Discovering Comprehension Pitfalls in Class Hierarchies

Petru Florin Mihancea   Radu Marinescu

LOOSE Research Group

"Politehnica" University of Timişoara, Romania

E-mail: {petrum, radum}@cs.upt.ro

Abstract

Despite many advances in program comprehension, polymorphism and inheritance are still the cause of many misunderstandings in object-oriented code. In this paper, we present a suite of such concrete, recurrent patterns where particular ways of using inheritance and polymorphism can easily mislead developers and maintainers during software understanding activities. We define these as comprehension pitfalls. *Furthermore, the paper describes a metric-based approach aimed to automatically detect such situations in code. The experimental results presented in this paper, based on three medium-sized systems, indicate that the identified comprehension pitfalls and the approach used to detect them are a promising support for maintenance.*

Keywords: program understanding, polymorphism, inheritance, metrics

1. Introduction

In order to maintain or to reengineer a software system one has to understand it first (at least partially). The software engineering practice has revealed that about a half of maintenance costs are due to software comprehension activities [12]. Therefore, a significant reduction of software maintenance and reengineering expenses can be obtained by creating powerful techniques to support program understanding.

As observed by Chikofsky and Cross in [6], the cost of understanding software is manifested in the time required to comprehend software, which includes the time lost due to *misunderstanding*. Thus, one way to reduce comprehension costs is to minimize time wasted due to misunderstandings.

Certainly, there are many potential sources of misunderstanding a program. In particular, various understandability issues raised by polymorphism have been recognized long time ago and continue to be emphasized and discussed in the state-of-the-art literature (*e.g.*, [3, 5, 18]). In a strongly-typed language for example, when a maintainer wants to track a dynamically bound method call, she is tempted to assume that the invoked method is defined in the class designated by the type of the target reference. However, she can realize later that the method is actually defined in an ancestor of the reference's class or that the method is overridden in one of the descendants of the reference's class. This "yo-yo" effect [3] is a clear evidence that polymorphism can easily mislead an engineer when trying to understand object-oriented programs. Unfortunately, the "yo-yo" effect is not the single misunderstanding trap set by polymorphism.

The state-of-the-art literature presents many empirical rules and heuristics that drive the usage of polymorphism and inheritance in good object-oriented design [10, 18, 23]. However, in particular design contexts, these rules are not entirely obeyed because a tradeoff had to be made between contradictory forces. Such tradeoffs can lead to hierarchies whose polymorphic manipulation can be easily misunderstood. We define these cases as *comprehension pitfalls*.

These situations are of real importance for maintenance. Understanding the polymorphic usage of a program entity in a legacy system (*e.g.*, a method from a base class) requires an in-depth analysis of its clients (*e.g.*, callers of the method) [18, 21]. Unfortunately, this is time consuming since a detailed manual investigation of the clients' code is required. Therefore, in order to save time, a maintainer is tempted to make implicit assumptions about the way the program entity is used, without taking a closer look at its clients (*e.g.*, all the methods from the base class have an uniform semantics for all the descendants of that base class). Thus, if a comprehension pitfall affects the investigated entity, the maintainer's assumptions might be false (*e.g.*, because of some tradeoff, the investigated method does not have an uniform semantics for all the descendants). As a result of this misunderstanding, he can easily insert bugs into the maintained application (*e.g.*, polymorphically invoking a base class method, that does not define an uniform semantics for all the descendants, can produce an unexpected

program behavior [18]).

Therefore, documenting and detecting comprehension pitfalls is of real importance for maintenance: engineers can avoid making wrong assumptions, can avoid introducing bugs into the maintained system, etc. In other words, misunderstanding costs can be avoided.

In this paper we first describe a simple generic process for defining comprehension pitfalls (Section 2). Next, we present in detail three concrete pitfalls that we have derived by investigating several design heuristics, rules and patterns related to the usage of polymorphism (Section 3). In order to automatically detect these pitfalls in code, we also propose three detection strategies (*i.e.*, metric-based rules) [17] that quantify the symptoms of the defined pitfalls using metrics. These metrics are described in Appendix. The tools used to evaluate our work are briefly presented in Section 4 while the experimental results are discussed in Section 5. The paper concludes presenting some related and future work (Sections 6 and 7).

2. Defining Comprehension Pitfalls

The generic process of defining comprehension pitfalls is presented in Figure 1. In step 1, we identify the main characteristics of a program entity (*e.g.*, a method) in good object-oriented design from the point of view of its polymorphic manipulation. The task is performed by identifying and analyzing design heuristics and patterns that govern the usage of that entity from the polymorphism perspective (*e.g.*, a method should have an uniform semantics for all the descendants of its declaration class).

In step 2, we identify situations in which the subject entity is used in a different manner with respect to its primary way of usage obtained in the previous step. This goal can also be achieved by investigating design heuristics, rules and patterns as many of them present tradeoffs or design contexts when a rule is neglected. Additionally, personal experience may also be applied during this definition step (*e.g.*, some design rules are neglected in commonly known design contexts). The result of this step consists in a list of informal descriptions of one or more comprehension pitfalls.

Next (step 3), a detection technique must be proposed for each identified pitfall. To achieve this task, different approaches may be used (*e.g.*, a graph-based approach). In our particular case, we have used a metric-based approach. That is, we have defined several detection strategies [17] that quantify the informal description of each pitfall. The definition of our metric-based logical rules has followed the process presented in detail in [17]. In short, the informal description of a pitfall is split into a correlated set of symptoms that can be captured by a single metric. Next, a proper metric is selected together with an adequate relational operator
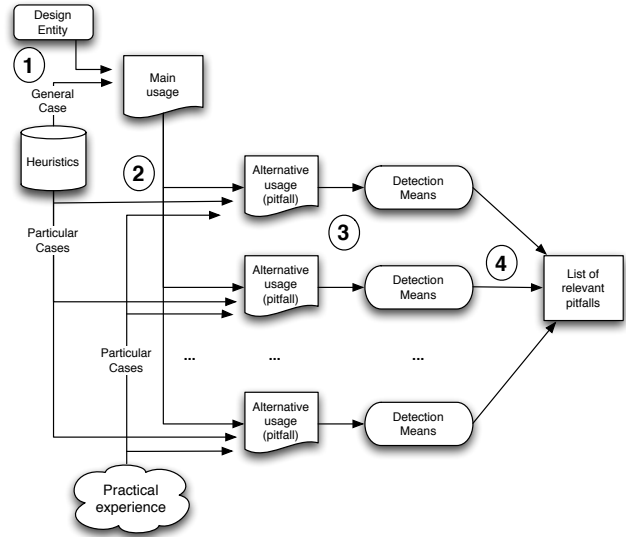

Figure 1. The Process of Defining Pitfalls

in order to quantify each of the previously identified symptoms. In the end, the quantified symptoms are linked together with logical operators following the manner in which the symptoms are correlated in the informal description of the pitfall.

In the last phase (step 4) of our generic process, an evaluation is performed in order to prove the relevance of each pitfall (*e.g.*, how frequently does it appear?) and to analyze the precision of the proposed detection means.

3. Three Concrete Comprehension Pitfalls

Following the process from Section 2, we have defined several comprehension pitfalls. In this section we describe three of them using the following template:

- **Name** - A name to identify the pitfall

- **Entity** - The program entity affected by the pitfall

- **Description** - The rationale of the pitfall

- **Example** - An example

- **Detection** - A detection strategy to detect the pitfall

- **Actions** - A description on how the pitfall detection may improve the maintenance process. Usually, it raises attention about difficult to observe implementation / design details. In some cases, refactoring actions may also be considered.

The results of the fourth step of the definition process and the selection of the thresholds from the proposed detection strategies are discussed in Section 5.

8

3.1. Partial Typing (PT)

Entity. Pure interfaces (*e.g.*, Java interfaces).

Description. Usually, a pure interface is used as a placeholder in the hierarchy for *all* the family of classes that implement it [10, 14, 18]. In this case, we say that the hierarchy is a type hierarchy [14]. However, there are situations when two type hierarchies are linked together by a common interface, although the resulting larger hierarchy is not a type hierarchy. In such cases, we say that this common interface is affected by the *Partial Typing* pitfall. This practice appears in the context of organizing libraries of similar but behaviorally different types [14]. Unfortunately, this situation is counter-intuitive because an interface is usually used as a placeholder for *any* class that implements it. As a result, if one assumes this fact for an interface affected by this pitfall, the risk of introducing bugs into the application is high.

Example. Consider the *Collection* interface from the Java collection system. The *List* and the *Set* interfaces extend the aforementioned interface. However, a *List* object cannot be used in place of a *Set* object because they are behaviorally different (*e.g.*, a list accepts duplicated elements while a set does not). Thus, the *List* and the *Set* sub-hierarchies are type hierarchies but, in spite of the natural expected way of using an interface, the objects they define cannot be usually treated uniformly as *Collection* objects. If one misunderstands this fact (although in this particular example it should not be the case) then there is a high risk of inserting bugs into the application (*e.g.*, substituting and/or permitting the substitution of a *List* for a *Set*).

Detection. The essential characteristic of this pitfall is that the invocations of the methods declared in the affected interface are predominantly directed to objects of several but not of all concrete subclasses from the implied hierarchy. This property of an interface can be captured by a high value of the *Average of Weak Uniformity (AWU)* metric [21] (see Appendix for details) [1]. The resulting detection strategy is presented in Equation A-1: we are looking in a system *S* for all the classes *C* that define pure interfaces and we select only those having *AWU* greater than the HIGH threshold.

$$PT(S) = S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ isPureInterface(C) \land \\ AWU(C) > HIGH \end{array} \right. \qquad \text{(A-1)}$$

---

[1] The uniformity metrics have been extended and their names have been changed in order to be more informative

Actions. When encountering a situation like this, it must be immediately documented. The maintainer can be warned that, usually, the clients of a hierarchy having such a root must be written in terms of some sub-hierarchy of the affected interface. At the same time, they should try to reduce drastically the number of clients defined in terms of the affected interface and they must intensively test such clients to check the objects substitutability.

3.2. Uneven Service Behavior (USB)

Entity. Public methods of a base class

Description. There are cases when a small subset of services are declared in a base class although they do not have uniform semantics for all the descendants of the base class (*i.e.*, they cannot be uniformly invoked because the client expectations are specific to each concrete subclass from the implied hierarchy). The services are declared in the base class simply because they must be provided by all the descendants. In such cases we say that the corresponding methods are affected by the *Uneven Service Behavior* pitfall. In the object-oriented technology, developers reason in terms of objects that provide a cohesive set of services [4]. Thus, a maintainer is tempted to think that, if a set of objects are strongly uniformly manipulated via a common interface then *all* the services of that interface are intended to be invoked in a strongly uniform manner. However, if one does not understand that some services cannot be uniformly called (*i.e.*, without making any assumption regarding the concrete type of the target object) then she could insert bugs into the application.

Example. In the *Prototype* design pattern [10], the cloning method is declared in the root of a hierarchy to force all descendants to implement that service. Additionally, the root also contains other methods to manipulate uniformly the cloneable objects. However, the cloning method is not usually invoked uniformly when the clone initialization depends on the concrete type of the cloned object (*e.g.*, the cloning method must take specific parameters whose meaning depends on the concrete type of the cloned object [10]). Consequently, if the method is invoked without knowing the concrete type of the target object the clone object can be erroneously initialized.

Detection. The detection of this pitfall is based on the following characteristics of the affected method:

1. Many interface services of the method declaration base class are predominantly invoked without knowing the concrete type of the target object (*i.e.*, in a strongly uniform manner). This characteristic of a base class is

emphasized by a high value of the *Average of Strong Uniformity (ASU)* metric (see Appendix for details).

2. By contrast, the problematic method is not intensively invoked in a strongly uniform way. A small value for the *Strong Uniformity (SU)* metric (see Appendix for details) distinguishes this property of a method.

3. The affected method tends to be invoked on instances of all subclasses of its declaration class when knowing the concrete type of the target object. This ensures the fact that the method has a relevant meaning for each descendant. A reduced value of *Type Affinity (TA)* metric (see Appendix for details) can capture this characteristic of a method declared in a base class.

The resulting detection strategy is presented in Equation A-2. We are looking in a system *S* for all the methods *M* whose declaration base classes have an *ASU* value higher than the HIGH threshold. After that, we select only those methods having LOW values for *SU* metric and *REDUCED* values for *TA*.

$$USB(S) = S' \left| \begin{array}{l} S' \subseteq S, \forall M \in S' \\ ASU(Class(M)) > HIGH \wedge \\ SU(M) < LOW \wedge \\ TA(M) < REDUCED \end{array} \right. \qquad \text{(A-2)}$$

Actions. The methods affected by this pitfall should be clearly emphasized in code (*e.g.*, via a documentation comment). In this way maintainers can be warned that the interface methods of the implied base class fall in two categories. The larger one contains methods that can be invoked in highly polymorphic contexts (*i.e.*, where the concrete type of the target object is unknown). The smaller one (the dangerous category) contains methods that can be invoked on instances of any descendant but only when the concrete type of the target object is known.

3.3. Premature Service (PS)

Entity. Public methods of a base class

Description. This pitfall affects methods that are not intended to be uniformly invoked, but which are declared in a base class containing many methods that have this purpose. Additionally, the affected methods do not have a significant meaning for all the descendants of the implied hierarchy. In other words, these methods are declared too high in the hierarchy. That is why we say that they are affected by the *Premature Service* pitfall.

Example. In the *Composite* design pattern [10], the *Component* interface is intended to transparently manipulate *Leaf* and *Composite* objects. Unfortunately, a proper trade-off must be found between transparency and safety when it comes to the declaration of the *addComponent* method. When it is declared in the *Component* interface it represents a premature service pitfall. This is because, usually, it will not be polymorphically invoked (it does not have sense for *Leaf* objects) although the remaining interface methods from the *Component* will. Such a situation is risky from the safety point of view if the maintainer is unaware of the manner in which the *addComponent* method should be called (*e.g.*, it might raise an unexpected exception if called on a leaf object).

Detection. The main characteristics of this pitfall are:

1. Many interface services of the method declaration base class are predominantly invoked without knowing the concrete type of the target object (*i.e.*, in a strongly uniform manner). This characteristic of a base class is emphasized by a high value of the *Average of Strong Uniformity (ASU)* metric (see Appendix for details).

2. By contrast, the suspected method is not intensively invoked in the previously mentioned manner. A small value for the *Strong Uniformity (SU)* (see Appendix for details) metric distinguishes this property of a method.

3. The affected method does not have a relevant meaning for all the descendants of the method declaration base class. Thus, when knowing the concrete type of the target object, the clients will invoke the method only on instances of those subclasses for which the service has a relevant meaning. If the method has a significant number of invocations (counted by the *Number of Calls (NOCALLS)* metric), this aspect can be caught by an increased value of the *Type Affinity (TA)* metric.

The resulting detection strategy is presented in Equation A-3. We are looking in a system *S* for all the methods *M* whose declaration base classes have an *ASU* value higher than the HIGH threshold. After that, we select only those methods having LOW values for *SU* metric. The last two terms of our strategy ensure that each detected method has a *NOCALLS* value higher than FEW and an *INCREASED* value for *TA*.

$$PB(S) = S' \left| \begin{array}{l} S' \subseteq S, \forall M \in S' \\ ASU(Class(M)) > HIGH \wedge \\ SU(M) < LOW \wedge \\ NOCALLS(M) > FEW \wedge \\ TA(M) > INCREASED \end{array} \right. \qquad \text{(A-3)}$$

Actions. When such a pitfall is encountered, the descendants for which the affected method has a significant meaning should be identified. To achieve this task, an in-depth analysis of the hierarchy and/or of its clients is required. Additionally, it would be worth to investigate the possibility of moving such a method down to the relevant subclasses in order to increase safety (*i.e.*, to avoid making unsafe uniform calls that may be targeted at runtime to instances for which the method does not have a relevant meaning).

Note. This pitfall is close to the *Uneven Service Behavior*. However, they have different causes and different actions should be considered when they are encountered.

3.4. Other Pitfalls

During our experience we encountered other potential pitfalls we plan to include in a larger catalog and/or to investigate and to better formalize then in the future: (i) An interface whose methods are almost always invoked in a non uniform manner (*i.e.*, the concrete type of the target object is known) may emphasize that the interface methods are designed to be invoked only in this manner (*i.e.*, non uniformly) (ii) Two overloaded methods of a base class are expected to be invoked in the same manner. However, if one can be uniformly invoked while the other cannot be invoked in this way, another misunderstanding trap may appear (iii) A method of a base class may tend to be uniformly invoked. However, if their is a relevant number of situations when it is non uniformly called, it would be worth to detect these methods and their non uniform calls. In this manner, one can learn the contexts in which the method must be invoked non uniformly (*e.g.*, similar situation may appear while defining new clients of that method).

4. Tool Support

The uniformity metrics and the detection strategies from Section 3 have been implemented in the IPLASMA software analysis environment [16].

In order to approximate the uniformity metrics, we have used an intra-procedural static class analysis (SCA) [7] implemented in MEMBRAIN, a static analysis tool we develop. This static analysis determines at particular program points the *set of classes for an object i.e.*, for any reference variable, at a particular program point, the possible set of classes of the instance to which that reference may refer to at runtime.

Based on this information captured for all potential callers of a method, the uniformity metrics for that method can be easily computed. By averaging the metrics values obtained for all the methods from a base class, we compute the uniformity metrics at class level.

5. Experiment

In Section 3, we have presented three comprehension pitfalls and for each we have defined a metric-based rule to support their automatic detection. In this section we discuss the most significant findings we obtained by applying these rules to three medium-sized Java programs.

5.1. The Analyzed Software

For our evaluation we have selected three public domain Java systems: *Recoder*, *FreeMind* and *Jung*. Table 1 presents several high-level characteristics of these systems. On one hand, they give an impression about the size of these programs (*e.g.*, *Lines of Code*). On the other hand, the *Average Number of Derived Classes (ANDC)* and the *Average Hierarchy Height (AHH)* system-level metrics [13] explain the reason for selecting the case studies. The *ANDC* metric is the average number of classes directly derived from a base class (if a class has no derived classes then it contributes with a value of 0 to *ANDC*) while the *AHH* metric is the average of the *Height of the Inheritance Tree (HIT)* for all the root classes from a system (a class is a root class if it is not derived from another one; stand-alone classes have a *HIT* of 0). According to the statistical thresholds from [13], the values of these metrics reveal that hierarchies are frequent in all three systems and hierarchies tend to be wide and relatively deep. These characteristics of the hierarchies make these systems a very good choice for a significant evaluation of our work.

5.2. Investigation Approach

In order to apply the proposed metric-based rules on the case studies, we had to establish concrete values for their thresholds. The proper threshold selection for a detection strategy (in general, for a metric) is difficult. One approach is the "tuning machine" methodology [20]. The idea is to infer the thresholds from a set of examples manually classified as "affected / unaffected" by a particular pitfall. In this way, it would be also possible to evaluate more precisely a strategy with respect to the developers' intuition regarding the quantified pitfall. Unfortunately, at this time, it is not possible to apply this methodology because of the lack of a sufficiently large set of examples. The construction of such an unbiased tuning set is a long-term activity that requires the recognition of the pitfalls by both the research and the practitioner communities.

However, in this paper we have followed a similar, manual approach. First, we applied the detection strategies from Section 3 on the case studies. In this step, we used light-weight thresholds (*i.e.*, inferred exclusively from metrics interpretation models) in order to ensure the detection of a

System	Lines of Code	Number of Classes	Pure Interfaces	Number of Methods	Base Class Methods	Average Number of Derived Classes	Average Hierarchy Height
Recoder	42 259	490	154	6795	1742	0.74	0.43
FreeMind	52 904	455	141	5228	1523	0.51	0.34
Jung	22 447	391	85	3038	908	0.41	0.34

Table 1. Overall Characteristics of the Analyzed Systems

Strategy	HIGH	LOW	REDUCED	FEW	INCREASED
Partial Typing	0.5	-	-	-	-
Uneven Service Behavior	0.5	0.4	0.3	-	-
Premature Service	0.5	0.4	-	2	0.3

Table 2. The Thresholds

significant number of design entities by our strategies. Next, we analyzed manually all the detected entities and classified them as true-positives or false-positives. Based on this manual investigation we fine-tuned the thresholds in order to maximize the precision of the detection rules. At the same time, we took care to minimize the number of true-positives that might get lost during this fine-tuning process (*i.e.*, minimize false-negatives). The final threshold values are presented in Table 2. Of course, the identified thresholds may be too specific (*i.e.*, particular for our experiment). However, by analyzing in similar experiments many other concrete examples of the described pitfalls (from many other systems), we may be able to establish more general thresholds by applying the "tuning machine" methodology.

5.3. Precision and Frequency

Table 3 presents the number of design entities that have been detected by each detection strategy when applied to each system. Moreover, we split this number of suspects into correctly identified pitfalls *i.e.*, true-positives (TP) respectively false-positives (FP).

Based on this information we can easily compute the precision of our detection strategies (*i.e.*, the number of true-positives divided by the total number of suspects). In the case of *Partial Typing* and *Uneven Service Behavior* strategies, the precision is high (*i.e.*, 90.47% respectively 71.87%). For *Premature Service* we obtained a smaller precision (around 45%).

Turning back to Table 3, we can observe that with one single exception each pitfall appears at least 2 times in each case study. Thus, the defined pitfalls occur in each system. Additionally, we can observe that the *Partial Typing* pitfall appears predominantly in *Recoder* and *Jung* while *Uneven Service Behavior* appears predominantly in *Recoder* and *Freemind*. Thus, the frequency of a pitfall could depend on the type of the analyzed software (*e.g.*, modeled

domain, used technologies). As a result, based on our relatively small case study and on our manual investigation, we conclude that the defined pitfalls are promising for the maintenance process and deserve to be further investigated on larger case studies.

5.4. Discussion of Several Findings

In this section we present three concrete pitfalls we encountered during our manual analysis. On one hand, these examples illustrate the manner in which concrete pitfalls should be studied in detail. On the other hand, they illustrate the problems that the defined pitfalls can cause when falling in their misunderstanding trap.

Case 1: Partial Typing. Using the corresponding detection strategy, we have identified in the *Jung* system the *ArchetypeVertex* interface as being affected by this pitfall. It means that the interface is not the root of a type hierarchy but it links together two or more type sub-hierarchies which are not behaviorally equivalent.

In Figure 2 we present the *ArchetypeVertex* hierarchy. Each rectangle represents an interface (*i.e.*, light-gray nodes), an abstract class (*i.e.*, dark-gray nodes) or a concrete class (*i.e.*, white nodes). The lines connecting the nodes represent *extends* or *implements* relations. We have decided to discuss in detail this example because, from all the interfaces affected by this pitfall, the *ArchetypeVertex* generates the second deepest hierarchy (Heigh in Inheritance Tree is 6). We have not presented here the pitfall that generates the tallest hierarchy because it is just a super-interface of *ArchetypeVertex*, that would needlessly complicate the discussion.

A manual investigation has revealed that the clients that invoke methods in the *ArchetypeVertex* fall essentially into 2 groups (1) clients defined in terms of *Vertex* interface and (2) clients defined in terms of *Hypervertex* interface.

Strategy	Recoder		Jung		Freemind		Overall True-Positives (TP)	Overall False-Positives (FP)	Overall Precision
	TP	FP	TP	FP	TP	FP			
Partial Typing	9	0	8	2	2	0	19	2	90.47%
Uneven Service Behavior	13	8	3	0	7	1	23	9	71.87%
Premature Service	4	3	0	0	2	4	6	7	46.15%

Table 3. Experimental Results
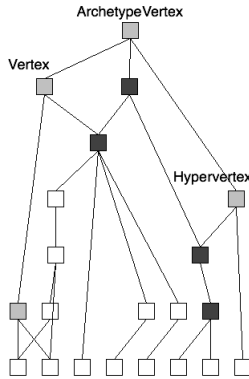
Figure 2. ArchetypeVertex Hierarchy

As shown in Figure 2 these interfaces divide the concrete classes (white nodes) from the hierarchy in two mutually exclusive sets: one containing 3 classes (the right-most white nodes) and another one including the remaining 10 classes. Thus, each group of clients uniformly manipulates instances of classes that are in the same set and almost never instances of classes that are in different sets. As a result, the *Vertex* and the *Hypervertex* sub-hierarchies may be type hierarchies but a *Vertex* object and a *Hypervertex* one cannot be uniformly treated as an *ArchetypeVertex* object. This conclusion can also be supported by taking a look at the names of these interfaces: a *Vertex* models a node of a graph while a *Hypervertex* models a node of a hypergraph. In a hypergraph, a hyperedge can connect more than 2 hypernodes. Thus, we can conclude that a *Hypervertex* behaves somehow different than a *Vertex*, and this is the reason why they are not uniformly treated as *ArchetypeVertex*. As a result, we conclude that *Arhetype-Vertex* is not the root of a type hierarchy but it may link together two type sub-hierarchies. Because of that, substituting a *Hypervertex* for a *Vertex* may produce an unexpected behavior from the system. Thus, it is important to inform a maintainer that, despite the fact that usually an interface is used as placeholder for any class from the implied hierarchy, the *ArchetypeVertex* should not be used in this manner (*e.g.*, she should not define clients in terms of this interface).

Case 2: Uneven Service Behavior. In *Recoder*, the *Operator* base class [2] has many descendants that models different kinds of operators. This base class declares methods that are predominately invoked in a strongly uniform manner by their clients (*ASU* is 0.67). However, several methods declared in this base class are not intensively invoked in this way (*e.g.*, the *setArguments* method has a *SU* value of 0.07). Additionally, these methods are invoked on instances of all subclasses from the *Operator* hierarchy since their *TA* values are also small (*e.g.*, 0.05 for *setArguments*). Thus, the discussed methods (*e.g.*, setArguments) appear to be affected by the *Uneven Service Behavior* pitfall.

We have decided to present in detail this example (*i.e.*, the *setArguments* methods) because (i) the method has one of the largest number of clients (*i.e.*, 27) when comparing with other instances of the *Uneven Service Behavior* pitfall and (ii) the modeled concepts (*i.e.*, Java language concepts) make the understanding of the problem easier.

As we have already mentioned, the *Operator* hierarchy models different kinds of Java operators (*e.g.*, *Instanceof*, *LessThan*, etc.) (see Figure 3). The purpose of the *setArguments* method is to provide by means of its single parameter a list of expressions representing the operands of an *Operator* object. However, the content of this list strongly depends on the concrete operator whose operands are set. For example, an *Instanceof* object requires as operators a reference and a type while a *LessThan* instance requires two numbers as its operators. As can be observed, these two requirements regarding the content of the parameter cannot be simultaneously satisfied. As a result, the *setArguments* method cannot be invoked without knowing the concrete kind of the target object (*i.e.*, is it a *Instanceof* or a *LessThan* object?). Generalizing, because of similar contradictory requirements imposed by different *Operator* objects, the *setArguments* method cannot be usually invoked in a strongly uniform manner. However, it is defined in the *Operator* class because all descendants must be capable to provide this service (*i.e.*, to let a client change the operator arguments). As a result, the *setArguments* method represents a clear example of *Uneven Service Behavior* pitfall.

It is important for a maintainer to be warned that the

---

[2]Recoder defines a representation for Java programs which explains the names of different designed entities from this system
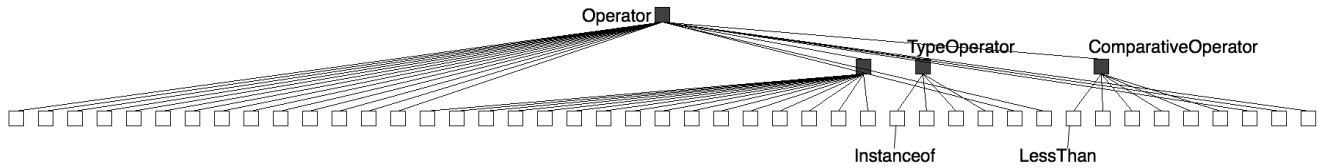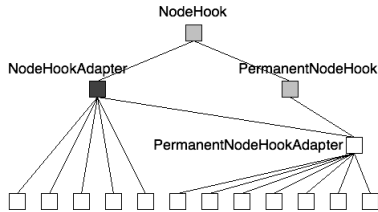
Figure 3. Operator Hierarchy

Figure 4. NodeHook Hierarchy from Freemind

setArguments method must be invoked knowing the concrete type of the target object. Otherwise, she can set incorrect arguments for an operator (*e.g.*, she could set a number in place of a type as the second argument of an *Instanceof* operator). This will produce *Operator* objects with incorrect state and eventually the system will present an unexpected behavior.

Case 3: Premature Service. A concrete example of this pitfall is exemplified by the *setMap* method declared in the *NodeHook* interface from *FreeMind* system. Many interface methods from this base class are predominantly invoked in a strongly uniform manner (its *ASU* metric is high *i.e.*, 0.78). However, one of its methods (*i.e.*, *setMap*) is not predominantly invoked in the same way (it has a small *SU* value *i.e.*, 0.33). Additionally, this method tends to be invoked only on instances of particular subclasses from the implied hierarchy (*TA* metric is 0.36). Thus, it is possible that the *setMap* method characterizes only some subparts of the hierarchy presented in Figure 4.

We observed that the *setMap* method is implemented in the *NodeHookAdapter* abstract class, where it just sets the value of a private field. The class also implements a getter method for the same field. Because the member variable is accessed only by this two methods and because the getter is protected, we expected to find in almost all the descendants invocations to the getter method. Surprisingly, the getter is accessed only from *PermanentNodeHookAdapter* class. Thus, it is possible that the value of the aforementioned field is significant only for classes that implement directly or indirectly the *PermanentNodeHook* interface. Implicitly, the *setMap* method is dedicated only for these kinds of objects.

We also examined the external clients of the *setMap* method. The interesting fact is that we have discovered a single call to this method that may be targeted to instances of any concrete class from the hierarchy. Another one is targeted to *PermanentNodeHook* objects and one is targeted directly to instances of a single class that is not of *PermanentNodeHook* type (although it is not clear where exactly this class uses the set value). In conclusion, we say that the detected method is actually meaningful only for *PermanentNodeHook* part of the hierarchy, a result consistent with the general profile of this pitfall. Thus, it would be worth to investigate the possibility of moving the declaration of the *setMap* method down to *PermanentNodeHook* interface.

6. Related Work

Polymorphism and class hierarchies are keys to increase the extensibility of object-oriented software systems. The type hierarchy nature of class hierarchies is intensively discussed in theory and practice [10, 14, 18, 19, 23]. The design and enforcement of correct behavioral type hierarchies is an important part of software development when designing highly reusable components *e.g.*, [8, 18].

The analysis of class hierarchies is also addressed in the context of maintenance and reengineering. Our work is especially related with design flaws detection. Catalogs of good object-oriented design heuristics can be used to identify design problems related to class hierarchies [23]. In [9] many "bad smells" are described, some of them (*e.g.*, *Refused Bequest*) pointing to design problems in class hierarchies (*e.g.*, inheritance used to achieve only code reuse). Arévalo et al. use concept analysis to automatically discover recurring dependency schemas in class hierarchies [1]. Some of these schemas are also associated with design problems. By contrast with these achievements we are focused on identifying recurring situations which can mislead an engineer during maintenance activities. However, they are not caused by design problems. The situations appear in particular design contexts, as a result of a tradeoff made between contradictory forces or represent accepted deviations from good object oriented design heuristics.

Several other related works may be seen as a way to avoid misunderstanding or facilitate the understanding of

a software. In [22], the authors map program entities and relations between them to concepts and relations recorded in an ontology. In this way, the authors have managed to describe and detect classes of *diffusion* of the domain knowledge in code. Such cases may also mislead an engineer during program understanding activities. In [15], the author managed to enhance the detection of two design problems by eliminating those false-positives that conform to specific design requirements of enterprise applications. Such false-positives can also mislead an engineer since they can be easily misinterpreted as design problems. In [24] a case study of an API redesign is presented. An usability evaluation showed that the new API significantly improved users' productivity emphasizing the importance of properly designed APIs.

7. Conclusions and Future Work

In this paper, we introduced the notion of comprehension pitfall as a design situation in which the polymorphic manipulation of a design entity (*e.g.*, method) can be easily misunderstood. Additionally, we presented a generic process that can be used to define such pitfalls. The applicability of this process has been proved by using it to define three concrete comprehension pitfalls. Moreover, we have introduced three metric-based rules to automatically detect these pitfalls in object-oriented code. Based on our experiment we conclude that the identified comprehension pitfalls and the approach used to detect them are a promising support for maintenance and deserve to be further investigated.

As a future work direction, we plan to perform larger experiments to study the identified comprehension pitfalls. Additionally, we plan an empirical study with human subjects in order to estimate the difficulties in code understanding and modification induced by the described pitfalls. As another future work direction, we plan to investigate the possibility of generalizing our approach beyond pitfalls related to the polymorphic usage of class hierarchies (*e.g.*, pitfalls related to classes) and even for systems which are not based on the object-oriented paradigm. Last but not least, we plan to investigate the possibility of estimating the uniformity metrics using other approaches (*e.g.*, [2, 11]). A more precise estimation of these metrics may have an important impact on our approach (*e.g.*, reducing false-positives, emphasizing false-negatives, etc.).

References

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

[2] D. F. Bacon. *Fast and effective optimization of statically typed object-oriented languages*. PhD thesis, 1997.

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

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

[5] L. Briand, Y. Labiche, and Y. Miao. Towards the Reverse Engineering of UML Sequence Diagrams. In *Proceedings of WCRE*, 2003.

[6] E. Chikofsky and J. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

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

[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, 1999.

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

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

[12] C. S. Hartzman and C. F. Austin. Maintenance productivity: observations based on an experience in a large system environment. In *Proceedings of CASCON*. IBM Press, 1993.

[13] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

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

[15] C. Marinescu. Identification of design roles for the assessment of design quality in enterprise applications. In *Proceedings of ICPC*, 2006.

[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 - Tool Volume*. IEEE Computer Society Press, 2005.

[17] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of ICSM*. IEEE Computer Society Press, 2004.

[18] R. C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

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

[20] P. Mihancea and R. Marinescu. Towards the optimization of automatic detection of design flaws in object-oriented software systems. In *Proceedings of CSMR*. IEEE Computer Society Press, 2005.

[21] P. F. Mihancea. Towards a client driven characterization of class hierarchies. In *Proceedings of ICPC*. IEEE Computer Society Press, 2006.

[22] D. Ratiu and F. Deissenboeck. From reality to programs and (not quite) back again. *In Proceedings of ICPC*, 2007.

[23] A. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.

[24] J. Stylos, B. Graf, D. K. Busse, C. Ziegler, R. Ehret, and J. Karstens. A case study of api redesign for improved usability. In *Proceedings of VL/HCC*, 2008.
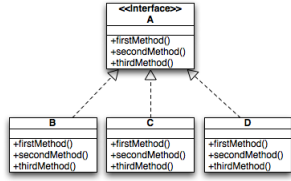
```
void aClient(A x) {
    0: //Here,x may refer B, C or D objects
    1: x.firstMethod();
    2: if(x instanceof B) {
    3:    //Here,x may refer only B objects
    4:    x.secondMethod();
    5: } else {
    6:    //Here,x may refer C or D objects
    7:    x.thirdMethod();
    8: }
}
```

Figure 5. Exemplifying Types of Invocations

APPENDIX

In this appendix, the extended suite of uniformity metrics (initially introduced in [21] in a particular form) is presented together with some related concepts. For the purpose of this paper we generalized them and we introduced a new metric (*i.e.*, *Type Affinity*). We changed also the metrics / concepts initial names in order to be more informative.

Types of Method Invocations

Implementors Set. The *Implementors* set of a method M is the set of classes composed of the method declaration class and all the descendants of that class. Abstract classes (including Java interfaces which are seen as pure abstract classes) are excluded from this set. For all the methods declared in A (Figure 5) the *Implementors* set is {B,C,D}.

Strongly Uniform Call. A strongly uniform call of a method M is a call made through a reference which may refer at runtime to instances of any class from *Implementors(M)*. The call from line 1 represents such an invocation.

Non Uniform Call. A non uniform call of a method M is a call made through a reference which may refer at runtime to instances of a single class from *Implementors(M)* set. In line 4 such an invocation is exemplified.

Weakly Uniform Call. A weakly uniform call of a method M is a call which is neither strongly uniform neither non uniform. An example is shown in line 7.

The Uniformity Metrics

Strong Uniformity (SU). Strong uniformity for a method M is defined as the relative number of *strongly uniform calls* to that method. *SU* close to 1 means that the measured method is predominately invoked in a strongly uniform manner (*i.e.*, almost all its invocations are directed to instances of *all* the classes from *Implementors(M)*).

Weak Uniformity (WU). Weak uniformity for a method M is defined as the relative number of *weakly uniform calls* to that method. *WU* close to 1 means that almost all the invocations to the measured method are directed to instances of several (but not all) classes from *Implementors(M)*.

Non Uniformity (NU). Non uniformity for a method M is defined as the relative number of *non uniform calls* to that method. *NU* close to 1 means that the measured method is predominantly invoked in a non uniform manner (*i.e.*, knowing that the target is an instance of a single class from *Implementors(M)*).

At class level, we define *ASU*, *ANU* and *AWU* metrics as the average of *SU*, *NU* respectively *WU*, for all the interface methods of the measured base class. Thus, the metrics capture the extent to which *all* the interface methods of a base class are invoked in a strongly uniform / non uniform / weakly uniform way. Their interpretation can be easily derived from the one discussed at the method level metrics. Note that for the same measured entity (*i.e.*, base class or method) the sum of these metrics is always 1.

Type Affinity (TA). To simplify our discussion we first introduce an auxiliary metric. The *Hit Rate (HR)* metric is defined for a method M with respect to a class from *Implementors(M)*. Its value represents the relative number of weakly or non uniform calls to M that may be directed to an instance of that class. *Type Affinity (TA)* for a method M is computed like this: (i) we compute the *HR* values for M with respect to each class from *Implementors(M)* (ii) we compute the absolute deviation from the mean[3] for each *HR* value obtained in the first step (iii) the maximum value obtained in the previous step represents the *TA* metric for M. The maximum value is used because it indicates that weak or non uniform invocations of M are mainly directed to some particular classes from *Implementors(M)*. Thus, a high value of the *TA* metric means that only some classes are preferred as targets in weak or non uniform invocations of M.

---

[3]en.wikipedia.org/wiki/Average_deviation