INTRODUCING *FireWorks*: A TOOL FOR REGRESSION TESTING WITHIN THE SMALLTALK ENVIRONMENT

by

Gerald C. Hurdle

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Computer Science

Ottawa–Carleton Institute for Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario
1998

The undersigned hereby recommend to

The Faculty of Graduate Studies and Research

acceptance of the thesis

Introducing *FireWorks*:

A Tool for Regression Testing within the Smalltalk Environment

submitted by

Gerald C. Hurdle

in partial fulfillment of the requirements

for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

January 1998

ABSTRACT

Object–oriented technology is gaining wide acceptance by academia, research establishments, and corporations. Increased popularity gives rise to an increased desire for adequate tools to aid during the development of object–oriented systems.

The software development lifecycle is a process that prescribes how to develop software systems, including object–oriented systems. One important phase within the software lifecyle is testing. With the advent of object–oriented technology, additional testing techniques are required to effectively test object–oriented systems.

This dissertation proposes a tool, named *FireWorks*, to aid software developers during the task of object–oriented regression testing. *FireWorks* is based on the theory of constructing software firewalls. A firewall attempts to isolate a class and all of its related classes. Regression testing is simplified since the firewall for a class identifies all other affected classes that should be considered for retesting.

ACKNOWLEDGMENTS

To my supervisor, Professor Jean–Pierre Corriveau. Thank–you for your guidance, support, and for helping me achieve my goal.

To my family, for all their love and understanding. I had to leave my family to realize that we really were a family all along. They are:

my Grandmother, Henrietta Francescutti
my Mother, Jean Elisa Hurdle
my Father, Gerald Hurdle
my Brother, Brendan Allan Hurdle

And, to my newest found friends, the rambunctious Kyle Raymond Smith, and his special mother, N Jaynie.

Gerald Christopher Hurdle

Carleton University
Ottawa, Ontario, Canada
January 1998

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

3GL	Third Generation (computer programming) Language
BBT	Black–Box Testing
CASE	Computer–Aided Software Engineering
CCFW	Computed Class Firewall
CFW	Class Firewall
CI	Continuous Improvement
CRC	Class, Responsibility, Collaborator
CRD	Class Relationship Diagram
CUT	Class Under Test
GBT	Grey–Box Testing
GUI	Graphical User Interface
LOC	Lines Of Code
LSC	Language Specific Classes
MTTF	Mean Time To Failure
MUT	Method Under Test
OMT	Object Modeling Technique
OO	Object–Oriented
OOA	Object–Oriented Analysis
OOD	Object–Oriented Design

OOPL	Object–Oriented Programming Language
OOT	Object–Oriented Testing
ORG	Object Relation Graph
SDA	Semantic Dependency Analysis
SQE	Software Quality Engineering
WBT	White–Box Testing

Chapter 1

THE NEED FOR OBJECT–ORIENTED REGRESSION TESTING

Introducing *FireWorks*

1.1 Background Information

Object–oriented (OO) technology is gaining wider acceptance by academia, research establishments, and corporations (Murphy *et al.*, 1994). The reasons for such acceptance are many: modeling the real world with objects, encapsulation of state and behavior, information hiding, and code reuse by inheritance, just to name a few. Many of the characteristics of OO have been purposely developed to address some of the issues concerning the building of large and complex (and thus expensive) software systems. The focus of OO technology seems to be slanted towards the activities closest to the start of the traditional software life–cycle. In particular, OO technology attempts to focus on requirements, analysis, and design[1] (Coad and Yourdon, 1991, 1992; Jacobson *et al.*, 1992; Rumbaugh *et al.*, 1991). But what does OO say about the important software phase of testing? The simplest answer is: very little. That is not to say that OO is without merit, and as such should be done away with. Rather, further research should be

[1] Some CASE tools allow for automatic code generation based on captured designs.

performed in the important area of OO testing (OOT)[2] (Binder, 1994 and 1996; Corriveau, 1996; Probert, 1994).

Regardless of technology used, testing as an activity seems to be treated less importantly than other software activities. The reasons are varied and range from psychological to mundane to shear neglect (Myers, 1979). The overall effect of improper testing is witnessed in the dramatic increase in the cost to produce quality software, where overall cost is a measure of time, money, and resources (Myers, 1979; Binder, 1994). The testing process (see Figure 1) was developed to address the need for thorough and adequate testing of software (Sommerville, 1996, p. 448). The ultimate goal of the testing process is simple: to detect the presence of defects as early as possible. Software defects "are errors in designer coding which are the root cause (or potential cause) of a system failure" (Probert, ibid., p. 39). Some of the stages of the testing process map well within the context of objects. For example, the traditional unit test is analogous to an OO method test. Still other testing levels (cluster, subsystem, system) have the same semantic meaning, but are performed very differently. Consider the testing level of regression testing.

---

[2] OOT refers specifically to testing object–oriented code. As will become evident in Chapter 2, OOT is one element in software quality engineering.

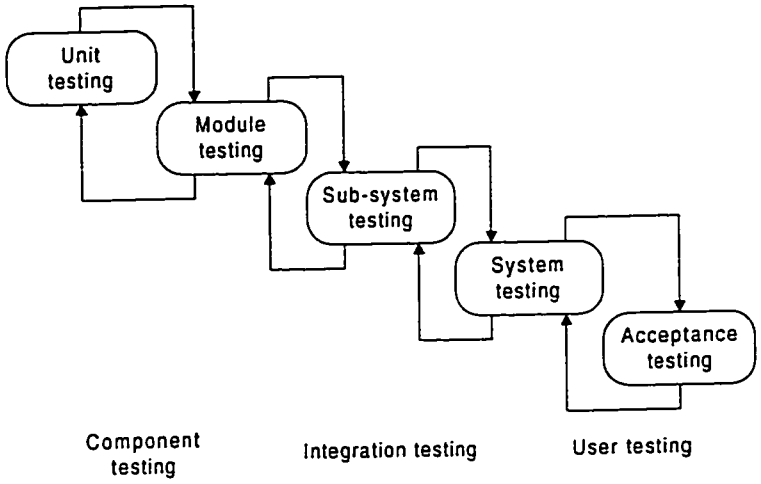


Figure 1: The testing process, as described by Sommerville.

Sommerville (1996, p. 448) defines regression testing as:

> *After a defect in a program has been discovered, it must be corrected and the system should then be re–tested. This form of testing is called* regression testing. *Regression testing is used to check that the changes made to a program have not introduced new faults into the system.*

Traditional approaches to regression testing are based on the control–flow model (Kung *et al.*, 1995). The control–flow model is well suited to capturing the sequential, step–by–step nature of a software system implemented in a typical third–generation computer programming language (3GL) (for example 'C'). However, the control–flow model is not well suited for OO programs (Kung *et al.*, ibid.). Given that OO programs are more data centric (due in part to encapsulation), the control–flow model fails to capture the series of relationships between various classes. So, what type of strategies are available to perform regression testing within OO systems?

One strategy would be to identify classes that are somehow related to the class being considered for regression testing. For example, suppose a software developer wishes to modify an existing class Car within an entire system of classes. Identifying which classes are related to class Car would aid the software developer during retesting. Such a strategy forms the basis for the proposed tool called *FireWorks*.

1.2 What is *FireWorks*?

FireWorks is a tool designed to aid software developers during the task of regression testing of object–oriented systems.

Given a class under test (CUT), the central task of *FireWorks* is to build the corresponding object relation graph (ORG). An ORG identifies all classes that are related in various ways to the CUT. ORGs themselves are built using a firewall class. A firewall, in the OO software sense, is the technique employed to determine the **largest** set of classes[3] related to the CUT (Kung *et al.*, ibid.). *FireWorks'* class firewall is able to detect the following relationships between classes[4]:

- inheritance relationships
- aggregation relationships

[3] This refers to all classes that have a direct relationship to the CUT. In some circumstances, a firewall can be used to determine the set of classes that have an indirect relationship to the CUT.

[4] Refer to Chapter 2 for the formal definitions of inheritance, aggregation, and association.

- association relationships

It is then possible to have an accurate representation of a class and all of its related classes using the above three types of relationships. Intuitively, the three class relationships are based on three methods of organization that people constantly employ (Coad and Yourdon, 1992, p. 1):

> 1. The differentiation of experience into particular objects and their attributes. For example, distinguish between a tree and its size or spatial relations to other objects.
> 2. The distinction between whole object and their component parts. For example, contrast a tree with its component branches.
> 3. The formation of and the distinction between different classes of objects. For example, form the class of all trees and the class of all stones and distinguish between them.

Regression testing of a CUT becomes easier given the corresponding ORG. The ORG identifies all related classes to a CUT that could be retested once the CUT itself has been changed.

1.3 An Example Object Relation Graph (ORG)

This section presents an example object relation graph (ORG) as a way to introduce the concepts associated with ORGs (see Figure 2).

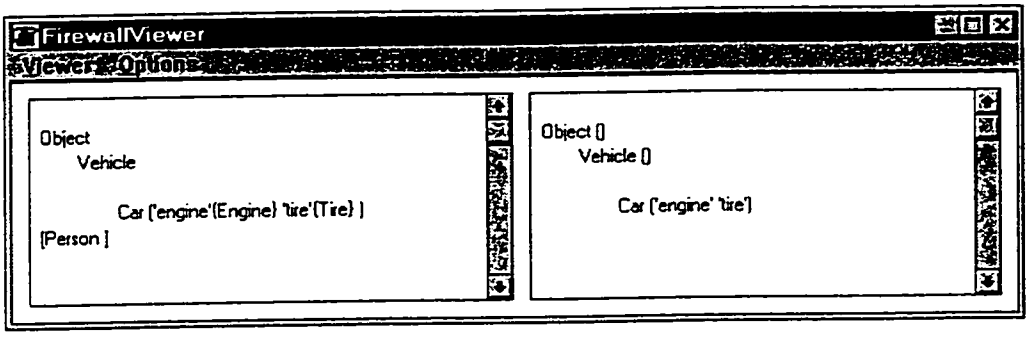


Figure 2: Object relation graph for class Car, as determined using the tool *FireWorks*.

The class under test (CUT) for this example is the class Car[5]. The ORG for class Car identifies the following relationships:

- an inheritance relationship exists between class Vehicle and class Car
- an aggregation relationship exists between class Car and classes Engine and Tire
- an association relationship exists between class Person and class Car

Thus, the ORG in the left half of Figure 2 identifies those classes (Vehicle, Engine, Tire, and Person) that could be re–examined and/or retested once changes have been made to class Car.

1.4 Thesis Organization

The remaining chapters of this thesis are as follows:

- Chapter 2 reviews current thought and practice on OO regression testing, and provides the theoretical basis for constructing firewalls using a tool such as *FireWorks*.

[5] Chapter 3 presents a more rigorous explanation for the class Car example.

- Chapter 3 presents various detailed examples of OO systems that were analyzed using *FireWorks*.
- Chapter 4 presents the design and implementation details of *FireWorks*.
- Chapter 5 summarizes the contribution of *FireWorks*, and examines possible directions for further research.
- Appendices that present the high–level and detailed design of *FireWorks*.

Enjoy!

Chapter 2

THE BASIS FOR CONSTRUCTING FIREWALLS

Foundations for *FireWorks*

Given the need for a tool such as *FireWorks*, this chapter presents the literature basis for constructing firewalls. This chapter begins with a brief review of object–oriented terminology. The next section briefly reviews software quality engineering, with an emphasis on software testing. The third and final section presents the current thought and theory associated with constructing firewalls.

2.1 Review of Object–Oriented Terminology

This section provides a very brief introduction to the common terminology used within object–oriented systems. It is assumed that the reader has a basic understanding of the concepts reviewed. For a more thorough explanation of objects, the reader can consult authors such as Budd (1991), Jacobson *et al.*, (1992), and Rumbaugh *et al.* (1991).

2.1.1 Objects and Classes

An *object* is the *encapsulation* of both structure and behavior. Thus, an object is defined both in terms of its attributes (what it is), and its services (what it can do). An object is typically thought of as an *abstraction* of something in a problem domain, reflecting the capabilities of a system to keep information about it,

interact with it, or both (Coad and Yourdon, 1992). Objects communicate with one another by the exchange of messages. A message send is analogous to a procedure call to the receiver (Adams, 1992).

A *class* serves as a template[6] for specifying a description of one or more objects with a common set of structure and behavior (Coad and Yourdon, ibid.). A class also includes a description of how to create new objects (*instances*) in the class.

2.1.2 Class Relationships

A brief review is offered of the inheritance, aggregation, and association concepts used in OO modeling (Rumbaugh *et al.*, ibid.). In OO programs, there are three different relationships between classes:

1. inheritance
2. aggregation
3. association

In OO programming languages (e.g., C++ and Smalltalk), the inheritance feature is provided to support generalization and specialization concepts, and to encourage the code reuse in the implementation. An *inheritance relation* between two classes means the properties defined for a class (i.e. the superclass) are automatically defined for all of its subclasses (unless selective and/or overriding inheritance is specified). Aggregation, an abstract concept, is also supported in OO programming languages through encapsulation. Using aggregation, a

composite object can be defined based on its component objects. The composite object is called an aggregated class object. The relation between an aggregated object class and its component object class is called an *aggregation relationship*. An *association relationship* means that two independent object classes[7] associate with each other in some other manner[8]. The associations include data dependence, control dependence, or message passing between two independent object classes.

2.2 Testing as a Software Quality Engineering Activity

This section provides a very brief overview of software quality engineering, and of software testing. For further in–depth detail, the reader is invited to consult any of the following authors: Beizer (1990), Myers (1979), Probert (1994, 1995a, 1995b, 1995c, and 1995d), or Sommerville (1996).

2.2.1 What is Software Quality Engineering?

Probert (1994, p. 58) defines software quality engineering (SQE) as "a comprehensive life–cycle approach concerned with every aspect of the software product development process." Probert (ibid.) further states that "an SQE program includes a comprehensive set of quality objectives, measurable quality attributes (quality metrics) to assess progress towards those objectives, and

---

[6] Not to be confused with templates in C++.

[7] Two object classes are independent if there are no aggregation and inheritance relations between them.

[8] Refer to section 2.3.1.1 The Object Relation Graph Defined (page 17).

quantitative certification targets for all component software development processes."

2.2.2 SQE Components

Probert (ibid., p. 97) identifies the following core components of SQE:

- Validation
- Verification
- Certification
- Metrics
- Continuous Improvement (CI)

Sommerville (1996, p. 446) points out that "verification and validation are sometimes confused." Yet, in fact, they are very different activities. Boehm (1979) succinctly summarizes the differences as:

- Validation: Are we building the right product?
- Verification: Are we building the product right?

Jacobson *et al.* (1992), borrowing from Boehm, alters the definitions slightly when considering objects:

- Validation: Are we building the right objects?
- Verification: Are we building the objects right?

Sommerville (ibid.) defines validation as "checking that the program as implemented meets the expectations of the software customer." Probert (ibid., p. 98) provides a definition of validation:

> *Any activity which helps establish the suitability or "worth" [to the client, of course] of the object [usually software] being validated.*

As an example, most functional testing activities are validation activities. For example, Input–Domain Partitioning considers the input domain of a software component (module) to identify normal and exceptional behaviors.

Verification, according to Sommerville (ibid.), "involves checking that the program conforms to its specification." Once again, Probert (ibid., p. 99) provides a definition of verification:

> *Its primary purpose and effect is to help demonstrate the integrity or self–consistency of the object being verified [component verification] or the integrity of the process employed to derive this object from another [transform verification].*

As an example, most syntax–directed quality assurance activities are verification activities. For example, code inspections by an inspection team for the purpose of finding and recording software defects.

Certification "is a set of activities intended to [totally] decide a binary relation on measures and metrics" (Probert, ibid., p. 100). More formally, certification is defined as (Probert, ibid.):

> *[Certification] Measures the completeness of a quality assurance activity or program against stated requirements, "if you can't measure what you've achieved, how do you know you're finished?" Certification provides a measurement of the thoroughness of validation and verification activities.*

A software metric, according to Sommerville (ibid., p. 623), "is any type of measurement which relates to a software system, process or related documentation." Such examples would include measuring the size of a product in lines of code (LOC), mean time to failure (MTTF), and availability. Refer to Sommerville (ibid.) for a more thorough treatment on the subject of software metrics.

Continuous improvement (CI) refers to the fact that SQE, as a process, must be continuously applied and improved upon. CI is pervasive in every SQE component and activity.

2.2.3 Software Testing

SQE is comprised of a wide range of activities. For the purposes of this document, only the activity of software testing is considered[9].

Probert (ibid., p. 38) defines software testing as "the process of systematically exercising software using a formal test plan for the specific purpose of finding failures." Failures are defined by Probert (ibid., p. 35) "as system misbehaviors which have been detected."

---

[9] Consult Probert (1994a, 1994b, 1994c, 1994d, 1994e) for a more thorough explanation of the remaining SQE activities.

According to Probert (ibid., p. 113), software testing is generally customer–directed and system–directed. Customer–directed in the sense that a customer would be concerned with such issues as performance, compatibility, dependability, and robustness. System–directed in the sense that a system designer would be concerned with such issues as design optimization, design for testability, and platform independence of both hardware and software.

2.2.3.1 Black–Box Testing

Customer–directed testing techniques include those of black–box testing (BBT). Also known as functional testing, BBT attempts to validate a program specification against a customer's functional requirements. Probert (ibid., p. 120) notes that BBT focuses on program features and behavior. Cause–Effect graphing is an example method of black–box testing. A Cause–Effect graph connects combinations of causes to visible system behaviors, called effects, to produce a connected logic network.

2.2.3.2 White–Box Testing

System–directed testing techniques include those of white–box testing (WBT). Also known as structural testing, WBT attempts to verify a program's structure and implementation. Probert (ibid., p. 122) notes that WBT "focuses directly on the internal logic to verify that the code is executing as intended." Probert (ibid., p. 124) warns that WBT "does not ensure that the correct features have been implemented or that the code meets functional requirements." Thus, WBT is

most appropriate at the completion of BBT execution. Code inspections are an example method of WBT. Specifically, code inspections are categorized as a static analysis technique. Static analysis, as opposed to dynamic, involves "the verification (checking) of code by means other than the direct execution of the software under test" (Probert, 1995c., p. 3)

2.2.3.3 Grey–Box Testing

With the advent of object–oriented technology, a third method of testing software has been identified. Known as grey–box testing (GBT), this method of testing, as the name suggests, is a mixture of both black–box and white–box testing. Also know as object–directed testing, GBT considers all system interfaces, platform discrepancies (hardware and software), design optimization, and design for testability (Probert, 1995d, p. 1). Probert (ibid., p. 4) notes that the key customer requirement of GBT is compatibility to existing systems. Both backward and forward compatibility are considered. Backward compatibility so that there are "no unpleasant surprises to customers" and "no interruption or negative impact to existing services" (Probert, ibid.). And forward compatibility for planning of new platforms. The data–flow directed method is an example of grey–box testing.

Interestingly, Probert (ibid., p. 8) states that "tools to support coverage tracking are essential."

2.2.3.4 Regression Testing

While there are numerous other methods for testing software, only one other testing method shall be discussed in this dissertation. Regression testing "tests that a system has not 'regressed' due to changes" (Probert, 1994, p. 129). See Figure 3 (Probert, ibid., p. 130). Regression testing attempts to answer the question "Does the base software still work and meet specification?" It is interesting to note that Probert (ibid.) states that regression testing "depends heavily on automation."

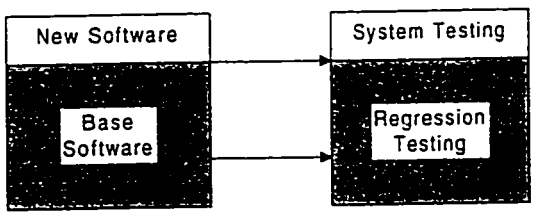


Figure 3: Regression testing.

2.3 Theory of Firewalls

Historically, Leung and White were the first researchers to introduce the concept of firewalls to aid in the regression testing of object–oriented (OO) systems (Leung and White, 1992). Their first attempt to define a firewall was based on a call graph. Inspired by programming in traditional third generation (computer) languages (3GLs), call graphs identify the sequential order of procedures/functions as they are invoked. Unfortunately, applying call graphs to objects yields little information. It is true that objects send messages to other objects, and that message sending reduces to invoking a procedure. However,

objects are much more than procedures given that objects contain both state and behavior. Realizing this difference, Leung and White introduced the notion of a firewall based upon a data–flow graph (Leung and White, ibid.). The basis of a data–flow graph is to enclose all affected software modules due to coupling through global data. A more succinct term for this type of enclosure is a firewall.

Kung *et al.* were the next group of researchers to refine and extend the firewall concept (Kung *et al.*, 1995a and 1995b). In an attempt to formalize firewalls, Kung *et al.* (1995a) provide definitions and an algorithm for the following:

1. Formal definitions for each of the three possible relationships between different classes: aggregation, association, and inheritance.
2. The definition of an object relation graph (ORG) to capture the various relationships between different classes.
3. An algorithmic approach for the formation of firewalls. The approach taken is formal and exploits the properties of items 1 and 2.

2.3.1 The Object Relation Graph Regression Test Model

The object relation graph (ORG) was first proposed as a regression test model by Kung *et al.* (ibid.). The ORG is used to represent the inheritance, aggregation, and association relationships that exist between various classes.

2.3.1.1 The Object Relation Graph Defined

The complete definitions of an ORG are supplied by Kung *et al.* (ibid., p. 53):

Definition 1. *An edge–labeled digraph $G = (V, L, E)$ is a directed graph, where $V = \{V_1, \ldots, V_n\}$ is a finite set of nodes, $L = \{L_1, \ldots, L_k\}$ is a finite set of labels, and $E \subseteq V \times V \times L$ is the set of labeled edges.*

Definition 2. *The ORG for an OO program P is an edge–labeled digraph $ORG = (V, L, E)$, where V is the set of nodes representing the object classes in P, $L = \{I, Ag, As\}$ is the set of edge labels, and $E = E_I \cup E_{AG} \cup E_{AS}$ is the set of edges defined below.*

Definition 3. *$E_I \subseteq V \times V \times L$ is the set of directed edges representing the inheritance relation between the classes. For any two classes $C_1, C_2 \in V$, $<C_1, C_2, I> \in E_I$ indicates the C_2 is a derived class of C_1.*

Definition 4. *$E_{AG} \subseteq V \times V \times L$ is the set of directed edges representing the aggregation relation between the classes. For any two classes $C_1, C_2 \in V$, $<C_1, C_2, Ag> \in E_{AG}$ indicates that class C_1 contains one or more objects of class C_2.*

Definition 5. *$E_{AS} \subseteq V \times V \times L$ is the set of directed edges representing the association relation between two independent classes[10]. For any two independent classes $C_1, C_2 \in V$, $<C_1, C_2, As> \in E_{AS}$ indicates that class C_1 associates with class C_2 in the following three ways:*

- Class C_1 uses data members of class C_2. This is called *data dependence.*
- Class C_2's member functions are invoked by some member functions of class C_1. This is called *message passing.*
- Class C_2's objects are defined as formal parameters of member functions in class C_1. This is called *object parameter passing.*

2.3.1.2 Class Firewall

The ORG can be thought of as the visualization of a class and its related structure. The mechanism used to build ORGs is the definition of a firewall class.

2.3.1.2.1 Class Firewall Defined

Any class can be changed in many ways. Kung *et al.* (ibid., p. 55) classify various changes into three types:

1. object behaviors, such as states or transitions, are affected
2. operations and behavior of its member functions are affected
3. relationships between this class and others are affected

Intuitively, a class firewall for a class C, denoted as *CFW(C)*, in an OO program/library is the set of classes that could be affected by changes to class C. From the view point of regression testing, these affected classes should be retested when class C is changed. Kung *et al.* only consider the first two types of changes. In other words, it is assumed that the relations between classes are not affected by the changed class.

The notion of class firewall defines the classes that are possibly affected (not necessarily affected) by the changes in class C. The following lemmas and proofs are provided by Kung *et al.* (ibid., p. 55):

Lemma 1. *Let class A be a subclass of class B in the inheritance hierarchy, and only class B is changed without affecting its relationships with other classes. If the change affects the inherited members (from class B) of class A, for adequate testing, not only should class B be unit retested, but class A should also be retested and reintegrated with class B.*

Proof. Since class A is a subclass of class B, class A must inherit some of class B's attributes. Thus, class A has a code dependency on class B due to the inherited attributes. Because the modification made in class B affects the inherited members of class A, according to Perry and Kaiser (Perry and Kaiser, 1990) they should be retested in class A to ensure that they work well in its reused context. Next, they should be reintegrated with other attributes of class A to make sure that they work well together. In addition, class A and class B should be reintegrated to make sure that correct member functions are executed in the inheritance hierarchy.

Example. Consider the inheritance relationship between class Vehicle and class Car (refer to Figure 7 on page 31). Class Vehicle is the generalized

---

[10] There is neither inheritance nor aggregation relationships between two classes.

class of the more specific Car class, as a car is a kind of vehicle. Suppose only class Vehicle is changed. According to Lemma 1, modifying class Vehicle requires that class Car be retested. Retesting of class Car is required because the modification made in class Vehicle directly affects the inherited attributes and behavior of class Car.

Lemma 2. *Let class A be an aggregate class of class B,*[11] *and only class B is changed, without affecting its relationships with other classes. For adequate testing, not only should class B be unit retested, but class A should also be retested with class B.*

Proof. Because a change of class B causes an effect on its object behavior or the operations of its member functions, class B should be retested at the unit level. Since class A contains some object of class B as its component, the change will affect the class A's object behavior in the following aspects:

1. the behavior of the aggregated part (e.g., class B's objects) is affected, and
2. other members of class A are affected if they are directly (or indirectly) dependent on the aggregated part[12]

Thus, those dependent members of class A should be retested and reintegrated with the instances of class B. Moreover, class A's object behavior should be retested.

Example. Consider the aggregation relationship between class Car and class Engine (refer to Figure 7 on page 31). Class Car is an aggregate class of class Engine, as a car has an engine. Suppose only class Engine is changed. According to Lemma 2, class Car should be retested and reintegrated with instance of class Engine.

Lemma 3. *Let A and B be two independent classes, and class A is associated to class B. If class B is changed without an effect on its relationships with other classes, then for adequate testing, not only should class B be unit retested, but class A should also be retested and reintegrated with class B.*

Proof. Since class A is associated to class B, one (or more) of the following cases must occur:

1. at least one member function of class A depends on some data members of class B,
2. at least one member function of class A has a message passing to class B[13], and

---

[11] A class is called an aggregate class if its object contains other class objects as its component.

[12] The dependency may be a data, state, message or control dependency.

[13] The message passing can be static or dynamic.

3. at least one member function of class A passes some class B objects as parameters in message passing.

In all three cases, one or more members of class A have some data dependency on some of class B members or objects. If the change in class B directly (or indirectly) affects those members or objects that the class A members are dependent on, then after class B retesting these dependent members of class A should be retested with class B. Later, these members should be reintegrated with other members of class A. Therefore, after changing class B, class A should be retested and reintegrated with class B to ensure the adequate testing.

Example. Consider the association relationship between class Person and class Car (refer to Figure 7 on page 31). Suppose only class Person is changed. According to Lemma 3, class Car should be retested and reintegrated with class Person.

2.3.1.2.2 Constructing a class Firewall

To compute the class firewall, Kung *et al.* (ibid.) introduced a binary relation R that is derived from the directed edges of an $ORG = (V, L, E)$:

$$R = \{ <C_2, C_1> \mid C_1, C_2 \in V \wedge (\exists l)(l \in L \wedge <C_1, C_2, l> \in E) \}$$

Call R *the dependence relation because it defines the dependence between the classes, according to the inheritance, aggregation, and association relations. More specifically,* $<C_2, C_1> \in R$ *if and only if one of the following cases holds:*

1. C_1 is a derived class of C_2,
2. C_1 is an aggregate class of C_2 (i.e., C_2 is part of C_1), or
3. C_1 is associated with C_2 by either accessing its data members or passing some messages.

In all of these cases, C_1 *is dependent on* C_2 *in the sense that code changes to* C_2 *would affect the behavior of* C_1*. The computed class firewall for a class C, denoted* CCFW(C) *then is defined as*

$$CCFW(C) = \{ C_i \mid <C, C_i> \in R^* \},$$

where R^* *is the transitive closure of* R. *That is, if* $<C_i, C_j> \in R$ *and* $<C_j, C_k> \in R^*$, *then* $<C_i, C_k> \in R^*$. *The transitive closure of* R *can be computed by the algorithm of Aho* et al. *(1983).*

Theorem 1. *Let G be an ORG of a given OO program P, and* R *be the dependence relation derived from G. Let C be a class in which a change is made to its defined or redefined members. Assume the dependencies between the classes of P are the dependencies of inheritance, aggregation, and association relations. Then, CCFW(C) = CFW(C), that is*

1. $CCFW(C) \subseteq CFW(C)$
2. $CFW(C) \subseteq CCFW(C)$

Proof(1). We want to prove that for any class $C_i \in CCFW(C)$, class C_i could be affected by the changes to class C and should be retested. The proof is by induction on n of dependence relation Rn.

Basis. For any class C_i, $<C, C_i> \in R^1$. Then $<C_i, C, l> \in E$ of G based on the definition of R. Thus, $<C_i, C, l>$ must be one of the three cases: inheritance edge, aggregation edge, or association edge. According to Lemmas 1, 2, and 3, in all of these cases C_i could be affected by changes to C, and C_i should be retested.

Inductive Hypothesis. Assume that for any class C_j, $<C, C_j> \in R^k$. Then class C_j would be affected by changes to class C and should be retested.

Inductive Step. We need to prove that for any class C_i, if $<C, C_i> \in R^{k+1}$, then class C_i could be affected and should be retested. According to the transitive nature of R, there must be $<C_j, C_i> \in R$ and $<C, C_j> \in R^k$. From the inductive hypothesis, class C_j could be affected by the changes to class C and should be retested. From the basis, class C_i could be affected by the changes of class C_j and should also be retested. Thus, class C_i could be affected by the changes to class C and should be retested.

Proof(2). We want to prove that if any class C_i, $C_i \in CFW(C)$, then class $C_i \in CFW(C)$. We prove this using contradiction. Assume there is class C_i which is not in CCFW(C), but $C_i \in CFW(C)$ in the sense that it could be affected by the changes to class C and should be retested.

Class C_i can be affected by the change of class C only when C_i is either directly or indirectly dependent on the defined members of class C. Thus, there must be one dependency change (denoted CHAIN*):* $\{<C_i, C_{j1}>, \ldots, <C_{jk}, C>\}$ *from class C_i to class C. As we know, the types of dependencies between two classes/objects can be classified into*

1. code dependency,
2. object dependency,
3. control dependency, and
4. data or state dependency

According to the assumption, they are dependencies of the three relations: inheritance, aggregation, and association. Hence, CHAIN $\subseteq$ R, *and* $<C_i, C> \in R^*$. *Therefore,* $<C_i, C> \in CCFW(C)$. *This is a contradiction. QED.*

From theorem 1, the following theorem can be easily derived.

Theorem 2. *Let CCFW(C) be the computed firewall for class C. For any class C_i, if C_i is not in CCFW(C), the C_i is not affected by the change in class C, and hence, no retest is needed.*

Using theorem 1, we can construct the class firewall for a changed class to enclose all possible affected classes that should be retested. According to theorem 1, these classes should also be reintegrated with their subclasses, aggregated classes, and associated classes to achieve adequate testing.

The notion of the class firewalls can be extended to a set of changed classes. Let $S = \{C_1, C_2, \ldots, C_k\}$ be a set of classes (that are changed). Then, the class firewall for S, also denoted CFW(S), is defined as follows:

$$CFW(S) = \bigcup_{\forall C \in S} CFW(C)$$

As an example, Figure 4 (Kung *et al.* ibid., p. 56) depicts an ORG for a subset of classes in the InterViews library (Linton, 1992), and Figure 5 depicts the class firewall for the class Subject in this subset (Kung *et al.* ibid., p. 57).

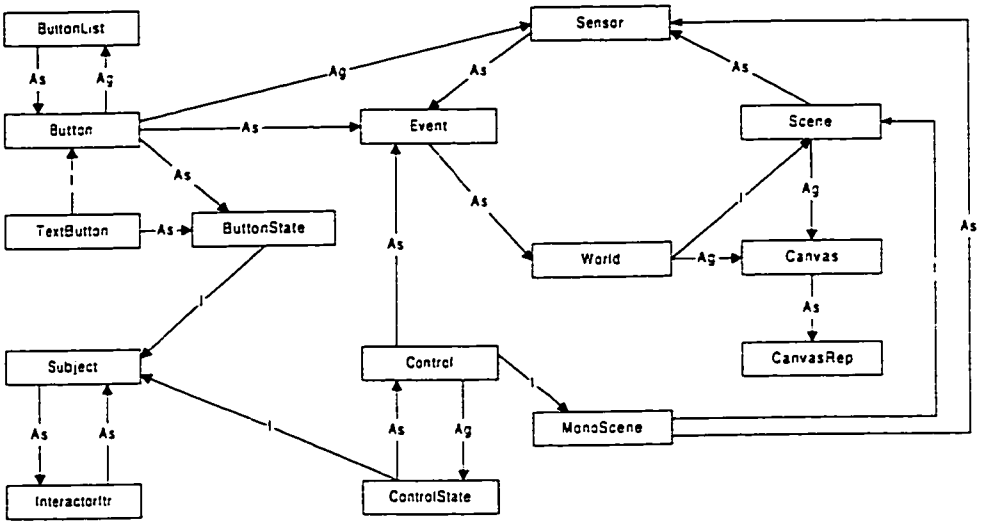


Figure 4: An ORG example for a subset of classes in the InterViews library, by Kung *et al.*

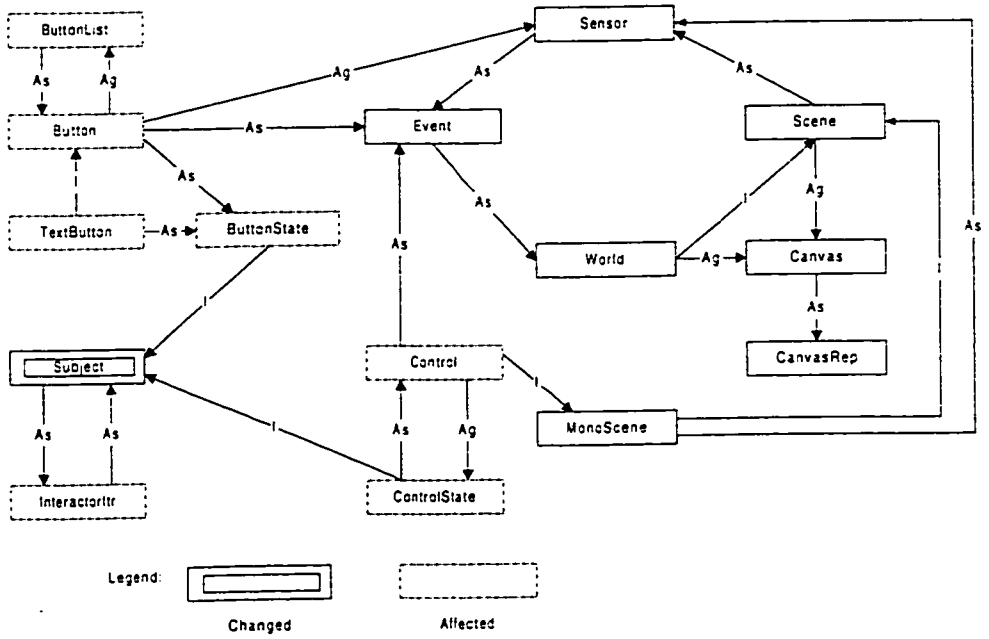


Figure 5: A class firewall for class Subject in a subset of the InterViews library, by Kung *et al.*

2.3.2 Cluster and Class Testing

In approximately the same time frame that Kung *et al.* refined the notion of a class firewall, Murphy *et al.* (1994) developed the concept of clustering classes.

The use of clusters for testing was developed as a result of experiences in testing a distributed network management system know as TRACS[14] (Trouble Advisor for Customer Services). TRACS is a mission–critical system designed for use by a telecommunications company to monitor services used by its large business customers, to identify network problems prior to service degradation, and to respond to discovered problems on a proactive basis.

The original version of the TRACS system was tested using system and cluster tests (Murphy *et al.*, 1993). System test cases were designed based on the functional and nonfunctional system requirements. Each system test case consisted of a series of steps for a software engineer to execute against the system and a natural language statement of the expected outcome. System testing was, thus, functional and dynamic because test cases were based on a user's view of the behavior of the system (Beizer, 1990). Cluster tests were also functional and dynamic. A cluster test, however, was executed against a portion of the entire system and sometimes required the development of additional drivers.

---

[14] TRACS is a registered trademark of MPR Teltech Ltd.

The original release of the TRACS system consisted of over 40 clusters. A cluster in the TRACS system was defined as a collection of classes related by a similarity in function. Classes within a cluster were either closely coupled and worked together to provide one unified behavior or they were independent and provided different kinds of similar functions. A diagram of a few of the clusters in the TRACS system is provided in Figure 6 (Murphy *et al*, 1994 p. 41). The Picture Display cluster was a collection of closely coupled classes that interacted to display alarms occurring in the network. The Graphics cluster, on the other hand, was a collection of largely independent classes supporting the display of various graphical shapes.

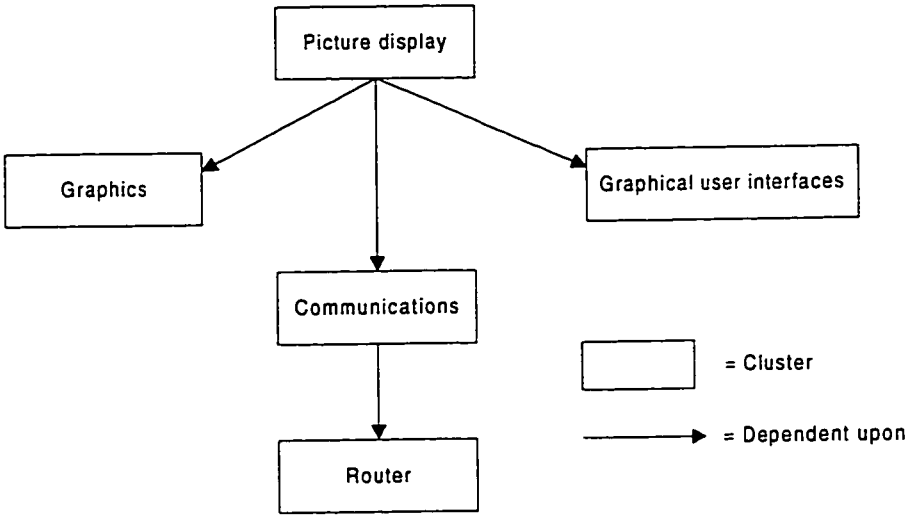


Figure 6: Cluster dependencies, as described by Murphy *et al.*

Since none of the programming languages used for implementing the TRACS system provided any support for cluster declaration, Murphy *et al.* (ibid., p. 40)

adopted the convention of placing all classes of the cluster in a common physical location, namely, a directory. The directory hierarchy was also used to classify clusters into two categories: application–specific clusters and library clusters. An application–specific cluster provided functionality that was directly related to the system specification. Library clusters provided functionality necessary for implementation.

The clustering of classes is very analogous to a firewall of classes. However, the major difference is that clusters of classes are formed very informally. That is, if two classes are somehow related, then the two classes will appear in the same cluster of classes. Precisely how the two classes are related is not clearly defined by Murphy *et al.* Rather the software designer is to recognize the two classes as related, and will arbitrarily place these two classes in the same cluster. The automatic creation of clusters thus proves unrealistic given the arbitrary placement of classes into clusters.

Chapter 3

DETAILED EXAMPLES

Exploring *FirelWorks* by Example

This chapter presents three example object relationship graphs (ORGs) in detail. Each of the example ORGs were created using the regression testing tool *FirelWorks*. Collectively, these examples attempt to demonstrate the various strengths and weaknesses of *FirelWorks* as an aid to regression testing. The first example revisits the Car example that was first introduced in Chapter 1, and provides more detail into the anatomy of an ORG. The second example, ORGs for various Collection classes, demonstrates the use of *FirelWorks* for a hierarchy of classes. Finally, the third example describes *FirelWorks* in terms of itself by giving the important ORGs of those classes that constitute *FirelWorks*.

The general format for each example is as follows. First, the class relationship diagram (CRD) is given for the class under test (CUT). The CRD pictorially represents:

i) Generalization–Specialization (Gen–Spec) Structures
ii) Whole–Part Structures
iii) Association Relationships

Generalization–Specialization (Gen–Spec) Structure is a term often used in OOA/OOD. Gen–Spec Structures may be viewed as part of the "distinguishing between Classes" aspect of the basic methods of organization[15] (Coad and Yourdon, 1992, p. 34). An example is the generalization Vehicle, and the specialization Car. Intuitively, a Gen–Spec Structure, from the specialization's perspective, can be thought of as an *is a* or *is a kind of* relationship. For example, a Car *is a* (*is a kind of*) Vehicle. Specializations are derived from generalizations, which group more general/common attributes (parts) and services (behaviors). Inheritance is the means used by OOPLs to portray Gen–Spec Structures.

Whole–Part Structure is another of the basic methods of organization (Coad and Yourdon, ibid.). An example is the whole Car, and the part (aggregate) Engine. Intuitively, a Whole–Part Structure, from the whole's perspective, can be thought of as a *has a* relationship. For example, a Car *has an* Engine. OOPLs typically use classes to represent the *Whole* in Whole–Part Structures. Instance variables[16] are typically used to represent the *Part* component.

Association relationships are yet another of the basic methods of organization (Coad and Yourdon, ibid.). Association relationships are used to classify remaining relationships, if indeed a relationship exists, that are not covered by

---

[15] The three basic methods of organization are described in Chapter 1.

[16] Class variables, class–instance variables, pool variables, and global variables are other variables, specific to Smalltalk, that could be used to represent Whole–Part Structures. Instance variables, by far, are the most common way to represent Whole–Part Structures.

either Gen–Spec or Whole–Part. For example, a Person building a Car forms an association between Person and Car. OOPLs typically adopt message passing as one way to form association relationships.

The CRD for a CUT represents the design of the CUT and its related classes, as envisioned by a software designer. Next, the ORG for the CUT, as determined by *FirelWorks* from the source code, is presented. Ideally, the CRD, and the corresponding ORG should be equivalent. CRD and ORG equivalency requires that all of the relationships described in a class's CRD should all be identified in the class's ORG.

Finally, a comparison is made between the CRD and its generated ORG. The comparison judges how equivalent the CRD is to its ORG. Comparing a CRD to its ORG is a manual process[17], best performed by the original software designer of the CUT. Comparing a CRD to its ORG may reveal relationships present in the CRD, yet absent in the ORG. This may indicate a possible limitation of *FirelWorks* in detecting relationships. Comparing a CRD to its ORG may also reveal relationships present in the ORG, yet absent in the CRD. This may represent a possible oversight by the software designer in designing the CRD.

---

[17] Future work could involve developing a tool that compares a CRD to its ORG. One strategy could involve representing a CRD as a directed graph. As ORGs are ultimately represented as directed graphs, comparing a CRD to its ORG would entail comparing their respective directed graphs.

3.1 Car Example

The Car example is meant to introduce the reader to the fundamental concepts concerning ORGs. The CRD is relatively simple and straightforward (Figure 7). The CRD contains each of the three types of possible relationships that can be detected by *FireWorks*:

1. Generalization–Specialization
2. Whole–Part Structures
3. Association Relationships

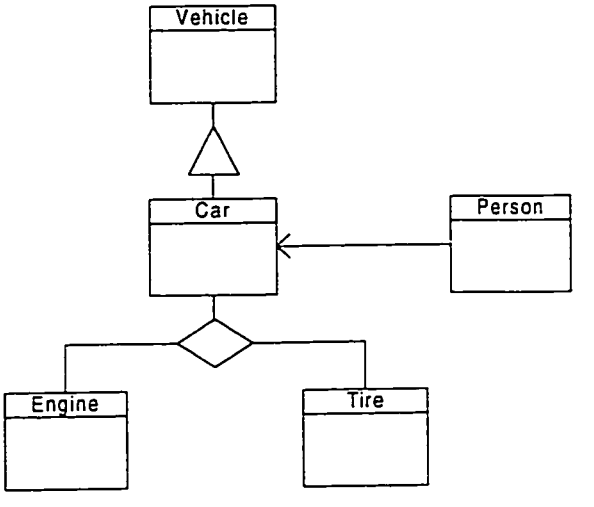


Figure 7: Class relationship diagram for class Car in OMT.

The Vehicle—Car relationship is an example of a Generalization–Specialization relationship. The Car—Engine relationship is an example of a Whole–Part Structure, as is the Car—Tire relationship. And, the Person—Car relationship is an example of an Association relationship.

The ORG for class Car was computed using *FireWorks* (see Figure 8).

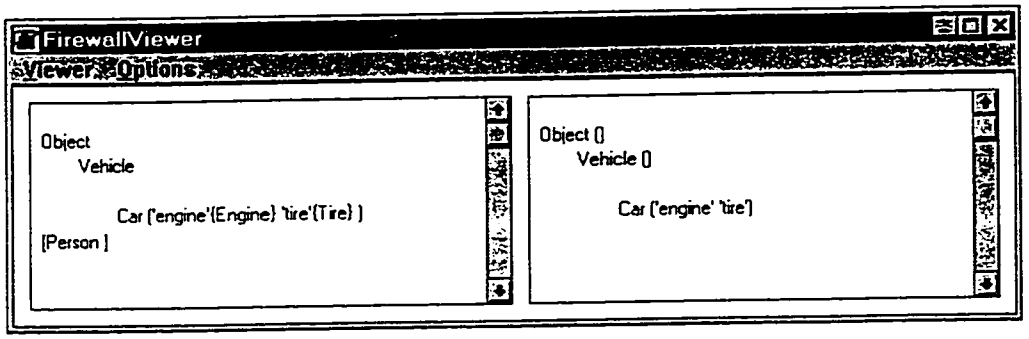


Figure 8: Object relation graph for class Car, as determined using the tool *FireWorks*.

The left pane of Figure 8 is the actual ORG as computed by *FireWorks*. The right pane is the object hierarchy as computed by Smalltalk. The right pane is offered as a means to quickly verify the correctness of computed ORGs. However, the object hierarchy of the right pane only reveals inheritance relationships, and partially provides for aggregation relationships. The right pane does not identify association relationships.

From Figure 8, notice that *FireWorks* represents inheritance by indentation, with the CUT on its own separate line. This is to draw attention by the reader to the CUT. Figure 8 shows that class Car inherits from the generalized class Vehicle. And that class Vehicle, in turn, inherits from the most generalized class, class Object.

From Figure 8, notice that *FireWorks* breaks class Car into two aggregates: *engine* and *tire*. In addition, the aggregate *engine* is of class Engine, and that aggregate *tire* is of class Tire.

ORGs constructed in *FireWorks* represent classes associated with the CUT in a list delimited by square brackets. From Figure 8, class Person is associated with class Car in some manner. The reasons for the association could be due to one of many types of dependencies: data, state, message or control. Refer to section 2.3.1.2.1 Class Firewall Defined for a more in–depth description of the possible reasons for an association relationship between two classes. Also, refer to section 4.3.2 Detecting Association Relationships for suggestions on how to classify association relationships according to dependencies.

Comparing the computer ORG (Figure 8) to the corresponding CRD (Figure 7) reveals that *FireWorks* correctly identified all of the possible relationships for class Car. In particular, the ORG for class Car identifies that:

1. Classes Object–Vehicle–Car form an inheritance hierarchy.
2. Class Car is an aggregate class of classes Engine and Tire.
3. Class Person is associated with class Car.

3.2 Collection Example

This section examines in detail the ORGs of one of Smalltalk's major work–horse data types: collections. Class Collection was chosen as an example as the ability to collect objects is ubiquitous amongst OOPLs. Class Collection (see Figure 9)

is an abstract class for objects that are containers for other objects. The specializations include such familiar classes such as Set, String, and Array. Also included are lesser knows classes such as Bag, Dictionary, OrderedCollection, and SortedCollection. Of all the classes found within the Collection hierarchy, only three classes will be analyzed by *FireWorks*:

1. Array
2. Dictionary
3. OrderedCollection

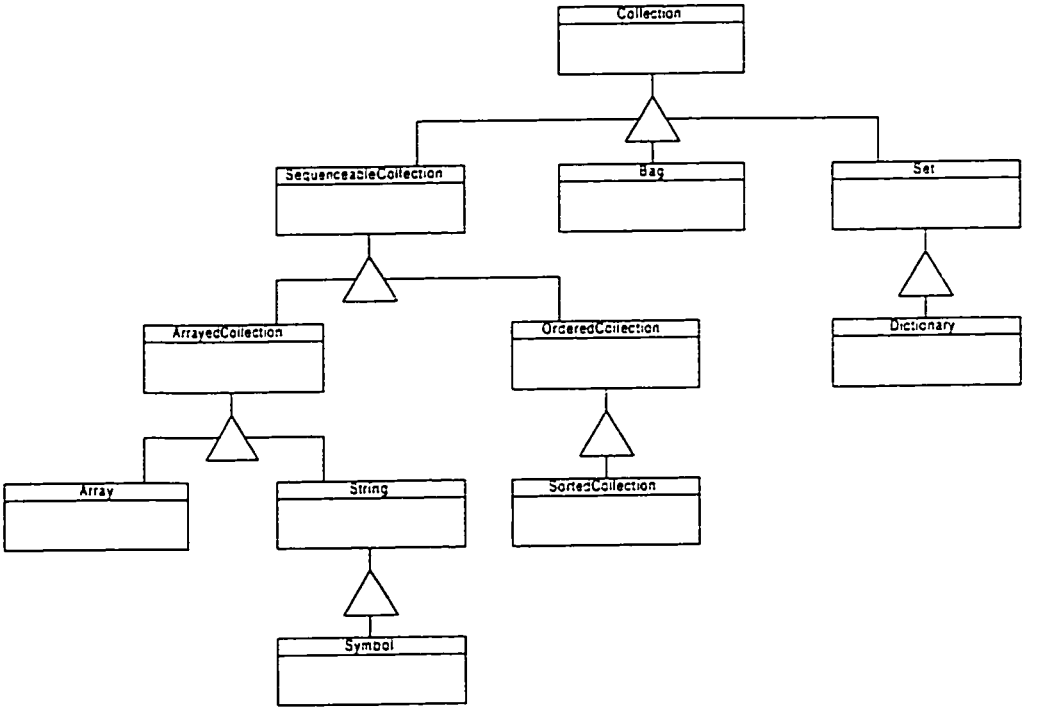


Figure 9: The most popular Collection classes in OMT.

3.2.1: Array

Class Array represents SequenceableCollections whose elements are any object. Arrays provide the concrete representation for storing a collection of elements

that have integers as external keys. Arrays are of fixed size, and therefore can only hold a specific number of objects. No more objects can be added to a full array.

Figure 10 shows the ORG for class Array, as determined by *FirelWorks*. Notice that *FirelWorks* correctly identified the superclasses of class Array: ArrayedCollection, SequenceableCollection, Collection, and Object. Also, the subclasses of class Array were correctly identified. The subclasses appear indented and below class Array. For example, class BOSSReaderMap is an immediate subclass of class Array. The ORG in the left pane of Figure 10 indicates that class Array does not have any aggregate relationships. This is confirmed in the right pane of Figure 10, which is the object hierarchy tree for class Array as determined by Smalltalk.

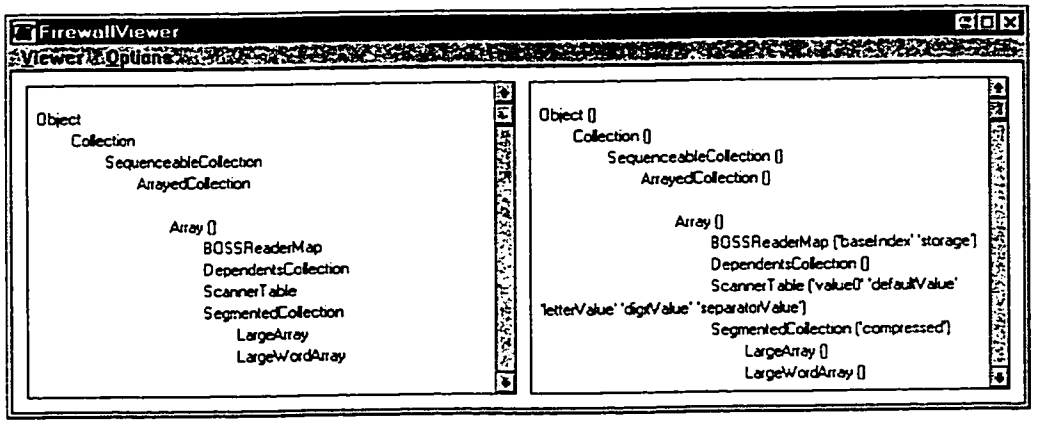


Figure 10: ORG for class Array, as determined by *FirelWorks*.

3.2.2: Dictionary

Class Dictionary is similar to class Array in that the objects contained within a dictionary are indexed. However, there are some major differences. Dictionaries are not of fixed size, and can grow as new objects are added. Dictionary indexes do not have to be integers, and can be any kind of object. The indexes, called keys, are not in any particular order.

Figure 11 shows the ORG for class Dictionary, as determined by *FireWorks*. *FireWorks* was correct in identifying the hierarchy for class Dictionary. The classes before class Dictionary represent all the superclasses. Those classes listed below class Dictionary represent all the subclasses.

FireWorks correctly identified *tally* as an aggregate of class Dictionary. Notice in the left pane, however, that *tally* has been incorrectly classified of type tally (tally in braces). In fact, *tally* is an integer count of the number of elements stored within the Dictionary. This example illustrates a limitation of *FireWorks*. *FireWorks* is always able to identify the aggregates for a class. However, *FireWorks* has difficulty in determining the *exact* type of class being collected for aggregates with a type of class Collection. Refer to section 4.3.1.1 Difficulties in Detecting Aggregation Relationships for further explanation.

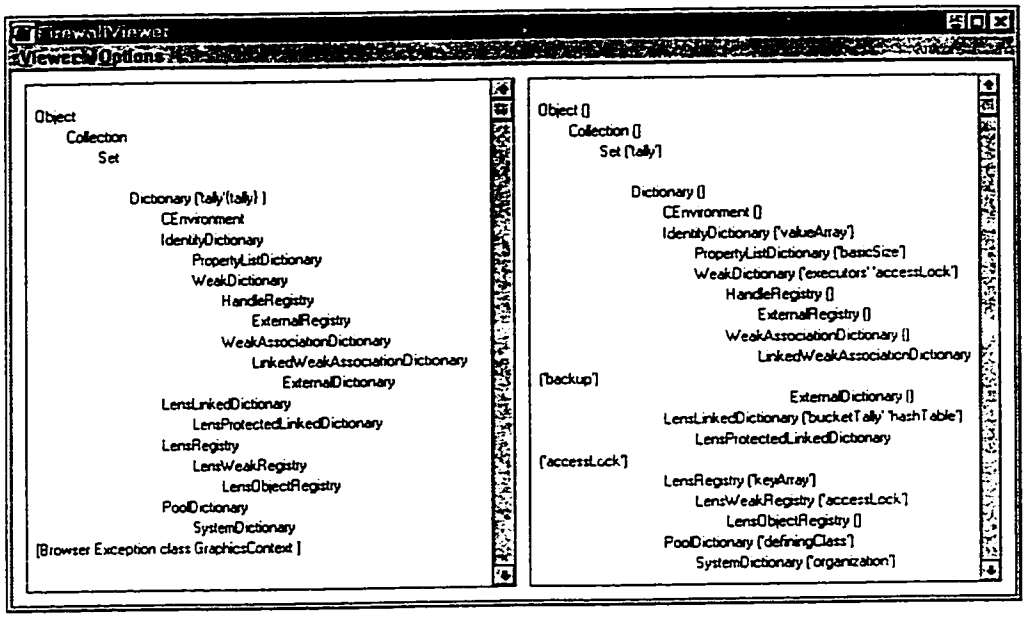


Figure 11: ORG for class Dictionary, as determined by *FireWorks*.

3.2.3: OrderedCollection

Class OrderedCollection is similar to class Array in that the objects contained within an ordered collection are indexed. The major differences include: ordered collections are not of fixed size, and the location (i.e. the index) does not have to be specified when adding an object.

Figure 12 displays the ORG for class OrderedCollection, as determined by *FireWorks*. *FireWorks* was correct in identifying the superclasses of class OrderedCollection: SequenceableCollection, Collection, and Object. The classes listed below class OrderedCollection represent all the subclasses.

FireWorks correctly identifies *firstIndex* and *lastIndex* as aggregates of class OrderedCollection. Notice in the left pane of Figure 12, however, that *firstIndex* has been incorrectly classified of type firstIndex (firstIndex in braces). Also, *lastIndex* has been incorrectly classified of type lastIndex (lastIndex in braces). In fact, *firstIndex* is the integer index of the first element of the collection, with *lastIndex* being the integer index of the last element. The use of type inference may allow for correctly determining that firstIndex and lastIndex are of type integer. Refer to section 4.3.1.2 Use of Type Inference for further explanation.

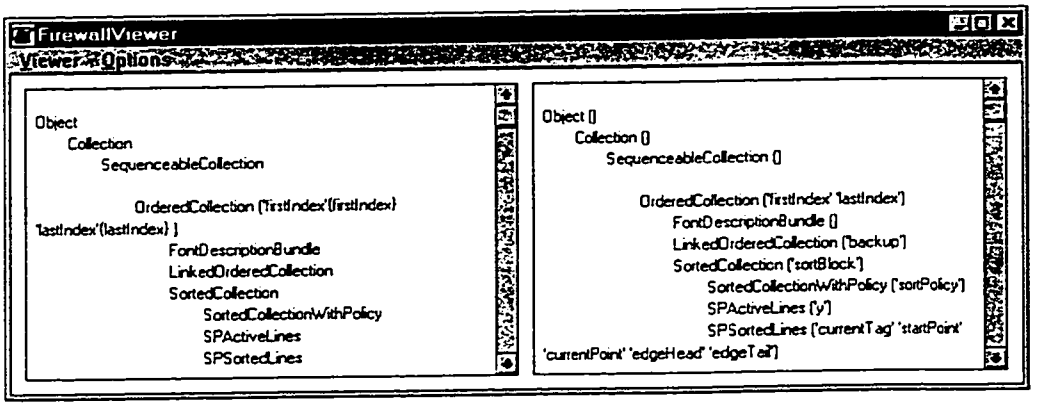


Figure 12: ORG for class OrderedCollection, as determined by *FireWorks*.

3.3 *FireWorks* as an Example

As a way of describing the design of *FireWorks*, this example presents *FireWorks* in terms of itself. That is, the ORGs for the various classes that constitute *FireWorks* are presented in detail.

The CRD for *FireWorks* is slightly more complicated then the previous examples (see Figure 13). This is to be expected, as *FireWorks* is a fully functional application. Indeed, of all the examples presented, this example best represents the complexity of OO systems that *FireWorks* was intended to analyze.

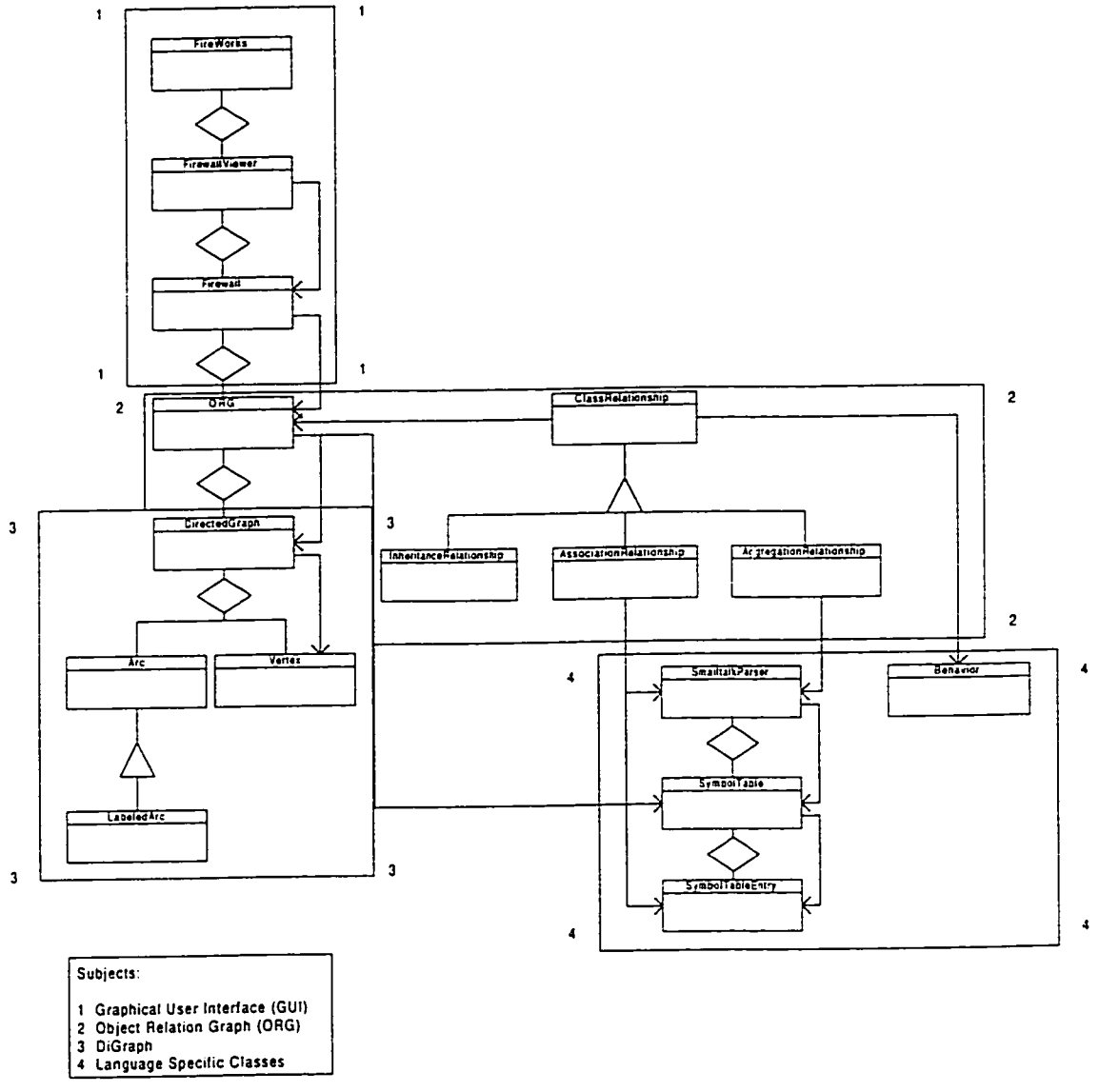


Figure 13: Class relationship diagram for *FirelWorks* in OMT.

Notice the use of subjects within the CRD to help aid in the understanding of the design of *FirelWorks*. The various subjects of *FirelWorks* include:

1. Graphical User Interface (GUI)
2. Object Relation Graph (ORG)
3. DiGraph
4. Language Specific Classes

The grouping of collaborating classes to form subjects is a technique often employed during OO analysis and OO design (Coad and Yourdon, 1991, 1992). The subject groupings on the CRD help lead the reader from one cluster of classes to another. Each of the above subjects will now be discussed in detail.

3.3.1 GUI Subject

The GUI subject represents the graphical user interface (GUI) to *FireWorks* (see Figure 13). The GUI subject is composed of the following classes:

- FireWorks, the user application
- FirewallViewer, to view firewalls
- Firewall, to construct firewalls

Comparing the CRD of *Fireworks* (see Figure 13) to the ORGs of the GUI subject reveals that *FireWorks* was able to correctly identify most of the relationships of the GUI subject. In particular, the following relationships were correctly identified:

- class FirewallViewer is an aggregate class for class Firewall (see Figure 15)

- class Firewall is an aggregate class for class ObjectRelationGraph (see Figure 16)
- class Firewall is associated with class FirewallViewer (see Figure 16)

However, *FireWorks* was unable to detect that class FireWorks is an aggregate class for a collection of class FirewallViewer (see Figure 14). The best that *FireWorks* could do was to determine that class FireWorks is an aggregate class of class OrderedCollection. While this is true, it is of more use to know precisely what instances of classes are placed within the ordered collection. The name of the instance variable, *firewallViewers*, seems to suggest a collection of many instance of class FirewallViewer. Naming an instance variable in its plural form is considered proper coding practice (Skublics *et al.*, 1996). However, a programmer is not forced by Smalltalk to follow this common coding practice. Indeed, the instance variable could have been named *x*, thereby offering no indication as to the type of class. Section 4.3.1.1 Difficulties in Detecting Aggregation Relationships presents the problem in more detail.

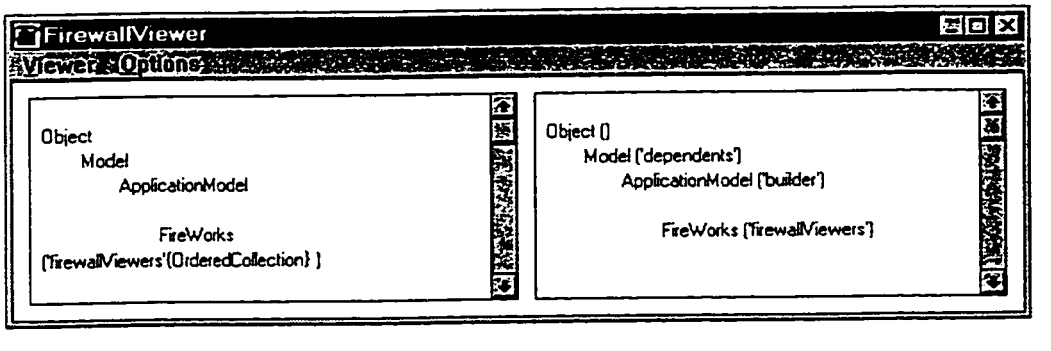


Figure 14: ORG for class FireWorks, as determined by *FireWorks*.

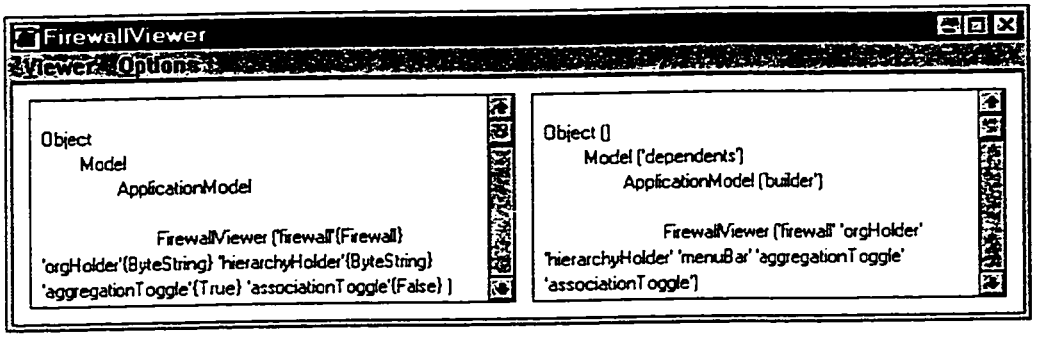


Figure 15: ORG for class FirewallViewer, as determined by *FireWorks*.

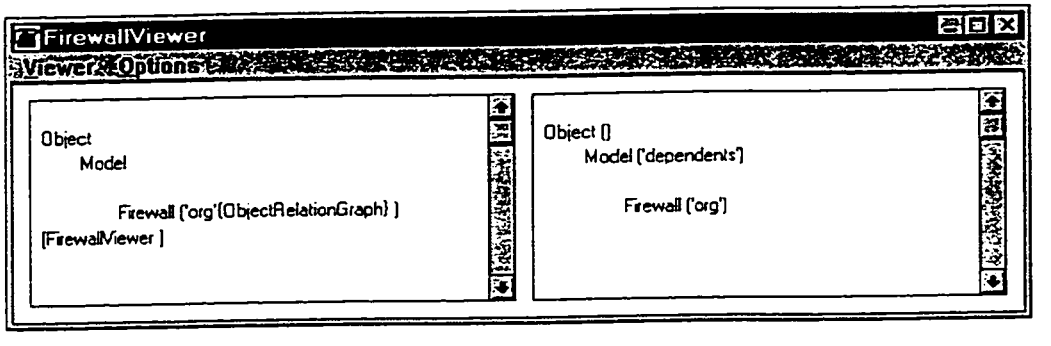


Figure 16: ORG for class Firewall, as determined by *FireWorks*.

3.3.2 ORG Subject

The ORG subject represents the object relation graph used by *FireWorks* (see Figure 13). The classes contained in the ORG subject include the following:

- ObjectRelationGraph
- ClassRelationship
- AggregationRelationship
- AssociationRelationship
- InheritanceRelationship

Comparing *FirelWorks'* CRD (see Figure 13) to the ORGs of the ORG subject indicates that *FirelWorks* was able to correctly identify the following relationships:

- class ObjectRelationGraph is an aggregate class for class DirectedGraph (see Figure 17)
- class ObjectRelationGraph is associated with class Firewall (see Figure 17)
- inheritance hierarchy of classes ClassRelationship, AggregationRelationship, AssociationRelationship, and InheritanceRelationship (see Figure 18, Figure 19, Figure 20, and Figure 21 respectively)

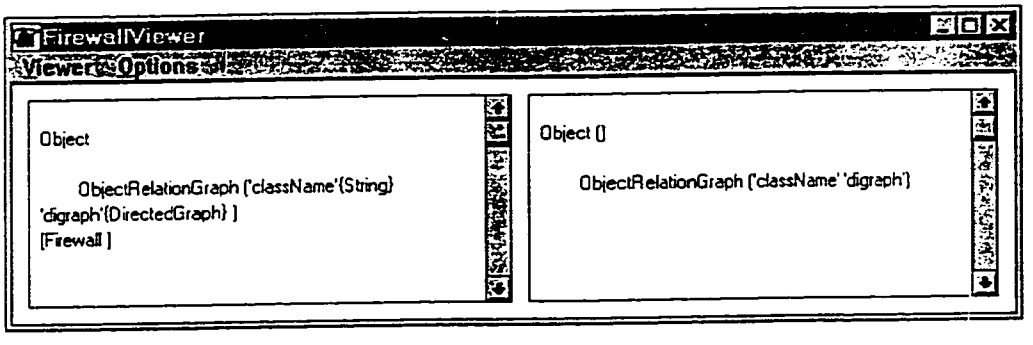


Figure 17: ORG for class ObjectRelationGraph, as determined by *FirelWorks*.

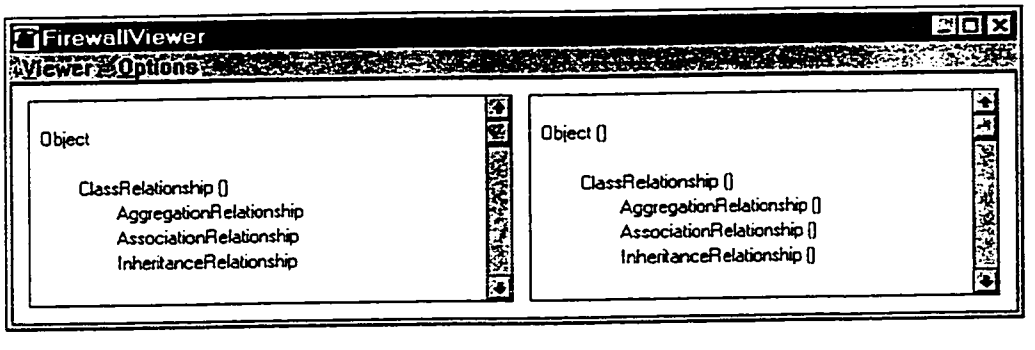


Figure 18: ORG for class ClassRelationship, as determined by *FirelWorks*.

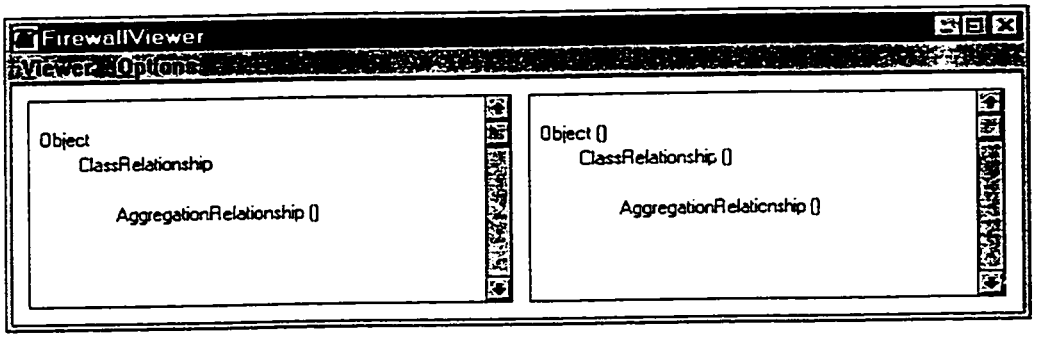


Figure 19: ORG for class AggregationRelationship, as determined by *FireWorks*.

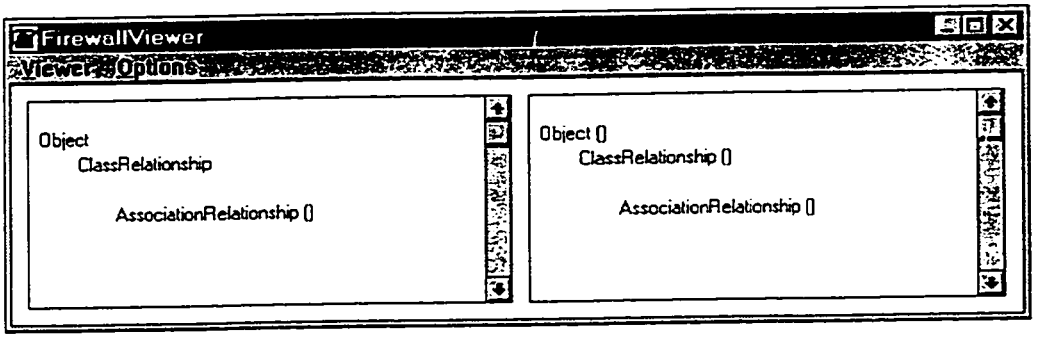


Figure 20: ORG for class AssociationRelationship, as determined by *FireWorks*.

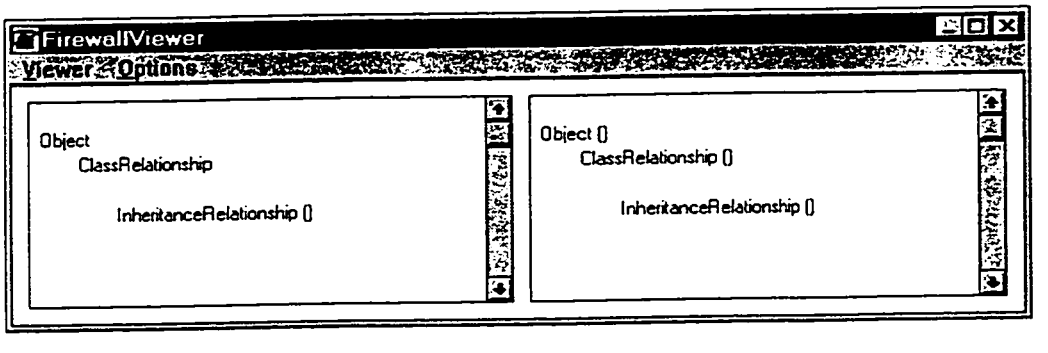


Figure 21: ORG for class InheritanceRelationship, as determined by *FireWorks*.

3.3.3 DiGraph Subject

The DiGraph subject represents the directed graph and its associated classes (see Figure 13). The DiGraph subject is composed of the following classes:

- DirectedGraph
- LabeledArc
- Vertex

Comparing the CRD of *Fireworks* (see Figure 13) to the ORGs of the DiGraph subject reveals that *FireWorks* was able to correctly identify most of the relationships of the DiGraph subject. In particular, the following relationships were correctly identified:

- class DirectedGraph is associated with class ObjectRelationGraph (see Figure 22)
- inheritance hierarchy of classes Edge, Arc, and LabeledArc (see Figure 23)
- class Vertex is associated with class DirectedGraph (see Figure 24)

FireWorks was unable to determine that class DirectedGraph is an aggregate class for a collection of class Arc, and a collection of class Vertex (see Figure 22). *FireWorks* identified class DirectedGraph as an aggregate class of classes IdentityDictionary and Dictionary. See section 4.3.1.1 Difficulties in Detecting Aggregation Relationships for further explanation of this problem. In reality, an identity dictionary was used to store all the arcs of the directed graph, and a dictionary was used to store all the vertices.

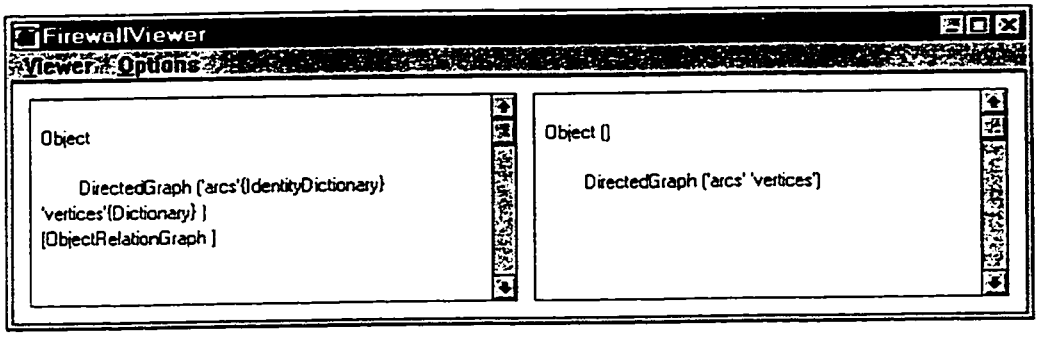


Figure 22: ORG for class DirectedGraph, as determined by *FireWorks*.

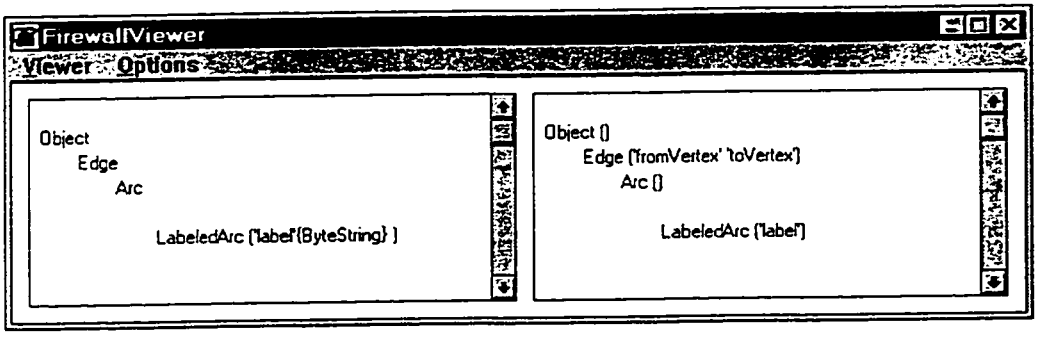


Figure 23: ORG for class LabeledArc, as determined by *FireWorks*.

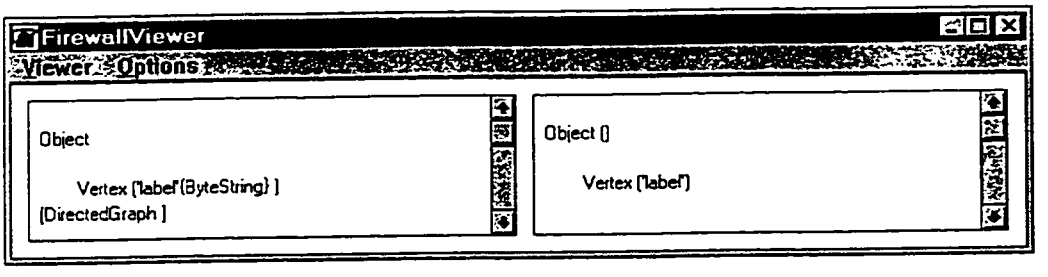


Figure 24: ORG for class Vertex, as determined by *FireWorks*.

3.3.4 Language Specific Classes Subject

The Language Specific Classes subject represents those classes that are dependent on the language used to implement *FireWorks* (see Figure 13). The classes contained in this subject include the following:

- SmalltalkParser
- SymbolTable
- SymbolTableEntry

Comparing *FireWorks'* CRD (see Figure 13) to the ORGs of the Language Specific Classes subject indicates that *FireWorks* was able to correctly identify most of the relationships of this subject. In particular, the following relationships were correctly identified:

- class SmalltalkParser is an aggregate class for class SymbolTable (see Figure 25)
- class SymbolTable is associated with classes ObjectRelationGraph and SmalltalkParser (see Figure 26)
- class SymbolTableEntry is associated with classes Association and SymbolTable (see Figure 27)

However, *FireWorks* was unable to detect that class SymbolTable is an aggregate class for a collection of class SymbolTableEntry (see Figure 26). The best that *FireWorks* could do was to determine that class SymbolTable is an aggregate class of class Dictionary. While this is true, it is of more use to know precisely what instances of classes are placed within the dictionary. In reality, the dictionary is

used to store symbol table entries. Refer to section 4.3.1.1 Difficulties in Detecting Aggregation Relationships for further explanation.

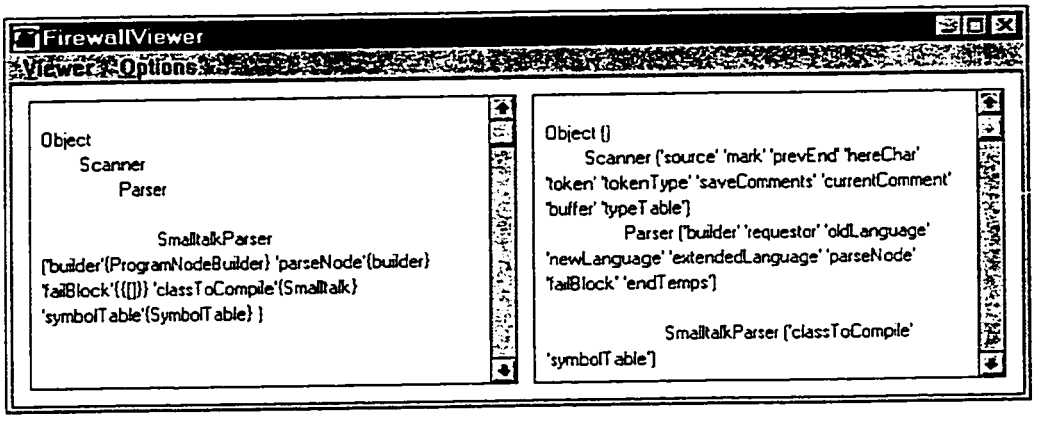


Figure 25: ORG for class SmalltalkParser, as determined by *FireWorks*.

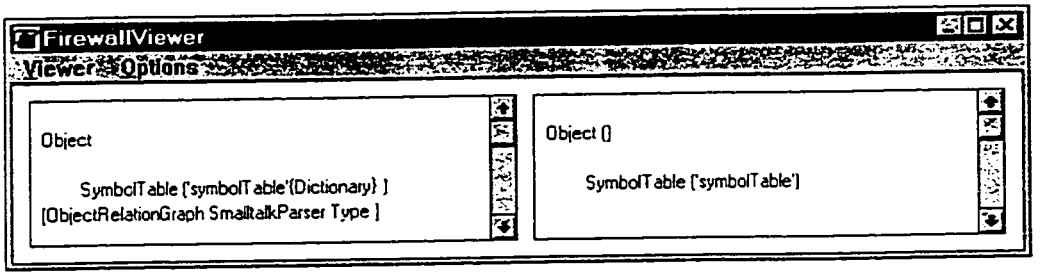


Figure 26: ORG for class SymbolTable, as determined by *FireWorks*.

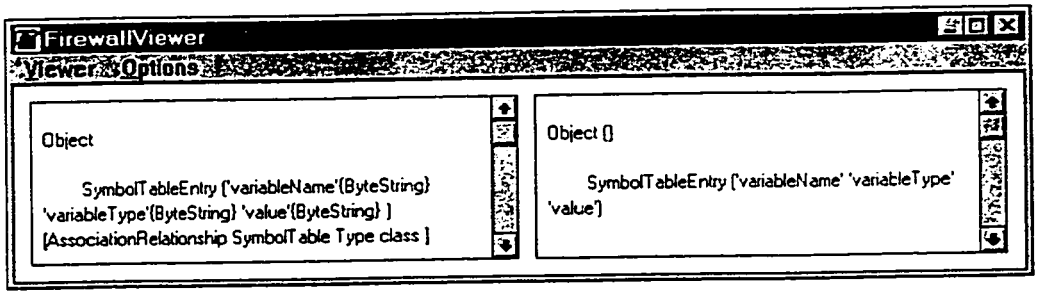


Figure 27: ORG for class SymbolTableEntry, as determined by *FireWorks*.

Chapter 4

DESIGN AND IMPLEMENTATION DETAILS

How *FireWorks* Works

This chapter presents the design and implementation details of *FireWorks*.

The design of *FireWorks* is based on the theory[18] of constructing firewalls. Briefly, a firewall attempts to isolate a class and all of its directly related classes from an entire system of classes. The design of *FireWorks* was implemented in the object–oriented programming language Smalltalk. Specifically, ParcPlace–Digitalk's VisualWorks release 2.5 was used.

The following sections describe in detail the design of *FireWorks*. Each section typically presents the algorithm used for implementation, followed by a detailed explanation of the algorithm by means of an example.

4.1 Constructing Firewalls

FireWorks employs the technique of constructing software firewalls[19] for a given class. The theoretical basis for constructing firewalls is presented in section 2.3

---

[18] Refer to section 2.3 Theory of Firewalls (page 16) for a more thorough explanation on the theory of constructing firewalls.

[19] Not to be confused with hardware firewalls, which provide a level of security for computer networks.

Theory of Firewalls. In essence, a firewall for a given class isolates that class and all of its related classes from an entire system of classes.

The essential component to a firewall is its object–relation graph (ORG). The main responsibility of a firewall it to behave as the manager to its ORG. The firewall instructs the ORG when to add various class relationships, when to remove various class relationships, and when to display itself. Kung *et al.* (1995a) describe other firewall responsibilities, such as determining the test order for retesting classes within an ORG. Figure 28 shows the ORG for class Car. Notice that each class has been assigned a number. The numbers indicate the recommended traversal order that could be followed for class retesting. The test order reduces time by serving as a guide in class retesting and class reintegration testing.

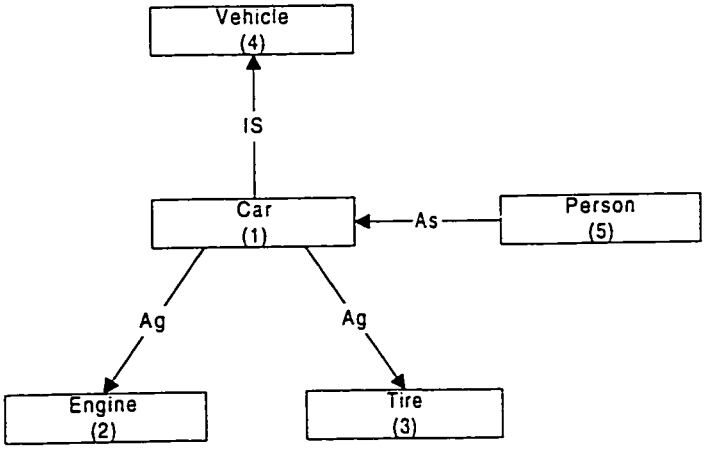


Figure 28: ORG for class Car showing test order.

At this time, *FireWorks* does not yield the traversal order for testing classes within an ORG. However, given that ORGs are implemented using directed graph data structures, determining the traversal order of an ORG would involve implementing a directed graph traversal order.

Figure 29 presents the algorithm for constructing firewalls.

For a given class:

1. Add all inheritance relationships of given class to firewall's ORG.
2. Add all aggregation relationships of given class to firewall's ORG.
3. Add all association relationships of given class to firewall's ORG.

Figure 29: Algorithm for constructing firewalls.

The algorithm is simple and straightforward: add each of the class relationships sequentially to the firewall's ORG. The sequential ordering of which class relationship is added to the ORG does not matter. Figure 29 lists the adding of class relationships in increasing order of time. That is, adding all inheritance relationships to an ORG takes the least amount of time; adding all association relationships to an ORG takes the most amount of time. Once a firewall for a specific class has been built, the firewall will instruct the ORG to display itself.

By way of demonstrating the algorithm of Figure 29, consider the Car example.

Step 1 adds all inheritance relationships for class Car. See Figure 30.

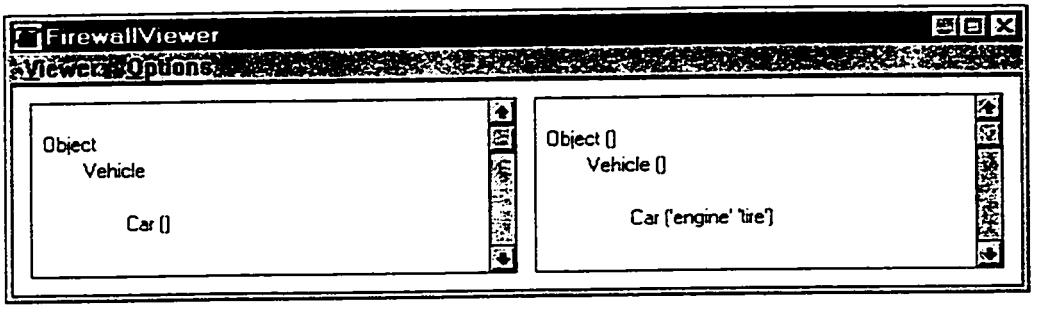


Figure 30: ORG showing all inheritance relationships for class Car.

Step 2 adds all aggregation relationships for class Car. See Figure 31.

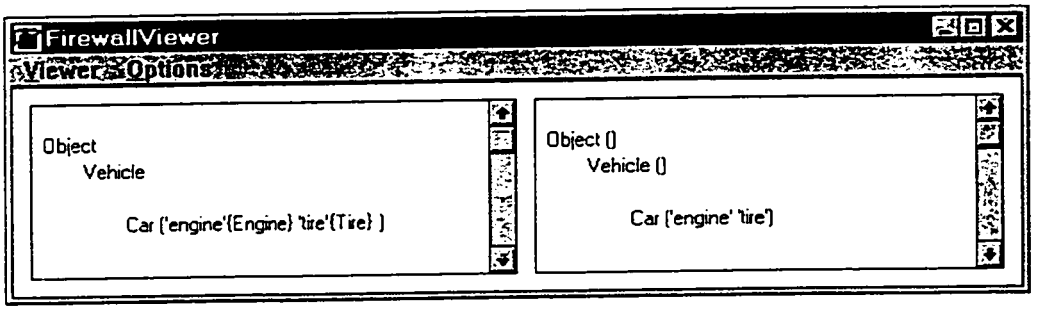


Figure 31: ORG showing all aggregation relationships for class Car.

Finally, step 3 adds all association relationships for class Car. See Figure 32.

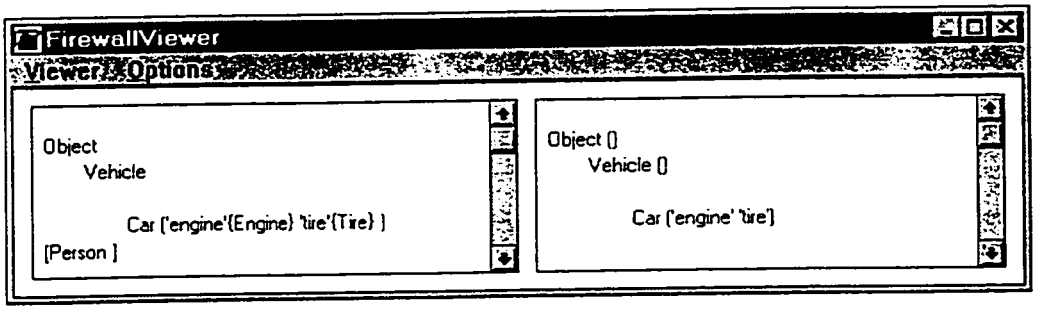


Figure 32: ORG showing all association relationships for class Car.

4.2 Constructing Object Relation Graphs

The essential data structure of *FirelWorks* is the ORG. ORGs essentially represent the various types of relationships that can occur between classes. ORGs ultimately rely on directed graphs as the fundamental data structure for representing class relationships. The next three sections describe how to add each of the class relationships to an ORG's directed graph.

4.2.1 Adding Aggregation Relationships

Figure 33 presents the algorithm for adding aggregation relationships to an ORG's directed graph.

For a given class:

1. Add a vertex labeled with given class's name to ORG's directed graph.
2. Detect all instance variables of given class.
3. For each instance variable:
   - 3.1 Add a vertex labeled with instance variable to ORG's directed graph.
   - 3.2 Connect instance variable vertex to given class vertex as an aggregation relationship.

Figure 33: Algorithm for adding aggregation relationships to an ORG's directed graph.

Consider the Car example to demonstrate the algorithm of Figure 33.

Step 1 adds a vertex, labeled *Car*, to the directed graph. Figure 34 shows the Car vertex.

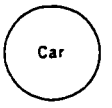


Figure 34: Car vertex.

Step 2 detects all instance variables of class Car. The result of step 2 is an ordered collection of classes: OrderedCollection(Engine, Tire). See section 4.3.1 Detecting Aggregation Relationships for further explanation of how the instance variables of class Car were detected.

The first iteration of step 3.1 adds a vertex, labeled *Engine*, to the directed graph. Step 3.2 connects the Engine vertex to the Car vertex as an aggregation relationship. See Figure 35.

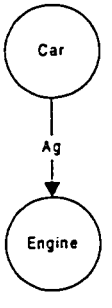


Figure 35: Result of adding Engine aggregation to directed graph.

The second iteration of step 3.1 adds a vertex, labeled *Tire*, to the directed graph. Step 3.2 connects the Tire vertex to the Car vertex as an aggregation relationship. See Figure 36.

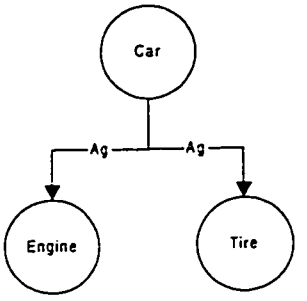


Figure 36: Result of adding Tire aggregation to directed graph.

4.2.2 Adding Association Relationships

Figure 37 presents the algorithm for adding associations relationships to an ORG's directed graph.

For a given class:

1. Add a vertex labeled with given class's name to ORG's directed graph.
2. Detect all classes that send any of given class's class methods
3. For each sender:
   3.1 Add a vertex labeled with sender's name to ORG's directed graph.
   3.2 Connect sender vertex to given class vertex as an association relationship.

Figure 37: Algorithm for adding association relationships to an ORG's directed graph.

Consider the Car example of the last section to demonstrate the algorithm of Figure 37.

Step 1 adds a vertex, labeled *Car*, to the directed graph. As the Car vertex has been previously added, this step is complete.

Step 2 detects all senders of any of class Car's class methods. The result of step 2 is an ordered collection of classes: OrderedCollection(Person). See section 4.3.2 Detecting Association Relationships for further explanation of how the senders of class Car's class methods were detected.

The only iteration of step 3.1 adds a vertex, labeled *Person*, to the directed graph. Step 3.2 connects the Person vertex to the Car vertex as an association relationship. See Figure 38.

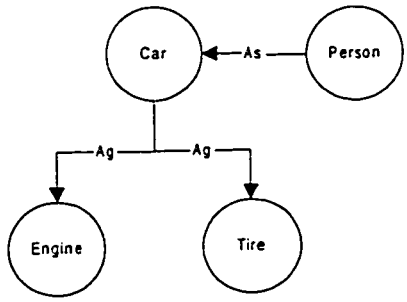


Figure 38: Result of adding Person association to directed graph.

4.2.3 Adding Inheritance Relationships

Figure 39 presents the algorithm for adding inheritance relationships to an ORG's directed graph.

For a given class:

1. Add a vertex labeled with given class's name to ORG's directed graph.
2. Detect all superclasses for given class.
3. Detect all subclasses for given class.
4. For each superclass:
   - 4.1 Add a vertex labeled with superclass's name to ORG's directed graph.
   - 4.2 Connect superclass vertex to given class vertex as an inheritance (superclass) relationship.
5. For each subclass:
   - 5.1 Add a vertex labeled with subclass's name to ORG's directed graph.
   - 5.2 Connect subclass vertex to given class vertex as an inheritance (subclass) relationship.

Figure 39: Algorithm for adding inheritance relationships to an ORG.

Consider the Car example of the previous section to demonstrate the algorithm of Figure 39.

Step 1 is complete, as the Car vertex has been previously added to the directed graph.

Step 2 detects all superclasses of class Car. The result of step 2 is an ordered collection of classes: OrderedCollection(Vehicle, Object). See section 4.3.3 Detecting Inheritance Relationships for further explanation of how the superclasses of class Car were detected.

Step 3 detects all subclasses of class Car. The result of step 3 is an empty collection of classes, as there are no subclasses of class Car. See section 4.3.3

Detecting Inheritance Relationships for further explanation of how the subclasses of class Car were detected.

The first iteration of step 4.1 adds a vertex, labeled *Vehicle*, to the directed graph. Step 4.1 connects the Vehicle vertex to the Car vertex as an inheritance (superclass) relationship. See Figure 40.

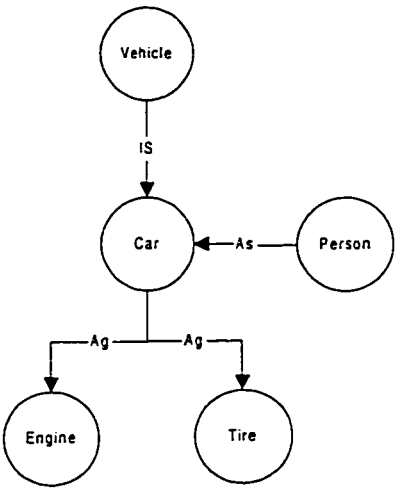


Figure 40: Result of adding superclass Vehicle to directed graph.

The second iteration of step 4.1 adds a vertex, labeled *Object*, to the directed graph. Step 4.2 connects the Object vertex to the Car vertex as an inheritance (superclass) relationship. See Figure 41.

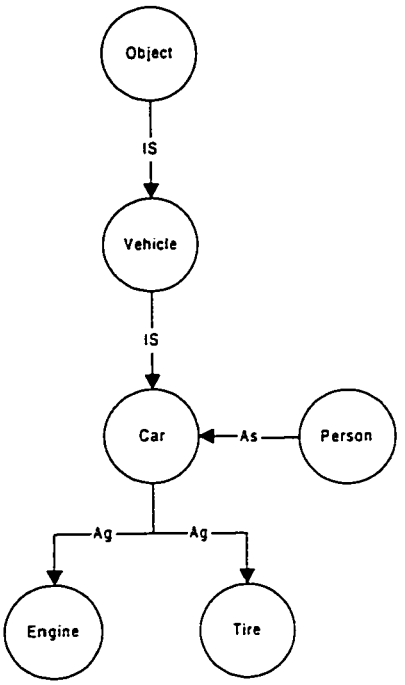


Figure 41: Result of adding superclass Object to directed graph.

Step 5 is not performed, as there are no subclasses of class Car.

4.2.4 Representing Object Relation Graphs

ORGs rely on directed graphs to represent the aggregation, association, and inheritance relationships between classes.

Formally, a directed graph is an ordered triple (V, A, f) where (Gersting, 1987, p. 162)

V = a set of vertices
A = a set of arcs
f is a function associating with each arc a an ordered pair (x, y) of vertices where x is the initial point and y is the terminal point of a.

In a directed graph, then, there is a direction associated with each arc. A path from vertex n_0 to n_k is a sequence $n_0, a_0, n_1, a_1, \ldots, n_{k-1}, a_{k-1}, n_k$ where for each i, n_i is the initial point of arc a_i and n_{i+1} is the terminal point of a_i. If a path exists from node n_0 to node n_k, then n_k is reachable from n_0.

Regarding ORGs, the above definition of a directed graph now becomes

V = a set of vertices representing various classes
A = a set of arcs representing various class relationships
f is a function mapping each class relationship to an ordered pair of classes.

For example, Figure 42 shows the directed graph for the aggregation relationship between class Car and class Engine.

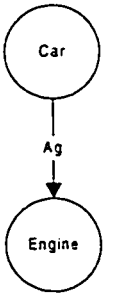


Figure 42: Directed graph representation of an aggregation relationship between class Car and class Engine.

From the example of Figure 42,

$$V = Set\{Car, Engine\}$$
$$A = Set\{Ag\}$$

f is a function that maps Ag to the ordered pair $(Car, Engine)$.

Precisely, there is an aggregation relationship *from* class Car *to* class Engine. This indicates that class Car is an aggregate class of class Engine. Note, however, that the aggregation relationship is strictly from class Car to class Engine, and that the reverse relationship does not exist.

The Smalltalk implementation of a directed graph consists of a dictionary with an entry for each vertex. Associated with each vertex is the set of arcs directed from the vertex. An arc consists of an ordered pair of vertices: a *from vertex*, and a *to vertex*.

Figure 43 shows the dictionary representation for the directed graph of Figure 42.

Key	Value
Car	$Set\{\}$
Engine	$Set\{Car - Ag- > Engine\}$

Figure 43: Dictionary representation of a directed graph.

4.3 Detecting Class Relationships

FireWorks is able to detect the following three types of relationships that can occur between classes:

1. Aggregation relationships.
2. Association relationships.
3. Inheritance relationships.

The techniques employed to detect each of the above class relationships is varied. The next three sections describe how to detect the above class relationships, starting with detecting aggregation relationships.

4.3.1 Detecting Aggregation Relationships

This section describes how to detect aggregate relationships between classes. Figure 44 presents the algorithm for detecting aggregation relationships[20]. Figure 45 presents the Smalltalk definition of class Car.

```
For a given class:

1. Collect all instance variables of given class.
2. For each instance variable:
   2.1 Collect all method selectors that write to instance variable.
   2.2 For each method selector:
      2.2.1 Perform syntax analysis on method code.
      2.2.2 Add all parsed symbols to a symbol table.
3. Return a symbol table of instance variables expressed as symbols.
```

Figure 44: Algorithm for detecting aggregation relationships.

```
Vehicle subclass: #Car
        instanceVariableNames: 'engine tire '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Relationships-Test'
```

Figure 45: Smalltalk definition of class Car.

[20] Appendix B, section B.2 Class Relationship Diagram for Subject ORG (page 117) presents the Smalltalk implementation.

Figure 46 shows class Car's instance method *initialize*.

```
Class: Car
Selector: initialize

initialize

  "Initialize all instance variables of a new instance of class Car."

 engine := Engine new.
 tire := Tire new.
```

Figure 46: Class Car's instance method initialize.

Consider the Smalltalk code fragment of Figure 46 to demonstrate the aggregation relationship detection algorithm of Figure 44.

Now apply the algorithm of Figure 44 to the code fragment of Figure 46.

Step 1 collects all instance variables for class Car. This step results in the following ordered collection: OrderedCollection(engine, tire). Consult Figure 45 to confirm the proper contents of the ordered collection.

The first iteration of step 2 considers the instance variable *engine*.

Step 2.1 collects all method selectors that write to the instance variable *engine*. This step results in the following collection: OrderedCollection(initialize).

The first iteration of step 2.2 examines the instance method *initialize* of class Car. Steps 2.2.1 and 2.2.2 result in adding an engine symbol to the symbol table. The

engine symbol will have a value of *Engine* (i.e. the class Engine) as its type. Thus, the aggregate *engine* has a type of class Engine.

The second, and last, iteration of step 2 considers the other instance variable *tire*. The steps for instance variable tire are very similar to those preceding steps for instance variable *engine*. The main difference being that a tire symbol is added to the symbol table. The tire symbol will have a value of *Tire* (i.e. the class Tire) as its type. Thus, the aggregate *tire* has a type of class Tire.

Step 3 returns a symbol table containing two symbols: the engine symbol, and the tire symbol

An important assumption is made regarding the detection of aggregation relationships. The algorithm of Figure 44 only considers instance variables as aggregations. Variables such as class variables, class–instance variables, pool variables, or global variables are not considered. Therefore, no attempt is made to detect these other types of variables. Figure 44 could be modified to detect aggregation relationships based on variables other than instance variables. Modifying the algorithm of Figure 44 would require changing step 1 to collect the appropriate variables, say class variables, for a given class. Also, step 3 would have to be modified to return a symbol table containing symbols of the supported variables in addition to the instance variable symbols. However, the vast majority

of aggregation relationships can be discovered by examining instance variables alone (Kung *et al.* 1995).

4.3.1.1 Difficulties in Detecting Aggregation Relationships

Determining the true type of an aggregate based on a collection class proves to be very difficult in a typeless OOPL, such as Smalltalk.

Figure 47 shows the code fragment for class Person's instance method *initialize*. Notice the instance variable *fleetOfCars*, which is assigned a new instance of class OrderedCollection.

```
Class: Person
Selector: initialize

initialize

  "Initialize all instance variables of a new instance of class Person."

 fleetOfCars := OrderedCollection new.
```

Figure 47: Class Person's instance method initialize.

Figure 48 shows the code fragment for class Person's instance method *acquireNewCar*.

```
Class: Person
Selector: acquireNewCar

acquireNewCar: aCar

 "Add aCar to self's (i.e. a Person) collection of cars. Answer aCar."

 fleetOfCars add: aCar.
 ^aCar
```

Figure 48: Class Person's instance method acquireNewCar.

FireWorks, following the aggregation detection algorithm of Figure 44, will detect *fleetOfCars* as an aggregate of class Person. Further, the aggregate *fleetOfCars* will have a type of class OrderedCollection. Strictly speaking, *FireWorks* has properly detected that *fleetOfCars* is of type class OrderedCollection, as *fleetOfCars* truly is assigned an instance of class OrderedCollection (see Figure 47).

Ideally, it would be useful information to know precisely what type of class is being collected by aggregate *fleetOfCars*. One may assume from the name *fleetOfCars* that the instance variable is a collection of cars. Naming an instance variable in its plural form is considered proper coding practice (Skublics *et al.*, 1996). However, a programmer is not **forced** by Smalltalk to follow this common coding practice. Indeed, the instance variable could have been named *x*, thereby offering no indication as to the type. *FireWorks* does not attempt to enforce a naming convention for instance variables. It was felt that such a constraint would restrict, and thereby limit Smalltalk as a truly typeless language.

Moreover, *FireWorks* was intended to be an automated tool with no manual intervention. Prompting a programmer to enforce the proper naming of variables is not the task of *FireWorks*.

The next section describes type inference as a possible solution to determining the type of class being collected by aggregates.

4.3.1.2 Use of Type Inference

This section describes type inference as a possible solution to the problem of determining the type of class being collected by aggregates.

Type inference attempts to infer the types of classes in an untyped OOPL, such as Smalltalk (Goldberg and Robson, 1983). The basic purpose of doing type inference for untyped object–oriented programs is to guarantee that all messages are understood (Borning and Ingalls, 1982).

Suzuki (1981) was the first to address the problem of type inference. However, his algorithm was not capable of checking most common programs written in Smalltalk. Next, Graver and Johnson (1989 and 1990) provided an algorithm for a simplified problem, where the types of instance variables must be specified by the programmer so that only the types of arguments are inferred. Hense (1990) addressed the problem of inferring types that are useful in connection with separate compilation. This means that Hense's algorithm is not allowed to reconsider the program text when new classes are added to the program.

Observing the difficulties encountered by other researchers, Palsberg and Schwartzbach (1991) presented a new approach to type inference. Their idea deviated from the traditional approach of inferring type based on a single, solitary class. For example, class Person's instance variable is of type OrderedCollection, and of type OrderedCollection only. Rather, Palsberg and Schwartzbach devised an algorithm based on inferring types as finite sets of classes. Their first attempt of a type inference algorithm was sound, and could handle most common programs. Also the algorithm was conceptually simple: a set of uniform type constraints is constructed and solved by fixed–point derivation.

However, there was one severe restriction to their algorithm. The algorithm could not infer types in programs that use collection classes. A collection class is used to contain different instances in different parts of the program. For example, a class OrderedCollection may be used to collect booleans in one place, and as an integer collection in another place (Oxhoj, Palsberg, and Schwartzbach, 1992).

Oxhoj, Palsberg, and Schwartzbach (ibid.) recognized the severe limitation of their first algorithm. Significant improvements were made to the first algorithm. The improvements include:

- An improved algorithm that handles collection classes in a useful way; and

- An implementation of the new algorithm, including two novel techniques for making type inference faster and less space consuming. The complexity has been reduced from exponential time to low polynomial time.

The general idea of the algorithm is to define a *type variable* for every expression occurring in the program (Wand, 1987; Schwartzbach, 1991). Next, a collection of *constraints* on these variables is generated. Finally, an attempt is made to solve these constraints. If they are solvable, then their *minimal solution* corresponds to the inferred typing; if not, then the program is not typable (Oxhoj, Palsberg, and Schwartzbach, ibid., p. 331). A summary of the entire algorithm is shown in Figure 49.

Input: A program in the language described by Palsberg and Schwartzbach.
Output: Either: a safety guarantee and type information about all expressions; or: "unable to type the program".

1. Expand away inheritance.
2. Construct the trace graph of the expanded program.
3. Extract a set of type constraints from the trace graph.
4. Compute the least solution of the set of type constraints. If such a solution exists, then output it as the wanted type information, together with a safety guarantee; otherwise, output "unable to type the program".

Figure 49: Algorithm for type inference, by Oxhoj, Palsberg, and Schwartzbach.

Adding type inference to *FirelWorks* could possibly allow for increased accuracy in detecting aggregation relationships. In particular, type inference may help determine the element type being collected by aggregates of an instance of a specific class. More importantly, however, adding type inference to *FirelWorks*

represents a significant amount of required work. Yet the gain offered by type inference may be minimal, due in part to this observation by Agesen and Ungar (1994, p. 359): "any practical type inference algorithm must be conservative, since exact type inference is uncomputable."

4.3.2 Detecting Association Relationships

This section describes how to detect association relationships between classes.

Figure 50 presents the algorithm for detecting association relationships[21].

For a target class:
1. Collect all class method selectors for target class.
2. For each class, except target class, within the Smalltalk environment (call this the candidate class):
   - 2.1 For each class method selector of target class:
     - 2.1.1 Collect all method selectors of candidate class that refer to class method selector of target class.
     - 2.1.2 For each method selector of candidate class:
       - 2.1.2.1 Collect candidate class name if the method selector actually sends class method selector to target class.
3. Return a collection of class names that send at least one of target class's class method selectors.

Figure 50: Algorithm to detect association relationships.

Figure 51 shows class Person's instance method *buildCar*.

[21] Appendix B, section B.2 Class Relationship Diagram for Subject ORG (page 117) presents the Smalltalk implementation.

```
Class: Person
Selector: buildCar

buildCar

 "Instruct self (i.e. a Person) to build a car. Answer a new instance of class Car."

 | aCar |

 aCar := Car new.
 ^aCar
```

Figure 51: Class Person's instance method buildCar.

Consider the code fragment of Figure 51 to demonstrate the association relationship detecting algorithm.

Now apply the algorithm of Figure 50 to the code fragment of Figure 51.

Step 1 collects all class method selectors of class Car. Recall that we desire to detect all the association relationships for class Car. Therefore, class Car is the target class. Class Car only understands the *new* class method selector. Thus, the result of this step is an ordered collection of all of class Car's class method selectors: OrderedCollection(new).

Step 2 will execute for every class, except class Car, within the Smalltalk environment. On the iteration for class Person, the following substeps will be performed.

Step 2.1 is performed considering class Car's class method selector *new*.

Step 2.1.1 results in a collection of class Person's method selectors that refer to class Car's *new* class method selector. In this example, only class Person's method selector *buildCar* refers to class Car's *new* method. This step results in the following collection: OrderedCollection(buildCar).

Step 2.1.2 is only performed once, as class Person only has one method selector that refers to class Car's *new* class method selector.

Step 2.1.2.1 adds class Person to the list of classes that are associated with class Car, since method *buildCar* sends the *new* message to class Car.

Step 3 returns a sorted collection of those class names associated with class Car. The only detectable class was class Person, so this step returns: SortedCollection(Person).

The detection of association relationships has an important assumption. Only examined classes that send class method selectors can be detected. No assumption is placed on the classes to be examined. That is, both method selectors and class method selectors are considered for the examined class. The assumption is justified given that the task is to detect association relationships between classes. Given this assumption, classes that make use of variables, such

as global variables, pool variables, or blocks, to send messages may not be detected.

Notice that the algorithm of Figure 50 does not classify association relationships according to the type of dependency. Recall from section 2.3.1.2.1 Class Firewall Defined, two classes may be dependent on each other in various ways. The dependency may be based on data, state, message, or control dependency. The association relationship detection algorithm could be modified to detect association relationships according to dependency. The modification would require refining step 2.1.2.1 of Figure 50 to further classify an association relationship according to its dependency.

The astute reader may question the need for an addition check provided by step 2.1.2.1 of Figure 50. One may feel that step 2.1.2.1 is redundant, as step 2.1.1 performs a similar check. However, step 2.1.2.1 is a necessary step.

Consider the code fragment of Figure 52.

```
Class: Factory
Selector: allComment

allComment

 "This entire method is a Smalltalk comment.

  | aCar |

  aCar := Car new.
  ^aCar"
```

Figure 52: Class Factory's instance method allComment.

The code fragment of Figure 52 serves no purpose other than to demonstrate the need for step 2.1.2.1. Step 2.1.1 would identify an association relationship between class Factory and class Person, based on class Factory's instance method *allComment*. Of course, this is not an association relationship, as instance method *allComment* does not actually send the *new* class message to class Car. Step 2.1.2.1 performs this exact check. Step 2.1.2.1 verifies that the method being considered actually sends an appropriate class message to the target class. Step 2.1.2.1 relies on syntax analysis of Smalltalk code to verify the message send. See section 4.4 Performing Syntax Analysis for further details.

4.3.3 Detecting Inheritance Relationships

This section describes how to detect inheritance relationships between classes.

Figure 53 presents the algorithm for detecting inheritance relationships[22].

For a given class:

1. Collect all superclasses of given class.
2. Collect all subclasses of given class.
3. Return a collection of all superclasses and all subclasses of given class.

Figure 53: Algorithm for detecting inheritance relationships.

From Figure 53, the task of detecting inheritance relationships is made easier by breaking the problem into two subtasks:

1. Identify all the superclasses of a given class.
2. Identify all the subclasses of a given class.

Collectively, all the superclasses and all the subclasses of a class form the inheritance hierarchy for that class.

Identifying all the superclasses of a class is made easy in VisualWork's Smalltalk by using class Behavior's instance method *allSuperclasses*. *AllSuperclasses* returns an ordered collection of the class's superclass and the class's ancestor's superclasses, with the immediate superclass first.

Identifying all the subclasses of a class is made easy in VisualWork's Smalltalk by using class Behavior's instance method *allSubclasses*. *AllSubclasses* returns an

[22] Appendix B, section B.2 Class Relationship Diagram for Subject ORG (page 117) presents the Smalltalk implementation.

ordered collection of the class's subclasses and the class's descendant's subclasses in breadth–first order, with the immediate subclasses first.

Figure 54 shows the Smalltalk definition for class Car.

```
Vehicle subclass: #Car
        instanceVariableNames: 'engine tire '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Relationships-Test'
```

Figure 54: Smalltalk definition of class Car.

Now apply the algorithm of Figure 53 to the class Car definition of Figure 54.

Step 1 collects all superclasses for class Car. The following ordered collection is formulated: OrderedCollection(Vehicle, Object).

Step 2 collects all subclasses of class Car. An empty ordered collection is formulated, as class Car does not have any subclasses.

Step 3 returns an ordered collection of all superclasses and all subclasses of class Car. The following ordered collection is returned OrderedCollection(Vehicle, Object, Car)

4.4 Performing Syntax Analysis

In order to detect the various types of class relationships, *FirelWorks* requires the ability to perform syntax analysis. Also known as parsing, syntax analysis "groups

tokens into syntactic structures much as we had to structure sentences back in eighth grade" (Lemone, 1992, p. 6). Tokens are "the basic lexical units much as words and punctuation are the basic language units of an English sentence" (Lemone, ibid. p. 5). Consider this example from Lemone (ibid., p. 7). In sentence structuring:

The little boy ran quickly

has a noun phrase (*The little boy*) and a verb phrase (*ran quickly*). The verb phrase consists of substructures, a verb (*ran*) and an adverb (*quickly*). The works are analogous to the tokens found by the lexical analyzer. The noun phrase, the verb phrase, the verb, the adjective and the adverb are all analogous to the structures found by a syntax analyzer.

The parser required by *FireWorks* must be able to perform syntax analysis on messages expressed in the OOPL used to implement *FireWorks*. As *FireWorks* was implemented in the Smalltalk OOPL, the parser must be able to understand messages written in Smalltalk syntax. Assuming *FireWorks* was implemented in another OOPL, such as C++, a C++ parser would also have to be implemented in order to parse C++ statements.

The dialect of Smalltalk used to implement *FireWorks*, VisualWorks release 2.5, provides a class Parser. Class Parser is able to perform syntax analysis on

Smalltalk code specific to VisualWorks. Refer to the VisualWorks release 2.5 *User's Guide* for the specifics of class Parser (ParcPlace–Digitalk, 1995, p. 324).

The services offered by class Parser were insufficient to assist in the detection of class relationships. In particular, class Parser does not provide for the recording of symbols discovered during syntax analysis.

Yet, *FirelWorks* requires symbols in order to detect the type of class for aggregates, and to detect association relationships between classes[23]. Instead, a subclass of class Parser was defined. Class SmalltalkParser was defined specifically to extend the services of class Parser in order to assist in the detection of class relationships.

Class SmalltalkParser extends class Parser in three ways:

- Addition of a symbol table.
- Extended instance method *assignment:startingAt:*.
- Extended instance method *messagePart:repeat:*.

An explanation for each of the above extensions follows.

4.4.1 Addition of a Symbol Table

Class SmalltalkParser extends class Parser by adding an instance of class SymbolTable as an instance variable.

---

[23] Refer to sections 4.3.1 Detecting Aggregation Relationships (page 63), and 4.3.2 Detecting Association Relationships (page 71).

A symbol table records and collects information about each symbol discovered during syntax analysis. A symbol associates a token with its value.

For example, consider this Smalltalk assignment expression:

engine := Engine new.

During syntax analysis, an engine symbol will be recorded in the symbol table. The token name is the string *engine*. The engine symbol has a value of *Engine*, and is determined to be an instance variable. The following sections elaborate exactly how the engine symbol was discovered.

4.4.2 Instance Method assignment:startingAt:

Class SmalltalkParser extends class Parser by extending the instance method *assignment:startingAt:*. The extended method is able to express assignment messages as symbols within a symbol table.

The extended version of *assignment:startingAt:* assists in the detection of aggregation relationships[24].

For example, consider class Car's instance method *initialize* (see Figure 55).

---

[24] Refer to section 4.3.1 Detecting Aggregation Relationships (page 63).

```
Class: Car
Selector: initialize

initialize

  "Initialize all instance variables of a new instance of class Car."

 engine := Engine new.
 tire := tire
```

Figure 55: Class Car's instance method initialize.

When considering aggregation relationships, *FireWorks* is able to detect aggregate *engine* of type class Engine, and aggregate *tire* of type class Tire.

The method *assignment:startingAt:* assists in detecting aggregates by reducing class Car's *initialize* method into symbols. Both tokens *engine* and *tire* are reduced to variable symbols. The assignment token (:=) is reduced to an assignment symbol. And the expressions *Engine new* and *Tire new* are reduced to assignment expressions. Both assignment expressions are recognized as message sends. The receivers of the messages, *Engine* and *Tire*, become the values for symbols engine and tire respectively.

4.4.3 Instance Method messagePart:repeat:

Class SmalltalkParser extends class Parser by extending the instance method *messagePart:repeat:*. The extended method is able to express messages as symbols within a symbol table.

The extended version of *messagePart:repeat:* assists in the detection of association relationships[25]. This version of the instance method is required to confirm the sending of a class method from one class to another. Confirming the sending of a class method between classes is the last required step in detecting association relationships[26].

For example, consider class Person's instance method *buildCar* (see Figure 56).

```
Class: Person
Selector: buildCar

build

 "Instruct self (i.e. a Person) to build a Car.  Answer a new instance of class Car."

 | aCar |

 aCar := Car new.
 ^aCar
```

Figure 56: Class Person's instance method buildCar.

When considering association relationships, *FireWorks* is able to detect that class Person forms an association relationship with class Car.

The method *messagePart:repeat:* assists in detecting associations by reducing class Person's *buildCar* method into symbols. The token *aCar* is reduced to a variable symbol. The assignment token (:=) is reduced to an assignment symbol. And the

[25] Refer to section 4.3.2 Detecting Association Relationships (page 71).

expression *Car new* is reduced to an assignment expression. The assignment expression is recognized as a message. The *Car new* message is detected as a unary message, with *Car* being the receiver of the unary selector *new*. The method *messagePart:repeat:* confirms that the message *new* is to indeed be sent to class Car, thereby forming a class association relationship between classes Person and Car.

Consider the consequences of not extending the method *messagePart:repeat:* to confirm the sending of class methods between classes.

Suppose that class Factory had an instance method named *buildCarStub*. See Figure 57.

```
Class: Factory
Selector: buildCarStub

buildCarStub

  "Stub for self (i.e. a Factory) to build a new Car."

 ^self
```

Figure 57: Class Factory's instance method buildCarStub.

Notice the words *Car* and *new* within the Smalltalk comment. The words *Car* and *new* could be recognized as a pattern by a pattern matcher. Algorithms relying on pattern matching for detecting association relationship may incorrectly identify an association relationship between class Factory and class Car. The extended

[26] See Figure 50 (page 71). In particular, notice step 2.1.2.1.

version of *messagePart:repeat:* would verify that a *new* message was not sent to class Car. As class Factory does not send a class method to class Car, no association relationship would exist between the two classes.

Chapter 5

THE CONTRIBUTION OF *FireWorks* AS AN OBJECT–ORIENTED REGRESSION TESTING TOOL

Current State and Future Work for *FireWorks*

This chapter summarizes the various contributions of *FireWorks* as a tool for regression testing of object–oriented systems. Firstly, the current state of *FireWorks* is described. Next, the various strengths and weaknesses of *FireWorks* are discussed. Finally, the future for *FireWorks* is pondered.

5.1 Current State of *FireWorks*

FireWorks was implemented as a complete application within the Smalltalk environment. Specifically, ParcPlace–Digitalk's VisualWorks release 2.5 was used as the object–oriented programming language.

As a complete, working application, *FireWorks* is composed of the following four components:

1. Graphical User Interface (GUI)
2. Object Relation Graph (ORG)
3. DiGraph
4. Language Specific Classes (LSC)

The GUI component includes the actual windows, text panes, buttons, and other widgets that form the graphical interface to *FireWorks*. The classes of the GUI

component are specific to VisualWorks, and would have to be re–implemented in other Smalltalk dialects.

The ORG component serves as the model for *FireWorks*. The major responsibility of the ORG component is to construct object relation graphs. The ORG component includes classes that are able to detect the three possible relationships between classes: aggregation, association, and inheritance. The ORG component is not dependent upon any specifics of the VisualWorks version of Smalltalk. As such, the classes of the ORG component could be easily implemented in other dialects of Smalltalk.

The DiGraph component implements the directed graph class, along with the necessary supporting classes. The ORG component uses a directed graph as the fundamental data structure in representing the relationships between classes. The classes of the DiGraph component are independent of VisualWorks' Smalltalk, and as such, could be easily implemented in other dialects of Smalltalk.

The Language Specific Classes (LSC) component represents those classes dependent upon the object–programming language. The LSC component contains two subcomponents: a parser subcomponent, and a behavior subcomponent. The parser subcomponent is responsible for parsing classes within the implementation language. In this particular version of *FireWorks*, the implementation language is Smalltalk. The parser subcomponent is used by the

ORG component to detect both aggregation and association relationships between classes. The behavior subcomponent is used by the ORG component to detect inheritance relationships between classes. The behavior subcomponent is also used to detect aggregation relationships by identifying instance variables for a class. Additionally, the behavior subcomponent helps detect association relationships by supplying class method selectors for a class.

5.2 Strengths and Weaknesses of *FireWorks*

This section discusses the various strengths and weaknesses of the current implementation of *FireWorks*.

5.2.1 Strengths

The various strengths of *FireWorks* include:

- A complete design of *FireWorks*, including a CRC design, and class relationship diagrams. See Appendices A and B respectively.
- Ability to detect each of the three possible relationships between classes: aggregation, association, and inheritance.
- A working application within the VisualWorks Smalltalk environment, complete with a graphical user interface.

5.2.2 Weaknesses

The current implementation of *FireWorks* is not without limitations. Here is a list of those limitations thought to be the most serious:

- Only instance variables are considered for aggregation relationships. Class variables, class–instance variables, pool variables, and global variables are not considered at this time.
- Regarding aggregates of a Collection class, *FireWorks* cannot reconcile what type of class is being collected. Currently, *FireWorks* can only detect the type of collection being utilized.
- Association relationships can only be detected with the explicit sending of class methods.
- Object relationship graphs are displayed in text form only. Indentation, line formatting, and delimiters are all used to represent the various relationships between classes.

Section 5.3.1 Improving *FireWorks* proposes various enhancements to *FireWorks*, some of which address the above limitations.

5.3 Future Work for *FireWorks*

Three general strategies can extend the usefulness of *FireWorks* into the future:

1. Improving *FireWorks*.
2. Porting *FireWorks* to other OOPLs.
3. Utilizing *FireWorks* as a tool in Object–Oriented Software Development

5.3.1 Improving *FireWorks*

The current implementation of *FireWorks* has certain limitations that could be improved upon. Some of the proposed improvements include:

- improve response time to detect association relationships
- concurrent construction of ORGs
- determine test order for classes
- identify affected methods and determine traversal order of methods to retest

- detect aggregation relationships based on variables other than instance variables
- attempt to resolve aggregates of a Collection class with type inference
- classify types of association relationships according to dependencies
- compute indirect relationships for a firewall
- display ORGs graphically

The algorithm for detecting association relationships currently examines every class within the Smalltalk environment[27]. Every class must be examined, as every class is potentially associated with the class under consideration. This strategy is time–consuming. Alternative strategies could be investigated to reduce the time required to examine each class. A possible strategy could involve ignoring those classes that are known not to contain any association relationships. This would require pre–determining which classes to examine for relationships, and which classes to ignore. Presumably, the set of pre–determined classes to examine would be much smaller than the total number of classes within the Smalltalk environment. Therefore, less time would be required to examine a smaller set of classes.

Similarly, the construction of ORGs could be performed concurrently instead of sequentially. Detecting the various class relationships for an ORG are independent of each other[28], and as such are well suited for a concurrent

---

[27] See Figure 50 (page 71). In particular, notice step 2.

[28] Refer to section 4.1 Constructing Firewalls (page 51).

environment. This would require implementing *FireWorks* in an OOPL that supports concurrency. Feasible candidates include Concurrent Smalltalk (Tokoro and Yonezawa, 1987; Agha *et al.*, 1993), or Java (Cornell and Horstmann, 1996), which has native support for threads.

FireWorks currently identifies those classes that should be retested once a given class has been modified. The next logical step would involve determining the test order for those identified classes[29]. The test order reduces time by serving as a guide in class retesting and class reintegration testing. As ORGs rely on directed graphs for representing class relationships, the test order for classes could be determined by implementing directed graph traversal algorithms.

FireWorks currently identifies classes related in some manner to a given class[30]. Proceeding to a finer degree of identification involves detecting which methods, both class and instance, for each identified class.

For example, consider the association relationship between class Car and class Person (see Figure 28 on page 51). Obviously, the Person class has been identified as a class that is associated with class Car. Suppose that class Person has an instance method *buildCar* (refer to Figure 51 on page 72). Part of the behavior of *buildCar* is to send the message *new* to class Car. This constitutes an association relationship between class Car and class Person. In addition to

---

[29] See Figure 28 (page 51) for an example ORG showing test order.

identifying class Person, the instance method *buildCar* could also be identified as the explicit method that is directly associated with class Car.

Additionally, the method order could also be determined. Determining the method order would be analogous to determining the class order, as previously explained.

FireWorks only considers instance variables of a class when detecting aggregation relationships[31]. *FireWorks* could be enhanced to consider other types of variables when detecting aggregation relationships. Other types of variables could include: class variables, class–instance variables, pool variables, and global variables. Considering other types of variables would presumably increase the number of aggregation relationships that could be detected by *FireWorks*. As pointed out by Kung *et al.* (1995a), the vast majority of aggregation relationships will be discovered by only considering instance variables.

FireWorks is unable to reconcile aggregates that have a type based on class Collection[32]. In an attempt to solve this problem, the use of type inference could be investigated[33]. Type inference attempts to infer the types of classes in an untyped OOPL, such as Smalltalk. Implementing type inference within *FireWorks*

---

30 *FireWorks* is able to detect aggregation, association, and inheritance relationships between classes.

31 Refer to section 4.3.1 Detecting Aggregation Relationships (page 63).

32 Refer to section 4.3.1.1 Difficulties in Detecting Aggregation Relationships (page 66).

33 Refer to section 4.3.1.2 Use of Type Inference (page 68).

could possibly allow for increased accuracy in detecting aggregation relationships based on collection classes.

One of the three possible class relationships detectable by *FirelWorks* is an association relationship. A coarse way of defining an association relationship is an attempt to classify a relationship between two classes that is neither an inheritance nor an aggregation relationship. However, *FirelWorks* could be extended to classify association relationships into various types of dependencies between two classes. From section 2.3.1.2.1 Class Firewall Defined, two classes may be dependent on each other in various ways. The dependency may be based on data, state, message, or control. Classifying association relationships according to dependencies offers no real gain to the usefulness of *FirelWorks* as a tool for regression testing. This feature would merely provide further information as to what type of association relationship was present between two classes.

FirelWorks currently isolates a class and all of its directly related classes from an entire system of classes. *FirelWorks* could be enhanced to compute the related classes for each of the related classes of the class under consideration. *FirelWorks*, then, would present both the direct as well as the indirect relationships for a class. The computation involves performing the transitive closure[34] on the firewall of directly related classes. An option could be supported in *FirelWorks* to determine what level of indirect relationships for a class the user wishes to see. In the case

of Smalltalk as the implementation language, every class is ultimately related to class Object. This is due to the ubiquity of class Object as the sole parent of all classes within the Smalltalk environment. In the case of other OOPLs as the implementation language, this enhancement may prove to be useful.

ORGs are currently displayed as formatted text. Indentation, tabulation and delimiter characters are all used to display ORGs. Future work could involve displaying ORGs as directed graphs. This is more intuitive, as ORGs already depend on directed graphs as the main data structure to represent the various relationships between classes.

5.3.2 Porting *FireWorks* to other OOPLs

C++ was chosen by Kung *et al.* (1995a and 1995b) as the implementation language for constructing firewalls. *FireWorks* was implemented in Smalltalk. Smalltalk was specifically chosen as an alternative to C++. Past researchers, such as Kung *et al.* (1995a and 1995b) and Murphy *et al.* (1993 and 1994), relied on, and thus exploited, the language properties of C++. For example, since C++ is a typed language, the ability to detect association relationships is greatly simplified. *FireWorks*, to contrast, was designed to be independent of the implementation language. *FireWorks* does, however, have a component that is dependent upon the implementation language. This language specific component is responsible for detecting various class relationships by performing syntax analysis. In general,

---

[34] Refer to section 2.3.1.2.2 Constructing a class Firewall (page 21).

FireWorks was designed to be independent of the implementation language, and as such could be implemented in any class–based, hierarchical OOPL, such as Smalltalk, C++, ObjectPascal, or Java. The design of *FireWorks*, however, is not applicable to non–class–based, nor non–hierarchical OOPLs, such as Self (Ungar and Smith, 1987).

FireWorks was initially developed in ParcPlace–Digitalk's dialect of Smalltalk, known as VisualWorks. Testing the claim that *FireWorks* is independent of language, future work could involve porting *FireWorks* to other dialects of Smalltalk, such as IBM's VisualAge for Smalltalk.

In addition to Smalltalk, other OOPLs could be chosen as an implementation language for *FireWorks*. Sun's very popular Java language could be considered. Owing to its C++ and Smalltalk roots, Java appears as an attractive compromise to C++ and Smalltalk (Cornell and Horstmann, 1996). C++ could also be considered as an implementation language. Future work could involve benchmarking a C++ version of *FireWorks* to previously built tools, such as the one built by Kung *et al.* (1995a). Metrics such as performance, and accuracy in detecting class relationships could be compared.

5.3.3 Utilizing *FireWorks* in OO Software Development

This section attempts to describe the usefulness of a tool such as *FireWorks* in the context of object–oriented software development.

5.3.3.1 What is Object–Oriented Software Development?

Corriveau (1996) observes that the popularity of an object–oriented approach to software development has lead to a multitude of methods and notations for object–oriented analysis (OOA) and design (OOD). Corriveau (ibid.) then states that "the scaling up of these methods to an industrial context continues to be problematic: there is still no commonly–accepted object–oriented process."

Corriveau (ibid.) then proposes an object–oriented software development (OOSD) process that can be scaled to large object–oriented projects. Two key facets of Corriveau's process are (ibid.):

a) the pervasive importance of *traceability*, and
b) the integration of development and testing activities in the lifecycle

Corriveau's OOSD process should be viewed as evolutionary rather than revolutionary. Indeed, OOSD attempts to extend the concepts of OOA and OOD by considering several important issues, such as object–oriented testing, which are often ignored in current methodologies. Thus, OOSD attempts to be a complete process for the entire software lifecyle for OO systems.

5.3.3.2 The Role of FireWorks in OOSD

Corriveau (ibid.) stresses the need for traceability, and the integration of two separate activities: development and testing.

5.3.3.2.1 FireWorks and Object–Oriented Traceability

Traceability involves cross–referencing every artifact within the OOSD process. Having all artifacts cross–referenced allows, in theory, to quickly predict the outcome should any artifact be changed. The theory is based on *semantic dependency analysis* (SDA). SDA involves capturing the semantic dependencies between all artifacts developed during the entire *history* of a project.

FireWorks could assist in capturing the semantic dependencies between all artifacts. *FireWorks* is able to detect all of the aggregation, association, and inheritance relationships for a given class. Applying *FireWorks* to every class artifact would yield all the class relationships for all the classes within the project. If necessary, *FireWorks* could be extended to detect other types of semantic dependencies, and to detect other types of artifacts. Other artifacts, for example, could include methods (class and instance).

5.3.3.2.2 Object–Oriented Testing

Jacobson *et al.* (1992) describes several levels of testing object–oriented code:

- method
- class
- cluster
- subsystem
- system

For simplicity, Corriveau (ibid.) includes method testing into class testing, and subsystem testing into cluster testing. Corriveau (ibid.) emphasizes that "both class and cluster testing admit black and white–box techniques." Corriveau (ibid.) then surveys these techniques[35].

Corriveau (ibid.) then argues that the survey of techniques "omits a multitude of specific considerations to be addressed elsewhere." As an example, Corriveau (ibid.) observes that:

> *We need to understand how inheritance and polymorphism affect OOT. We need to elaborate on method and subsystem testing. And, most importantly, we need [to] explain how OOD views are used in OOT.*

Corriveau (ibid.) then presents the necessary steps required for proper OOT. Corriveau (ibid.) concludes his section on OOT by stating that:

> *In the context of an incremental process with short micro–iterations, we cannot afford to retest all functionality at each iteration. Thus, testing is incremental in that it typically focuses on what has been added and modified in the current iteration. Regression testing is therefore especially important in order to quickly verify that basic functionality is still correct.*

Authors such as Corriveau (ibid.), Probert (1994, 1995a, 1995b, 1995c, and 1995d), and others have emphasized the need for proper OOT. As a means towards proper OOT, the creation of tools to assist during OOT is often mentioned. To this end, *FireWorks*, as a tool to aid during regression testing of

---

35 Refer to section 2.2 Testing as a Software Quality Engineering Activity (page 10) for a similar review of BBT and WBT.

OO systems (in particular OO systems developed in Smalltalk) was designed and implemented.

5.4 The Final Word

Testing software is a complicated, time–consuming task. Moreover, with the advent of object–oriented technology, additional techniques and tools are required to effectively test object–oriented systems.

FireWorks was designed and implemented to address the specific needs of regression testing of object–oriented systems. Using the concept of constructing firewalls, *FireWorks* has the ability to isolate a specified class and all of its related classes. Regression testing is therefore simplified, as the firewall for a class identifies all other affected classes that could be considered for retesting.

In short, *FireWorks* works!

Appendix A

HIGH LEVEL DESIGN OF FIREWORKS

CRC Cards for *FireWorks*

A.1 Class, Responsibility, Collaborator Notation

The CRC (Class, Responsibility, Collaborator) card notation was introduced by Beck and Cunningham (1989), and used by several object–oriented methodologies (Wirfs–Brock, 1990).

CRC cards correspond to an index card that represents a class of objects, their behavior, and their interactions. CRC cards are usually created in informal brainstorming sessions. The card usually contains the class name, a list of responsibilities (i.e. a list of services provided), and a list of collaborators (i.e. services required by the class to fulfill its responsibilities). See Figure 58.

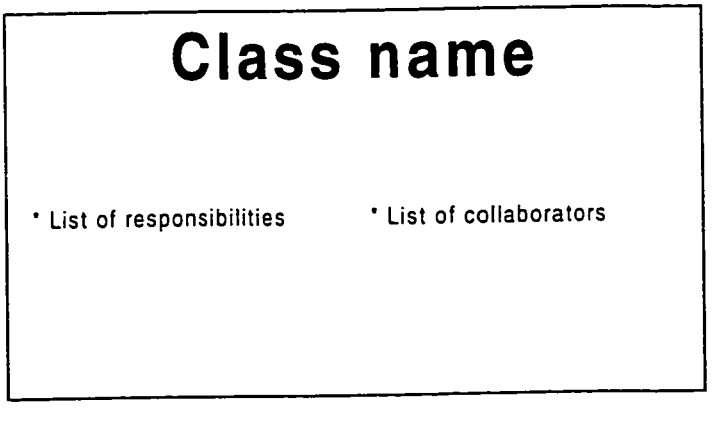


Figure 58: Class, Responsibility, Collaborator card.

The following sections present all of the CRC cards used to implement *FireWorks*. These CRC cards reflect the current state of *FireWorks*, and were essential during implementation. The CRC cards are grouped together according to subject: GUI, ORG, DiGraph, and LSC.

A.2 CRC Diagrams for Subject GUI

FireWorks	
• Allow user to select a class. • Construct firewall for selected class. • Display firewall. • Destroy firewall.	• Firewall

Figure 59: CRC card for class FireWorks.

Firewall	
• Represent various relationships between classes. • Manage ORG: build; display; destroy.	• ObjectRelationGraph

Figure 60: CRC card for class Firewall.

A.3 CRC Diagrams for Subject ORG

The essential component to *FireWorks* is the ORG subject. The ORG subject is responsible for representing the aggregation, association, and inheritance relationships between classes. Before representing these class relationships, they must first be detected.

Chapter 4 presents each of the algorithms used to detect aggregation[36], association[37], and inheritance[38] relationships. In particular, the algorithms are supplied to gently introduce the reader to the complexities of detecting class relationships.

For the sake of completeness, the Smalltalk implementation for detecting each class relationship is provided after its respective CRC card. Hopefully, the reader should notice a gradual continuum from the high–level CRC card for detecting a class relationship, to its medium–level algorithm, to its low–level Smalltalk implementation.

---

[36] Refer to section 4.3.1 Detecting Aggregation Relationships (page 63).

[37] Refer to section 4.3.2 Detecting Association Relationships (page 71).

[38] Refer to section 4.3.3 Detecting Inheritance Relationships (page 75).

ObjectRelationGraph	
• Represent the aggregation, association, and inheritance relationships between classes. • Manage all class relationships: add, remove, and display.	• ClassRelationship • DirectedGraph

Figure 61: CRC for class ObjectRelationGraph.

ClassRelationship	
For a given class: • Detect all aggregation relationships. • Detect all association relationships. • Detect all inheritance relationships.	• AggregationRelationship • AssociationReltionship • InheritanceRelationship

Figure 62: CRC card for class ClassRelationship.

```
'From VisualWorks(R), Release 2.5 of September 26, 1995 on November 13,
1997 at 5:55:28 pm'!

Object subclass: #ClassRelationship
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Relationships-Class'!
```

```
!ClassRelationship methodsFor: 'accessing'!

classNameOfClass: aClass
"Answer the class name of aClass."
"Reference: Behavior's instance method name"

^aClass name!

classWithName: aString
"Answer the class with aString for a name."

^Smalltalk at: aString!

metaClassOfClass: aClass
"Answer the meta-class of aClass."

^aClass class!

methodSourceForSelector: selector ofClass: aClass
"Answer the method source for aClass's selector."

^aClass sourceCodeAt: selector!

methodSourceForSelector: selector ofClassWithName: aString

^self methodSourceForSelector: selector ofClass: (self classWithName: aString)! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

ClassRelationship class
instanceVariableNames: "!


!ClassRelationship class methodsFor: 'examples'!

example
" (ClassRelationship example) "

^self new! !
```

Figure 63: Smalltalk implementation of class ClassRelationship.

AggregationRelationship

- Detect all aggregation relationships for a given class.
- Collect all instance variables of a given class.
- Collect which selectors of a given class write to an instance variable.
- Parse the method of a selector.
- Collect symbol representations of instance variables for a given class.

- Behavior
- SmalltalkParser

Figure 64: CRC for class AggregationRelationship.

```
'From VisualWorks(R), Release 2.5 of September 26, 1995 on November 13,
1997 at 5:55:36 pm'!

ClassRelationship subclass: #AggregationRelationship
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Relationships-Class'!


!AggregationRelationship methodsFor: 'accessing instances and variables'!

allInstVarNamesOfClass: aClass
"Answer an Array of the names of the receiver's instance variables."
"Reference: Behavior's instance method allInstVarName"

^aClass allInstVarNames!

allInstVarNamesOfClassWithName: aString

^self allInstVarNamesOfClass: (self classWithName: aString)! !

!AggregationRelationship methodsFor: 'testing method dictionary'!
```

```
whichSelectorsWriteTo: instVarName ofClass: aClass
"Answer a set of selectors whose methods access the argument, instVarName,
as a named instance variable."
"Reference: Behavior's instance method whichSelectorsAccess:"

| accessSelectors instVarIndex |
accessSelectors := aClass whichSelectorsAccess: instVarName.
instVarIndex := (self allInstVarNamesOfClass: aClass) indexOf: instVarName
ifAbsent: [^Set new].
^accessSelectors select: [ :selector | (aClass compiledMethodAt: selector)
writesField: instVarIndex]!

whichSelectorsWriteTo: instVarName ofClassNamed: aString

^self whichSelectorsWriteTo: instVarName ofClass: (self classWithName:
aString)! !

!AggregationRelationship methodsFor: 'private'!

parseMethodForSelector: selector ofClass: aClass
"Answer the symbolTable for the method parsed at a selector."

| method parser |
method := self methodSourceForSelector: selector ofClass: aClass.
parser := SmalltalkParser new.
parser parse: (ReadStream on: method) class: aClass.
^parser symbolTable!

parseMethodForSelector: selector ofClassNamed: aString

^self parseMethodForSelector: selector ofClass: (self classWithName: aString)! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

AggregationRelationship class
instanceVariableNames: "!


!AggregationRelationship class methodsFor: 'examples'!

example
" (AggregationRelationship example) "
```

```
^self new!

example1
" (AggregationRelationship example1) "

^(self new) allInstVarNamesOfClassWithName: self fullName!

example2
" (AggregationRelationship example2) "

^(self new) whichSelectorsWriteTo: 'vertices' ofClassNamed: 'DirectedGraph'!

example3
" (AggregationRelationship example3) "

| agg selectors symbolTable |
agg := self new.
selectors := agg whichSelectorsWriteTo: 'vertices' ofClassNamed:
'DirectedGraph'.
selectors do: [ :selector |
symbolTable := agg parseMethodForSelector: selector ofClassNamed:
'DirectedGraph'].
^symbolTable!

example4
" (AggregationRelationship example4) "

^(self new) allInstVarNamesOfClassWithName: 'FirewallViewer'! !
```

Figure 65: Smalltalk implementation of class AggregationRelationship.

AssociationRelationship	
• Detect all association relationships for a given class. • Collect all class method selectors of a given class. • Collect all senders that send any of a given class's class methods. • Parse method of a selector verifying whether or not a class method is actually sent to a given class.	• Behavior • SmalltalkParser • SymbolTableEntry

Figure 66: CRC for class AssociationRelationship.

```
'From VisualWorks(R), Release 2.5 of September 26, 1995 on November 13,
1997 at 5:55:41 pm'!

ClassRelationship subclass: #AssociationRelationship
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Relationships-Class'!


!AssociationRelationship methodsFor: 'retrieving'!

allClassMethodsOfClass: aClass
"Answer a SortedCollection of all the class methods that appear in the receiver's
forClass."
"Reference: Browser's class method allClassMethodsInProtocol:"

| aCollection organization |

aCollection := SortedCollection new.
organization := (self metaClassOfClass: aClass) organization.
organization categories do: [:category |
 (organization listAtCategoryNamed: category) do: [:selector |
aCollection add: selector]].
```

```
^aCollection!

allClassMethodsOfClassWithName: aString

^self allClassMethodsOfClass: (self classWithName: aString)!

allSendersOfClassMethods2
"Answer a SortedCollection of all the classes that call on any of receiver's class
methods."
"Reference: Browser's class method allCallsOn:"

| aSortedCollection classSelectors |

aSortedCollection := SortedCollection new.
classSelectors := self allClassMethods.
Smalltalk allBehaviorsDo: [:class |
 ((class ~= self forClass) & (class ~= self metaClassOfClass))
ifTrue: [classSelectors do: [:classSelector | | list list2 |
list := (class whichSelectorsReferTo: classSelector).
list size > 0
ifTrue: [list2 := (class whichSelectorsReferTo: self className).
list2 size > 0
ifTrue: [list do: [:sel | (list2 includes: sel) ifTrue: [aSortedCollection add: class
fullName]]]]]]].
^aSortedCollection!

allSendersOfClassMethodsForClass: aClass
"Answer a SortedCollection of all the classes that call on any of receiver's class
methods."
"Reference: Browser's class method allCallsOn:"

| aSortedCollection classSelectors |

aSortedCollection := SortedCollection new.
classSelectors := self allClassMethodsOfClass: aClass.
Smalltalk allBehaviorsDo: [:class |
"class := Smalltalk at: #ObjectRelationGraph."
 ((class ~= aClass) & (class ~= (self metaClassOfClass: aClass)))
ifTrue: [classSelectors do: [:classSelector | | list |
list := (class whichSelectorsReferTo: classSelector).
list size > 0
ifTrue: [(list findFirst:
 [:sel | self doesMethodSelector: sel ofClass: class sendSelector: classSelector
```

```
toClass: aClass]) > 0
ifTrue: [aSortedCollection add: class fullName]]]]].
^aSortedCollection!

allSendersOfClassMethodsForClassWithName: aString

^self allSendersOfClassMethodsForClass: (self classWithName: aString)! !

!AssociationRelationship methodsFor: 'testing'!

doesMethodSelector: selector ofClass: class sendSelector: anotherSelector toClass:
anotherClass
"Answer whether or not class's selector sends the message: antherClass
anotherSelector"

| method parser symbol |
method := self methodSourceForSelector: selector ofClass: class.
parser := SmalltalkParser new.
parser parse: (ReadStream on: method) class: class.
symbol := SymbolTableEntry new.
symbol
variableName: anotherClass fullName;
variableType: 'Receiver';
value: anotherSelector.
"(aClass == (Smalltalk at: #ObjectRelationGraph))
ifTrue: [(selector == #initialize) ifTrue: [self halt]]."
^parser symbolTable includesSymbolTableEntry: symbol! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

AssociationRelationship class
instanceVariableNames: "!


!AssociationRelationship class methodsFor: 'examples'!

example
" (AssociationRelationship example) "

^self new!

example1
" (AssociationRelationship example1) "
```

```
^(self new) allClassMethodsOfClassWithName: self fullName!

example2
" (AssociationRelationship example2) "

^(self new) allSendersOfClassMethodsForClassWithName: 'DirectedGraph'!

example2a
" (AssociationRelationship example2a) "

^(self new) allSendersOfClassMethodsForClassWithName: 'A'!

example2b
" (AssociationRelationship example2b) "

^(self new) allSendersOfClassMethodsForClassWithName: 'Car'!

example3
" (AssociationRelationship example3) "

^(self new) allSendersOfClassMethods2! !
```

Figure 67: Smalltalk implementation of class AssociationRelationship.

InheritanceRelationship

For a given class:

* Behavior

* Detect all inheritance relationships.
* Collect all super classes.
* Collect all sub-classes.

Figure 68: CRC for class InheritanceRelationship.

```
'From VisualWorks(R), Release 2.5 of September 26, 1995 on November 13,
1997 at 5:55:46 pm'!

ClassRelationship subclass: #InheritanceRelationship
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Relationships-Class'!


!InheritanceRelationship methodsFor: 'accessing class hierarchy'!

allSubclassesOfClass: aClass
"Answer an OrderedCollection of the receiver's subclasses and the receiver's
descendant's
subclasses in breadth-first order, with the immediate subclasses first."
"Reference: Behavior's instance method allSubclasses"

^aClass allSubclasses!

allSubclassesOfClassWithName: aString

^self allSubclassesOfClass: (self classWithName: aString)!
```

```
allSuperclassesOfClass: aClass
"Answer an OrderedCollection of the receiver's superclass and the receiver's
ancestor's
superclasses, with the immediate superclasses first."
"Reference: Behavior's instance method allSuperclasses"

^aClass allSuperclasses!

allSuperclassesOfClassWithName: aString

^self allSuperclassesOfClass: (self classWithName: aString)! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

InheritanceRelationship class
instanceVariableNames: "!


!InheritanceRelationship class methodsFor: 'examples'!

example
" (InheritanceRelationship example) "

^self new!

example1
" (InheritanceRelationship example1) "

^(self new) allSuperclassesOfClassWithName: self fullName!

example2
" (InheritanceRelationship example2) "

^(self new) allSubclassesOfClassWithName: self fullName! !
```

Figure 69: Smalltalk implementation of class InheritanceRelationship.

A.4 CRC Diagrams for Subject DiGraph

DirectedGraph

- A collection of nodes and one-way edges.
- Connect two vertices with a labeled arc.
- Traverse graph.

- Dictionary
- LabeledArc
- Vertex

Figure 70: CRC card for class DirectedGraph.

LabeledArc

- Record from vertex.
- Record to vertex.
- Record label of arc.

- String

Figure 71: CRC card for class LabeledArc.

Vertex	
• A point or vertex in a graph. • Record label of vertex.	• String

Figure 72: CRC card for class Vertex.

A.5 CRC Diagrams for Subject Language Specific Classes

SmalltalkParser	
• Perform syntax analysis on Smalltalk methods. • Determine parse tree for a Smalltalk method. • Manage SymbolTable: add, remove, retrieve symbols.	• LexicalAnalyzer • SymbolTable

Figure 73: CRC card for class SmalltalkParser.

LexicalAnalyzer

- Perform lexical analysis on Smalltalk methods.
- Determine all tokens for a Smalltalk method.

- Stream

Figure 74: CRC card for class LexicalAnalyzer.

SymbolTable

- Manage symbols: create, destroy, compare.
- Manage collection of symbols: add, remove, retrieve.

- Dictionary
- SymbolTableEntry

Figure 75: CRC card for class SymbolTable.

SymbolTableEntry

- Record name.
- Record type.
- Record value.
- Compare two symbols.

- String

Figure 76: CRC card for class SymbolTableEntry.

Appendix B

DETAILED DESIGN OF FIREWORKS

Class Relationship Diagrams for *FireWorks*

The following sections present all of the CRDs for *FireWorks*. These CRDs reflect the current implementation of *FireWorks*. The CRDs are grouped together according to subject: GUI, ORG, DiGraph, and LSC.

B.1 Class Relationship Diagram for Subject GUI

Figure 77 shows the CRD for the GUI subject. The GUI subject represents the graphical user interface (GUI) to *FireWorks*. The GUI subject is composed of the following classes:

- FireWorks, the user application

  firewallViewers is an ordered collection of instances of class FirewallViewer

- FirewallViewer, to view firewalls

  aggregationToggle is a Boolean indicating whether or not to detect aggregation relationships

  associationToggle is a Boolean indicating whether or not to detect association relationships

  firewall is an instance of class Firewall

  hierarchyHolder displays the object hierarchy for a class as determined by Smalltalk

  menuBar supports the various drop–down menu options for the FirewallViewer window

orgHolder displays the ORG for a class as determined by *FireWorks*

- Firewall, to construct firewalls

  org is an instance of class ORG

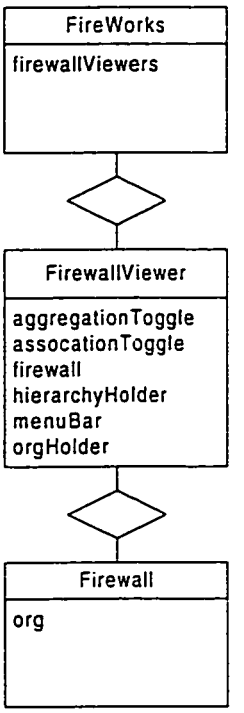


Figure 77: Class relationship diagram for Subject GUI.

B.2 Class Relationship Diagram for Subject ORG

Figure 78 shows the CRD for the ORG subject. The ORG subject represents the object relation graph used by *FireWorks*. The classes contained in the ORG subject include the following:

- ObjectRelationGraph

className is the name of the class for the ORG

digraph is an instance of class DirectedGraph

- ClassRelationship
- AggregationRelationship
- AssociationRelationship
- InheritanceRelationship

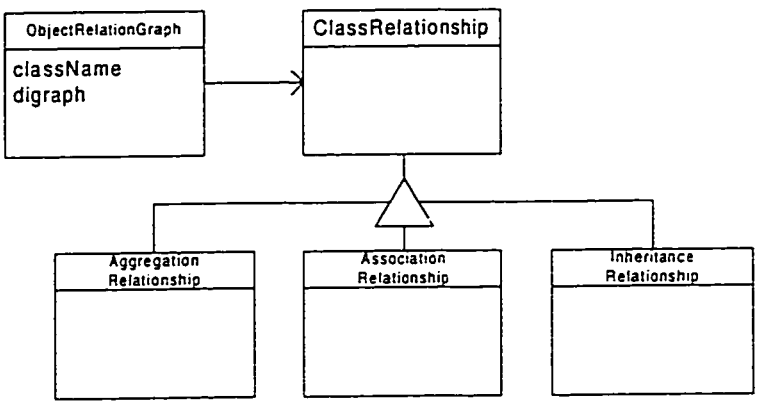


Figure 78: Class relationship diagram for Subject ORG.

B.3 Class Relationship Diagram for Subject DiGraph

Figure 79 shows the CRD for the DiGraph subject. The DiGraph subject represents the directed graph and its associated classes. The DiGraph subject is composed of the following classes:

- DirectedGraph

  arcs is an identity dictionary of arcs, with the key being the fromVertex of an arc

  vertices is a dictionary of vertices, with the key being the label of a vertex; each vertex entry is associated to its set of arcs

- LabeledArc

  fromVertex is the arc's initial vertex

  toVertex is the arc's final vertex

  label is the arc's label, as a string

- Vertex

  label is the vertex's label, as a string

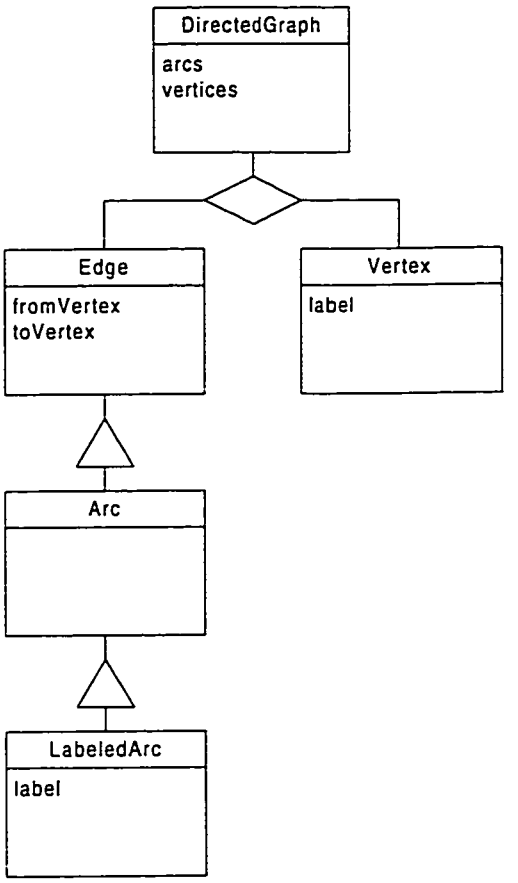


Figure 79: Class relationship diagram for Subject DiGraph.

B.4 Class Relationship Diagram for Subject Language Specific Classes

Figure 80 shows the CRD for the Language Specific Classes (LSC) subject. The LSC subject represents those classes that are dependent on the language used to implement *FireWorks*. The classes contained in this subject include the following:

- SmalltalkParser

  classToCompile is the Smalltalk class that contains the various methods to be compiled

  symbolTable is an instance of class SymbolTable

- SymbolTable

  symbolTable is a dictionary of symbol table entries

- SymbolTableEntry

  variableName is the symbol's name, as a string

  variableType is the symbol's type, as a string

  value is the symbol's value, as a string

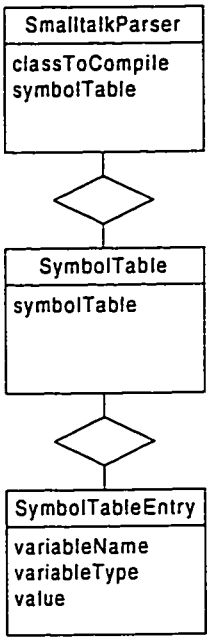


Figure 80: Class relationship diagram for Subject LSC.

BIBLIOGRAPHY

Adams, G. (1992) *Describing Groups of Interacting Objects Using Path Expressions*, Carleton University Master's Thesis, Ottawa, Ontario, Canada.

Agesen, O. and Ungar, D. (1994) 'Sifting Out the Gold', *ACM Conference on Object–oriented Programming Systems, Languages and Applications (OOPSLA)*, October 1994, ACM Press, New York, U.S.A.

Agha, G.A., Wegner, P, and Yonezawa, A. (1993) *Research Directions in Concurrent Object–Oriented Programming*, MIT Press, Cambridge, Massachusetts, U.S.A.

Aho, A.V., Hopcroft, J.E., and Ullman, J.D. (1983) *Data Structures and Algorithms*, Addison–Wesley, Reading, Massachusetts, U.S.A.

Beck, K. and Cunningham, W. (1989) 'A laboratory for teaching object–oriented thinking', *ACM Conference on Object–oriented Programming Systems, Languages and Applications (OOPSLA)*, October 1989, ACM Press, New York, U.S.A.

Beizer, B. (1990) *Software Testing Techniques*, 2nd edn, Van Nostrand Rheinhold, New York, U.S.A.

Binder, R.V. (1994) 'Object–Oriented Software Testing', *Communications of the ACM*, 37(9), 87–101.

Binder, R. V. (1996) 'Testing Object–Oriented Software: a Survey', *Software Testing, Verification and Reliability*, 6, 125–252.

Boehm, B.W. (1979) 'Software engineering: R & D trends and defense needs', *Research Directions in Software Technology*, MIT Press, Cambridge Massachusetts, U..S.A.

Borning, A.H. and Ingalls, D. H. H. (1982) 'A type declaration and inference system for Smalltalk', *Ninth Symposium on Principles of Programming Languages*, January 1982, ACM Press, New York, U.S.A., pp. 133–141.

Budd, T. (1991) *An Introduction to Object–Oriented Programming*, Addison–Wesley, Reading, Massachusetts, U.S.A.

Coad, P. and Yourdon, E. (1991) *Object–Oriented Design*, Prentice–Hall, Englewood Cliffs, New Jersey, U.S.A.

Coad, P. and Yourdon, E. (1992) *Object–Oriented Analysis*, 2nd edn, Prentice–Hall, Englewood Cliffs, New Jersey, U.S.A.

Cornell, G., and Horstmann, C.S. (1996) *Core Java*, SunSoft Press, Mountain View, California, U.S.A.

Corriveau, J.–P. (1996) 'Traceability and Process for Large OO Projects', *TOOLS–USA'96*, August 1996, Santa Barbara, California, U.S.A.

Ellis, M., and Stroustrup, B. (1990) *The Annotated C++ Reference Manual*, Addison–Wesley, Reading, Massachusetts, U.S.A.

Fisher, C.N. and LeBlanc, R.J. (1988) *Crafting a Compiler*, Benjamin/Cummings, Menlo Park, California, U.S.A.

Gersting, J.L. (1987) *Mathematical Structures for Computer Science*, 2nd edn, W.H. Freeman and Company, New York, U.S.A.

Goldberg, A., and Robson, D. (1983) *Smalltalk–80: The Language and its Implementation*, Addison–Wesley, Reading, Massachusetts, U.S.A.

Graver, J.O. (1989) *Type–Checking and Type–Inference for Object–Oriented Programming Languages*, Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana–Champaign.

Graver, J.O. and Johnson, R.E. (1990) 'A type system for Smalltalk', *Seventeenth Symposium on Principles of Programming Languages*, January 1990, ACM Press, New York, U.S.A., pp. 136–150.

Hense, A.V. (1990) 'Polymorphic type inference for a simple object oriented programming language with state', *Technical Report No. A 20/90*, Fachbericht 14, Universität des Saarlandes.

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1992) *Object–Oriented Software Engineering A Use Case Driven Approach*,. Addison–Wesley, Reading, Massachusetts, U.S.A.

Kung, D.C., Gao, J., Hsia, P., Lin, J., and Toyoshima, Y. (1995a) 'Class firewall, test order, and regression testing of object–oriented programs', *Journal of Object–Oriented Programming*, 8(2), 51–65.

Kung, D.C., Gao, J, Hsia, P., Toyoshima, Y., Chen, C., Kim, Y., and Song, Y. (1995b) 'Developing an Object–Oriented Software Testing and Maintenance Environment', *Communications of the ACM*, 38(10), 75–86.

Lemone, K.A. (1992) *Fundamentals of Compilers: An Introduction to Computer Language Translation*, CRC Press, Boca Raton, Florida, U.S.A.

Linton, M. (1992) 'Implementing Resolution Independence on top of the X Window System', *Proceedings of the 6th X Conference*, January 1992, Boston, Massachusetts, U.S.A.

Murphy, G.C., Townsend, P., Laberge, P., and Juzenas, M. (1993) 'Engineering software with objects', *Object–Oriented Applications* (B. Meyer and J.M. Nerson, eds.), Prentice–Hall, Englewood Cliffs NJ, U.S.A.

Murphy, G.C., Townsend, P., and Wong, P.S. (1994) 'Experiences with Cluster and Class Testing', *Communications of the ACM*, 37(9), 39–47.

Myers, G.J. (1979) *The Art of Software Testing*, Wiley, New York, U.S.A.

Oxhoj, N., Palsberg, J. and Schwartzbach, M.I. (1992) 'Making type inference practical', Object–oriented type inference', *European Conference on Object–Oriented Programming (ECOOP)*, Springer–Verlag, New York, U.S.A., pp. 329–349.

Palsberg, J. and Schwartzbach, M.I. (1991) 'Object–oriented type inference', *ACM Conference on Object–Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 1991, ACM Press, New York, U.S.A., pp. 146–161.

Palsberg, J. and Schwartzbach, M.I. (1994) 'Static typing for Object–oriented programming', *Science of Computer Programming*, 23(1), 19–53.

ParcPlace–Digitalk (1995) *VisualWorks Release 2.5 User's Manual, Revision 2.0*, Sunnyvale, California, U.S.A.

Parrington, N., and Roper, M. (1989) *Understanding Software Testing*, Horwood, Chichester, West Sussex, U.K.

Perry, D.E., and Kaiser, G.E. (1990) 'Adequate testing and object–oriented programming', *Journal of Object–Oriented Progamming*, 2(1), 13–19.

Probert, R.L. (1994) *Volume 1: Introduction to Software Quality Engineering*, University of Ottawa, Ottawa, Ontario, Canada.

Probert, R.L. (1995a) *Volume 2: Requirements Capture and Validation – Black Box (Functional) Analysis Methods*, University of Ottawa, Ottawa, Ontario, Canada.

Probert, R.L. (1995b) *Volume 3: Planning for Test Automation and Test Tool Support*, University of Ottawa, Ottawa, Ontario, Canada.

Probert, R.L. (1995c) *Volume 4: Static and Dynamic White–Box (Structural) Verification Methods*, University of Ottawa, Ottawa, Ontario, Canada.

Probert, R.L. (1995d) *Volume Product Assessment/SQE Management Techniques*, University of Ottawa, Ottawa, Ontario, Canada.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991) *Object–Oriented Modeling and Design*, Prentice–Hall, Englewood Cliffs, New Jersey, U.S.A.

Schwartzbach, M.I. (1991) 'Type inference with inequalities,' *Proceedings of TAP–SOFT'91*, Springer–Verlag (LNCS 493).

Skublics, S., Klimas, E.J., and Thomas, D.A. (1996) *Smalltalk with Style*, Prentice–Hall, Englewood Cliffs, New Jersey, U.S.A.

Sommerville, I. (1996) *Software Engineering*, 5th edn, Addison–Wesley, Reading, Massachusetts, U.S.A.

Stroustrup, B. (1992) *The Design and Evolution of C++*, Addison–Wesley, Reading, Massachusetts, U.S.A.

Suzuki, N. (1981) 'Inferring type in Smalltalk', *Eighth Symposium on Principles of Programming Languages*, pp. 187–199.

Tokoro, M. and Yonezawa, A. (1987) *Object–Oriented Concurrent Programming*, MIT Press, Cambridge, Massachusetts, U.S.A.

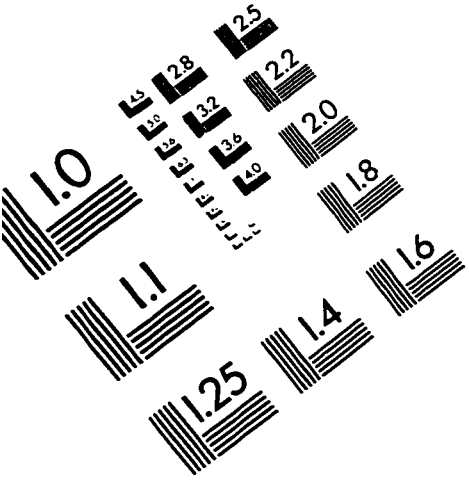
Ungar, D., and Smith, R.B. (1987) 'Self: The Power of Simplicity', *ACM Conference on Object–Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 1987, ACM Press, New York, U.S.A., pp. 227–241.

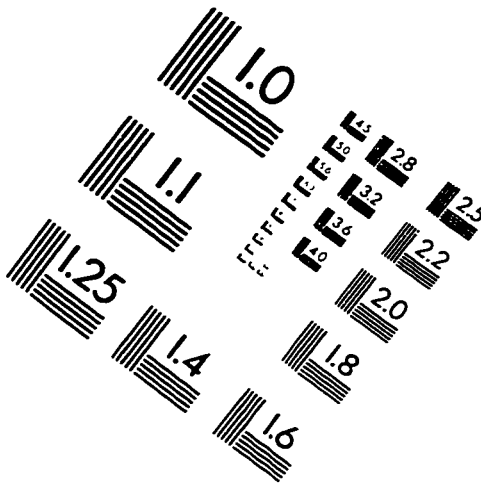
Wand, M. (1987) 'A simple algorithm and proof for type inference', Fundamentae Informaticae, X, pp. 115–122.

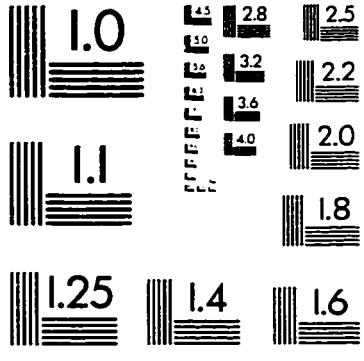
White, L., and Leung, H.K.N. (1992) 'A firewall concept for both control–flow and data–flow in regression integration testing', *Proceedings of Conference on Software Maintenance*, Orlando, Florida, U.S.A, pp. 262–271.

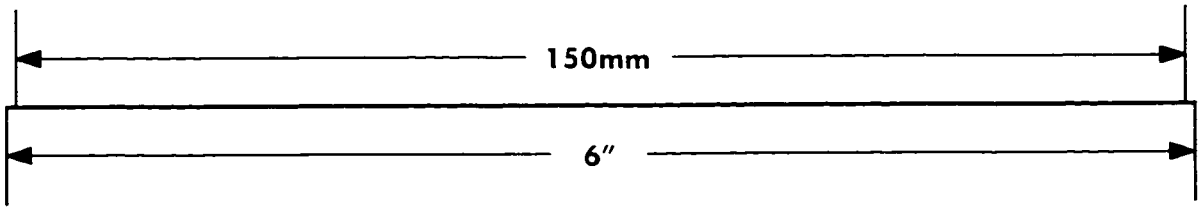
Wirfs–Brock, R., Wilkerson, B. and Wiener, L. (1990) *Designing Object–Oriented Software*, Prentice–Hall, Englewood Cliffs, New Jersey, U.S.A.

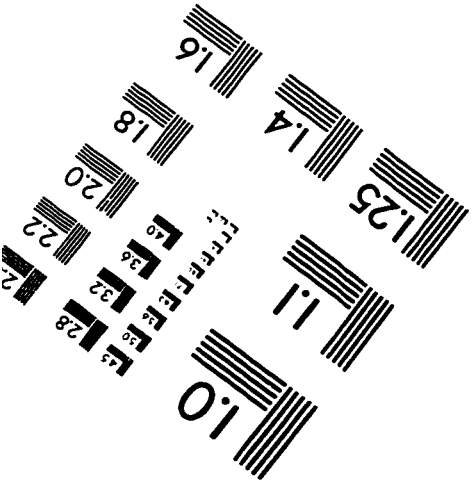
IMAGE EVALUATION TEST TARGET (QA-3)

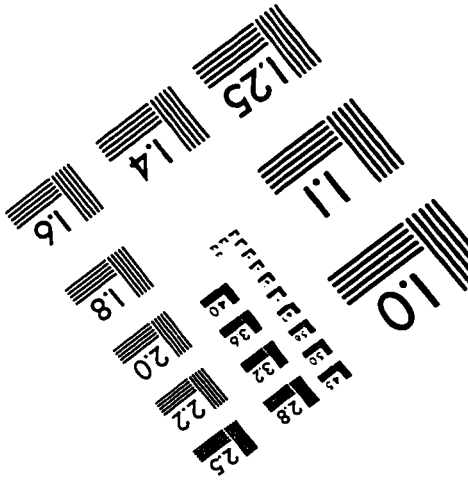












APPLIED IMAGE . Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989