Probabilistic Evaluation of Object-Oriented Systems

Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides and Ignatios Deligiannis[1]
Department of Applied Informatics, University of Macedonia
[1]Department of Informatics, Technological Education Institute of Thessaloniki
54006 Thessaloniki, Greece
E-mail: {it0157, achat, steph}@uom.gr, igndel@it.teithe.gr

Abstract

The goal of this study is the development of a probabilistic model for the evaluation of flexibility of an object-oriented design. In particular, the model estimates the probability that a certain class of the system will be affected when new functionality is added or when existing functionality is modified. It is obvious that when a system exhibits a large sensitivity to changes, the corresponding design quality is questionable. Useful conclusions can be drawn from this model regarding the comparative evaluation of two or more object-oriented systems or even the assessment of several generations of the same system, in order to determine whether or not good design principles have been applied. The proposed model has been implemented in a Java program that can automatically analyze the class diagram of a given system.

1. Introduction

Although the merits of object-orientation are numerous and well-known, one fundamental property of good object-oriented designs, namely flexibility [9], has not been given considerable attention within the metrics community. By flexibility it is meant that the principles of encapsulation, information hiding, abstraction, inheritance and polymorphism should be correctly applied so as to remove any odors of fragility and rigidity [9]. These properties characterize a design that is easy to break or difficult to change, respectively.

More specifically, the addition of new functionality in a system should have as limited impact on existing modules as possible. If the modification of a class method imposes refactoring to a number of existing classes, object-orientation is of limited value. This feature has been successfully captured by the *Open-Closed* Principle which states that software entities should be open for extension but closed for modification. There are many design principles [9], heuristics [12] and patterns [4] that help to enforce good programming practices in order to built more stable and flexible systems.

Numerous metrics are available for evaluating several aspects of a software system [3] and since the initial work of Chidamber & Kemerer [2] the field of software metrics has been expanding to the object-oriented domain as well. However, we believe that most of the existing metrics evaluate the degree of object-orientation or measure static characteristics of the design, which are not always helpful in answering the question whether a specific design is good or not [7]. When trying to answer such a question, an expert would assess the conformance of the design to well established rules of thumb, heuristics and principles. This work attempts to systematize this process, at least towards one desirable property of object-oriented systems, namely flexibility.

In this paper we attempt to estimate the potential flexibility of a given system, employing a probabilistic approach. In brief, the goal is to assess the probability that a method, class or the system itself, will change in a future generation. In order to calculate these probabilities, axes of change, through which a change in one module can affect another module of the design, are identified. The analysis is based on simple probabilistic analysis and can be easily automated. The extracted probabilities can be refined by taking into account past data for an evolving design, such as the ones developed in the open source community.

The rest of the paper is organized as follows: Section 2 briefly mentions the types of changes that have been considered, while in section 3 the analysis process is explained. Two sample applications are extensively analyzed in section 4, while in the next section a software tool that has been developed in order to automate the proposed methodology is briefly described. In section 6 we discuss the limitations and our plans for future work. Finally we conclude in section 7.

2. Types of changes

According to the rules governing object-orientation there are changes that affect in an absolute manner other classes of the system and changes that do not affect other portions. The following observations and terminology focus mainly on systems developed using Java; However, the conclusions can be easily ported to any object-oriented programming language.

2.1 Changes with a definite impact

Interface: The addition of a new method in an interface or a change in the signature of an existing method enforces all classes implementing this interface to modify themselves in order to be compliant with this change.

Instance: A change in the signature of the constructor of one class implies that all classes that create instances of this class, have to modify the constructor call.

Abstract class: The addition of a new abstract method or a change in the declaration of an existing abstract method enforces all classes that extend (inherit) this abstract class to modify the corresponding implementing methods.

Non-abstract class: In case one class employs the constructor of its superclass or explicitly uses a method of the superclass (e.g. via the `super` identifier), any change in the signature of the constructor or method imposes the subclass to be modified, in order be compliant with its superclass.

Methods: A change in the declaration of one method (whether static or non-static) enforces all classes using that method to modify the corresponding method calls.

2.2 Changes with no impact

Body changes: All changes in the body of the constructor of one class or in the body of one method do not affect classes that create instances of that class or employ that method respectively (encapsulation).

The proposed model employs for the evaluation of object-oriented systems changes with a definite impact on other classes of the system, i.e. changes that can propagate through the system. Such changes are assumed to propagate to all possible target classes. Consequently, our analysis can be described as a worst-case analysis of the system under study.

3. Probability estimation

3.1 Axes of change

In order to emphasize the interference between the classes of a system, the proposed model defines several axes of change through which a change in a class can affect other classes enforcing them to be modified. Each class can change because of its involvement in one or more axes of change.

In general, axes fall in two main categories, each one containing two sub-axes of change:

Inheritance axis:

- *Interface*: A class implements one or more interfaces (inherits pure abstract classes for C++)
- *Class Inheritance*: A class extends another class and calls one of the super class methods or constructors

Reference axis:

- *Direct instance*: A class instantiates an object (employing the `new` operation for example)
- *Reference*: A class employs an object as a parameter in its constructor or one of its methods. (In order for a change in the class' object to affect the using class, one of the object methods should be called).

The above two axes are related to any possible modification of a class' probability of change due to other classes and therefore will be called **external axes** of change.

However, since each class can also change due to modifications to the class itself, we define also an **internal axis** of change that summarizes all possible causes of change: modification to method declarations, addition of new methods/attributes, change of implementation, etc. This axis, although not affecting other classes, has to be taken into account, since a class with a "bad" history of changes will contribute to the overall system's probability of change.

At this point, it should be noted, that dependence on library classes (such as STL in C++ or API's in Java) is not considered a source of changes, since these classes are not likely to change. Axes involving such classes are not taken into account in the analysis.

3.2 Analysis

Given a class A in which a change can occur, the aim is to calculate the probability that another class B that can be affected, has to change.

The probability that a class C might change in the next generation of the software will be denoted as $P(C)$. Since the only possible events are a) that the class changes and b) the class does not change (i.e. the sample space is S = {"*change*", "*no change*"}), the proposed probability on the sample space of the two outcomes satisfies the following properties:

1. Any probability is a number between 0 and 1: $0 \leq P(C) \leq 1$. (The probability is physically measured in a class that undergoes a number of modifications through successive generations, as the ratio of the number of changes over the number of generation upgrades. Since this number can be at least 0 and at maximum 1, it follows that the range of the probability function is [0..1].)
2. The sample space, S, of all possible outcomes has a probability of 1: $P(S) = 1$.
3. Since the two events are disjoint, $P(\text{"}change\text{"} \cup \text{"}no\ change\text{"}) = P(\text{"}change\text{"}) + P(\text{"}no\ change\text{"})$ and thus P is a valid probability measure [11].

As already mentioned, every class is subject to change due to its involvement in several axes of change. Since, even one change will be a reason for editing the code, the probability in which we are interested is given by the *joint probability* of all events (i.e. change in any axis), also known as probability of the OR of two or more events. For example, if a class *A* can change due to two axes of change, *axisA* and *axisB*, the probability that *A* will change is given by [1]:

$$
\begin{aligned}
&P(A) = P(A : axisA \cup A : axisB) \\
&= P(A : axisA) + P(A : axisB) - P(A : axisA \cap A : axisB) \\
&= P(A : axisA) + P(A : axisB) - P(A : axisA) \cdot P(A : axisB)
\end{aligned}
$$

This probability is always lower than one. P(*A*:*axisX*) symbolizes the probability that class *A* will change due to *axisX*.

Since the final goal is to be able to characterize a system according to its probability of change in a future generation, the probability of the system is calculated as the mean value of the probabilities of all classes in the system. The mean value has been selected since it captures not only the probabilities of each class but takes also into account whether the effect of change is localized or not: For a good object oriented design, change is expected to affect a limited number of classes, while the majority will remain unaffected. This is reasonably captured by the mean value. On the other hand, for a badly designed system where all functionality has been placed into one class, the mean value will reveal the odors of rigidity and fragility [9].

One issue that has to be clarified before the application of the proposed model is the order according to which probabilities will be estimated. For any inheritance tree, probabilities should be calculated starting from the classes higher in the hierarchy, since the probability of a superclass is required in order to extract the probability of a subclass.

Another issue that requires non-trivial handling concerns mutually dependent classes. For example, consider the following hypothetical design where classes *C*1 and *C*2 are abstract:

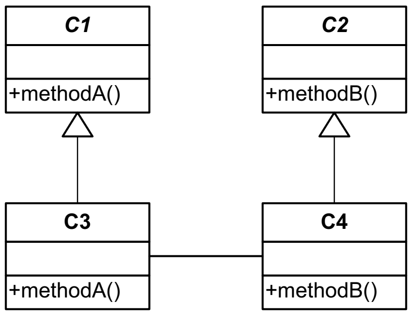


Figure 1: Circular dependencies of probabilities

In the above system, the probabilities of classes *C*3 and *C*4, have a mutual or circular dependence on each other. The probability for class 3 is given by:

$$
\begin{aligned}
P(C3) = P(&C3 : extension\ axis \cup C3 : reference\ axis \cup \\
&C3 : internal\ axis)
\end{aligned}
$$

where $P(C3 : reference\ axis) = P(C4)$

The probability for class *C*4 is extracted in a similar manner. However, the above probabilities lead always to a set of first-order equations of the form:

$$P(C3) = a + (1 - a) \cdot P(C4)$$

$$P(C4) = b + (1 - b) \cdot P(C3)$$

where *a* and *b* are coefficients resulting from all other probabilities. Unfortunately, no matter what the values of *a* and *b* are, the solution to such a system is always equal to 1 ($P(C3) = P(C4) = 1$), as a result of the circularity in the joint probabilities.

In order to cope with this problem without adding to the complexity of the software, our implementation initially considers the two or more classes as not associated (their association is temporarily broken) and proceeds to the estimation of probabilities as already described. Once the probabilities are extracted, the association is restored and the probabilities for each class are calculated again as the joint probabilities of their prior value and that of the associated classes.

To summarize the proposed methodology, for each class the probability of change due to its involvement to any axes of change is evaluated first. Next, the joint probability due to all axes of change is extracted and as a last step the system's probability of change is calculated as a mean value.

3.3 Assumptions

In the previous analysis, it has been assumed that the events associated with each axis of change (i.e. a change due to *axisA* and a change due to *axisB*) are independent, which is reasonable since the outcome of one event does not affect in general the probability of the other.

In addition, in case of multiple axes of change associating two classes (e.g. inheritance and containment), these axes are counted as one. That is because, under the worst-case analysis model, even one axis is sufficient for propagating the change from one class to another.

Initially, for a new design, since there are no statistical data from past generations in order to estimate the frequency of change in any part of the system, the model assumes that for classes where changes originate, the probability of change is 0.5 (this includes the probability related to the internal axis of change).

Finally, since this is a worst-case analysis, if one class changes, all classes to which the change can eventually propagate, are assumed to be affected, regardless of the nature of that particular change.

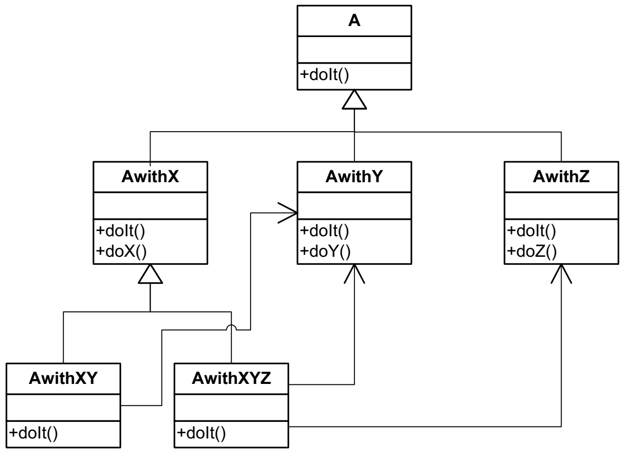


Figure 2: Naïve design

4. Application Results – Discussion

There is a general agreement on the fact that Design Patterns improve the quality of an object-oriented design by applying best practices for *design for change* [4]. Such a qualitative improvement should be measurable by the applied metrics; in order to evaluate the efficiency of the proposed model we have measured the system probability of change for two designs that have the same functionality: One "naïve" design which does not employ any Design Patterns and a more sophisticated solution in which a suitable Design Pattern has been used. The first case tests the proposed model against the *Decorator* Design Pattern while the second concerns a system that exploits the *Bridge* Design Pattern [4].

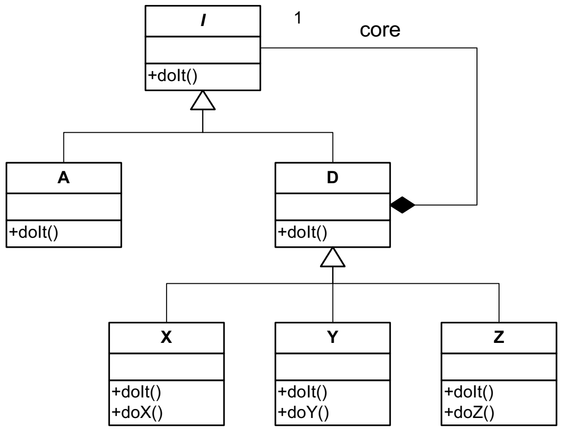


Figure 3: Improved design using Decorator DP

4.1 Decorator

For the naïve design the system's basic functionality is simply enhanced using inheritance and composition by adding new subclasses that inherit a base functionality and call the methods of contained objects [5]. The UML diagram corresponding to this design is shown in Figure 2. All `doIt()` methods call the `doIt()` method of their superclass, among with calling any specific methods declared in that class or accessed through references.

The improved design has anticipated that future classes with enhanced functionality might be added in the system and therefore employs the Decorator pattern. The corresponding UML class diagram is shown in Figure 3. This time classes *X*, *Y*, *Z* call in their `doIt()` method the corresponding method of the *D* class, while *D* class calls in the `doIt()` method the corresponding method of its contained core object.

The two systems vary dramatically in their need to undergo modifications in case of a change. As an example, a change in the signature of the `doY()` method in the "naïve" design will affect classes *AwithXY* and *AwithXYZ* through the corresponding reference axes as displayed in Figure 4. Shadowed classes and methods with a border will be affected.

The class diagram of the improved system will not change at all in order to accommodate the new functionality since each object can be dynamically decorated with the functionality of subclasses of class *D*. The difference in flexibility between the two designs is even more intense if a new kind of functionality (e.g. AZXY) is required, which demands the addition of a new class in the naïve system, while the static structure of the improved design need not be changed at all.

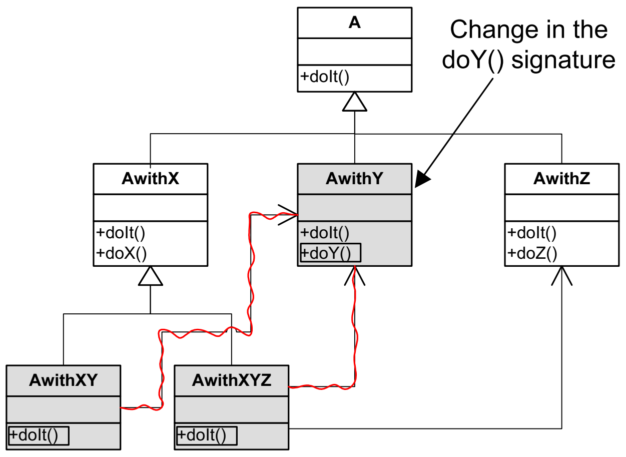


Figure 4: Propagation of changes

Analysis

Naïve Design: Classes *AwithX*, *AwithY* and *AwithZ* have all one axis of change with regard to the extension of class *A* and are also themselves subject to change (internal axis). A change in the declaration of method `doIt()` of superclass *A* will cause a change to the subclasses with probability 1, because method `doIt()` of the subclasses calls the `doIt()` method of the superclass (through the `super` keyword). Therefore the conditional probability $P(AwithX \mid A)$ is equal to one. Assuming that a change in class *A* will occur with a probability of 0.5, then

$$P(AwithX : extension\ axis) = P(AwithX \mid A) \cdot P(A) = 0.5$$

The same holds for classes *AwithY* and *AwithZ* as well.

The probability of change for these classes is given by the joint probabilities of the extension and the internal axis:

$$P(AwithX) =$$
$$P(AwithX : extension\ axis \cup AwithX : internal\ axis) =$$
$$0.5 + 0.5 - 0.5 \cdot 0.5 = 0.75$$

Class *AwithXY* is involved in two external axes, one with respect to the extension of class *AwithX* and one with respect to its reference to class *AwithY*. In a similar manner:

$$P(AwithXY : extension\ \ axis) =$$
$$P(AwithXY \mid AwithX) \cdot P(AwithX) = 1 \cdot 0.75 = 0.75$$

Concerning the reference axis, a change to class *AwithXY* could be caused by two events: a change in the declaration of the constructor of *AwithY* or by change in the signature of the `doIt()` method in class *AwithY*. Consequently, $P(AwithXY \mid AwithY) = 1$.

The probability of change for class *AwithY* has already been calculated and finally:

$$P(AwithXY : reference\ axis) =$$
$$P(AwithXY \mid AwithY) \cdot P(AwithY) = 1 \cdot 0.75 = 0.75$$

The final probability that a change occurs in class *AwithXY* is given by the OR of the probabilities with respect to each axis (at this point the internal axis should also be taken into account):

$$\begin{aligned} P(AwithXY) = P(AwithXY &: extension\ axis\ \cup \\ AwithXY &: reference\ axis\ \cup \\ AwithXY &: internal\ axis) = \\ &= 0.969 \end{aligned}$$

Similarly, class *AwithXYZ* is involved in four axes of change (one extension axis with respect to class *AwithX*, two reference axes with respect to *AwithY* and *AwithZ* and the internal axis). The final propability is again calculated as the OR of the probabilities for each axis, resulting in: $P(AwithXYZ) = 0.992$.

The system probability of change as already explained, is estimated as the mean value of the probabilities of all classes:

$$P_s = \frac{\sum_{i=1}^{6} P(C_i)}{6} = 0.785$$

Improved Design: Abstract classes *D* and *A* participate in only one external axis of change with respect to the declaration of method `doIt()` in interface *I*. (Class *D* appears to have a second axis due to the containment relationship with class *I*. However, as already mentioned, multiple axes of change from one class to another are counted only once). Any change in the signature will cause a change to the implementing classes with probability 1, i.e. $P(D \mid I) = 1$. Therefore, under the same assumptions as previously:

$$P(D : extension\ axis) = P(D \mid I) \cdot P(I) = 1 \cdot 0.5 = 0.5$$

Consequently, considering also the internal axis:

$$P(A) = P(D) = 0.75$$

Classes *X*, *Y*, *Z* have only one external axis regarding the extension of their superclass *D*. Any change in the signature of method `doIt()`, will cause definite changes to the subclasses, since the `doIt()` method of the subclasses calls the `doIt()` method of the superclass. Taking the joint probability due to this axis and that due to internal changes, results in:

$$P(X) = P(Y) = P(Z) = 0.875$$

The system probability is given by:

$$P_s = \frac{\sum_{i=1}^{6} P(C_i)}{6} = 0.771$$

The system probabilities in the two designs might seem very similar. However, it becomes clear that the situation will get worse as new functionality is added: In case AZXY functionality is required, the naïve design in order to accommodate this change in the requirements will have to add a new class AwithZXY worsening the system probability of change ($P_S \to 0.815$). On the other hand the improved class design and its corresponding probability will remain unchanged.

However, such a change favors the design pattern solution which has been applied in anticipation of such kind of changes. If another kind of change occurs (e.g. the addition of a formal parameter in the declaration of `doIt()` method) both systems will be affected to a similar degree.

Discussion

The results from the probabilistic analysis validate well-established rules of good object-oriented design: Although "deep and narrow" inheritance hierarchies should be preferred theoretically [12], in practice shallow hierarchies prove to be more maintainable and extendible [10], [13]. The complexity that deep hierarchies incur has also been mentioned in the discussion of the DIT metric [2].

Another well-known heuristic states that the number of classes with which another class collaborates should be kept minimal [12]. The same point is implied by the Law of Demeter: "Each unit should only use a limited set of other units" [8]. The proposed probability measure reveals clearly any violation of this principle: For a system where the number of associations between classes becomes large, the axes of change through which changes can propagate will also increase, resulting in a high probability of change.

As a last example, the proposed measure also validates the *Interface Segregation Principle* (ISP) according to which *clients should not be forced to depend on methods that they do not use* [9]. Stated differently, a designer should avoid interface pollution, i.e. the incorporation of methods solely for the benefit of one of its subclasses. One solution to this problem is achieved by separation of interfaces (or multiple inheritance in C++), since in that case, only the interested parties will inherit the required additional functionality. This fact is also recognized by the probabilistic analysis: A fat interface which is being implemented by all of its subclasses will lead to a much higher probability of change, than a design in which only some of the subclasses implement additional interfaces. This might not be clear in the preceding discussions where all changes are assumed to have a probability of 0.5, but in a real environment a fat interface that would have a worse history of changes than a non-polluted one, would unavoidably affect all of its implementing classes.

4.2 Bridge

The second example refers to a primitive application that can draw rectangles and circles with either of two drawing programs (DP1 and DP2). Since the rectangles and circles know during instantiation which drawing program to use, the straightforward solution is to have two different kinds of each shape, one for each drawing program. This novice designer's approach results in the UML class diagram shown in Figure 5 that is based on subclassing of an abstract base class to provide alternative implementations.

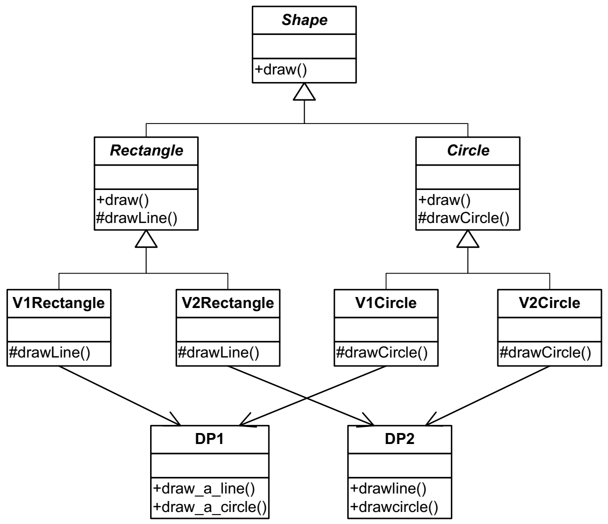


Figure 5: Naïve design for Drawing application

Clearly this solution suffers from combinatorial explosion in case of new drawing programs or new shapes and exposes the problem that arises from the overuse of inheritance [13].

On the other hand, the improved design anticipates that future requirements for further kind of shapes might arise and attempts to "decouple an abstraction from its implementation so that the two can vary independently" [4]. Although for the design of a single shape the development team would obviously come up with the same solution as in the previous case, when more shapes have to be drawn, the designers choose to encapsulate what is varying (Shapes and Drawing programs) and to favor composition over inheritance. This calls for the application of the Bridge Pattern, which according to the GoF [4] "allows the combination of different abstractions and implementations and the extension of them independently". The resulting UML diagram is shown in Figure 6.

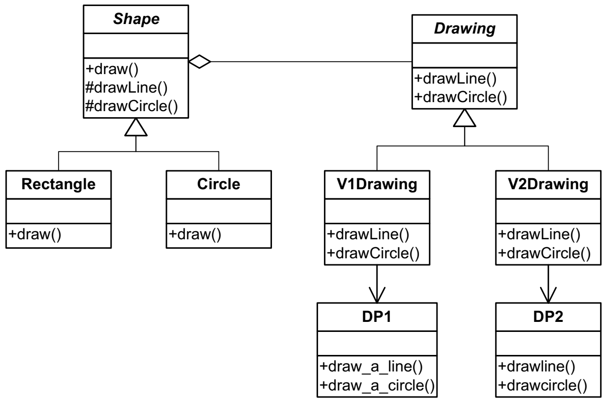


Figure 6: Improved design for Drawing application

Analysis

Naïve Design: Classes *Rectangle* and *Circle* are involved in one extension axis plus their internal one. Each of the concrete Shape classes is involved in one extension axis, one reference axis to the corresponding drawing program as well as their internal one. The final system probability of change is equal to 0.75.

Improved Design: The inheritance tree has now a lower depth resulting in lower theoretical values for the probability of change. Classes *Rectangle* and *Circle* are involved in three axes: one extension axis with regard to *Shape* class, one reference axis with regard to the *Drawing* class and their internal one. Each of the added classes *V1Drawing* and *V2Drawing* is also involved in two external and one internal axis. The system probability of change is 0.734.

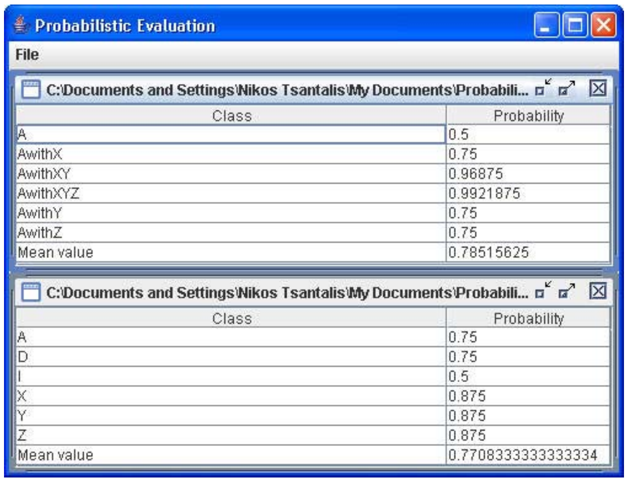
Again, this might not be a very distinct difference for a system that is supposed to be better designed than the initial one. However, as more requirements are implemented, the stability of the improved design is reflected in an increase of the difference between the two probabilities of change: Consider for example the addition of a new Shape (e.g. a Triangle), which can be drawn by either of the two drawing programs. The system probability of change in the naïve design becomes 0.781 while that of the improved design becomes 0.756.

5. Implementation

One of the goals of analyzing the probabilities of change in a system, is to enable the automation of the process by means of an appropriate parser and analyzer. To this end, a Java program has been developed that parses the complete hierarchy of directories that include the project under study (or an input XML file containing the description of the static structure of an object oriented design. The XML file includes tags for annotating each class with the required information concerning associations and inheritance relationships). Next, the program applies the aforementioned methodology by calculating probabilities for each class, propagates these probabilities to the affected classes and finally calculates the system's probability of change.

The tool as well as sample XML files can be downloaded from [6].

The developed software can also generate system probabilities for various designs in order to enable a comparative analysis of several alternatives. A sample screenshot with estimated probabilities for the naïve and improved designs shown in Figures 2, 3 is shown below. The probabilities for each class can be either set freely by the designer or extracted from previous data as already mentioned in order to explore the evolution of a design.



Probabilistic Evaluation
File
C:\Documents and Settings\Nikos Tsantalis\My Documents\Probabili...

Class	Probability
A	0.5
AwithX	0.75
AwithXY	0.96875
AwithXYZ	0.9921875
AwithY	0.75
AwithZ	0.75
Mean value	0.78515625

C:\Documents and Settings\Nikos Tsantalis\My Documents\Probabili...

Class	Probability
A	0.75
D	0.75
I	0.5
X	0.875
Y	0.875
Z	0.875
Mean value	0.7708333333333334

Figure 7: Sample screenshot for the application

6. Future Work

So far, it has been assumed that for any unknown change, the probability is 0.5. In case actual data can be collected from several generations of the system, these probabilities can be refined in order to extract more realistic information. It is obvious that changes due to new requirements or maintenance will not have a uniform distribution among all components of the system. To this end, experiments can be performed to "tune" the proposed model to past data.

A rich source of information for such kind of past data can be open source projects, which have evolved through several generations. Our current goal is to apply the proposed methodology to such kind of projects in order to refine the class probabilities but more important to assess the efficiency of probability estimations.

For such systems high class probabilities could highlight modules of the design that have reached a level of saturation, beyond which any attempt to enhance the system functionality would cause severe changes to the existing code. Such a threshold could possibly indicate that appropriate refactoring is necessary.

7. Conclusions

One methodology for estimating the overall degree of flexibility for an object-oriented design has been proposed. The goal is to assess the probability that a class/system will change in a future generation. For a well-designed system, e.g. one that employs Design Patterns where appropriate, this probability should be low, while a system that can easily break is characterized by large probability values. Apart from the probability that a change occurs in a class itself, changes can propagate through so-called axes of change, affecting the overall probability value. The results confirm the expectation that the application of good programming principles helps to built a more flexible system, while the proposed methodology can be easily automated and applied to any object-oriented software system.

8. References

[1] A. V. Aho and J. D. Ullman, *Foundations of Computer Science*, 3rd ed., Computer Science Press, New York, 1995

[2] S. R. Chidamber, C.F. Kemerer, “A Metrics Suite for Object Oriented Design”, *IEEE Transactions on Software Engineering*, vol. 20, no. 6, June 1994, pp. 476-493.

[3] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach,* International Thompson Publishing, Boston, MA, 1997.

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

[5] Huston Design Patterns http://home.earthlink.net/~huston2/dp/patterns.html

[6] http://java.uom.gr/~nikos/probabilistic-evaluation.html

[7] C. Kirsopp, M. Shepperd, S. Webster, “An Empirical Study into the Use of Measurement to Support OO Design Evaluation”, *Proc. 6th IEEE Int. Symposium on Software Metrics*, Boca Raton, FL, USA, Nov. 1999, pp. 230-241.

[8] K. J. Lieberherr, I. M. Holland, "Assuring Good Style for Object-Oriented Programs", *IEEE Software*, vol. 6, no. 5, September 1989, pp. 38-48.

[9] R. C. Martin, *Agile Software Development: Principles, Patterns and Practices,* Prentice Hall, Upper Saddle River, NJ, 2003.

[10] B. K. Miller, P. Hsia, C. Kung, "Object-Oriented Architecture Measures", *Proc. 32nd Hawaii International Conference on System Sciences* (HICSS'99), January 1999.

[11] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, 2nd ed. McGraw-Hill, New York, 1984.

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

[13] A. Shalloway, J. R. Trott, *Design Patterns Explained*, Addison-Wesley, Boston MA, 2002.