Measuring Levels of Abstraction in Software Development

Frank Tsui, Abdolrashid Gharaat, Sheryl Duggins, Edward Jung
School of Computing and Software Engineering
Southern Polytechnic State University
Marietta, Georgia, USA

Abstract – In software engineering and development, we are expected to utilize the technique of abstraction. Yet, it is one of the most confounding topics. In this paper we explore the concept of abstraction as applied to software engineering, define and discuss a conceptual metric called levels-of-abstraction, LOA, and show some attributes of LOA.

Keyword: software, abstraction, measurement

I. Introduction

In developing software from requirements we are often faced with a “first step” syndrome of where should one start. Many high level architectural styles and patterns [1] have helped overcome this initial hurdle. However, we are still faced with further analysis to group similar requirements together along functional line or along some data usage line into sub-components. The question that faces many developers during this early stage of analysis is the decision of what should be the appropriate level of abstraction for specification, design and implementation. In this paper we first explore the general notion of abstraction and enhance the concept to include levels-of-abstraction as it applies to software engineering. Next we propose a “conceptual” metric for levels-of-abstraction, LOA, which helps us gauge the amount of abstraction. Finally, we show some interesting attributes of LOA. This is a report on the current status of our research. This research is showing some promise in the area of explaining and formulating guidelines for the amount of abstraction and the depth of abstraction required in performing different software engineering activities.

II. General Concepts Related to Abstractions in Software Engineering

One of the fundamental reasons for engaging in the task of abstraction in software analysis, design and development is to reduce the complexity to a certain level so that the “relevant” aspects of the requirements, design and development may be easily articulated and understood. This starts with the requirements definition through design to actual code implementation. The general relationships of the individual world domain, the abstractions of those domain entities, and the artifacts specifying those abstractions are shown in Figure 1. The bold, vertical arrows represent the intra-transformations occurring within each individual world domain of requirements, design, and implementation. The horizontal arrows represent the inter-transformations occurring across those domains.

The term abstraction used in the form of a verb, as represented with bold vertical arrows in Figure 1 from individual world domain to abstractions, would include the notion of simplification. Simplification represents the concept of categorizing and grouping domain entities into components and relating those components. We simplify: (a) by reduction and (b) by generalization. By reduction, we mean the elimination of the details. Generalization, on the other hand, is the identification and specification of common and important characteristics. Through these two specific subtasks of reduction and generalization we carry out the task of abstraction. Rugaber[6] states that design abstraction is a “unit of design vocabulary that subsumes more detailed information.” This notion of abstraction via simplification is also similar to the notion explicated by Kramer [3,4], Perry [5],
Tsui, Gharaat, Duggins and Jung [7] and Wagner and Deissenboeck[8]. Thus via the process of abstraction, we aim to simplify or decrease the complexity of the domain of software design solution.

At the early stage of software development, requirements represent the needs and wants of the users and customers. Different degrees of abstraction may be employed depending on the amount of details that need to be portrayed in the requirements. The intra-transformation is represented by the vertical arrow from User Needs/Wants domain to Requirements Models of Abstraction in Figure 1.

As we move from requirements towards the development of the solution for the user requirements, a new form of abstraction takes place. The new form of abstraction is necessitated due to the fact that the solution world includes the computing machines and other logical constructs that may not exist in the original user requirements. One of the first artifacts from the solution side is the software architecture and high level design of the software system. The inter-transformation of the requirements models of abstraction to the design models of abstraction is shown as the horizontal arrow in Figure 1. Note that the box labeled “Design Models of Abstraction” is the result of two types of transformations:

a) inter-transformation from the requirements domain and
b) intra-transformation within the design solution domain.

The “Implementation Models of Abstraction” box in Figure 1 represents the packaging/loading models of the processes, information and control of the execution of the solution in the form of source code to satisfy the functionalities and attributes described in the requirements and design documents. Thus it is a result of the inter-transformations from requirements models of abstraction through the design models of abstractions and the intra-transformations from the actual software execution domain. The “Implementation Code” box is the specification of this abstraction. The execution of the Implementation Code and the interactions with the users form the “Executing Software System” box in Figure 1. The employment of various abstractions and the transformations of these abstract entities are crucial to software engineering.

III. Measuring Abstraction

Abstraction, both as a verb and as a noun, is a crucial element in software engineering. As a verb, we have defined it as the activity of simplification, composed of reduction of details and the generalization of crucial and common attributes. Now, it is relevant to ask how much abstraction would be appropriate so that we can arrive at the “Implementation Code” and the “Executing Software System” boxes in Figure 1.

Jackson [2] admonishes us that we need to be careful with abstraction and the degree of abstraction because so many seemingly good designs fall apart at implementation time. His warning is well founded in that many design abstractions, the noun, are often missing some vital information for the detail coding activities. In the past, we have utilized the technique of decomposition to move from abstraction to details. However, if our abstraction is generalizing too much to not include the vital information, then Jackson’s warning will turn into reality. The levels of abstraction should be different for various software artifacts and be dictated by the purpose of abstraction. Wang [9] has expressed a similar concern and defined a Hierarchical Abstraction Model for software engineering; his hierarchical model of abstraction describes the necessary levels of preciseness in representing abstractions of different objects. In terms of our Figure 1, Wang addressed the issue of rigor of specifications of abstraction in the requirement and design documents, not how much should be included in the abstraction.

The how much, or the amount, of abstraction is a reflection of the result of the simplification activity. Measuring the amount of abstraction is gauging the extent of reduction and generalization that took place. For example, this may be possible in the requirements domain. We may consider the set of the original requirements statements of needs and wants as X in the “User Needs/Wants” box in Figure 1. Then, |X|, the cardinality of X is a count of the raw requirement statements collected through some solicitation process. These are the pre-analysis requirements statements. We then designate Y as the statements in the “Requirements Models of Abstraction” box. The cardinality of Y, |Y|, is a count of the statements that resulted from requirements analysis, which include activities such as organizing, grouping, prioritizing, etc. In other words, the post-analysis of the solicited requirements is a form of abstraction of the raw user needs and wants requirements statements. Then the “difference” between |X| and |Y| is:

\[(\text{Level-of-Abstraction})_{\text{REQ.}} = |X| - |Y|\].
(Level-of-Abstraction)\textsubscript{REQ} represents the “difference” between pre-analysis and post analysis of requirements, and it may be considered a “conceptual” metric of abstraction for requirements.

Since simplification is a vital characteristic of abstraction, we expect \(|Y|\) to be less than \(|X|\). Thus we will need to further refine this definition with the constraint that if \(|Y|\) is not less than \(|X|\), then no abstraction activity really took place. We will also take the subscript, \textsubscript{REQ}, off the terminology for the general case. In general, let \(X\) be the statements in the domain world, and let \(Y\) be the set of statements in the abstraction, the noun, then

\[
\text{Level-of-Abstraction (LOA)} = \begin{cases} 
|X| - |Y|, & \text{if } |X| > |Y| \\
0, & \text{else}
\end{cases}
\]

Note that when the abstraction activity is carried to its extreme, \(|Y|\) should just be 1. For example, all the raw requirement statements of needs and wants are abstracted into one abstract statement. Thus Level-of-Abstraction is bounded by \((|X| - 1)\) and 0.

\[\text{IV. Requirements Abstraction Example:}\]

In this section we will further explore the Level-of-Abstraction measurement concept, using requirements analysis as an example. Note that it is very likely that \(X\) and \(Y\) are not expressed with the same language. English sentences and some diagrams may be the main ingredients of the wants and needs expressed by the users and customers. The result of requirements prioritization, categorization and analysis is some form of abstraction, \(Y\), which may be expressed with a Use Case Diagram. The amount of requirements abstraction defined as the “difference” between pre and post analysis of requirements is shown through an example of functionally partitioning the requirements.

Suppose there are \(X = \{x_1, x_2, \cdots, x_z\}\) raw requirement statements. The set \(X\) may contain a variety of statements, referring to functionality, data and other attributes. A common type of abstraction that may be employed is to categorize and group \(X\) by functionality. Thus only a subset of \(X\), \(X'\), is addressed. \(X'\) is the subset of requirement statements that addresses functionality needs. Let \(X' = \{x_1, xx_2, \cdots, xx_n\}\), where \(|X'| \leq |X|\). The subset \(X'\) is analyzed and then partitioned into some set of categories of functionalities. This partitioned set will be called set \(Y\). \(Y\) may look as follows.

\[Y = \{xx_1, xx_2; (xx_3, xx_5, xx_10); \cdots\}\]

Every functionality \(xx_i\) \(X'\) is in one of the partitions of \(Y\) and in only one of the partitions. Renaming the partitioned set \(Y\) as follows would yield:

\[Y = \{y_1, y_2, \cdots, y_k\} \text{ where}
\]

\[
y_1 = (xx_1, xx_2)
\]

\[
y_2 = (xx_3, xx_5, xx_10)
\]

The set \(Y\) may be represented by a Use Case diagram where \(y_1, y_2, \cdots, y_k\) are the named interaction represented as “bubbles” in the Use Case diagram. Clearly there is more than one way to partition \(X'\); thus there may be different \(Y\)'s. A use case diagram with one “bubble” would be an extreme case as well as a use case diagram with a bubble for each \(xx_i\). The extreme points of \(|Y| = 1\) or \(|Y| = |X'|\) would be very rare.

Now consider a specific case where the domain set \(X\) has 4 functional requirements \(x_1, x_2, x_3, x_4\). Then there are the following partitioning sets, \(P\)'s for different functional abstractions, \(Y\)'s.

P0 has \(Y01 = \{(x_1); (x_2); (x_3); (x_4)\} = X\)

P1 has \(Y11 = \{(x_1); (x_2); (x_3,x_4)\}, \text{Y12 = \{(x_1); (x_3); (x_2,x_4)\}}, \)

\(Y13 = \{(x_1); (x_4); (x_2,x_3)\}, \text{Y14 = \{(x_2); (x_3); (x_1,x_4)\}}, \)

\(Y15 = \{(x_2); (x_4); (x_1,x_3)\} \text{ and} \text{ \ \ \ \text{Y16 = \{(x_3); (x_4); (x_1,x_2)\}}\)

P2 has \(Y21 = \{(x_1); (x_2,x_3,x_4)\}, \text{Y22 = \{(x_2); (x_1,x_3,x_4)\}}, \)

\(Y23 = \{(x_3); (x_1,x_2,x_4)\}\text{ and} \text{ \ \ \ \text{Y24 = \{(x_4); (x_1,x_2,x_3)\}}\)

P3 has \(Y31 = \{(x_1,x_2); (x_3,x_4)\}, \text{Y32 = \{(x_1,x_3); (x_2,x_4)\}}, \text{ and} \text{ \ \ \ \text{Y33 = \{(x_1,x_4); (x_2,x_3)\}}\)

P4 has \(Y41 = \{(x_1,x_2,x_3,x_4)\}\)

At P0, there is only one abstraction, Y01, which is the same as the original requirement set \(X\). So Level-Of-Abstraction is \(|X| - |Y| = 0\). There is no abstraction at P0. At P1, any of the \(Y1x\) has a cardinality of 3. So at P1, \(|X| - |Y| = 4 - 3 = 1\). The Level of Abstraction is 1. At P2, the \(Y2x\)'s are grouped differently and each has a cardinality of 2. Thus, at P2, \(|X| - |Y| = 4 - 2 = 2\). P3 partitioning has the functionalities grouped.
VI. SUMMARY AND RESULTS

In this paper we explored the general notion of abstraction as applied to software engineering. We further proposed a “conceptual” metric for levels-of-abstraction, LOA, which allows us to gauge the amount of abstraction. Lastly, we showed some characteristics of LOA. Our current research direction is to analyze LOA further to help us pick the “right” level of LOA for a specific domain.

References