ACTIVITY-BASED CLASS DESIGN: AN ANALYTICAL METHOD FOR DERIVING OBJECT-ORIENTED CLASSES

Dr. Yousif Mustafa, West Liberty State College, ymustafa@wlsc.edu
Dr. Ayodele Awofala, DaimlerChrysler Corporation, aoa@dcx.com

ABSTRACT

This paper presents a new method to identify classes during object-oriented software development. The approach uses process mapping and clustering technique for achieving unambiguous transition from requirements to the definition of object-oriented classes. We draw analogy between forming cells in cellular manufacturing and class formation in object-oriented software development and accordingly borrow some clustering techniques from cellular manufacturing. By linking the class definition directly to process mapping, our Activity Based Class Design, provides a consistent method for efficiently achieving coherent and agile classes with minimum subjectivity. Thus, the method provides a systematic approach for reducing the complexity and cost of software development.

Keywords: Class Design, Cell formation, Clustering, Cellular, Object

INTRODUCTION

The object-oriented approach to software development holds the promise of significantly improving both the quality and productivity of software development. However, the benefits of object-orientation can be reaped only if objects can be identified in a manner that is both efficient and consistent with easy implementation and a quality software product. Having an appropriate set of objects, for a given application domain, assures high reusability, promotes extensibility and helps ensure the quality and productivity improvements inherent in the object-oriented paradigm (12). In fact, the design of an object-oriented system begins with class design. Finding the right classes is the central challenge of the object-oriented analysis and design (11). According to Booch, identification of classes is the hardest part of the object-oriented analysis and design (2). It still remains one of the most tedious tasks of systems analysis, and the quality of abstraction still remains closely linked to the experience of the modeling specialist (2). The process of identifying the right classes, in most existing approaches, utilizes one or more of the following techniques:

Linguistic Parsing: A concise written description of the problem, with nouns and verbs underlined, is used. Nouns are then identified as candidate classes while verbs represent candidate operations on the classes (2), (8), (7), and (9). Abstract nouns identify ideas or quantities that have no physical existence, are used as attributes for those classes (8).

Problem Requirements Analysis: According to Grady Booch (2), there are typically three sources for deriving classes from problem requirements: (a) tangible things, roles, events, and interactions (Shlaer and Mellor), (b) people, places, things, organizations, concepts, and events (Ross), and (c) devices, events remembered, roles played, locations, and organizational units (Coad and Yourdon). Others, such as (3), propose that candidate classes can be identified by exploring abstractions of things, people, concepts, and roles within the software application domain.
Domain analysis: Booch points out that the analyst should start with abstractions that have proven useful in other related systems (2). Following this approach, (5) presents an approach developed by Coad, Yourdon, and Nicola where the analyst consults with domain experts, reads all related documents, and browses previous designs of interrelated domains.

Use-Case Analysis: Both (2) and (4) also discuss the application of Use-case analysis as proposed by Jacobson, where scenarios or "cases" are developed to describe system functions. By studying these cases through storyboarding techniques the analyst is able to identify the objects that participate in these scenarios.

Behavior Analysis: Booch in (2) presents the behavior analysis approach of Rubin and Goldberg, where system functions are the primary sources of classes. This approach emphasizes the understanding of system behaviors, and assigns behaviors to different parts of the system. Initiators and participators that play significant roles are then recognized as classes.

Despite the availability of the approaches described above, it is necessary to stress the fact that we still have the outstanding problem of finding the right classes in a manner that is most consistent with users' requirements. Booch states that except for the most trivial abstractions, we have never been able to define a class exactly right the first time (2). Unfortunately, much of the current object-oriented class design is still characterized by a hit or miss approach that is often expensive and prone to errors. These approaches depend primarily on personal judgment, intuition, and deep domain insight. Therefore, the major deficiencies of existing approaches are such that:

1. Intuition can not be trusted as a basis for class design when requirements are incomplete, contradictory, and poorly structured.
2. Linguistic approaches are very subjective since many words in English can often be interpreted as both names or nouns (12). Linguistic approaches also suffer from the problem of informality.
3. Domain knowledge is not always available and domain concepts are inherently subjective (12).
4. Class attributes are designed, in most cases, by identifying attributes based on what "seems reasonable" as "real-world characteristics" and not the minimal set of attributes for modeling the domain for the purpose of developing the end user software. So, it is not unusual that more attributes are defined than are necessary, which typically results in “fat” classes.

It is clear that there is a need for a systematic method for identifying the right classes without relying on intuition or special domain expertise, and without excessive trial and error. In this paper we present a new method for systematically deriving the right classes in a single pass so that classes have the following properties:

- "lean" having only the attributes needed for proper functioning of the application software.
- coherent and self-sustained in order to minimize the amount of data flow between classes, which results in a more modular, efficient, and maintainable software.

**ACTIVITY-BASED CLASS DESIGN**

Our method, the Activity-Based Class Design (ABCD), utilizes clustering concepts from cellular manufacturing (10) and the Behavior Analysis approach mentioned in the previous section. Cellular manufacturing, according to Singh, is an approach in manufacturing where a firm’s manufacturing system is divided into cohesive cells. A cell is a cluster of machines or processes
located in a close proximity and dedicated to manufacture a family of specified parts, which are similar in their processing requirements (10). One of the major reasoning behind cellular manufacturing is to create cohesive and self-sustained cells in order to minimize material handling between the cells. The idea of cellular manufacturing is based on the concept of Group Technology (GT), a theory of management based on the principle that similar things should be done similarly (1). A data format, similar to the one shown in Figure 1, (referred to as a machine-part indicator matrix) is the basis of many techniques for cell formation. We would like to identify dense blocks of 1’s in the matrix where adjacent parts use the same machines (1).

In this research we draw analogy between machines, parts, and manufacturing cells on one hand, and activities, data items, and classes on the other hand, then borrow the techniques used in cell formation to derive classes.

In order to emulate cell formation in manufacturing, the ABCD method performs the following steps in order to arrive at the right classes for a given software system:

1. Understand fully the functions of the software system, then generate a process map showing all the processes that the software will perform to accomplish the outcome that the user needs to achieve.
2. Decompose the process map into smaller and smaller unitary activities.
3. Identify the data items for the system by finding the input and output for each activity.
4. Perform clustering by using the Activity/Data Clustering Algorithm described in section 2.1 below.
5. Identify classes by applying the Class Formation Algorithm, described in section 2.2.

An example borrowed from (10) will be used to facilitate the presentation of the algorithm.

2.1. Activity/Data Clustering Algorithm:

Let’s assume that a “to be” software system has been decomposed into 5 unitary processes, which use 10 different data items. In this step, we first develop a two-dimensional matrix where each column represents a data item, and each row represents an activity. The listing of columns and rows is in arbitrary order and no precedence between activities or data items is assumed. A “1” in an intersection cell indicates that an activity uses the corresponding data item, and a blank indicates that the data item is not used by the corresponding activity. A sample Activity/data matrix is shown in Figure 1.

Next, we apply the Binary Ordering Algorithm discussed in (10) and (1) by following these steps:

1. Assign a binary value (2 raised to the power of n-1, n-2, n-3, n-4, etc.) for each column in matrix respectively), where n is the number of data items. Similarly, for each row, assign a binary value (2 raised to the power of m-1, m-2, m-3, etc. respectively), where m is the number of activities in the matrix. In Figure 1 (where n=10 and m=5), column 9 will be assigned the value of $2^{1}$, row 2, on the other hand, will be assigned a binary value of $2^{3}$.
2. Compute the decimal value for each row and column by adding all the decimal values for every cell in that row or column. The decimal value of a single cell is the result of multiplying the number in that cell (1, if exists or 0 when blank) by the binary value for that row or column.
3. Sort the rows in a descending order based on their decimal values.
4. Sort the columns in a descending order based on their decimal values.
5. Repeat steps 2 through 4 until all rows and columns have stabilized and are in a strictly descending order. The final resulting matrix [22] is shown in Figure 2, below.

![Figure 1, A sample Activity/Data matrix](image1.png)

**Figure 1, A sample Activity/Data matrix**

<table>
<thead>
<tr>
<th>Activities</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
<th>D7</th>
<th>D8</th>
<th>D9</th>
<th>D10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>2^4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1007</td>
</tr>
<tr>
<td>A2</td>
<td>2^3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>451</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>2^2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>568</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>2^1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>455</td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>2^0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1020</td>
<td></td>
</tr>
<tr>
<td>Dec. Value</td>
<td>21</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>21</td>
<td>5</td>
<td>21</td>
<td>19</td>
<td>26</td>
<td>26</td>
</tr>
</tbody>
</table>

**Figure 2, The sorted Activity/Data matrix**

<table>
<thead>
<tr>
<th>Activities</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
<th>D7</th>
<th>D8</th>
<th>D9</th>
<th>D10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A5</td>
<td>2^0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1020</td>
</tr>
<tr>
<td>A1</td>
<td>2^4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1019</td>
</tr>
<tr>
<td>A3</td>
<td>2^2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>900</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>2^1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>2^3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>115</td>
<td></td>
</tr>
</tbody>
</table>

**2.2. Class Formation Algorithm:**
The Binary Ordering Algorithm explained in section 2.1 provides a very simple and efficient way to bring chaos to order. Fortunately, the algorithm is highly efficient for taking an arbitrary machine-part matrix and producing a nearly block diagonal structure. Block diagonal structure means that we can partition the matrix in “boxes” or blocks some of which are on the main diagonal, such as the two colored blocks shown in Figure 3 below, while the rest are off-diagonal (1). Diagonal blocks have no overlapping rows or columns, and will eventually become classes as will be demonstrated below. In other words, for a given matrix, if we draw G-1 horizontal lines and G-1 vertical lines (red dotted lines in Figure 3) to partition the matrix, we will obtain GxG total blocks. The number of diagonal blocks will be equal to G (1). Depending on the final Activity/Data matrix, the developer may decide to partition the matrix into 1,2,3,...,N diagonal blocks. Whether the developer decides to partition the matrix in two or three classes, the question is where is the best place to draw the lines and which is the best class formation for that partitioning decision? Is it better or worse to move a line up or down, left or right to include or exclude some cells? Fortunately there exists an algorithm to evaluate the goodness of cell formation using the Grouping Efficacy Goodness Test, which will be detailed in section 2.3. In the worst case scenario the developer may explore all possible combinations for forming two classes (using the two class example of Figure 3) and select the one formation with the highest...
Grouping Efficacy. Assuming that \( d \) and \( a \) are the number of data items and activities in the matrix respectively, partitioning the matrix into \( N = 1 \) blocks simply means that the whole matrix becomes a single class containing all the activities and data items. Partitioning the matrix into \( N = 2 \) diagonal blocks would mean forming two classes, as shown in Figure 3. The first class will contain \( K \) activities and \( I \) data items, while the second contains \((a-K)\) activities and \((d-I)\) data items. In order to exhaust all the different possible combinations of forming two classes, we implement nested loops similar to the following:

For \( I = 1, \ I < d, \ I++ \)

For \( K = 1, \ K < a, \ K++ \)

In the general case for forming \( N \) classes, we require \( N-1 \) embedded loops for the rows and \( N-1 \) embedded loops for the columns in order to enumerate and evaluate all possible activity/data combinations. To summarize the class formation algorithm, we execute the following steps:

1. Depending on the final matrix, partition the matrix into a number of blocks.
2. Enumerate all possible combinations for that partition by conducting an exhaustive search, if needed, and evaluate the goodness of the class formation for each combination using the Grouping Efficacy.
3. Select the cell formation, which has the highest Grouping Efficacy.

2.3. Grouping Efficacy Goodness Test:
The Grouping Efficacy is a quantitative metric that measures the goodness of partitioning the activity/data matrix into \( N \) classes. This concept is also borrowed from cellular manufacturing, where it is used to measure the goodness of partitioning the operations and machines into cells. As suggested by (6), the Grouping Efficacy \( GE \) is computed as follows:

\[
GE = \frac{(1 - E)}{(1 + V)}
\]

Where \( E = \) total number of exceptional cells / total number of operations, and \( V = \) total number of void cells / total number of operations.

Exceptional cells are those cells with 1's, which do not belong to any diagonal block. In manufacturing they represent material handling and the cost associated with it. In software, they indicate data, which must be transferred between classes, thereby imposing an extra burden of data exchange between classes. The total number of operations is the total number of 1’s in the whole matrix, which represents the total number of functions operating on data items.

Void cells, on the other hand, are blank cells within the diagonal blocks. A void cell simply means that certain part (or data item) do not visit some machine in the corresponding cell (6). The ideal partitioning would yield a Grouping Efficacy of 1, where there are no exceptional cells and no void cells. Additionally, Grouping Efficacy cannot be negative and its value is always between zero and 1 since by definition:

\[
0 \leq E \leq 1 \quad \text{and} \quad 0 \leq V \leq 1, \quad \text{therefore} \quad 0 \leq GE \leq 1.
\]

Figure 5 below shows the variation of GE with E and V respectively (6). The figure shows that GE is a monotonically decreasing function with one minimum and one maximum. The Figure shows that any increase in E, which means increase in data transfer between classes or less cohesion will lead to lower GE. Moreover, it shows that a change in E has a greater impact than the change in V, which implies that a higher cell cohesion will always lead to higher GE.

![Figure 5, Variation of GE with E and V](image)

**CONCLUSION**

We believe that the ABCD method will have the following benefits in the area of object-oriented software engineering:

1. Establishing an interesting well-defined systematic method to class design to reduce guesswork, reliance on intuition, or previous domain knowledge.
2. Attribute design is activity-driven which necessarily means that each attribute will be utilized by an activity. No attribute will be included unless it contributes to certain activity, which results in lean classes.

3. Class attributes and methods are designed in one single pass, unlike most existing approaches where attributes and methods are designed in two separate phases in an iterative fashion.

4. The method is built on a 30 years old success story of Group Technology and Cellular Manufacturing and its rigorously validated theoretical foundations. The GT technology was established to minimize cost, material handling and maximize throughput.

5. It does not require specific computer science knowledge and is a good tool for software developers who come from non-computer science discipline.

We view the ABCD as an addition to the existing body of knowledge and another alternative to O.O. class design. The ABCD has an important weakness though, which lies in its initial step of decomposing of the system. System decomposition may be a subjective process and dependant on the analyst point of view of the system. Another weakness is the fact that computational complexity and scalability of the algorithm to large software systems has not been explored in this research. We are planning, however, to study it in our future research, as well as automating the ABCD itself. A real-life case study was part of the original research, however the case was removed due to space limitations. The method was recently automated by one of the authors using a combination of Visual Basic and Excel.

REFERENCES