Measuring Inheritance Patterns in Object Oriented Systems: the Dynamic Inheritance Ratio Metric

Niculescu M.
Haute ecole de gestion
Univ. of Applied Sciences of Western Switzerland
7 route de Drize,
CH 1227 Geneva, Switzerland
+41 22 388 17 00
mihnea_niculescu@hotmail.fr

Dugerdil Ph.
Haute ecole de gestion
Univ. of Applied Sciences of Western Switzerland
7 route de Drize,
CH 1227 Geneva, Switzerland
+41 22 388 17 00
philippe.dugerdil@hesge.ch

Canedo Blanco M.
Haute ecole de gestion
Univ. of Applied Sciences of Western Switzerland
7 route de Drize,
CH 1227 Geneva, Switzerland
+41 22 388 17 00
michaelcanedob@gmail.com

ABSTRACT

Among the code structuration mechanisms in object oriented systems, class hierarchies based on the generalization relationship play a prominent role. Indeed it is used to represent and code hierarchies of abstractions supposed to help with code understanding, maintenance and extension. But it is common to see class hierarchies and the associated inheritance mechanism be diverted from this noble role to become a mere code sharing mechanism. In this case, rather than helping, the inheritance mechanism confuses the understanding of the code. Hence, we have developed a metric to analyze the inheritance mechanism at work in a running system, what we have called the inheritance pattern. Although the metrics measuring inheritance are numerous, our approach is original since it observes the actual inheritance in the running code at the class level as well as among the packages (i.e. among the classes through package). In some sense, this metric measures how well the inheritance mechanism has been leveraged in the software. But interpreting raw numbers can be hard. Then we developed a visual and hierarchical representation of the metric values at the scale of a whole system. This helps to assess the quality of the code from the point of view of code abstraction.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *Product metrics*

General Terms

Measurement, Experimentation, Theory.

Keywords

Abstraction hierarchies, inheritance, metrics, system architecture

1. INTRODUCTION

In software engineering, a lot of metrics have been created for the last decades. But, in practice, we observe that only few of them are really used. Often the latter are old metrics that are more or less well accepted by the software engineering community, but barely used. We think three reasons could explain the lack of practical use of some of the newest metrics. First, these metrics are often designed with some theoretical concepts in minds but therefore lack a practical purpose [12]. A case in point is illustrated by the SEMAT initiative which acknowledges the lack of credible experimental evaluation and validation of most of the academic proposals [9]. Second, the metrics are often too broad in scope, characterizing the entire system, or too narrow, characterizing only the smallest structures i.e. the classes ignoring larger structures such as packages and sub-packages. Indeed, global metrics are often difficult to interpret because they can show similar values for very different situations. On the other hand, metrics that are too narrow in scope generate a large number of values whose global interpretation is difficult. Finally, metrics are generally computed trough static program analysis, i.e. analysis of the source code of a system without actually executing it, as opposed to dynamic analysis which is performed on the result of executing the programs. Static approaches have difficulties measuring behavior that arises only at runtime, such as dynamic binding and polymorphism. But this is not a problem for dynamic analysis.

In summary, most of the popular inheritance metrics suffer from the problems explained above. They are often too broad in scope and computed statically. In this paper, we propose a new metric that overcomes these problems. Although inheritance is a feature of classes, we overcome the “narrow scope problem” by showing how the metric can be computed hierarchically i.e. at package level whatever the level in the containment hierarchy. Of course, this technique generates a much larger number of values than a narrow scoped metric, which could be harder to interpret. To overcome the problem, we developed a visual technique to show the metric values on the full system, using colors to help interpreting the values on a global scale. The key contributions of our paper are: the definition of a dynamic metric for inheritance, the inclusion of the packages in the metric and the design of a visual representation to ease the interpretation of the metric values.

In this paper, we present in section 2 the notion of the inheritance pattern from which we could, in section 3, explain the problem that might arise in systems. Next, to characterize the “quality” of the inheritance pattern we propose in section 4 a new metric that is both dynamic (i.e. based on the actual execution of the system) and hierarchical (characterizing not only the classes but also the containing packages). In section 5 we explain how the metric is computed and in section 6 the way the metric is visually represented for an entire system. Section 7 presents the application of the metric to a medium size open source system: Argo UML. Section 8 presents the related work and section 9 concludes the paper.

ISEC '15, February 18 - 20, 2015, Bangalore, India
Copyright 2015 ACM 978-1-4503-3432-7/15/02…$15.00
http://dx.doi.org/10.1145/2723742.2723755.

2. INHERITANCE PATTERNS

Since we are dealing with inheritance of behavior, our work applies to object oriented programs. In order for our approach to stay general, we use the term “substructure” to mean the syntactic grouping of programming elements above the method/procedure level which obeys to a composition relationship (which means that one substructure can be composed of other substructures and so on). Then, our work is applicable to whatever programing paradigm that supports inheritance of behavior.

An inheritance graph is a graph whose nodes are the substructures and whose edges are the relations enabling inheritance. Hence, parent substructures define behavior that is inherited by child substructures. The behavior of a child extends (add new) and/or overrides (redefines existing) the behavior inherited from its parents. The *inheritance path* from a given class is the sequence of inheritance relation to travel to retrieve some inherited behavior. The *length* of the path is the number of relation traveled. It is important to note that our general definition of a substructure which works at different granularity levels, allows us to observe the inheritance of behavior beyond the class level. Indeed, packages do not actually define behavior but we could nonetheless be interested to know that a package contains classes whose behavior is inherited by the classes located in another package. Then, by analogy with the classes, we say that some package “inherits” behavior from another package if some class in the former inherits behavior from a class of the latter, whatever the nesting level of the class in each package’s containment hierarchy.

With these definitions, the (behavior) *inheritance pattern* is the description of the way some specific configuration of substructures share behavior through inheritance. The goal of our work is to propose a method and a metric to evaluate the quality of the design of the inheritance patterns and thus helping to spot patterns that can potentially be improved. This will be especially useful to check the quality of some maintenance. For example one could compare the inheritance patterns before and after some maintenance to check if the architecture has been damaged from the point of view of inheritance. Indeed, it is well known that maintenances could deteriorate the quality of the architecture of a system. However this phenomenon has traditionally been observed on the static point of view i.e. from the responsibilities of the components. From a program understanding point of view it is well know that inappropriate inheritance may lead to a source code that is harder to understand [4][14]. Then, our metric can play a important role in assessing the change in the inheritance pattern hence the understandability of the resulting code.

For the sake of readability of the examples, in the UML class diagrams we directly display the body of the methods in curly brackets. Figure 1 shows an example of an inheritance pattern highlighted in the blue rectangle. ClassB (child) extends ClassA (parent). With our extended definition of inheritance, PackageB “inherits” behavior from PackageA because it contains ClassB which inherits from ClassA located in PackageA. ClassB declares a new method m4 and redefines (overrides) the method m2 which is already declared in ClassA. When the method m4 is executed on an instance of ClassB, the inherited method m1 is executed because it is called by m4. When the inherited method m3 is executed on an instance of the ClassB, the overridden method m2 is also executed. This configuration of declared, inherited and overridden methods represent the inheritance pattern of the structure under study. Our metric measures the ratio of the number of inherited methods among all the methods executed in an instance of a class when running some scenario of business value. This ratio will also be computed for the packages. In the example of Figure 1, PackageC is not included in the inheritance pattern since its class does not inherit any behavior from ClassA or ClassB.

3. INHERITANCE PATTERNS PROBLEMS

The inheritance patterns may sometimes show problems due to some flaw in the initial design or during maintenance. Because of the quantity of maintenance a typical system undergo during its lifetime, it is very likely to see changes in its inheritance patterns over time, often for worse.

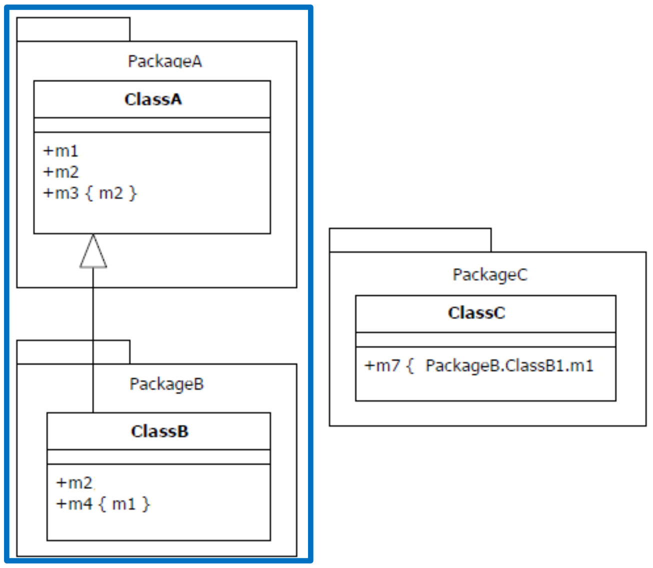

Figure 1. Inheritance pattern

There are two ways in which problems can be spotted: by analyzing the design with respect to some known problem patterns or by comparing the architecture before and after maintenance. In the first case, one could for example detect substructures that involve too little or too much inheritance (with respect to some agreed threshold values) and decide to further investigate them. In the second case one can spot changes and check if these are improving or spoiling the inheritance pattern.

There are various maintenance scenarios in which the inheritance patterns are deteriorating. We describe thereafter some of the problematic scenarios we may encounter. For the sake of simplicity, the inheritance patterns shown in the examples are at the lowest level of granularity, the class-level, but the same patterns exist at higher level of granularity such as between packages. The examples show the problem with one or two methods, but in a real maintenance the updates could be on a much larger scale involving a lot of methods. Moreover the length of the inheritance path is limited to 1 to reduce the size of the figures. While the problems illustrated below may sound trivial when the inheritance path length is 1, they are much more frequent when the inheritance patterns involve paths longer than 1. In the first maintenance situation (Figure 2), a new method m3 must be added to the Child Class to implement some feature. To implement this method, the maintenance engineer needs to call a method whose functionality is equivalent to that of m1 but, ignoring that m1 exists in the Parent Class, he writes a new method m11 in in the Child Class. The problem here is that the functionality (semantics) of m1 is duplicated.

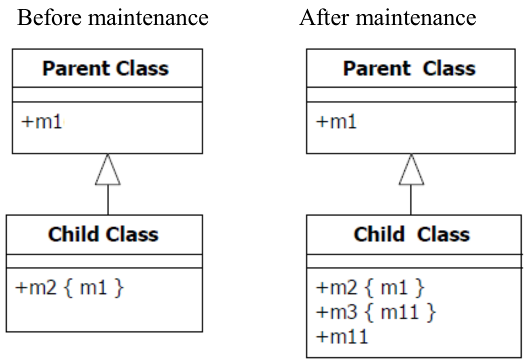

Figure 2. First maintenance scenario

Runtime diagnostic: after maintenance the number of executed method in a scenario involving the feature is greater than before maintenance, while the number of executed methods that are inherited is the same.

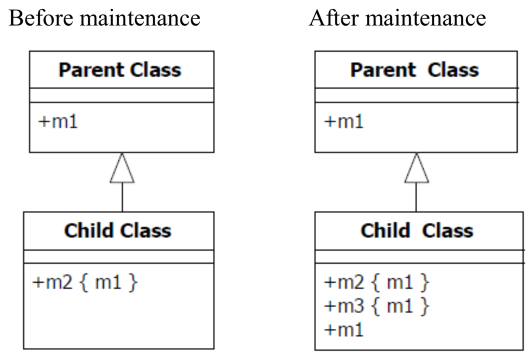

Figure 3. Second maintenance scenario

The second situation (Figure 3) is similar to the first, except that the maintenance engineer uses the same name m1 for the called method implemented in the Child Class, ignoring the method already implemented in the Parent Class. In addition to the code duplication problem, a worse problem occurs: the method m2 was designed to work with m1 that is inherited from the Parent Class. Now it calls the new implementation of m1 in the Child Class (which overrides m1 of Parent Class). Since these two methods are not necessarily equivalent, a bug is likely to result.

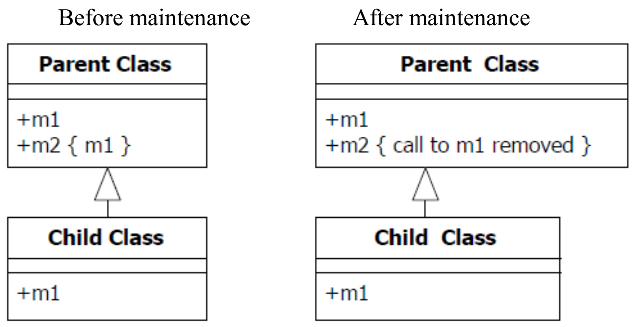

Figure 4. Third maintenance scenario

Runtime diagnostic: after maintenance the number of executed method in the scenario is greater than before maintenance, while the number of executed inherited methods is lower.

In the third situation (figure 4) m2 in the Parent Class call m1 which is specialized in the Child Class. Suppose that the maintenance engineer modifies m2 in order not to call m1 anymore for some reason (for example for code optimization). After the maintenance, when m2 is executed on an instance of the Child Class, it does not call m1 in the Child Class anymore. Since the behavior corresponding to m1 was specialized in the Child Class this specialized behavior is not invoked anymore resulting to a likely bug.

Runtime diagnostic: after maintenance, when the m2 is executed on an instance of Child Class, the number of executed inherited methods is the same while the total number of executed methods is less than before maintenance.

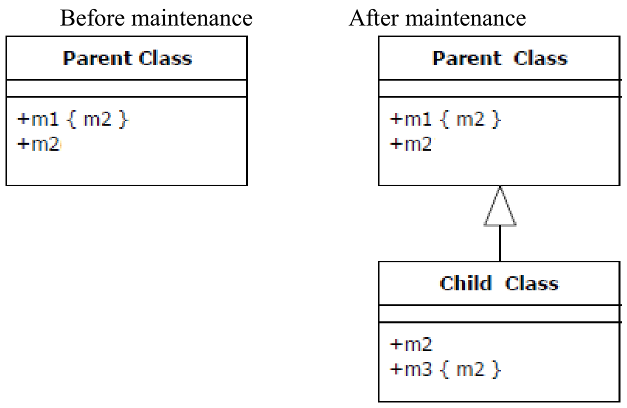

Figure 5. Fourth maintenance scenario

Fourth situation (figure 5): during maintenance, a new class Child Class is created with a new method m3. Suppose that m3 needs to invoke the functionality of m2 in a specialized form. The maintenance engineer then creates a new method m2 in the Child Class while ignoring the fact that m1 uses m2. After maintenance, when executing m1 on an instance of Child Class, m1 will call m2 of Child Class rather than m2 in Parent Class as it was initially designed to work.

Runtime diagnostic: after maintenance when m1 is executed on an instance of Child Class, the number of methods executed in Parent Class is lower than before maintenance, while the functionality of m1 was supposed to be unaffected by the maintenance. It is worth mentioning that the last two situations described above represent instances of a well-known problems referred in literature as the Fragile Base Class Problem [13].

4. DYNAMIC INHERITANCE RATIO

In order to observe and measure the behavior involved in the inheritance patterns we propose a dynamically computed metric, which we called the DIR (Dynamic Inheritance Ratio). This metric, as explained earlier, is computed dynamically when the system is executed following some business-relevant scenarios (use-case instances). Moreover, our inheritance metric can be computed not only at class level but also at larger granularity level (packages). Since we are interested by the patterns of inheritance, we wish to observe where the behavior of the substructures is implemented not how many times some behavior is executed. This is why we count the number of *distinct* methods executed. Hence, we define the following two functions where S denotes the set of substructures in a program:

fimpl S → Integer : number of distinct executed methods implemented in $s \in S$.

finh S → Integer : number of distinct executed methods inherited by some $s \in S$ and directly called on an instance of s or called by one of the executed methods implemented in s.

So fimpl(s) and finh(s) do not overlap. The first function counts the executed methods *implemented* in s and the second counts the executed methods *not implemented* in s. It follows from these definitions that if a class does not implement any executed method and if none of its instances execute any inherited method, the DIR metric is undefined (0/0). For example, in the right part of figure 5, if a scenario would lead to the call of method m3 on an instance of *Child Class*, then *Parent Class* would have an undefined DIR metric value. In the case of the packages these functions are interpreted the following way. *fimpl* : the method must be implemented in some class in the package s, whatever the depth of the class in the containment hierarchy. *finh* : the method must be inherited by any class c in the package s, whatever the depth of c in the containment hierarchy starting from s, from a class defined in another package outside s. The instance for which the methods are executed is that of c. With these functions, the Dynamic Inheritance Ratio is, for any $s \in S$ [2]:

$$\boxed{\mathrm{DIR}(s) = \frac{\mathrm{finh}(s)}{\mathrm{finh}(s) + \mathrm{fimpl}(s)}}$$

Specifically, DIR represents the ratio of the behavior that is inherited versus the behavior that is inherited by and defined in the substructure when running some business related scenario (use case instance).

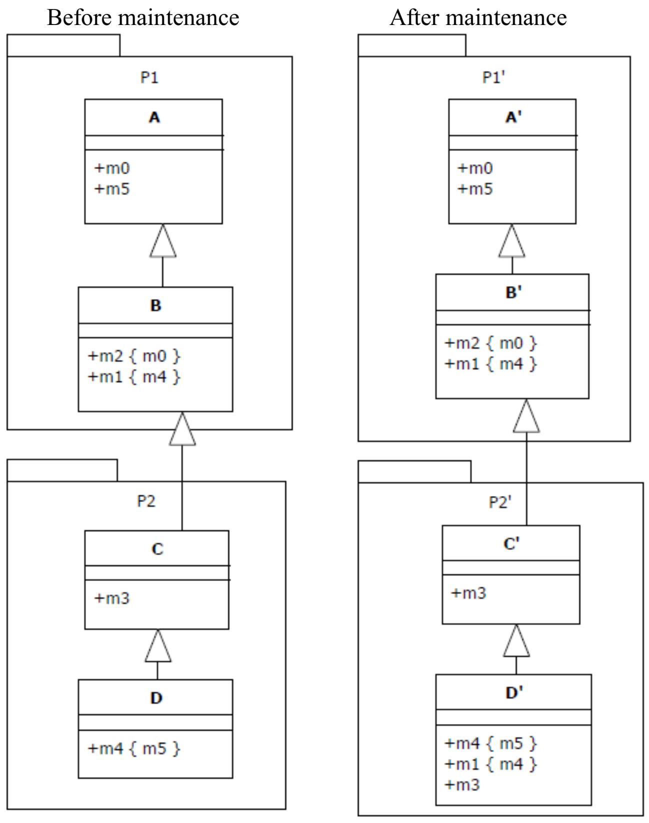

Figure 6. Computing the DIR metric

Figure 6 presents examples where the metric is calculated before and after some maintenance. The maintenance consists in the addition of two methods **m1** and **m3** to the class D which override the inherited methods with the same name. The scenario for which the metric values are calculated is supposed to lead to the call of **m1**, **m2** and **m3** on an instance of D. When the ratio is 0/0 we define the metric's value as N/A (not applicable or undefined). This ratio must nonetheless be recorded since it may help to highlight the changes in the inheritance pattern before and after maintenance.

Metric value before maintenance

DIR(A) = $|\varnothing| / |\varnothing \cup \{m0, m5\}| = 0/2 = 0$

DIR(B) = $|\{m0\}| / |\{m0\} \cup \{m1,m2\}| = 1/3$

DIR(C) = $|\varnothing| / |\varnothing \cup \{m3\} = 0/1 = 0$

DIR(D) = $|\{m0,m1,m2,m3,m5\}| /$
$|\{m0,m1,m2,m3,m5\} \cup \{m4\}| = 5/6$

DIR(P1) = $|\varnothing| / |\varnothing \cup \{m0,m1,m2,m5\}| = 0/4 = 0$

DIR(P2) = $|\{m0,m1,m2, m5\}| /$
$|\{m0,m1,m2,m5\} \cup \{m3, m4\}| = 4/6$

Metric value after maintenance

DIR(A') = $|\varnothing| / |\varnothing \cup \{m0, m5\}| = 0/2 = 0$

DIR(B') = $|\{m0\}| / |\{m0\} \cup \{m2\}| = 1/2$

DIR(C') = $|\varnothing| / |\varnothing \cup \{m3\} = 0/0 =$ N/A

DIR(D') = $|\{m0,m2,m5\}| /$
$|\{m0,m2,m5\} \cup \{m1,m3,m4\}| = 3/6$

DIR(P1') = $|\varnothing| / |\varnothing \cup \{m0,m2,m5\}| = 0/3 = 0$

DIR(P2') = $|\{m0,m2, m5\}| /$
$|\{m0,m2,m5\} \cup \{m1,m3, m4\}| = 3/6$

For example, fimpl(A) = 2 because this class implements 2 methods (m0 and m5) that are executed in an instance of its subclass D. These two methods are included in the set of executed methods inherited by D and counted by finh(D).

5. COMPUTING THE DIR METRIC

To perform any dynamic analysis of a system one must record the sequence of methods that have been executed for a specific scenario, what we call the *execution trace*. Along with each method we record: the class of the instance that execute the method and the class in which the method is declared. There are four basic techniques to generate an execution trace [8]:

1. Instrument the code i.e. introduce extra lines in the source code of the methods in a program to write some information in a file when the method is executed.
2. Instrument the execution environment i.e. change this environment to generate the trace while it executes the original code. The Eclipse TPTP project is an example of such an approach.
3. Run the program in a debugger and insert breakpoints in locations of interest.
4. Use a profiler provided by the development environment.

Since execution traces can be as long as several million calls, the option 3 and 4 are not scalable to such a volume of data. The second option is viable for programming languages in which there are some facilities to instrument the environment. Since we want to be able to apply our method to whatever programming language, we opted for the first technique above. But even in this case, there is an alternative to be chosen: either to build an instrumentor that will analyze the source code and insert the required extra lines or use an Aspect Oriented Programming (AOP) approach to inject the extra lines in the original code. Indeed the latter is very convenient to instrument Java source code. But is it limited to programming languages for which an AOP environment exists. We finally decided to implement our own instrumentor which, in the case of Java, is based on the java parser of Eclipse. The format of the events (calls) generated by the instrumented code is, roughly:

<declaration class><execution class><method signature>

Where "execution class" means the class whose instances execute the method. The generated events are stored in a database from which it is easy to compute the DIR metric.

6. DYNAMIC INHERITANCE RATIO MAP

As explained in the sections above, the metric presented in this work is computed at different granularity (containment) levels of substructures. But the raw numbers are nonetheless difficult to interpret on a global scale. For example we cannot see immediately where the problems with the inheritance patterns are, or what patterns have been changed following some maintenance. To overcome the problem, we proposed to represent the metric values on a hierarchical map showing the containment hierarchy: the DIR Map. This map uses the Treemap visualization tool from Microsoft which can nicely represent the hierarchy of substructures by nested boxes. Each box represents a substructure (package, subpackage, class). Only substructures with defined DIR metric values are represented. So substructure with undefined DIR value (N/A) are not displayed at all. Moreover, if some use-case involves only a subset of all the classes of a system, then only this subset will be displayed.

The label of each box represents the short name of the substructure followed by the metric value and the ratio (in brackets). The color of the box is set according to the metric value as shown in figure 7.

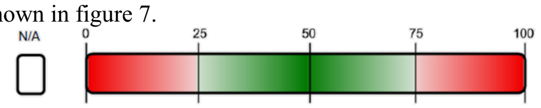

Figure 7. Color scale in the DIR map

If a substructure is colored red its inheritance pattern is potentially problematic. The greener the color the better the inheritance pattern of the substructure. If a substructure is not involved in a given scenario, it is colored white. This will allow us to spot the change in the color of the substructures among different scenarios. The figure 8 presents an example of a DIR map. Navigating the different levels of granularity is easy: when clicking on a box, the Treemapper will zoom the map on the selected substructure.

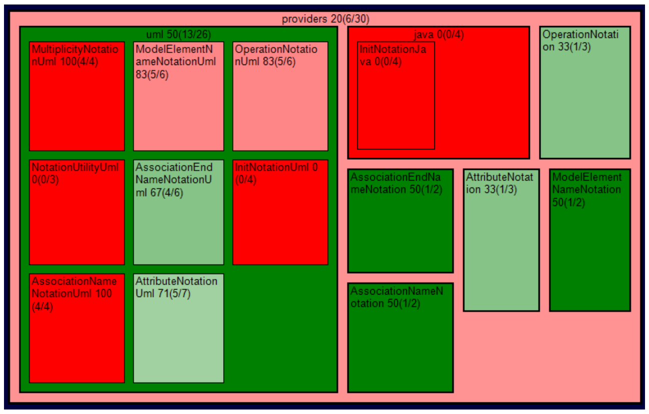
providers 20(6/30)
uml 50(13/26)
MultiplicityNotationUml 100(4/4)
ModelElementNameNotationUml 83(5/6)
OperationNotationUml 83(5/6)
NotationUtilityUml 0(0/3)
AssociationEndNameNotationUml 67(4/6)
InitNotationUml 0 (0/4)
AssociationNameNotationUml 100 (4/4)
AttributeNotationUml 71(5/7)
java 0(0/4)
InitNotationJava 0(0/4)
OperationNotation 33(1/3)
AssociationEndNameNotation 50(1/2)
AttributeNotation 33(1/3)
ModelElementNameNotation 50(1/2)
AssociationNameNotation 50(1/2)

Figure 8. DIR map

Based on the OO principles, a system should be structured as hierarchies of classes representing abstractions with subclasses at each level specializing the abstractions above them. Following this structuring principle, the classes in an inheritance tree will declare methods as well as inherit method from ancestor classes. Hence we can identify two situations where this OO principle would clearly not be leveraged:

- A class with no inherited methods. If all the behavior of the class is declared in the class, then the abstraction mechanism is not used and the very abstraction mechanism is useless.
- A class with no declared methods. If no behavior is defined in the class but everything is inherited then the class is useless (since, in the OO paradigm, the usefulness of a class is relative to its behavior not its structure).

These represent the upper and lower limits in the metric values and the corresponding substructure will be painted red. So a "good" metric value should be situated in between these two limit values. Broadly speaking, we could say that a well-designed class should implement a "good" balance between the behavior it inherits and the behavior it implements. But the proper setting of the threshold values between green and red depends on the evaluation of the acceptable inheritance patterns by a quality engineer. We are well aware that this setting is somewhat arbitrary since there is no reference value we could rest on. Hence the tool must be calibrated according to the subjective values defined by the quality engineer.

The DIR map can be used in two situations:

1. To assess the quality of a system based on the inheritance patterns.
2. To compare the quality of a system before and after maintenance on the basis of the inheritance patterns.

In the first situation above, the map is used to assess the "absolute" quality of some system and the proper setting of the threshold is important. In the second situation, we are interested in the change of the values and the colors represent a simple technique to identify these changes. In this case the color themselves are less critical. This is the situation presented in the figure 9 below which illustrates the situation of figure 6 (same metric values).

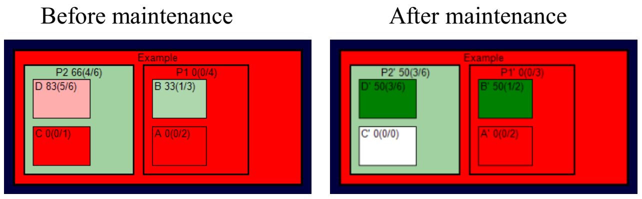

Figure 9. DIR map of the example of figure 6

It is easy to see the difference between the two situations which, on the basis of the colors, can be interpreted the following way:

- Class C does not have any role in the scenario after maintenance which should trigger some further investigations by the maintenance engineer.
- Class D implements a much better balance between the inherited and implemented behavior.
- Class B significantly improved the balance between the inherited and implemented behavior.
- Package P2 slightly improved the balance between the inherited and implemented behavior.

All other substructures did not change their inheritance patterns. Such a technique let us identify changes by spotting different color patterns and then allows us to further investigate the code of the "red" substructures to check if their design could be improved.

7. CASE STUDY: ARGO UML

7.1 Single version display

We analyzed, with the DIR Map approach, the structure of ArgoUMLv035, a medium sized open source java application to draw UML diagrams (approx. 2000 classes organized in 150 packages). In the example presented in figure 10, we ran a small scenario in which we used only a subset of the functionality of the application. Specifically, we created a simple UML class diagram and saved it. This shows one of the key features of our metric: it can be computed at different levels of containment. We can immediately observe that a majority of the substructures are painted red which, in this case, represent low metric values, with the notable exception of the *uml.diagram.static_substructure* which holds most of the functionality executed in the scenario. The majority of the packages are red because only a few of the system's functionality have been used. If the use of the application would be limited to the scenario we ran, then the architecture of the system would be badly designed because the implementation of the behavior would be needlessly scattered among several packages and classes. In order to truly analyze an entire system, we should run all the scenarios representing the real uses of the system. Fig 11 shows the DIR map of the package that is the most specific to the scenario. We can see that about half of the substructures are painted green representing a good balance between the inherited and implemented code.

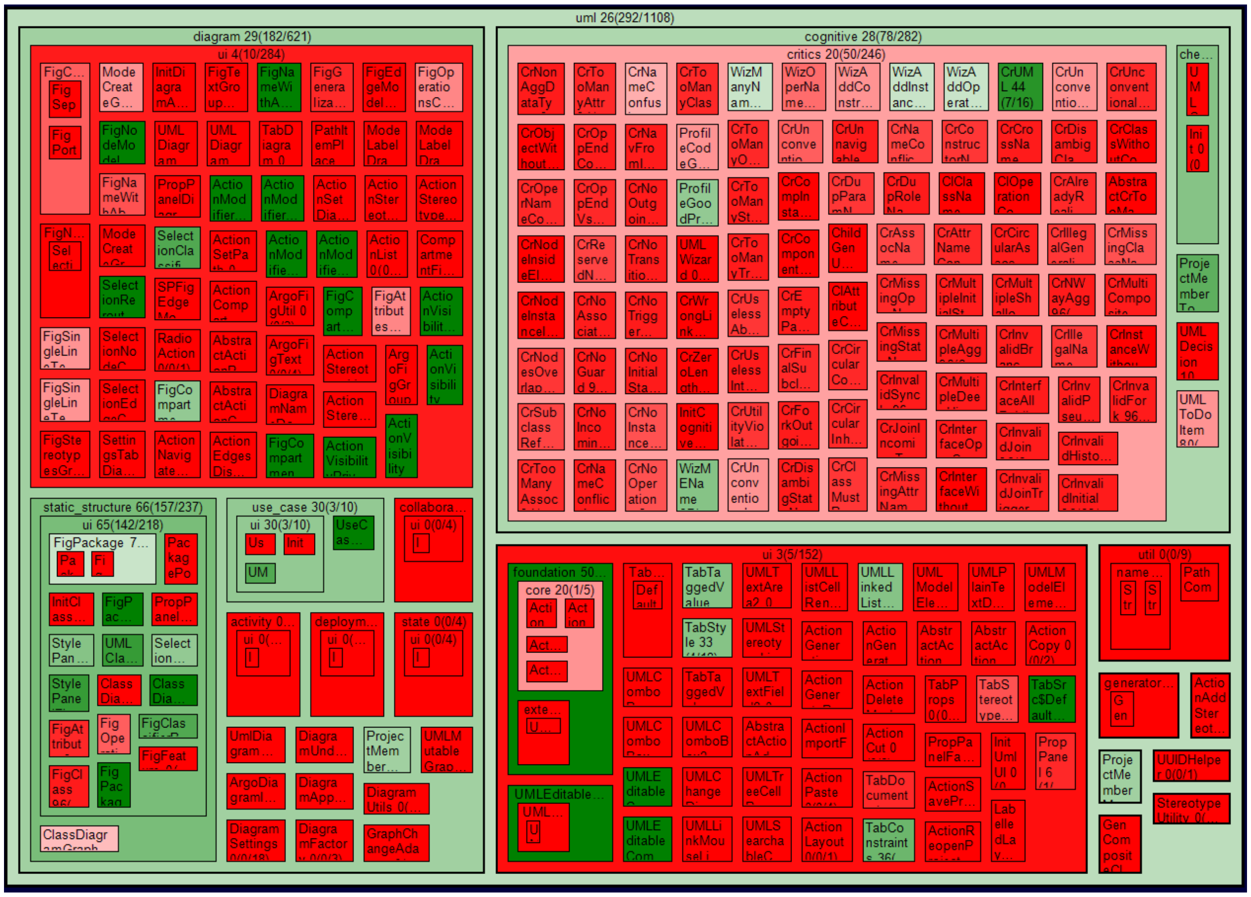

Figure 10. DIR map of Argo UML v035

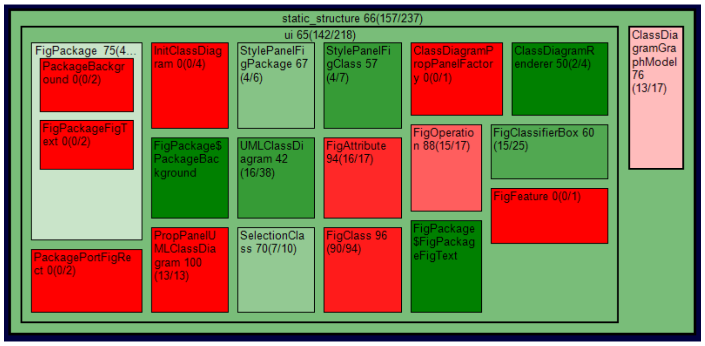

Fig 11: DIR map of the uml.diagram.static_substructure package of ArgoUML

7.2 Version comparison

In this experiment, we compared the DIR maps of versions 028 and 035 of ArgoUML. In order for the maps to be comparable, we must of course run the same use-case in both versions and record their respective execution trace. Next we compute the DIR metric for all the substructures in each system. Finally we build the maps using the union of the substructures of both versions. This is required to be able to easily compare the versions. Indeed if some of the substructures are absent in one of the versions, then the common substructures will be placed in different location in both maps by the Treemapper. Then the maps will be hardly comparable. Finally we create the two DIR maps with the same set of packages and we paint them with the result of the DIR metric for both cases. When a package is present only in one of the versions, it will be painted white in the other version to highlight that it is absent from the version.

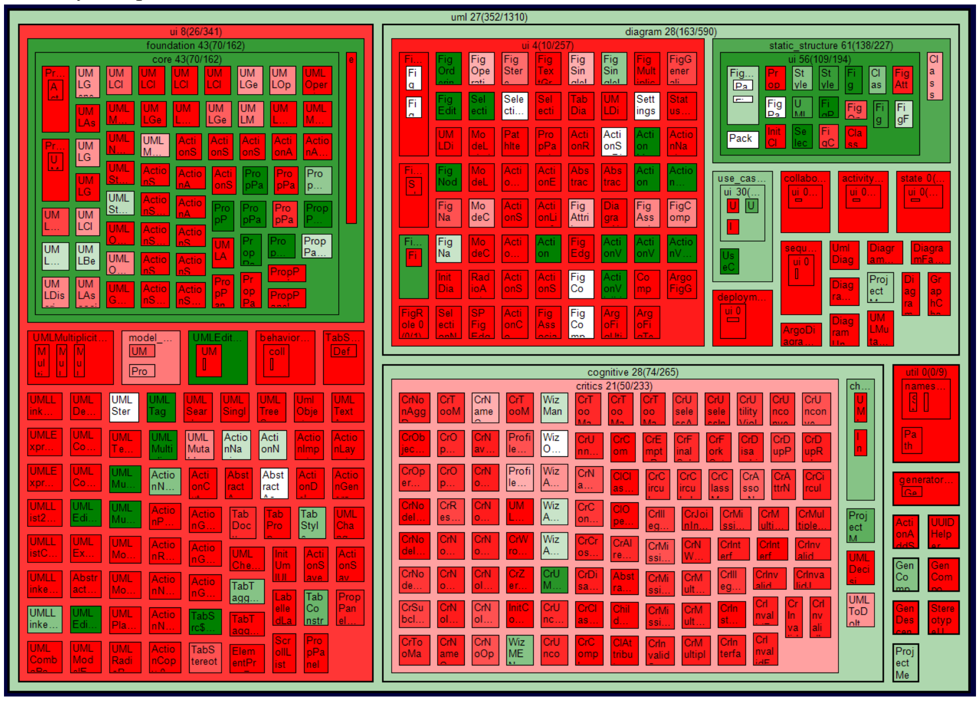

Figure 12. DIR map of Argo UML v028 (union of the substructures with v035)

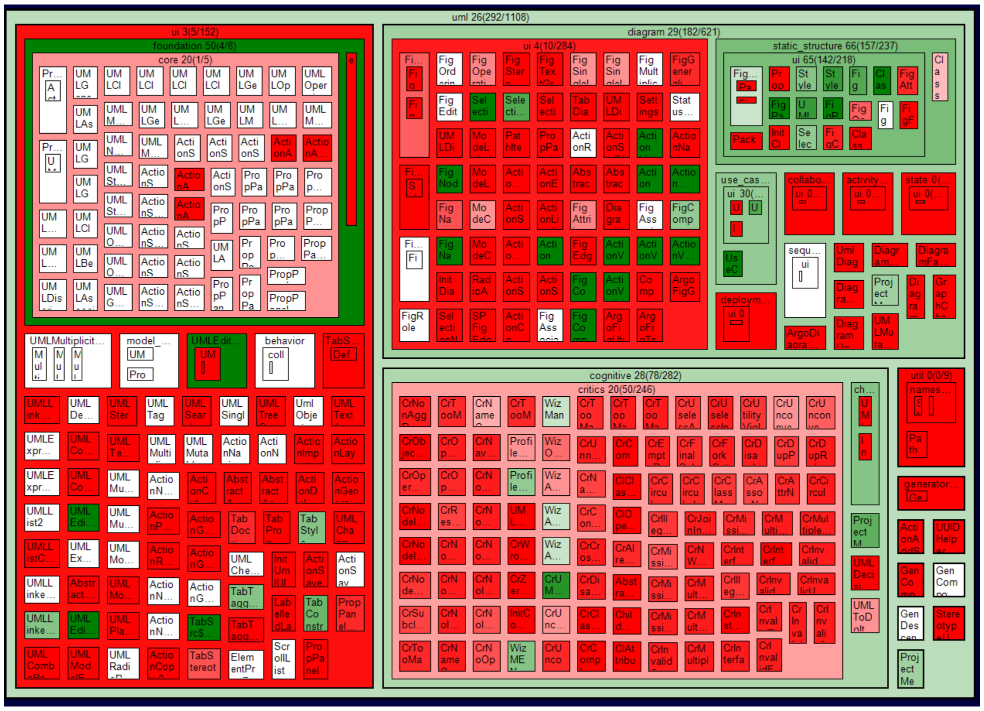

Figure 13. DIR map of Argo UML v035 (union of the substructures with v028)

As we can see by comparing both figures, a lot of the classes involved in the first version disappeared from the second version. A further investigation showed that these classes have indeed been removed from the package in the new version. They have been relocated in another package and are not involved in the implementation of the use-case anymore. What is also notable is the drop of the DIR metric of the uml/diagram/static_structure/ui package (dark green to light green). This may be due to the fact that the package in version 035 implements new classes.

8. RELATED WORK

Most of the metrics relating to inheritance have been defined quite a long time ago and are rather primitive [12]. For example, Chidamber and Kemerer [3] defined the classical Depth of Inheritance Tree (DIT i.e. the number of the level in the inheritance tree where a substructure is positioned) and the Number of Children (NOC, i.e. the number of direct children of a substructure) metrics. Compared with our work, these metrics are static (i.e. not based on the actual execution of the system) and not hierarchical (i.e. can only characterize classes). Wei Li [11] further extended these metrics by adding the Number of Ancestor Classes (NAC i.e. the number of parent classes which differs from DIT when multiple inheritance is used) and Number of Descendants Classes (NDC i.e. the number of direct and indirect children). Again, compared with our work, these metrics are static and not hierarchical. F.T. Sheldon et al. [15] defined the Average Understandability (AU i.e. the average NAC of a substructures among all substructures of an application) and the Average Modifiability" (AM i.e. the average of (NAC+NDC)/2 of a substructures among all substructures of an application). The purpose of such metrics are not clear and again they are neither dynamic nor hierarchical. Brito e Abreu [1] proposed the "Method Inheritance Factor" (MIF) which represents the average ratio of the inherited methods among all the methods a class contains (inherited, implemented and overridden) for all the classes of the application. But this it is too global since it only gives an average value over the entire system. It is then hard to know where and what could be done to improve the value of the metric. Moreover, the metric, as originally proposed, is computed on static source code therefore it does not represent how the methods are used when the system is executed. Gill and Sikka [7] proposed the "Method Reuse Per Inheritance Relation (MRPIR)" metric which represents the average inherited functionality through an inheritance relationship. This metric is quite similar to the MIF but it is less global since it is calculated for each inheritance relation rather than computing an average over the entire application. But it nonetheless computes the inheritance of behavior for all the classes situated below each relation in the inheritance graph. Then, it remains a rather global metric and shares the same other shortfalls as the MIF metric. Sahraoui and Denier [5] proposed a visual representation of some metrics related to inheritance in a form called "sunburst layout". This visualization uses a radial space-filling technique: a sunburst map is a circle with slices and sub-slices representing hierarchies. As opposed to our maps which show the containment hierarchy of substructures, sunburst maps show the hierarchy of inheritance between classes. Moreover, there are two inheritance related metrics shown on the Sahraoui's and Deiner's sunburst : hierarchical relationship and similarity between siblings. Hierarchical relationship characterizes the sub-classing behavior, i.e. whether the class has a tendency to add new methods or to override methods from its parents. This metric is represented on a scale of five colors, ranging from pure extender to pure overrider. The similarity metric shows to which extend the children of a given class shares a common set of functionality. Both of the metrics and the visual representation describes information which is useful to understand the design of inheritance patterns, but they do not help identifying bad design since they do not represent the amount of methods inherited between parent and children substructures. Moreover, the metrics, as originally proposed, are computed on the source code (static approach) and are not hierarchic. Stasko and Zhang [16] claim that a radial space-filling visualization offers a better intuitive overview of a hierarchy when compared to a Treemap. The comparison was done on a file system hierarchy display without the use of colors for substructures, as opposed to our Treemap where all substructures have colors. So the findings of these authors do not really apply to our case. In their book, Lanza and Marinescu [10] proposed a visualization called "class blueprint" which displays a table for each class with a tree-like representation of functionalities (methods) and their dependencies. Methods are classified by visibility and utility (initializers, public, private,…). They use colors to highlight some inheritance related characteristics such as overriding, delegating and extending behavior. This information is useful to understand to some extent the design of a class and some (but not all) characteristics related to inheritance. However it does not help identifying bad design because it does not represent the amount of methods inherited or overridden between parents and children substructures nor does it take the containment hierarchy into account. Ducasse [6] proposed a visualization called "package surface blueprint" similar to the one of Lanza and Marinescu but at the level of packages instead of classes. The visualization shows the inheritance hierarchy of classes contained in each package and also the classes from other packages from which functionality is inherited. This information is useful to understand, to some extent, the design of inheritance class hierarchies, how the behavior is scattered among the packages and some (but not all) the characteristics related to inheritance. But again, like for the work of Sahraoui and Denier and others, it does not help identifying bad design because it does not represent the amount of functionality inherited or overridden between parent and children substructures. Besides, it is a static metric hence it does not analyze the actual inheritance while the system is executing. Finally the metric addresses only one level of packages and cannot be computed for higher level of containment. The work of Makkar et al. [12] focuses on the reuse aspect of inheritance and proposes a new metric that is theoretically well grounded. However, it is again a static metric which does not address the substructures beyond the class level.

9. CONCLUSIONS

This paper presents a new metric to observe the dynamic inheritance patterns among substructures. It brings several contributions. First it defines the notion of inheritance pattern among substructures (classes and packages).

Second it proposes a new metric to characterize the inheritance pattern at any level of containment in the code containment hierarchy. This metric is computed based on the actual running of the system and can therefore measure the real use of inheritance in the system. Since the metric can be applied at any level in the containment hierarchy, it can scale up to whatever system size. It is important to note that the computation of the metric above the class level is not simply an aggregation of the metric values at class level. It is the result of the application of the same computation technique at all these levels. It is the clear definition of the finh and fimpl functions for the substructures that makes it possible. To our best knowledge, no inheritance metric has been defined so far for other containment level than classes. This feature allows one to display informative maps at any level of containment (classes, packages, packages of packages,…) with the exact same semantics at all these levels.

Third, the metric values are presented in a hierarchical map, the Treemap, to allow the interpretation of the inheritance patterns at the global system's scale. We then presented a case study using Argo UML, a medium size system written in Java. This metric can be used to assess the quality of a system from the inheritance pattern point of view and specifically help the maintenance engineer identify the changes that impacted a system after maintenance. This metric is part of a set of tools we developed to assess the quality of a system from an architecture point of view. As the next step we intend to systematically record the different color patterns together with their interpretation in terms of the quality of the inheritance pattern. Then, equipped with these color pattern we could quickly diagnose a system by searching for these patterns in the DIR map of a system.

10. REFERENCES

[1] F. Brito e Abreu, "The MOOD Metrics Set," Proc. ECOOP'95 Workshop on Metrics, 1995

[2] Canedo Blanco M. - Visualisation des patterns d'héritage à l'aide de l'analyse dynamique. Bachelor Thesis. Geneva Schol of Business Adnministration (HEG). Geneva, 2013

[3] S. R. Chidamber et C. F. Kemerer, A Metrics Suite for Object Oriented Design. IEEE Trans. on Software Engineering 20(6). 1994.

[4] J. Daly, J. Miller, A. Brooks, M. Roper, M. Wood, "The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study". In Proc. of the IEEE International Conf. on Software Maintenance, 1995, pp. 20-29.

[5] S. Denier, H.A. Sahraoui, "Understanding the Use of Inheritance with Visual Patterns", in: Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, USA, October 2009, pp.79–88.

[6] S.Ducasse, D.Pollet, M.Suen, H.Abdeen, I.Alloui, "Package Surface Blueprints: Visually Supporting the Understanding of Package Relationships", ICSM 2007.

[7] N. S. Gill et S. Sikka, "Inheritance Hierarchy Based Reuse & Reusability Metrics in OOSD", International Journal on Computer Science and Engineering (IJCSE), vol.3, June 2011.

[8] Hamou-Lhadj A., Lethbridge T.C. - A Survey of Trace Exploration Tools and Techniques. Proc of the Conference of the Centre for Advanced Studies on Collaborative Research CASCON 2004, October 5-7, 2004, Markham, Canada.

[9] Jacobson I, Meyer B., Soley R. - The SEMAT Initiative: A Call for Action, Dr Dobbs journal, December 09, 2009.

[10] M. Lanza et R. Marinescu, Object-Oriented Metrics in Practice, Springer 2006.

[11] W. Li, Another metric suite for object-oriented programming, Journal of Systems and Software, Volume 44, Issue 2, December 1998, Pages 155–162.

[12] G. Makkar, J.K Chhabra, R.K Challa,.- Object oriented inheritance metric-reusability perspective. Int. Conf. on Computing, Electronics and Electrical Technologies (ICCEET) 2012.

[13] L. Mikhajlov et E. Sekerinski, A Study of The Fragile Base Class Problem,1998, European Conference on Object-oriented Programimng.

[14] G. Poels, G. Dedene, "Evaluating the Effect of Inheritance on the Modifiability of Object-Oriented Business Domain Models", in: Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001), Lisbon, Portugal, 2001, pp. 20–28.

[15] F. T. Sheldon, K. Jerath et H. Chung, "Metrics for maintainability of class inheritance hierarchies", Journal of Software Maintenance and Evolution: Research and Practice, Vol. 14, pp. 1-14, 2002.

[16] Zhang et J. Stasko, Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations.In Proc. IEEE Symposium on Information Vizualization, pages 57–65, 2000.