Identifying Problematic Legacy Algorithms by their Extension Complexities

Eliezer Kantorowitz
Computer Science Dept.
Technion- Israel Institute of Technology
32000 Haifa, Israel
kantor@cs.technion.ac.il

Abstract
An object oriented reengineering of a legacy system led to the development of the *extension complexity* concept, which enables fast detection of problematic algorithms. A software system may be considered as an implementation of a number of different algorithms. An evaluation of the extension complexities of these algorithms can reveal algorithms, that are difficult to implement and to extend. Similar to space and time complexities the extension complexity of an algorithm may be evaluated in few minutes and give an indication of its suitability.

This paper is based on observations made in an object oriented (O-O) reengineering of a roughly 20 years old CAD legacy system for design and manufacture of large mechanical structures[KA97]. Such a structure is typically composed of $2 \cdot 10^5$ different parts belonging to 85 different part types. The legacy system was composed of a number of distinct subsystems. Moving data from one subsystem to another was done by the operators. This semi manual operation gave room for many human errors. The legacy system had also a number of bugs that could not be removed with a reasonable debugging effort. Typically the removal of one bug involved the creation of new bugs. The appearance of a new kind of manufacturing robots required a major system extensions. It was realized that such an extension was not feasible, and it was decided to reengineer the sub-system for the design of a steel structure composed of parts belonging to 9 different types. The idea was to reengineer this subsystem such that it could be extended to handle all the 85 different types of parts of the manufactured structures, and such that future extensions would be relatively easy. The new system was based on a commercial O-O database system. Software bridges were built to move data between the old sub-systems and the new system. This enabled using the old and the new software together.

The design of a large mechanical structures starts with the first part. Thereafter the engineers add the second part and so on. The central activity is thus adding a single part to the already constructed structure. Adding a part may however require changes in parts adjacent to it in the structure. It may for instance be necessary to push some of these adjacent parts to make room for the new part. Changes in these adjacent parts may involve further changes in parts adjacent to the adjacent parts, and so on. The parts must obviously be designed such that change propagation stops. The algorithm for calculation of the *change propagation* was one of the problems of the legacy system. The legacy algorithm considered all types of faces of the n=9 different part types. With 6 different faces per type, there were 6n types of faces. The legacy system had a separate piece of code for handling each one of the $(6n)(6n)=36n^2$ possible types of interfaces between the 6n different types of faces. In the terminology we developed later it may be said that the implementation complexity of the legacy change propagation algorithm is $O(n^2)$. With n=9 in the legacy system the implementation involved $36 \cdot 9^2=2916$ different code segments! It is, therefore, not surprising that getting this code right was difficult. Extending the legacy system to support one further part types would involve writing of $36(n+1)^2-36n^2=36(2n+1)$ new pieces of code. In our terminology the extension complexity of the legacy change propagation algorithm is O(n). As n grows the effort for extending the system grows linearly. Extending the legacy system from 9 to 10 part types would thus involve $36*(2 \cdot 9+1)=684$ pieces of code! This is a part of the explanation of why it was difficult to extend the legacy system.

The O-O change propagation algorithm developed by the author of this paper [KA97] considered, in accordance with the O-O paradigm, each object type separately. Such an object, which models a part in the mechanical system, may be pushed to give space to an adjacent part. The object can do all the required computations without knowing the properties of the part that pushed it. The object needs only to know the direction in which it is being pushed and by how much it is pushed. The message that the object receives thus contains only information on the direction and amount of the push, nothing about the sender. The object can then compute its own movement. If as a result of this movement, it has to push one of its neighbors, it will send it a message giving direction and amount of push. This will continue until no further pushing is needed. To implement this algorithm only one method is required for each one of the 6 faces of the object type. This method calculates what happens when the face is pushed. For n object types 6n methods are needed. The implementation complexity of the O-O algorithm is thus O(n). For 9 object types the O-O algorithm thus requires 9*6=54 methods as compared to the 2916 code segments of the legacy algorithm. The code that implemented the O-O algorithm was considerably smaller and easier to debug than the code that implemented the legacy algorithm. Extending the O-O change propagation algorithm to support a new object type (class) requires writing of 6 methods, i.e. the extension complexity is O(1). The amount of code required to extend the algorithm to support one further object type is thus constant and independent of n. This was also observed in praxis. It was, therefore, relatively easy to extend the O-O system from the 9 object types, that were possible in the legacy system, to the 85 different object types, that were needed to model the actual system. The table below compares the complexities of the two algorithms:

	Legacy algorithm	O-O algorithm
Implementation Complexity	$O(n^2)$	O(n)
Extension Complexity	O(n)	O(1)

The implementation and extension complexity concepts will be explained in the following. Some examples of their application will be given later.

Poor programmers can implement an algorithm in a way that produces a considerable amount of superfluous computations and code. The performance and implementation characteristics of such a code may be quite different from the complexities of the algorithm. It is therefore assumed in the following that sound software engineering practices are employed in the implementation of the algorithms. This means that all program modules (functions and classes) are reasonably small and simple, such that their codes may be quite easily verified. Algorithms implemented by a single method are therefore not expected to involve implementation problems. Algorithms implemented by methods residing in n different classes may be difficult to implement. An algorithm of this kind involves at least n different methods residing in n different environments (classes). Getting all this to work adequately can be quite difficult. Our hypothesis is therefore that the number n of different classes that are involved with algorithm is the principal parameter for estimating the amount of code that implements the algorithm. One of the difficulties with extending an algorithm to support a new class is related to getting the new class to work together with the other classes employed by the algorithm. It is therefore also hypothesized that the principal parameter for characterizing the amount of code required for extending the algorithm is the number n of different classes that it employed. In order to express this we introduce the following definitions**:**

> The *domain of an algorithm* is the set of all the object types employed by the algorithm.
>
> The *size of the domain of an algorithm* n is the number of different object types in the domain of the algorithm.

It is recalled that one of the basic principles of OOP is to model the problem domain as precisely as possible. The construction of such a model is typically based on an analysis of all the envisioned use cases [JA92]. The set of object types identified in this analysis represent therefore a model of the domain (an object schema) that is expected not to be biased toward any one of single use case. The definition of the size of the domain of an algorithm, that is based on a count of such object types, is therefore expected not be oriented toward any one of the compared algorithms. The ability to produce this expectedly neutral measure for the domain of an algorithm is one of the reasons for employing OOP concepts in the definition of the implementation and extension complexities. Users of other programming paradigms may determine the size of the domain of an algorithms in a similar way, i.e. by making some kind of object analysis of the problem domain and counting the number of problem domain object types employed by the algorithm. We can now define:

> The *implementation complexity* of an algorithm is an indicator of the number of code segments required to implement the algorithm as function of the size of the domain of the algorithm. A code segment may be a class, a function, or any other unit of code. We do not look for an accurate estimate of the required amount of code, as may be achieved by an elaborate time consuming software metrics analysis [FE91].
>
> The *extension complexity* is a measure for the number of code segments required in order to extend the domain of an algorithm with a single new object type.

The above example suggests that an extension complexity of *O(1)* is desirable. It means that the effort required to extend the domain of the algorithm with one object type is independent of n. Extension complexity of *O(N)* is usually undesirable as it means that it becomes exceedingly difficult to extend the algorithm as n grows. We introduce therefore the notion:

a *simple algorithm* as an algorithm whose extension complexity is *O(1).*

The following theorem has been proved:

Theorem: An algorithm having an extension complexity O(1) has an implementation complexity of O(n).

A simple algorithm has therefore an implementation complexity of *O(N).* It can be shown that an implementation complexity of *O(N)* does not imply that the extension complexity is *O(1).* The extension complexity is in this sense more fundamental concept than the implementation complexity. From a practical point of view we are mostly interested in simple algorithms. We shall therefore in the following examples only check if the algorithms have an extension complexity *O(1)* , i.e. are simple.

The following example shows that polymorphism facilitates the design of simple algorithms. Consider, for instance, that we have a super class `Shape` from which we derive the sub classes `Square` and `Circle`. The super class has a virtual method `area()` which has different concrete implementations for `Square` and `Circle`. We consider now an algorithm `total_area` for calculation of the sum of the areas of all the object instances derived from `Shape`, i.e. both `Square` and `Circle` instances. The algorithm is to send the `area()` message to all the objects and to sum the returned results. In order to evaluate the extension complexity of this algorithm we consider the case of adding one new class, say `Triangle`, which is also derived from `Shape`. In order to extend the `total_area` algorithm it is required to implement the `area` method for the new `Triangle` class, i.e. one code segment. The extension complexity is thus *O(1)* , i.e. the algorithm is simple.

Genericity can also facilitate the design of simple algorithms. Consider, for example, a simple inventory system composed by a number of different lists, e.g. a list of books and a list of journals. Assume that all the different lists are implemented with a single C++ template for a list. The inventory system may have an algorithm `list_all_items`. This algorithm will step throughout all the lists of the inventory and print their content. Consider the extension of the system with a list of CDs. This involves the declaration the class `Cd` with an appropriate output operator. Then the list template is employed for creating the class list of CDs. We have thus implemented two classes in order to extend the algorithm. The extension complexity is thus *O(1),* i.e. the algorithm is simple.

References:

[FE91] N.E. Fenton, '*Software Metrics a Rigorous Approach',* Chapman Hall, New York, 1991

[JA92] I. Jacobson, M.Christerson, P. Jonsson, and G. Overgaard, '*Object Oriented Software Engineering'*, Addison Wesley, Reading. Massachusetts, 1992

[KA97] E. Kantorowitz, '*Algorithm Simplification Through Object Orientation',* Software Practice and Experience, 27(2), (Feb. 1997), 173-183