-cost computers that could be owned by individuals and small businesses. Coupled with one of Intel's other products, the RAM chip, the microprocessor gave fourth generation computers significant performance boosts, and allowed them to be smaller than ever. The evolution of microprocessors has continued ever since, and been known to follow Moore's Law when it comes to steadily increasing performance over the years.

> "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer."
>
> *Moore's Law*

The perhaps most popular formulation of this law is of the doubling of the number of transistors on integrated circuits (a rough measure of computer power) every 18 months, and unconsciously, this rule has been generally followed since the early 1970s, continuously providing faster, cheaper, and more reliable computer hardware.

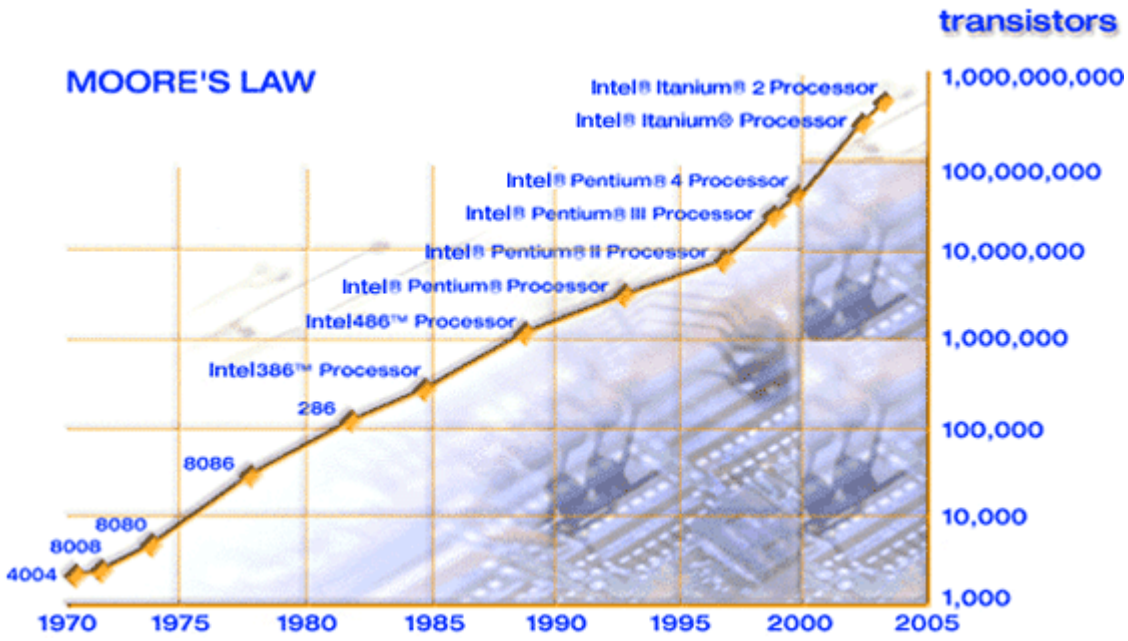


Figure 2-1: Moore's Law and Intel's processors.

On the other hand, software was initially simply viewed as an accessory for hardware but began to receive more attention as hardware technology evolved. At the beginning, computers were programmed using machine code written in binary format [1]. Programming was a tedious task; software was error-prone and extremely difficult to maintain due to the bad readability of programs and the fact that programs had to be written using absolute addressing. This attracted not too many people, leading to a shortage of programmers. Expensive computers sat idle for long periods of time while software was being developed, and the costs of software often exceeded the costs of hardware. Those factors finally resulted in the development of *assemblers* and *assembly languages*, making programming possible using more meaningful alphanumeric languages rather than binary code. The evolution continued, and as from the mid 1950's, high-level programming languages were developed such as FORTRAN, ALGOL 58, and COBOL. These languages provided the programmer with expressions/syntax that closely resembled English, and so both writing software and maintenance of software became much easier. The high-level languages, often described as problem-oriented languages, abstracted away from machine specific details, making the programmer able to focus more on the problems to solve. Programming with such languages was however done in an *unstructured* way. The languages didn't have any built-in structure at all, and so all code was contained within the same block. As a result, these languages had to rely on execution flow statements such as the GOTO statement, to jump to specific sections in the code. Unstructured source code is notoriously difficult to read and debug, and is often referred to as *spaghetti code*, a tangled mass of jumps, calls, and returns that is difficult to follow. As of this, a new programming paradigm emerged in the 1960's; *structured programming*. Popular languages evolved to include well-defined control statements, subroutines with local variables, and other improvements. Using such *structured* or *procedural* languages, it became possible to write moderately large programs. However, it was not enough to be keeping pace with the steady evolution of computer hardware. More computer power together with changes in architectures and technology increased

the capabilities of computers and their domains. This eventually led to a bigger, more exigent market for software vendors. Programs had to solve increasingly larger tasks, becoming more complex than ever. The area of software development was still in an early, undeveloped phase, with no solution found to dramatically increase software productivity. Programmers had little experience on the field, and the design of steady larger systems turned out to be much more difficult than had been presumed. In spite of multiple efforts, the increasing demands of the customers were met at the expense of a decrease in the quality of the software systems delivered. The symptoms constituted what would be known as the *software crisis* [2, 3]. This crisis emerged at the end of the 1960's and manifested itself in several ways:

- Projects running over budget.
- Projects running over time.
- Software was of low quality.
- Software often did not meet requirements.
- Projects were unmanageable and code difficult to maintain.

The problems software developers faced in the efforts of designing reliable and robust software in a productive way, ultimately gave rise to a discipline of computer science called *software engineering*.

Software engineering mainly addressed the software crisis by implementations of various *processes* and *methodologies*, most notably Royce's *waterfall method* [4, 5], introduced in 1970. Together with structured languages they responded well on some of the problems with the software development process, but as previously mentioned, structured programming methodologies could only handle a certain limit of complexity and size. To put the ever growing power of computers to good use, software of much greater complexity were required. Although more complex, the software also had to be reliable and easy to maintain. As much as 80 percent [6] of software costs are not associated with the original efforts to develop the software, but instead are related to the continued evolution and maintenance of that software throughout its lifetime.

The main drawbacks of structured approaches were the often missing consistency between the data and the behavioral part within the overall system. Software developers had to map concepts of the real world into expected sequences of executing instructions.

The attempt to design and debug programs by thinking through the order in which the computer runs things, ultimately leads to software extremely difficult to understand. It merely breaks down when the complexity level is too high.

As a solution to these drawbacks, the concept of an *abstract data type* became popular within the 1980s. This concept then formed the base for the *Object-Oriented* paradigm and for the development of a variety of new object-oriented programming languages, database systems, as well as modeling approaches. Instead of thinking about processes and the decomposition of processes, focus was moved towards objects and their behavior, resembling concepts of the real world. Translation from real-world phenomena/objects (and conversely) was eased because there is a (generally many-to-one) direct mapping from the real-world into an object-oriented program. Programming in this paradigm is

known as *Object-Oriented Programming*, and has become the de facto standard in the development of large-scale systems.

> "As computers become more complex, humans should give up trying to think like computers. Instead, computers should be made to think like humans."
>
> *J. Martin, J. Odell, "Object-Oriented Analysis & Design"*

The fundamental concepts of object-orientation, and which all *Object-Oriented Programming Languages* are based upon, are *encapsulation*, *inheritance*, and *polymorphism*. Together they shape the characteristics of object-oriented technology. It is a packaging scheme that provides a clear modular structure for programs well arranged for defining abstract data types where implementation details are hidden and units have a clearly defined interface. This basically denotes that object-orientation is a black box technology in which all internal details are abstracted away from the user. Knowing how the objects behave and how to use them is enough. Object-orientation facilitates the creation of meaningful, highly reusable software units focused on particular application areas, and makes it possible to deal with the ever growing complexity of software components; objects/concepts may be complex internally, but analysts do not need to know the details (unless they have designed it).
Object-oriented software has shown itself to be more understandable because it is better organized and has fewer maintenance requirements, and are inherently more stable and reliable compared to those of conventional approaches. With the advent of highly popular object-oriented programming languages such as C++ (ready for mainstream use since the beginning of 1990s) and Java (late 1990s), the object-oriented paradigm has become the standard approach throughout the whole software development process. Although object-oriented techniques alone cannot provide the magnitude of change needed in software development, their use has resulted in significant productivity gains.

2.2 Graphical specification languages

> "The word 'modeling' comes from the Latin word *modellus*. It describes a typical human way of coping with the reality. Anthropologists think that the ability to build abstract models is the most important feature which gave homo sapiens a competitive edge over less developed human races like homo neandertalensis."
>
> *Hermann Schichl, Models and history of modeling*

In software engineering, abstract system specifications have to compress an informal description of real-world fragments (*universe*) into *compact* and *comprehensible* texts suitable for *communication* between system designers, programmers, experts on the universe as well as users of the systems to be designed. To accommodate these needs, a natural (and practically without alternatives) choice is to use graphical languages, and indeed, a vast variety of graphical notational systems have been developed for those purposes. In particular, extremely popular are *Entity-Relationship Diagrams* (*ERD*) [7] and the *Unified Modeling Language* (*UML*), each of which has become a kind of *de facto* standard in its area. These are examples of *graphical specification languages*; (more or less) formal languages that are used within software engineering during system analysis, requirements analysis and design to express abstract system models.

The usefulness of an abstract system model was already recognized in the 1970's, when structured methods were proposed as software development methods. These methods offered Entity-Relationship diagrams to model the data aspect of a system, and data flow diagrams or functional decomposition techniques to model the functional, behavioral aspect of a system [8].
The benefits of models also applied to other engineering disciplines besides software development, such as database management and design. It quickly became clear that modeling provided a way to cope with complexity, encourage collaboration, and generally improve design in all aspects of software engineering. The reason for this is that models abstract away unnecessary details, and rather emphasizes the overall communication and behavior of the system on a superior level in a visually clear way.
With the advent of object-oriented methods, the dependency on good modeling solutions has only increased, and the two most popular graphical specification languages used today, ERD and UML, will be discussed next.

2.2.1 Entity-Relationship Diagram (ERD)

The *Entity-Relationship Model* (*ER Model*) is a *data model* for high-level descriptions of *conceptual* data models. In information system design, data modeling is the analysis and design of the information in the system, concentrating on the logical entities and the logical dependencies between these entities (their relationships), and the produced artifact is a visual model describing the design. There are three levels of data modeling. Ranging from higher to lower level they are *conceptual*, *logical* and *physical*. The former two deals with logical concepts; entities and relationships between entities, without regards to how they will be physically implemented, while the latter deals with more practical matters, taking into account the physical storage constraints and requirements.
The ER Model provides a graphical notation for representing such data models in the form of *Entity-Relationship Diagrams* (*ERD*), a basic component of the ER Model.
Originally proposed by Dr. Peter Chen in 1976 in [9] as a way to unify the network and relational database views, the ER Model has been further extended and is today commonly used for relational database designs. It serves as the foundation of many system analysis and design methodologies, including *Object-Oriented Analysis and Design* (*OOA/D*), and of *CASE* tools and repository systems. Chen's original article has become one of the most cited papers in the computer software field, and has also been rated to be one of the most influential papers within Computer Science.
An example of a simple ER Diagram is shown in Figure 2-2, stating the existence of a *Man*-object and a *Woman*-object, both being a type of *Person* as indicated by the special double body arrows. It also states a *relationship* between the *Man*- and the *Woman*-object through the *Married*-object, explicitly stating that each *Married*-object is a pair of *Man*-object and *Woman*-object, and which is specially designated by a diamond in contrast to rectangle entity nodes, thus becoming intrinsically different from other entity objects leading to heterogeneity of object classes.

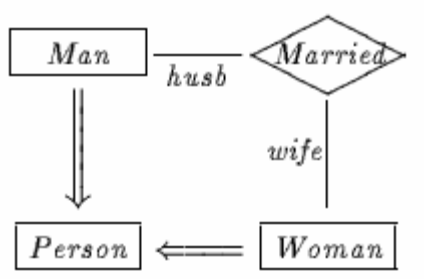


Figure 2-2: A simple ER Diagram.

2.2.2 Unified Modeling Language (UML)

> "The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems."
>
> *The Object Management Group (OMG)*

With the advent of object-oriented techniques, the need for modeling solutions increased with the growth in numbers and sophistication of software systems. The number of different modeling methodologies literally exploded, but the widely diverse efforts were inefficient in that they lacked the necessary collaboration to produce results that could be widely applied by the IT community. Rather, they contributed to what were affectionately called the "method wars", quarrels between method authors with their loyal followers over whom had the best solution [10]. As a result, tool vendors labored to support many different notations in the same tool, and companies struggled to identify and follow a single best method that could fully meet their needs.

Emerging from the "method wars" and the out coming need for a comprehensive modeling solution, the Unified Modeling Language (UML) aimed at truly becoming a standard that would address the practical needs of the software development community.

It is a welding of three major notations and a number of modeling techniques drawn from widely diverse methodologies that have been in practice over the previous two decades. It started with two prominent leaders on the field of methods and notations joining forces at Rational Software Corp in October, 1994. James Rumbaugh as the developer of the *Object-Modeling Technique* (*OMT*) with an emphasis on the analysis of business and data intensive systems for defining a target problem, and Grady Booch, the man behind the *Booch method* with particular strengths in design and implementation, defining and mapping a solution to the target problem. Together Rumbaugh and Booch attempted to merge their two approaches into a Unified Method.

In the fall of 1995, Rumbaugh and Booch had completed the first draft of the merged method, now referred to as Unified Modeling Language version 0.8. But already in June the same year, the Object Management Group (OMG) had put out a call for a common modeling approach to reconcile the greater than 50 named approaches in the market. The result was that Ivar Jacobson, another prominent leader on the field, and his company Objectory joined Rational Software Corp with the purpose of integrating Jacobson's *Object-Oriented Software Engineering* (*OOSE*) into the UML standard. OOSE was based around the use case concept that proved itself by achieving high levels of reuse by facilitating communication between projects and users, and brought to UML the essential

user centric elements that completed the range of features to make UML the comprehensive standard that it needed to be to gain wide acceptance. Booch, Rumbaugh, and Jacobson, also known as the “three amigos”, established four goals for the Unified Modeling Language:

1. Enable the modeling of systems (not just software) using object-oriented concepts.
2. Establish an explicit coupling to conceptual as well as executable artifacts.
3. Address the issues of scale inherent in complex, mission-critical systems.
4. Create a modeling language usable by both humans and machines.

The result of this collaborative effort was the release of UML versions 0.9 and 0.9.1 in the fall of 1996. However, despite the fact that they sought feedback from the development community, they recognized the need for broader involvement if the UML was truly to be a standard. OMG and the UML Partners consortium were broached to finish what had been started. This consortium included a mix of vendors and system integrators: Digital Equipment Corporation, HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, and Unisys, and the result of their efforts were published in January 1997 as UML 1.0.
At the same time, another group of companies (IBM & ObjecTime, Platinum Technologies, Ptech, Taskon & Reich Technologies, and Softteam) was working on and submitted another proposal for UML. This new team eventually joined the UML Partners consortium, and the work of the two groups was merged to produce UML 1.1 in September 1997. UML 1.1 was approved by OMG, and since then, the OMG has assumed formal responsibility for the ongoing development of the standard, promoting in MDA. Most of the original consortium members still participate however. The UML standard has progressed through several versions, and is now in version 2.0.
Despite early competition from existing modeling notations, UML has become the industry-standard language for modeling object-oriented software for nearly 70 percent of IT shops [10], simplifying the complex process of software design by creating a “blueprint” of construction. It is a rich but large specification language, and as from UML 2.0 defines thirteen types of diagrams, divided into three categories:

- **Structure Diagrams** include the Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.
- **Behavior Diagrams** include the Use Case Diagram, Activity Diagram, and State Machine Diagram.
- **Interaction Diagrams** include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

2.2.3 Problems with modern modeling languages

Models should be formalized in order to be easily (the more so, computer-aided) transformed into software specifications on the further stages of design. Thus, a good modeling language should be *graph-based*, *formalized* and sufficiently *expressive* to capture all the peculiarities of the real world. To the current state of the art in modern graphical specification languages, the synthesis has not been achieved; specification languages employed are either semi-formal, or have a very restricted expressive power, or both.

Well recognized among many graphical notation systems are the presence of ambiguous notations and the lack of a clear *semantics*, where attributes regarding semantics in the context of modeling languages are being precise and have detailed meanings of concrete constructs of the language, and where being precise means being formal. They suffer from *semantic relativism*, that is, different interpretations from different point of views. This lack of formal semantics or expressiveness deprives these modeling languages the creative powers of high-level specifications intended in the first place in the very setting of graphical specification goals.

What is not a well recognized fact though is that these shortcomings of modern modeling languages originate in poor logic foundations. The point is that any specification – as it is presented to its reader – is actually a *visual presentation* of a certain underlying *logical specification*. Thus, the problem is in specificational logics rather than in notational tricks; in a healthy modeling language *ML*, its notational organism (left side of Figure 2-3) is built upon a solid logical skeleton determined by formal semantics of *ML*'s constructs (right side). In contrast, the lack of formal semantics destroys the chain and removes reasonable foundations in the choice of specificational logic and syntax, ultimately leading to arbitrary syntax with unintentional synonymy and homonymy, and arbitrary (if any) specification logic, in which is unfortunately the case with many popular modeling languages used in software engineering.

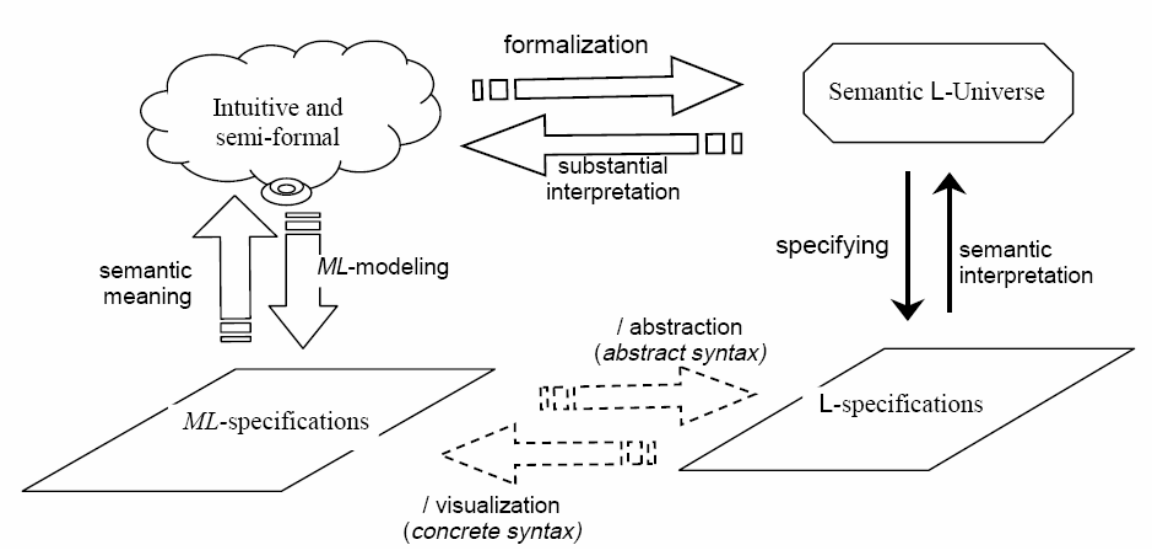


Figure 2-3: Formal logic for a modeling language: ML – a language, L – a logic.

Development of languages integrating the desired properties, that is, being *graph-based*, *formalized*, and sufficiently *expressive*, is just in the focus of *category theory*. Its experience has shown that dealing with graphical yet formalized specifications requires a special kind of thinking (*arrow thinking*) and the corresponding machinery (*arrow logic*) [11], and the success hardly can be achieved with *ad hoc* approaches built from scratch. This explains the far from ideal situation with modern graphical specification languages, and, on the other hand, suggests that category theory should possess a great potential for software engineering applications.

2.3 Category theory

"Category theory is, definitively, the mathematics of the internet-age (and beyond)!"
Jose Luiz Fiadeiro, "Categories for Software Engineering"

Introduced in 1945 by Samuel Eilenberg and Saunders Mac Lane, category theory is in mathematics an alternative fundament to set theory, dealing in an abstract way with mathematical structures and relationships between them. The original purpose of category theory was to establish a uniform framework to speak about *isomorphisms* and *natural equivalences* appearing in different areas of mathematics, in particular algebra and topology [12]. Since then, category theory has been further developed, and shown itself as a unifying notion influencing not only almost all branches of structural mathematics, but also the development of several areas of theoretical computer science. It arose from the fundamental idea of representing a function by an arrow, and indeed, a distinctive attribute of category theory as a mathematical formalism is exactly that it is essentially graphical. This means that most concepts and properties can be defined, proved and/or reasoned about using diagrams of a formal nature [13], what are called *commutative diagrams*. This diagrammatical nature of category theory is one aspect that makes it so applicable to software engineering.

Formal descriptions in mathematical logic are traditionally given as formal languages with rules for forming terms, axioms and equations. Algebraists long ago invented a formalism based on tuples, the method of signatures and equations, to describe algebraic structures. For example, within set theory a mathematical object can be characterized only by describing its inner structure, that is, it can be separated into different parts and elements and the interrelations between these inner components can be represented by means of ordered tuples.

Category theory on the other hand, can be presented as the branch of mathematics that, par excellence, addresses *structure*. Instead of focusing merely on individual mathematical *objects* and being able to characterize an object only by describing its inner structure, as has been the case with traditional mathematical theories, category theory takes an opposite viewpoint and emphasizes the *morphisms* – the structure-preserving mappings – between these objects, that is, the connections and interactions between the object in question and the surrounding objects. This basically means that category theory provides a more implicit way of characterizing objects, and it does so in terms of their social life, or more accurately their specific role within the net of relationships among all objects in the universe of discourse [12], as identified by *universal properties*. Thus, the study of categories is an attempt to capture what is commonly found in various classes of related mathematical structures, and so a category can be seen as a structure that formalizes a mathematician's description of a type of structure [14]. This is the role of category as theory.

The basic idea underlying the approach of category theory consisting of specifying any universe of discourse as a collection of *objects* and *morphisms*, which normally are in function of context, mappings, or references, or transformations or the like between objects, has an interesting side effect; as a result, the universe can be specified by a directed graph whose nodes are objects and arrows are morphisms. Formally, to specify a graph, its *nodes* (or *objects*) must be specified together with its *arrows*. Each arrow must have a specific *source* (or *domain*) node and a *target* (or *codomain*) node. This is given in Definition 2-1 below.

Definition 2-1: Graphs

A graph is a tuple $< G_0, G_1, src, trg >$ where:

- G_0 is a collection (of nodes).
- G_1 is a collection (of arrows).
- src maps each arrow to a node (the source of the node).
- trg maps each arrow to a node (the target of the node).

Such that for each arrow $f : x \to y$, $src(f) = x$ and $trg(f) = y$.

In addition, graphs have a "social life" of their own that is useful to know about. Relationships between graphs are called *graph homomorphisms*, as defined in Definition 2-2.

Definition 2-2: Graph Homomorphisms

A *homomorphism* of graphs $\varphi : G \to H$ is a pair of maps $\varphi_0 : G_0 \to H_0$ and $\varphi_1 : G_1 \to H_1$ such that for each arrow $f : x \to y$ of G we have $\varphi_1(f) : \varphi_0(x) \to \varphi_0(y)$ in H. That is, nodes are mapped to nodes and arrows to arrows but preserving sources and targets.

The concept of a graph is also a precursor to the concept of a category itself: a category is, roughly speaking, a graph in which arrows can be composed. Categories provide an abstraction over graphs by making arrows the working elements – as captured by *morphisms*. Compositions of arrows, or paths, provide richer information about "social life" than just one-to-one relationships. For that purpose, categories add to graphs an identity map that "converts" nodes to morphisms (null paths), and a composition law on morphisms that internalizes path construction. Morphism composition is required to be associative as for path concatenation, and the identities are just the identities with respect to composition. This is all summarized in Definition 2-3 of categories below, where G_2 is given by $G_2 = \{fg \mid f, g \in G_1, trg(f) = src(g)\}$ (G_n denotes the set of paths of length n).

Definition 2-3: Categories

A category C is a triple $< G, ;, id >$ where:

- G is a graph, often denoted by $graph(C)$.
- ; is a map from G_2 into G_1 (called the *composition law*); for every fg in G_2, we denote by $f;g$ the arrow that results from the composition.
- id is a map from G_0 into G_1 (called the identity map); for every node x, we denote by id_x its identity arrow.

And for every f in G_1, fg in G_2, and fgh in G_3:

- $src(f;g) = src(f)$ and $trg(f;g) = trg(g)$.
- $src(f;g) = src(f)$ and $trg(f;g) = trg(g)$.
- $src(id_x) = trg(id_x) = x$.
- $(f;g);h = f;(g;h)$.
- If $f : x \to y$, $id_x ; f = f ; id_y = f$.

A popular example of a category is the category SET where objects are all sets, morphisms are all total functions between sets, composition is functional composition – i.e. $(f;g)(x) = g(f(x))$ – and the identity map assigns to every set the identity function on that set. Because function composition is associative and the identity map is both a left and right identity for function composition, all conditions are met.

A category is also a kind of mathematical structure, and hence has a social life on its own. This is captured by the corresponding notion of morphisms between categories, called functors. A functor is a structure-preserving map between categories, in the same way that a homomorphism is a structure-preserving map between graphs. A formal definition of functors is given in Definition 2-4.

Definition 2-4: Functors

Let C and D be categories. A *functor* $\varphi : C \to D$ is a graph homomorphism from $graph(C)$ into $graph(D)$ such that:

- $\varphi_1(f;g) = \varphi_1(f);\varphi_1(g)$ for each path fg in $graph(C)$.
- $\varphi_1(id_x) = id_{\varphi_0(x)}$ for each node x in $graph(C)$.

Category theory supports and encourages forms of diagrammatic reasoning, also called *diagram chasing*; a practice that is consistent with the modern culture in computing. Contrary to what often happens in software engineering, the notion of diagram is formal and the reasoning that can be done with diagrams is mathematical. Whereas objects and morphisms provide the main elements of the vocabulary used in category theory, diagrams provide the basic sentences that can be built to express concepts. Diagrams are formally defined in Definition 2-5.

Definition 2-5: Diagrams

Let C be a category and I a graph. A *diagram* in C with *shape* I is a graph homomorphism $\delta : I \to graph(C)$. I is called the *shape graph* of the diagram δ.

A diagram D in the sense of Definition 2-5 in the category SET is pictured on the page with a drawing of nodes and arrows as for example pictured in Figure 2-4 b), with a shape graph as pictured in Figure 2-4 a), defined by $D(i) = A$, $D(j) = B$, $D(k) = C$, $D(u) = f$, $D(v) = g$ and $D(w) = h$.

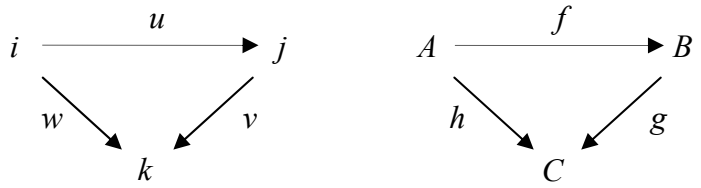


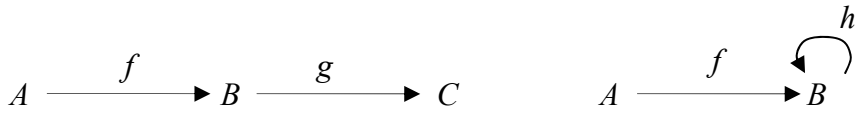
Figure 2-4: A diagram with its corresponding shape graph.

Though it may seem to have little to do with what are informally called diagrams, the definition of a diagram as a graph homomorphism, with the domain graph being the shape, captures both the following ideas:

- A diagram can have repeated labels on its nodes and on its arrows.

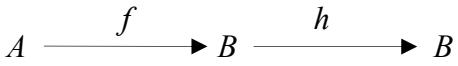
- Two diagrams can have the same labels on their nodes and arrows but be of different shapes.

Consider the two diagrams:



a) b)

These are clearly of different shapes. But the diagram



c)

is the same shape as a) even though as a graph it is the same as b). Diagrams b) and c) are *different diagrams* because they have different shapes.
When the target graph of a diagram is the underlying graph of a category some new possibilities arise, in particular the concept of *commutative diagrams*, which is the categorist's way of expressing equations. That is, the property that a diagram commutes establishes a set of equalities between arrows. Hence, diagrams and commutativity provide the capability of doing equational reasoning in a visual form, an advantage that has not been fully exploited yet in software engineering.
An example of the arrow reasoning that is typical for category theory can be given with the categorical formulation of an *injective* function, as represented by a *mono*, and the somehow related notion of *jointly mono* which are defined below.

Definition 2-6: Mono and jointly mono

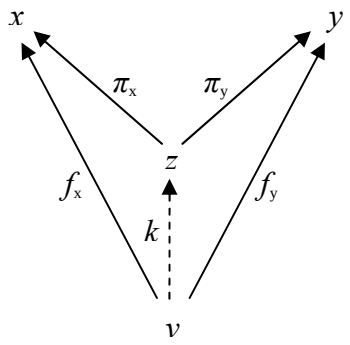
Consider an arbitrary category C and morphism $f : x \to y$ in C. f is a *monomorphism*, or a *mono*, or *monic*, if and only if, for every pair of morphisms $g, h : z \to x$, $g; f = h; f$ implies $g = h$.

Now, two morphisms $f_1 : x \to y_1$ and $f_2 : x \to y_2$ are *jointly mono* if and only if, for every pair of morphisms $g, h : z \to x$, $g; f_1 = h; f_1$ and $g; f_2 = h; f_2$ implies $g = h$.

Further, an example of a universal property is the one of *product*. In category theory, a *product* handles the relationships from the environment towards collections of two unrelated objects. A formal definition of products follows in Definition 2-7.

Definition 2-7: Products

Consider an arbitrary category C and two C-objects x and y. A C-object z is a product of x and y with projections $\pi_x : z \to x$ and $\pi_y : z \to y$ if and only if for any object v and pair of morphisms $f_x : v \to x,\ f_y : v \to y$ of C there is a unique morphism $k : v \to z$ in C (often denoted by $< f_x, f_y >$) such that $k;\pi_x = f_x$ and $k;\pi_y = f_y$, or that the following diagram commutes:

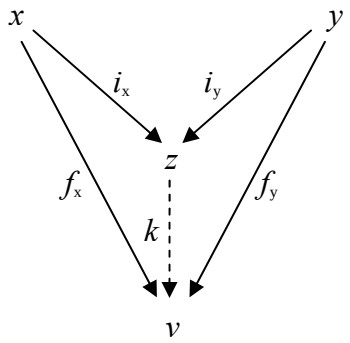


For example, in the category SET, the product of two sets is given, up to isomorphism, by their Cartesian product, and the morphism $k =< f_x, f_y >$ by $< f_x, f_y > (c) = (f_x(c), f_y(c))$ for every $c \in v$. π_x and π_y are jointly mono but not mono.

Every construction in category has a dual concept, obtained by reversing the direction of all arrows in the graph. The dual notion of product is *coproduct*, also called *sum*. A sum in category theory handles the relationships that a collection x, y of objects has towards its environment. A formal definition of sums follows in Definition 2-8.

Definition 2-8: Sums (coproducts)

Let C be a category and x, y objects of C. An object z is a *sum* (or *coproduct*) of x and y with injections $i_x : x \to z$ and $i_y : y \to z$ if and only if for any object v and pair of morphisms $f_x : x \to v,\ f_y : y \to v$ of C there is a unique morphism $k : z \to v$ in C (often denoted by $[f_x, f_y]$) such that $i_x;k = f_x$ and $i_y;k = f_y$, or that the following diagram commutes:



For example, in the category SET, the sum of two sets is a disjoint union and the morphism k is defined by case distinction as expressed by the equations $i_x ; k = f_x$ and $i_y ; k = f_y$.

That many kinds of constructions within different areas of mathematics, like conjunction/disjunction, intersection/union, product/sum etc., can be described by universal properties without reference to the inner structure has been a crucial discovery of category theory. In fact, it is proven that category theory is absolute expressible; any higher-logic specification can be described in category theory as well.

The abstract viewpoint of category theory to look only at the morphisms between objects and not inside the objects has two obvious consequences. Firstly, objects can be described only up to *isomorphism*, that is, independently from a particular representation and implementation. Secondly, category theory is mainly devoted to problems with a rich and complex structure of relationships between objects. This focus on social aspects of object lives is exactly the reason for the applicability of category theory to computing in general, and software engineering in particular.

Current software development methods, namely object-oriented ones, typically models the universe as a society of interacting objects as an attempt at tackling the increasing complexity of modern software systems. Category theory is ideal for exactly this type of reasoning. It manages to abstract away from unnecessary implementation details, and is even well fitted for automation processes due to its formal nature, and the fact that precisely defined mathematical procedures can be used for interpretations and translations of specifications, but is in its current state not immediately applicable for systematic use in software engineering. Due to the existence of compositions, categories modeling complex universes are usually infinite because they (implicitly) specify also a lot of derived information about the universe. On the other hand, in practice one deal with finite presentations of infinite categories. Thus, a presentation-oriented trend in category theory is associated with the concept of *sketch* invented by Charles Ehresmann in France in 1968. Ehresmann's sketches were directly intended for *finite* and *effective* presentation of complex mathematical structures but are too restrictive for software engineering as they tend to produce auxiliary nodes and arrows, and hence, do not manage to give *comprehensible* models which are a must if specifications are intended for real usage by software engineers. Because of this, a proposal was made by M. Makkai [15] and independently by Z. Diskin[1] [16] of a generalization towards much more flexibility of the

[1] In addition to having arbitrary diagram predicates, these sketches also have arbitrary operations.

sketch language in order to make adaptation of the arrow framework for practical needs real. Besides commutativity and the universal properties, it was desirable to be able to use any appropriate properties of diagrams (see [11, 17] for more information on this subject). Makkai called them *generalized sketches*.

2.4 Generalized sketches

Generalized sketches introduce a general notion of diagram predicates. An arrow specification with diagram predicates is a generalized sketch, and it follows from the general results of categorical logic that any specification which can be described formally (say, in higher order logic) can be replaced by a corresponding equivalent sketch[2], that is, any formal construction can be expressed by a generalized sketch as well.
Despite the absolute expressiveness of the generalized sketch language, its vocabulary of basic terms is strict and extremely brief and even more, there is a fixed (and not very big) collection of diagram predicates and operations, compositions of which cover all formal structures. Roughly, a *generalized sketch* is a graph in which some diagrams (fragments of the graph closed in some technical sense) are marked with predicate labels taken from a predefined *signature*. That is, the vocabulary of specificational constructs is set by a collection (signature) of diagram predicates, and a specification (sketch) appears as a graph in which some diagrams are labeled by markers from the signature.
Syntactically, a diagram predicate is a marker having an *arity* shape designating those graphs on which the marker can be hung. Semantically, a marker is interpreted as a certain property of systems whose configuration fits in with the shape. The labels (markers) are predicate symbols and marked diagrams are nothing but predicate declarations, to be understood as statements about nodes and arrows occurring in the diagram.
This gives the following outline:

- A sketch is a (usually finite) graph together with diagrams for a fixed signature.
- A signature is a collection of diagram predicates.
- A diagram predicate is a name together with an arity.
- An arity is a fixed shape.

And so any graphical specification, whether it is an ER diagram, a UML class diagram, use case diagram or etc., can be considered as sketches for a fixed signature, where the signature is its corresponding diagram type.
Continuing from the example of the ER diagram shown in Figure 2-2, a relation in categorical terms is a set R together with a jointly monic family of arrows (projections) $p_i : R \to E_i, 1 \le i \le m,$ so that specifying internal structure of R-objects is removed to an arrow diagram adjoint to the node. A standard categorical way of specifying jointly monic families is with a product together with a monomorphism, and from Definition 2-6 and Definition 2-7 this obviously introduces auxiliary nodes and arrows. Hence, a much

[2] Further on in the thesis, the term *sketch* means *generalized sketch*. Both will be used alternately.

more direct way would be to introduce a predicate of being jointly monic for families of arrows with a common source, and then declare that the family $(p_1,...,p_m)$ satisfies the predicate. By denoting such declarations with an arc, the required specification of the class *Married* will look as shown in Figure 2-5, which is a corresponding generalized sketch for the ER diagram of Figure 2-2.

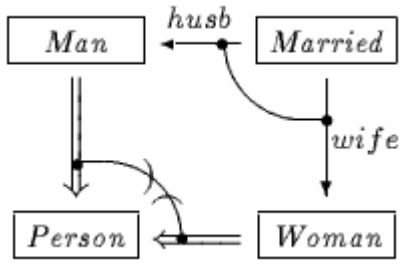


Figure 2-5: A simple generalized sketch.

The *Married*-object is no longer associated with any predicates, but instead, is distinguished by predicates on diagrams, thus giving rise to a general notion of *arrow diagram predicate* and providing a shift in paradigm by rather specifying internal structures of classes externally via diagram predicates.

Figure 2-5 also contains two other diagram markers; the double body of arrow and the arc with brackets. The former is a marker that might be hung on diagrams consisting of a single arrow and denotes the property of being IsA-arrow. The arc marker denotes the property of commonly targeted family of functions to cover the target disjointly, that is, each object in the target class is in the image of one and only one source class. Thus, the specification states that (i) each *Person*-object is either *Man*-object or *Woman*-object but not both and (ii) each *Married*-object is actually a pair $(m,w) \in Man \times Woman.$

Marked diagrams in a sketch must be easily recognizable; otherwise sketches lose their benefits of evident graphical specifications. So, the shape graph D of a predicate (marker) should occur in the carrying graph G of the sketch without topological deformations, that is, as is standard in category theory, a diagram appears as a visually clear presentation of a graph homomorphism $\delta : D \to G$.

Shape graph of predicates (arity shape) and carrier graphs of sketches must be labeled by names and diagrams are graph homomorphisms compatible with naming. For example, the shape of identity is an arrow between two different nodes with the same name so that an identity diagram in a labeled graph is an arrow for which names of the source and target nodes coincide.

A diagram predicate whose arity shape contains multiple arrows, all having common source or target, is typically visualized by an arc spanning over arrows. Otherwise, as long as the arity shape contains multiple nodes and/or arrows, the diagram predicate is typically visualized by a string-based marker over the corresponding diagram.

Diagram predicates whose arity shape contains only a single node however – a special case of diagrams – is visualized by the node itself. Thus, the predicate symbol is a predicate symbol on nodes, and these special cases of diagram predicates can therefore also be referred to as node predicates.

In addition, in the same way it is possible to have predicates on nodes, it is also possible to have predicates on arrows – diagrams with a single arrow. The key point in semantics of sketches is how to interpret arrows, and these arrow predicates are means of intermediary their interpretation.
Thus, a signature should be able to define three types of predicate symbols, being markers for:

- Predicates on nodes, to be interpreted as node constraints.
- Predicates on arrows, to be interpreted as arrow constraints.
- Predicates on diagrams (more than one arrow), to be interpreted as diagram constraints.

So, a sketch specification is a graphical construct that consists of only three kinds of items: (i) nodes, (ii) arrows, (iii) marked diagrams, i.e., labeled collections of nodes and arrows, to be interpreted as constraints imposed on diagrams labeled by these markers. But a sketch specification is also a precisely defined formal construct, and can therefore be studied by mathematical methods. This is subject to category theory, and the basic framework is already established.
Among the principal advantages of generalized sketches are the following:

- **Nice amalgamation of logical rigor and graphical evidence.** Generalized sketches are graph-based images yet they are precise formal specifications.
- **Universality, in the precise sense of the word.** It can be mathematically proven that any specification whose semantic meaning can be formalized, can also be expressed by a generalized sketch, or in other words, is sketchable.
- **Unifying power.** Many graphical specification languages can be simulated by generalized sketches in the corresponding signature of diagram markers. That is, each graphical notation, say ERD or the different types of diagram in the UML language, corresponds to a given signature.
- **Semantic capabilities.** The generalized sketch language is inherently object-oriented and provides a quite natural way of specifying OO class-reference schemas.
- **Easy and flexible modularization mechanism**. A complex specification can be presented by a generalized sketch whose nodes are sketches and arrows are sketch mappings. This pattern can be reiterated if necessary.

Clear distinction between logical specifications and its visualization is provided by the presence of formal semantics for generalized sketches, and makes the sketch notation favorable in comparison with many notational systems in software engineering. Sketch specifications enjoy a unique combination of rigor, expressiveness and comprehensibility, and provide a unified specification framework for the entire field of software engineering. The diversity of graphical notations can be translated into a variety of sketch models in different signatures by converting their vocabulary of specificational constructs into signatures of diagram predicates, in order for their specifications to be converted into

generalized sketches. Sketches in different signatures are nevertheless sketches, and they can be uniformly compared and integrated via relating/integrating their signatures. This task is precisely formulated and can be approached by mathematical methods already developed in category theory [18].
The flexibility [17] of generalized sketches as a modeling (correspondingly, specification) tool is provided by existence of different substantial interpretations of sketch items. The following interpretations are possible:

1. *Semantic data modeling*: nodes are sets and arrows are functions.
2. *Object oriented design*: nodes are object classes and arrows are references.
3. *Functional programming*: nodes are data types and arrows are programs (procedures).
4. *Logic (and logic programming)*: nodes are statements (propositions) and arrows are deductions (proofs).
5. *Process modeling*: nodes are interfaces and arrows are processes.
6. *Transaction modeling*: nodes are data states and arrows are transactions.
7. *Meta-modeling*: nodes are data (process) schemas (e.g. sketches) and arrows are mappings between them (functors).

For further information on generalized sketches, see references [11, 17-19].

3 PROBLEM ANALYSIS

3.1 Task

The core of this thesis is the development of a drawing tool software application for *generalized sketches*; a highly expressive mathematical formalism with a special diagrammatical notation. It is a four-way separated workmanship, in which the following parts are involved:

- Obtaining the internal structure of generalized sketches.
- Implementing the machinery of a drawing tool application.
- Persistence of data.
- Implementing functionality specifically adjusted for generalized sketches.

First item is to a large degree self-explanatory. It is a purely theoretical subject consisting of searching various mathematical literatures on the fields of category theory and its role in software engineering, and of the generalized sketch formalism itself.
Second item is on the other hand perhaps a more diffuse area. It involves the work of a drawing surface in which the user interacts with, and a related set of tools that define all possible operations that could be performed on the drawing surface. This especially embraces which elements it is possible to draw (draw objects), and the management of these draw objects.
Third item is for late use by the tool and perhaps for communication with future tools. It involves data persistent methods for signatures and sketches.
Item four deals with pure functionality issues of the software application regarding the generalized sketch formalism, in which most of it has been absorbed by studying the existing solution that is discussed later in section 3.2, and some taken from literature concerning generalized sketches.
It should be noted however that the topic this thesis builds upon is quite huge, and that the generalized sketch formalism possess a great deal of features and possibilities. Thus, no final version of a drawing tool software application for generalized sketches is expected to result from the present thesis due to its evident limitations in time. Instead, the intention was that the program should serve as a basic drawing tool framework for the generalized sketches formalism, well arranged for further extensions. Hence, several demarcations on the functionality of the program have been necessary to make, and an approximate measure of the specific functionality to be implemented is examined in section 3.1.1. For a more complete list of possible functionality of the generalized sketches formalism to be implemented, but also as a drawing tool program in its entireness, see section 6.3.

3.1.1 Functional requirements specification

Functional requirements can be divided into two main parts; the ones regarding standard drawing tool characteristics, and the ones regarding the generalized sketch formalism. Firstly, for the drawing tool aspect of the thesis the main focus is on drawing a kind of graphs, i.e. nodes and arrows between nodes. This involves the functionality for:

- Specifying new nodes at arbitrary positions with arbitrary size, and with an optional name to be positioned within the node.
- Specifying new arrows that are attached to existing nodes, and with an optional name to be positioned next to the arrow.
- Maintain integrity of graphs; all arrows should at all times be connected to a source node and a targeting node.

Further on, it should be possible to manage already drawn objects, involving:

- Selecting (possible multiple) nodes and arrows.
- Repositioning nodes and arrows.
- Resizing nodes.
- Removing nodes and arrows.

Secondly, for the requirements regarding generalized sketches, this roughly involves:

- To set a signature of diagram markers.
- To draw and edit sketches in a given signature.

A more thoroughly examination of these requirements produces the following operations:

- Create new signatures.
- Change names and descriptions of each element in signatures.
- Add node-, arrow-, and diagram- constraints to signatures.
- Change node-, arrow-, and diagram- constraints in signatures.
- Remove node-, arrow-, and diagram- constraints in signatures.
- Save or discard changes made to a signature.
- Open earlier saved signatures.
- Create new sketches based on a given signature.
- Add nodes and arrows between nodes to a sketch.
- Add and remove predicate declarations on nodes and arrows occurring in a sketch.
- Add and remove predicate declarations on diagrams occurring in a sketch.

- Remove nodes and arrows occurring in a sketch.
- Save or discard changes made to a sketch.
- Open earlier saved sketches.

In addition, in order to create custom markers of the constraints in a signature, the application should offer a range of dialogs that allows the user to juggle with the markers, providing a rich selection of various custom applicable visual styles, including:

- Custom shapes for nodes
- Custom arrowheads for arrows
- Custom colors
- Custom dash styles for lines
- Custom width for lines
- Custom fill patterns

3.1.2 Non-functional requirements specification

Important non-functional requirements of the software application cover the following:

- **Low response time:** As a GUI drawing application, the program should respond to user inputs immediately for serving as a user friendly program.
- **Appealing:** Offering a professional and nice looking GUI increases the user-friendliness of the program.
- **Good and intuitive user interfaces:** Put to practice, it is important to keep a consistence behavior of the tool, preferably with evident user interfaces.
- **Extendable:** Due to time limitations, and because of the many features and possibilities of the generalized sketch formalism, not all functionality can be expected to be implemented in the present thesis. Therefore, the work should result in a framework providing the basic functionality designed for extensibility.

3.2 Existing solution

Currently there exists a drawing tool program developed for the generalized sketches language, but it is old and unfortunately also a little unstable, thus not fitting as a commercial product. This program will be discussed shortly where the functionality and architecture will be explored in detail, but first a little history about the origin of the program.

3.2.1 History

In 1995, a group from Latvia implemented a Windows 16-bit version of a drawing tool for generalized sketches. At that time they didn't have the new, revolutionary 32-bit version of the Windows platform, Microsoft Windows 95. Later on, in 1996-1997, when they did a pure technical job of rewriting this tool for the 32-bit version of the Windows operating system, the company they worked for stopped the project due to some crash in funding. The principal programmer moved to USA shortly after, and in the rush they forgot to conserve the project accurately. The result is a 32-bit version with a few bugs, and it is therefore not too suitable for practical work. It is an MFC application written in the C++ language, designed for the Microsoft Windows platform, and will be referred to as Sketcher 95. An executable version of this application is still available. Unfortunately, neither source code, nor system documentation is easy to get at. Yet, this version will be valuable when designing the system, because it demonstrates some reasonable architectural decisions, and also gives great examples of possible user interfaces.

3.2.2 Description of solution

Sketcher 95 is an MFC (Microsoft Foundation Classes) application developed for 32-bits versions of the Microsoft Windows platform. It is implemented as an MDI (Multiple Document Interface) application, that is, it consists of a single parent window (the main application window) acting as a container for multiple child windows which typically display some sort of documents, hence the name Multiple Document Interface. A screenshot of Sketcher 95 is shown in Figure 3-1.

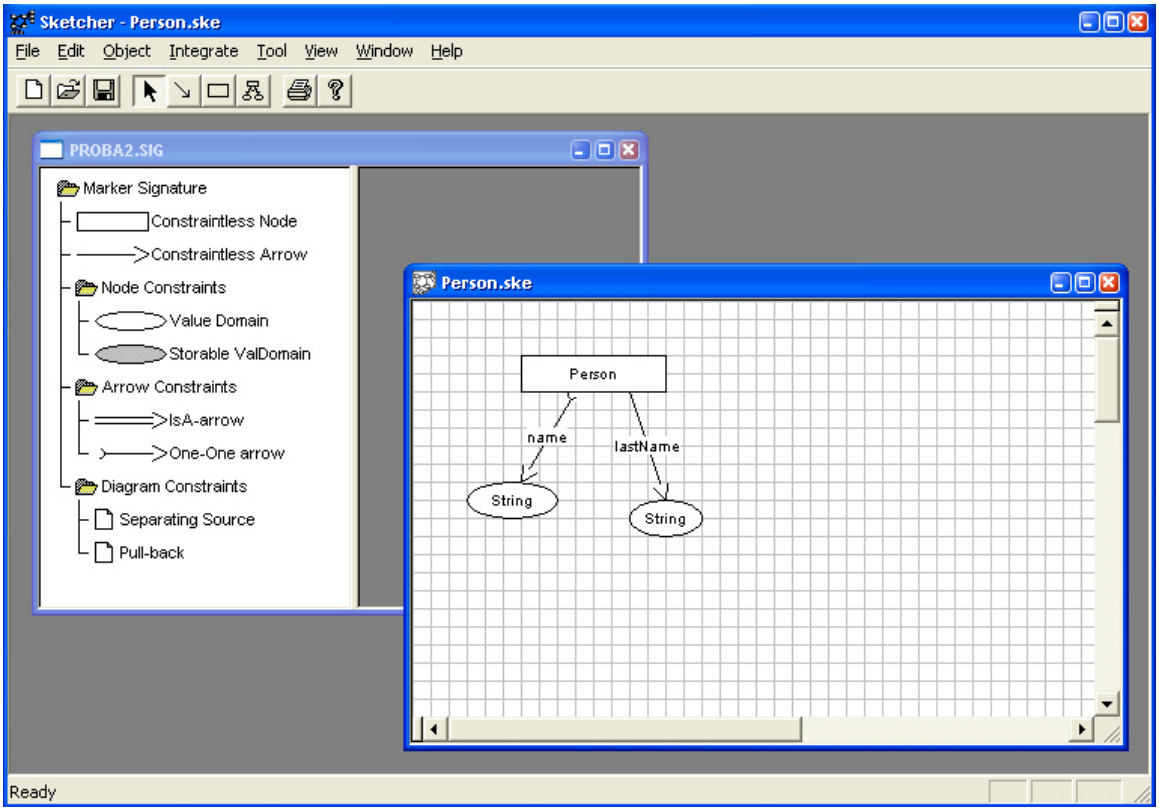


Figure 3-1: Sketcher 95 – A Multiple Document Interface (MDI) application.

The parent window of Sketcher 95 is edified by a menu, a toolbar that contains shortcuts to the most commonly used operations as offered by the menu, and a status bar positioned in the bottom. The child windows residing in the parent window of Sketcher 95 provide the user interfaces towards the two important main parts of the generalized sketches formalism; signatures and sketches. This way, Sketcher 95 manages to separate the concept of a sketch from the concept of a signature in a very clearly manner. Also, the properties of MDI make it possible to manage multiple signatures and sketches simultaneously in the same work session – each signature and each sketch is just represented in a new separate child window.

An important characteristic/aspect of the generalized sketches formalism however, is that every sketch is based upon a specific signature, and hence creates a dependency issue. Sketcher 95 has solved this problem by forcing the user to define an existing signature when creating new sketches, and to always have the corresponding signature file open when working on a sketch. The user cannot close any signature windows without closing all dependant sketches first.

The rest of the principal functionality provided by the program logically belongs to either one of the two main concepts of generalized sketches as represented by the child windows, and is dealt with in the subsequent sections 3.2.2.1 and 3.2.2.2, covering the signature domain and the sketch domain respectively.

3.2.2.1 The signature domain

A signature is represented by a tree view control, where the top root element of the tree is the name of the signature. As from section 2.4, a signature is defined as an organized collection of diagram predicates, or markers, to be interpreted as constraints imposed on diagrams labeled by these markers. The diagram predicates are divided into three parts; predicates on nodes, predicates on arrows, and (more complex) predicates on diagrams, i.e. collections of nodes and arrows. Hence, a signature holds a collection of node constraints, a collection of arrow constraints, and a collection of diagram constraints.
In addition, a signature has to at least define one predicate for nodes and one predicate for arrows, or else no nodes or arrows can occur in corresponding sketch specifications. These two predicates can be seen as very basic constructions as they are not meant to impose any constraints on nodes and arrows that are declared with this predicate. They can be interpreted as constraint less node and arrow predicates, respectively.
Together, the five elements discussed are the only allowed direct children of the root element as they constitute the framework of every signature. Their properties can be altered, but their meaning/interpretation remains the same. Nor is it possible to remove them – they are the vital parts of any signature.
In Sketcher 95 all collections in the signature is associated with a folder icon. It is possible to further organize each collection by creating additional subfolders, and hence, the signature becomes a sort of hierarchy, similar to file systems in computers, only that files in this context are the explicit constraints. Simple constraints, that is node and arrow constraints, are associated with a somewhat larger icon representing their visual appearance. This is a quite powerful way to easily manage such constraints.
Complex constraints on the other hand, the ones on diagrams containing multiple nodes and arrows, are just associated with an icon displaying a blank sheet, giving a clue that their notion is too compound to be represented by a simple icon. When selected, the shape of the diagram is displayed in a temporary sheet within the signature window, as demonstrated in Figure 3-2 below.

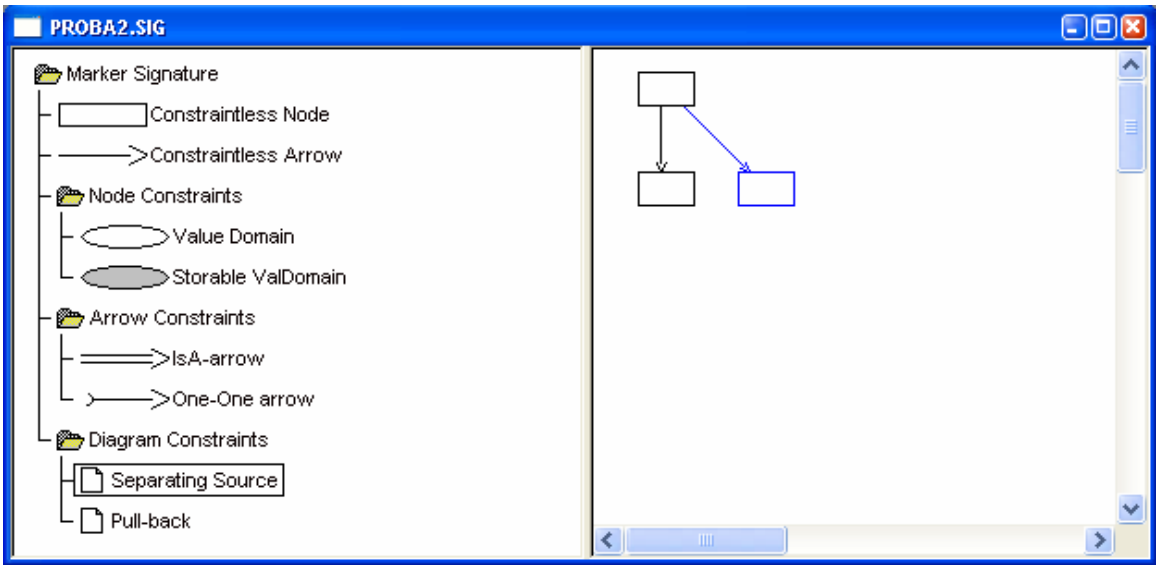


Figure 3-2: Separating source as diagram constraint.

This diagram constraint is a separating source, involving three arbitrary nodes and two arbitrary arrows with common source, but with distinct targets. While nodes and arrows normally are colored black, one of the target nodes and the arrow that targets it is colored blue, indicating that they are repeatable. That is, a separating source is not only restricted to two targeting nodes; as long as all arrows have common source and different targets, any number of targeting nodes higher or equal to two satisfies the shape of the diagram, and so, this constraint designates all diagrams in a sketch with common source and an arbitrary number of target nodes higher or equal to two.

The tree view control in the signature window provides the basic functionality of managing signatures, that is, adding and removing constraints. For changing constraints, it makes use of dialog boxes. The dialog box for node constraints is shown in Figure 3-3. It provides different predicates that can be used to change the visual appearance (marker) of a node, and in that regard, markers are parametric.

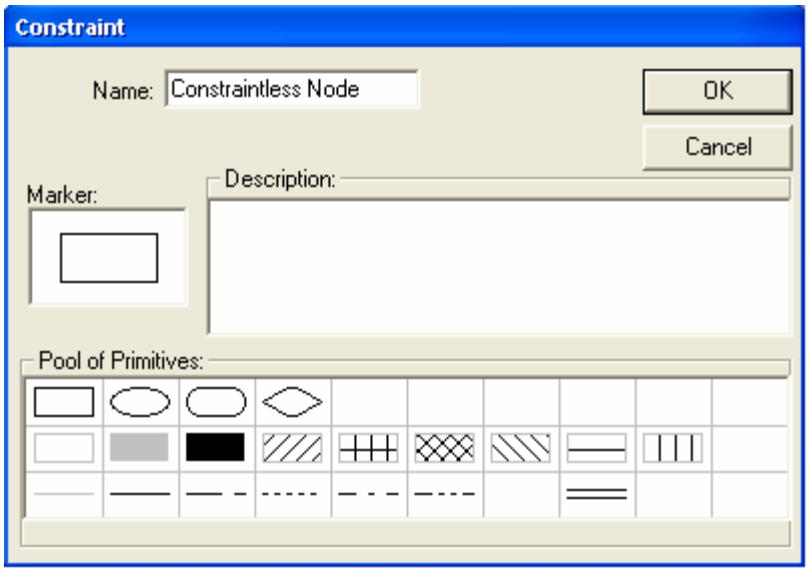


Figure 3-3: Dialog box for node constraints.

3.2.2.2 The sketch domain

The child window representing sketches is principally a drawing surface in which users may draw simple nodes and simple arrows between nodes, where simple means nodes and arrow with no constraints imposed on it – they are constraint less nodes and arrows, as represented by the corresponding signature. Also, it is possible to reposition and resize drawn nodes and arrows, and to remove them from the sketch.
All this functionality is available through tools on the toolbar, but also from the menu. These tools are represented by a node tool for drawing constraint less nodes, an arrow tool for drawing constraint less arrow, and a pointer tool used for selecting objects, resizing objects, and repositioning objects. Also, the pointer tool exhibits a popup menu whenever the user presses the right mouse button on a drawn object. This popup menu contains functionality for renaming and deleting the corresponding object, in addition to – if the object is either a node or an arrow – add, remove or change what constraints, as given by the corresponding signature, the object should satisfy, if any, through a properties dialog box. Such a dialog box for nodes is shown in Figure 3-4 below.

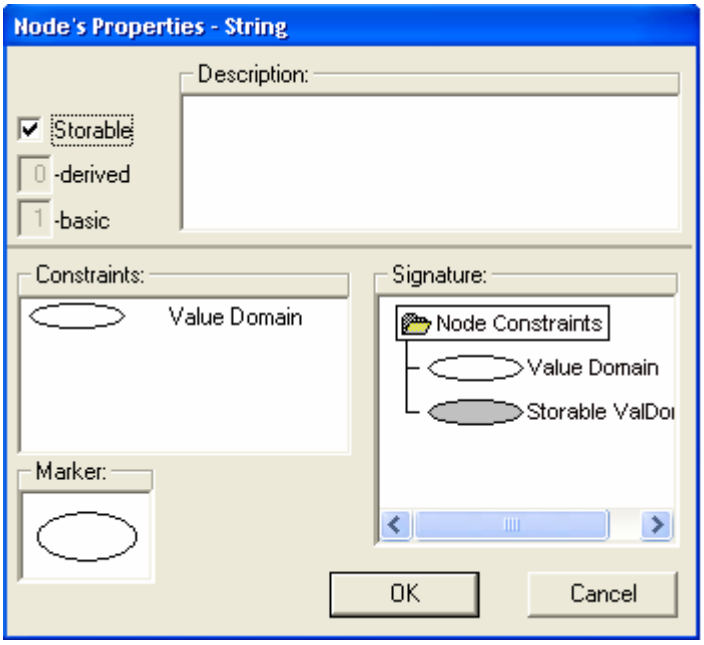


Figure 3-4: Dialog box for a node's properties.

At all times, nodes and arrows occurring in a sketch have the current visual appearance of the corresponding marker to which constraint from the signature it satisfies. In cases where a node or an arrow has several constraints imposed on it, the last attached constraint is the asserted one, and hence provides the marker to which the node or arrow has similar visual appearance. In cases where there are no constraints imposed (standard) on a node or an arrow, the marker used to represent the current object is the corresponding marker of the constraint less node or arrow, as given by the signature.
Besides the capabilities mentioned, the sketch window also provides the functionality to add constraints to complex diagrams appearing in a sketch, i.e. collections of nodes and arrows, through the diagram tool available from the toolbar as well as the menu. This operation is the equivalent of marking diagrams with predicate declarations.

3.2.3 Drawbacks in the solution

Unfortunately, Sketcher 95 is subject to some application errors. This is mainly due to the sudden stop in the project of converting the original 16-bit version to a 32-bit Windows 95 version. Because of this, the application is not satisfactory as a ready-to-use program. Much functionality of the generalized sketch formalism has been covered though, and works in the way a user would expect it to do. Where the program fails however, is in the common drawing tool behavior. Some erroneous painting and unpredictable application failures causing the program to close are the most serious flaws.
Less important is the lack of common GUI functionality and drawing tool functionality such as cut, copy, paste, undo, redo and zooming capabilities, as well as some selection tool shortcomings and lack of keyboard support on the drawing surface.

3.3 Challenges

At first, there seemed to be four obvious challenges with the thesis, namely those listed in section 3.1.
First of all, a basic understanding of the generalized sketches language itself was crucial in order to develop a dedicated drawing tool software application for it. The mathematical foundation of the language is both unaccustomed and difficult to grasp because it works in an entire different environment than standard set theory taught in school, and so, many mathematical concepts and ideas on the subject are very abstract and therefore not directly comprehensible.
Second, as no considerable experience on the subjects of graphical programming had been achieved in the past, managing the GUI application development model would probably bid on a few challenges on its own. But especially challenging was the drawing tool aspect of the thesis, where the main focus laid in both visual creation and representation of two dimensional diagrammatical sketches.
Third, saving states of the program was a necessity if the program was intended for practical use. A decisive feature was that of being standardized to facilitate communication with future applications.
Fourth, reflecting the inner workings of generalized sketches in a graphical application seemed like a non-trivial and time consuming task.
However, putting these challenges into consideration became the very reason why this thesis was so appealing and interestingly in the first place. Especially alluring was the complete development of a stand-alone window application and the drawing-oriented focus concerning representation of diagrams. Also, no restrictions were put on which technologies to use, and accordingly this thesis was an opportunity to explore technologies of personal interest and obtain a better knowledge of those.

4 AVAILABLE TECHNOLOGY

4.1 Developing platforms and tools

For the software in this thesis, there are four possible developing platforms of current interest: the Microsoft Windows operating system, the Linux operating system, the Java 2 platform and the Microsoft's competitive technology, Microsoft.NET. Important common features of these platforms are that they all run on personal computers and their ability of running applications with graphical user interfaces, thus allowing users to interact with programs visually. The platforms will be examined by turns in the following sections.

4.1.1 Microsoft Windows

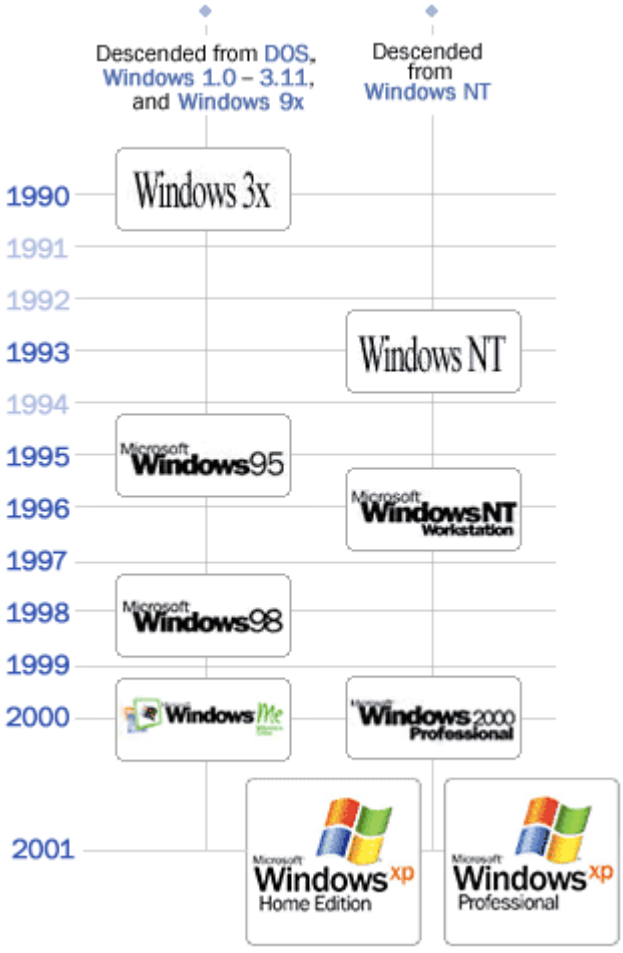


Figure 4-1: Microsoft Windows versions timeline.

Microsoft released the first version of Windows in the fall of 1985. Since then, Windows has been progressively updated and enhanced, most dramatically in Windows NT (1993) and Windows 95 (1995), when Windows moved from a 16-bit to a 32-bit architecture.

Today it is the most widely used operating system for desktop computers, with a market share of over 90 percent (96 percent in 2004 according to a Gartner Inc. research [20]). Hence, it is the operating system of most interest to develop commercial programs for.
An important object in the Windows environment is the window. Every object that the user can see and interact with is a window. They come in many different shapes, where the most obviously are the application windows and dialog boxes. However, various push buttons, list boxes, text-entry fields etc. that adorn the surface of dialog boxes and application programs are also a type of window. More specifically, these are called *child windows* or *control windows*, but are often abbreviated as just *controls*.
Windows operating systems are based on *messages*, which mean that they send messages to windows on different *events*. Although such events might occur due to reasons caused by the system itself, most typically they occur when the user gives input to the system. A window also uses messages to communicate with other windows. When a Windows program begins execution, Windows creates a *message queue* for the program. This message queue stores messages to all the windows a program might create. A Windows application includes a short chunk of code called the *message loop* to retrieve these messages from the queue and dispatch them to the appropriate *window procedure*, which processes messages to the window.
When Windows was first released, there was really only one way to write Windows applications, and that was by using the C language to access the Windows application programming interface (API). Such access is gained through the Windows *Software Development Kit* (*SDK*). SDK programming is the fastest, most flexible, but also the most difficult way to program Windows, as it is very closely connected to the operating system.
Over the years, many other languages have been adapted for doing Windows programming, including Visual Basic and C++. With the introduction of .NET, Microsoft currently offers three approaches to writing Windows applications using a C-based language:

Year Introduced	**Language**	**Interface**
1985	C	Windows application programming interface (API)
1992	C++	Microsoft Foundation Class (MFC) Library
2001	C# or C++	Windows Forms (part of the .NET Framework)

Table 4-1: How to write a Windows application using a C-based language (Microsoft-centric view).

Programmatically speaking, both the MFC and Windows Forms interfaces work by making calls to the Windows API. Architecturally, they can be said to sit on top of the API. These higher-level interfaces are intended to make Windows programming easier. Generally, it is possible to do specific tasks in MFC or Windows Forms with fewer statements than when using the API. While high-level interfaces such as MFC or Windows Forms often improve programmer's productivity, any interface that makes use of another interface is obviously less versatile than the underlying interface.

4.1.2 Linux

Linux is a freely (although copyrighted) distributable version of the Unix operating system, originally developed by Linus Torvalds, who began work on Linux in 1991 as a student at the University of Helsinki in Finland. Because Linux is free, and all the source code is available, anyone can modify the system to fit their own needs. Linux is, per se, only the kernel of the operating system, the part that controls hardware, manage files, separate processes, and so forth [21]. The word Linux is however also widely and correctly used to refer to an entire operating system built around the Linux kernel. There are several combinations of Linux with sets of utilities and applications to form a complete operating system. Each of these combinations is called a distribution of Linux, and there exists many different distributions from multiple vendors, highly customizable for specific tasks. Examples of some popular distributions are:

- Red Hat
- MandrakeSoft
- SuSE
- Knoppix

The standard graphical subsystem for UNIX and Linux, the X Window System, release 11, called X11 for short, has its own libraries for GUI development. They provide a low-level programming interface to X11, but tend to be hard to use. Old end-user applications and other toolkits of course make good use of them. Nowadays the Linux GUI scene is dominated by GTK+ [22] and Qt [23], since two popular, complete user environments – GNOME and KDE – are based on them. GTK+ is a multi-platform toolkit for creating graphical user interfaces, and thus offers a complete set of widgets. It has been designed from the ground up to support a range of languages, not only C/C++. Using GTK+ from languages such as Perl and Python (especially in combination with the Glade GUI builder [24]) provides an effective method of rapid application development. GTK+ is free software and part of the GNU Project [25]. Qt will be discussed in section 4.1.3.

4.1.3 Qt

Trolltech's Qt [23] is a comprehensive C++ development framework that includes an extensive array of features, capabilities and tools that enable development of high-performance, cross-platform rich-client and server-side applications. The current version of the Qt framework is Qt 4 [20, 26], consisting of [23]:

- **The Qt Class Library** is a growing library of over 400 C++ classes, which encapsulates all infrastructure needed for end-to-end application development. The Qt API includes a mature object model, a rich set of collection classes, and functionality for GUI programming, layout, database programming, networking, XML, internationalization, OpenGL integration and more.

- **Qt Designer** is a powerful GUI layout and forms builder, enabling rapid development of high-performance user interfaces with native look and feel across all supported platforms.
- **Qt Linguist** is a set of tools designed to smooth the internationalization workflow, increasing accuracy and greatly speeding the localization process.
- **Qt Assistant** is a fully customizable, redistributable help file/documentation browser that can be shipped with Qt-based applications, speeding the documentation process in software development.

It is the leading framework for native cross-platform application development, providing an API and tools that are consistent across all supported platforms. Using Qt, developer teams can create native applications for all supported operating systems – from all supported development platforms, covering:

- Qt/Windows (Microsoft Windows XP, 2000, NT4, Me/98)
- Qt/Mac (Mac OS X)
- Qt/X11 (Linux, Solaris, HP-UX, IRIX, AIX, many other Unix variants)

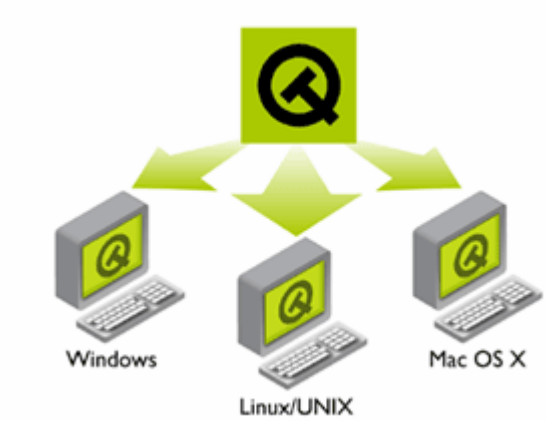


Figure 4-2: Qt supports cross-platform development.

Qt requires no virtual machines, emulation layers or runtime environments in order to be platform independent. Deployment in a different environment only requires the recompilation of the single source code on the target platform, providing executable native machine code. This allows Qt applications to run at native speed.
A license follows with Qt though. Through Trolltech's Dual Licensing Business Model [27], software developers may provide their products for two distinct uses – commercial and open source software development. Either, the developer has to purchase commercial licenses from Trolltech keeping the source code private, or by contributing to the Open Source community by placing the application under an Open Source License (e.g. the GPL). The last option secures all users the rights to obtain the application's full source code, modify it, and redistribute it. By employing dual licensing, Trolltech is able to staff a full-time dedicated development team while at the same time being an active member of the open source community, with all advantages this offers brought into the

development cycle of Trolltech products, and reaching commercial stability far more quickly.

4.1.4 The Java 2 platform

The Java 2 platform is a software-only platform that runs on top of other hardware-based platforms [28, 29]. It provides a host of off-the-shelf class libraries covering everything from classes for creating client applications to classes for creating server and Internet applications. Because of its large area of application, Java technology has grown to include a number of specialized platforms such as

- Java 2 Platform, Standard Edition (J2SE)
- Java 2 Platform, Enterprise Edition (J2EE)
- Java 2 Platform, Micro Edition (J2ME)
- Java Card technology

Each platform is based on a *Java Virtual Machine* (*Java VM*, or *JVM*) that has been ported to the target hardware environment. The ported Java VM provides the runtime environment for Java applications, and is referred to as the *Java Runtime Environment* (*JRE*). Java VM is defined as an abstract computing machine, in that it has an instruction set and manipulates various memory areas at runtime. It is responsible for converting the bytecode compiled source code into platform specific commands, and hence serves as an emulator for Java applications in the target system. Java VM is therefore the component of the Java technology responsible for its hardware and operating system independence. This is illustrated in Figure 4-3 below.

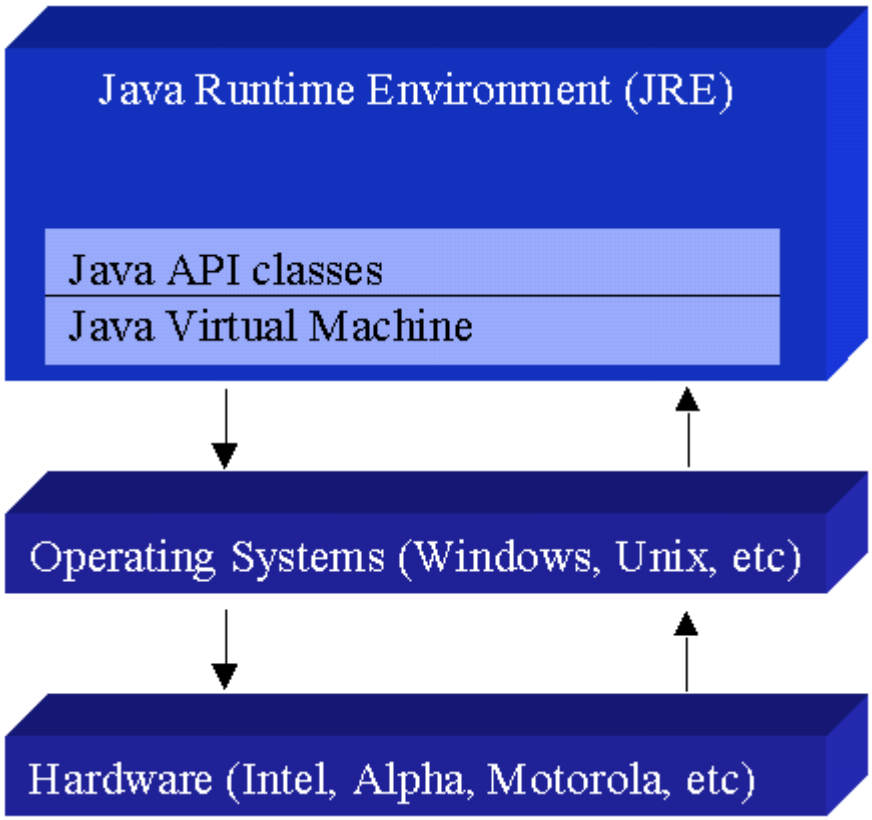


Figure 4-3: Java as a platform independent technology.

J2SE provides an environment for Core Java and Desktop Java application development, and is therefore the edition to employ for the work in this thesis. It has the compiler, tools, runtimes, and Java APIs that let one write, test, deploy and run applets and applications.

J2SE provides the *Java Foundation Classes* (*JFC/Swing*), a set of Java class libraries that support building GUI and graphics functionality for client applications [30]. As with the rest of the Java API, JFC/Swing is well designed with a good density of object-oriented design patterns put to practice, and therefore encourages to good object-oriented practices among programmers. Especially, it utilizes the MVC design pattern extensively, separating the data from the view.

An alternative to JFC/Swing is however the *Standard Widget Toolkit* (*SWT*) developed by IBM as a part of the Eclipse [31] platform. SWT is a graphics library and a widget toolkit integrated with the native window system (especially with Windows but Linux and Solaris are supported as well) [32], providing the native look and feel of the application. Despite the tight integration with the native target platform though, SWT is an operating system independent API and can be seen as a thin wrapper over the native code GUI of the host operating system. While SWT performs slightly better than JFC/Swing, it is also more complex as it requires explicit de-allocation of native resources from the programmer. For cross-platform development JFC/Swing resides as the conservative choice.

The Java 2 platform also benefits from the wide range of good IDEs available. Many of them are free and support more features than Microsoft's competitive IDE for the .NET

platform, Visual Studio .NET. However, most of them are also developed entirely in Java, and therefore has slightly slower performance than Visual Studio .NET.

4.1.5 The Microsoft .NET platform

> ".NET is the Microsoft Web services strategy to connect information, people, systems, and devices through software. Integrated across the Microsoft platform, .NET technology provides the ability to quickly build, deploy, manage, and use connected, security-enhanced solutions with Web services."
>
> *Microsoft*

Microsoft .NET (pronounced "dot net"), also referred to as the ".NET Initiative", consists of four core components:

- .NET Building Block Services or programmatic access to certain services as file storage, calendar and Passport .NET (a service verifying identity).
- .NET device software capable of running on new platforms.
- .NET user experience including functionality such as native interface, information agents, and SmartTags, a technology who automatically hyperlinks to information, related to words and sentences in user created documents.
- .NET infrastructure covering .NET Framework, Microsoft Visual Studio .NET, .NET Enterprise Servers and Microsoft Windows .NET.

However, the latter is the only part of current interest. At the heart of .NET is the *.NET Framework* [33]. This framework is a development and execution environment that allows different programming languages and libraries to work together seamlessly to create Windows-based application that are easier to build, manage, deploy and integrate with other networked systems. An overview of this framework is shown in Figure 4-4.

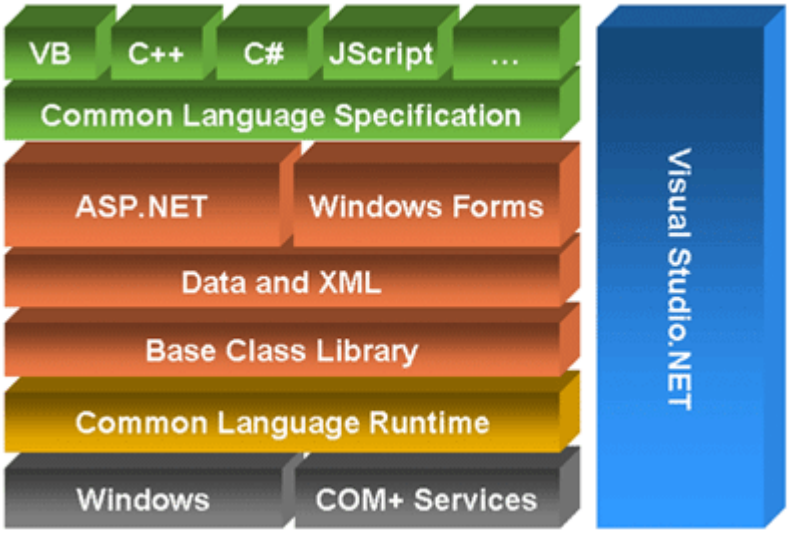


Figure 4-4: An overview of the .NET Framework.

The .NET Framework manages and executes applications, enforces security and provides many other programming capabilities. It defines two important entities; the runtime itself called *Common Language Runtime* (*CLR*) which will be discussed in details later, and an extensive collection of language-neutral, reusable classes called *Base Class Library* (*BCL*) or *Framework Class Library (FCL*). BCL gives programs access to the runtime environment and provides the fundamental building blocks for any .NET application, being it an ASP.NET application, a Windows Forms application, or a Web Service. It is organized into *namespaces* – logical groupings of related classes – each containing types and additional namespaces related to common functionality.
The portion of the .NET Framework of most interest for the present thesis is *Windows Forms*, which is the part covering Windows programming in .NET. Windows Forms and its application development model will be discussed in details later.
The details of the .NET Framework are found in the *Common Language Specification* (*CLS*), a set of rules that compilers need to comply with in order to make .NET applications capable of running under the CLR. Compilers targeting the .NET Framework convert source code to an intermediate language in an .exe file. At first runtime, the intermediate language is compiled into appropriate microprocessor machine code. Thus, the .NET Framework is potentially platform independent.
The CLS also describes a set of features that different languages have in common, including a subset of the *Common Type* System (*CTS*) that defines rules concerning data types, important for supporting cross-language collaboration. It has been submitted for standardization to ECMA [34] (the European Computer Manufacturers Association), allowing independent software vendors to create the .NET Framework for other platforms. The .NET Framework exists only for the Windows platform, but is being developed for other platforms as well, such as Microsoft's *Shared Source CLI* (*Common Language Infrastructure*) [35]. The Shared Source CLI is an archive of source code that provides a subset of the Microsoft .NET Framework for both Windows XP and the FreeBSD operating systems. Worth mentioning here is the Mono Project [36], which started in 2001 as an effort to implement the .NET Framework to UNIX. It is an open development initiative sponsored by Novell [37] to develop an open source, UNIX

version of the Microsoft .NET development platform, allowing developers to build Linux and cross-platform applications with improved developer productivity. Mono's .NET implementation is based on the ECMA standards for C# and the CLI, and includes both developer tools and the infrastructure needed to run .NET client and server applications. In addition, the Mono Project offers an IDE, debugging and a documentation browser, and is positioned to become the leading choice for development of Linux applications.
There are no licenses with the Microsoft .NET Framework; it is freely available at [38] as both a software development kit (SDK) and as a redistributable package. The .NET Framework SDK includes everything developers need to write, build, test and deploy .NET Framework applications – documentation, samples, and command-line tools and compilers. The redistributable package includes everything needed to run applications developed using the .NET Framework. This basically means that development and deployment of commercial applications for the .NET Framework can be done free of charge. Software developers can use a normal text editor or a special purpose code editor to write source code that is then compiled using the free command-line compiler. When deploying, the redistributable package can be included at no costs.
But, using plain text editors to program .NET applications can be cumbersome, especially for larger software systems. The normal procedure for .NET development is by using Visual Studio .NET, the latest IDE in the Visual Studio family. Like other IDEs for other technologies, Visual Studio .NET is a highly productive environment providing a range of tools that together bring a weld of benefits for the individual developer as well as teams of developers. Amongst the more important features is the GUI designer and the code editor with color-coded syntax, brace matching and IntelliSense – an auto completion facility for variable names, method names etc. Another usable feature is the documentation framework, which exposes the documentation of components in design time and eases the documentation process of large software systems.
Visual Studio .NET is licensed though, and together with operating system licenses on application and web servers as well as developing machines constitutes the costs for software vendors targeting the Microsoft platform. Visual Studio .NET is available in a free light-weighted student edition however, so the licensing issues are of no concern for this thesis.

4.2 Programming languages

The choice of programming language is closely related to the choice of development platform. There is however three languages that distinguish themselves from the abundance of available programming languages based on the introduced development platforms. They are all object-oriented programming languages, and they are somewhat related to each other as they all belong to the C programming language family. Also, each of the programming languages can be used in conjunction with an *Integrated Development Environment* (*IDE*). An IDE is a programming environment integrated in a software application that provides a GUI builder, a text or code editor, a compiler and/or interpreter and a debugger. This provides a highly usable, efficient way to write programs for the modern enterprise computing environment, including desktop applications, Internet applications, components, and so on. With the IDE, a programmer can create,

run, test and debug programs conveniently, thereby reducing the time needed to produce a working program to a fraction of the time needed without using the IDE. The process of creating an application rapidly using an IDE is referred to as *Rapid Application Development* (*RAD*). The three languages will be further discussed in their order of appearance.

4.2.1 C++

C++ is the oldest programming language of current interest for the thesis, and somehow serves as a base for the other two; Java and C#. Driven by the need of a fast multi-paradigm object-oriented programming language, it began as an expanded version of one of the most liked and widely used professional programming languages at the time; C. C was invented by Dennis Ritchie in the 1970's at Bell Labs, and grew out of the structured programming revolution of the 1960's, being the first structured language to combine power, elegance and expressiveness successfully.
C was initially developed as a programming language for UNIX with the intention of facilitating the system development [39], and so it can be said to be a middle-level language [40], as it manages to combine the advantages of high-level languages with the functionalism of assembly languages. With the first implementations of UNIX in C, its use quickly spread throughout the programming community and its popularity increased in time with the popularity of UNIX. C has many weaknesses though, but it is general-purpose, fast and extremely portable, and was chosen as a base for the C++ language because of its named features.
Bjarne Stroustrup first invented the C++ extensions in 1979 at Bell Laboratories in Murray Hill, New Jersey. This new language was initially called "C with Classes", but in 1983 the name was changed to C++ [40, 41]. Inspired by another object-oriented language called Simula67 – the first object-oriented programming language ever developed in the 1960s by Kristen Nygaard and Ole-Johan Dahl at the Norwegian Computing Center [42] – most additions made by Stroustrup to C support object-oriented programming, and therefore made C++ able to respond to the increasing complexity of software in ways structured methodologies and languages like C could not.
Since the invention of C++, it has undergone a few major revisions, with each adding to and altering the language. The first revision was in 1985 and the second in 1990 [40]. The third occurred during the standardization of C++, which took far longer time than first presumed, mainly due to the advent of the Standard Template Library (STL), created by Alexander Stepanov [40]. In any case, an ANSI/ISO standard for the C++ language finally became a reality in 1998, and its specification is commonly referred to as *Standard* C++.
During its lifetime, C++ has lived a great expansion in its use and today is in many scenarios the preferred language for developing professional applications on all platforms because of its raw power, both regarding to its expressiveness and to its performance. It is also compatible with many other technologies.

4.2.2 The Java language

Java is a general-purpose object-oriented language developed by James Gosling and colleagues at Sun Microsystems in the early 1990s. The initial release came in January 1996, but the language has continued to evolve ever since. Unlike conventional languages, generally designed to be compiled to native code, Java is compiled to a bytecode which in turn is run by a *Java Virtual Machine*. Thus, the Java language is intimately related to its runtime environment, which also provides the libraries used in Java, and hence the actual functionality of the language.
Java borrows much syntax from C++. However, it omits many of the features that make C++ complex and unsafe, and has a much simpler object model. For instance, Java is fully object-oriented as everything is packed in classes and objects, and it strictly enforces the object-oriented programming paradigm. Unlike C++, Java also provides automatic garbage collection, sparing the burden on the programmer of having to perform manual memory management. Overall, this makes Java a slightly simpler language and thus easier to learn and understand compared to C++.

4.2.3 C#

Released to the public in June 2000 [43], C# [44, 45] (pronounced “C-Sharp”) represents the next step in the ongoing evolution of programming languages. It was developed at Microsoft by a team led by Anders Hejlsberg and Scott Wiltamuth, designed specifically for the Microsoft .NET platform as a language that would enable programmers to migrate to .NET [44]. C# is directly descended from two of the world’s most successful computer languages: C and C++. From C, it derives its syntax, many of its keywords, and operators, while it builds upon and improves the object model defined by C++. C# is also closely related to another very successful language: Java. Sharing a common ancestry, but differing in many important ways, C# and Java are more like cousins. This relationship is sketched in Figure 4-5 below.

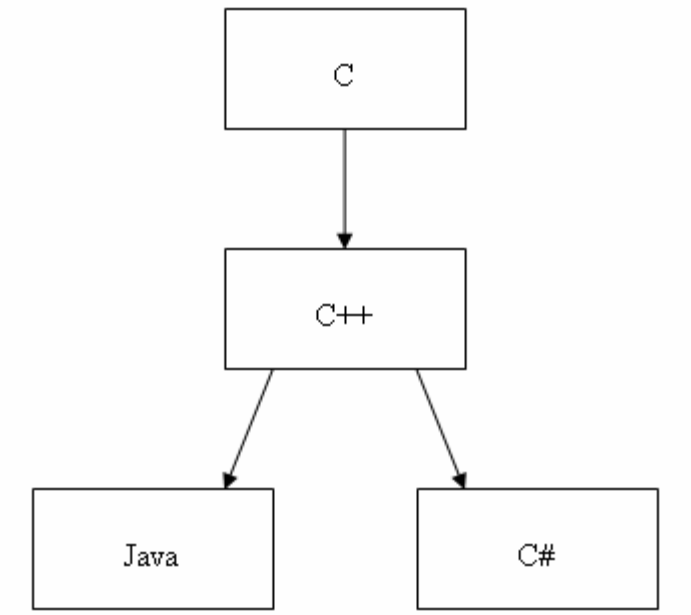


Figure 4-5: Simplified C# family tree.

Naturally, C# provides an improvement of the programming languages it evolves from, and in that regard, it adapts the best features of each and adds at the same time new features of its own. It is a modern and simple language, removing some of the complexities and pitfalls of related languages, while providing the features that are expected in a modern language. C# is a fully object-oriented, component-oriented, type safe, visual programming language in which programs are typically created using an IDE, and in that regard, it daintily combines the high productivity of RAD languages and the raw power of C++.
C# also directly supports events and is considered to be an event-driven language. This is a highly interesting feature and of great value as it provides a very clean, structured, and perspicuous way to program event-based applications.

4.3 Data persistence method

A notoriously property of the software lies in its ability to produce and alter documents. To accomplish this, an important feature is that of being able to store the documents to be regained for later use. There are basically two ways to persist the state of such documents: either with the use of a database, or by directly use of the file system. These approaches will be explored next.

4.3.1 Database

A database is an organized collection of data [46], adjusted for easily and quickly access to its contents. When databases are computerized, the program used to manage and query the databases is known as a database management system (DBMS). The DBMS approach

provides an interface between physical data and the applications, as demonstrated in the figure below.

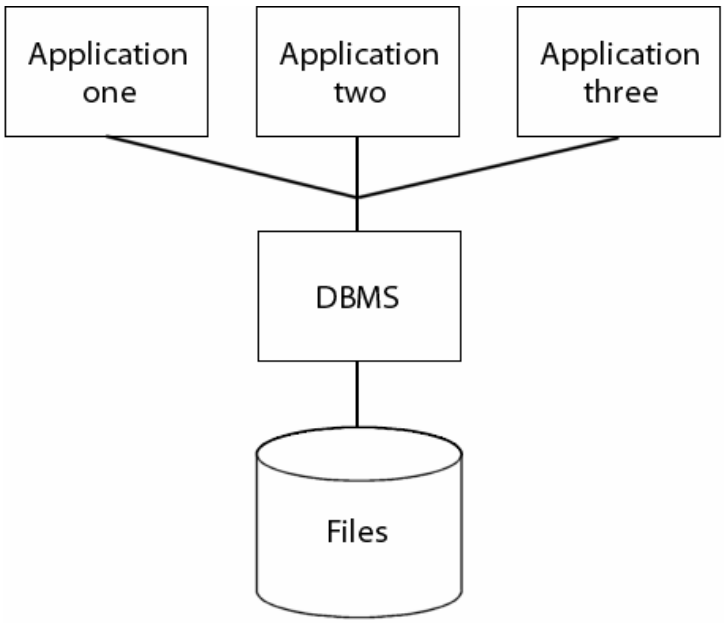


Figure 4-6: The DBMS approach.

Typically for a database, there is a structural description of how the data is to be held; information needed to be stored, and relations between data. This description is known as a *schema*. The database schema can be organized, or be modeled, in multiple ways. The model in most common use today is the relational model, and accordingly most databases available are so-called relational databases. As discussed in section 1.3.2.1 the ER model maps well to relational databases, and has become the de facto for modeling complex information systems, the mainstream application area for databases.
In such systems, data needs to be shared among multiple users, often simultaneously. It is of absolute most importance that data is kept consistently to ensure the integrity of the systems. The main advantages of databases over file systems are:

- Data independence
- Efficient data access
- Data integrity and security
- Removal of redundancy
- Consistency
- Data administration
- Economy of scale
- Improved accessibility
- Improved maintenance
- Increased concurrency

- Increased productivity

However, given all these advantages over file systems, there are still times where databases do not fit the storage requirements of an application. Typically this yields for so-called *document-centric* applications; programs with main focus on producing documents of some sort, like a text document or an image document. In such cases the use of a database could be considered as overkill as databases raises the complexity level and puts restrictions on the hardware.

4.3.2 File system

In terms of computers, the main tasks of the *file system* are effectively permanent storage of and organization of *computer files*. Modern file systems are also responsible for security issues such as access rights for different *users*, and offer facilities to create, move, and delete both files and *directories*. Generally, a file is a collection of diverse information, while a directory is a special type of file that contains a file allocation table of some sort, meaning it associates files in a file. Directories in a file system provides a way to gather related files in collections, and makes it possible to store files in a hierarchical matter, where directories can have subdirectories. This is an essential feature in file systems containing many thousands, or hundred thousands of files. Users of a file system may be either concrete human beings using the computer, or it may also be applications running on the computer. Their *path* uniquely identifies files located in the file system. Most operating systems provide a file system, as a file system is an integral part of any modern operating system. Commonly, these file systems are disk based, directly or indirectly connected to the computer. Although these file systems differs from one another on some territories, they have similar features.
Programs targeting the creation of specific documents, often referred to as document-centric applications, generally adopt direct use of the file system. There are basically two ways to build up a file: either by using plain text readable by a human being, or in a binary format. Either way, the information must be stored in a unified, sequential matter, with a united way of separating data, so that the computer program knows how to manage the data. This result in some sort of file structure, facilitating the application's input from file.

4.3.3 XML

Another way to structure a file is with the *Extensible Markup Language* (XML). XML is a W3C [47] recommended general-purpose *markup language*, as it is a set of rules for building special-purpose markup languages [48, 49]. As a simplified subset of *Standard Generalized Markup Language* (SGML), XML's primary purpose is to facilitate the sharing of data across different systems, serving as a format for application-to-application data exchange. It has quickly gained wide popularity for documents containing structured information, and has become an important part in many modern computer technologies such as Java and .NET.

Markup is information added to a document that enhances its meaning in certain ways, especially important to electronic documents. A *markup language* is a set of symbols that can be placed in the text of a document to demarcate and label the parts of that document, providing a mechanism to identify structures in a document. A simple XML document is demonstrated in the code sample of Table 4-2.

```
1.  <?xml version="1.0" encoding="utf-8" ?>
2.  <VisualSignature
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-  instance"
4.     xmlns="Sketcher.Xml">
5.     <Name>Markers Signature</Name>
6.     <Description>A signature file.</Description>
7.     ...
8.  </VisualSignature>
```

Table 4-2: An example of an XML structure.

Because XML is a specification for creating markup languages, it doesn't have any predefined tags. Rather, such tags are user-defined in markup languages that are based on XML. An XML document being *well-formed* means that it satisfies the syntax rules of XML documents, and that any XML processor can read it.
Besides being well-formed, it is often desirable to be able to validate XML documents against special rules. In order to control if a particular document (referred to as a *document instance*) matches a specific language specification, XML provides a way to describe a language in no uncertain terms. This is called *document modeling*, because it involves creating a specification that lays out the rules for how a document can look, and is in effect the model documents are compared against. Document modeling can be done with a *document type definition* (*DTD*) [49], or with *XML Schemas* [49, 50]. DTD is a set of rules or *declarations* that specify which tags can be used and what they can contain. Despite the many advantages that DTD brings along, it suffers from limitations such as:

- DTD syntax differs from the XML syntax
- Bad support for XML namespaces
- Bad support for data types
- Limitations of content-model descriptions

Despite its limitations though, DTD is a fundamental part of the XML recommendation [51], and has been the common approach to model documents. But eventually, it will have to give way for the new document modeling standard known as XML Schema, or XML Schema Definition (XSD).
In general, a *schema* is an abstract representation of an object's characteristics and relationship to other objects. An XML schema represents the interrelationship between the attributes and elements of an XML object. It uses XML fragments called *templates* to demonstrate how a document should look, that is, it defines each structural elements of the document in a correct sequence.
One of the greatest strengths of XML Schemas is the support for data types. Another advantage to a XML Schema over earlier document modeling approaches is that it is written in XML itself, thus being extensible and more direct; it doesn't require

intermediary processing by a parser. Other benefits include self-documentation, automatic schema creation, and the ability to be queried through XML Transformations (XSLT). Thus, XML Schemas provide a nice approach to validate XML documents.

4.4 Technologies of choice

As a starting point, no restrictions were put on which technologies to use. The outset was a developing platform supporting a GUI, and tools that would provide the necessary functionality in order to draw elements on screen and to perform basic storage operations on the produced documents. The intention with the thesis was the starting of a project of sufficient size and complexity, where the final product would be a complete drawing tool application for a graphical specification language with implementations of all possible features of this language. Due to the evidently time limitations, the exact boundaries for the development work were difficult to predict. The main focus was to build an extensible framework of a drawing tool, as there were no possibilities to carry through the whole project within this period. Choosing right technologies would certainly facilitate the development process, conduct in that as much as possible would be done during the thesis. But what were the right technologies?
The choice was finally based upon four elements:

- Satisfaction of functional and non-functional requirements.
- Ease-of-use.
- Acquaintance.
- Personal interest.

Most considerable is the satisfaction of the requirements set by the kind of program to be developed, where support for GUI applications is probably the most prominent feature. Another property of just as much importance however, is support for the necessary drawing functionalities so that the creation of a graphical program could be realized. From a simple point of view, the main focus of the program to be developed is the representations of user-created sketches like it had been created using pen and paper. And so especially important is the support for a 2-D graphic system with the ability to perform common 2-D operations such as translation, rotation, and scaling, in order to express the drawings in a correct manner (i.e. proper position and size of elements). Although in different ways, these needs are covered by all developing platforms introduced in earlier sections.
Important were also the technologies' ease-of-use, and their support for and qualities in rapid application development (RAD). However, there is a balance here. High-level features of the development tools should not affect the performance of the application too much as it is an event-driven GUI program in which the user interacts visually with the program. Anyway, using high-productive technology would speed up the development process increasing the amount of work to be done during the thesis.
Another element that naturally played a vigorous part was acquaintance with the technologies. As not much time would be available to study unknown technology, the

technologies to be used should at least to some extent, preferably be familiar. In this way, the work of solving the actual tasks of the thesis could begin in a much earlier phase, establishing a good basis for the subsequent work rather than having to throw away valuable time on needless boloney. Enough time had to be spent on learning more theoretical issues anyway, such as a basic understanding of the generalized sketches formalism itself and its mathematical foundation.
Taken these elements into consideration, there were only three realistic programming environments to choose between:

- Windows programming with Windows Forms and C#
- Java programming
- Qt programming with C++

Using the Qt framework would allow cross-platform development for the major platforms of desktop computers and probably yield in best performance of the program because of the optimization gains in C++ compilers compared to C# and Java JIT compilers. However, of the current programming languages, C++ is the most cumbersome. Also, licensing issues and unfamiliarity with the Qt framework led to dismissal of this technology.
The Java platform was a strong candidate because of its platform independency and extensively utilization of design patterns throughout its API. It is commonly used for educational ends, and since the Java language has a very similar syntax to other C-based languages, it is quite adoptable.
Even though there is much talk about platform independency, aiming at developing a true Windows application for the Microsoft Windows operating system can be justified because of its sovereign popularity. As a matter of fact, the existing solution is an MFC application. Windows programming towards the Windows API is capable of taking advantage of the native operating system and therefore provides the most powerful way to develop GUI applications for Windows. There are several ways to do this, but the one of current interest is with C# and Windows Forms, third generation Windows programming after MFC and Win32 SDK programming. Although the combination of C# and Windows Forms technically has a little slower performance than former Windows programming techniques due to the efficiency gains of C++ compilers, it is superior when it comes to productivity and easiness that it levels the small performance hit by far. Among others it can show off with automatic memory management, an extensive library of reusable classes, as well as a completely renovated development framework, making development of complex applications faster, easier and more secure.
Of the proposed technologies, Windows Forms and C# were evaluated to be a natural choice for GUI development, as Microsoft Windows was most likely to be the development platform and this technology provides the most suitable method for Windows programming. Also, painted with personal interests, the thesis was a worthy opportunity to establish acquaintance with the .NET Framework and C# in particular. For data persistence, the choice was easy. XML and XML Schemas would provide a clean and standardized way of persisting and validating data. Also, it will take part in facilitating communication between applications in future use.

4.5 A closer look at the .NET Framework

4.5.1 Common Language Runtime (CLR)

CLR is the core of the .NET infrastructure. It is a runtime environment where applications written in different languages can work together. This is known as *language interoperability*. Table 4-3 shows a list of supported programming languages for the .NET Framework.

APL	Fortran	Pascal
C++	Haskell	Perl
C#	Java Language	Python
COBOL	Microsoft JScript®	RPG
Component Pascal	Mercury	Scheme
Curriculum	Mondrian	SmallTalk
Eiffel	Oberon	Standard ML
Forth	Oz	Microsoft Visual Basic®

Table 4-3: Supported programming languages in the .NET Framework.

Code that is developed with a language compiler targeting the CLR is called *managed code*. Simply put, managed code is a code run under the management of CLR. It benefits from features such as cross-language integration, cross-language exception handling, enhanced security, and versioning and deployment support, a simplified model for component interaction, and debugging and profiling services. In a managed code environment it is necessary with a number of rules to ensure that the applications behave in a globally, unified way, independent of what language they are written in. The unified behavior of .NET applications is the very essence in .NET.

To enable the CLR to provide services to managed code, language compilers must emit metadata that describes the types, members, and references in the code. Metadata is stored with the code; every loadable CLR portable executable (PE) file contains metadata. The CLR uses metadata to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set run time context boundaries, so that the application is run in a most effective matter. The way this metadata is fetched is by *reflection*. The BCL provides an entire set of reflection methods making every .NET application, not only the CLR, capable of fetching the metadata of other .NET applications.

The CLR also automatically handles object layout and manages references to objects, releasing them when they are no longer being used. Objects whose lifetimes are managed in this way are called managed data. Garbage collection eliminates memory leaks as well as some other common programming errors.

.NET applications are not restricted to managed code only however, but could also take advantage of communicating with components that is not developed in a managed environment. These interoperability features are called *Platform Invoke*, or *P/Invoke*.
Such unmanaged code does not benefit from the named features of a managed environment, but provides a powerful way to communicate with the Windows API or any other COM component that would seem suitable for some purpose.
Programming languages targeting the .NET Framework are compiled into an intermediate language called *Common Intermediate Language* (*CIL*). CIL is the lowest-level human-readable programming language in the CLI and in the .NET Framework. With the beta releases of the .NET languages, CIL was originally known as *Microsoft Intermediate Language*, or *MSIL*. Due to standardization of C#, the main language in .NET, and the Common Language Infrastructure, MSIL is now officially known as CIL. Still, it is often being referred to as MSIL.
CIL resembles an object-oriented assembly language and is entirely stack-based. It is assembled into bytecode, providing platform independence. At runtime it is then compiled in a manner known as *just-in-time compilation* (*JIT*) into native machine code. This is a technique for improving the performance of bytecode-compiled programming systems, and is described in the Figure 4-7 below.

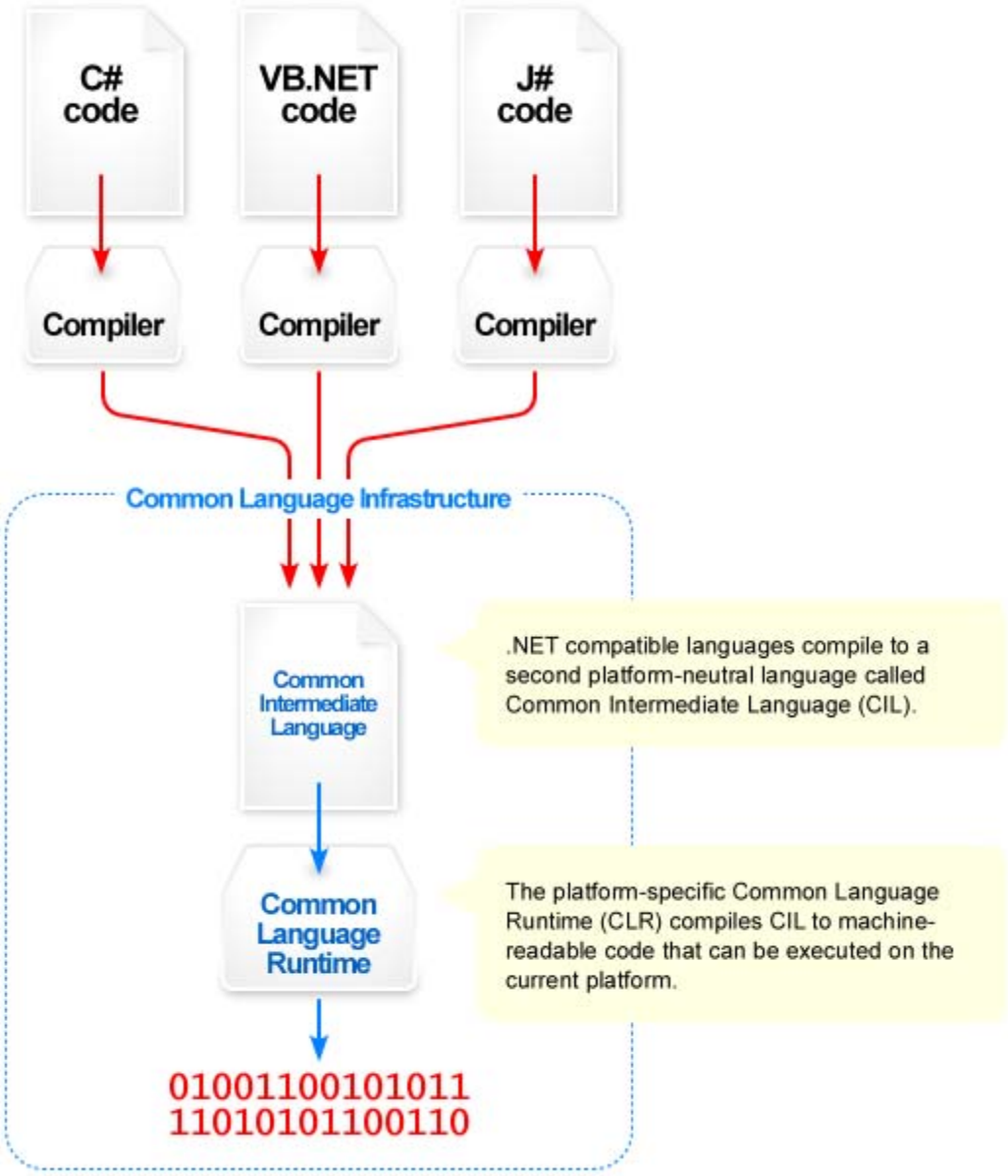


Figure 4-7: Common Language Infrastructure.

In a JIT environment, bytecode compilation is the first step, reducing source code to a portable and optimal intermediate representation. The bytecode is then deployed onto the target system and executed, making the runtime environment's compiler translating it into executable native machine code. The goal of JIT is to combine many of the advantages of native and bytecode compilation:

- Much of the "heavy lifting" of parsing the original source code and performing basic optimization is handled at compile time, well prior to deployment.
- Compilation from bytecode to machine code is much faster than from source code.

- The deployed bytecode is portable, unlike machine code for any given architecture.
- Native machine code runs much faster than bytecode interpretation, resulting in significant gains in overall performance of the application.

In .NET there are three different JITs available;

- Install-time code generation
- Standard JIT
- Economy JIT

Install-time code generation compiles all code at once. This is typically used when developing commercial installation applications, making sure of delivering a complete optimized version "out-of-the-box". The standard JIT only compiles the code sequences used. With this option, compilation only happens whenever a method is called for the first time. The last JIT, Economy JIT, is developed especially for systems with limited resources, such as PDA's with small memory. The biggest difference between this JIT and the regular JIT is a technique called *code pitching*. Code pitching makes it possible for Economy JIT to discard generated or compiled code, in case the system runs low on memory. A benefit of this is that memory is regained. On the other hand, recompilation of pitched code runs at the cost of performance.

4.5.2 Windows Forms

> "Windows Forms is nothing less than a modern-day programming model for GUI applications."
>
> Jeff Prosise, MSDN Magazine February 2001

Windows Forms is the portion of the .NET Framework responsible for Windows client UI application development [52], and is the only part of the .NET Framework that is not potentially platform independent as exceedingly utilizes the native Windows API, as it is architecturally placed on top of it.

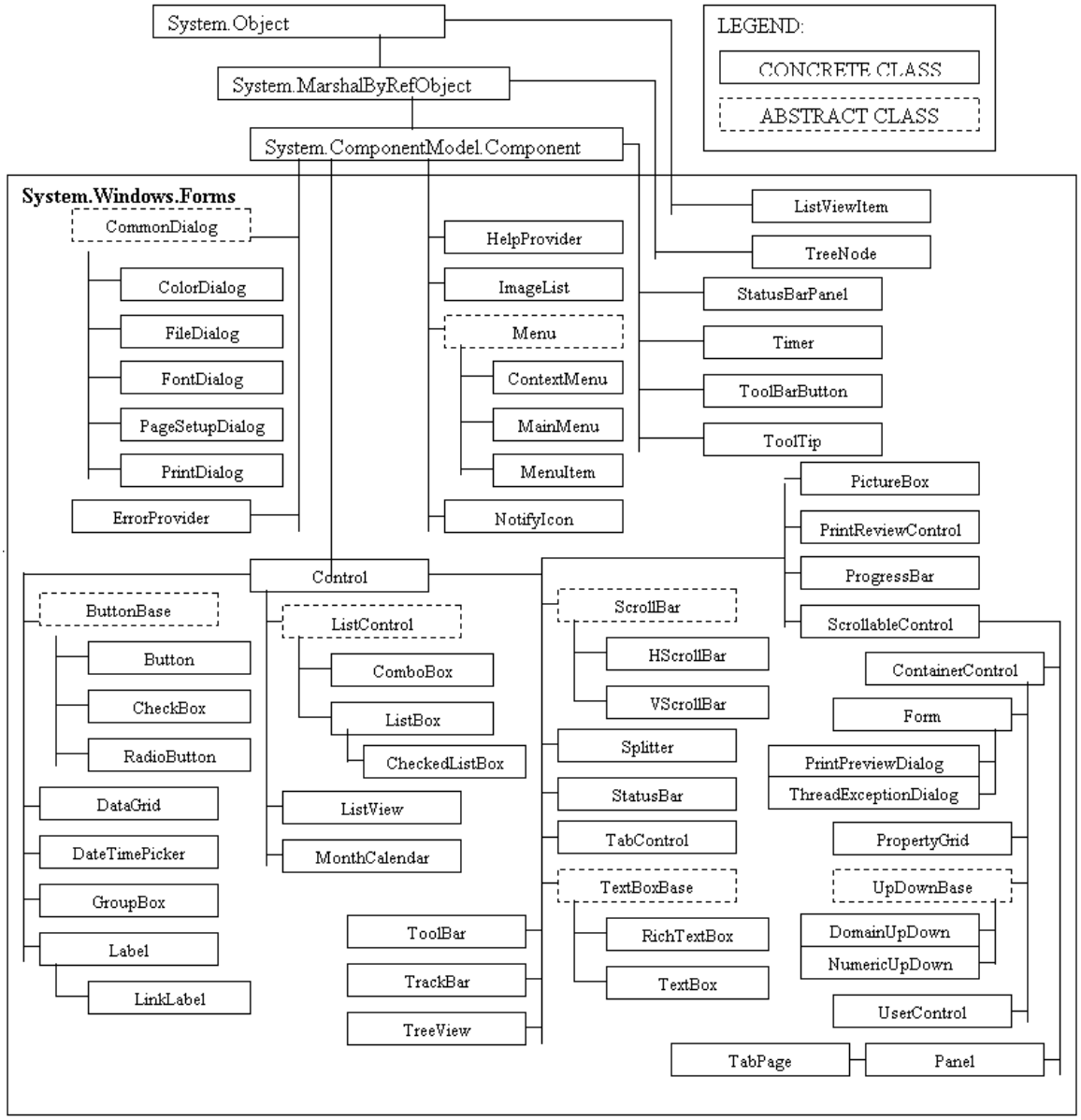


Figure 4-8: Windows Forms hierarchy.

As Figure 4-8 shows, Windows Forms is a complex and extendable but well-formed hierarchy of predefined ready-to-use classes, providing everything needed to develop rich, full-featured Windows applications. It improves [53] on traditional Windows programming models based on the Win32 API and MFC with a cleaner, more robust and more consistent model run in a managed environment of the .NET CLR. It homogenizes the programming model and eliminates many of the bugs, quirks, and inconsistencies that plague the Windows API, providing a more unified model with improved security, and a better way to write Windows GUI applications. Using the underlying Windows API and the GDI+, Microsoft's next generation 2-D graphics system, Windows Forms offers the most powerful way to draw in the Windows environment.

Windows Forms is based on a *broadcasted event-driven* model, which basically means that Windows Forms applications execute code in response to *events*. An event is a

message sent by an object to signal the occurrence of an action that it is possible to respond to, or "handle", in code, in so-called *event handlers*. Events can be generated by a user action, such as clicking the mouse or pressing a key; by some program logic in code; or by the operating system. The object that raises the event is called the *event sender*, or *publisher*. The object that captures the event and responds to it is called the *event receiver*, or *subscriber*. The event functionality is provided by three interrelated elements: a class that provides event data, an event delegate, and the class that raises the event.

The Windows Forms application programming model is primarily comprised of *forms*, *controls*, and their events. Controls in Windows Forms are every interactive control window that adorns the surface of an application, such as buttons, list boxes and toolbars, while forms are a special type of controls – top-level controls – that represents any window displayed in an application including standard and modal windows as well container windows for other windows, often referred to as Multiple Document Interface (MDI) Forms. Each form and control exposes a predefined set of events that it is possible to program against. These include the Paint event which causes a control to be drawn, events related to displaying a window, such as the Resize and Layout events, and low-level mouse and keyboard events. Some low-level events are synthesized by the Control class into semantic events such as Click and DoubleClick. If one of these events occurs and there is code in the associated event handler, that code is invoked.

Events in the .NET Framework are based on the *delegate* model which uses delegates to bind events to the methods used to handle them. This model is an example of the publisher-subscriber design pattern, which keeps the state of cooperating components synchronized by decoupling the publisher and subscriber of information. In such event communication, the event sender class does not know which object or method will receive, or handle, the events it raises. What is needed is an intermediary, or pointer-like mechanism, between the source and the receiver. The .NET Framework defines the special type delegate that provides the functionality of a function pointer. A delegate is a class that can hold a reference to a method. Unlike other classes, a delegate class has a signature, and it can hold references only to methods that match its signature. A delegate is thus equivalent to a type-safe function pointer or a callback. While delegates have other uses, the discussion here focuses on the event handling functionality of delegates.

A delegate declaration is sufficient to define a delegate class. The declaration supplies the signature of the delegate, and the common language runtime provides the implementation. By convention, event delegates in the .NET Framework have two parameters, the source that raised the event and the data for the event. Moreover, event delegates are multicast, which means that they can hold references to more than one event handling method, and so a delegate acts as an event dispatcher for the class that raises the event by maintaining an invocation list of registered event handlers for the event. This allows for flexibility and fine-grain control in event handling; when an event is recorded by the application, the control raises the event by invoking the delegate for that event. The delegate then calls each bound method in the invocation list in turn, which provides a one-to-many notification. Delegates also enable multiple events to be bound to the same method, allowing a many-to-one notification.

4.5.3 GDI+ support

Microsoft Windows operating systems consists of three core components, or subsystems. The subsystem responsible for displaying graphics on video displays and printers is known as the *Graphics Device Interface* (*GDI*). It is a standard for representing graphical objects and transmitting them to output devices such as monitors and printers, and perhaps the most significant capability of GDI over more direct methods for accessing the hardware is its scaling capabilities and abstraction of target devices. Windows programs are able to run without problems on any graphics output device that Windows support, and in that regard, GDI supports device independent graphics by providing facilities to insulate programs from particular characteristics of different output devices.
With the introduction of Windows XP, GDI was deprecated in favor of its successor, the C++ based *GDI+* subsystem. GDI+ is a next-generation 2-D graphics environment, adding advanced features such as alpha blending, gradient shading, more complex path management, and intrinsic support for modern graphics-file formats like JPEG and PNG.
.NET Framework and Windows Forms take full advantage of the GDI+ subsystem, and provides an API against it which is found in the *System.Drawing* namespace. Especially important for the work of the present thesis is the support for various 2-D transformations and the capabilities of working with paths. Also, to be able to perform any drawing on the screen display (and printers), Windows Forms provides access to a *Graphics* object, most commonly as reference through the arguments of paint events but it can be obtained in a few other ways as well. In any case, the received Graphics object is connected to either a control or an image. The special property of the Graphics class that makes it so important in the drawing aspect of GUI applications, is that it encapsulates a GDI+ drawing surface, that is, it defines the area in which it is possible to perform the drawing. This area is the *client area* of any control or the image itself to which the Graphics object is connected to. Moreover, when dealing with painting controls, it also exposes information about which areas of the control that has been *invalidated*, and therefore needs to be repainted, as given by the *clip area*. Also, another important property of the Graphics object is that all drawing is done only towards this single object. Hence, it is the object that is used to create any graphical images, irrespective of whether it should be output to the screen or the printer, and thus provides the abstraction over hardware devices.

4.5.4 XML support

The .NET Framework provides a large library of XML APIs built on industry standards such as DOM, XPath, XSD, and XSLT, available through the following self-explaining namespaces:

- System.Xml
- System.Xml.Schema
- System.Xml.Serialization
- System.Xml.XPath

- System.Xml.Xsl

These namespaces contains everything needed to deal with XML structured hierarchies at runtime, either represented in-memory or by files. For the data persistence work in the thesis, System.Xml.Schema and System.Xml.Serialization are of special interest. The former provides the functionality for parsing and validating XML documents against XML Schema Definition files, while the latter offers the necessary support for serialization of objects to XML structured files, and for de-serialization of XML files into program logic.

4.5.5 Assemblies

An *assembly* [44] in .NET is a file, or files, containing all deployment and version information for a program. Assemblies define scopes, and are fundamental to the .NET environment in that they are the mechanisms that support safe component interaction, language interoperability, and versioning. An assembly is composed of four parts. The first is the assembly *manifest*. The manifest contains information about the assembly itself, including name, version number, type mapping information, and cultural settings. The second part is *type metadata*, which is information about the data types used by the program. Type metadata among others aids in cross-language collaboration. The third part of an assembly is the *program code*, stored in CIL format, and the fourth constituent of an assembly is the *resources* used by the program.

4.5.6 .NET Framework 1.1 vs. .NET Framework 2.0

At the beginning of this thesis, the latest stable version of the .NET Framework was version 1.1, released in April 2003 and an integral part of the second release of Visual Studio .NET, released as Visual Studio .NET 2003. There was announced however, the release of a new major forthcoming version of the framework[3], that of version 2.0 included as part of Visual Studio 2005 and Microsoft SQL Server 2005. This naturally caused some attention as regards to the selection of tools, especially since beta releases were already available from the Internet at that time. The uncertainty of the final release and the fact that the beta releases was to a certain extent unstable, caused the ideas about employing version 2.0 for the programming work of the thesis to be temporarily postponed. Also, as version 1.1 of the framework had been around for quite some time, the development community for this version was naturally larger. This way, the programming work could start in a much safer and well established environment, with the contingency of future adaptation to version 2.0.
Due to severe time limitations this never became a reality, even though not too much work needs to be done for the conversion. This might be a request in the future however,

[3] For curious readers, the official release date of .NET Framework 2.0 along with Visual Studio 2005, Microsoft SQL Server 2005 and BizTalk 2006, fell on November 7, 2005.

because there are no questions about it; .NET Framework 2.0 is a new major release, and naturally brings along with it many improvements from the older ones.

5 SOLUTION

As a basis, the solution takes the same approach as the existing solution, that is, separating the concept of signatures from the concept of sketches in a clearly manner. However, instead of creating a standard MDI application, a more modern graphical user interface is preferred, bringing towards the ideas of a *Tabbed Document Interface* (*TDI*) for sketches together with a set of docking signatures. Besides providing a nicer looking application, it also results in a more intuitive graphical interface for the generalized sketches framework, and the outcome is shown in Figure 5-1.

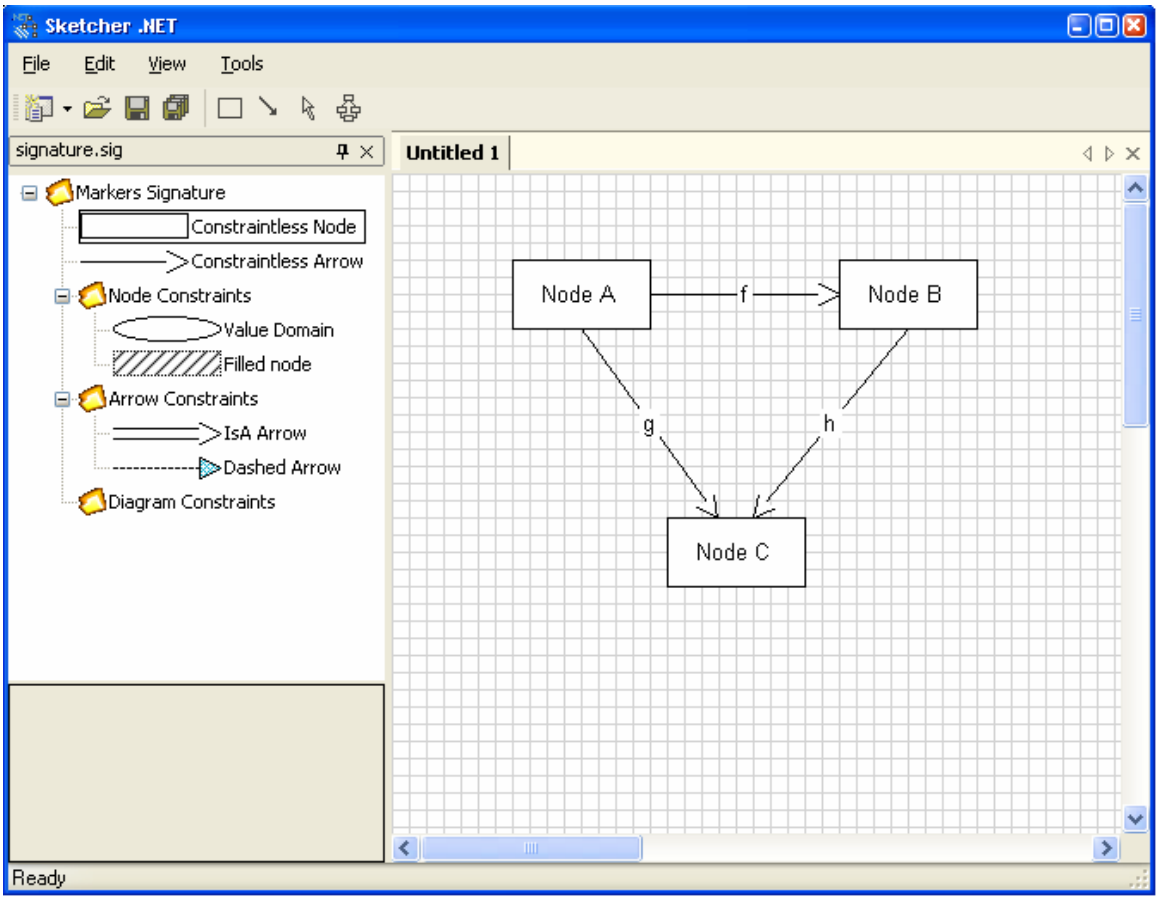


Figure 5-1: Sketcher .NET – A Tabbed Document Interface (TDI) application.

In section 3.1 it was stated that the task was a four-way separated workmanship, constituted by the following parts:

- Obtaining the internal structure of generalized sketches.
- Implementing the machinery of a drawing tool application.
- Persistence of data.
- Implementing functionality specifically adjusted for generalized sketches.

The first part is of a more theoretical character, while the three latter are more practical fitted. The work of the thesis somehow reflects these in a systematic manner, and the solution implies a more thoroughly discussion of each subject in their order of appearance.

5.1 Implementing the core of generalized sketches

The initial problem was to obtain the internal structure of the generalized sketches formalism. As most papers on generalized sketches focuses on more mathematical subjects and its possibilities, not much detailed information is available about the internal structure except for a brief outline. By studying the Sketcher 95 program, it became possible to pull out the missing pieces of information regarding the core of generalized sketches.
Briefly, a *sketch* specification is a directed graph in which some parts (diagrams) are marked with *predicate labels* taken from a predefined *signature*. This was carefully discussed in section 2.4, and therefore no further explanations are given here. What is important for the implementation is that these predicate labels can be logically separated into three types, which are:

- Predicates on nodes, to be interpreted as node constraints.
- Predicates on arrows, to be interpreted as arrow constraints.
- Predicates on diagrams, to be interpreted as diagram constraints.

These constraints have similar features including a name, a description, but also more important, a *marker*. Markers are the elements who denote predicate declarations on nodes, arrows, and diagrams (collections of nodes and arrows) appearing in a sketch specification, and are in that regard pure visual constructs but also of most importance as they play an essential part in building comprehensible sketch specifications.
On the other hand, a sketch is a graph and thus engaged with nodes and arrows, but it is also a special type of graph, introducing the notion of marked diagrams. Ultimately, this leads to the following class diagram (rough description) as shown in Figure 5-2 below.

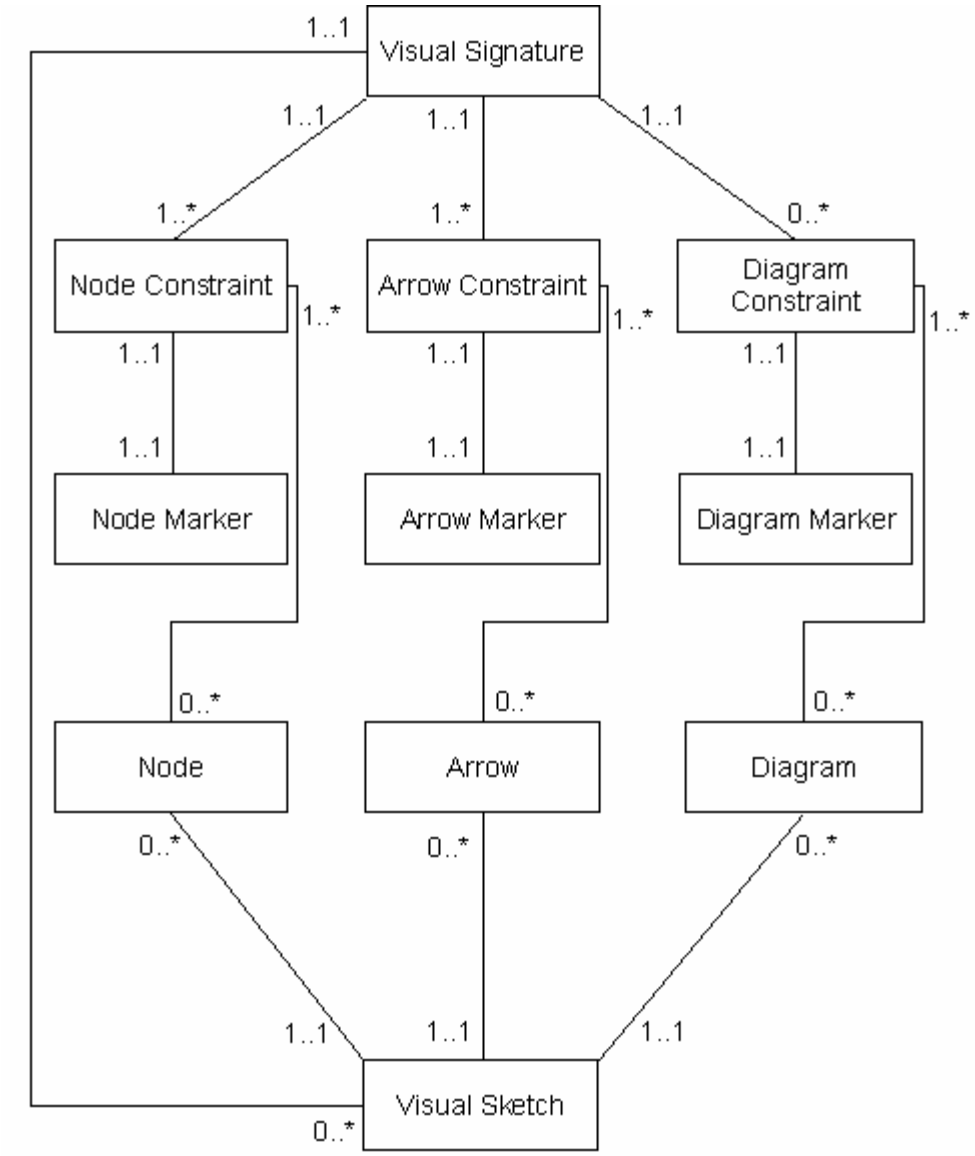


Figure 5-2: Structure of the generalized sketches formalism.

Each class' role and their relationships with one another will be more clearly explained in subsequent sections implementing the drawing tool aspect of the thesis.

5.2 Implementing drawing tool related functionality

An application that should function as a drawing tool requires at least two things. First, it needs a drawing surface where all the drawing is performed and the drawn image is displayed. Second, it needs the tools that make it possible to draw certain elements and to manipulate the drawings. These tools are most often gathered in a tool case visible to the user.

Although the drawing surface and the tool case have different areas of responsibility, they are also tightly connected to each other. This relationship will be more clearly when

investigating both in subsequent sections. But before going there, section 5.2.1 will continue where the previous section left off.

5.2.1 Draw objects

An important preparation before diving into the design process of the drawing tool is to clarify what exactly are the draw objects in which the drawing tool will operate on. These draw objects comprise the elements that the user is able to draw on the drawing surface of the application, and once drawn, is possible to perform typical drawing tool operations on, such as resizing and repositioning, as well as operations specifically adjusted for the generalized sketches formalism. In short, they are the elements that constitute a generalized sketch.
Figure 5-2 suggests that a sketch object can hold a number of node-, arrow-, and diagram-objects, and a reasonable conclusion to draw would be that these are the concrete draw objects of the drawing tool application. Hence, the program should support three different types of draw objects on the screen. Although their visual representation and behavior differs, they share many similar features especially in the way the drawing occurs, but also in the way already drawn objects are managed. This chain of thought ultimately leads to a proposal of an abstract base class DrawObject which gathers the common features that constitutes a draw object for Sketcher .NET. Since this is such an important part of the program, the necessary details of a co-operative draw object are listed in Table 5-1 below.

public properties	
Selected	Boolean flag. True if draw object is selected, otherwise false.
HandleCount	Returns the number of handles. Can be overridden.
public methods	
Draw	Draws the object.
GetHandle	Given a handle number, returns the corresponding point of the draw object.
GetHandleRectangle	Given a handle number, returns the handle rectangle.
DrawTracker	Draws the corresponding handles of the draw object if selected.
HitTest	Returns an integer value defining whether the object is hit by the mouse, and in that case, whether it is the object itself or an appurtenant handle as defined by its handle number.
PointInObject	Returns true if a point is in the object, otherwise false.
GetHandleCursor	Given a handle number, returns the corresponding mouse cursor.
IntersectsWith	Returns true if a given rectangle intersects with the draw object.
Move	Moves the draw object.
MoveHandleTo	Resizes the draw object.
Normalize	Normalizes the draw object.
Invalidate	Invalidates the draw object.

public events	
Moved	Occurs whenever the draw object is moved. Used for notifying connected draw objects.

Table 5-1: Interface of the abstract DrawObject class.

Because of the multiplicity issues of possible draw objects in a sketch, the need for a strongly typed collection class for the DrawObject type became highly valid, and so the DrawObjectList class was created. Besides just being a strongly typed collection class though, the DrawObjectList could also be used to facilitate work on sketches that concerns the entire collection of visible draw objects. So, DrawObjectList is in addition equipped with a range of extra properties and methods not commonly associated with collection classes.
The link between visual sketches and the draw objects occurring in it is perhaps more evidently described by a class diagram, as presented in Figure 5-3.

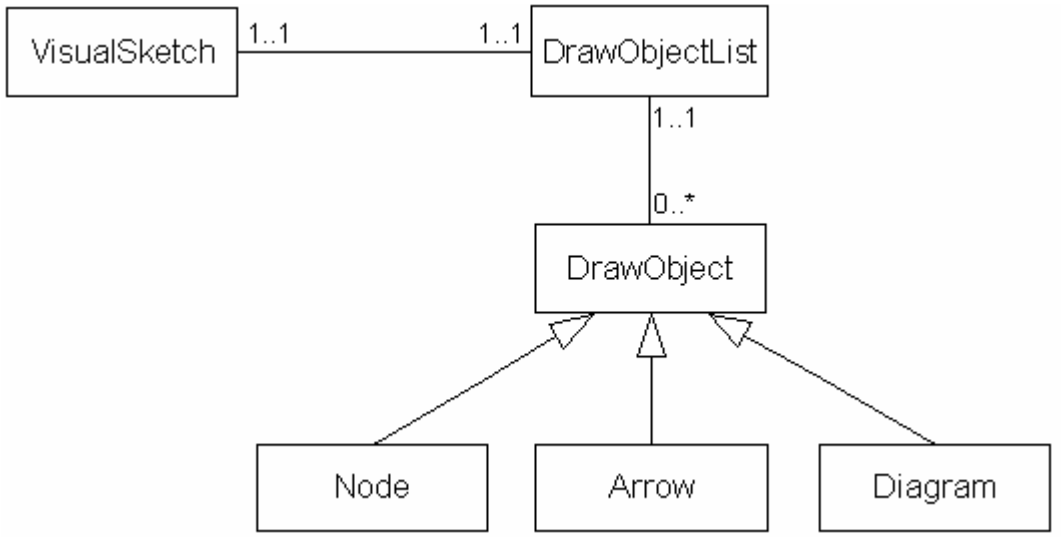


Figure 5-3: Class diagram over draw objects.

5.2.2 The tool case

Most document-centric window applications are normally equipped with a toolbar at the top of its main widget, listing shortcuts to the essential functionality. For drawing tool applications perhaps the most essential feature is that of the tool case; the “black box” in which all substantial tools for manipulating the drawing surface is comprised. It would therefore be a good idea for Sketcher .NET to explicitly place the tool case within the toolbar of the main application window, an instance of the MainForm class. The main window then becomes directly responsible for the visual representation of the tool case, and the entailing interaction with the user. Whenever the user selects a tool to be used, the MainForm instance has to reflect this back to the tool case, setting which tool that is currently active. This scenario will be demonstrated in the communication model of Figure 5-4 on page 66.

As a starting point, the tool case should explicitly include a tool for drawing nodes, a tool for drawing arrows, a tool for adding predicate declarations to diagrams, and a general-purpose tool for selection and common manipulation of drawn objects. Later expansions of the application could include a tool for zooming of the drawing surface and different tools supporting generalized sketches specific functionality. However, the former are preliminary enough for constituting a drawing tool framework for the generalized sketch formalism. These are represented in the following classes respectively:

- NodeTool
- ArrowTool
- DiagramTool
- PointerTool

Each of these tools has a similar interface that amongst other involves event handlers for the different mouse events that can occur on the drawing surface. They derive from an abstract base class ToolObject, that is, they are all tool objects. The ToolCase class which represents the tool case is a strongly typed collection class for the ToolObject type.
For demonstration of common behavior of the tools and the interplay with the drawing surface, the method declarations for the mouse event handlers in the ToolObject class is documented in the code sample of Table 5-2. Extending tool classes implement specific functionality for these event handlers, providing the necessary logic for correct execution.

```
 1.  /// <summary>
 2.  /// Base class for all tools.
 3.  /// </summary>
 4.  public abstract class ToolObject
 5.  {
 6.     ...
 7.
 8.     public virtual void OnMouseDown( VisualSketchControl vs, ... )
 9.     {
10.     }
11.
12.     public virtual void OnMouseMove( VisualSketchControl vs, ... )
13.     {
14.     }
15.
16.     public virtual void OnMouseUp( VisualSketchControl vs, ... )
17.     {
18.     }
19.
20.     ...
21.  }
```

Table 5-2: Common interface of tools.

The continuous collaboration with the drawing surface is further explained in the subsequent section.

5.2.3 The drawing surface

The drawing surface is the part responsible for user interactions with the sketch domain, allowing users to draw on screen. For the purpose of drawing sketches, it is an area in which the user may draw nodes and arrows between nodes. It is also arranged for further management of the drawn image, determined by both generalized sketches – and common drawing tool – functionality. The drawing surface is represented in the VisualSketchControl class. This is a control class, which basically means it has a visual appearance and is susceptible for a lot of events associated with interactive components, such as mouse events, keyboard events and paint events, and with the ability to respond to them in appropriate terms. This is important though, because the user interaction mainly happens through a mouse device, but also in some cases through a keyboard device, and so the drawing surface should at least be able to respond to those events.
The VisualSketchControl class is bisected in that it serves primarily as a remedy for visual representations of – and user interactions with – generalized sketches. While visual representation of sketches is the topic of section 5.3.2, the user interaction is the main focus here.
In order to perform operations on the drawing surface, there must be a close relation between the drawing surface and the tool case that provides the tools for manipulating it. When user actions occur on the drawing surface, the active tool of the tool case needs to be alerted so that it can perform the operations intended by the user. The drawing surface must therefore have some knowledge about the tool case, and the currently active tool in particular. On the other hand, if the active tool is to make changes to the proper sketch then it needs to receive information about which drawing surface the event has taken place. This information is retrieved in the argument list of the event handler methods through the VisualSketchControl object as demonstrated in the code sample of the previous section. An example on how an instance of the VisualSketchControl class notifies the active tool about an event is listed in the code sample of Table 5-3.

```
 1.  /// <summary>
 2.  /// Mouse move event.
 3.  /// </summary>
 4.  /// <param name="e"></param>
 5.  protected override void OnMouseMove(MouseEventArgs e)
 6.  {
 7.     ToolCase.ActiveTool.OnMouseMove( this, e );
 8.
 9.     base.OnMouseMove (e);
10.  }
```

Table 5-3: Responding to the event raised by the mouse movement over the drawing surface.

The interplay between the user, the tool case, and the drawing surface is pointed out in Figure 5-4 below.

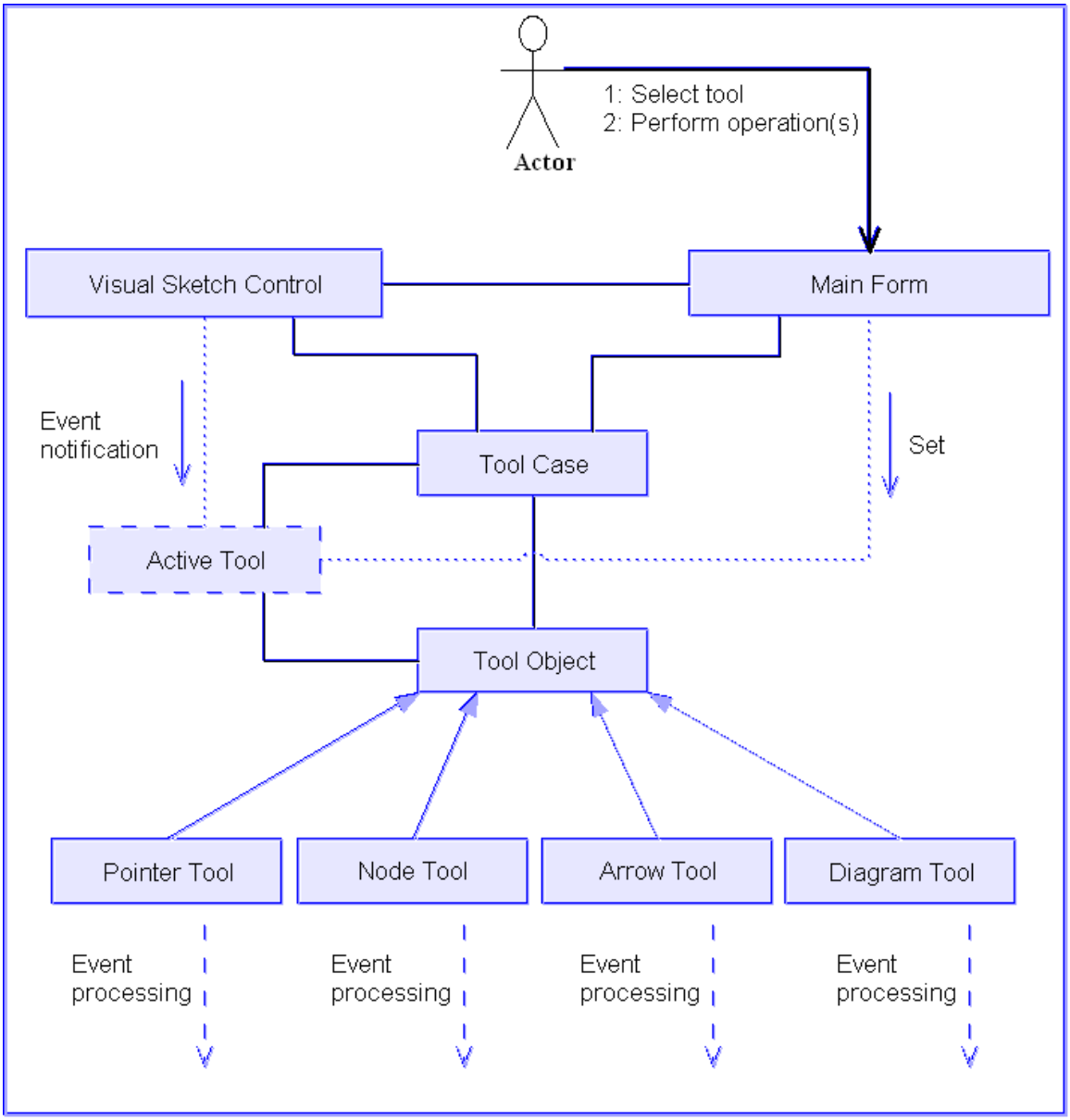


Figure 5-4: Communication model of the drawing tool.

For management issues of sketches, the drawing surface also needs to recognize whenever the user presses the right mouse button in order to display a corresponding popup menu if necessary, that is, when pressing the right mouse button with the mouse pointing over a draw object. Also, it could sometimes be convenient to use the keyboard, and so the drawing surface needs to listen to keyboard events for the appropriate keys. This can easily be done, as the code sample in Table 5-4 illustrates.

```
1.  protected override bool ProcessCmdKey(..., Keys keyData)
2.  {
3.     ...
4.
5.     switch ( keyData )
6.     {
```

```
 7.       case Keys.Up:
 8.          ...
 9.
10.       case Keys.Down:
11.          ...
12.
13.       ...
14.
15.       case Keys.Del:
16.          ...
17.     }
18.  }
```

Table 5-4: Handling key events.

5.2.4 Drawing issues

Besides figuring out how the internal workings of the generalized sketches language could be organized and commissioning of the drawing tool's event-based machinery, managing the drawing aspect was the most demanding task of the thesis. This was partially due to generalization issues of visual markers; the program should be able to manage almost any visual appearance of a given node, arrow or diagram predicate.

At the beginning, different approaches were tested that mainly made use of predefined methods provided by the .NET Framework for drawing of certain common node shapes and arrowheads. While this was a very easy way to draw and manage simple objects, it could not be used for more advanced visual elements such as a pentagon or any other self-made object which was desirable for the application. For example, it would sometimes be convenient to be able to have two arrowheads on the same end of the arrow or a diamond shape on a node, and this was simply not possible to achieve without some extra processing. As a result, the need for a path that would specify all coordinates of any given visual element emerged. In the .NET Framework, methods are provided for graphical management of such paths, including drawing, filling, and perform various 2-D transformations on them, and so this part of the drawing issues are resolved.

Further on, section 5.1 suggests that a marker is the element responsible for the visual appearance of its corresponding constraint, so it would be quite logical to place the drawing responsibility of a constraint within its marker.

A sketch on the other hand, is not directly engaged in constraints but instead presents concrete draw objects of the generalized sketches language which are nodes, arrows, and marked diagrams. Each of these draw objects has their visual appearance determined by a certain constraint, and so the corresponding marker should be the element also responsible for the visual appearance of concrete draw objects. The problem is that draw objects such as nodes and arrows, might have a number of different constraints attached, and as a consequence could be associated with a number of different visual markers, all defining its own visual appearance. For the sake of simplicity, this has been solved by making the last added constraint the asserted one, and hence, its corresponding marker determining its visual appearance.

Another crucial detail concerning the visual appearance of draw objects occurring in a sketch is its validity against its corresponding signature. Whenever a specific constraint changes its visual appearance in the signature, any present draw object in appurtenant

sketches satisfying that constraint should be automatically updated. A solution for this is to connect draw objects and constraints through references. That is, draw objects holds references to their belonging constraints instead of copies of them. However, in order to force corresponding sketch specifications to redraw themselves immediately, refreshing the screen display with updated markers, events must be used; whenever a signature changes, it fires an event to let appurtenant sketch specifications be notified about it being changed. This is possible because of the natural relationship between sketches and signatures that a sketch is based on a specific signature and therefore is required to hold a reference to this signature.
A last important detail that is part of the comprehensive drawing tool solution is the way draw objects indicate whether they are currently selected. Common drawing tool applications usually display *handles* – small black rectangles surrounding the draw object's boundaries and which it is possible to "grab" with a pointing device for resizing purposes. In this case, there are three different draw objects to consider; nodes, arrows, and marked diagrams.
For nodes it will be naturally to display eight such handles; one on each corner of its boundaries, which is given by a rectangle structure, in addition to one in the middle of each side of this rectangle.
For arrows it will be naturally with only two handles; one on each end of the arrow. However, in a future extended version of the program, it could be desirable to have additional handles for more advanced arrow management. In the meantime this is not supported though.
Marked diagrams on the other hand are either labeled with a string or an arrow, and thereby having two possible visual representations, both requiring different numbers of handles. An arrow representation should display two handles just like above, while a natural solution to the string representation would be to think of the string as a rectangular box, and hence displaying eight handles similar to that of nodes.

5.2.4.1 Drawing nodes

Nodes are represented by the Node class located in the Sketcher.Sketch namespace, extended from the abstract DrawObject class in order to obtain the necessary interface and properties for a draw object of the sketch domain. In addition to the mutual properties of draw objects, a node has a collection of currently attached node constraints, as well as a bounds property represented by a rectangle structure. This bounds property is consistent with the actually boundaries of a node object as it is updated in accordance to changes made to a node, either by location or by size.
The part responsible for the visual appearance of nodes is the node marker associated with the asserted node constraint. The relationship between nodes and the node markers responsible for their visual appearance is demonstrated in line 14 in the code sample of Table 5-5, where the node gives drawing instructions to the node marker.

```
1.  public class Node
2.  {
3.     ...
4.
```

```
 5.    public override void Draw( Graphics g )
 6.    {
 7.      if ( nodeConstraints.Count > 0 )
 8.      {
 9.        // Only draw the last constraint.
10.        int last = nodeConstraints.Count - 1;
11.        VisualNodeConstraint vnc = nodeConstraints[ last ];
12.        Rectangle norm = Node.GetNormalizedBounds( bounds );
13.        vnc.NodeMarker.Bounds = norm;
14.        vnc.NodeMarker.Draw( g );
15.
16.        // Draw node text.
17.        ...
18.      }
19.    }
20.    ...
21.
22.  }
```

Table 5-5: Drawing nodes.

The code that performs the actually drawing of nodes is located in the NodeMarker class. Since node objects use a GraphicsPath object to determine their coordinates and form, the drawing code for node markers is quite simple and contains only a few commands[4], as listed in Table 5-6.

```
 1.  public class NodeMarker : Marker
 2.  {
 3.    ...
 4.
 5.    public override void Draw( Graphics g )
 6.    {
 7.      g.FillPath( FillBrush, Path );
 8.      g.DrawPath( EdgePen, Path );
 9.    }
10.    ...
11.
12.  }
```

Table 5-6: Drawing node markers.

5.2.4.2 Drawing arrows

Arrows are represented by the Arrow class located in the Sketcher.Sketch namespace, and like the Node class, Arrow also extends the abstract DrawObject. Drawing arrows is a more advanced subject than drawing nodes however, mainly due to the fact that arrows are composed draw objects – an arrow consists of a body (line) holding possible multiple arrowheads, also called *caps* – but also the complications entailing from its mobility; an arrow can appear with any length and in any possible rotated form. This implies that not only the arrow body must be displayed with the correct angle, but also that each appurtenant arrowhead must be rotated comparatively and positioned correctly. And so

[4] Some extra processing is required in cases where displaying double edge lines on the node markers.

the work of drawing arrows is heavily influenced by the use of trigonometric formulas and functions for finding angles, horizontal, and vertical values, as illustrated in Figure 5-5.

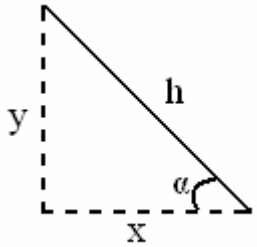


Figure 5-5: Decomposition of lines.

For example, if the space between the two possible caps on the same end of the arrow is fixed, for every angle the arrow may occur in, this space must be de-composed into a corresponding x-value and y-value, horizontal and vertical values respectively, by using the trigonometric functions Sine and Cosine in order to position the alternate cap with correct distance to the other.

Beyond that, drawing arrows follows the same strategy as drawing nodes in that it is the marker of the asserted (arrow) constraint, as represented by the ArrowMarker class, who does the actual drawing. Because of the different behavior and appearance of nodes and arrows however, an arrow only has two coordinate points instead of a boundary, referencing to where the arrow begins and where it ends respectively, in order to know where it should be drawn. Another necessity is that the arrow knows its source node and its target node. Every draw object is associated with a *Moved* event, and every node object is associated with a *Resized* event, and hence, the arrow manages to keep track of where its belonging nodes are at any time so that it can position itself accordingly, that is, the arrows are *always* visually connected to their sources and targets. In addition, an algorithm had to be developed to make the arrow ends position themselves on the borders of source and target nodes. This algorithm calculates the intersection points between arrow lines and the border line on the connected node, and is run whenever source nodes or target nodes changes its position, size, or visual appearance.

When it comes to the arrowheads, these are actually markers themselves represented by the ArrowCapMarker class, because they share the same properties as other markers and are responsible for their own visual appearance. Hence, the ArrowMarker class contains references to four different ArrowCapMarker objects, two on each end of the arrow. The drawing code within ArrowCapMarker is quite simple, as it is very similar to the drawing code of the NodeMarker class. Like nodes, arrow caps also have a path that specifies its coordinate points, and to draw and fill this path with the desired parameters can be done in the same way with nodes. However, the big difference is that arrow caps may be, and most often are, rotated as a result of the nature of arrows. What is needed is to manipulate the Graphics object with the proper 2-D transformations before performing the drawing operations on its GDI+ drawing surface. This is subject to the ArrowMarker class however, and the technique is shown in the code sample of Table 5-7.

```
1.  public override void Draw(Graphics g)
```

```
 2.  {
 3.     ...
 4.
 5.     // Draw single arrow body.
 6.     g.DrawLine( EdgePen, lineStart, lineEnd );
 7.
 8.     ...
 9.
10.     // Draw arrowheads.
11.     Matrix m = new Matrix();
12.     int x = StartPoint.X;
13.     int y = StartPoint.Y;
14.
15.     if ( StartCap != null )
16.     {
17.       m.Translate( x, y );
18.       m.RotateAt( angle, new Point( 0, 0 ) );
19.       g.Transform = m;
20.       StartCap.Draw( g );
21.       m.Reset();
22.     }
23.     ...
24.
25.     // Reset transform on the graphics object for further drawing.
26.     g.ResetTransform();
27.  }
```

Table 5-7: Drawing arrows markers.

5.2.4.3 Drawing diagrams

Drawing diagrams does not refer to drawing the nodes and arrows occurring in a diagram, but rather drawing the predicate label on marked diagrams (more than one arrow) in the sketch. As with nodes and arrows, diagrams leave the responsibility for its visual appearance to the corresponding marker of the asserted diagram constraint. The feature has not been fully tested due to time limitations, but the following outline shows how it should be done:

The marker has two possible representations; either by a string label positioned over the diagram, or an arc that spans over the arrows occurring in the diagram. This is the case whenever all arrows in the diagram have common source or target.

In the first case, the job consists of calculating the total area of which the diagram occurs in, then drawing a string representing the label at the center of this area. In the latter case, a path must be used representing all points on the corresponding arrows that it should span, and then draw an arc that intersects these points, taking into consideration the position of the common node area for these arrows so that the arc does not intersect with it. Also, the arc should somehow show that it is connected to its arrows by painting larger dots at each point in its path. The .NET Framework provides the necessary support for drawing arcs.

5.2.4.4 Drawing sketches

Drawing sketches is heavily connected to the visual representation of the sketch domain. This will be discussed in section 5.3.2. While section 5.3.2 is more concerned about the machinery behind interactive controls and how to connect the concept of a sketch to a visual component in the application window, this section deals with the actual sketch drawing and shows which characteristics should be present.
A sketch is represented by the VisualSketch class, and contains a list of draw objects as is represented by the strongly typed collection class DrawObjectList. This class exposes a public method for drawing all draw objects in the list. Basically, what this method does is only to give instructions to each draw object that it should draw itself, in a sequence such that the draw objects are represented correctly according to their z-order.
So when a sketch needs to redraw itself (exactly when this happens is subject to section 5.3.2), it passes on instructions to the corresponding DrawObjectList instance to draw the objects contained within the list. This process is illustrated in Figure 5-6.

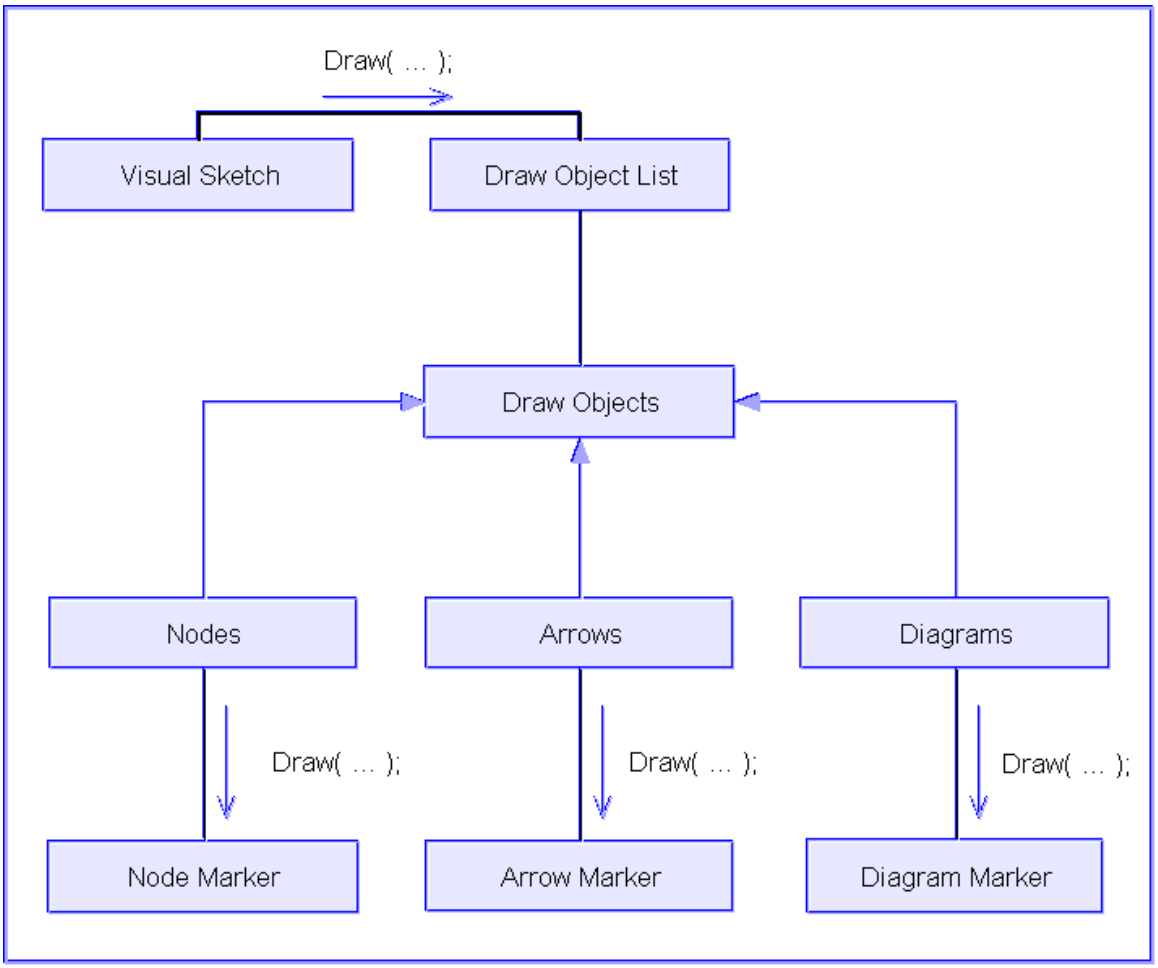


Figure 5-6: Communication model for drawing sketches.

5.3 Implementation of generalized sketches specific functionality

5.3.1 Visual representation of signatures

A tree structure will provide a lucid presentation of the signature as the signature's content is of a somewhat hierarchical format. With the signature's name as the root node in the tree, five non-erasable direct children nodes and no chance of adding more direct children is everything needed in order to display the signature's content in a well arranged matter. These five children consists of a general-purpose node and a general-purpose arrow with no constraints imposed on itself, and three different collections of the various types of constraints that is defined for a signature; node constraints, arrows constraints, and diagram constraints.
The Microsoft .NET Framework Base Class Library (BCL) offers a tree view control class that provides all the basic functionality for representations and navigation of tree structures, and also provides some nice methods for different common tree view operations. Located in the System.Windows.Forms namespace, the TreeView class

behaves suspiciously in the exact same way as the tree view control used in Windows Explorer in the Microsoft Windows XP operating system. This originates from the very fact that the majority of the controls in the System.Windows.Forms namespace use the underlying Window common control [54] as a base to build on, wrapping them into managed code for the .NET Framework. The updated common controls delivered in Microsoft Windows XP are a set of windows that are implemented by the common control library, which is a dynamic link library (DLL) included with the operating system.

If the signature should be viewed in a regular manner with the same image size on all items, using the provided tree view control would be a straightforward and time saving way, encapsulating the dirty details of the inner workings of a tree view control.

The existing solution suggests another view of the signature however, a view that would give a much more intuitive user interface for the signature. For displaying the visual appearance of constraints in a clearer manner, larger icons on constraints were desirable. Also, a better way to show the user the currently selected item in the signature involves some surrounding rectangular frames. Both of these innovative changes were of a visual character however, requiring doing some owner drawing of the TreeView control class by subscribing to its Paint event. However, the problem with the TreeView class (yields also for a few other common controls) is that it does not reveal any paint notification. That is, it is subject to an OnPaint method and provides the paint event which is possible to subscribe to, but the OnPaint method is never called, and hence, there is no opportunity to handle the paint event even if it has been subscribed.

In order to add the missing call to OnPaint, it requires doing some rather advanced work; what is needed is to enter the window procedure of the control and handle the WM_PAINT message in appropriate ways, including adding the call to OnPaint with correct arguments. To be able to override the window procedure of the tree view control, the class must be extended. Parts of the code are shown below in Table 5-8 for illustration.

```
 1.  public class TreeViewWithPaint : TreeView
 2.  {
 3.     ...
 4.
 5.     /// <summary>Occurs when a message is dispatched.</summary>
 6.     /// <param name="message">Message to process.</param>
 7.     /// <remarks>Overrides WM_PAINT, WM_ERASEBKGND.</remarks>
 8.     protected override void WndProc(ref Message message)
 9.     {
10.       const int WM_PAINT = 0x000F;
11.       ...
12.
13.       switch( message.Msg )
14.       {
15.          ...
16.
17.          case WM_PAINT:
18.             // Do some internal processing and drawing.
19.             ...
20.
21.             //Add the missing OnPaint() call.
22.             OnPaint( new PaintEventArgs( internalGraphics,
```

```
23.                     Rectangle.FromLTRB(
24.                       updateRect.left,
25.                       updateRect.top,
26.                       updateRect.right,
27.                       updateRect.bottom ) ) );
28.
29.           ...
30.
31.           return;
32.       }
33.
34.       base.WndProc(ref message);
35.     }
36.
37.     ...
38.
39.  }
```

Table 5-8: Adding the missing OnPaint.

Now it is possible to owner draw the entire surface of the tree view control, and the work of drawing icons with arbitrary size in the tree can be done in a relative easy manner. However, all logic concerning the drawing of the tree itself must be done manually, that is, drawing only tree nodes that are currently visible and in draw them at right places.
Another issue that pops up when personal performing all the drawing of a control is that the automatic scrolling provided by the control does not work exactly the way it should. The workaround for this problem that would provide the easiest way of adding correct automatic scrolling was to disable the auto scroll within the tree view control, and put the tree view control inside a System.Windows.Forms.UserControl class that would then act as a wrapper for the tree view and provide the automatic scroll features. But then, the control is no longer a tree view control, instead it is a user control, and so all functionality relevant for a tree view control had to be re-implemented, such as keyboard event handling, mouse event handling, and especially the expand/collapse functionality of the hierarchy view. The result was ending up with an (almost) owner implemented tree view control, as represented by the SignatureView class. VisualSignatureControl then uses SignatureView to properly display signatures in the desired way.

5.3.2 Visual representation of sketches

For representation of sketches, there were really not many alternatives. The sketch domain mainly deals with the drawing aspect of the thesis, and should provide an environment that both acts a drawing surface as well as the sketch representation itself. The user will generally interact with the drawing surface through the mouse device, but it is also possible to use the keyboard device for some basic operations on the drawn image. It is important with proper visual presentation and to reflect the user's actions immediately and correct are therefore prioritized. The drawing tool issues of the sketch domain were discussed in section 5.2.2. This section deals with the visual representation of sketches.
In Sketcher .NET, the visual representation of the sketch domain is comprised by the VisualSketchControl class. This class is a control class as it has a visual appearance, and

is susceptible for a lot of events associated with interactive components. For the purposes of sketch representations, the VisualSketchControl class consists of a visual sketch object as identified in section 5.1. As no predefined control class with similar features like the ones of VisualSketchControl exists, the solution Windows Forms offer is to extend the general-purpose System.Windows.Forms.UserControl class with the features needed for specific tasks. This is done in the class declaration of the extended class as the following code sample in Table 5-9 shows.

```
1.  using System;
2.  using System.Windows.Forms;
3.
4.  namespace Sketcher.Controls
5.  {
6.    public class VisualSketchControl: UserControl
7.    {
8.      ...
9.    }
10. }
```

Table 5-9: Extending the System.Windows.Forms.UserControl class.

Because sketches are visually concepts, their representation is closely connected to the painting of the control. Every time a control is *invalidated*, or in other words needs to redraw itself, a paint event is raised for that control. This must be utilized if the VisualSketchControl class should be able to draw sketches. In an extended control class, this can be done in two different ways; either by subscribing to the paint event in the same way a control subscribes to events of another control, or by overriding the base class method that fires the paint event, the OnPaint method. The latter is possible because such event raising methods are declared with the *virtual* keyword, making extending classes able to override those methods, and is the preferred way to do it because it somehow gives cleaner code. However, overriding the OnPaint method must be handled with care to avoid failure to the program.
In addition to the regular incidents causing all controls to be redrawn, the VisualSketchControl class should also redraw itself whenever the user performs operations on it – either by creating new draw objects or by repositioning or resizing existing ones – or whenever the corresponding signature of the sketch changes bringing about possible changes to the visual appearances of existing draw objects occurring in the sketch. At such circumstances, repainting of controls can be manually forced or provoked by invalidating pieces of – or the entire – client area of the control with a single method call to the control. The invalidated area is then dealt with in accordance to the code in the OnPaint method of the control as well as registered paint event-handlers.
The code that paints sketches is listed below in Table 5-10.

```
1.  protected override void OnPaint(PaintEventArgs e)
2.  {
3.    DrawGrid( e.Graphics );
4.
5.    if ( VisualSketch != null )
6.    {
7.      VisualSketch.Draw( e.Graphics );
8.
```

```
 9.       if ( VisualSketch.TmpObject != null )
10.          VisualSketch.TmpObject.Draw( e.Graphics );
11.    }
12.
13.    DrawNetSelection( e.Graphics );
14.
15.    base.OnPaint( e );
16.  }
```

Table 5-10: Drawing a sketch.

The code that performs the actual sketch drawing is located in the line segment 5 – 11. It starts with a check to see whether its VisualSketch object is not a null pointer. If so, the Draw method on the VisualSketch object is called with the PaintEventArgs argument. This method in turn examines each draw objects contained within the VisualSketch object, and performs the necessary actions for drawing those objects in a correct manner using the Graphics object defined by the PaintEventArgs object. The code on line 9 and 10 performs the drawing of the temporary element that is, if existing, currently under creation and therefore not physically belonging to any sketch yet. More details on drawing may be found in section 5.2.4.
The other code lines in the OnPaint method include calls to a Drawgrid method and a DrawNetSelection method which draws a background grid and a net selection rectangle respectively, if needed. Before returning from the OnPaint method, a call to the base class' method lets subscribers of the control's paint event be noticed about the event.

5.3.3 Signature operations

Besides the common functionality for management of the constraint hierarchy as given by the VisualSignatureControl class, a popup menu tailored to each element in the signature has been implemented to separate the valid operations on them. This is due to the special nature of signatures that some elements are mandatory and must be present while other elements have no such terms.
The rest of the signature operations relevant for the present thesis are those dealing with the creation of the different constraints. A natural approach and also how the existing solution solves them, is to use dialog boxes for each type of constraint; node, arrow and diagram. These dialog boxes should provide advanced interfaces for the properties of its corresponding constraint, that is, name, description and marker. The challenging interfaces are the ones directed towards the markers, which are highly customizable visual elements. Although creating these interfaces is almost a pure technical job, it is also a quite time consuming effort.
An example of a possible dialog box for node constraints is displayed in Figure 5-7 below.

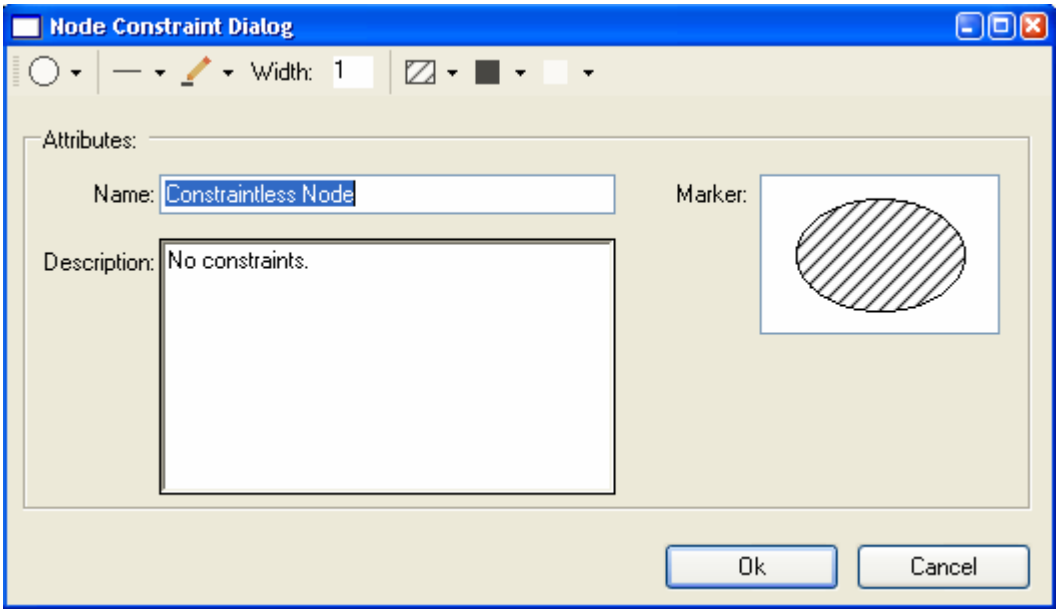


Figure 5-7: Node constraint dialog.

Here, the interface on marker manipulation is solved by a surface that is responsible for displaying the current marker, and which is able to respond to mouse click events. Associated marker tools that can be used on this surface – all of which represents parameters to the visual appearance of a node marker – are located in a toolbar, consisting of a:

- Shape tool
- Edge line tool
- Edge color tool
- Fill pattern tool
- Foreground color tool
- Background color tool

Hence, manipulating a marker would be a quite similar task to that of manipulating images in ordinary drawing applications.
Due to time limitations, the interfaces for arrow constraints and diagram constraints have not been implemented yet, but can be approached in the same way as the nodes constraint dialog, with a few minor modifications.

5.3.4 Sketch operations

Sketch operations that are substantial for the work of this thesis mainly comprise of two things:

- Support for the drawing tool activity of the program, that is, creation and management of sketches.
- Support for adding predicate declarations to nodes, arrows, and diagrams occurring in sketches, i.e. marking elements of sketches with predicate labels taken from their corresponding signature.

First of all, the drawing tool activity of the program has been implemented as described in section 5.2, accordingly the user is now capable of drawing nodes and arrows between nodes, and performing other general management operations for sketches such as repositioning, resizing, and deletion of selected elements (possible multiple simultaneously).
For the second part, most of the underlying logic is implemented for marking sketches with predicate declarations, but no GUIs have been created yet for adding such declarations to nodes, arrows and diagrams. While the two former GUIs are pretty much a pure technical job consisting of displaying the corresponding predicate labels that are available from the signature, with the ability to add and remove them from a list of declared predicates for the node or arrow in question, the latter requires more heuristic efforts to succeed. A solution to this problem could include the following steps:

- Display available diagram predicates (taken from the corresponding signature), making the user choose what type of predicate that should be used to mark some diagram in the sketch.
- Requiring the user to select the diagram that should be marked with the predicate label by displaying the arity shape of the asserted diagram predicate with some kind of indications on what type of draw object should be selected next in the corresponding sketch. This ought to be done to enforce correct typing which is a crucial criterion, and the procedure should start with all arrows in turn, thus ending up with independent nodes of the shape (if any).

5.4 Implementation of data persistence methods

The program should be able to produce both signature documents and sketch documents, and so support for object persistence should be implemented for:

- Signatures
- Sketches

Technologies applied are XML Schemas for validating files and XML serialization for persisting and constructing objects through serialization and de-serialization. The approach these two technologies offer when coordinated is the following:
First, the structure of each type must be carefully analyzed and then implemented in a corresponding XML Schema Definition (XSD) file. Every owner-defined type in the C# code whose state is required to be persisted also needs to be defined in the XSD file as well. When building complex data objects, which are the case with both signatures and

sketches, this is often a non-trivial task and must be performed in a precise way. But once done, it provides an elegant way to validate data objects. An example on the structure of XML Schemas is shown in Table 5-11.

```
 1.  ...
 2.
 3.  <xs:complexType name="VisualNodeConstraint">
 4.    <xs:sequence>
 5.      <xs:element name="Name" type="xs:string" />
 6.      <xs:element name="Description" type="xs:string" />
 7.      <xs:element name="NodeMarker" type="sketcher:NodeMarker" />
 8.    </xs:sequence>
 9.  </xs:complexType>
10.
11.  <xs:complexType name="VisualNodeConstraintCollection">
12.    <xs:sequence minOccurs="0" maxOccurs="unbounded">
13.      <xs:element name="VisualNodeConstraint"
14.        type="sketcher:VisualNodeConstraint" />
15.    </xs:sequence>
16.  </xs:complexType>
17.
18.  ...
```

Table 5-11: Defining complex types in XML Schema.

Second, each class (type) in the C# code that is part of the structure obtained by the analysis of the initial task should be marked with special XML attributes declaring their types. Also, each element of these classes as defined by the XSD file should be public fields or properties with both get and set functionality, tagged with XML attributes their type. Public properties or fields not part of the XSD structure should be explicitly declared to be ignored. Examples of such XML attributes are given in Table 5-12.

```
 1.  [XmlType("VisualNodeConstraint",Namespace="Sketcher.Xml")]
 2.  public class VisualNodeConstraint
 3.  {
 4.    ...
 5.
 6.  }
 7.
 8.  [XmlElement("NodeMarker",typeof(NodeMarker))]
 9.  public NodeMarker NodeMarker
10.  {
11.    get { ... }
12.    set { ... }
13.  }
14.
15.  [XmlIgnore()]
16.  public Rectangle Bounds
17.    ...
```

Table 5-12: XML attributes.

When all this has been done, saving and loading signatures and sketches are performed in a quite simple way as shown in Table 5-13.

```
 1.  // Serialize the signature to a file.
```

```
 2.  XmlSerializer ser = new XmlSerializer( typeof(VisualSignature) );
 3.  StreamWriter writer = new StreamWriter( filename );
 4.  ser.Serialize( writer, sign );
 5.  ...
 6.
 7.
 8.  // Set up validation.
 9.  XmlValidatingReader xval = new XmlValidatingReader( new
10.  XmlTextReader(filename) );
11.  xval.ValidationType = ValidationType.Schema;
12.  xval.Schemas.Add( _xschemaCache );
13.
14.  // De-serialize a signature from file.
15.  XmlSerializer ser = new XmlSerializer( typeof(VisualSignature) );
16.  VisualSignature = (VisualSignature) ser.Deserialize( xval );
17.  ...
```

Table 5-13: Serializing and de-serializing.

5.5 Other issues

5.5.1 Resources

Resources in Sketcher .NET consist mainly of icons and XML Schema Definition files for validation of document files, that is, signature and sketch files. In order to ensure that these resources are available to the program at runtime, they are not treated as system files appearing in some predefined catalog forcing the application to rely on their existence, but rather explicitly encapsulated with the assembly that contains the executable file, at the expense of a somewhat bigger executable. They are so-called *embedded resources*.

5.5.2 Use of external libraries

The visual appearance of many .NET provided controls might be thought of as somewhat plain and boring. A couple of freely available external libraries have been used to spice up the graphical interface. This involves the toolbar, the menu, the tab control holding sketches and the docking manager for control flow on the main form of the application, All of which simulates the appearance of Microsoft Office products as well as Visual Studio .NET. The improvement of the toolbar is the icon feedback when hovered by the mouse, in addition to a neater look. The menu used adds icons to the menu options, greatly improving the visual experience of the application. The tab control has overall a more professional look and is coordinated according to the other control components for a better visual totality, while the docking manager makes it possible to give a more intuitive interface and brings along a convenient way to manage signatures in proportion to sketches.

6 CONCLUSION

6.1 Prospective features

The generalized sketches formalism and its corresponding mathematical foundations established in category theory offer many promising features as a unifying framework for all graphical notation systems used within software engineering.
A drawing tool program for this formalism would allow translation of existing modeling languages and diagram types into a unified and formal format, ideal for comparison, integration and automation of conversion- and translation processes between such sketch specifications and lower-level specifications such as programming languages and SQL. Thus, it should have a great potential for practical use within software engineering, contributing to development of formal software specifications in a comprehensible way, and with that adjusting the level of bad software.
The program developed in this thesis is not there yet, but is rather intended to serve as a basis for further development of a tool that will, some day, inherit all these features and hopefully be conducive to an alternative way of thinking within current software development methods.

6.2 Status of the program

To the current state of the art, the program has implemented the internal structure of the generalized sketch formalism together with data persistence methods, and provides a comprehensive drawing tool platform for drawing graphs and maintaining their integrity.
It has successfully managed to connect the concept of generalized sketches to a drawing tool capable of drawing graphs, so now users can create sketches with constraint less nodes and constraint less arrows with the corresponding predicate symbols taken from a predefined signature. The relationship between signatures and sketches is treated in such a way that visual markers in sketches at all times reflect the signature correctly.
The GUIs for adding predicate declarations to nodes, arrows and diagrams have not been created yet, although most of the underlying logic has been implemented. Nor are the GUIs for adding arrow constraints and diagram constraints created, and so the basic framework is not completed. Thus, not all goals set in the requirements specification have been achieved, but as intended, the program is capable of serving as a basis for further work.

6.3 Further work

The first priority should be to complete the framework of basic operations for the generalized sketches formalism. This involves implementing GUIs for the remaining generalized sketches functionality, thus creating the following:

- An arrow constraint dialog for signatures.
- A diagram constraint dialog for signatures.
- A properties dialog for nodes and arrows occurring in sketches.
- A diagram tool dialog for adding predicate declarations to diagrams occurring in sketches.

After adding the rest of the basic operations, it could be convenient to:

- Implement a more punctilious algorithm for automatic arrow positioning.
- Implement support for arrow bends and/or right-angled arrows.
- Implement edge brackets on nodes for standardized arrow attachment.
- Implement zooming capabilities.
- Implement common GUI functionality such as cut, copy, paste, undo, redo, and print.
- Implement support for converting sketch specifications into different image formats.

Before implementing the rest of desired functionality like:

- Integration of signatures.
- Integration of sketches in the same or different signatures.
- Support for various operations with sketches.
- Illuminating and shading of arrow diagram predicates.
- Decompositions of graphs.
- Support for meta-sketches whose nodes are sketches and arrows are sketch mappings, and with the ability to maintain integrity of meta-sketches.

Further, the program could be augmented with modules interpreting sketches and diagram markers and translating sketches to specific code, and vice versa, thus providing a major universal platform for formal specifications within software engineering.

REFERENCES

1. Sebesta, R.W. (1993). *Concepts of Programming Languages*. 2 ed., The Benjamin/Cummings Publishing Company, Inc.
2. Loeckx, J., H.-D. Ehrich, and M. Wolf. (1996). *Specification of Abstract Data Types*. 1 ed., John Wiley & Sons. 272 p.
3. Unknown. (Unknown). *Software crisis*. Accessed 03.16 2006 on World Wide Web: http://en.wikipedia.org/wiki/Software_crisis.
4. Larman, C. (2001). *Applying UML and Patterns: An introduction to Object-Oriented Analysis and Design and the Unified Process (2nd Edition)*, Prentice Hall PTR.
5. Chapman, J.R. (1997). *Software Development Methodology*. Accessed 04.02 2006 on World Wide Web: http://www.hyperthot.com/pm_sdm.htm.
6. Martin, J. and J.J. Odell. (1992). *Object-oriented analysis and design*. New Jersey, Prentice-Hall, Inc.
7. Chen, P.P.-S. (Unknown). *Homepage of Dr. Peter Chen at Louisiana State University*. Accessed 03.28 2006 on World Wide Web: http://bit.csc.lsu.edu/~chen/chen.html.
8. Engels, G. and L. Groenewegen. (2000). Object-Oriented Modeling: A Roadmap.
9. Chen, P.P.-S. (1976). The Entity-Relationship Model - Toward a Unified View of Data.
10. Pender, T. (2003). *UML Bible*. Indianapolis, Indiana, Wiley Publishing, Inc.
11. Diskin, Z. (1997). Generalized sketches as an algebraic graph-based framework for semantic modeling and database design.
12. Ehrig, H., M. Grosse-Rhode, and U. Wolter. (Unknown). On the Role of Category Theory in the Area of Algebraic Specifications.
13. Fiadeiro, J.L. (2005). *Categories for Software Engineering*, Springer-Verlag Berlin.
14. Barr, M. and C. Wells. (1999). *Category Theory for Computing Science*. 3 ed., Prentice Hall International.
15. Makkai, M. (1993). Generalized sketches as a framework for completeness theorems.
16. Diskin, Z. and B. Cadish. (1995). Variable Sets and Functions Framework for Conceptual Modeling: Integrating ER and OO via Sketches with Dynamic Markers.
17. Diskin, Z. and B. Kadish. (1998). The Arrow Manifesto: Towards software engineering based on comprehensible yet rigorous graphical specifications.
18. Diskin, Z., B. Kadish, and F. Piessens. (1999). What vs. How of Visual Modeling: The Arrow Logic of Graphic Notations.
19. Diskin, Z. (2003). MATHEMATICS OF UML: Making the Odysseys of UML less dramatic.
20. Lohr, S. (2004). *One Small Step in Uphill Fight as Linux Adds a Media Player*, New York Times. Accessed 04.10 2006 on World Wide Web: http://www.nytimes.com/2004/06/28/technology/28linux.html?ex=1246161600&en=97d3f89979101d82&ei=5088&partner=rssnyt.

21. Sobell, M.G. (2001). *A Practical Guide to Linux*, Addison-Wesley.
22. Unknown. (Unknown). *Homepage of GTK+ - The Gimp Toolkit* Accessed 05.17 2006 on World Wide Web: http://www.gtk.org/.
23. Trolltech. (Unknown). *Qt Overview*. Accessed 03.15 2006 on World Wide Web: http://www.trolltech.com/products/qt/index.html.
24. Unknown. (Unknown). *Homepage of Glade User Interface Builder*. Accessed 05.17 2006 on World Wide Web: http://glade.gnome.org/.
25. Unknown. (Unknown). *Homepage of the GNU Project*. Accessed 05.02 2006 on World Wide Web: http://www.gnu.org/.
26. Jager, M. (2005). *Trolltech Launches Major New Version of Qt*. Accessed 04.10 2006 on World Wide Web: http://www.linuxpr.com/releases/7928.html.
27. Trolltech. (Unknown). *Trolltech's Dual Licensing Business Model*. Accessed 03.15 2006 on World Wide Web: http://www.trolltech.com/company/model.html.
28. Hawlitzek, F. (2002). *Java 2*. Programming Series, Pearson Education Limited.
29. Sun Microsystems Inc. (Unknown). *Java Technology Overview*. Accessed 03.24 2006 on World Wide Web: http://java.sun.com/overview.html.
30. Sun Microsystems Inc. (Unknown). *Java Foundation Classes (JFC/Swing)*. Accessed 04.25 2006 on World Wide Web: http://java.sun.com/products/jfc/.
31. Unknown. (Unknown). *Homepage of Eclipse*. Accessed 04.20 2006 on World Wide Web: http://www.eclipse.org/.
32. Marinilli, M. (2003). *Swing and SWT: A Tale of Two Java GUI Libraries*. Accessed 04.25 2006 on World Wide Web: http://www.developer.com/java/other/article.php/2179061.
33. Microsoft. (Unknown). *Technology Overview*. Accessed 04.16 2006 on World Wide Web: http://msdn.microsoft.com/netframework/technologyinfo/overview/default.aspx.
34. Microsoft. (2005). *ECMA C# and Common Language Infrastructure Standards*. Accessed 03.28 2006 on World Wide Web: http://msdn.microsoft.com/netframework/ecma/.
35. Stutz, D. (2002). *The Microsoft Shared Source CLI Implementation*. Accessed 03.10 2006 on World Wide Web: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/mssharsourcecli.asp.
36. Unknown. (2006). *What is Mono?* Accessed 03.28 2006 on World Wide Web: http://www.mono-project.com/Main_Page.
37. Novell. (Unknown). *Homepage of Novell*. Accessed 04.25 2006 on World Wide Web: http://novell.com.
38. Microsoft. (Unknown). *Downloads*. Accessed 04.12 2006 on World Wide Web: http://www.microsoft.com/downloads.
39. Olefsky, J. (Unknown). *The History and Popularity of the C Programming Language*. Accessed 05.07 2006 on World Wide Web: http://www.jakeo.com/words/clanguage.php.
40. Schildt, H. (2003). *C++: The Complete Reference, Fourth Edition*. 4 ed. The Complete Reference, McGraw-Hill/Osborne.
41. Soulie, J. (2005). *History of C++*. Accessed 04.24 2006 on World Wide Web: http://www.cplusplus.com/doc/information/history.html.

42. Norwegian Computing Center. (Unknown). *Homepage of Norwegian Computing Center*. Accessed 05.24 2006 on World Wide Web: http://www.nr.no/.
43. Jones, B.L. (2001). *Sams Teach Yourself C# in 21 days*, Sams.
44. Schildt, H. (2002). *C#: The Complete Reference*. The Complete Reference, McGraw-Hill/Osborne.
45. Deitel, H.M., et al. (2002). *C# For Experienced Programmers*. 1 ed., Prentice Hall PTR.
46. Ramakrishnan, R. and J. Gehrke. (2003). *Database Management Systems*. 3 ed. New York, McGraw-Hill.
47. Unknown. (Unknown). *Homepage of W3C*. Accessed 04.16 2006 on World Wide Web: http://w3c.org.
48. Ray, E.T. (2001). *Learning XML*. 1 ed., O'Reilly & Associates, Inc.
49. Hunter, D., et al. (2002). *XML, 2. utgave*. 2 ed., IDG Norge Books AS.
50. Fallside, D.C. and P. Walmsley. (2004). *XML Schema Part 0: Primer Second Edition*. Accessed 05.06 2006 on World Wide Web: http://www.w3.org/TR/xmlschema-0/.
51. Bray, T., et al. (2004). *Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation 04 February 2004*. Accessed 03.14 2006 on World Wide Web: http://www.w3.org/TR/REC-xml/.
52. Burke, S. (2001). *Using the Microsoft .NET Framework to Create Windows-based Applications*. Accessed 05.10 2006 on World Wide Web: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/netwinforms.asp.
53. Prosise, J. (2001). *Windows Forms: A Modern-Day Programming Model for Writing GUI Applications*. Accessed 05.10 2006 on World Wide Web: http://msdn.microsoft.com/msdnmag/issues/01/02/winforms/.
54. Microsoft. (Unknown). *General Control Reference*, Microsoft Accessed 04.07 2006 on World Wide Web: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/shellcc/platform/commctls/common/refs.asp.