Mendel: A Model, Metrics, and Rules to Understand Class Hierarchies

Simon Denier and Yann-Gaël Guéhéneuc
Ptidej Team – GEODES – DIRO
University of Montreal, Quebec, Canada

E-mail: {deniersi,guehene}@iro.umontreal.ca

Abstract

Inheritance is an important mechanism when developing object-oriented programs with class-based programming languages: it enables subtyping, polymorphism, and code reuse. Inheritance is also known as a difficult feature to grasp and to use correctly because of its many purposes. We propose a model of inheritance to help understand class hierarchies of class-based object-oriented programs. We define metrics and rules to highlight interesting classes and behaviours with respect to inheritance. Thus, we provide the programmer with insight on how inheritance is used in a program. We illustrate our approach on JHotDraw *and validate it further on three other programs: ArgoUML, Azureus, and Log4J. We also show that our model can describe existing rules, such as micro patterns.*

1. Introduction

Inheritance is an important feature when developing object-oriented programs with class-based programming languages, because it enables subtyping [1], polymorphism [6], and code reuse [23]. In the past decades, inheritance has been put forward [4] as a means to enhance reusability, to provide specialization and generalization, and to define hierarchical taxonomies of real world objects. However, it is known to be a a difficult feature to grasp and use correctly [4, 20].

Our goal is to help programmers understand the uses of inheritance in their programs by providing Mendel, a systematic approach to perform inheritance analysis in programs. The two objectives of our approach are, first, to design a simple yet detailed-enough model to allow mappings from different programming languages; and, second, to contribute metrics and rules to qualify classes and behaviours to help programmers in answering: *What are the interesting classes in an unknown program or framework?* and *What particular relationships exist between a subclass and its superclass?*

Our model combines data provided by the inheritance tree and by the public interface of classes. Methods in a class are classified according to their inheritance status: new, inherited, specialised, and replaced. We propose metrics and rules based on this model to classify interesting classes in the program as *large*, *budding*, and *blooming* classes, as well as to identify different subclassing and overriding behaviours.

We illustrate Mendel using the running example of JHotDraw. We show that using our approach we are able to isolate the *base* classes documented by the authors of the framework. We also apply and validate our approach on three other programs: ArgoUML, Azureus, and Log4J. We show that our model is able to highlight interesting classes as well as different subclassing and overriding behaviours.

In the rest of this paper, Section 2 presents related work. Section 3 introduces the running example of JHotDraw and further defines our research objectives. Section 4 introduces our model and associated definitions. Section 5 contains the definitions of metrics and rules used to understand and assess the uses of inheritance. Section 6 presents experiments done on four programs. Section 7 discusses some extensions of our model to accommodate previous work, such as client driven characterisation and micro patterns. Section 8 concludes and presents future work.

2. Related Work

The work presented in this paper relates to several areas of software engineering: (1) the definition, models, and properties of inheritance in programming languages; (2) the rules on "good" uses of inheritance; and the use of inheritance to (3) assess software quality and (4) identify behaviors and defects in programs.

Inheritance is one of the main concept introduced by the object-oriented paradigm. In class-based programming languages, inheritance plays three major roles. First, inheritance allows subtyping [1]. In conjunction with method overriding, it also enables class specialization and inclusion polymorphism [6]. Second, inheritance allows reusing code [23]: common code can be factored out in superclasses for the benefit of subclasses, avoiding code duplication. Third, inheritance allows defining taxonomies of objects, to express the conceptual relationships among the modelled objects and real-world objects. Programming languages may offer various flavors of single, multiple, or mixin inheritance, depending on the rules governing its roles [21].

Although fundamental to object-oriented programming languages, inheritance is well-known for being difficult to grasp and to use correctly by beginners [4, 20]. In their book [12, page 20], Gamma *et al.* issued a warning and suggested to "favor object composition over class inheritance". Consequently, several educators and researchers provided guidelines on the uses of inheritance. For example, Brown and Fowler [5, 11] described typical examples of "bad" uses of inheritance and suggest corrective measures. Recently, Gil and Maman [14] introduced the concept of micro patterns in Java, where several micro patterns are directly concerned with inheritance. They show that micro patterns, such as Implementor and Overrider, occur frequently in programs like Sun JRE and JBoss Server.

Due to its importance in object-oriented languages, several works offered metrics related to inheritance to understand its impact on the quality of programs. For example, the object-oriented metric suite by Chidamber and Kemerer [8] includes the metrics Depth of Inheritance Tree and Number Of Children. Lorenz and Kidd [18] defined metrics to count the number of inherited, overridden, or added methods in a subclass. Beyer *et al.* [3] used some *flattening* functions to take inheritance into account when computing classical metrics like size, coupling, or cohesion. By comparing flattened and unflattened metrics, they can find some particular subclassing behaviors. Our work offers the same metrics and uses them to build new ones. Li-Thiao-Té *et al.* [17] defined special metrics to detect misuses of inheritance related to overriding. Recently, Mihancea [19] proposed metrics to characterise hierarchies based on the use of their classes by clients. The metrics allow characterising a hierarchy in terms of uniformity and detecting anomalies. Our approach is complementary to this previous work, as further discussed in Section 7.

Previous work also highlighted the problems faced by programmers when working with frameworks (or any large program): mapping, understanding functionalities, and understanding the framework architecture [16]. Understanding the role played by classes through inheritance when understanding framework is important, as shown for example in [15] where programmers often asked questions about *inherited feature* (45 over 300 questions) and *variation points* (what method to use or override) because the framework studied (Java Swing) was *heavily inheritance-based.* Demeyer *et al.* [9] proposed a hybrid approach combining metrics and visualisation to understand object-oriented programs. Ducasse and Lanza [10] built on this previous work *class blueprints*, a visualisation of class metrics. They defined visual patterns to detect behaviors and defects, related to inheritance among others. Our approach has similar goals. However, class blueprints require some expertise to identify visual patterns, while we want to provide information to start the comprehension process for programmers without any know-how.

3. Problem and Definitions

We build on previous work to propose a new model, metrics, and rules to help programmers understand the uses of inheritance in their programs.

Running Example. We illustrate this paper using JHotDraw v5.2. JHotDraw is a framework to build 2D vector-based graphical editors [13]. It has been originally designed by Gamma and Eggenschwiller to illustrate the use of several design patterns and to be easily extended, using both the inheritance and composition mechanisms. A programmer willing to understand JHotDraw would need to identify and understand the main classes of the framework, their methods, and how she should use, redefine, or override these methods to implement her editor [15]. To help her in understanding JHotDraw, Gamma and Eggenschwiller provided an overview of the main Java interfaces in the form of a UML-like class diagram, adapted in Figure 1.

JHotDraw Base Classes. The class diagram in Figure 1 highlights the six classes (out of 148) in the program that should be the starting point of any comprehension effort. Each of the classes shown in the diagram is the direct implementation of one of the six Java interfaces provided by Gamma and Eggenschwiller. We focus on these classes because they should be understood by the programmer to grasp the "mechanics" of JHotDraw. We call *base* this set of classes, namely `DrawApplication`, `StandardDrawingView`, `StandardDrawing`, `AbstractFigure`, `AbstractHandle`, and `AbstractTool`.

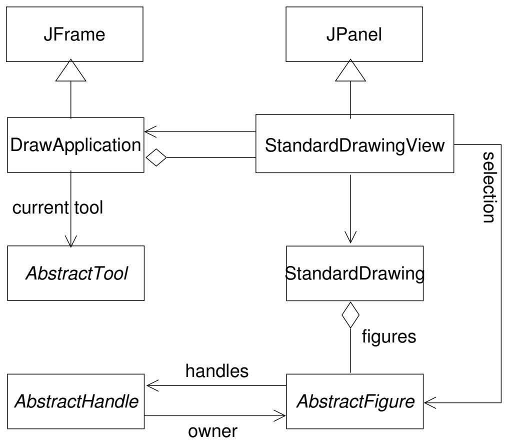

Figure 1. JHOTDRAW **UML-like class diagram, adapted from [13]**

Problem Statement. Unfortunately, few other real-world programs and frameworks provide such documentation [22], and the example of JHOTDRAW illustrates the first question that we would like to help programmers in answering: *What are the interesting classes in an unknown program or framework?*

By virtue of inheritance, subclasses hold special bonds with their superclasses. Some bonds indicates how a subclass–superclass couple should behave, such as specialisation or extension. It is of great help for the programmer to know and understand such behaviour. Thus, the second question is: *What is the relationship between a subclass and its superclass?*

Therefore, we propose a language-independent model of inheritance and then define metrics and rules that help programmers answer these questions. We focus on the five following concepts.

Large Classes. A simple way to identify interesting classes in a program is by the numbers of methods that they declare locally. Indeed, the more methods a class defines, the more it contributes to the program. We call *large class* any class with a large interface compared to the other classes in the program. We use the number of methods declared in a class rather than the number of lines of code because methods can be arbitrarily long while the number of methods provide information on the services that the class supports.

Budding Classes. Classes that provide more methods than their superclasses are likely to be interesting, with respect to inheritance. Thus, they can be understood relatively independently of their superclasses. We call *budding classes* these classes standing out locally in their inheritance tree, but not necessarily visible at first sight in the whole program.

Blooming Classes. Finally, we define *blooming classes* as classes that seem to "blossom" in the inheritance tree. They are often both large classes (declaring many methods locally) and budding classes (adding a lot of methods with respect to their superclasses) and are prime choice to begin understanding a program.

Subclassing Behaviours. There are different ways for a class to inherit from its superclasses. For example, a class can extend its superclass with new methods or override some existing methods. Therefore, it is also important to identify such *subclassing behaviours* to understand the incremental difference between a class and its superclass.

Overriding Behaviours. Similarly to subclassing behaviour, but finer-grain, we introduce the concepts of *overriding behaviours* to characterise classes specialising or replacing methods of their superclasses—because specialisation and replacement are important features of inheritance with different impacts on the class–superclass couple.

Approach for the Programmer. With our model, a programmer could focus first on large, budding, and blooming classes to get a good understanding of the overall inheritance tree of a program. Subclassing and overriding behaviours would then help the programmer understand the relationship of a class to its superclass more finely, in particular if the superclass is itself a large, budding, or blooming class.

4. Model Definition

Our model divides into two levels. The first level describes the tree formed by a set of classes in a hierarchy. It provides selection of classes within the tree. The second level describes classes as an interface with different subsets of methods, based on their inheritance status. It provides definition of those subsets.

In our model, we assume that a class is identified by a name and provides a set of methods through its interface. It is in relationship with a superclass similarly defined. We only consider single inheritance for the sake of simplicity, because multiple inheritance has many different semantics [7]. Java interfaces are also ignored because our model focuses on concrete methods and subclassing behaviours.

We call *classes* the set of all classes defined in a program. It includes classes defined in imported libraries, in case a class inherits from a superclass defined in a

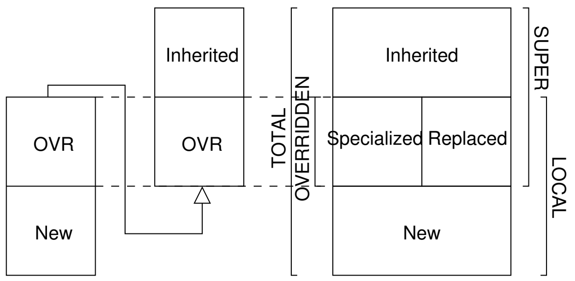

Figure 2. Method Subsets in the Class Model

library. However, our approach is still applicable if we choose not to include libraries. This can eliminate, at the expense of completeness, some obfuscation of program classes by large library classes (such as Swing classes).

4.1. Tree Model

The tree model focuses on describing relationships between different classes connected through inheritance. We define the *inherits* relation as follow:

$$\begin{aligned} &\mathit{inherits}(c,s) \text{ if } c \in \mathit{classes} \wedge s \in \mathit{classes} \\ &\qquad \wedge\, s \text{ is the superclass of } c \end{aligned}$$

We call *parents* the set of direct superclasses of a class (of cardinality 1 in single inheritance). The *ancestors* of a class is the transitive closure on *parents* (noted $*$), up to the root of the inheritance tree. The *branch* of a class includes the class and its ancestors.

$$\begin{aligned} \mathit{parents}(c) &= \{s : s \in \mathit{classes} \wedge \mathit{inherits}(c,s)\} \\ \mathit{ancestors}(c) &= \mathit{parents}*(c) \\ \mathit{branch}(c) &= \{c\} \cup \mathit{ancestors}(c) \end{aligned}$$

4.2. Class Model

The local interface of a class is the set of its *locally* declared public methods (excluding inherited and private methods, including protected methods), each method being identified by its unique signature [2]. The selector of the method, usually composed from its name and its parameters types, is used as its signature.

For a class $c \in \mathit{classes}$, we define its pseudo-superclass $s_{flat} = \mathit{flatten}(\mathit{ancestors}(c))$, where $\mathit{flatten}$ means that s_{flat} is a single class in which all methods from the ancestors of c appear if and only if they are visible and accessible from c.

Local Subset LocS**:** all methods defined by c.

Super Subset SupS**:** all methods defined by s_{flat}.

Total Subset TotS**:** all methods that c can answer, $\mathit{TotS} = \mathit{LocS} \cup \mathit{SupS}$.

The purpose of the class model is to distinguish subsets of methods in the interface of a class: new, inherited, specialised, and replaced. These subsets are used to compute metrics and rules.

New Subset NewS**:** methods only declared in c and not in s_{flat}, $\mathit{NewS} = \mathit{LocS} \backslash \mathit{SupS}$.

Inherited Subset InhS**:** methods purely inherited from s_{flat}, *i.e.*, that are not declared in c, $\mathit{InhS} = \mathit{SupS} \backslash \mathit{LocS}$.

Overridden Subset OvrS**:** methods defined in s_{flat} and redefined in c, $\mathit{OvrS} = \mathit{LocS} \cap \mathit{SupS}$.

Specialized Subset SpcS**:** methods of c specialising methods inherited from s_{flat} (methods calling their overridden parent method).

Replaced Subset RplS**:** methods of c replacing methods inherited from s_{flat}, *i.e.*, methods overriding methods from s_{flat} but not calling the overridden methods, $\mathit{RplS} = \mathit{OvrS} \backslash \mathit{SpcS}$.

Figure 2 gives a visual representation of the previous subsets. Alternative definitions are of course possible, for example $\mathit{LocS} = \mathit{OvrS} \cup \mathit{NewS}$.

In JHotDraw, we can compute the model of each class. The following table shows sets cardinalities for the class `StandardDrawing`. First are LocS, SupS, and TotS. Those sets can be refined as NewS, InhS, and OvrS. OvrS divides into SpcS and RplS.

`StandardDrawing`					
$\lvert\mathit{LocS}\rvert$	12	$\lvert\mathit{NewS}\rvert$	5	$\lvert\mathit{SpcS}\rvert$	3
$\lvert\mathit{SupS}\rvert$	68	$\lvert\mathit{OvrS}\rvert$	7	$\lvert\mathit{RplS}\rvert$	4
$\lvert\mathit{TotS}\rvert$	73	$\lvert\mathit{InhS}\rvert$	61		

5. Metric and Rule Definitions

We now introduce metrics and rules to identify *large*, *budding*, and *blooming* classes, as well as to characterise different *subclassing* and *overriding* behaviours.

Threshold. Each of the following metrics produces a ranking of *classes*. We use a threshold t to select a small interesting subset of classes IS according to a metric X. We define t as the sum of the mean of X and the standard deviation of X. Thus, our threshold selects only high above average classes yet still retrieves a number of classes relative to the program size.

$$\begin{aligned} t_X &= \overline{X(\mathit{classes})} + \sigma(X(\mathit{classes})) \\ IS_X &= \{c : c \in \mathit{classes} \wedge X(c) > t_X\} \end{aligned}$$

5.1. Large Classes

We first define a threshold $t_{|LocS|}$ to distinguish large classes from other classes, with respect to the size of their local interfaces $|LocS|$. Using this threshold, we define the set of *large classes* $IS_{|LocS|}$.

In JHotDraw, the mean $|LocS|$ is 6.57 and $t_{|LocS|} = 12.04$. The set $IS_{|LocS|}$ contains 15 classes out of 148. Among these 15 large classes, three of the base classes appear and make the top three (cf. Table 1). This confirms their status as interesting class to start understanding the framework.

5.2. Budding Classes

We define a *novelty index* as a metric to identify budding classes. The novelty index nvi is computed on the class and its ancestors. It provides a measure of the contribution of the class to its branch in term of new definitions, *i.e.*, the size of its local subset $LocS$ with respect to its ancestors.

To compute nvi, we first compute the branch mean size, bms, for the ancestors. This is the mean number of new definitions available among the ancestors. We compute bms independently of overriding—we only count the methods that are visible in the class. The set IS_{nvi} defines the *budding classes*.

$$\text{with} \quad bms(c) = \frac{|TotS(c)|}{DIT(c) + 1}$$
$$nvi(c) = \frac{|LocS(c)|}{bms(parents(c))}$$

In JHotDraw, the mean nvi is 0.42 and $t_{nvi} = 0.71$. Above t_{nvi}, we find 25 budding classes out of 148, including 4 of the 6 base classes (cf. Table 1). As explained in Section 3, they can be understood independently from their hierarchy by a programmer.

In Table 1, the high nvi of `AbstractFigure` is due to its large interface combined with the fact that it only inherits from `java.lang.Object`. On the contrary, `DrawApplication` and `StandardDrawingView` have a smaller nvi because they inherit from large classes in the Swing framework, such as `JFrame` and `JPanel`.

The set of budding classes includes `AbstractHandle` and `AbstractTool`, while large and blooming sets do not. This status shows that both classes highly contribute to the interface in their branch, presumably to be refined by their subclasses. Thus, these two classes are interesting classes: they stamp their mark on their hierarchies. Their comparatively low $|LocS|$ shows however that they have less responsibility than the top three *large* classes.

5.3. Blooming Classes

A weighting of the novelty index by the size of the interface $|LocS|$ of a class leads to the definition of a *novelty score*. The novelty score nvs highlights classes with a large interface as well as classes with a smaller interface but having an important impact on their hierarchies. Thus, the novelty score provides a measure of the importance of a class while not relying only on its number of methods.

$$nvs(c) = |LocS(c)| \times nvi(c)$$

In JHotDraw, the mean nvs is 5.36 and $t_{nvs} = 12.19$. The nvs subset includes 14 classes out of 148, among which three base classes (cf. Table 1). These three classes are the same appearing as large classes, although in a different order.

The class `StandardDrawingView` extends the `JPanel` Swing class. Although it is the largest class (55 methods) in JHotDraw, it is overwhelmed by the number of inherited methods from the Swing hierarchy (432 methods). Consequently, its novelty index is low (0.63) and it does not qualify as a budding class. Yet, using the nvs, `StandardDrawingView` is clearly highlighted as a blooming class.

The status of `AbstractFigure` as an interesting class is confirmed by its first place in the blooming set. On the contrary, `StandardDrawing` is the only base class to not appear in any interesting subsets, as shown by its low nvs. Code review shows that `StandardDrawing` inherits from `CompositeFigure`, which ranks in fourth position in the blooming set (cf. Table 2): thus `StandardDrawing` must be understood in conjunction with `CompositeFigure`.

5.4. Subclassing Behaviours

We distinguish in subclassing between *overriding* and *extending* behaviours. An *overriding behaviour* targets specialisation of the superclass. An *extending behaviour* indicates that the prime interest is reuse. We formally define the subclassing behaviours using the following equations:

Pure Overrider $PurOv$**:** set of classes such as $|NewS| = 0 \wedge |LocS| > 0$.

Overrider Ovr**:** set of classes such as $|OvrS| \geq |NewS| \wedge |LocS| > 0$.

Pure Extender $PurExt$**:** set of classes such as $|OvrS| = 0 \wedge |LocS| > 0$.

Base Classes	$\lvert LocS\rvert$	rank	nvi	rank	nvs	rank	$\lvert NewS\rvert$	$\lvert OvrS\rvert$
AbstractFigure	**34**	**3**	**3.09**	**1**	**105.09**	**1**	33	1
AbstractHandle	11	20	**1.00**	**13**	11.00	15	11	0
AbstractTool	10	22	**0.91**	**16**	9.09	19	10	0
DrawApplication	**53**	**2**	**0.76**	**21**	**40.32**	**6**	53	0
StandardDrawing	12	16	0.53	42	6.35	26	5	7
StandardDrawingView	**55**	**1**	0.63	28	**34.61**	**10**	50	5
Threshold	*12.04*		*0.71*		*12.19*		-	-

Table 1. Metrics Results and Ranks for Base Classes in JHOTDRAW. Bold Numbers are Above Threshold of Their Respective Metric.

Extender Ext**:** set of classes such as
$|NewS| > |OvrS| \wedge |LocS| > 0$.

Classes that do not match any of the previous definitions are classified in the set *Other* (in Java, it includes classes with no methods or only static methods). Classes having a *pure* behaviour—overriding or defining new methods only—are especially interesting.

In JHotDraw, we obtain the sets with the following cardinalities:

Set	$PurOvr$	Ovr	$PurExt$	Ext	$Other$
Classes	55	30	34	24	5
Base	0	1	3	2	0

This table shows a majority of pure behaviours in JHOTDRAW, which implies that most classes have a straightforward relationship to their superclasses. Also, the majority of classes refine their superclasses instead of extending them, which shows that polymorphism is well used in JHOTDRAW.

The tendency for the six base classes is on extending. We explain this tendency by the fact that most base classes perform well on *nvi* and *nvs*, providing new methods in their hierarchies. Indeed, the two rightmost columns in Table 1 show that base classes have more methods in $NewS$ than in $OvrS$, except for `StandardDrawing`. Interestingly, `StandardDrawing` is the only Overrider, which confirms its strong link with its superclass `CompositeFigure`.

5.5. Overriding Behaviours

We define five overriding behaviours by distinguishing between specialisation and replacement. Similarly to subclassing behaviours, we separate pure and mixed behaviours. The interpretation of overriding behaviours should be made with care, as we only look at the $OvrS$ set and not, for example, at the $NewS$ set.

Pure Specialiser $PurSpc$**:** set of classes such as
$|RplS| = 0 \wedge |OvrS| > 0$.

Specialiser Spc**:** set of classes such as
$|SpcS| > |RplS| \wedge |OvrS| > 0$.

Pure Replacer $PurRpl$**:** set of classes such as
$|SpcS| = 0 \wedge |OvrS| > 0$.

Replacer Rpl**:** set of classes such as
$|RplS| > |SpcS| \wedge |OvrS| > 0$.

Equals Eq**:** set of classes such as
$|SpcS| = |RplS| \wedge |OvrS| > 0$.

Classes that do not override any method are classified in the set $NoOvr$. This corresponds to the $PurExt$ and *Other* subclassing behaviours.

In JHotDraw, we have the following classification of classes:

Set	$PurSpc$	Spc	Eq	Rpl	$PurRpl$	$NoOvr$
Classes	26	15	17	15	36	39
Base	0	0	0	1	2	3

The pure behaviours are in majority, which means that most classes are consistent when overriding methods. However, we observe the presence of more Pure Replacers than Pure Specialisers and of 37 classes with mixed behaviours (*Spc*, *Eq*, and *Rpl*): this shows that many classes have a tendency to *change* their superclass behaviour (instead of specializing it), which can make understanding the framework more complex.

6. Validation

We first show how to map Java programs to our model. We then discuss further facts on JHOTDRAW and results on three other programs.

6.1. Mapping Java Programs to the Model

We use the BCEL library for bytecode engineering to parse Java class files. BCEL gives flexibility as we can parse any Java program, even in the absence of source code. In particular, we can build the model of any superclass defined in a library. Our BCEL implementation of MENDEL computes model and metrics in 9 ± 2 seconds for a program with 1,600 classes (data for Azureus 2.4 on a 2GHz computer with 1Go memory). However, the inclusion of generics starting with Java 1.5 introduces some changes in classfiles, such as bridge methods, which BCEL does not currently handle.

The root of any inheritance tree in Java is the `java.lang.Object` class. We consider $SupS(Object) = \emptyset$. We ignore Java interfaces as we focus on subclassing concrete classes.

Constructors, private methods, and static methods are not inherited in Java classes. Thus, they are ignored. Other modifiers for visibility (public, protected, package) do not impact the sets definitions.

Another specificity of the Java language is the abstract modifier. The following rules apply: (1) an abstract method is in $LocS$; (2) if an abstract method overrides a method from the superclass, it belongs to $RplS$, otherwise to $NewS$, because making a concrete method abstract prevents subclasses to call it for specialisation; (3) a concrete method defining an abstract method belongs to $SpcS$. It is the responsibility of classes subclassing an abstract class to specialize it with their own methods.

6.2. Blooming Classes in JHOTDRAW

We provide a detailed overview of the blooming classes in JHOTDRAW as this set provides a small interesting selection, with 12 (out of 14) classes also belonging to large and budding sets. Table 2 shows the 14 blooming classes in JHOTDRAW. We include the $|LocS|$ and nvi metrics for comparison. The distribution is even between sets: this means that all large classes are also budding classes, which points to an even distribution of responsibilities.

Most of the blooming classes are special figures in JHOTDRAW, with non-trivial behaviours: `TextFigure`, `LineConnection` (a figure able to connect to other figures), `CompositeFigure`, `PolygonFigure`, `DecoratorFigure`, `PolyLineFigure`. Overall, this indicates that the figure hierarchy is, as expected, the most important piece of software in JHOTDRAW.

Two blooming classes do not appear as large and budding class as the others. `StandardDrawingView` does not appear as a budding class due to inheritance from Swing classes (cf. Section 5.2). `TextTool`, which is not a large class, is a budding class with a sufficient contribution to be blooming.

Class	$\|LocS\|$	nvi	nvs
AbstractFigure	34	3.09	105.09
TextFigure	30	1.84	55.1
LineConnection	32	1.55	49.55
CompositeFigure	31	1.41	43.68
PolygonFigure	26	1.59	41.39
DrawApplication	53	0.76	40.32
DecoratorFigure	29	1.32	38.23
PolyLineFigure	29	1.32	38.23
StandardDrawingView	55	0.63	34.61
PertFigure	24	1.06	25.41
ConnectionTool	15	1.43	21.43
StandardStorageFormat	14	1.27	17.82
GraphicalCompositeFigure	19	0.84	15.93
TextTool	10	1.3	13.04

Table 2. Blooming Classes in JHOTDRAW

The classes `ConnectionTool` (related to `LineConnection`) and `TextTool` (related to `TextFigure`) also belong to the blooming set. The complexity of the connection and text concerns is thus underlined by the presence of these classes.

Table 1 shows that `AbstractHandle` ranks just after the blooming classes, and that `StandardDrawing` comes last of the base classes at the 26th rank. Overall, all base classes are in the upper quartile of our metrics. This justifies a posteriori their presence in the documentation of JHOTDRAW. However, we can make a clear distinction between a primary set of base classes composed of `AbstractFigure`, `DrawApplication`, and `StandardDrawingView`, and a secondary set composed of `AbstractHandle`, `AbstractTool`, and `StandardDrawing` (in close relationship with `CompositeFigure`).

6.3. Programs Comparison

We now show that our model is able to highlight the subclassing and overriding behaviours in different programs by applying our model, metrics, and rules to four different programs and comparing qualitatively and quantitatively the obtained sets. In addition to JHOTDRAW, we also analyse:

Log4J 1.2.1 is a library for application logging, comprising 206 classes and 15,731 lines of code.

ArgoUML 0.19.8 is a UML modeling tool, comprising 1431 classes and 163,126 lines of code.

Azureus 2.4 is an extensible peer-to-peer client, comprising 1681 classes and 222,154 lines of code.

Program	$\|large\|$	$\|budding\|$	$\|blooming\|$	$\|classes\|$
JHotDraw	15	25	14	148
	10%	17%	9%	100%
Log4J	30	29	10	206
	15%	14%	5%	100%
ArgoUML	259	121	45	1431
	18%	8%	3%	100%
Azureus	219	204	97	1681
	13%	12%	6%	100%

Table 3. Interesting Subsets

Program	$PurOvr$	Ovr	$PurExt$	Ext	$Other$
JHotDraw	55	30	34	24	5
	37%	20%	23%	16%	3%
Log4J	29	17	68	37	55
	14%	8%	33%	18%	27%
ArgoUML	575	216	370	117	153
	40%	15%	26%	8%	11%
Azureus	137	115	980	137	312
	8%	7%	58%	8%	19%

Table 4. Subclassing Behaviours

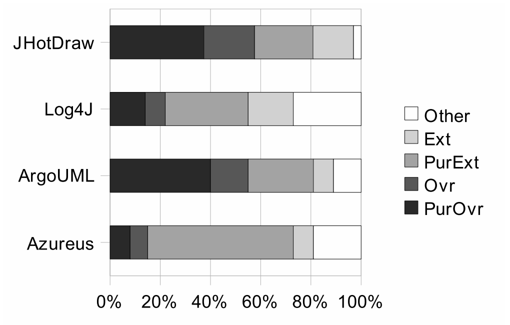

Figure 3. Subclassing Proportions

Program	$PurSpc$	Spc	Eq	Rpl	$PurRpl$	$NoOvr$
JHotDraw	26	15	17	15	36	39
	18%	10%	11%	10%	24%	26%
Log4J	34	12	4	1	32	123
	17%	6%	2%	0%	16%	60%
ArgoUML	491	35	36	71	275	523
	34%	2%	3%	5%	19%	37%
Azureus	164	18	22	23	162	1292
	10%	1%	1%	1%	10%	77%

Table 5. Overriding Behaviours

We observe in Table 3 that the class selection is much smaller for blooming classes than for large and budding classes. This high selectivity confirms the usefulness of the novelty score metric. However, the selection can still be quite extensive for large programs: a programmer could rely on the ranking given by the metric to get a smaller selection.

The case of ArgoUML stands out with a small selection on budding and blooming classes compared to other programs. Value examination shows high deviation of nvs between blooming classes, with the first two classes (`NSUMLModelFacade`, `FacadeMDRImpl`) defining each more than 300 methods. Their names hint toward the Façade design pattern: we thus highlight a fundamental architecture choice in ArgoUML.

We also observe a very large class in Log4J: `LogBrokerMonitor` provides more than 100 methods. Other data shows that the `Appender` hierarchy plays a key role in the library.

The ditribution in Azureus is more even with a number of large classes and few standing out in nvi. According to their recurrence in the names of the blooming classes, download, peer, and the DHT protocol are among the first key concerns.

Table 4 shows a clear dichotomy between two subclassing behaviours: JHotDraw and ArgoUML makes a good use of inheritance for polymorphic behaviour by overriding. Log4J and Azureus display a tendency to extend classes. In particular, 77% of Azureus classes ($PurExt \cup Other$) do not override any methods at all. This confirms previous findings in [24], which mentions that Azureus "barely uses inheritance".

Figure 3 shows the proportions in the four programs for each subclassing behaviours. The distinctive pattern of overriding in JHotDraw and ArgoUML versus extending in Log4J and Azureus is apparent.

In Table 5, all programs except JHotDraw show a tendency to favor pure behaviours. However, this tendency is less strong in the case of Log4J and Azureus, which only make a small use of overriding. ArgoUML stands out as a Specialiser champion, which shows that its design is strongly organized for polymorphism.

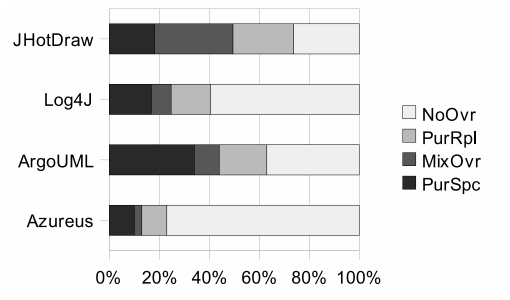

Figure 4. Overriding Proportions

Figure 4 shows the different proportions in the four programs for each overriding behaviours. Mixed behaviours are summarized by $MixOvr = Spc \cup Eq \cup Rpl$. The even distribution between behaviors in JHotDraw is apparent, whereas ArgoUML displays a dichotomy between $PurSpc$ and $NoOvr$. The predominance of $NoOvr$ in Log4J and Azureus is clearly visible.

7. Discussions

Our model is simple and can be easily extended to take into account previous work. It is thus more general than and complementary to previous work on micro patterns [14] and client-driven characterisations [19].

7.1. Client-driven Characterisation

Similarly to Mihancea [19], we can distinguish between two extremes subclassing behaviours, *polymorphic behaviour* and *extending behaviour*.

Polymorphic behaviour implies that a class is meant to be used in place of its superclass. Any reference to the superclass can hold an instance of the class. Hence a method call on the superclass is dispatched to the class "down" in the hierarchy. The class can override methods to specialise the behaviour of its superclass. It does not define new methods because these cannot be called using a reference on the superclass. This is the typical behaviour of taxonomies and frameworks, with the Template Method design pattern being the standard example of such a behaviour. In our model, Pure Overrider classes are the most keen to display *polymorphic behaviour*.

Extending behaviour implies that a class is meant to be used by itself. It is typical of classes that subclass only for reuse. These classes call methods "up" in their hierarchy to define their own behaviour. They mostly provide new methods. Hence, they can only be used through direct references, as no superclass references or overriding methods (accessible by the superclass) can call their new methods. Thus, in our model, Pure Extender classes display *extending behaviour*.

These two behaviours are similar to the *code reuse perspective* and *interface reuse perspective* defined by Mihancea; the main difference being that in our model we only consider class hierarchies, while Mihancea considers also the uses of the classes by their clients. However, Mihancea approach relies on program analyses based on call graph or data flow, which make it sensitive to these analyses. Our approach provides a simpler model and only relies on a parser or reverse engineering tools to build the model. Therefore, these two approaches are complementary.

7.2. Micro Patterns

Our model can be extended to accommodate more specific properties of the interfaces of classes. For example, we can divide each defined subsets into two subsets of methods to distinguish abstract from concrete methods. We divide the set $LocS$ into the subsets $AbsLocS$ and $ConLocS$, such as $LocS = AbsLocS \cup ConLocS$. This extension of our model is *pure*, because previous relations are unchanged, including the definitions of $LocS$, yet it allows to define new metrics and rules. In particular, this extension allows expressing in our model some of the micro patterns proposed previously [14] and related to inheritance. The three micro patterns in the *Inheritors* category can be precisely expressed with the following equations:

Implementor: set of classes such as
$LocS(c) = AbsC(parents(c)) \wedge AbsC(c) = \oslash$.

Overrider: set of classes such as
$LocS(c) = OvrS(c)$
$\wedge\ OvrS(c) \cap AbsC(parents(c)) = \oslash$.

Extender: set of classes such as
$NewS(c) \neq \oslash \wedge OvrS(c) = \oslash$.

Other micro patterns related to inheritance can also be expressed in this extension of our model but less precisely, *i.e.*, we cannot distinguish certain micro patterns, because we would need to include field declarations. Yet, we can define equations that provide *candidate* classes for the given micro patterns:

Designator Candidate: set of classes such as
$TotS(branch(c)) = LocS(Object) \wedge LocS(c) = \oslash$.

Taxonomy Candidate: set of classes such as
$LocS(c) = \oslash$.

Function Pointer/Function Object Candidate: set of classes such as $|LocS(c)| = 1$.

Outline/Trait Candidate: set of classes such as
$AbsC(c) \neq \oslash$.

Pure Type/Pseudo Class Candidate: set of classes such as $LocS(c) = AbsC(c) \wedge LocS(c) \neq \oslash$.

8. Conclusion and Future Work

We presented MENDEL, an approach to help programmers understand the uses of inheritance in their programs, based on a simple model of inheritance focused on methods in classes. Methods were classified according to their inheritance status: new, inherited, specialised, and replaced. Using this model, we defined metrics and rules to identify in class hierarchies *large classes*, *budding classes*, and *blooming classes* as well as *subclassing* and *overriding* behaviours in classes.

We illustrated MENDEL using the running example of JHOTDRAW to show that our model, novelty metrics, and rules are able to highlight interesting classes. We also showed that MENDEL could help in identifying interesting classes in ArgoUML, Azureus, and Log4J.

Therefore, MENDEL helps in answering two important questions: *What are the interesting classes in an unknown program or framework?* using the sets and rankings of large, budding, and blooming classes; and *What particular relationships exist between a subclass and its superclass?* using the concepts of *subclassing* and *overriding behaviours*.

Among other research directions, we are currently extending our model to characterise families of subclassing and overriding behaviours, depending on the prevalence of certain behaviours in class hierarchies. Although we focus on metrics in this article, method sets in the MENDEL model also convey semantic information provided by method signatures: we plan to use this information for fine-grained analyses. We will also integrate our model with previous work, in particular Mihancea's [19], to further improve the characterisation of class hierarchies, and to compare MENDEL with other existing metrics. Finally, we plan to map our model to other programming languages, such as Smalltalk or C♯.

Results. All results are available online at `http://www-etud.iro.umontreal.ca/~deniersi/icpc08`.

Acknowledgements. We would like to thank the anonymous reviewers for their comments and the provided references. This work has been partially funded by the Égide Lavoisier fund (France), NSERC, and FCI (Canada).

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, second edition, 1998.

[2] G. Antoniol, Y.-G. Guéhéneuc, E. Merlo, and P. Tonella. Mining the lexicon used by programmers during software evolution. In *Proceedings of the* 23^{rd} *International Conference on Software Maintenance*. IEEE Computer Society Press, October 2007.

[3] D. Beyer, C. Lewerentz, and F. Simon. Impact of inheritance on metrics for size, coupling, and cohesion in object-oriented systems. In *IWSM*, volume 2006 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2000.

[4] R. Biddle and E. Tempero. Explaining inheritance: A code reusability perspective. In *SIGCSE Bulletin*, 28(1):217–221, March 1996.

[5] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. M. III, and T. J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1^{st} edition, March 1998.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. In *Computing Surveys*, 17(4):471–522, December 1985.

[7] B. Carré and J.-M. Geib. The point of view notion for multiple inheritance. In *Proceedings of the* 20^{th} *European Conference on Object-Oriented Programming and of the 5th conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 312–321. Springer Verlag and ACM Press, October 1990.

[8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. In *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[9] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering approach combining metrics and program visualization. In *Proceedings of the* 6^{th} *Working Conference on Reverse Engineering*, pages 175–186, 1999.

[10] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. In *IEEE Transactions on Software Engineering*, 31(1):75–90, 2005.

[11] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1^{st} edition, June 1999.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.

[13] E. Gamma *et al.* JHotDraw. Web site, 2001. http://www.jhotdraw.org/.

[14] Y. Gil and I. Maman. Micro patterns in java code. In *Proceedings of the* 20^{th} *Conference on Object-Oriented Programming Systems Languages and Applications*, pages 97–116. ACM Press, October 2005.

[15] D. Hou, K. Wong, and H. J. Hoover. What can programmer questions tell us about frameworks? In *Proceedings of the* 13^{th} *International Workshop in Program Comprehension*, pages 87–96. James R. Cordy and Harald Gall, May 2005.

[16] D. Kirk, M. Roper, and M. Wood. Identifying and addressing problems in framework reuse. In *Proceedings of the* 13^{th} *International Workshop on Program Comprehension*, pages 77–86. IEEE Computer Society Press, May 2005.

[17] P. Li-Thaio-Té, J. Kennedy, and J. Owens. Assessing inheritance for the multiple descendant redefinition problem in oo systems. In *Proceedings of the 1997 International Conference on Object Oriented Information Systems*, pages 197–210, 1997.

[18] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: a Practical Guide*. Prentice-Hall, 1994.

[19] P. F. Mihancea. Towards a client driven characterization of class hierarchies. In *Proceedings of the* 14^{th} *International Conference on Program Comprehension*, pages 285–294. IEEE Computer Society Press, June 2006.

[20] I. Milne and G. Rowe. Difficulties in learning and teaching programming—views of students and tutors. In *Education and Information Technologies*, 7(1):1360–2357, March 2002.

[21] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *Proceedings of the* 17^{th} *European Conference on Object-Oriented Programming*, pages 248–274. Springer Verlage, July 2003.

[22] M. J. C. Sousa and H. M. Moreira. A survey on the software maintenance process. In *Proceedings of the* 14^{th} *International Conference on Software Maintenance*, pages 265–274. IEEE Computer Society Press, November 1998.

[23] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of the* 11^{th} *conference on Object-Oriented Programming Systems, Languages and Applications*, pages 268–285. ACM Press, October 1996.

[24] S. Vaucher and H. Sahraoui. Do software libraries evolve differently than applications? In *Proceedings of the* 3^{rd} *ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD'07)*, October 2007.