# Commonality Analysis: A Case Study in the Compositional Approach

by

Grzegorz Roman Kacy
B.Sc., Silesian Technical University, Gliwice, Poland, 1993

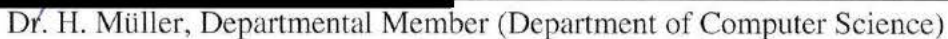
A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the
Department of Computer Science

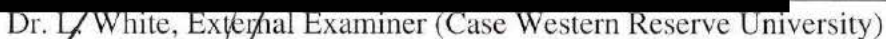
We accept this thesis as conforming
to the required standard

Dr. D. Hoffman, Supervisor (Department of Computer Science)



Dr. H. Müller, Departmental Member (Department of Computer Science)

Dr. G. McLean, Outside Member (Department of Mechanical Engineering)



Dr. L. White, External Examiner (Case Western Reserve University)

© Grzegorz Kacy, 1997

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author

Supervisor: Dr. Daniel M. Hoffman

## Abstract

With today's pressures on software developers, many organizations are investigating product line architectures in order to increase software productivity and quality. A strategy for deciding about family members, known as Commonality Analysis, is in use at Lucent Technologies. With this strategy a family is defined by identifying commonalities, i.e., assumptions that are true for all family members and variabilities, i.e., assumptions about what can vary among family members. This strategy is a part of a domain engineering process known as FAST (Family-Oriented Abstraction, Specification, And Translation). The goal of this process is to develop facilities for rapidly generating members of a family. In this thesis the Commonality Analysis process is applied to the Graph Editor family, and based on its results, an Application Engineering Environment (AEE) is created. The AEE consists of the framework, codebase and application generator. Using the AEE, four different family members (Java applications) are generated.

Examiners:

Dr. D. Hoffman, Supervisor (Department of Computer Science)

Dr. H. Müller, Departmental Member (Department of Computer Science)

Dr. G. McLean, Outside Member (Department of Mechanical Engineering)

Dr. L. White, External Examiner (Case Western Reserve University)

# Table of Contents

**Abstarct** ii

**Table of Contents** iv

**List of Figures** viii

**List of Tables** x

**Acknowledgements** xi

**1 Introduction** 1

1.1 The Problem 1

1.2 Solution Strategy 2

1.3 The FAST process 3

1.4 Commonality Analysis for Graph Editors 4

1.5 Thesis Overview 5

**2 Related Work** 6

2.1 Domain Engineering and Domain Analysis 7

2.2 Methods 8

2.3 Other related work ___ 10

**3 Terms and Concepts ___ 13**

2.1 Domain Engineering ___ 13

2.2 Commonality Analysis ___ 14

2.2.1 Commonalities ___ 14

2.2.2 Variabilities ___ 14

2.2.3 The Commonality Analysis Document ___ 15

2.2.4 The Application Engineering Environment ___ 15

2.3 The Object-Modeling Technique object diagrams ___ 16

2.3.1 Class ___ 17

2.3.2 Generalize (Inherits) Relationship ___ 18

2.3.3 Aggregate Relationship ___ 19

2.3.4 Multiplicity Relationship ___ 19

2.2 The Model-View-Controller Paradigm ___ 19

2.3 Observer Design Pattern ___ 20

2.4 Graph Terminology ___ 22

**4 Commonality Analysis for Graph Editors ___ 24**

4.1 Analysis ___ 24

4.2 Commonalities ___ 25

4.3 Variabilities ___ 27

4.4 Parameters of Variation ___ 29

**5 Application Engineering Environment** **35**

5.1 Design Issues 35

5.2 Library 37

5.2.1 GUI Module 38

5.2.2 Graph Module 42

5.2.3 Tools Module 44

5.2.4 Custom 45

5.2.5 Model-View-Controller classes 46

5.2.6 Application class diagram 47

5.3 Framework 47

5.4 Application Generation 49

5.4.1 Model File 49

5.4.2 Generator (appgen.pl) 52

**6 Application Engineering** **56**

6.1 Testgraph Editor 56

6.2 Graph Draw 59

6.3 Mathmania Game 62

6.4 Image Overlay 65

**7 Conclusion and Future Work** **67**

7.1 Summary and Conclusion 67

7.2 Lessons Learned and Future Work 69

**Bibliography** ____ **71**

**Appendices**

**A Commonality Analysis Document** ____ **76**

A.1 Introduction ____ 76

A.2 Overview ____ 77

A.3 Dictionary of Terms ____ 77

A.3.1 GUI ____ 77

A.3.2 Graph Abstraction ____ 78

A.4 Commonalities ____ 80

A.4.1 GUI ____ 80

A.4.2 Graph Abstraction Commonalities ____ 81

A.4.3 Other Commonalities ____ 81

A.5 Variabilities ____ 81

A.5.1 GUI ____ 81

A.5.2 Graph Abstraction ____ 82

A.5.3 Other ____ 82

A.6 Parameters of Variation ____ 83

**B Code Base Module Guide** ____ **87**

**C DSDL Semantics** ____ **91**

# List of Figures

Figure 2.1: Domain Engineering and Application Engineering (Two Life Cycles)............7

Figure 3.1: Application generation process using Commonality Analysis........................16

Figure 3.2: OMT Class symbol ........................................................................17

Figure 3.3: OMT Generalize relationship..............................................................18

Figure 3.4: OMT Agregation relationship symbol ...................................................18

Figure 3.5: OMT Multiplicity relationship symbol ....................................................19

Figure 3.6: Model-View-Controller paradigm .........................................................21

Figure 3.7: Sample graph....................................................................................22

Figure 3.8: Graph and one of its subraphs..............................................................23

Figure 5.1: Codebase modules..............................................................................38

Figure 5.2: Toolbar class family diagram................................................................39

Figure 5.3: Dialog class family diagram..................................................................40

Figure 5.4: Sample dialog box...............................................................................40

Figure 5.5: Execute class family diagram................................................................41

Figure 5.6: Graph family class diagram...................................................................44

Figure 5.7: Model-View-Controller paradigm in the Graph Edito..................................46

Figure 5.8: Graph Editor class diagram...................................................................48

Figure 5.9: Application Engineering Environment framework........................................49

Figure 5.10: Meta-tag replacement scheme during the code generation............................53

Figure 6.1: Testgraph Editor with the IntSet class testgraph...........................................57

Figure 6.2: Graph Draw with sample graphs..................................................................60

Figure 6.2: Mathmania game...........................................................................................62

Figure 6.4: Image Overlay application.............................................................................64

# List of Tables

Table 4.1: Parameters of Variation for V1 .......... 29

Table 4.2: Parameters of Variation for Variabilities V2-V22 .......... 30

Table 4.3: Parameters of Variation for V13 .......... 31

Table 4.4: Parameters of Variation for V14 .......... 31

Table 4.5: Parameters of Variation for V17 .......... 32

Table 4.6: Parameters of Variation for V15 .......... 32

Table 4.7: Parameters of Variation for V18 .......... 33

Table 4.8: Parameters of Variation for V16 .......... 33

Table 4.9: Parameters of Variation for V19 .......... 34

Table 5.1: DSDL keywords .......... 50

Table 5.2: Meta-tags and Variabilities .......... 55

Table 6.1: AEE files summary .......... 68

# Acknowledgements

Special thanks to my supervisor Dr. Daniel Hoffman of the Department of Computer Science for his guidance throughout the program and for his help in the preparation of this manuscript. Thanks to my dear parents who are always there for me and thanks to all my friends for making my life in Victoria fun and enjoyable. Financial assistance received from Dr. Hoffman and from the University of Victoria under NSERC grant OGP000 8067 is also gratefully acknowledged.

# Chapter 1

# Introduction

## 1.1 The Problem

Today's demanding software industry puts a lot of pressure on designers, engineers and developers. They are often expected to perform rapid production while maintaining careful engineering of their products. With today's large software systems with millions lines of code, multiple versions, tight release deadlines and competitive market, this is not a trivial task. Software engineers have begun to look at other engineering fields, where methods for rapidly producing carefully engineered products have long been explored, and hope that applying some of their production strategies will be beneficial. Many organizations today are investigating product line architectures. Such architectures would allow them to deliver new products faster, improve quality, reengineer legacy systems and manage many product variations.

## 1.2 Solution Strategy

Weiss [1] suggests a strategy based on three hypotheses:

- *The Redevelopment Hypothesis*: Most software development is mostly redevelopment. In particular, most software development consists of creating variations of existing software systems. Usually, each variation has more in common with other variations than it has differences from them.
- *The Oracle Hypothesis*: It is possible to predict the types of changes that are likely to be needed to a system over its lifetime. In particular the types of variations of a system that will be needed are predictable.
- *The Organizational Hypothesis*: It is possible to organize both software and the organization that develops and maintains it in such a way as to take advantage of predicted changes. In particular, the software and its developers may be organized so that a change of any predicted type can be made independently of changes of other types and so that making such a change requires changing at most a few modules in the system. The task of producing a new version of the software then consists of making relatively independent changes in different modules of the software.

These hypotheses suggest a software production strategy in which one plans for a system to exist in a number of variations, attempts to predict those variations, identifies what they have in common, and reuses the common aspects in producing the variations [1]. Such a system can be viewed as a software family. Predicting the family members is not trivial. The idea of family is not well formalized, there are no rules that enable engineers

to identify families easily, and prediction of expected variations is difficult [1]. However the payoff for conducting such an analysis can be quite high.

## 1.3 The FAST process

A strategy for deciding about family members, known as commonality analysis, is in use at Lucent Technologies. In this strategy a family is defined by identifying commonalities, i.e., assumptions that are true for all family members, variabilities, i.e., assumptions about what can vary among family members, and common terminology This strategy is a part of a domain engineering process known as family-oriented abstraction, specification, and translation (FAST). The goal of the FAST process is to develop facilities for rapidly generating members of a family. Performing a commonality analysis is an early step in this process[1]. The FAST process consists of three major steps:

1. Domain Engineering: defining family (Commonality Analysis).
2. Development of an Application Engineering Environment (AEE): production framework for rapidly generating family members.
3. Application Engineering: production of family members.

In the second step there are two known approaches. The so called "language approach" and the *compositional approach.* In the language approach a domain specific programming language is designed as a part of the AEE for generating the family members. In the compositional approach the family members are generated from chunks of reusable code.

## 1.4 Commonality Analysis for Graph Editors

The Graph Editor family consists of applications whose main task is to display or modify graphs. A Graph Editor allows the user to create, remove and edit arcs and nodes in a graph and to store graphs in files. Besides that, the application will have other features specific to particular family members.

In this thesis Lucent's Commonality Analysis is applied to the Graph Editor family, and based on its results, an AEE is created. Creation of the AEE is the original and the most significant part of the thesis. Using the AEE, four different family members are generated.

The AEE consists of the framework, codebase and application generator. The framework provides the skeleton for the environment, directory structure, and rules for creating the family members. The codebase is a set of Java source files from which the target application is built. The application generator automatically generates family member applications based on their model files. The model files are written in a simple Domain Specific Description Language developed for. Four family members have been generated: Testgraph Editor, GraphEd, Mathmania Game, and Image Overlay. All of them based on displaying the graph, yet quite different.

The work presented here is a non-trivial example of the *compositional approach* to Commonality Analysis. In the compositional approach the target application is incrementally built (composed) from the existing chunks of source code in the codebase. The only other example of the compositional approach - Floating Weather Station [1] is much simpler than this example. In the FWS there are five classes and 330 lines of code,

while in this work there 55 classes and 9438 lines of code. The major work in this thesis is in the design and development of the AEE. In the application generator, an improved scheme for implementing variabilities and parameters of variation in the generation process has been introduced. In the codebase a reusable GUI and graph classes have been created. Also, this is the first example of where Design Patterns are used in the codebase.

## 1.5 Thesis Overview

Chapter 2 of this thesis reviews the previous work in the area of domain analysis in general and Commonality Analysis in particular, and places this work in perspective. Chapter 3 describes the terms and concepts used in this thesis. Commonality Analysis for Graph Editors is presented in Chapter 4; Chapter 5 provides the details about the Application Engineering Environment. The four generated applications and the generation process are shown in Chapter 6. The thesis research work is summarized, and the directions that future work may take are explained in Chapter 7.

# Chapter 2

# Related Work

## 2.1 Domain Engineering and Domain Analysis

Commonality Analysis is part of the area of Software Engineering called domain analysis. Domain analysis and domain engineering are often used interchangeably. Although domain analysis as a term may pre-date domain engineering, domain engineering is the more inclusive term, and is the process of

- defining the scope, i.e., domain definition
- analyzing the domain, i.e., domain analysis
- specifying the structure, i.e., domain architecture development
- building the components, e.g., requirements, designs, software code, and documentation

for a class of subsystems that will support reuse [2][40].

Figure 2.1 [3] presents the process and products of the overall domain engineering activity, and shows the relationships and interfaces of domain engineering to the

conventional (individual) system development (application engineering) process. This has come to be known as the two life cycle model [40].

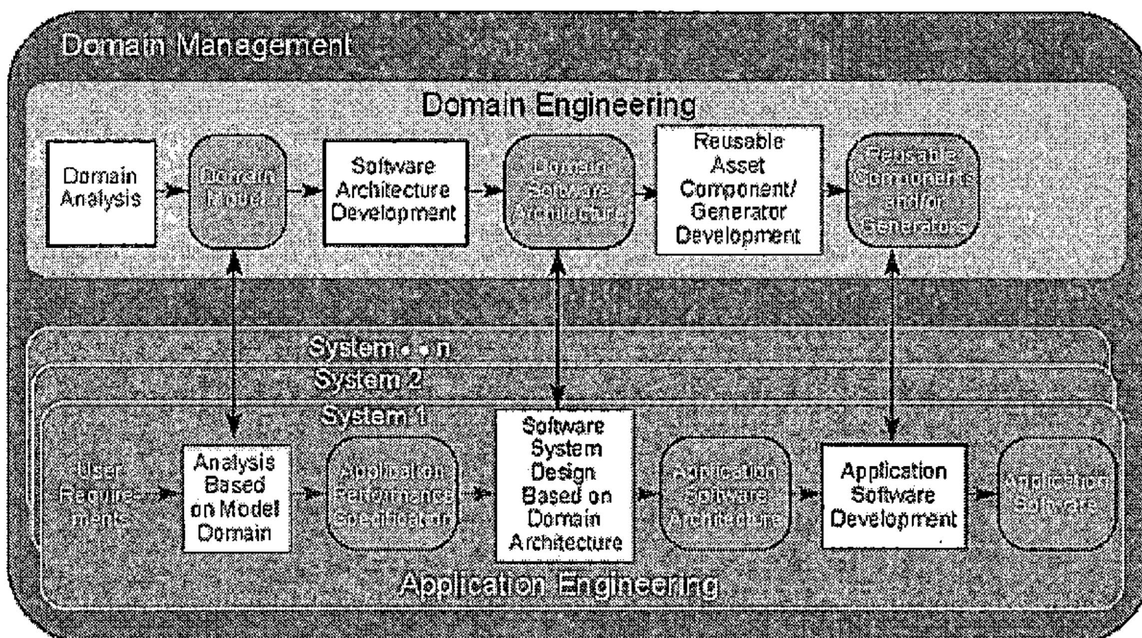



Figure 2.1: Domain Engineering and Application Engineering (Two Life Cycles)

Domain analysis was first introduced by J.Neighbours in 1984 [4]. It is an activity within domain engineering and is the process by which information used in developing systems in a domain is identified, captured, and organized with the purpose of making it reusable when creating new systems in that domain [5].

Although domain analysis is a simple concept, its application is not simple. Domain analysis is similar to systems analysis but is applied to a family of systems rather than a single application. The objective in domain analysis is to find commonalities among systems in the same domain and synthesize generic models or architectures that characterize families of applications. Domain analysis is a key activity in planned reuse [7].

## 2.2 Methods

Research on domain analysis in recent years has produced many approaches. Examples of domain analysis methods include:

- Feature Oriented Domain Analysis (FODA) from the Software Engineering Institute (Carnegie Mellon University) [8].
- The Synthesis methodology from Software Productivity Consortium (SPC) [12]
- Organization Domain Modeling (ODM) from Organon Motives, Inc. [15].
- Commonality Analysis/FAST from Lucent Technologies [1]
- Prieto-Diaz's Sandwich method [10].
- Domain Analysis and Design Process from Defense Information Systems Agency/Center for Information Management (DISA/CIM) [16]

FODA is a domain analysis method based upon identifying the prominent or distinctive features of a class of systems. It resulted from an in-depth study of other domain analysis approaches [8]. The FODA process is divided into three phases:

- Context analysis.
- Domain modeling (Feature analysis, Information analysis, Operational analysis)
- Architecture Modeling

The FODA method is well-defined and has been applied on both commercial and military applications. It was applied to the Army Movement Control Domain [9], In-Transit Visibility Modernization (ITVMOD) domain analysis effort [11], and Telecommunication Automated Prompt and Response Domain at NORTEL [13]

The ODM method evolved and was subsequently formalized by Mark Simos (Organon Motives, Inc.) with collaboration and sponsorship from Hewlett-Packard Company, Lockheed-Martin, and the DARPA STARS. ODM was developed and refined as part of the overall reuse/product line approaches developed under the STARS program. The STARS reuse approach decomposes reuse technologies into several layers of abstraction, specifically: Concepts, Processes, Methods, and Tools. An example of a "concept" is the Conceptual Framework for Reuse Processes (CFRP), a conceptual foundation and framework for understanding domain-specific reuse in terms of the processes involved [14]. ODM has been applied to small-scale and relatively large-scale projects. It has been used at Hewlett-Packard [15], the Air Force CARDS Program, and the Army STARS Demonstration Project [14].

Synthesis (SPC) is a methodology for constructing software systems as instances of a family of systems having similar descriptions. Its distinguishing features are:

- Formalization of domains as families of systems that both share many common features and also vary in well-defined ways.
- Adaptive reuse of software artifacts.
- Reduction of system building to resolution of necessary requirements and engineering decisions.
- Model-based analyses of applications to help understand the implications of system-building decisions [12].

A principal concern of Synthesis is to make software easy to produce. Synthesis methodologies are concerned both product and process together. To achieve this,

Synthesis processes consist of two subprocesses: domain engineering and application engineering [17].

The Commonality Analysis process (and FAST) evolved from Synthesis and was started as an experimental process at Lucent Technologies in 1992 and is still evolving. Some projects have gained sufficient confidence in it that they are starting to make it a standard part of their software production process [1]. Out of the 17 different domains at Lucent where a commonality analysis has been tried, 10 have been completed, one never finished, and six are in progress [1]. Although some development groups at Lucent consider the analysis to be just an early step in their application of the FAST process, nearly all have come to view it as a worthwhile endeavor in itself.

## 2.3 Other related work

Similar work is conducted at University of York (UK) with cooperation of Rolls-Smiths Engine Controls Ltd. Their work focuses on creating reusable architectures in the avionics domain (aero-engine control systems - EEC) and the aim was to develop an EEC architecture for the control of thrust reverse systems which can be reused across a wide variety of engines and thrust reverser designs. Their technology was termed RACE (Reusable Architecture Creation and Employment). In RACE a variablity analysis is used to identify possible variations in a family of systems, which then guides the creation of a generic architecture and a set of architectural "plug-ins" [20]. An outline of the RACE process is shown below

- Study Evidence

- Variability Identification
- Architecture Creation
- Architecture Usage

At Motorola, Inc. a domain analysis approach similar to Commonality Analysis served to help create reusable architectures and components in the development of a real-time embedded system in the domain of domain portable wireless communication devices. Their approach can be summarized as follows [21]:

1. *domain analysis*, i.e., analyzing the common features and their variations across current and future products in the domain. This entails scoping the domain, gathering the relevant product information, identifying and classifying the features, and analyzing their commonalities.
2. *architecture*, i.e., defining the high-level partitioning of the software into components. This includes identifying the application layer subsystems, extending the object model, defining the layered view and the task view, and developing the scenarios.
3. *subsystem/component design*, i.e, organizing the design of each subsystem/component using the commonalities and variations identified in the domain analysis.

Research conducted in Fachhochschule Konstanz (Germany) initiated and led the development of the Open Software Framework for Manufacturing (OSEFA). This framework is a domain-specific black-box framework containing components that model an application domain's entities, concepts, and logic so one can create an application from components. OSEFA focuses on object technology and design patterns [22].

Software product line engineering has been also tried in CelsiusTech Systems, one of Sweden's (and Europe's) leading suppliers of command and control systems. Their

case study documented in [24], is to our knowledge the largest example of a software product line that has been described in detail in the open literature [23].

# Chapter 3

# Terms and Concepts

## 2.1 Domain Engineering

Domain engineering has been described as a controlled, reliable system of activities to convert a community's knowledge into repositories of reusable information. A reuse system within an organization consists of a set of agents (people or tools) that execute a reuse plan. A reuse system infrastructure consists of one or more repositories which contain code, code specifications, code generators, and reuse plans for any of the items in the repository [26]. Arango [35] divides Domain Engineering into three phases:

- *Domain Analysis* - an attempt to identify the objects, operations, and relationships between what domain experts perceive to be important about the domain.
- *Infrastructure Specification* - The infrastructure specification must define the aspects of the problem domain that should be supported by the repositories of components so a particular reuse system achieves the desired level of performance. This involves selecting the functionality to be captured and determining how it should be packaged into components and how the components should be indexed.

- *Infrastructure Implementation* - Implementation of the infrastructure.

## 2.2 Commonality Analysis

Weiss [1] defines a Commonality analysis (CA) as a one approach to defining a software family by identifying commonalities, i.e., assumptions that are true for all family members, variabilities, i.e., assumptions about what can vary among family members, and common terminology for the family. A commonality analysis forms the basis for designing reusable assets that can be used to produce rapidly family members.

### 2.2.1 Commonalities

Identifying common aspects of the family is a central part of the analysis. Accordingly, a commonality analysis contains a list of assumptions that are true for all family members [1].

### 2.2.2 Variabilities

Whereas commonalities define what's always true of all family members, variabilities define how family members may vary. Variabilities define the scope of the family by predicting what decisions about family members are likely to change over the lifetime of the family. A commonality analysis contains a list of variabilities and the range of values for each variability. These ranges of values are known as parameters of variation. Choosing a value for a parameter of variation specifies a subset of the family [1].

### 2.2.3 The Commonality Analysis Document

The Commonality Analysis Document plays the central role in the analysis. It is a compendium of the knowledge for the analysis and a guide for developers. The document consists of the following eight sections [1]:

1. *Introduction*: Describes the purpose and expected use of the analysis.
2. *Overview*: Briefly describes the domain and its relationships to other domains.
3. *Dictionary of Terms*: Provides a standard set of key technical terms used in discussions about and descriptions of the domain.
4. *Commonalities*: Provides a structured list of assumptions that are true for all members of the domain.
5. *Variabilities*: Provides a structured list of assumptions about how family members may vary.
6. *Parameters of Variation*: Quantifies the variabilities, specifying the range of values and the decision time for each. The decision time specifies the time when the concrete value is chosen for the parameter of variation (generation/runtime).
7. *Issues*: Provides a record of the alternatives considered for key issues that arose in analyzing the family.
8. *Appendicies*: Includes a variety information useful to reviewers, designers, language designers, tool builders for the family, and other potential users of the analysis

### 2.2.4 The Application Engineering Environment

The Application Engineering Environment (AEE) is a software environment for automated generation of the software family members. This software environment is

based on the Commonality Analysis. Figure 3.1 presents the AEE in the application development process based on the Commonality Analysis.

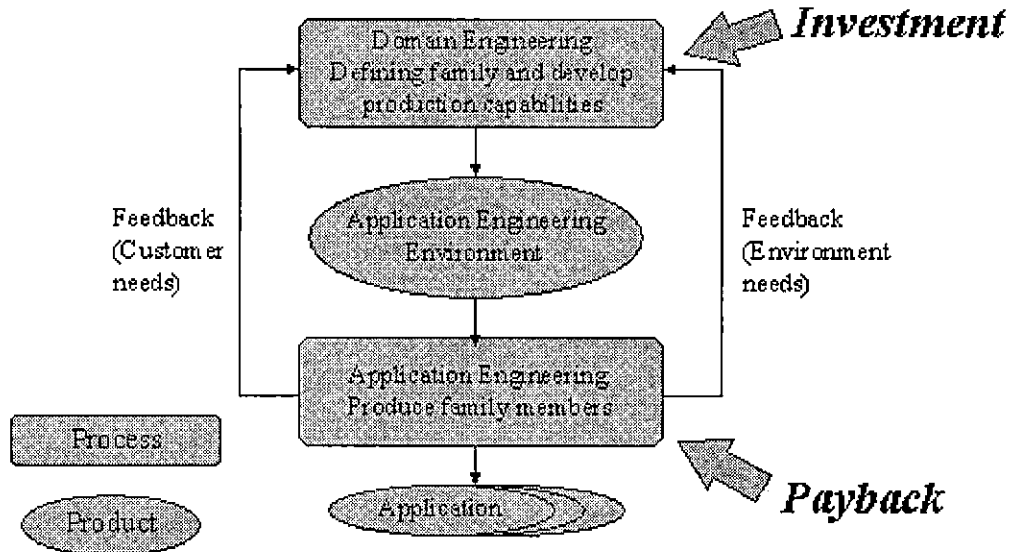



Figure 3.1: Application generation process using Commonality Analysis

# 2.3 The Object-Modeling Technique object diagrams

The OMT Object diagram is part of the Object Modeling Technique (OMT). OMT is a language-independent graphical notation, which was developed to be part of an object-oriented approach to software development. It came out of the work of James Rumbaugh at General Electric Research and Development. OMT consists of three related but different viewpoints, each capturing important aspects of the system. The object model represents the static, structural aspects of a system, while the other two models (dynamic and functional model) describe the transformational and functional aspects of a system. The object model is used to describe the structure of objects in a system, by representing graphically by means of object diagrams containing object classes. In these diagrams the classes define the attribute values and operations of each object instance. Furthermore, an

object diagram contains associations to describe the relations between different classes [28]. The following is the explanation for the OMT graphical symbols used in this work.

## 2.3.1 Class

A class icon is drawn as a 3-part box, with the class name in the top part, a list of attributes (with optional types and values) in the middle part, and a list of operations (with optional argument lists and return types) in the bottom part. The attribute and operation sections of the class box can be suppressed to reduce detail in an overview. Suppressing a section makes no statement about the absence of attributes or operations, but drawing an empty section explicitly states that there are no elements in that part.

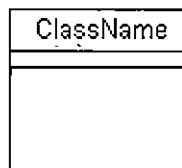



Figure 3.2: OMT Class symbol

## 2.3.2 Generalize (Inherits) Relationship

A generalize relationship between classes shows that the subclass shares the structure or behavior defined in one or more superclasses. A generalize relationship is a solid line with an arrowhead pointing to the superclass.

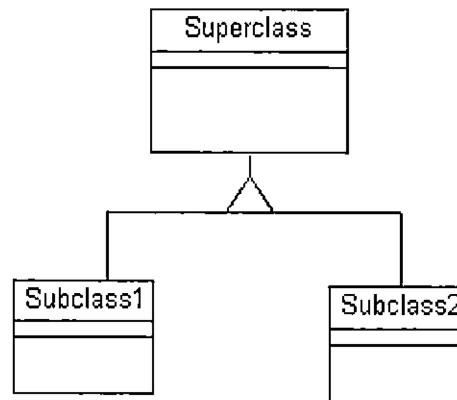



Figure 3.3: OMT Generalize relationship

## 2.3.3 Aggregate Relationship

The aggregate relationship is the "part-whole" or "a-part-of" relationship between two classes. The class at the client end of the aggregate relationship is sometimes called the aggregate class. An instance of the aggregate class is an aggregate object. The class at the supplier end of the aggregate relationship is the part whose instances are contained or owned by the aggregate object.

The aggregate relationship is used to show that the aggregate object is physically constructed from other objects or that it logically contains another object. The aggregate object has ownership of its parts. An aggregate relationship is a solid line with a diamond at one end. The diamond end designates the client class.[28]

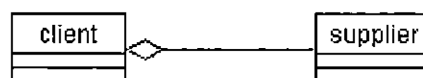



Figure 3.4: OMT Agregation relationship symbol

### 2.3.4 Multiplicity Relationship

The multiplicity relationship specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity constrains the number of related objects. There are special line terminators to indicate certain common multiplicity values. A solid ball is the symbol for "many", meaning zero or more. No multiplicity symbols means a one-to-one association.

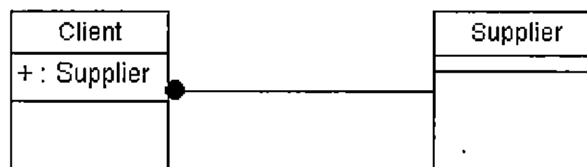



Figure 3.5: OMT Multiplicity relationship symbol

## 2.2 The Model-View-Controller Paradigm

One popular way to structure user interfaces is the Model-View-Controller (MVC) paradigm. In this scheme, user interface is divided into three distinct but coresponding components:

- *Model*: The application data being manipulated and conducting all the computations
- *View*: A specification of how aspects of a model are represented to the user
- *Controller*: A specification of how the user can communicate or interact with the application in order to request changes to the view, or to the underlying model

The concept was formulated first for Smalltalk programming environments, but in fact it is independent of object orientation.

In the MVC the state of the model can only be changed through the controller. Views can present one underlying model in multiple ways to a user, for example, graphically and textually. The controller regulates and synchronizes the flow of messages between the objects. Many frameworks combine the view and the controller roles, because there is a close coupling between the way in which the information is presented and how it is manipulated.

## 2.3 Observer Design Pattern

Design Patterns are the descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. A pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue [27].

Design patterns can be divided into three groups:

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

The Observer pattern is a behavioral pattern and it defines one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically. The key objects in this pattern are the subject and the observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.

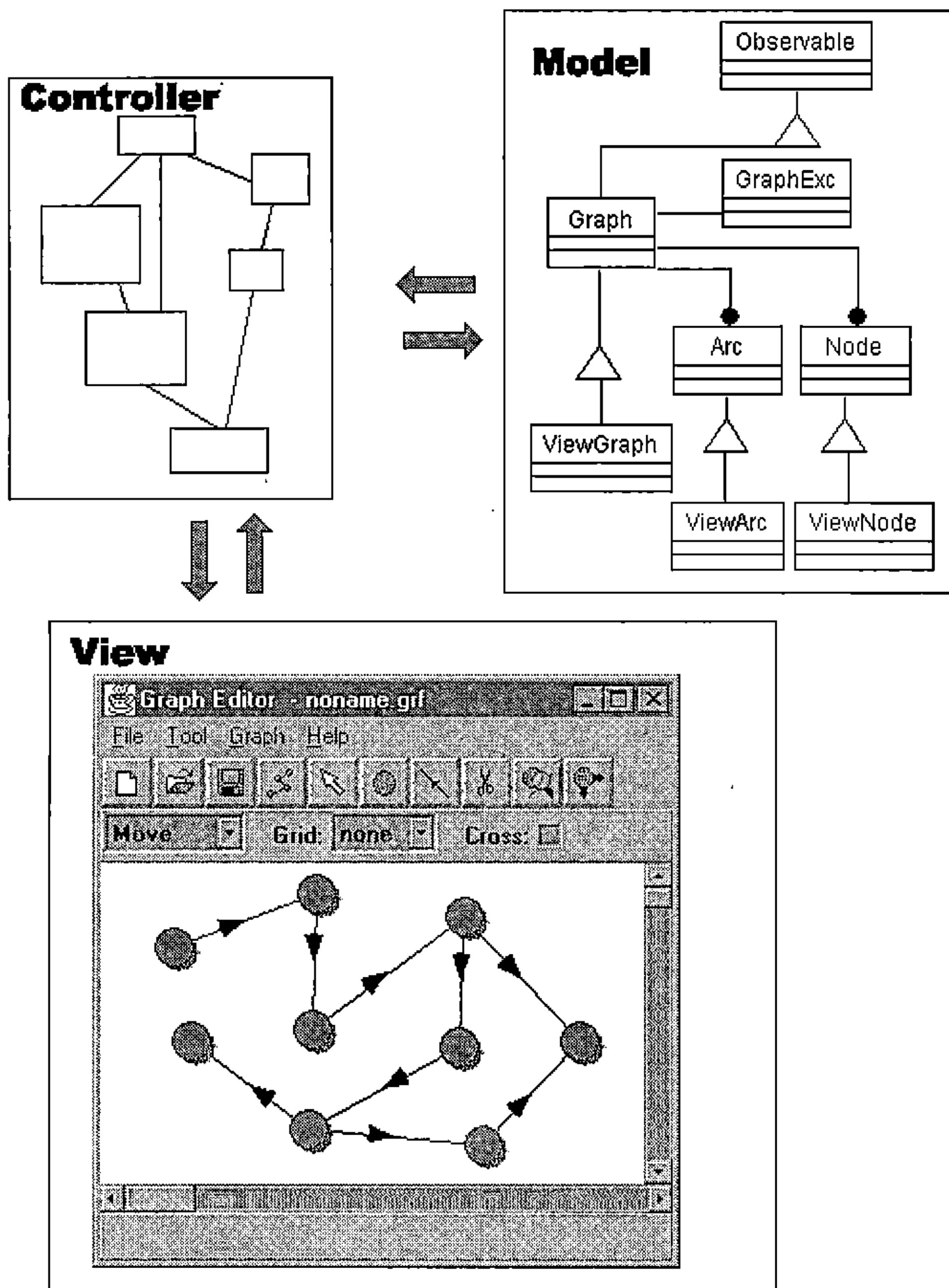



Figure 3.6: Model-View-Controller paradigm

A design pattern names, abstracts, and identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design. The Observer pattern may be used in any of the following situations:

- When an abstraction has two aspects, one dependent on the other.
- When a change to one object requires changing others, and their number is unknown or there are no assumptions about what these objects are.

The Observer pattern can be used to implement the Model-View-Controller paradigm. The subject will correspond to the Model and observer to the View.

## 2.4 Graph Terminology

A **Graph** is a finite set of dots called **nodes** (or vertices) connected by links called **arcs** (or edges). More formally: a simple graph is a (usually finite) set of nodes **(V)** and set of unordered pairs of distinct elements of **V** called arcs **(E)**.

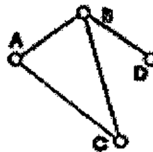



Figure 3.7: Sample graph

For the graph showed on Figure 3.7 the node and arc set are respectively:

***V={A,B,C,D}***

***E={(A,B), (A,C), (B,C), (B,D)}***

**Directed graph**

A graph in which the arcs are directed. Formally: a digraph is a (usually finite) set of nodes V and set of ordered pairs **(a, b)** (where **a, b** are in **V**) called arcs. The node **a** is the source (initial) node of the arc and **b** the destination (terminal) node. The directed arcs are marked with arrowheads pointing to the destination node.

## Subgraph

A subgraph of a graph is some smaller portion of that graph.

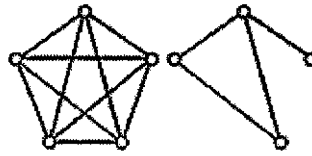


Figure 3.8: Graph and one of its subraphs

# Chapter 4

# Commonality Analysis for Graph Editors

This chapter presents the Commonality Analysis (CA) for the Graph Editors family. It discusses the chosen and rejected alternatives, looks closely at the CA document in particular, focusing on the commonalities, variabilities and parameters of variation.

## 4.1 Analysis

While conducting the Commonality Analysis for the Graph Editors domain, the analyst/developer has to ask himself or herself a number of fundamental questions including:

- What is the purpose of the analysis?
- What functionality must be provided by the graph editor?
- How will the graphs be presented in the editor?
- What will be the implementation language of the editor?
- What kind of family members are likely to be generated?
- What changes might appear in the future versions of the applications?

The answer to these and other questions determines the commonalities and variabilities for the domain.

It has been decided that each Graph Editor will be a Java application. The graph will be presented in a variety of ways in the view area, which is the central, largest part of the application window. Each of the nodes and arcs can have a different size, color, shape, label, etc. These few sentences already suggest a number of commonalities and variabilities. For example, the first commonality is that each application is a Java program. The Presentation of the graph determines the variabilities for the node, arc and graph properties. The desired look of the family members determines the GUI commonalities and variabilities. These include the presence and properties of menu, status bar, toolbar view area and dialog boxes. The commonalities and variabilities are described in detail in Sections 4.2 and 4.3. There were a number of rejected alternatives during the analysis process. One of them was variability for the presentation of the graph. The current version does not include graph layout algorithms. This alternative was rejected to minimize the already quite large size of the code and work for this thesis.

## 4.2 Commonalities

The commonalities are the assumptions that are true for all family members of the Graph Editor Application family. They are divided into three categories:

- GUI commonalities
- Graph Abstraction commonalities
- Other commonalities

The GUI commonalities describe the common GUI features of all family members:

1. Each family member is a standalone Java application.
2. Each application's user interface will consist of a window with the View Area (View Canvas).
3. Each application will have the Close, Minimize and Full Screen controls.
4. The View Area will contain an image of a graph.
5. The user can click or drag the mouse cursor on the View Area.
6. Depending on the current mode the user can select, add, remove, edit and move arcs and nodes.

The Graph Abstraction Commonalities describe the common graph related aspects of all family members:

1. Nodes and arcs can be added or removed from graph.
2. There is no arc connecting a node with itself.
3. Each node and arc has a finite number of properties.
4. Each graph has a finite number of properties.
5. The image of the graph in View Area represents the current graph stored in main memory.
6. Graphs are stored in files.

The `Other Commonalities' contains only one entry which is:

The code documentation for the family member will be generated with *javadoc*. Javadoc is a Java Development Environment tool used to extract the documentation from the source code

## 4.3 Variabilities

While commonalities identify what is common to all members of the family, variabilities describe how members of the family may differ. The variabilities are also divided into three groups:

- GUI variabilities
- Graph Abstraction variabilities
- Other variabilities

There are 22 variabilities in all, with identifiers V1 through V22. The GUI variablities are as follows:

1. *Properties of the Application Window (V1):* describing the main application window's properties such as size and resizability.
2. *Presence of the Menu (V2):* describing the presence application's menu. Menu is placed in the top part of the window below the caption bar.
3. *Items on the Menu (V3):* describing the additional custom menu items with associated command classes.
4. *Presence of the Toolbar (V4):* describing the presence of the toolbar. Toolbar is placed in the top part of application window above the control bar (if present) and below the menu (if present).
5. *Buttons on the Toolbar (V5):* describing the additional custom toolbar buttons with associated command classes.

6. *Presence of the Control Bar (V6):* describing presence of the application's control bar. The Control bar is located above the View area and it holds the controls graph editing tool selection, grid size, current coordinates and rubberbanded guide checkbox.
7. *Presence of the Status Bar (V7):* describing the presence of the application's status bar. Status bar is always is located in bottom part of the application window and it is used to display messages.
8. *Contents of the About Box (V8):* describing the message string in the About dialog box. This message usually includes the name of the application and application engineer (or author).
9. *Name of the application in the Caption Bar (V9):* describing the name appearing in the caption bar of the window. This name is usually the name of the application.
10. *Presence of Graph Properties Dialog Box (V10)*: describing the presence of the Graph Properties Dialog Box. This dialog box allows changes to the global properties of the graph, e.g., default node and arc colors, and auto labelling.
11. *Presence of Node Properties Dialog Box (V11)*: describing the presence of the Node Properties Dialog Box. This box allows the change of the properties of the node.
12. *Presence of Arc Properties Dialog Box (V12)*: describing the presence of the Arc Properties Dialog Box. This box allows changes to the properties of the arc.

The Graph abstraction variabilities are:

1. *Graph properties (V14)*: describing the current properties of the graph.
2. *Node properties (V15)*: describing the current properties of the node
3. *Arc properties (V16)*: describing the current properties of the arc

4. *Startup graph properties (V17)*: describing the initial properties of the graph.
5. *Startup node properties (V18)*: describing the initial properties of the node.
6. *Startup arc properties (V19)*: describing the initial properties of the arc.
7. *Graph File format (V20)*: describing the type of file format used for storing the graphs on the disk.

Other variabilities:

1. *Networking (V21):* describing the application's ability to read graphs from URL's.
2. *Additional, user defined options (V22):* describing other user defined options that can be added to the application.

## 4.4 Parameters of Variation

The Parameters of Variation quantify the variabilities, specifying the range of values and the decision time for each. This section presents the parameters of variation for all the variabilities, shown in tabular form. The tables contain the ranges for the parameters, decision time and the default values. In the decision time, GT stands for 'Generation Time', which means that this parameter of variation will be set during the automated code generation. RT stands for 'Run Time', which means that this parameter can be changed during the execution of the application.

| V1 Application Window Properties | | | |
|---|---|---|---|
| **Parameter** | **Values** | **Creation** | **Default** |
| size | <0..MAX_X_DISPL,0..MAX_Y_DISPL> | GT | <600,400> |
| resizable | {TRUE,FALSE} | GT | TRUE |

Table 4.1: Parameters of Variation for V1

| Variablility | Description | Values | Creation | Default |
|---|---|---|---|---|
| **V2** | Menu | {TRUE,FALSE} | GT | FALSE |
| **V3** | Items in the Menu | Additionl item depending on custom functions | GT | n/a |
| **V4** | Toolbar | {FALSE,TRUE} | GT | FALSE |
| **V5** | Buttons on the Toolbar | {FILE, TOOLS, GRAPH, ABOUT, CUSTOM} | GT | {} |
| **V6** | Control Bar | {FALSE,TRUE} | GT | FALSE |
| **V7** | Status Bar | {FALSE,TRUE} | GT | FALSE |
| **V8** | About Box | STRING | GT | STRING |
| **V9** | Title in Caption | STRING | GT | STRING |
| **V10** | Graph Properties Dialog | {FALSE,TRUE} | GT | FALSE |
| **V11** | NodeProperties Dialog | {FALSE,TRUE} | GT | FALSE |
| **V12** | ArcProperties Dialog | {FALSE,TRUE} | GT | FALSE |
| **V20** | Graph file format | {STANDARD, TESTGRAPH, CUSTOM} | GT | STAND ARD |
| **V21** | Networking | {TRUE,FALSE} | GT | TRUE |
| **V22** | User defined option | n/a | GT | n/a |

Table 4.2: Parameters of Variation for Variabilities V2-V22

| **V13** View Area Properties | | | |
|---|---|---|---|
| **Parameter** | **Values** | **Creation** | **Default** |
| color | {black, .. ,white} | GT | white |
| size | <200..3000,200..3000> | GT | <1000,1000> |
| scrollable | {TRUE,FALSE} | GT | FALSE |
| wallpaper | gif image | GT | none |
| wallpaper layout | {CORNER,TILE,CENTER} | GT | CORNER |
| grid | {none,5,10,15,20,25,30,40,50,70,90,110} | GT | none |

Table 4.3: Parameters of Variation for V13

| **V14** Graph Properties | | | |
|---|---|---|---|
| **Parameter** | **Values** | **Creation** | **Default** |
| directed | {TRUE,FALSE} | RT | FALSE |
| auto label for node | {TRUE,FALSE} | RT | FALSE |
| auto label for arc | {TRUE,FALSE} | RT | FALSE |
| mark start node | {TRUE,FALSE} | RT | FALSE |
| default node shape | {Circle,Oval,Square,Rectangle,Cloud} | RT | Circle |
| default node color | {white, .. , black} | RT | blue |
| default node label color | {white, .. , black} | RT | yellow |
| default node size | {10,20,30,40,50,70,80,90,100,110,120} | RT | 30 |
| 3d node | {TRUE,FALSE} | RT | FALSE |

Table 4.4: Parameters of Variation for V14

| **V17** Startup Graph Properties | | | |
|---|---|---|---|
| **Parameter** | **Values** | **Creation** | **Default Value** |
| directed | {TRUE,FALSE} | GT | FALSE |
| auto label for node | {TRUE,FALSE} | GT | FALSE |
| auto label for arc | {TRUE,FALSE} | GT | FALSE |
| mark start node | {TRUE,FALSE} | GT | FALSE |

Table 4.5: Parameters of Variation for V17

| **V15** Node Properties | | | |
|---|---|---|---|
| **Parameter** | **Values** | **Creation** | **Default Value** |
| shape | {Circle,Oval,Square,Rectangle,Cloud} | RT | Circle |
| x | [0 .. ViewAreaWidth] | RT | n/a |
| y | [0 .. ViewAreaHeight] | RT | n/a |
| size | {10,20,30,40,50,70,80,90,100,110,120} | RT | 30 |
| color | {white, .. , black} | RT | blue |
| active | {TRUE,FALSE} | RT | TRUE |
| selected | {TRUE,FALSE} | RT | FALSE |
| label | STRING | RT | EMPTY STRING |
| label size | {8,10,12,15,16,18,20,22,24,32,40,52} | RT | 12 |
| label color | {white, .. , black} | RT | yellow |
| label location | {Top,Center,Bottom} | RT | Center |
| label visible | {TRUE,FALSE} | RT | TRUE |
| shadow | {TRUE,FALSE} | RT | FALSE |

Table 4.6: Parameters of Variation for V15

| **V18** Startup Node Properties | | | |
|---|---|---|---|
| **Parameter** | **Values** | **Creation** | **Default** |
| shape | {Circle,Oval,Square,Rectangle,Cloud} | GT | Circle |
| size | {10,20,30,40,50,70,80,90,100,110,120} | GT | 30 |
| color | {white, .. , black} | GT | blue |

Table 4.7: Parameters of Variation for V18

| **V16** Arc Properties | | | |
|---|---|---|---|
| **Parameter** | **Values** | **Creation** | **Default** |
| source node | n/a | RT | n/a |
| destination node | n/a | RT | n/a |
| color | {white, .. , black} | RT | blue |
| arrow | {TRUE,FALSE} | RT | FALSE |
| active | {TRUE,FALSE} | RT | TRUE |
| selected | {TRUE,FALSE} | RT | FALSE |
| label | STRING | RT | EMPTY STRING |
| label size | {8,10,12,15,16,18,20,22,24,32,40,52} | RT | 12 |
| label color | {white, .. , black} | RT | yellow |
| label location | {Above, Below} | RT | Center |
| label visible | {TRUE,FALSE} | RT | TRUE |

Table 4.8: Parameters of Variation for V16

| **V19** Startup Arc Properties | | | |
|---|---|---|---|
| **Parameter** | **Values** | **Creation** | **Default Value** |
| color | {white, .. , black} | RT | black |

Table 4.9: Parameters of Variation for V19

# Chapter 5

# Application Engineering Environment

This chapter presents the Application Engineering Environment (AEE). In Section 5.1 the design issues are discussed. Sections 5.2, 5.3 and 5.4 describe in detail the Application Engineering Environment which consists of three main components:

- library
- application generator
- framework

## 5.1 Design Issues

Development of the Application Engineering Environment required making a number of design decisions regarding:

- choice of platform and tools,
- language for a code generator,
- graph storage, and
- framework and code generation.

The first issue was the choice of the development platform and tools, and in particular, the language for the generated application source code. Three languages were considered: C++ (Borland or Microsoft), Visual Basic, and Java.

Among these languages Java seemed to be the most suitable. First of all, it is portable, which was quite important since one of the generated applications (TestGraph Editor) was expected to run in the UNIX environment. Besides, Java has a standard GUI library and facilities for the Model-View-Controller paradigm (Observer Pattern), which is the most suitable for the visual graph editor application. Symantec's Café was chosen as the Java development environment.

Another issue was the choice of a language for the application generator. Based on observations from FWS and previous experience with text processing, the author chose Perl. Perl's builtin facilities for pattern matching and regular expressions were ideal for this purposes. Perl is also portable.

One of the first issues in development of the Java Code Base classes was the issue of the graph storage in memory. Based on the previous experience with graph implementations in C++ and utilizing Java's object modeling features, a family of graph classes was developed. The most significant feature of that family is separation of the graph connectivity information from the display information (colors, shapes, etc.). The parent abstract graph class stores the node and arc connectivity data, while the derived form arc and node classes store the display data.

Another issue was the organization of the AEE framework and code generation. Just as in FWS a `codebase` directory with all the component source code was created. Because the GECA code base was substantially larger than FWS code base and for

reasons discussed later in this section, the 'codebase' directory was divided into five subdirectories: `const`, `var`, `select`, `custom`, and `image`.

In terms of code generation, the FWS approach was unsuitable for this work. The FWS is a simple example and the code generation scheme does not scale up well. There the source code for each class was generated from small chunks of Java code. With fifty-five classes in GECA, this approach would be extremely cumbersome. Instead, the author developed a scheme where pairs of meta tags were placed in the code base Java code. During generation, code was inserted or replaced between the tags.

The tags were in the form of Java comments. Because some source Java classes did not require any modifications, the code base was divided into two groups: the code that is modified during generation (`var`) and the code that remains unchanged (`const`).

## 5.2 Library

The library (also called the "codebase") is a set of Java classes divided into four main modules (Figure 5.1) :

- GUI module
- Graph module
- Tools module
- Custom module

There are fifty-five classes in the codebase which makes around 8500 lines of code. The class diagrams presented in the next section use the Rumbaugh OMT diagrams [28].

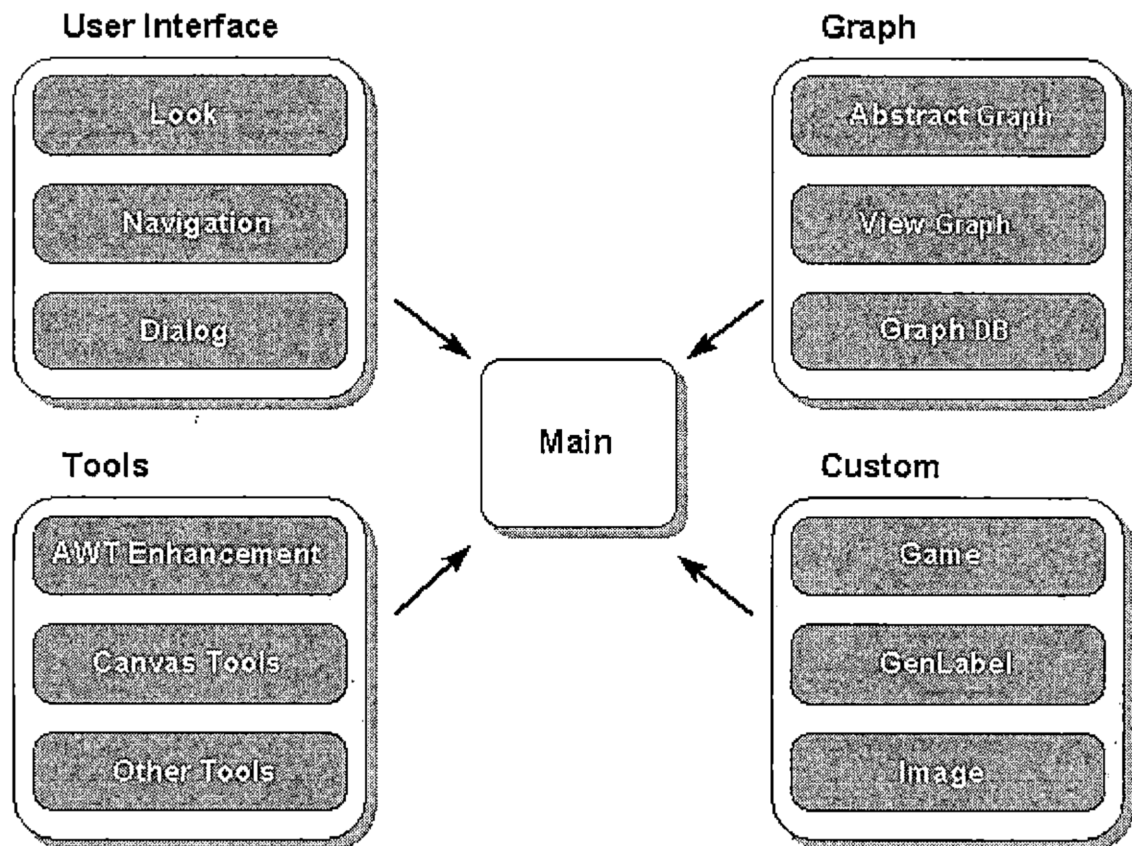



Figure 5.1: Codebase modules

### 5.2.1 GUI Module

The GUI Module is a library of user interface classes. It consists of three submodules:

- Look
- Dialog
- Navigation

**Look submodule**

The Look module contains classes responsible for the look and feel of the GraphEditor application. The Classes `ViewCanvas`, `StatusBar`, and `ControlBar`, all derived

from the AWT Panel class, are the core of that module. These classes implement the panels that are inserted in the application frame.

Since the AWT does not supply a builtin toolbar, a set of toolbar classes was developed. The Toolbar class is derived from the AWT Panel class and it contains a vector of ToolbarItem objects which correspond to the buttons on the toolbar. Each ToobarItem object contains an ImageButton object and a Command object (Figure 5.2). The Command class execute method is invoked when the toolbar button is pressed. The toolbar class family is a standalone, reusable library and can be used in any application.

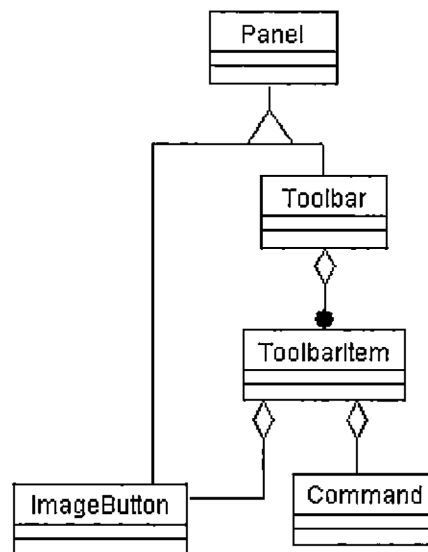



Figure 5.2: Toolbar class family diagram

**Dialog Submodule**

The Dialog submodule contains a set of classes derived from AWT's Dialog class which are responsible for implementing the dialog boxes. Figure 5.3 presents the class diagram for the Dialog family.

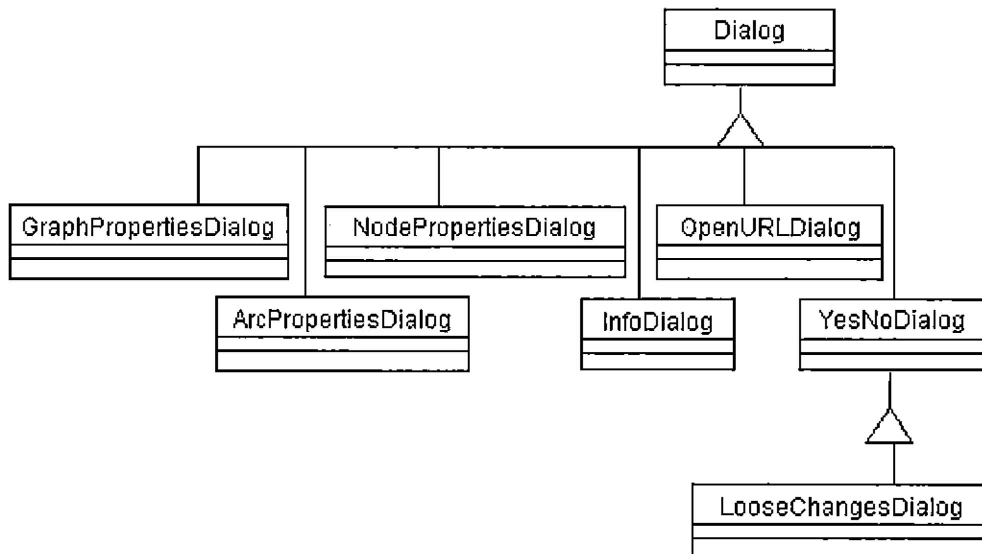



Figure 5.3: Dialog class family diagram

All dialog classes share a similar implementation. Each of them has a similar constructor, event handler and handlers for dialog controls. The constructor of the class creates the dialog controls and initializes them. The dialog event handler overrides the parent AWT dialog box's event handler and calls the handler routines appropriate for the event. Figure 5.4 presents a sample dialog box.

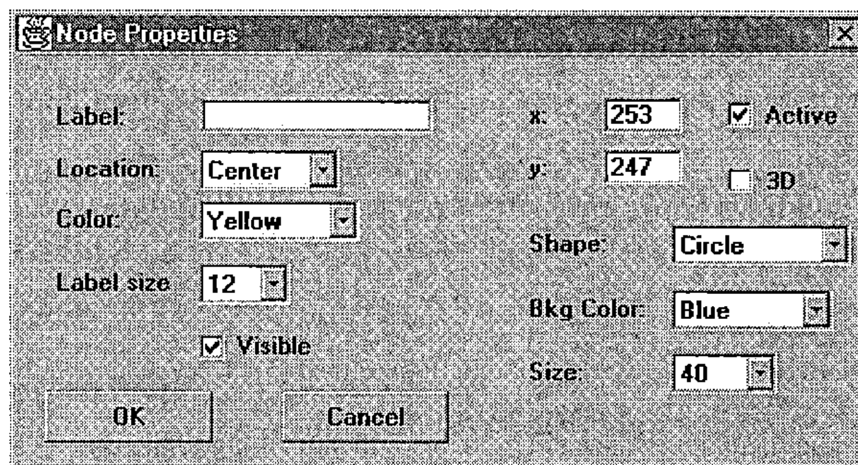



Figure 5.4: Sample dialog box

**Command Submodule**

The Command submodule contains classes that correspond to the items on the menubar and tool bar. All these classes are derived from the abstract parent class `Command`. This class does not implement any code but it defines the `execute()` method that is overridden by the child classes. The `Command` objects are members of `ExMenuItem` and `ToolbarItem` classes. Upon selection of the menu item or toolbar button, the `Command` the object's `execute()` method is called and by virtual binding the proper `execute` method from the appropriate child object is called. This scheme is a convenient way to bind the execution code with the events triggered by the user. The above approach is known as a Command design pattern. Figure 5.5 presents the command class submodule.

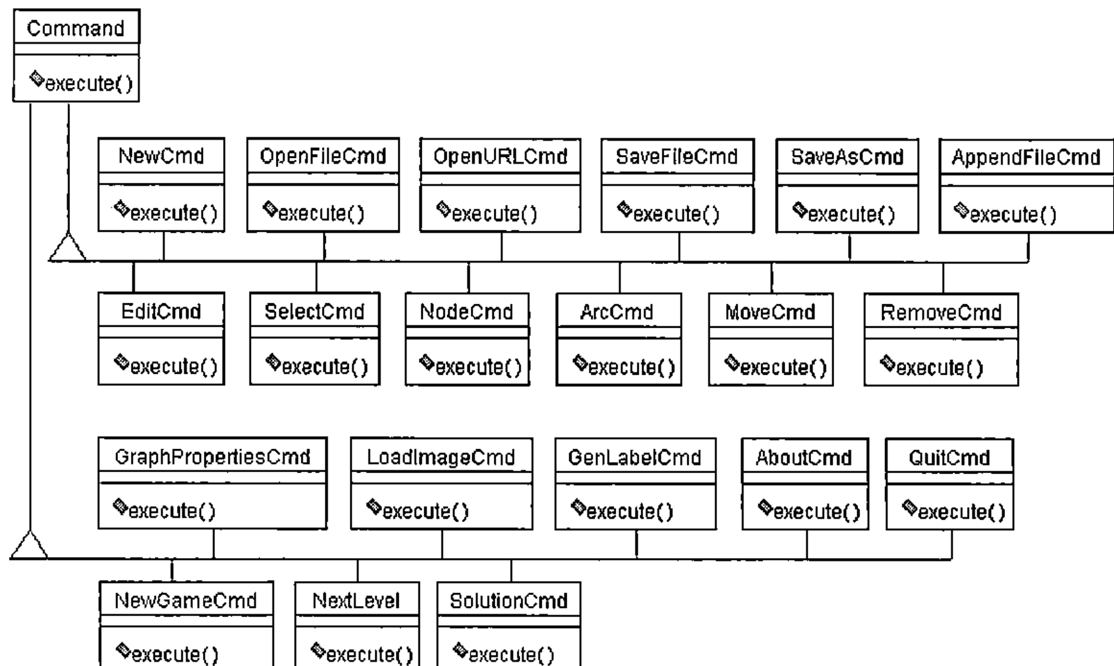



Figure 5.5: Execute class family diagram

### 5.2.2 Graph Module

This module provides means for storing a graph, the main data structure in the GraphEditor application. It consists of three submodules:

- Abstract Graph
- View Graph
- Graph DB

**Abstract Graph**

The Abstract Graph submodule, with classes `Graph`, `GraphExc`, `Node`, and `Arc`, stores the graph connectivity information. Internally the `Graph` class contains two lists (Java `Vector` class objects), the node list and the arc list. The `Arc` class objects contain the source and destination `Node` objects. The Graph class provides methods for adding and deleting arcs and nodes, various iterators and other methods. The `GraphExc` class is used by the `Graph` class to signal exceptions.

**View Graph**

The View Graph submodule stores the display information for the graph: color, shape, label, coordinates and other values that form the runtime variabilities for graph, arc or node. These values are stored in the `ViewGraph`, `ViewArc`, `ViewNode` classes which are derived form the `Graph`, `Arc`, and `Node` classes, respectively. This inheritance scheme introduces a separation between the connectivity and display information which allows the Abstract Graph classes to be easily reused in other applications. If there was a need for storing different information in the graph, the user of the `Graph` class should

derive new classes from the existing Node and Arc classes, retaining the connectivity information and methods.

The additional classes in the View Graph submodule are GraphPlotter and GraphModifier. The first class provides a means for drawing the graph. There are methods for drawing, nodes, arcs, subraphs and entire graphs. The GraphModifier class provides methods for modifying the graph. This class could be considered a link between the ViewGraph and the ViewCanvas class. All user graph editing events handled by ViewCanvas use methods from the GraphModifier class in order to make changes in the ViewGraph.

**GraphDB**

This module provides means for the persistent storage of the graph. It supplies methods for reading and writing the graph. Because Java's DataInputStream class is used, the graph may be read from a file or a URL.

The following graph file format is used:

```
NUMBER OF NODES
NUMBER OF ARCS
NODE LIST
ARC LIST
```

This format is one of the variabilities. In the case of the TestGraph editor application, it will be slightly different. This is a sample of a typical graph file:

```
Nodes: 5
Arcs: 4

0 376 107 70 White Cloud NoShadow Active "Shape" 12 Black Center
LabelVisible
1 190 168 70 White Cloud NoShadow Active "Circle" 12 Black Center
LabelVisible
2 308 236 70 White Cloud NoShadow Active "Rectangle" 12 Black Center
LabelVisible
3 470 243 70 White Cloud NoShadow Active "Oval" 12 Black Center
LabelVisible
```

```
4 553 143 70 White Cloud NoShadow Active "Square" 12 Black Center
LabelVisible
1 0 Arrow Black Active "" 12 Black Above LabelVisible
2 0 Arrow Black Active "" 12 Black Above LabelVisible
3 0 Arrow Black Active "" 12 Black Above LabelVisible
4 0 Arrow Black Active "" 12 Black Above LabelVisible
```

The entire graph family hierarchy class diagram is shown on Figure 5.6. The figure also shows that the `Graph` class is derived from Java's `Observable` class. This fact will be discussed later and was purposely omitted in Section 5.2.2 for simplicity.

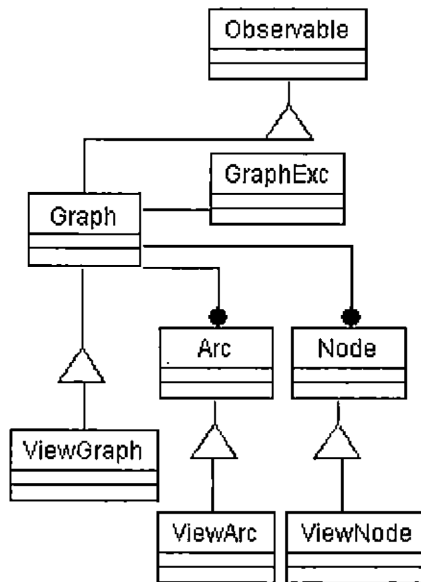



Figure 5.6: Graph family class diagram

### 5.2.3 Tools Module

The Tools module consists of three submodules:

- AWT Enhancements
- Canvas Tools
- Other Tools

**AWT Enhancements**

The AWT Enhancements submodule consists of a few independent classes that implement some extensions to Java's AWT classes. `Wallpaper` provides a means for easily drawing images (patterns) on any object derived from `Component`. Class `MultilineLabel` implements a label with multiple lines, classes `Ctrl3D` and `Border` give a 3D look to any object derived from `Component`, class `ExMenuItem` derived from `MenuItem` extends its features by adding the `Command` class capabilities.

**Canvas Tools**

The Canvas Tools consists of two classes: `CrossTool` and `GridTool` classes that provide drawing tools used for alignment. `GridTool` is an adjustable grid on the `ViewCanvas` and `CrossTool` is a rubberbanded guide used for aligning.

**Other Tools**

This module consists of the static `Utility` class with various methods that did not fit into any other module, and a static `Cloud` class that provides means for drawing the cloud shape nodes.

### 5.2.4 Custom

The Custom module consists of three submodules:

- Game
- GenLabel
- Image

This module contains the custom code used in application generation: the class files that differ from one family member to the next. The Game submodule with classes:

NewGame, NextLevel, Solution provides the code for the graph game. The GenLabel submodule with GenLabelCmd class provides the code used in TestGraph Editor for generating the C++ label files. The Image submodule with its LoadImageCmd class provides the code for loading the image into the graph editor

that is used by GraphImage Editor.

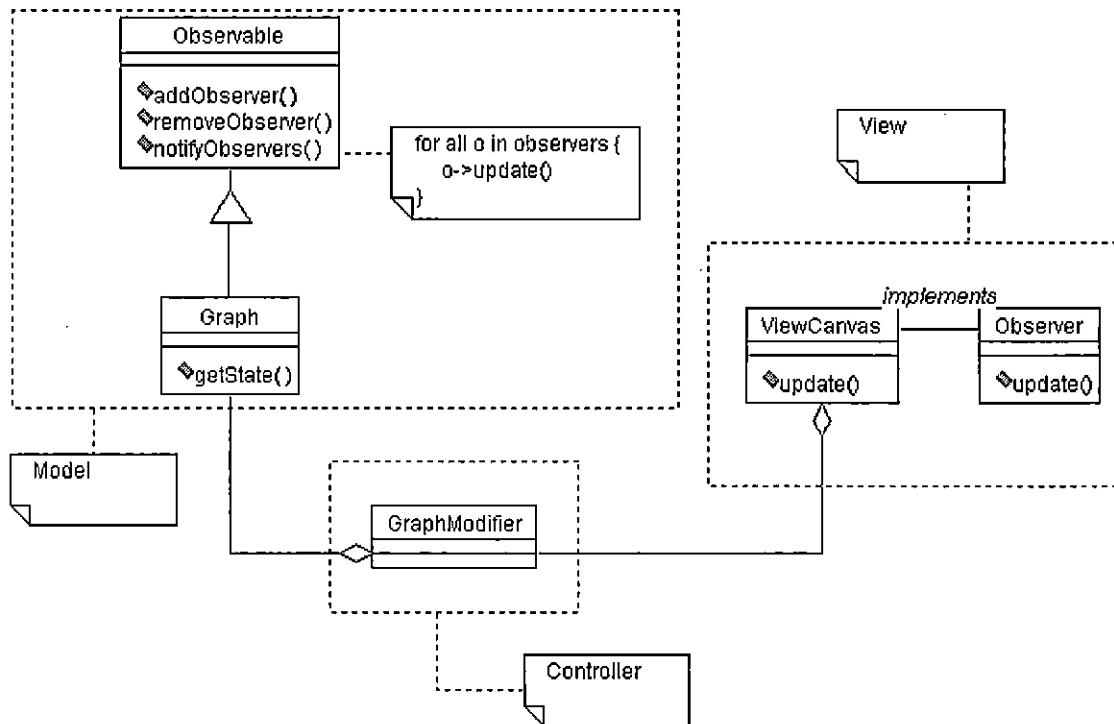



Figure 5.7: Model-View-Controller paradigm in the Graph Editor

### 5.2.5 Model-View-Controller classes

The Graph Editor application utilizes the Model-View-Controller paradigm. This paradigm can be implemented using the Observer Pattern. Java supports this pattern in its Util package. This package includes, among others, the Observable class and the Observer interface.

The `Graph` class is the model (concrete subject); it inherits from the Observable class. By this relationship the Graph class is able to register and notify other classes that implement the `Observer` interface (concrete observers). In the current version of GraphEditor there is only one observer: the `ViewCanvas` class. This class displays the graph (concrete subject) on an associated canvas. Any changes in the `Graph` object are reflected in the `ViewCanvas` object when its update method is called. Changes in the graph and updates are triggered by the GraphModifier object which plays the role of the controller. Figure 5.7 presents the Model-View-Controller paradigm in the Graph Editor Application.

### 5.2.6 Application class diagram

Figure 5.8 presents the GraphEditor application class diagram, except for the Dialog class family and Command class family. Those families are shown on Figures 5.3 and 5.5.

## 5.3 Framework

The AEE framework is portable and is almost identical on both the UNIX and PC platforms. Physically the environment is placed in the `aee` directory. There are no constraints upon the name of that directory. At the root level of that directory is the application generator, `appgen.pl`, and several of `bat` or `csh` scripts, depending on the platform. Besides that there is the `codebase` directory which contains the AEE library and graphic image files. There are five subdirectories in the `codebase` directory:

- `var`
- `const`

- `select`
- `custom`
- `image`

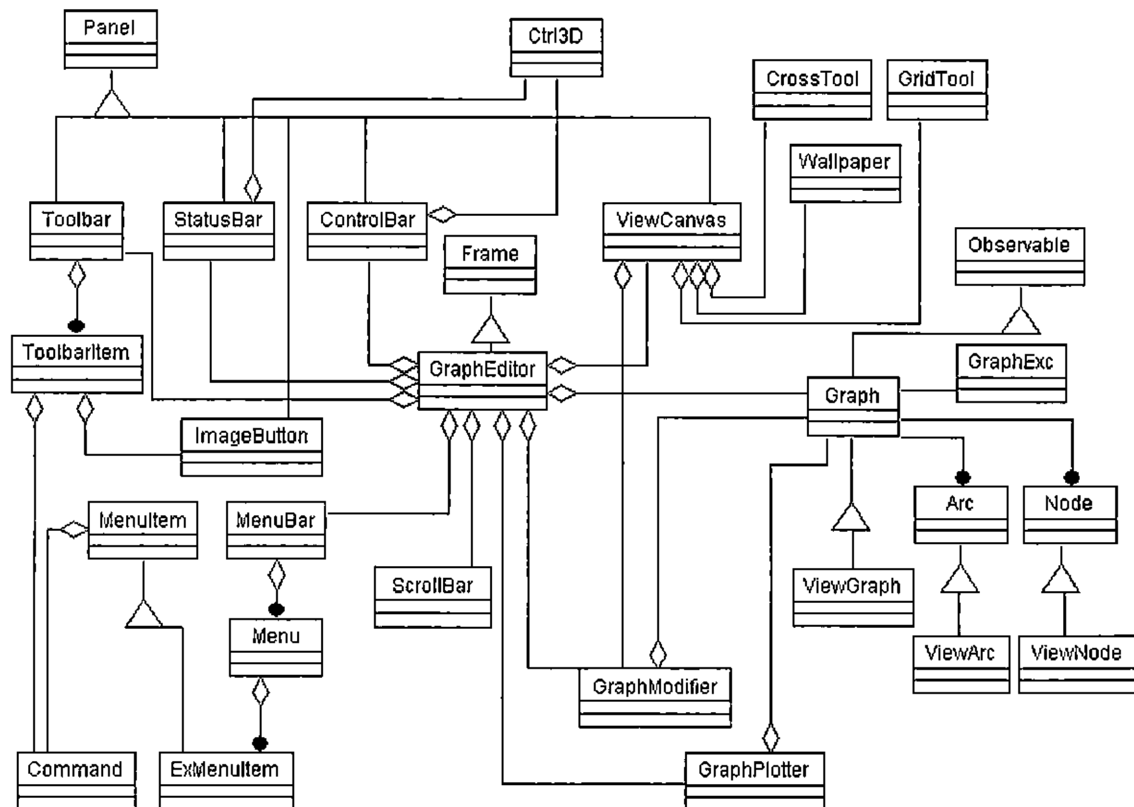



Figure 5.8: Graph Editor class diagram

The `var` directory stores the Java class files that are modified during the code generation, and the `const` directory stores the files that remain unchanged. The select directory contains the classes that replace entirely the corresponding class in the `var` directory. In the current version of AEE there is only one file in that directory: `FileHandler.tg` which replaces the `FileHandler.java` when the TestGraph Editor is generated. The `custom` directory contains user-defined classes. In the current version of AEE the

custom classes are: `GenLabelCmd`, `LoadImageCmd`, `NewGameCmd`, `NextLevelCmd`, and `SolutionCmd`. The `image` directory contains the image files. There are toolbar button and background images.

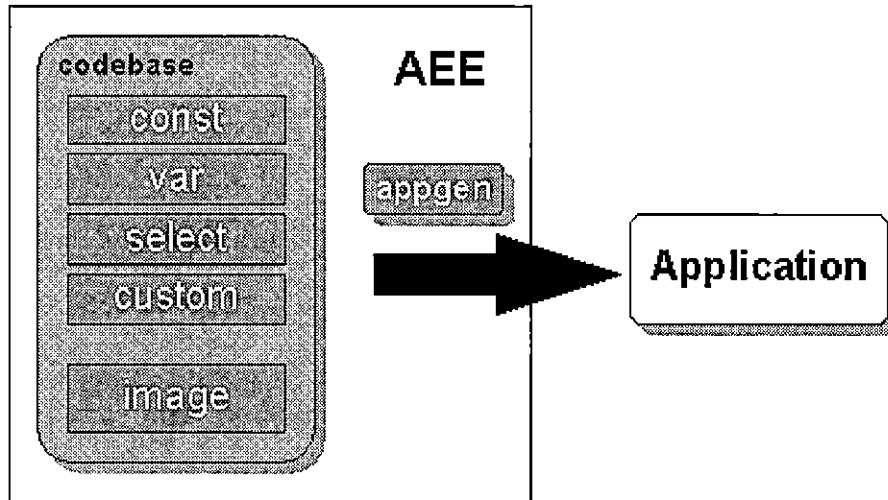



Figure 5.9: Application Engineering Environment framework

# 5.4 Application Generation

The goal of the Commonality Analysis process was to prepare for automated code generation. One of the most important parts of the AEE is the code generator. The code generator reads the input model file and creates the application Java source code. This section describes the generation process in more detail.

## 5.4.1 Model File

The application model is specified in the model file, written in a DSDL (Domain Specific Design Language). Such a language is intended to formally specify a software design and in case of CA a particular family member. The model describes the Graph Editor

application: its look, behavior and functionality. The DSDL introduced here is a simple declarative language with seventeen keywords. Each of the keywords has different parameters and options. The syntax of the language is similar to POD (Plain Old Documentation) used for documenting PERL environments and code. Each of the keywords starts with '=', options start with '-' sign, comments are placed after '#'. The keywords, parameters, and options are related to the variablities and parameters of variation. Table 5.1 shows all the keywords and parameters.

| Keyword | Options |
|---|---|
| `=about` | `n/a` |
| `=arc` | `-color` |
| `=build` | `n/a` |
| `=button` | `-image -class` |
| `=controlbar` | `n/a` |
| `=fileformat` | `-testgraph -custom` |
| `=graph` | `-directed -autolabelnode -startnode -` |
| `=menu` | `-file -tools -graph` |
| `=networking` | `n/a` |
| `=node` | `-shape -color -size` |
| `=project` | `-dir - author` |
| `=run` | `n/a` |
| `=set` | `n/a` |
| `=statusbar` | `n/a` |
| `=toolbar` | `-files -properties -tools` |
| `=viewarea` | `-wallpaper -layout - size - color -` |
| `=window` | `-size` |

Table 5.1: DSDL keywords

The DSDL keywords can be grouped in a three groups. First group are the keywords that control the creation and execution of the application. These are: `=project`, which specifies the name of the project and output directory, `=build` which causes the

invocation of the Java compiler, =run which executes the generated and compiled application and =set which is used to define macros.. The second group are the GUI keywords: =about, =button, =controlbar, =statusbar, =toolbar, =viewarea, and =window. The third group are the graph keywords, which describe the node, arc and graph properties: =graph, =node, =arc, =fileformat. There is also the =networking keyword which is used to enable networking capabilities in the generated application. The details about the DSDL semantics can be found in Appendix D.

A sample DSDL file is shown below. It describes a graph editor application which source code will be generated in myoutput directory. The application window will be initially 600x400 pixels and the caption bar will say: Graph Editor. The application will have a statusbar, menubar (with file, tools, and about submenus) and a toolbar (with file and tools related buttons). The view area will be scrollable, 1000x1000 pixels, white without grid. The graph will be directed, with auto labeling of arc and nodes. The start node will be marked. The nodes will be blue ovals, 50 pixels in size, arcs will be red. After generation the application will be compiled and run.

```
##############################################################
#
# Sample model file
#
#############################################################

=set PROJECT Graph Editor # project name
=set DIR myoutput          # output directory
=set AUTHOR Greg Kacy     # application engineer

=project "$PROJECT" -dir $DIR -author "$AUTHOR"

=about $PROJECT by $AUTHOR (C) 1997 University of Victoria

=networking
```

```
=window -size 600 400

=statusbar

=viewarea -size 1000 -color white -scrollable -grid none

=menu -file
=menu -tools
=menu -about

=toolbar -files -tools

=graph -directed -autolabelnode -startnode -autolabelarc
=node -shape OVAL -color blue -size 50
=arc -color red
=build
=run
```

## 5.4.2 Generator (appgen.pl)

The application generator is implemented in Perl. It reads the model file and generates the target Graph Editor application with a two-phase process. In the first phase the input model file is scanned and a symbol table with variables and their values is built. In the second phase the variables are expanded and the code is generated.

During the generation process the target directory is created and the Java codebase classes are copied into that directory. This happens when the keyword `=project` is interpreted; therefore that keyword must always present in the model file and precede all the keywords except `=set`. Later, as the lines from the model file are interpreted, the appropriate files are modified in the target directory. These are the files that have been copied from the `codebase/var` directory.

The Java source files are modified using the meta-tags scheme. For every variability there is an associated tag pair that is placed in the source file. The tags are placed in the comments allowing independent compilation of the source files and

identification of syntax errors before the generation process begins. Table 5.2 presents all the used tags and their corresponding variabilities.

When a particular keyword is interpreted the code inside the tag is replaced by the generated code. Figure 5.10 shows this process.

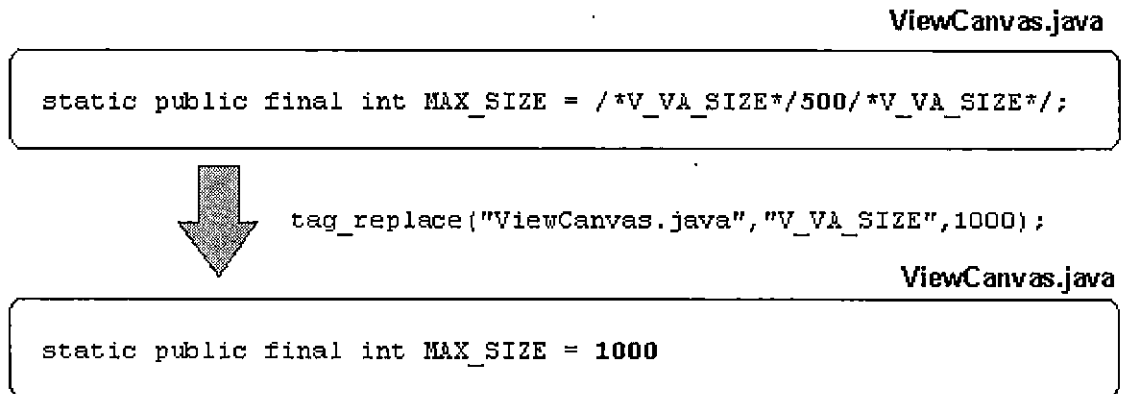



Figure 5.10: Meta-tag replacement scheme during the code generation

The following is the example how the toolbar custom button code is generated. This code fragment comes from the `GraphEditor` class constructor:

```
.
setLayout(new BorderLayout());

// init toolbar and control bar
Panel top = new Panel();
top.setLayout(new BorderLayout());

/*V_TOOLBAR*/ //toolbar  /*V_TOOLBAR*/
/*V_TOOLBAR_I1*/ //item /*V_TOOLBAR_I1*/
.
.
```

When the `=button` keywords are interpreted, the `tag_replace` routine is called with appropriate replacement code depending on parameters and options of the `=button` keyword.

```
################################################################
# Handles line with =button
#
sub k_button
{
	local($line) = @_;
	local($tail,$image,$class,$code);

	$line =~ m/=button\s+(.*)/;
	$tail = $1;

	if ($tail =~ m/-image\s+"(.*)"/) {
		$image = $1;
	}
	if ($tail =~ m/-class\s+(\w*)/) {
		$class = $1;
		install_custom_file($class);
	}
	$gCustomButtonCount++;
	$code=$code."tool_bar.addButton(\"$image\", new $class(this));\n";
	tag_replace("$gOutputDir/GraphEditor.java",
	"V_TOOLBAR_I$gCustomButonCount",$code);
}
```

The resulting generated code for the above will be:

```
tool_bar.addButton("image/b_c_gen:gif", new GenLabelCmd(this));
```

The details for how the code is generated for the rest of the keywords can be found in Appendix C.

| Tag | Variability |
|---|---|
| V_ABOUT | V8 |
| V_A_COLOR | V19 |
| V_CONTROLBAR | V6 |
| V_FIXEDSIZE | V1 |
| V_GD_AUTOLABELARC | V16 |
| V_GD_AUTOLABELNODE | V16 |
| V_GD_DIRECTED | V16 |
| V_GD_STARTNODE | V16 |
| V_GD_UPDATECODE | V16 |
| V_HEIGHT | V1 |
| V_MENUBAR | V3 |
| V_NETWORKING | V21 |
| V_N_COLOR | V18 |
| V_N_SHAPE | V18 |
| V_N_SIZE | V18 |
| V_STATUSBAR | V7 |

| V_TOOLBAR | V5 |
|---|---|
| V_VA_COLOR | V13 |
| V_VA_GRID | V13 |
| V_VA_SCROLLABLE | V13 |
| V_VA_SIZE | V13 |
| V_VA_WALLPAPER | V13 |
| V_VA_WALLPAPER_IMAGE | V13 |
| V_VA_WALLPAPER_LAYOUT | V13 |
| V_WIDTH | V1 |

Table 5.2: Meta-tags and Variabilities

# Chapter 6

# Application Engineering

This chapter describes the four members of the Graph Editors family that have been generated by the AEE. The chapter will focus on their functionality and generation.

## 6.1 Testgraph Editor

The Testgraph Editor (TGE) is a graph editor used for editing testgraphs. Testgraphs are directed graphs used in the ClassBench environment (Framework for Automated C++ Collection Class Testing [30]). A testgraph is a graph representing an abstraction of the state-transition graph of the Class Under Test (CUT). The testgraph's nodes represent states the CUT assumes during the testing process. Arcs between nodes depict transitions between these CUT states. Rooted paths in the graph represent a sequence of CUT testgraph states and transitions, starting with the state represented by the start node [30].

The TGE displays testgraphs in a graphic form. Each node and arc has a label. The start node is drawn in different color. At any given time each node and arc can be active or inactive. The testgraph stored on disk is loaded by the ClassBench's test driver

which traverses rooted paths in the testgraph, and based on that, performs testing of the CUT.

The TGE provides a means for building and editing testgraphs, storing them on disk and also generating C++ source files also used by ClassBench. Figure 6.1 shows a TGE application screen snapshot with the testgraph for the `IntSet` class taken from the UNIX version of TGE.

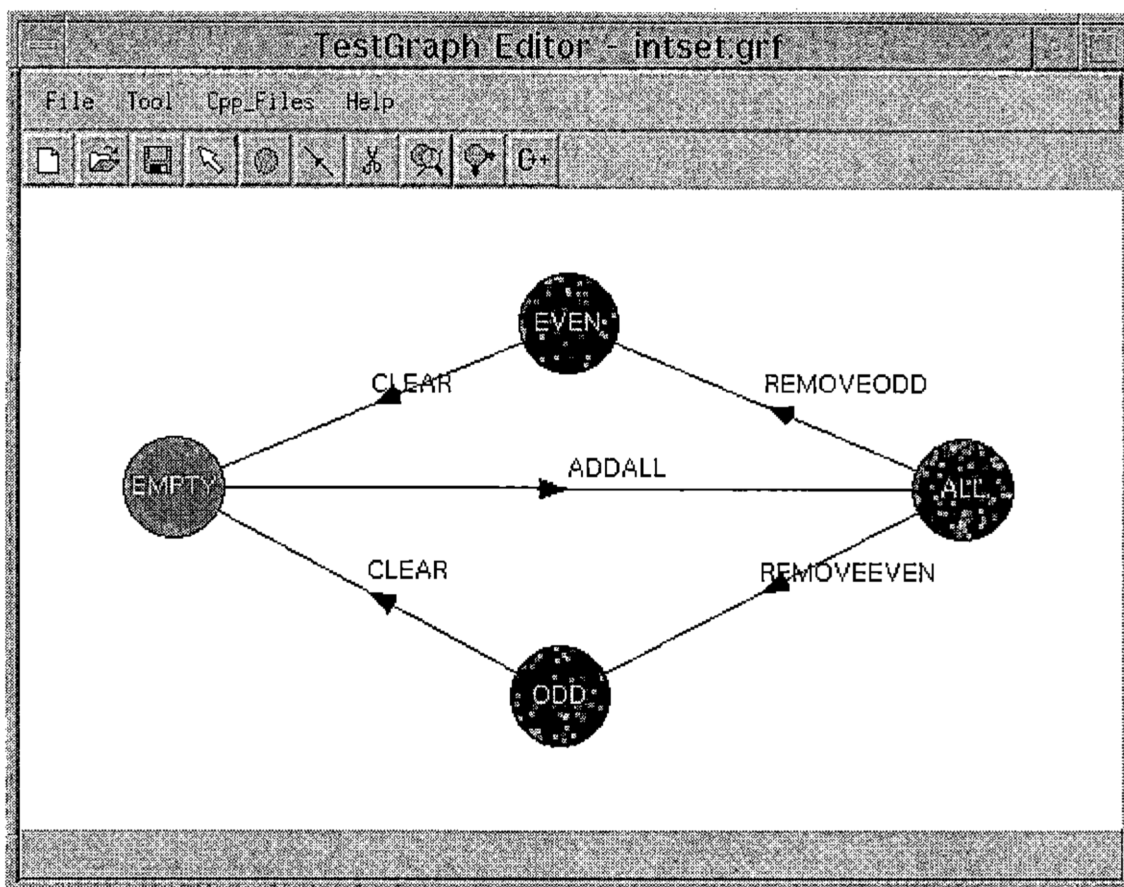



Figure 6.1: Testgraph Editor with the IntSet class testgraph

Editing of the graph involves creation, deletion, moving and setting properties of arcs and nodes. TGE has built-in tools for all of the above tasks. Moving of nodes and arcs and

drawing arcs is rubberbanded. TGE has also built-in file manipulation options: Save, Save As, New, Open, Open from URL. Node and arc properties are set in dialog boxes.

TGE was entirely generated automatically by the AEE. Its model file, `tge.dsl` is presented below. The line numbers are added to simplify explanation.

```
1  ############################################################
2  ## tge.dsl
3  ## PURPOSE: AEE Model file for the Test Graph Editor
4  ## by Greg Kacy, Computer Science, University of Victoria
5  ############################################################
6
7  =set PROJECT TestGraph Editor # project name
8  =set DIR tge                  # output directory
9  =set AUTHOR Greg Kacy         # application engineer
10
11 =project "$PROJECT" -dir $DIR -author "$AUTHOR" # Proj. init.
12
13 =about $PROJECT by $AUTHOR (C) 1997 University of Victoria #About Box
14
15 =networking # Enable URL reading
16
17 =window -size 600 400 # Application window properties
18
19 =statusbar # Statusbar present
20
21 # View Area is 1000 x 1000 pixels.
22 # There is no grid upon init and the color of View Area is white
23 =viewarea -size 1000 -color white -grid none
24
25 # Menubar with standard File, Tools Selection and About
26 # menus present. Custom menu option -generating the C++ label files
27 =menu -file
28 =menu -tools
29 =menu -Cpp_Files | Generate -class GenLabelCmd
30 =menu -about
31
32 # Toolbar with the standard File and Tools buttons present
33 =toolbar -files -tools
34
35 # Custom button for generating the C++ label files
36 =button -image "image/b_c_gen.gif" -class GenLabelCmd
37
38 # Default graph, node and arc properties
39 =graph -directed -autolabelnode -startnode -autolabelarc -update
40 =node -shape CIRCLE -color blue -size 50
41 =arc -color black
42
43 =fileformat -testgraph # ClassBench file format
44
45 # Build application and execute
46 =build
47 =run
```

In lines 7, 8, and 9, three variables defining the name of the project, output directory and the application engineer are created. Line 11 defines the project. The Testgraph Editor is created in the `tge` directory. In line 13 the message in the About box is initialized. Line 15 enables networking capabilities. The graphs can be read from the URL. In lines 17-36 the application's look and feel is defined. The main application window is in line 17, statusbar in 19 and the view area in line 23. The view area is 1000x1000 pixels, it is white and has no grid. Lines 27-30 define the menu. TGE has a file menu, tools menu, one custom menu and the About menu. Lines 33-36 define the toolbar. There are file and tools buttons and a custom button for C++ file generation. Lines 39-41 define the graph, node and arc properties. The graph is directed, the start node is marked, and the nodes and arcs are autolabeled when created. In line 43 the file format for the testgraph is set. TGE file format must be compatible with the ClassBench testgraph format. In line 46 the application is built and in line 47 it is executed.

## 6.2 Graph Draw

Graph Draw is a graph drawing application used for creating graph images. Because it provides the user with a variety of node and arc shapes, colors and sizes, it can be used to create graph diagrams, hierarchical diagrams, Booch class diagrams, etc.

Just like TGE, Graph Draw contains graph creation and editing tools, file options, and node and arc property dialog boxes. In addition Graph Draw provides the user with a variable grid and rubberbanding guide.

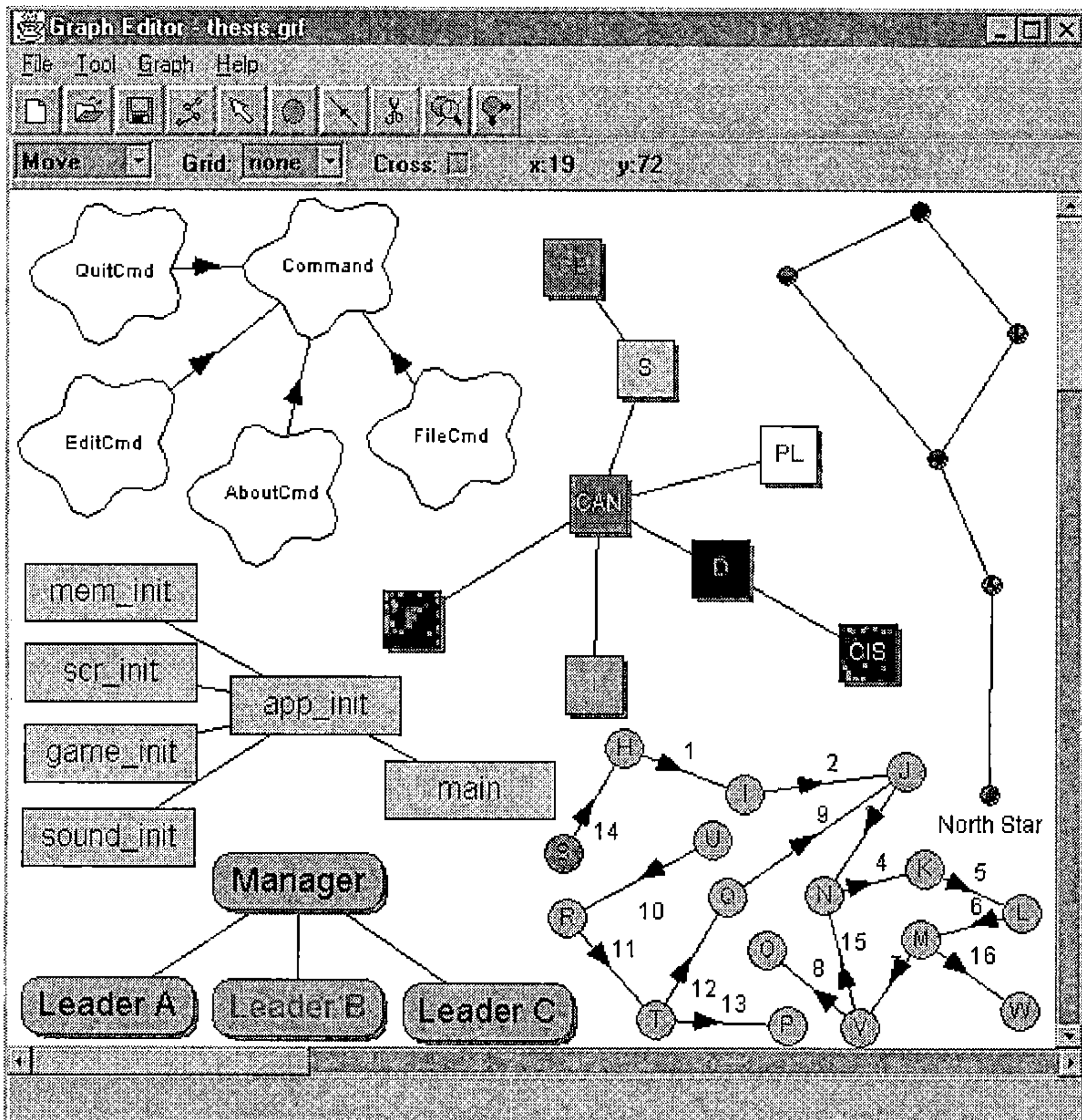



Figure 6.2: Graph Draw with sample graphs

It is also possible to change the global properties of the graph in the Graph Properties dialog box. The drawing canvas is scrollable. Graph Draw also allows graphs to be appended from a file or URL to the current graph. Figure 6.2 presents Graph Draw with sample graphs. The model file (`gdraw.dsl`) used to automate generation of the application is shown below:

```
1  ##########################################################
2  ## gdraw.dsl
3  ## PURPOSE: AEE Model file for the Graph Draw
4  ## by Greg Kacy, Computer Science, University of Victoria
5  ##########################################################
6
7  =set PROJECT Graph Draw   # project name
8  =set DIR gdraw            # output directory
9  =set AUTHOR Greg Kacy     # application engineer
10
11 =project "$PROJECT" -dir $DIR -author "$AUTHOR" # Proj. init.
12
13 =about $PROJECT by $AUTHOR (C) 1997 University of Victoria #About Box
14
15 =networking # Enable URL reading
16
17 =window -size 700 500 # Application window properties
18 =statusbar # Statusbar present
19 =controlbar # Controlbar present
20
21 # View Area is 2000 x 2000 pixels. It is scrollable
22 # There is no grid upon init and the color of View Area is white
23 =viewarea -size 2000 -color white -scrollable -grid none
24
25 # Menubar with File,Tools,Graph Properties About menus
26 =menu -file
27 =menu -tools
28 =menu -graph
29 =menu -about
30
31 # Toolbar with the File,Graph Properties,Tools buttons
32 =toolbar -files -properties -tools
33
34 # Default node and arc properties
35 =node -shape CIRCLE -color blue -size 40
36 =arc -color black
37
38 =build # Build application and execute
39 =run
```

The model file is similar to the one for TGE with differences in lines: 7, 8, 17, 19, 23, 28, 32, 35, and 36. The window size and the view area are different. The view area in Graph Draw is scrollable and it has a grid. Also there is the control bar below the toolbar, and more buttons on the toolbar. Unlike in TGE, it is possible to change the global graph properties in Graph Draw. Also the file format is different (default) and the initial properties for graph, node and arcs are different too.

# 6.3 Mathmania Game

Mathmania is a set of games and activities that teach elementary school students the concepts of advanced mathematics [39]. One of the Mathamania games is based on the minimal dominating set concept. In this game the task of the player is to find and select a minimal number of nodes so that every unselected node has one or more selected neighbour node.

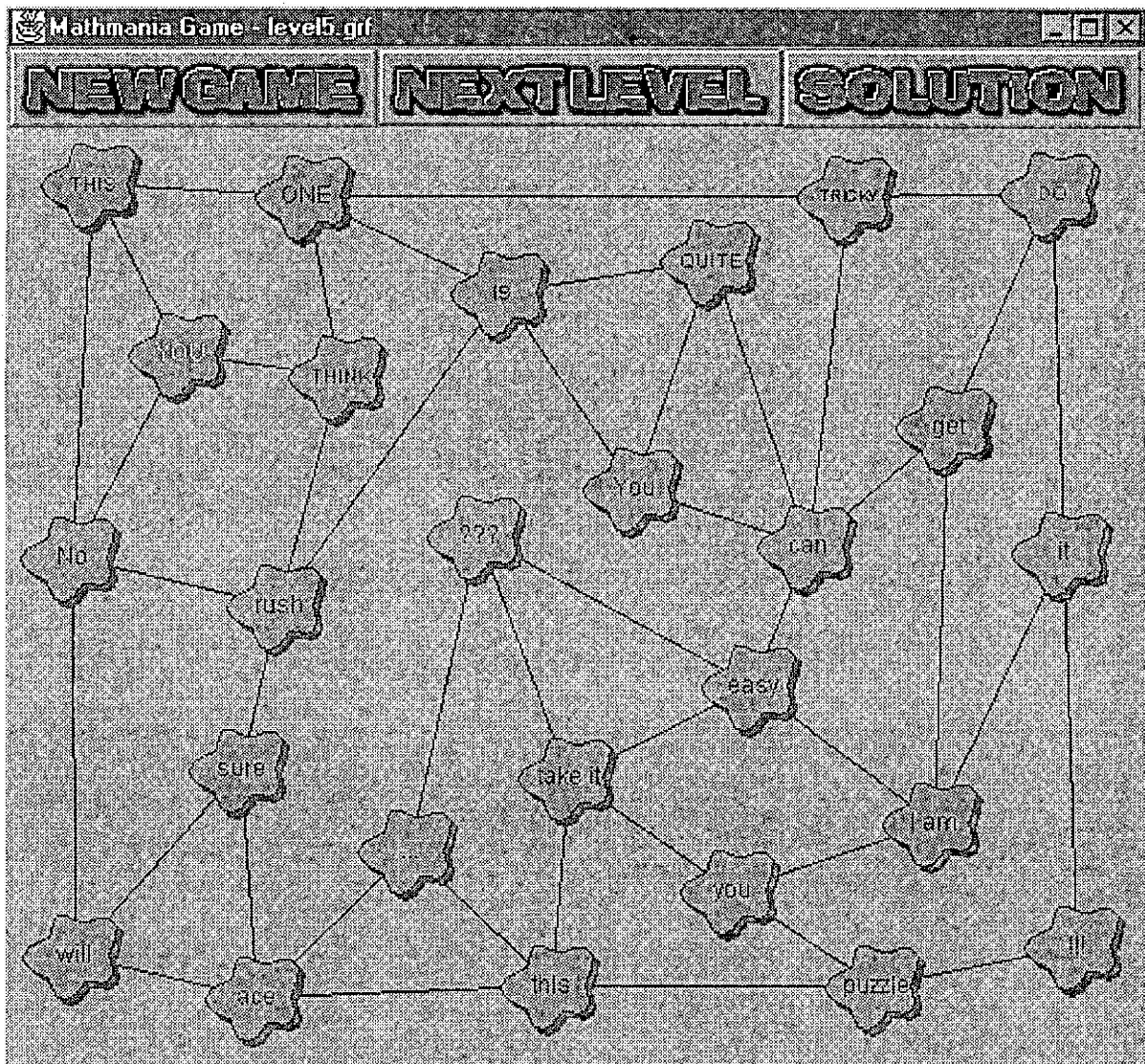



Figure 6.2: Mathmania game

In Mathmania, this game is called Tourist Town and it is one of the generated applications in AEE. Figure 6.3 presents the game's screen snapshot. There are five levels in the game; the higher the level number, the more complex the graph. A player can always see the solution by pressing the solution button. It is possible to create more levels to the game using Graph Draw. This is the model file for the game (`game.dsl`).

```
1  ##########################################################
2  ## game.dsl
3  ## PURPOSE: AEE Model file for the Mathmania Game
4  ## by Greg Kacy, Computer Science, University of Victoria
5  ##########################################################
6
7  =set PROJECT Mathmania Game # project name
8  =set DIR game               # output directory
9  =set AUTHOR Greg Kacy       # application engineer
10
11 =project "$PROJECT" -dir $DIR -author "$AUTHOR" # Proj. init.
12
13 # Text in the About Box
14 =about $PROJECT by $AUTHOR (C) 1997 University of Victoria
15
16 # Application window properties. The window has fixed size
17 =window -size 570 510 -fixed
18
19 # View area has a gif image tiled in the background
20 =viewarea -wallpaper "image/wallpaper.gif" -layout TILE
21
22 # Toolabr with three custom buttons present
23 =toolbar
24 =button -image "image/g_bt1.gif" -class NewGameCmd   # custom option
25 =button -image "image/g_bt2.gif" -class NextLevelCmd # custom option
26 =button -image "image/g_bt3.gif" -class SolutionCmd  # custom option
27
28 # Build application and execute
29 =build
30 =run
```

The main difference between the Mathmania Game's model file and the previous two files is that the application window is fixed in size, and there is no menu , statusbar or

controlbar. The view area has an image tiled in the background and there are three custom buttons on the toolbar.

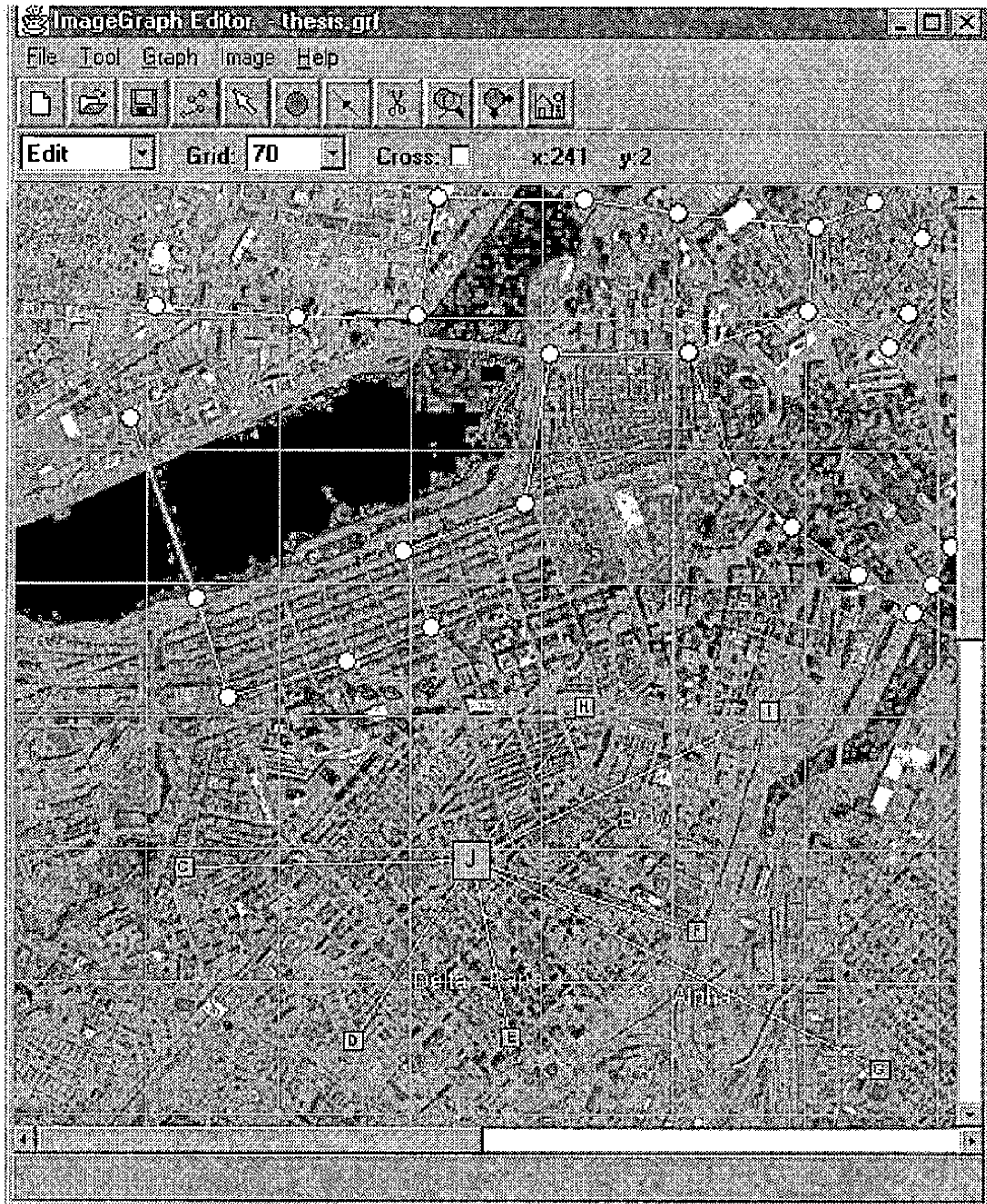



Figure 6.4: Image Overlay application

# 6.4 Image Overlay

Image Overlay is an application allowing graphs to be drawn on top of images. It can be used as simple GIS (Geographic Information System) to map layers of networks on top of a map or an aerial photograph, or to map distinctive lines and points on top of images (for example an image of face). Figure 6.4 presents the screen snapshot for the Image Overlay application.

The application includes all the features of Graph Draw with additional image retrieval and different default settings. The model file (`gis.dsl`) for Image Overlay is presented below.

```
1  ############################################################
2  ## gis.dsl
3  ## PURPOSE: AEE Model file for the Image Overlay
4  ## by Greg Kacy, Computer Science, University of Victoria
5  ############################################################
6
7  =set PROJECT ImageGraph Editor # project name
8  =set DIR gis                   # output directory
9  =set AUTHOR Greg Kacy          # application engineer
10
11 =project "$PROJECT" -dir $DIR -author "$AUTHOR" # Proj. init.
12
13 =about $PROJECT by $AUTHOR (C) 1997 University of Victoria #About Box
14 =networking # Enable URL reading
15
16 =window -size 700 500 # Application window properties
17 =statusbar # Statusbar present
18 =controlbar # Controlbar present
19
20 #View Area is 1000 x 1000 pixels. It is scrollable
21 #There is 70 pixels grid upon init and the color of View Area is blue
22 =viewarea -size 1000 -color blue -scrollable -grid 70
23
24 # Menubar with File, Tools, Graph Properties and About
25 # menus present. Custom menu option for loading the gif image
26 =menu -file
27 =menu -tools
28 =menu -graph
29 =menu -Image | Load -class LoadImageCmd
30 =menu -about
31
32 # Toolbar with the standard File, Graph Properties and Tools
33 # buttons present
34 =toolbar -files -properties -tools
```

```
35 # Custom button for loading the gif image
36 =button -image "image/b_image.gif" -class LoadImageCmd
37
37 ## Default node and arc properties
38 =node -shape CIRCLE -color black -size 10
39 =arc -color red
40
41 =build # Build application and execute
42 =run
```

The Image Overlay application's mode file is quite similar to the Graph Draw with the exception that a custom menu and toolbar option for loading images has been added. Also the graph properties and the view area properties are different.

# Chapter 7

# Conclusion and Future Work

## 7.1 Summary and Conclusion

Today's software industry with large software systems, multiple versions, tight release deadlines and competitive market puts a lot of pressure on developers. This is why many organizations today are investigating product line architectures. Such architectures would allow them to deliver new products faster, improve quality, reengineer legacy systems and manage many product variations.

A strategy for deciding about family members, known as commonality analysis, is in use at Lucent Technologies. In this strategy a family is defined by identifying commonalities, i.e., assumptions that are true for all family members, variabilities, i.e., assumptions about what can vary among family members, and common terminology. This strategy is a part of a domain engineering process known as FAST (Family-Oriented Abstraction, Specification, And Translation). The goal of the FAST process is to develop facilities for rapidly generating members of a family. Performing a commonality analysis is an early step in this process [1].

In this thesis Lucent's Commonality Analysis method was applied to the Graph Editor application family and, based on its results, an Application Engineering Environment was created. The creation of the AEE is the original and the most significant part of the thesis. Using the AEE, four different family members were generated. One of the generated applications, the TestGraph Editor will be used in the ClassBench [31] C++ Collection Class Testing Environment. The AEE consists of the framework, codebase and application generator. The framework provides the skeleton for the environment, directory structure, and rules for creating the family members. The codebase is a set of Java source files from which the target application is built. There are 55 classes in total with 8586 lines of code (See Table 6.1 for details). The application generator automatically generates family member applications based on their model files. The model files are written in a Domain Specific Description Language. The language is a simple declarative DSDL with abilities to create macros.

| Module | No. of Classes | LOC |
|---|---|---|
| Main | 1 | 365 |
| Look | 6 | 1424 |
| Navigation | 16 | 816 |
| Dialog | 8 | 1635 |
| Abstract Graph | 4 | 403 |
| ViewGraph | 5 | 1737 |
| Graph DB | 1 | 423 |
| AWT Enhancements | 5 | 910 |
| Canvas Tools | 2 | 177 |
| Other Tools | 2 | 291 |
| Game | 3 | 224 |
| GenLabel | 1 | 57 |
| Image | 1 | 71 |
| | | **Total Java LOC:** 8589 |
| appgen.pl | | 849 |
| | | **Total LOC: 9438** |

Table 6.1: AEE files summary

The work presented here is a non-trivial example of the *compositional approach* to Commonality Analysis. The only other example of the compositional approach - Floating Weather Station [19][1], with 5 classes and 330 lines of code, is much simpler than this Graph Editor. The major work in this thesis is in the design and development of the AEE. An improved scheme for implementing variabilities and parameters of variation during the generation process has been introduced in the application generator. This scheme is more flexible than the existing one (FWS). One of the main advantages is that it allows independent compilation of the codebase source files and identification of syntax errors before the generation process begins. Also it is much easier to maintain. The FWS code fragments are hard to understand. In the codebase a reusable GUI and graph classes have been created. The Toolbar class family is a completely standalone, reusable subsystem. It can be easily tailored for other applications. (AWT does not support toolbar as a standard class). Also, this is the first example of where the Design Patterns are used in the codebase. The Observer pattern was used to implement the Model-View-Controller paradigm for graph display. Also a reusable graph class family has been created.

The above contributions agree with the current and future trends in software engineering where developers should endeavor to incorporate frameworks, design patterns and reusable components [41][42].

## 7.2 Lessons Learned and Future Work

Using Java for implementing the codebase suggests that OO languages in general are suitable for implementing application engineering environments based on Commonality

Analysis. Commonalities or variabilities quite often could be isolated to a single entity (such as a class) and could be implemented by subclassing or modifying the constructors. The Command class and it subclasses provide a good example. Adding a new option to a program involves creating a class derived from the abstract Command class and overriding Command's run method. Modifying the constructors can accommodate a number of variabilities in the generated family members. Also it seems that using Java interfaces is quite suitable for automated code generation, by decoupling of the component classes.

Future work should include further decoupling of the classes in the codebase. Some classes in the codebase depend more tightly on each other than is necessary. This could be changed by utilizing Java interfaces and the JDK1.1 Reflection API. In the Reflection API it is possible to load a class at runtime and examine the interfaces it supports. Then the calling class can perform appropriate actions depending on the methods found. Decoupling of the classes and usage of interfaces would make the classes more general and suitable for usage in other application engineering environments.

Another concern is whether the Graph Editor family was a good choice for demonstrating Commonality Analysis and AEE development. Certainly this example shows the ideas of implementing the commonalities and variabilities in a non-trivial system. However, it focuses quite a lot on automated GUI generation, which in the modern Application Development Environments is partially automated by a usage of the WYSIWYG GUI builders. A Commonality Analysis for a system without a GUI would provide useful new information.

# Bibliography

[1] M.Ardis and D.Weiss, *Defining Families: The Commonality Analysis*, in Proceedings of the Nineteenth International Conference on Software Engineering, pp. 649-650, May 1997.

[2] S.Katz, et al. *Glossary of Software Reuse Terms*, Gaithersburg. MD: National Institute of Standards and Technology, 1994.

[3] J.Foreman, *Product Line Based Software Development- Significant Results, Future Challenges*. Software Technology Conference, Salt Lake City, UT, April 23, 1996.

[4] J.Neighbours, *The Draco Approach to Constructing Software from Reusable Components*, IEEE Transactions on Software Engineering, SE-10,1984.

[5] R.Prieto-Diaz, *Domain Analysis: An Introduction*. Software Engineering Notes, 15April 1990: 47-54.

[6] S.Wartik and R.Prieto-Diaz, *Criteria for Comparing Domain Analysis Approaches*, in Proceedings of the Fourth Workshop on Institutionalizing Software Reuse, 1991.

[7] R. Prieto-Diaz, Some Experiences in Domain Analysis, in Proceedings of the Sixth Workshop on Institutionalizing Software Reuse, 1993.

[8] K. Kang, S.Cohen, J. Hess, W. Novak, and S. Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Tech. Rep. CMU/SEI-90-TR-21, Carnegie Mellon Software Engineering Institute, Nov. 1990.

[9] S.Cohen, et al. *Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain* (CMU/SEI-91-TR-28, ADA 256590). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1992.

[10] *Sandwich Method - A Domain Analysis Method Developed by Reuse, Inc and Available in Reuse Library Process Model,* Tech. Rep. IBM STARS CDRL 03041-001, U.S. Air Force Electronic Systems Center, Hanscom Air Force Base, MA, July 1991.

[11] J.Petro, A.Peterson, W.Ruby, In-*Transit Visibility Modernization Domain Modeling Report Comprehensive Approach to Reusable Defense Software* (STARS-VC-H002a/001/00). Fairmont, WV: Comprehensive Approach to Reusable Defense Software, 1995.

[12] G.Campbell Jr. *Synthesis Reference Model.* Software Productivity Consortium, Herndon, Virginia 1990.

[13] K.Schnell, N.Zalman, A.Bhatt, *Transitioning Domain Analysis: An Industry Experience* (CMU/SEI-96-TR-009). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

[14] *Conceptual Framework for Reuse Processes* Volume I, Definition, Version 3.0 (STARS-VC-A018/001/00). Reston, VA: Software Technology for Adaptable Reliable Systems, 1993.

[15] M.Simos, *Organization Domain Modeling (ODM) Guidebook Version 2.0*, Software Technology for Adaptable Reliable Systems (STARS), (STARS-VC-A025/001/00). Manassas, VA: Lockheed Martin Tactical Defense Systems, 1996. Also available [online] WWW URL: http://www.asset.com/WSRD/abstracts/ABSTRACT_1176.html (1996).

[16] DISA/CIM Software Reuse Program, *Domain Analysis Guidelines*, Arlington, VA, March1993. POC: Sherrie Chubin (703) 285-6900.

[17] D.M.Weiss, *Synthesis Operational Scenarios*, Software Productivity Consortium, Aug 1990.

[18] L.Walton & J.Hook *On Understanding a Commonality Anaysis.* in Proceedings of the OOPSLA'96 Workshop on Domain Analysis: Processes and Results, Oct. 1996.

[19] M.Ardis and D.Weiss, *Definig Families: The Commonality Analysis - Tutorial Notes*, Proceedings of the 1997 International Conference on Software Engineering, Boston, May 1997.

[20] W.Lam, *Creating Reusable Architectures: Initial Experience Report,* Software Engineering Notes vol 22 no 4, ACM SIGSOFT, July 1997.

[21] D.Dikel, *Applying Software Product-Line Architecture.* IEEE Computer, August 1997.

[22] H.A.Schmid, *Creating Applications From Components: A Manufacturing Framework Design,* IEEE Software, Nov 1996.

[23] P. Clements, *Successful Product Line Engineering Requires more than Reuse*, in Proceedings of the Eighth Workshop on Institutionalizing Software Reuse, 1997.

[24] Clements, Brownsword, *A Case Study in Successful Product Line Development*" Software Engineering, Institute technical report, November 1996.

[25] J.Meekel et al, *From Domain Models to Architecture Frameworks*, in Proceedings of the ACM SIGSOFT 1997 Symposium on Software Reusability, Boston, MA, May 1997.

[26] G.Arango, R.Pietro-Diaz, *Domain Analysis and Software Systems Modeling*, Tutorial, IEEE, 1991.

[27] E.Gamma & all, *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, MA, 1994.

[28] J.Rumbaugh & all, *Object-Oriented Modeling and Design,* Prentice Hall, 1991.

[29] L.Wall and R.L.Schwartz, *Programming perl,* O'Reilly & Associates, Inc, CA, 1992.

[30] D.M.Hoffman and P.Strooper, *Software Design, Automated Testing, and Maintenance - A Practical Approach,* International Thomson Computer Press, 1995.

[31] D.M.Hoffman and P.Strooper, *The Testgraph Methodology: Automated testing of collection classes.* Journal of Object Oriented Programming, 1995

[32] D.L.Parnas, P.C.Clements, D.M.Weiss, *The Modular Structure of Complex Systems.* in Proceedings of the Seventh International Conference on Software Engineering, FL March 1984.

[33] D.Flanagan, *Java in a Nutshell - A Desktop Quick Reference for Java Programmers,* O'Reilly & Associates, Inc, CA, 1996.

[34] D.Tkach and R.Puttick, *Object Technology in Application Development.* The Benjamin/Cummings Publishing Company, Inc.

[35] G.Arango, *Software Reusability*, Ellis Horwood Ltd., 1994.

[36] H.M.Deitel and P.J.Deitel, *Java - How to Program.* Prentice Hall, NJ, 1997.

[37] R.Kadel, *Learn how to extend the AWT with your own image buttons.* In *JavaWorld,* Web Publishing Inc. February 1997.

[38] M.Fellows and N.Casey, *This is MEGA-Mathematics,* Los Alamos National Laboratory, 1993.

[39] *Graph Glossary*, Mathmania Homepage, http://csr.uvic.ca/~mmania/graphs/glossary.htm, University of Victoria.

[40] Domain Engineering and Domain Analysis, Homepage, http://www.sei.cmu.edu/technology/str/descriptions/deda_body.html, Carnegie Mellon University.

# Appendix A

# Commonality Analysis Document

# Graph Editors Commonality Analysis

## A.1 Introduction

The purpose of this analysis is to define the requirements for the Graph Editor family and provide basis for:

- a way of specifying family members
- a way of generating some or all of the code and documentation for family members
- an environment for composing family members from a set of components that are designed for use in many family members

## A.2 Overview

Graph Editors is a family of visual editors for creating graphs. Graph Editor provides means for adding, removing, and editing arcs and nodes. Also it allows to save or read the graph from the file.

## A.3 Dictionary of Terms

### A.3.1 GUI

**application window**

The window where the application is executed

**menubar, menubar item**

The standard GUI navigation menu

**toolbar**

The panel with image buttons used for navigation

**caption bar**

The top part of the application window

**control bar**

The panel with controls in the top part of the window

**control bar item**

The Control placed on the control bar

**status bar**

The bottom part of the application window used for displaying messages

**view area**

The area where graphs are displayed

**dialog box**

A window where user can imput some data

**scroll bar**

A bar used for scrolling the contents of the View area

**grid tool**

The grid used for an easy alignment when drawing nodes

**cross tool**

Rubberbanded drawing guide

**{black .. white}**

{black, blue, cyan, gray, green, magenta, orange, pink, red, white, yellow}

### A.3.2 Graph Abstraction

**graph**

A finite set of dots called nodes (or vertices) connected by links called arcs (or edges). More formally: a simple graph is a (usually finite) set of nodes V and set of unordered pairs of distinct elements of V called arcs.

**directed graph**

A a graph in which the arcs are directed. (Formally: a digraph is a (usually finite) set of nodes V and set of ordered pairs (a,b) (where a, b are in V) called arcs. The

vertex a is the initial (source) node of the edge and b the terminal (destination) node.

**node adding, node removal**

Adding the node (arc) to the graph

**arc adding, arc removal**

Deleting the node (arc) from the graph

**node moving**

The changing of the node geometrical location

**node properties, arc properties, graph properties**

The node (arc, graph) attributes

**node editing, arc editing**

The changing the values of the node (arc) properites

**AEE**

Application Engineering Environment

**java application**

Application written in Java programming language

**javadoc**

A software tool for generating the documentation from the source code

**generation time (compile time) variability (GT)**

Variability that is established during the generation of the family member

**runtime variability (RT)**

Variablity that is established during the runtime of the application

VITA

Surname: Kacy Given Names: Grzegorz Roman

Place of Birth: Wodzislaw Slaski, Poland

Educational Institutions Attended:

University of Victoria 1995 to 1997

Silesian Technical University, Gliwice, Poland 1988 to 1993

Degrees Awarded:

B.Sc. Silesian Technical University 1993

Honours and Awards:

Silesian University Deans Award 1989-93

Publications:

Magister Degree Thesis, Silesian Technical University 1993

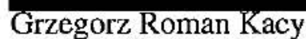
# PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis (or dissertation) to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

Commonality Analysis, Case Study in the Compositional Approach

Author:



Grzegorz Roman Kacy

Date: Apr 3, 1998