# Towards a Client Driven Characterization of Class Hierarchies

Petru Florin Mihancea

LOOSE Research Group
"Politehnica" University of Timişoara, Romania
E-mail: `petrum@cs.upt.ro`

## Abstract

*Object-oriented legacy systems are hard to maintain because they are hard to understand. One of the main understanding problems is revealed by the so-called "yo-yo effect" that appears when a developer or maintainer wants to track a polymorphic method call. At least part of this understanding problem is due to the dual nature of the inheritance relation* i.e.*, the fact that it can be used both as a code and/or as an interface reuse mechanism. Unfortunately, in order to find out the original intention for a particular hierarchy it is not enough to look at the hierarchy itself; rather than that, an in-depth analysis of the hierarchy's clients is required. In this paper we introduce a new metrics-based approach that helps us characterize the extent to which a base class was intended for interface reuse, by analyzing how clients use the interface of that base class. The idea of the approach is to quantify the extent to which clients treat uniformly the instances of the descendants of the base class, when invoking methods belonging to this common interface. We have evaluated our approach on two medium-sized case studies and we have found that the approach does indeed help to characterize the nature of a base class with respect to interface reuse. Additionally, the approach can be used to detect some interesting patterns in the way clients actually use the descendants through the interface of the base class.*

**Keywords:** inheritance relation, design metrics, object-oriented design, software understanding, static analysis.

## 1. Introduction

Many object-oriented developers hate maintenance. Is it because it is perceived as a dumb job that does not involve much intellectual activity? On the contrary: it is hated because of its difficulty! Consequently the understanding and quality assessment of object-oriented systems have become vital concerns in today's software industry.

In this context, inheritance is both the "beauty" of object-oriented design and at the same time the "beast", when it comes to maintain or reengineer it. It is the beauty when we design or discover some high level policies which can then be reused in different contexts. However, it usually starts as being the beast because it makes the system hard to understand, due to the so-called "yo-yo effect" [4]. The "yo-yo effect" works like this: in a strongly typed language, a reader of the code is tempted to think that a particular method invoked at some program point is defined in the class designated by the type of the target reference used in the call, only to realize later that the method is actually defined in one of its ancestor classes. Even worse, the method could be overridden in one of the descendants of the reference's class making the reader become very confused.

A further difficulty in understanding an object-oriented system arises from the dual nature of class hierarchies. As stated by Snyder [20], "one can view inheritance as a private decision of the designer to reuse code [...] alternatively, one can view inheritance as making a public declaration that objects of the child class obey the semantics of the parent class, so that the child class is merely specializing or refining the parent class".

The nature of a class hierarchy is very important in the context of understanding a legacy system. Knowing if a particular hierarchy is primarily intended for *type reuse* or *code reuse* would help the maintainer in using it correctly and systematically. Furthermore, this would help him/her locate design fragments where the instances of hierarchy classes are treated uniformly, and thus he/she could spot the places in the system where high level business policies are expressed. Last but not least, understanding how a hierarchy is used could help detecting some potential anomalies which reveal design problems in the subject hierarchy.

In the last decade many software analyses related to the understanding of class hierarchies were defined [13, 17, 14, 2, 11]. Almost all these analyses have one major characteristic: they use only information extracted from the hierarchy itself (*e.g.*, "this class only overrides some methods from the base class" or "this method is a specialization of a method inherited from the base class"), analyzing thus the hierarchy in *isolation*. While this is highly necessary, we believe that it is insufficient. Analyzing only the hierarchy will shed some light on the intention of reusing code from ancestor classes, but it will be very hard to assess if the hierarchy is in fact intended to express only code reuse, only subtyping, or both. As Martin states [18] " A model, viewed in isolation, can not be meaningfully validated. The validity of a model can only be expressed in terms of its clients".

In order to exemplify the important role of clients in understanding the aim of a class hierarchy, let's consider a simple example.

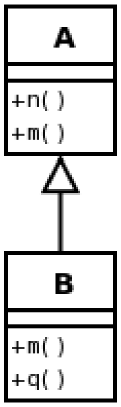

**Figure 1. A simple class hierarchy**

By looking only at the elements of the hierarchy in Figure 1 we may regard *B* as a specialization of *A*. But if we take a look at the clients and see that all of them *refuse* to use the two types defined in the hierarchy in an uniform manner, we can strongly suspect that the inheritance is not used for subtyping, but only for code reuse, in spite of its appearance.

```
void aClient(A ref) {
    if(!(ref instanceof B))
        ref.n();
}
```

**Figure 2. A simple client of the above hierarchy**

Furthermore, there is an additional key aspect related to the specific knowledge about a hierarchy, which is hidden in the client code. For example, let's consider again the hierarchy in Figure 1 and look now closer at one of its clients (Figure 2). Looking only at the invocation of the *n* method and at the type of the *ref* reference we can conclude that *aClient* method could invoke *n* on any instance of the classes *A* or *B*. But at a closer look, we see that *ref* can only refer to an *A* instance in the context of the call and as a result we can conclude that the real intention of the programmer of this client was to invoke the *n* method only for *A* objects and not for *B* instances.

This example makes it clear that in order to analyze the way clients use a class hierarchy we need to go beyond simple information (*e.g.*, a simple call graph based only on method resolution) and employ more advanced techniques like data flow analysis [1], class hierarchy analysis [6], etc.

In this paper we present a new approach that aims to enhance the understanding of class hierarchies, by analyzing how clients use a hierarchy in terms of uniformity (*i.e.*, to which extent calls are made in a polymorphic manner being targeted towards one common superclass rather than towards a direct usage of many subclasses). For this purpose, the paper first defines a suite of metrics which quantify the uniformity of clients' calls with respect to the services provided by a hierarchy. The definition of these metrics is based on a combination of simple structural information and more advanced techniques like data flow analysis. After defining them we will show how these metrics can help both to *characterize a hierarchy* and to detect *anomalies of clients' usage* which often point to design problems.

## 2. Characterizing Base Classes

In a real software system, it is almost impossible to reach a uniform characterization for an entire class hierarchy, since different parts of the same hierarchy might be used in different ways. Consequently, we aim to characterize all the sub-hierarchies and therefore the analysis presented next must be applied to every *base class*.

### 2.1. Two Characterization Dimensions

Discovering the nature of a base class requires a bi-dimensional characterization: one from the perspective of *code reuse* and a second one from the perspective of *interface reuse*. Although we characterize classes, both code and interface reuse are determined by how the *public methods* of the class are defined and used. In other words, each public method contributes to the characterization of a base class. Therefore, next we will focus on a base class with a single public method, while in Section 2.3 and Section 3.2 we will show how the characterization of the base class can be inferred for *all* of its public methods.

*Code Reuse Perspective.* Table 1 captures the rules based on which we infer that a base class was intended or not for code reuse, as this is reflected by one of its public methods. This characterization is based on the way the method is reused in the descendants. We identify the following extreme cases:

| Base Class Public Method | Usage in descendants | | |
|---|---|---|---|
| | Inherited | Specialized | Overridden |
| Inherited for code reuse? | Yes | Yes | No |

**Table 1. Code Reuse characterization based on a base class method's usage in descendants (the vertical perspective)**

| Base Class Public Method | Calls from clients on descendant objects | | |
|---|---|---|---|
| | Uniform | Uniform for some descendants | Non uniform |
| Inherited for interface reuse? | Yes | Partially Uniform | No |

**Table 2. Interface Reuse characterization based on a base class method's invocation by its clients (the horizontal perspective)**

- When all concrete descendants simply inherit the method implementation (it is not a redefined implementation) then we say that the base class is inherited by descendants for *pure code reuse*.
- When all concrete descendants inherit or define a specialized implementation of the method (it is a redefined implementation which invokes the old implementation) then we say that the base class is inherited by descendants for *specialized code reuse*.
- When all concrete descendants have an overridden version of the original implementation (including here the implementation of an abstract operation) then we say that the base class is not inherited for code reuse.

*Interface Reuse Perspective.* In Table 2 we summarize the rules based on which we decide that a base class is intended for *interface reuse*. Again, the characterization takes into account one single public method, and more precisely how the method is used by the *external clients* of the hierarchy *i.e.*, by the methods of a class from outside the hierarchy that call the method on instances of its definition class descendants.

The notion of *uniform usage* is central in this characterization. We say that in a client a call of a method from a base class uniformly uses a set of its concrete descendants when the target reference used in the call may refer to instances of any set members at runtime. When this reference may refer only to instances of one particular descendant the usage is *non uniform*. In this context, we identify the following extreme cases:

- When all the clients always uniformly use all the concrete descendants of a base class, we say that the base class is inherited for *interface reuse*.
- When all the clients always non uniformly use all the concrete descendants of a base class, we say that the base class is not inherited for *interface reuse*.
- When all the clients always uniformly use a subset of the concrete descendants of a base class, we say that the base class is inherited for *partially uniform interface reuse*.

## 2.2. Refined Goal Setting

As a complete bi-dimensional characterization of base classes exceeds the possibilities of a single paper, we decided to set the following boundaries to the approach presented next:

- We focus on *Interface Reuse Perspective*, by aiming to characterize base classes from the perspective of their users.
- Concerning the *Code Reuse Perspective* we limit ourselves to base classes that are intended for *pure code reuse i.e.*, the case where the methods in the base class are not overridden or specialized in their descendants.

Consequently we will try to answer the question: how do the clients of the concrete descendants actually use the interface defined by the base class? Are the concrete descendants predominantly invoked knowing the exact type of the receiver object? If this is true then the base class is not actually intended for interface reuse. If, on the contrary, clients use descendants uniformly then we can say that the base class is intended not only for code reuse, but also for *interface reuse*.

### 2.3. Measuring Intention of Code Reuse

In order to detect the base classes which are inherited for pure code reuse we define the *Pure Code Reuse* (PCR) metric for a public method as the number of concrete descendants of its definition class which inherit the method's implementation in a non-overridden form divided by the total number of concrete descendants of the same class. If the method is abstract then PCR is 0. At the class level, the PCR metric can be aggregated as the average of PCR metric for each of its public methods.

Consequently, we can now say in a more precise manner, that in this paper we are focusing on the characterization of base classes with average PCR values close to 1.0, as this indicates that the base class defines concrete methods and that these methods are mostly inherited as they are by the descendants of the base class.

## 3. Characterizing the Intention of Interface Reuse

In this section we will describe how to characterize the extent to which a base class is intended for *interface reuse* by analyzing how clients use its public methods. As discussed before (see Table 2) the intention of interface reuse is reflected by the extent to which clients use a hierarchy *uniformly i.e.*, to which extent client calls are made in a polymorphic manner being targeted towards the common base class rather than towards many concrete descendant classes.

Thus, in order to characterize the intention of interface reuse of a base class we need to find proper means to make this property quantifiable. Therefore, we define next for this purpose an adequate suite of metrics.

### 3.1. Uniformity Related Concepts

Before introducing the metrics we have to introduce some supplementary concepts on which the metrics definitions rely. These are accompanied by a concrete example (Figures 3 and 4) aimed to illustrate the concepts.

**`ResponsibleFor Set`** . [1] The `ResponsibleFor` set for a method M is the set of classes composed of its class and all the descendants that inherit the original implementation of the method. Abstract classes are excluded from this set.

**Example.** For the concrete methods defined in class *A* (Figure 3) the `ResponsibleFor` sets are:

$$ResponsibleFor(m) = \{C, D\} \qquad (1)$$

$$ResponsibleFor(n) = \{B, C, D\} \qquad (2)$$

**Totally Uniform Call**. A totally uniform call of a method M is a call which may invoke M's implementation through a reference which may refer instances of any class from `ResponsibleFor(M)` set at runtime.

**Totally NonUniform Call**. A totally non uniform call of a method M is a call which may invoke M's implementation through a reference which may refer instances of only one class from the `ResponsibleFor(M)` set at runtime.

**Partially Uniform Call**. A partially uniform call of a method M is a call which is neither total uniform neither total non uniform.

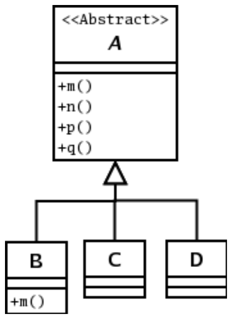

**Figure 3. The hierarchy used to explain the metrics**

```
void method1(int i) {
    A a;
    if(i == 0) a = new B();
    else if(i == 1) a = new C();
    else a = new D();
    a.n(); // a totally uniform call to n
}
void method2(B b) {
    b.p(); // a totally non uniform call to p
           // b can refer only B instances
}
void method3(A a) {
    if(!(a instanceof B)) {
        a.q(); // a partially uniform call to q
           // a can refer only C and D instances
    }
}
```

**Figure 4. The clients illustrating the uniformity related concepts**

1 In the context of this paper where we analyze only base classes having PCR close to 1.0 this set is similar with the Applies-To set from [6]. We did not use this name because generalizing the characterization will require a different definition for this set.

### 3.2. Uniformity Design Metrics

Based on the previously introduced concepts, we will now introduce three uniformity design metrics at the method, and, respectively, at the class level.

**Total Uniformity (TU).** Total uniformity for a concrete method M is defined as the number of *Totally Uniform Calls* of method M, divided by the total number of calls which may invoke M's implementation at runtime. At the class level we define $AVG\_TU$ as the average of the TU metric for all of its public, concrete methods.

**Partial Uniformity (PU).** Partial uniformity for a concrete method M is defined as the number of *Partially Uniform Calls* of method M, divided by the total number of calls which may invoke M's implementation at runtime. At the class level we define $AVG\_PU$ as the average of the PU metric for all of its public, concrete methods.

**Total Non-Uniformity (TNU).** Total non-uniformity for a concrete method M is defined as the number of *Totally Non-Uniform Calls* of method M divided by the total number of calls which may invoke M's implementation at runtime. At the class level we define $AVG\_TNU$ as the average of the TNU metric for all of its public and concrete methods.

### 3.3. Characterizing *Interface Reuse* by using the Uniformity Metrics

In the following we are going to interpret the values of the above defined metrics, from the perspective of characterizing the *Interface Reuse* of base classes.

$AVG\_TU$ ***close to 1.0*** indicates that the instances of the subclasses of the measured base class are almost always used in an uniform way with respect to the base class public methods. This means that clients invoke these methods without being concerned about the concrete type of the invoked object and as a conclusion, their classes do not only reuse the code of the measured class but also reuse its interface.

$AVG\_TNU$ ***close to 1.0*** should indicate that the instances of the subclasses of the measured class are almost always used in an non uniform way with respect to the base class public methods. This means that clients invoke these methods knowing exactly the concrete type of the invoked object and as a conclusion, their classes only reuse the code of the measured class and should not be considered subtypes of the base class.

$AVG\_PU$ ***close to 1.0*** – which logically implies low values for $AVG\_TU$ and $AVG\_TNU$ – indicates that the instances of the subclasses of the measured class are almost always used in an partially uniform way with respect to the base class public methods. This means that clients almost always invoke these methods knowing that the target object is an instance of some subset of these subclasses. As a conclusion, these methods are inherited for code reuse but it might be possible that clusters of these classes actually intend to form a type hierarchy.

## 4. Metrics Computation. Tool Support

Our approach has been implemented using iPLASMA [16]. It is a reengineering environment built on top of the MEMORIA meta-model [16] . Additionally, we have used MEMBRAIN, a prototypical tool developed by us to perform data flow analyses on Java source code. MEMBRAIN has been integrated in the iPLASMA reengineering platform which permits us to combine in an easy way results of data flow and design analyses.

In order to approximate the aforementioned uniformity metrics we have implemented the intra-procedural static class analysis (SCA) [6] using MEMBRAIN. This data flow analysis determines at particular program points the set of classes for an object. In other words, it determines for any reference variable, at a particular program point, the possible set of classes of the instance to which that reference points.

Based on this information computed for all the potential callers of a method *M* and based on the *ResponsibleFor* set of the same method, we can easily compute the uniformity metrics for that method. Computing the uniformity metrics at the base class level is a trivial job once they are computed at the method level.

## 5. Experimental Setup

In the previous sections we have theoretically discussed our approach in order to discover the interface reuse nature of a base class. In this section we present the results we have obtained by applying our methodology to some concrete Java software systems.

### 5.1. The Case Studies

For our case study we have chosen two public domain systems: **Recoder** [2] and **FreeMind** [3]. Table 3 presents an overview of these systems. While the first three metrics (*i.e.*, Number of Classes, Number of Methods, Number of LOC) help us to understand the size of the system, the last two (*i.e.*, Average Number of Derived Classes (ANDC) [14] and Average Hierarchy Height (ANH) [14]) provide an overall characterization of inheritance usage in the system. The ANDC metric is the average number of classes directly derived from a base class (if a class has no derived classes

2 for more details see `http://recoder.sourceforge.net/`
3 for more details see `http://freemind.sourceforge.net/`

| System | Number of Classes | Number of Methods | Number of LOC | Average Number of Derived Classes | Average Hierarchy Height |
|---|---|---|---|---|---|
| Recoder | 490 | 6795 | 42259 | 0.74 | 0.43 |
| FreeMind | 455 | 5228 | 52904 | 0.51 | 0.34 |

**Table 3. Overall characteristics of the analyzed systems**

| System | All base classes | Base classes having PCR $> 0.85$ |
|---|---|---|
| Recoder | 219 | 29 |
| FreeMind | 239 | 19 |

**Table 4. The analyzed base classes**

then it contributes with a value of 0 to ANDC) while the ANH metric is the average of the Height of the Inheritance Tree for all the root classes from the system (a class is a root class if it is not derived from another one; standalone classes have a HIT of 0). The values of these two inheritance-related metrics tell us that the two systems contain some hierarchies which tend to be wide and not very deep.

## 5.2. Investigation Approach

First, we have computed the PCR metric for all base classes from the two analyzed systems and, in conformity with the aspects discussed in Section 2.3, we have kept for the rest of the investigation process only those that had a PCR value close to 1.0. Table 4 presents the total number of base classes from each system and the number of base classes having a PCR metric greater than 0.85.

After this step, for the remaining base classes we have computed the uniformity metrics and we have interpreted their values with respect to the interpretation model from Section 3.3. Based on these metrics values we have chosen a set of three base classes for a manual inspection in order to see if our interpretation model is confirmed by the reality in the code. The metrics values for these base classes are shown in Table 5.

## 5.3. Discussion of Most Significant Findings

**Case 1:** *The `AbstractArrayList` Class (from **Recoder**).* The class `AbstractArrayList` has a high value for the *AVG_TNU* metric. This means that the instances of the subclasses which inherits the public methods of the `AbstractArrayList` without any modifications (not in an overridden form) are almost always called knowing their concrete type. According to our interpretation model this class is inherited only for code reuse.

After manually analyzing this class we found that it has 42 descendants and an height in the inheritance tree of 1. A partial class diagram is shown in Figure 5. All of these descendent classes model different kinds of lists like `ConstructorList`, `ClassTypeList`, etc, and implement their added functionality based on the protected interface of the `AbstractArrayList`. Some of these added operations are `add`, which inserts a particular type of object in the list, and the corresponding access methods `getConstructor` and `getClassType`. It seems that this hierarchy has appeared in the system because the version of the Java language that was used to implement the system didn't have generic types. Based on these observations and based on the fact that any list provides *`trim`*, *`indexOf`*, *`isEmpty`*, *`size`* operations, it is clear that the inheritance relations between the `AbstractArrayList` and its descendants tend to be oriented only to code reuse.

**Case 2:** *The `HookAdapter` Class (from **FreeMind**).* The `HookAdapter` class has a high value for the *AVG_PU* metric. According to our interpretation model, such a value means that this base class is inherited for code reuse but we can expect to find some concrete descendants which are treated uniformly by some potential clients with respect to the methods defined in the base class.

After manually analyzing this class we found that it has 25 descendants and an height in the inheritance tree of 4. A partial class diagram is shown in Figure 6. We have depicted in this base class some methods inherited but not modified by any of the descendants. The rest of the public methods of this class appear to be invoked only from its descendants, which indicates the code reuse intention of this base class.

Analyzing the invocations of the `getName` method we have observed that it is invoked by the clients of the hierarchy only for `PermanentNodeHookAdapter` which explains the high value of the *AVG_PU* metric for the base class. When we analyzed the invocations of the `setController` method we found that some calls seems to be uniformly

| Class Name | PCR | *AVG_TU* | *AVG_PU* | *AVG_TNU* |
|---|---|---|---|---|
| AbstractArrayList | 1 | 0.003 | 0.127 | 0.87 |
| HookAdapter | 0.91 | 0 | 0.9 | 0.1 |
| AbstractHashSet | 0.89 | 0.98 | 0 | 0.02 |

**Table 5. The metrics values for some base classes**

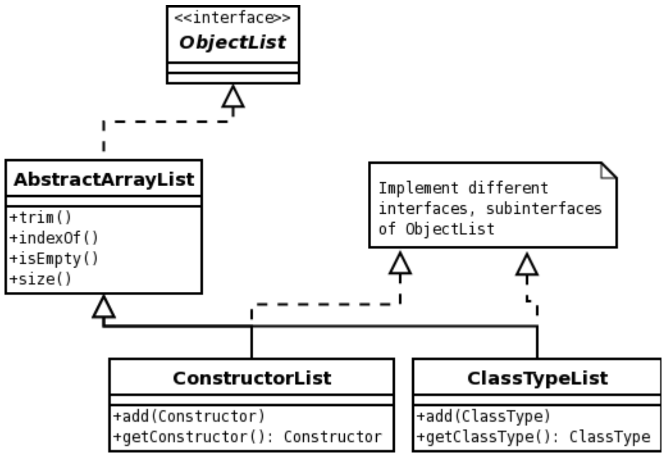

**Figure 5. A partial view of the `AbstractArrayList` hierarchy**

for the descendants of `ModeConctrollerHookAdapter` and others seems to be uniformly for the other part of the hierarchy. There are no calls which may uniformly treat a descendant of `ModeControllerHookAdapter` and one descendant from the other part. This again explains the high value of the *AVG_PU* for subject base class. In conclusion, the `HookAdapter` seems to be inherited only for code reuse, and, as our *AVG_PU* anticipated, it defines a common interface through which only some parts of the hierarchy descendants are treated in an uniform way.

**Case 3:** *The `AbstractHashSet` Class (from **Recoder**).* The `AbstractHashSet` base class has a high value for the *AVG_TU* metric. According to our interpretation model, such a value means that this base class is inherited for code reuse but it is also used to uniformly invoke its concrete descendants.

Manually analyzing this class we have found it has only 3 descendants and a height in the inheritance tree of 1. A partial class diagram of this hierarchy is shown in Figure 7. We have noticed that this hierarchy implements different kinds of sets which only differ in the way the `equals` and `hashCode` operations are computed for the set members (these operations being overridden by each descendant in a specific manner). This explains the smaller value of the PCR metric for this base class with respect to the other based classes we have already discussed and also confirms its code reuse intention. At the same time, analyzing the potential client invocations to its concrete descendants we have found a large number of calls having a target reference of `AbstractHashSet` type. Based on this observation one could say that this hierarchy is also used as a type hierarchy as the *AVG_TU* metric anticipated, since the client code is written in terms of a super-type.

The interesting part with this experiment is that it has shown us that the things can be more complicated. Manually analyzing the clients invocations we have also found that some of these calls are targeted to instance variables that are initialized (outside the client methods) with concrete descendants of the `AbstractHashSet` base class, and never changed! Yet, these calls are incorrectly identified as uniform calls using an intra-procedural SCA which cannot observe these initializations. On the other hand, an approximation of the uniformity metrics based on inter-procedural SCA would have produced a smaller value for the *AVG_TU*, and according to our interpretation model, the `AbstractHashSet` wouldn't have been considered intended for interface reuse. However, such an implementation could be too conservative. In our concrete experiment, a particular kind of set might be substitute with another one without breaking the client code. Thus, despite the restrictive initializations, we can still speak about a type hierarchy. As a conclusion, we should further investigate how to use the object instantiation information in the context of our analysis.

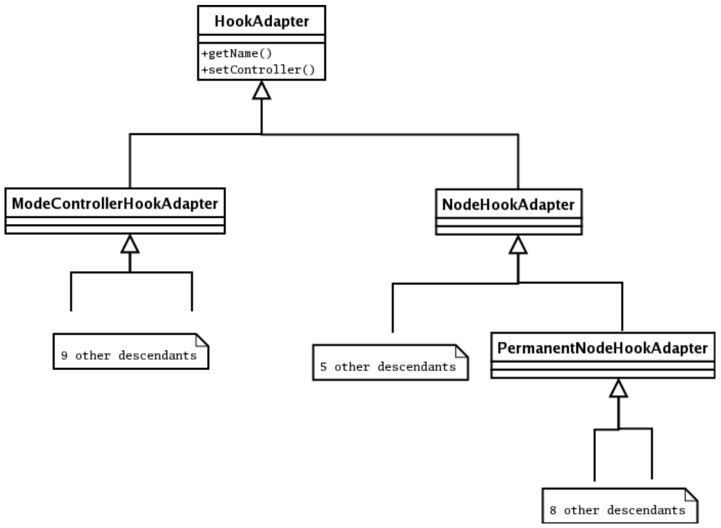

**Figure 6. A partial view of the top of `HookAdapter` hierarchy**

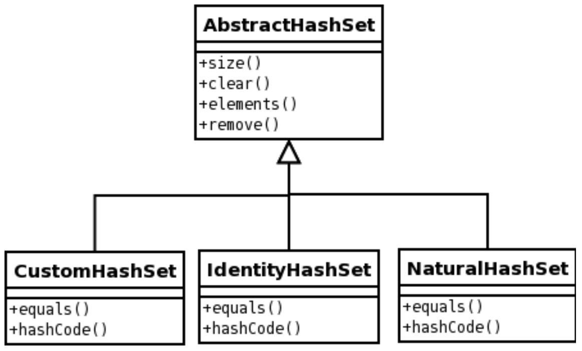

**Figure 7. A partial view of the `AbstractHashSet` hierarchy**

### 5.4. An Interesting Method-Level Abnormality Pattern

As mentioned at the beginning of this paper, analyzing the clients of a base class may also reveal some abnormalities in the design of the class hierarchy below a given base class. While analyzing the aforementioned systems, we identified at the method-level such an *abnormality pattern* which appears to be very interesting. We will briefly discuss it next.

***Interface Method Imposed by Subclass*** This abnormality pattern usually appears in a public method *M* of a base class which has the following characteristics:

1. the ResponsibleFor(M) set has at least 3 members
2. a very low total uniformity (TU $< 0.2$);
3. a null partial uniformity (PU = 0);
4. all the totally non-uniform calls have as receivers instances of one single descendant class from the `ResponsibleFor(M)` set.

This pattern has been even detected twice within a single class hierarchy from the **Recoder** case-study system. The partial class diagram of the hierarchy is shown in Figure 8.

By analyzing the names of the methods and classes we can understand the reason for this abnormality. As **Recoder** is a framework that extracts a full-fledged representation of Java programs, the *LoopStatement* base class models a regular loop statement from Java. But it is well known that the *Do* and the *While* statements have neither `initializers` nor `updates` expressions, as imposed by the common interface (see Figure 8). So, these two methods do not actually characterize these two descendants, but were inserted in the common base class just in order to treat the objects of descendant classes uniformly, for the few situations when this is required (see condition: TU $< 0.2$).

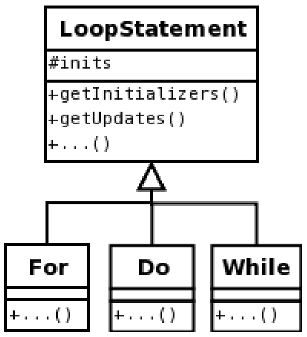

**Figure 8. Example of *Interface Method Imposed by Subclass***

Of course, a *For* statement may also have empty updates and initializers but because these two methods will never be invoked by a client when it knows that it works with *Do* and *While* instances we can say that this is a *refused bequest* design flaw which affects the understandability of the hierarchy [9]. In this case a refactoring option would be to ”Push Down” those methods into the *For* class. In general, in order to take such a decision, a developer or maintainer should also investigate the need for uniformity which appears in some clients.

## 6. Related Work

The dual nature of class hierarchies in object-oriented software systems is intensively discussed in theory and practice [5, 10, 15, 18, 19] especially in the context of forward engineering. The design and enforcement of correct behavioral type hierarchies is an important part of software development especially in the context of designing reusable components *e.g.*, [8].

Class hierarchies are also analyzed in the context of compiler optimizations. Thus, in [6] the authors present a method, named Class Hierarchy Analysis (CHA), which, based on the class hierarchies from a system, helps a compiler to transform a dynamical dispatched method invocation into a simple procedure call. An extension of this analysis is Rapid Type Analysis [3] which refines the results of CHA with information about object instantiation.

In the reverse engineering community, much effort has been spent in the last decade to decompose and analyze the complexity of class hierarchies from multiple viewpoints. As our work is also placed in the context of reverse engineering we will relate next our work to several valuable state-of-the-art contributions.

In order to understand the details of class hierarchies Lanza [13] defines a number visualizations, called polymetric views, which help to reveal whether a hierarchy is built on code reuse by means of extending and overriding methods or on mere addition of functionality. This is useful in order to find the classes that have a big impact on their subclasses, or to understand class implementation in the presence of inheritance. While these visualization techniques can partially help to discover the code reuse nature of a base class they do not take into consideration the actual usage of the hierarchy in clients. Because of this they can not discover the second nature of a base class, namely interface reuse.

In [14] two detection strategies [17] are defined in order to detect *Refused Parent Bequest* bad smell [9] and a further inheritance related problem called *Tradition Breaker* [14]. These design problems usually indicate that the hierarchy is ill designed and so their detection is important in order to improve the hierarchy. However, the detection of *Refused Parent Bequest* is limited because the detection strategy is using information only from the hierarchy itself, more precisely, the usage of the protected interface of a base class in descendants. It does not take into consideration the client perspective which sees the public interface. Thus, it may be that a public method inherited by a descendant from its base class is never invoked by clients on its instances. This is also a sign of *Refused Parent Bequest* which could be detected knowing that a descendant does not inherit a base class for interface reuse.

Another important problem in the context of class hierarchies is the presence of client type checks [7]. Usually, these checks appear because the provider hierarchy does not implement some service, enforcing its clients to implement themselves the service for different types of objects in the hierarchy. Identifying and eliminating client type checks can dramatically reduce the complexity of the hierarchy clients improving the maintainability and understandability of the subject system.

There are many other reverse engineering analyses focused on class hierarchies. Arévalo et al. [2] use concept analysis to automatically discover well known and also unanticipated dependency schemas in class hierarchies which make them hard to extend and maintain, while Gîrba et al. [11] define measurements and rules by which they detect different characteristics of the evolution of class hierarchies.

## 7. Conclusions and Future Work

We have presented in this paper a new way to characterize a class hierarchy of a legacy system from the perspective of its clients. Understanding an object-oriented legacy system is a difficult job especially because of the dual nature of the inheritance relation. To address this issue, we have first introduced a bi-dimensional characterization of class hierarchies: the code reuse dimension and the interface reuse dimension. Then, we have defined a simple metric in order to limit ourself on those base classes which are intended for pure code reuse. Next, we have focused our efforts on characterizing the interface reuse intention of these base classes which implies a closer look to the clients of the hierarchy. We have introduced a suite of uniformity metrics which quantify the tendency of uniform, partial or non uniform usage of the descendant's instances through the base class interface. Based on these metrics we characterize the base class as being intended only for code reuse or for code reuse and interface reuse.

We have applied our characterization methodology on two medium-size case studies and we have found that it might help us in understanding the nature of a base class from a legacy system since the metrics interpretation appears to be consistent with the conclusion reached when we have manually analyzed these systems. We have also identified a pattern in the usage of a base class which might indicate a potential design problem in the hierarchy. Although our case studies have provided promising results we believe that a stronger case study is required in order to evaluate our approach in more detail.

In the near future our research will be focused on the following directions.

- We want to complete our characterization methodology by defining two other code reuse related metrics to characterize the situations when a base class public method is specialized or overridden by some of its class descendants. The extension might also require some modifications in the definitions of the uniformity related concepts and metrics because they will have to be more general. After the completion and implementation of the entire characterization methodology we plan a larger case study.
- Working with a final suite of 6 metrics might become too difficult. As a result we also plan to aggregate the two pairs of 3 metrics into 2 metrics, one for each dimension.
- In order to compute the uniformity metrics more precisely we also plan to implement in MEMBRAIN an inter-procedural SCA algorithm such as that found in [12]. At the same time, we will also investigate how to properly use the object instantiation information in our particular analysis.

## References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, Reading, MA, 1986.

[2] G. Arévalo, S. Ducasse, and O. Nierstrasz. Discovering unanticipated dependency schemas in class hierarchies. In *Proceedings of CSMR '05*. IEEE Computer Society Press, 2005.

[3] D. F. Bacon. *Fast and Effective Optimisation of Statically Typed Object-Oriented Languages*. Ph.D. thesis, University of California at Berkeley, 1997.

[4] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.

[5] G. Booch. *Object Oriented Analysis and Design with Applications*. The Benjamin Cummings Publishing Co. Inc., 2nd edition, 1994.

[6] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of ECOOP '95*. Springer-Verlag, 1995.

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, San Francisco, 2002.

[8] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of ESEC / SIGSOFT FSE ' 01*, 2001.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, Reading, MA, 1999.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.

[11] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of CSMR '05*. IEEE Computer Society Press, 2005.

[12] D. Grove. The impact of interprocedural class analysis on optimisation. In *Proceedings of CASCOM '95*, 1995.

[13] M. Lanza. *Object-Oriented Reverse Engineering—Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[14] M. Lanza, R. Marinescu, and S. Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006. to appear.

[15] B. Liskov. Data Abstraction and Hierarchy. In *Proceedings OOPSLA '87*, page addendum, Dec. 1987.

[16] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel. Iplasma: An integrated platform for quality assessment of object-oriented design. In *Proceedings of ICSM '05, Industrial and Tool Volume*. IEEE Computer Society Press, 2005.

[17] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, "Politehnica" University of Timişoara, 2002.

[18] R. C. Martin, editor. *Agile Software Development. Principles, Patterns and Practices*. Prentice Hall, 2003.

[19] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[20] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 38–45, Nov. 1986.

IEEE COMPUTER SOCIETY