On the Syllogistic Structure of Object-Oriented Programming

Derek Rayside and Kostas Kontogiannis
Electrical & Computer Engineering
University of Waterloo
Waterloo, Canada
{drayside, kostas}@swen.uwaterloo.ca

Abstract

Recent works by Sowa and by Rayside & Campbell demonstrate that there is a strong connection between object-oriented programming and the logical formalism of the syllogism, first set down by Aristotle in the Prior Analytics. *In this paper, we develop an understanding of polymorphic method invocations in terms of the syllogism, and apply this understanding to the design of a novel editor for object-oriented programs. This editor is able to display a polymorphic call graph, which is a substantially more difficult problem than displaying a non-polymorphic call graph. We also explore the design space of program analyses related to the syllogism, and find that this space includes Unique Name, Class Hierarchy Analysis, Class Hierarchy Slicing, Class Hierarchy Specialization, and Rapid Type Analysis.*

1 Introduction

Discovering the relations between program entities is one of the most important tasks in understanding a program. We propose that the kinds of relations that are unique to object-oriented programming are expressed in the form of the syllogism — in other words, that object-oriented programs are distinguished by their *syllogistic structure*.

Our enquiry has four main parts: a *methodology* for deriving structural queries from a logical formalism, an *analytical framework* composed of these queries, an *evaluation* of some common programmer's aides based on this framework, and the novel design of a *program editor* for understanding polymorphic method invocations.

The purpose of the methodology is to systematically explore the design space of program analyses that are based on the syllogistic structure of object-oriented programs. This approach has three objectives: to identify the kinds of questions that programmers may ask about the syllogistic structure of their programs, to provide a basis for evaluating object-oriented programmer's aides, and to present a principled means for designing programmer's aides.

This kind of methodology could be based on a formalism other than the syllogism: we could use a more sophisticated formalism to derive more sophisticated queries. However, using the syllogism establishes some reasonable grounds for the common claim that object-oriented programming is easy to understand: the syllogism is both simple and well known. The study of formal logic began with the syllogism and was focused on it almost exclusively until the nineteenth century.

The concrete technical contribution of this paper is to demonstrate how polymorphic method invocations can be understood in terms of the syllogism, and how this insight can be applied to the design of a novel program editor. This editor is designed to display a polymorphic call graph, which may be computed statically by techniques such as *Rapid Type Analysis* [2], or more advanced analyses such as [16, 18]. Displaying a polymorphic call graph is a substantially more difficult problem than displaying a non-polymorphic call graph, precisely because each invocation in a polymorphic call graph may have many potential targets.

2 Background

Sowa [14] and Rayside & Campbell [12] show that deductions such as the following two examples are the characteristic structures of class-based, single-dispatch, object-oriented programming languages:

Every animal is mortal.
Every human is an animal.
∴ Every human is mortal.

Every human is mortal.
Socrates is human.
∴ Socrates is mortal.

The main difference is that the first deals with classes only (*i.e.* the specialization relation), whereas the second moves from class to object (*i.e.* the instantiation relation).

The first argument (*i.e.* deduction) is an example of a *syllogism* in the first *figure*, universal affirmative *mood*: the *figure* of the syllogism refers to the logical arrangement of the terms, and the *mood* refers to the quality (affirmative or

0-7695-1050-7/01 $10.00 © 2001 IEEE

negative) and quantity (universal or particular) of the statements. The other figures of the syllogism are discussed in [1, 10, 12, 13] and many other works.

The field of formal logic was initiated by Aristotle's exploration of this kind of argument in the *Prior Analytics* [1]. This form of argument was inspired by Plato's method of logical division, and was called perfect by Aristotle. Rose [13] provides a fairly comprehensive catalogue of the reason for considering this form of argument as perfect. Logicians in the Middle Ages named these syllogisms BARBARA, according to an intricate naming convention that codified much of the mediæval knowledge of the syllogism as a formalism (this convention is discussed briefly by Sowa [14], and in more detail by Rose [13]). The first modern mathematical formalism of the syllogism was developed by Łukasiewicz in the middle of the twentieth century [10].

In the original Greek texts, Aristotle usually gives syllogisms in the implicational (*if, then*) form, rather than the inferential (*therefore*) form commonly used in Latin texts and object-oriented programming.

The second example argument was first given by Sextus Empiricus (as noted by Łukasiewicz [10]), and Empiricus attributed it to Aristotle. However, as Łukasiewicz quite adamantly argues, Aristotle would not consider such an argument in the *Prior Analytics* because it contains a singular term ('Socrates'). Consequently, these kinds of arguments do not have a name in the traditional Latin nomenclature, which was developed only for the forms of the syllogism discussed in the *Prior Analytics*. Nevertheless, some authors have referred to this second kind of arguments as BARBARA or DARII (*e.g.* [14]).

Syllogistic Terminology. Every statement and every term in a syllogism has a name. The three terms in the syllogism are the *major extreme*, the *minor extreme* and the *middle term*. From the first example, the major extreme is 'mortal', the minor extreme is 'human', and the middle term is 'animal'. The reason for these names can be seen more clearly when the example is grammatically re-arranged so that the ordering of the terms is the same as in the Greek:

$$\frac{\text{'mortal' is said of 'animal'} \wedge \text{'animal' is said of 'human'}}{\therefore \text{'mortal' is said of 'human'}}$$

The three statements are the *major premise*, *minor premise* and *conclusion*. The *major premise* is so named because it contains the major extreme term, whereas the minor premise contains the minor extreme term. The premises may be given in any order but, it is customary to give the major premise first. Both premises also contain the middle term. The conclusion brings the two extreme terms together through ('because of') the middle term.

3 Methodology

Here we develop a methodology for deriving structural queries about object-oriented programs from the form of the syllogism. Our intention is to systematically explore the design space of object-oriented program analyses by identifying the questions that programmer's may ask about the syllogistic structure of their object-oriented programs.

Each statement in a syllogism expresses a simple relation between two things: subject and predicate. A syllogism as a whole expresses a complex relation between three things: the major extreme, the minor extreme, and the middle term.

We can derive two queries from the simple relation expressed in each statement by binding one term to a program entity; the result of the query is the set of program entities that may be bound to the other term. Likewise, we can derive three queries from the complex relation expressed in the syllogism as a whole by binding two terms to program entities; the result of the query is the set of program entities that may be bound to the third term.

3.1 Scope

We are focused on the syllogistic structure of object-oriented programs. There are many other important things that the programmer may wish to know about a given program other than its syllogistic structure: *e.g.* data flow, control flow, comments *etc*. This simply indicates that programming languages are more sophisticated than the syllogism, but also that they share its basic structure.

This kind of methodology could also be developed for formalisms that are more general or sophisticated than the syllogism. However, the criteria upon which to judge a formalism's suitability to this kind of methodology is not generality or sophistication but, rather, if the queries derived from it are useful and resonate with the programmer's understanding of the program. As we shall show, the syllogism meets this criteria very well.

This paper focuses on class-based object-oriented programming languages that support single-dispatch, static typing, and method overriding semantics, such as JAVA [8] and C++ [15]. It should be fairly easy to relax most of these criteria for languages like SELF [20] (prototype-based), SMALLTALK [7] (dynamic typing), or BETA [11] (method extension semantics). Multiple-dispatch languages, such as CECIL [4], may present a greater challenge.

3.2 On Interpretation

Some care needs to be taken when comparing the form of the syllogism as expressed in prose with the form of the syllogism as expressed in code. There are three major areas of concern: the verbs chosen to express each relation; whether the relation expressed in the minor premises is meant in a direct or in a transitive sense; and whether singular terms refer to variables or to objects.

Verbs. Each of the example 'syllogisms' given in the background section of this paper may be characterized by the relation expressed in the minor premise: either *specialization* or *instantiation*. Therefore, we will refer to these two kinds of syllogisms as the 'specialization syllogism' and the 'instantiation syllogism'.

The major premise (*e.g.* 'Every animal is mortal') usually tells us something about the subject other than what it is [12]: in other words, the 'is' in the example is used in a strictly copulative sense. Within the context of object-oriented programming, the term 'declares' may be used to describe the relation expressed in the major premise.

In prose, the conclusion usually employs the same verb copula as the major premise, since both statements have the same predicate (but different subjects). However, in object-oriented programming we commonly use the term 'inherits' as the verb copula in the conclusion, to indicate the difference between the conclusion and the major premise. It has been pointed out that this 'inheritance' metaphor is not particularly appropriate within the context of class-based programming languages and syllogistic logic [12]. We may now rephrase the first example syllogism as:

Animal *declares* Mortal
Human *specializes* Animal
∴ Human '*inherits*' Mortal

In programming languages with method overriding semantics it is useful to interpret the conclusion in both ways: as 'declares' and as 'inherits'. For example, the following query may be interpreted as 'Which members from class X does class Y override?':

X declares $\{members\}$
Y specializes X
Y declares $\{members\}$

If we use the term 'inherits' in the conclusion, rather than 'declares', we get the query 'Which members from class X does class Y 'inherit'?'. Notice that the 'therefore' symbol (∴) has been removed from the 'declares' conclusion: when expressed in this way, it is not a conclusion in any meaningful sense; it is merely a juxtaposition of statements.

Transitivity of Relations. Sometimes it is convenient to interpret the relation in the minor premise (*i.e.* specializes or instantiates) in a transitive rather than a direct sense. In other words, to use the argument on the right (below) as a short form for the argument on the left (which is known as a 'sorites' in traditional logic):

Every animal is mortal	Every animal is mortal
Every mammal is an animal	
∴ Every mammal is mortal	Every human is an animal
Every human is a mammal	
∴ Every human is mortal	∴ Every human is mortal

Singulars. Finally, we need to be careful about how we interpret singular terms (*e.g.* 'Socrates') in the instantiation syllogisms, because programs have two notions of the singular: objects and variables. Moreover, we can consider the identity of objects in both a static and dynamic sense: the latter is usually what is meant by the identity of an object, but the former is often used in program analysis, where objects are identified by their instantiation site.

4 Analytical Framework

This section presents an analytical framework derived using the methodology presented in the previous section. This framework identifies the design space that includes analyses such as *Unique Name* [3], *Class Hierarchy Analysis* [5], *Class Hierarchy Slicing* [17], *Class Hierarchy Specialization* [19], and *Rapid Type Analysis* [2]. We also show how this framework explains the fundamental tasks of type checking polymorphic method invocations and dynamic dispatch.

Tables 1, 2, 3 and 4 list the queries that make up the framework. Tables 1 and 3 list the queries derived from individual statements in the specialization and instantiation syllogisms, respectively. Tables 2 and 4 list the queries derived from the specialization and instantiation syllogisms, respectively, in their entirety. The queries are numbered according to the table, section, and order that they appear in. The notation used for the queries is explained below.

One might initially think that the framework contains 18 queries: two per statement times six statements, plus three per syllogism times two syllogisms. However, there are actually 28 queries, since some structures may be meaningfully interpreted in more than way (as discussed previously).

Notation. Capital letters signify classes; plain lower case letters ('y') signify variables; lower case letters with a dot ('ẏ') signify objects. Four symbols are used to signify the different kinds of query results: '∘' represents a set of members (*i.e.* fields or methods); '□' represents a set of classes; '◊' represents a set of variables; '◆' represents a set of objects. Since it is difficult to express particularly quantified statements in most object-oriented programming languages, statements tend to be universally quantified [12]. Therefore, for convenience, we have simply dropped the quantifiers.

4.1 Discussion of Selected Queries

This section clarifies some of the queries presented in Tables 1, 2, 3 and 4, and discusses their importance in object-oriented programming.

Command Completion. The query `[Q3C1]` ('Which members may be used by variable y?') is equivalent to member command completion. ('Member command completion' is the task performed by many IDE's when the programmer types '`y.`' and then the IDE lists all of the members that may be used by variable `y`.)

Statement	Query	Interpretation	No.
Major Premise	X declares ∘	What members are declared by class X?	Q1A1
	□ declares M	Which classes declare member M? (*i.e.* unique name [3])	Q1A2
Minor Premise	Y specializes □ (transitive)	What are the super-classes of class Y?	Q1B1
	□ specializes X (transitive)	What are the sub-classes of class X?	Q1B2
Conclusion	∴ Y 'inherits' ∘	What members are 'inherited' by class Y?	Q1C1
	∴ □ 'inherits' M	Which classes 'inherit' member M?	Q1C2

Table 1. Queries derived from the specialization syllogism — per statement

Missing Term	Query	Interpretation	No.
Major Extreme	X declares ∘ Y specializes X (transitive) ∴ Y 'inherits' ∘	Which members from class X does class Y 'inherit'?	Q2A1
	X declares ∘ Y specializes X (transitive) Y declares ∘	Which members from class X does class Y override?	Q2A2
Minor Extreme	X declares M □ specializes X (transitive) ∴ □ 'inherits' M	Which classes 'inherit' member M from class X?	Q2B1
	X declares (or 'inherits') M □ specializes X (transitive) □ declares M	Which sub-classes of class X override member M? (*i.e.* lower part of Class Hierarchy Analysis [5])	Q2B2
Middle Term	□ declares M Y specializes □ (transitive) Y 'inherits' (and/or declares) M	Which super-classes of class Y declare member M? (*i.e.* upper part of Class Hierarchy Analysis [5])	Q2C1

Table 2. Queries derived from the specialization syllogism — whole *argument*

Statement	Query	Interpretation	No.
Major Premise	Y declares or 'inherits' ∘	What members does class Y have? (*c.f.* Q1A1 & Q1C1)	Q3A1
	□ declares or 'inherits' M	Which classes have member M? (*c.f.* Q1A2 & Q1C2)	Q3A2
Minor Premise	y instantiates □ (direct)	What is the declared type of variable y?	Q3B1
	ẏ instantiates □ (direct)	What is the actual type of object ẏ?	Q3B2
	◊ instantiates Y (direct)	Which variables are declared as class Y?	Q3B3
	♦ instantiates Y (direct)	Which objects instantiate class Y?	Q3B4
Conclusion	y may use member ∘	Which members may be used by variable y?	Q3C1
	y uses member ∘	Which members are used by variable y?	Q3C2
	ẏ may use member ∘	Which members may be used by object ẏ?	Q3C3
	◊ uses member M	Which variables use member M?	Q3C4
	♦ may use member M	Which objects may use member M?	Q3C5

Table 3. Queries derived from the instantiation syllogism — per statement

Missing Term	Query	Interpretation	No.
Major Extreme	Y declares ○ y instantiates Y (transitive) y uses ○	Which members from class Y are used by variable y? (*i.e.* Class Hierarchy Slicing [17] & Specialization [19])	Q4A1
	Y declares ○ ẏ instantiates Y (transitive) ẏ may use ○	Which members from class Y may be used by object ẏ?	Q4A2
Minor Extreme	Y declares M ◇ instantiates Y (transitive) ◇ uses M	Which variables use member M from class Y? (*i.e.* Class Hierarchy Slicing [17] & Specialization [19])	Q4B1
	Y declares M ◆ instantiates Y (transitive) ◆ may use M	Which objects use member M from class Y?	Q4B2
Middle Term	□ declares M y instantiates □ (transitive) y uses M	Which classes that variable y instantiates declare member M? (*i.e.* static type safety: an empty result set indicates that this invocation is illegal)	Q4C1
	□ declares M ẏ instantiates □ (most immediate) ẏ uses M	What is the most immediate class that ẏ instantiates that declares M?(*i.e.* dynamic method dispatch)	Q4C2

Table 4. Queries derived from the instantiation syllogism — whole argument

Unique Name. The basis of the *Unique Name* analysis proposed in [3] is expressed in [Q1A2]:

□ declares M

In other words, 'Which classes declare member M?' For example, if the program contained both a `Shape` and a `Cowboy` class, then the result of the query would contain both of their `draw` methods. [This example is borrowed from Boris Magnusson.] The Unique Name analysis evaluates this query for each member M in the program and looks for result sets with cardinality one.

Class Hierarchy Analysis. *Class Hierarchy Analysis* (CHA) [5] is a reasonably well known technique for the static construction of a (conservative) call graph. CHA can be expressed as a combination of two of the queries in our framework: [Q2B2] and [Q2C1]. The query 'Which sub-classes of class X override member M?' is expressed as [Q2B2]:

X declares (or 'inherits') M
□ specializes X (transitive)
□ declares M

We will refer to this as the 'lower part' of CHA, since it looks 'down' the class hierarchy. CHA also looks 'up' the class hierarchy, and the 'upper part' of CHA is expressed in our framework as [Q2C1]:

□ declares M
Y specializes □ (transitive)
Y 'inherits' (and/or declares) M

In other words, 'Which super-classes of class Y declare member M?' This query is concerned with the middle term.

The Importance of the Middle Term. The middle term plays a crucial role in syllogistic: it is the 'cause' of the argument, the reason why subject and predicate are brought together in the conclusion. The queries derived with respect to the middle term are particularly interesting, *e.g.* [Q4C1]:

□ declares M
y instantiates □ (transitive)
y uses M

In other words, 'Which classes that variable y instantiates declare member M?' This query is essentially equivalent to the task of statically type-checking a polymorphic method invocation: if the result set of this query is empty, then the invocation is illegal; if the result set of this query is non-empty, then the invocation is legal. In programming terms, the task is to determine if method M can be invoked on variable y; in logic terms, the task is to determine if the subject (*e.g.* y) and the predicate (*e.g.* M) can be combined. If we interpret the singular term in this query as an object instead of as a variable, we get a query of similar interest, [Q4C2]:

□ declares M
ẏ instantiates □ (most immediate)
ẏ uses M

In other words, 'What is the most immediate class that ẏ instantiates that declares M?' This query is essentially equivalent to dynamic method dispatch (in a single dispatch language with method overriding semantics). That is, the purpose of a virtual function table is to resolve this kind of query.

Class Hierarchy Slicing & Specialization. The essential task in Phase I of *Class Hierarchy Specialization* [19] is to resolve queries such as [Q4A1] and [Q4B1] ('Which variables use member M from class Y?' and 'Which members from class Y are used by variable y?'). It is important to note, however, that the queries as stated here are only valid for static member access in a language with single 'inheritance' (*i.e.* without the full generality of the C++ 'inheritance' mechanism). The formulation given in [19] addresses both of these substantial difficulties. Moreover, hierarchy specialization requires further analysis that is beyond the scope of our approach.

Class Hierarchy Specialization can be viewed as a refinement of *Class Hierarchy Slicing* [17]: the main difference is that hierarchy slicing only addresses cases where the result set of [Q4A1] is empty, whereas hierarchy specialization can also work with cases where the result set is non-empty. A possible refinement of hierarchy specialization may be an analysis based on [Q4A2] and [Q4B2], which interpret singulars as objects instead of as variables.

Rapid Type Analysis. *Rapid Type Analysis* (RTA) [2] is a fast and effective technique for statically resolving polymorphic method invocations. The basic insight of RTA is to prune the results of Class Hierarchy Analysis by eliminating classes that are never instantiated by the program. In other words, to prune the results of [Q2B2] with the results of [Q3B4]. However, like Class Hierarchy Slicing and Specialization, RTA also involves analysis that is beyond the scope of our approach.

5 Evaluation

In this section we use the framework developed in the previous section to evaluate four common aides for working with object-oriented programs: **Source** code with a basic text editor; **UML/CD** class diagrams; **javadoc** (JDK v1.3); and **IDE**, IBM's VisualAge for Java (v3.5).

The purpose of this evaluation is to assess how well these programmer's aides reveal the syllogistic structure of object-oriented programs. As we have mentioned, there is more to an object-oriented program than its syllogistic structure, and so the scope of our evaluation is limited. Moreover, these aides are not all directly comparable with each other: one couldn't choose to use javadoc instead of a text editor. For these reasons, this evaluation should not be seen as an attempt to rank these aides against each other but, rather, to rank them against the framework we have just developed.

We use a very simple scoring system for the evaluation: '•' indicates the aide is explicitly structured to provide the desired information in a convenient fashion; 'o' indicates that the programmer can get the information from the aide with a modest amount of effort; '·' indicates that it is possible for the programmer to derive the information, although perhaps with much effort; ' ' (a blank space) indicates that it is not possible for the programmer to derive the desired information from the aide's representation.

The evaluation is presented in Table 5. Two queries, [Q3A1] and [Q3A2], have been omitted from the table because they are equivalent to queries [Q1A1, Q1C1] and [Q1A2, Q1C2], respectively. The reason for this equivalence is that the conclusion of the specialization syllogism is used as the major premise of the instantiation syllogism.

First, notice that the structure of the source code makes it difficult to discern almost anything at all about the program's syllogistic structure. In other words, text editors are made to work with linear documents, and this is in fundamental conflict with the inherent syllogistic structure of object-oriented programs. The consequence of this conflict is that the programmer must either expend substantial cognitive effort to understand the syllogistic structure of the program, or enlist the use of some other aide.

The other three aides all support the queries derived from the specialization syllogism reasonably well, with UML providing the weakest support and IDE providing the strongest support. From this we conclude that if UML or javadoc are used in addition to IDE, then it is because of other information that they provide and not because they reveal the syllogistic structure of the program. We do note, however, that other IDE's may not provide the same functionality as IBM's VisualAge for Java (v3.5).

Of course, IDE provides much better support for queries derived from the instantiation syllogism than either UML class diagrams or javadoc, neither of which support these queries at all (nor are they intended to). Certain queries, such as [Q3B2] ('What is the actual type of object $\dot{y}$?') have an inherently dynamic nature that the debugger in IDE is particularly well suited to. Others, such as [Q3C1] ('Which members may be used by variable y?'; *i.e.* command completion) represent one of the advantages of using an IDE over a basic text editor.

6 Proposed Program Editor

Polymorphic method invocations present a number of important technical challenges: efficient implementation (*e.g.* [6]); static resolution (*e.g.* [3, 5, 2, 16, 18]); and program understanding. Many aides provide support for understanding non-polymorphic invocations using HTML-style hyperlinks. This approach is obviously insufficient for polymorphic invocations, precisely because they may have many potential targets, and an HTML-style hyperlink has only one target. Moreover, it is not just a question of showing multiple targets but, of showing why each target should be included. This is the technical problem that our proposed program editor is designed to address.

To understand a polymorphic method invocation, one must know why this subject (*i.e.* variable) and this predicate (*i.e.* method) can be combined, and what this combi-

No.	Query / Interpretation	Source	UML/CD	javadoc	IDE
Q1A1	What members are declared by class X?	○	●	●	●
Q1A2	Which classes declare member M?	·	·	●	●
Q1B1	What are the super-classes of class Y?	·	●	●	●
Q1B2	What are the sub-classes of class X?	·	●	●	●
Q1C1	What members are 'inherited' by class Y?	·	○	●	●
Q1C2	Which classes 'inherit' member M?	·	○	·	○
Q2A1	Which members from class X does class Y 'inherit'?	·	○	●	●
Q2A2	Which members from class X does class Y override?	·	○	○	●
Q2B1	Which classes 'inherit' member M from class X?	·	○	·	○
Q2B2	Which sub-classes of class X override member M?	·	○	·	○
Q2C1	Which super-classes of class Y declare member M?	·	○	○	●
Q3B1	What is the declared type of variable y?	·			●
Q3B2	What is the actual type of object ẏ?	·			●
Q3B3	Which variables are declared as class Y?	·			○
Q3B4	Which objects instantiate class Y?	·			○
Q3C1	Which members may be used by variable y?	·			●
Q3C2	Which members are used by variable y?	·			○
Q3C3	Which members may be used by object ẏ?	·			·
Q3C4	Which variables use member M?	·			○
Q3C5	Which objects may use member M?	·			·
Q4A1	Which members from class Y are used by variable y?	·			·
Q4A2	Which members from class Y may be used by object ẏ?	·			·
Q4B1	Which variables use member M from class Y?	·			·
Q4B2	Which objects use member M from class Y?	·			·
Q4C1	Which classes that variable y instantiates declare member M?	·			○
Q4C2	What is the most immediate class that ẏ instantiates that declares M?	·			○

Table 5. Evaluation of object-oriented programmer's aides

```
package ca.uwaterloo.swen;
import ca.uwaterloo.graphics;

public class Main {
  public static void main(String[] args) {
    Shape shape;
    shape = new Circle();
    foo(shape);
    shape = new Square();
    foo(shape);
  }
  static void foo(Shape s) {
    s.draw();
  }
}
```

```
package ca.uwaterloo.graphics;

abstract class Drawable {
  abstract void draw();
}
abstract class Shape extends Drawable {}
class Circle extends Shape {
  void draw() { printf("Circle"); }
}
class Triangle extends Shape {
  void draw() { printf("Triangle"); }
}
class Rectangle extends Shape {
  void draw() { printf("Rectangle"); }
}
class Square extends Rectangle {}
```

Figure 1. Source code for example (*c.f.* Figure 2)

nation means. The 'why' is the missing middle term, as seen in [Q4C1], and the 'what' is missing minor extreme, as seen in [Q2B2]. The 'what' may be refined by a more advanced analysis, *e.g.* [2, 16, 18]. Both [Q4C1] and [Q2B2] are complex queries that require an entire syllogism to express.

In light of this understanding, we have designed our program editor around the form of the syllogism. There are a variety of different concrete forms that can be used to express the abstract form of the syllogism, some of which are discussed in [13]. These include both text-based and diagrammatic notations. One of the most common forms is the one that we have used throughout this paper, that of a three-line stanza: major premise / minor premise / conclusion. Consequently, we have designed our editor with three horizontal panels, roughly corresponding to each of these three statements, as illustrated by the sketch in Figure 2.

In overview, the bottom panel contains the polymorphic method invocation that the programmer wishes to understand; the top panel informs the programmer why this invocation is legal, *i.e.* [Q4C1]; and the middle panel presents what this invocation means, *i.e.* [Q2B2]. The rest of this section describes our design in more detail, using the example source code listed in Figure 1 to illustrate specific points.

The Example. The sketch in Figure 2 shows our proposed program editor working with the source code presented in Figure 1 — a trivial program for the sake of the illustration. The program contains only one polymorphic invocation, from `Main::foo(Shape)` to `Shape::draw()`. The sketch in Figure 2 shows how our proposed editor can help in understanding this polymorphic invocation.

Simply reading the code of this trivial example reveals some obvious facts: the `draw` method is actually declared in `Drawable`, which is a super-class of `Shape`; the `Circle`, `Triangle` and `Rectangle` classes implement the `draw` method; the `Square` class 'inherits' the implementation from `Rectangle`. All of these facts (and more) are formally represented by the proposed program editor.

Bottom Panel. The bottom panel contains the polymorphic method invocation that the programmer wishes to understand, because a method invocation corresponds to the conclusion of the instantiation syllogism. The information presented in the top two panels is context sensitive to the particular invocation under examination in the bottom panel. It is this context-sensitivity that allows our editor to display the results of advanced call graph construction algorithms, *e.g.* [2, 16, 18]. Note that the context sensitivity of the editor does not mandate a context sensitive program analysis.

The bottom panel of Figure 2 is similar to the interface of conventional IDE's: the left-hand side shows the hierarchical structure of the program (organized by package/directory), and the right-hand side shows the text for the selected program element. In the example, we can see that `Main::foo(Shape)` is selected in the hierarchy on the left and its code is displayed on the right. The polymorphic invocation `s.draw()` is highlighted as the current editing point; the information presented in the top and middle panels is with respect to this editing point.

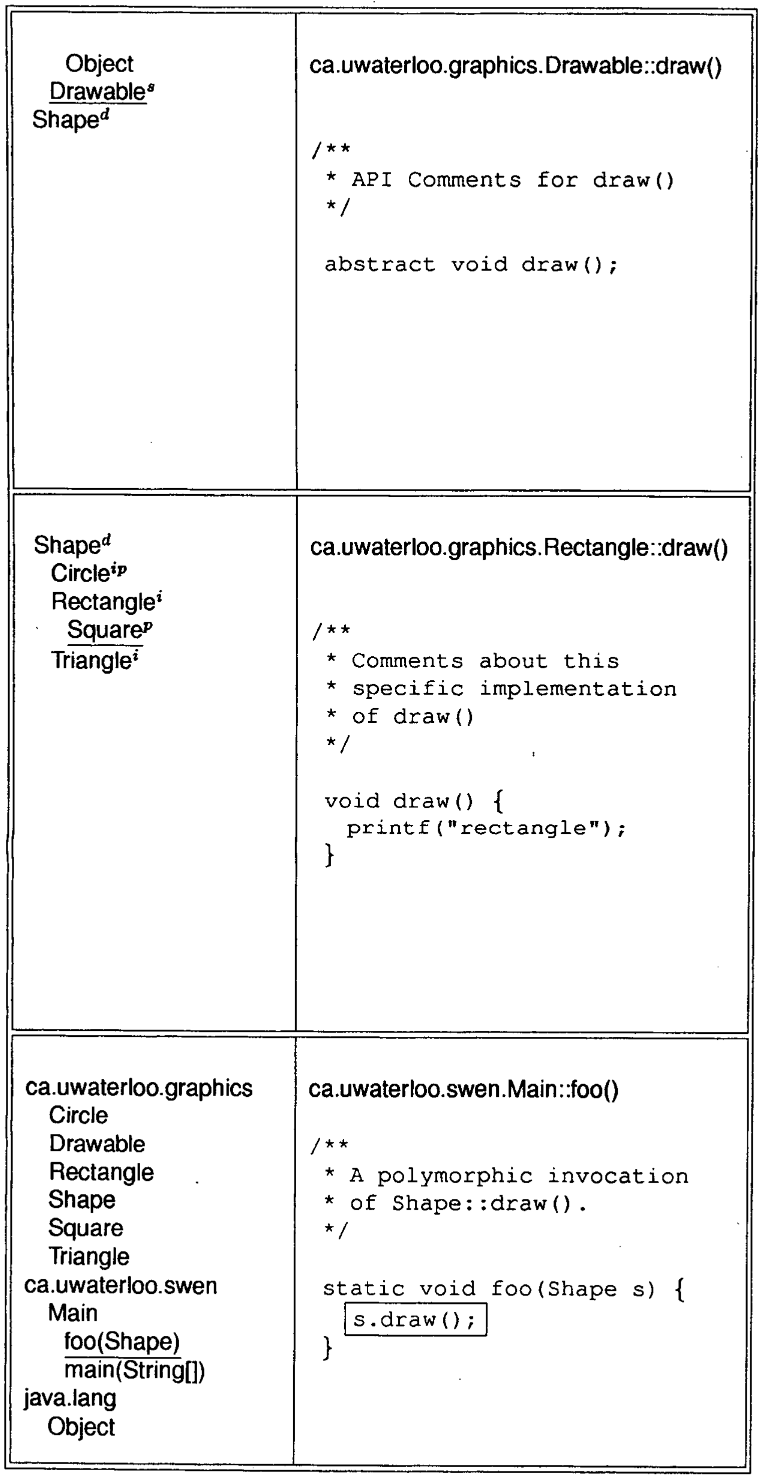
Top Panel. The purpose of the top panel is to show where the method invoked in the bottom panel (*e.g.* `draw`) is introduced into the hierarchy, and to show the API documentation for it. The left-hand side shows all super-classes of the declared type of the variable selected in the bottom panel: in the example, variable `s` is selected, which is of type `Shape`, and so the super-classes of `Shape` are shown here in the top panel. It is possible to use this space to show multiple supertypes (*i.e.* 'multiple inheritance'), and one way to do this is to use an inverted hierarchy, as we have suggested. There are, no doubt, other ways in which that information could be conveyed in this space.

The (inverted) hierarchy on the left-hand side is annotated in two ways: to distinguish the *declared* type of the variable selected in the bottom panel (*e.g.* `Shape`), and to distinguish the *signature* type (*e.g.* `Drawable`) of the method invoked on that variable (*e.g.* `draw`).

The right-hand side of the top panel displays the text *correlating* to the element selected on the left-hand side: in the example, `Drawable` is selected, so the text for `Drawable::draw()` is displayed (and this is where the API for `draw` is documented). This differs slightly from the common convention, because it is not the text of the *selected* element (*e.g.* `Drawable`) that is displayed but, rather, the text of the *correlating* element (*e.g.* `Drawable::draw()`).

Middle Panel. The purpose of the middle panel is to show all of the potential target implementations of the polymorphic method invocation selected in the bottom panel. The middle panel is similar to the top panel, except that it displays the sub-classes of the variable's declared type, rather than the super-classes. The hierarchy on the left-hand side is annotated in three ways: to distinguish the *declared* type of the variable selected in the bottom panel (*e.g.* `Shape`), to distinguish the classes that *implement* the method invoked on that variable (*e.g.* `Circle`, `Rectangle` & `Triangle`), and, to distinguish the classes that may *potentially* reach the invocation statement selected in the bottom panel (*e.g.* `Circle` & `Square`). *Rapid Type Analysis* [2] is sufficient to compute the potential types for the given example program; a more accurate analysis may also be used, *e.g.* [16, 18].

The Call Stack. Figure 2 shows our proposed editor displaying a single polymorphic method invocation: in other words, a call stack of depth one. Our design can easily be generalized for call stacks of arbitrary depth by adding a UI control for the programmer to transfer a method from being viewed in the middle (or top) panel to being viewed in the bottom panel. Another UI control can list these transfers, each of which represents another slot in the stack.



- Object
  - Drawable^s
- Shape^d

ca.uwaterloo.graphics.Drawable::draw()

```
/**
 * API Comments for draw()
 */

 abstract void draw();
```

LHS:

- inverted super-class hierarchy [Q1B1] (super-classes of the declared class of variable s)
- distinguish *declared* class (^{d}) [Q3B1]
- distinguish *signature* class (^{s}) [Q2C1]
- distinguish selected class

RHS:

- signature and source code of method corresponding to class selected on LHS
- API comments (i.e. 'javadoc')

- Shape^d
  - Circle^{ip}
  - Rectangle^i
    - Square^p
  - Triangle^i

ca.uwaterloo.graphics.Rectangle::draw()

```
/**
 * Comments about this
 * specific implementation
 * of draw()
 */

 void draw() {
   printf("rectangle");
 }
```

LHS:

- sub-class hierarchy [Q1B2] (sub-classes of the declared class of variable s)
- distinguish *declared* class (^{d}) [Q3B1]
- distinguish *implementing* classes (^{i}) [Q2B2] and [Q2B1]
- distinguish *potential* classes (^{p})
- distinguish selected class

RHS:

- signature and source code of method implementation corresponding to class selected on LHS
- implementation specific comments

- ca.uwaterloo.graphics
  - Circle
  - Drawable
  - Rectangle
  - Shape
  - Square
  - Triangle
- ca.uwaterloo.swen
  - Main
    - foo(Shape)
    - main(String[])
- java.lang
  - Object

ca.uwaterloo.swen.Main::foo()

```
/**
 * A polymorphic invocation
 * of Shape::draw().
 */

 static void foo(Shape s) {
   s.draw();
 }
```

LHS:

- containment hierarchy [Q1A1]
- distinguish selected item

RHS:

- signature and source code of item selected on LHS
- distinguish editing point (cursor)

Figure 2. Sketch (left) and summary description (right) of proposed program editor

7 Conclusion

In this paper, we have reasoned from Aristotle's *Prior Analytics* [1], the treatise that defined the form of the syllogism and thereby began the study of formal logic, to the design of a novel editor for object-oriented programs.

In pursuing this line of reason, we have developed an understanding of polymorphic method invocations within the context of syllogistic logic. We have also explored the design space of program analyses related to the program's syllogistic structure, and found that this space includes such analyses as *Unique Name* [3], *Class Hierarchy Analysis* [5], *Class Hierarchy Slicing* [17], *Class Hierarchy Specialization* [19], and *Rapid Type Analysis* [2]. In general, the syllogism is only sufficient to describe the basic idea of these analyses, and more intricate formalisms must be developed to implement them in actual programming languages.

We have applied our understanding of polymorphic method invocations to the design of a novel editor for object-oriented programs. This editor can display a polymorphic call graph, constructed with analyses such as [2, 16, 18]. Displaying a polymorphic call graph is a substantially more difficult problem than displaying a non-polymorphic call graph, and we are not aware of any other aide that performs this function as well as our proposed editor.

This paper advances earlier work on the syllogistic structure of object-oriented programming by Sowa [14] and by Rayside & Campbell [12] with a more in depth examination of the structural similarity between object-oriented programming and the syllogism. We support their conclusion that if object-oriented programming is easy to understand, it is because of its similarity with syllogistic logic. The syllogism, specifically in the first figure, universal affirmative mood, is the formalism at the very beginning of logic: it is simple, it is well known, and it is considered perfect by many (including Aristotle). It seems to be a very reasonable structure to base a class of programming languages on.

Future Work. In the future, we would like to build a working prototype of the proposed editor and conduct some empirical studies of its effectiveness, both for expert programmers and for teaching. A more expansive future version of this work may give greater consideration to languages such as BETA [11], SELF [20], SMALLTALK [7] and CECIL [4].

Acknowledgments

We thank Paul Asman, Erik Ernst, Scott Kerr, Thomas Kühne, Margaret-Anne Storey, Peter Sweeney, Tarja Systä, Tarver Szwejkowski, Kresten Krab Thorup, Mads Torgersen and the anonymous referees for commenting on earlier drafts of this paper. The first author would also like to acknowledge the encouragement of Gerard Campbell, Karel Driesen and Doug Lea. Any errors or omissions remain the responsibility of the first author.

References

[1] Aristotle. Prior Analytics. In W. D. Ross, editor, *The Works of Aristotle, Volume 1: Logic*. Oxford University Press, 1928.

[2] D. F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California at Berkeley, December 1997. Supervised by Susan Graham. UCB/CSD-98-1017.

[3] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *POPL'94*, pages 397–408, Portland, Oregon, January 1994.

[4] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP'92*, Utrecht, The Netherlands, July 1992. Springer-Verlag.

[5] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *ECOOP'95*, Århus, Denmark, August 1995. Springer-Verlag. LNCS 952.

[6] K. Driesen. *Software and Hardware Techniques for Efficient Polymorphic Calls*. PhD thesis, University of California at Santa Barbara, June 1999. Supervised by Urs Hölzle. TRCS99-24.

[7] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[8] J. Gosling, B. Joy, and G. Steele Jr. *The Java Language Specification*. Addison Wesley, 1996.

[9] D. Lea, editor. *Proceedings of ACM/SIGPLAN Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, Minneapolis, Minnesota, October 2000.

[10] J. Łukasiewicz. *Aristotle's Syllogistic From the Standpoint of Modern Formal Logic*. Oxford Clarendon Press, 2^{nd} edition, 1957.

[11] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-oriented Programming in the* BETA *Programming Language*. Addison Wesley, 1993.

[12] D. Rayside and G. T. Campbell. An Aristotelian Understanding of Object-Oriented Programming. In Lea [9], pages 337–353.

[13] L. E. Rose. *Aristotle's Syllogistic*. Charles C. Thomas Publisher, 1968.

[14] J. F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole Thomson Learning, 2000.

[15] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[16] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, and E. Gagnon. Practical virtual method call resolution for Java. In Lea [9], pages 264 – 280.

[17] F. Tip, J.-D. Choi, J. Field, and G. Ramalingam. Slicing class hierarchies in C++. In J. Coplien, editor, *OOPSLA'96*, pages 179 – 197, San Jose, California, October 1996.

[18] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In Lea [9], pages 281 – 293.

[19] F. Tip and P. F. Sweeney. Class hierarchy specialization. In T. Bloom, editor, *OOPSLA'97*, pages 271–285, Atlanta, Georgia, October 1997.

[20] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA'87*, pages 227–241, Orlando, Florida, October 1987.