A Uniform Treatment of Reusablility of Software Engineering Assets *

Murali Sitaraman

Software Portability and Reusability Research Group
Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506-6330
Tel: (304)-293-3607, fax: (304)-293-2272
E-mail: murali@cs.wvu.wvnet.edu

Abstract

There has been much recent activity in identifying methods to reuse all software engineering life cycle artifacts, including requirements, design documents, specifications, and implementations [Biggerstaff 89, Prieto-Diaz 90]. While some artifacts are clearly and readily reusable, even the meaning of reuse is not evident for some others. An important source of confusion is the absence of any common framework and terminology to discuss reuse of different artifacts. Is it possible to have a single framework that permits such uniform treatment? We take the position that this question can be answered affirmatively, by extending the "3C reference model" developed in [Tracz 89] and discussed at the Third and Fourth Annual Workshops on Software Reuse [Edwards 90, Latour 90]. The paper also includes a glossary of common software engineering terms as viewed from this extended model.

Keywords: framework, reuse, software engineering assets, terminology

Workshop Goals: Interact; advance theory and practices of software reuse

Working Groups: component certification, design guidelines, education, and reuse terminology standards.

*This research is funded in part by NSF Grant CCR-9204461.

1 Background

Over the past several years, we have been investigating various aspects of software reuse including specification of abstract functionality and performance, formal verification and testing, language design, portability, distributed and real-time computing, and education. We have regularly presented papers and participated in reuse conferences and workshops, including the Annual Workshops on Software Reuse. Our research in software reusability and portability issues at the West Virginia University is currently funded in part by NSF Grant CCR-9204461 and by DARPA.

2 Position

It is possible to provide a uniform treatment of reusability of software engineering assets by extending the 3C reference model.

3 Justification for the Position

The 3C reference model, first explained at the Third Annual Workshop on Software Reuse, has the potential to become accepted as the standard for discourse in the software reuse community [Latour 90, Weide 91]. It defines and distinguishes three important aspects of a reusable component: concept, content, and context. The original purpose of this model apparently was to describe "code" components. In this paper, we show how it can be extended and uniformly applied to all software life cycle assets. The paper includes a glossary of common software engineering terms as viewed from this model.

The paper is organized into the following sections. Section 1 describes the elements of the extended 3C model. Section 2 gives a set of examples to illustrate some important ideas. Section 3 contains a glossary of software engineering terms as viewed from the 3C model.

3.1 An Extended Definition of the 3C Model

First, we define the 3C model. Our definitions (slightly modified from the original sources) apply equally well to all software engineering life cycle artifacts, not just code.

Concept This is a statement of what functionality a software component provides. It may be presented in a formal specification language [Wing 90] or in an informal form such as natural language, or something in between. It is important that a concept does not show any implementation bias. Ideally, a concept should be "abstract."

Content This is a statement of how a component achieves the abstract behavior described in its concept. As with a concept, a content may also be expressed formally (e.g., as in code) or informally (e.g., as in design documents, data structure diagrams, natural language text, etc.). There may be several contents for the same concept.

Context It is usually impossible to explain a concept, or a content, or the relationships between concepts and contents, in absolute terms. Typically, there is an environment in which concepts, contents, and the relationships are discussed. The context is a statement that defines

this environment. Again, it may be formal or informal. A content typically has a different ("bigger") environment than that of its concept. Therefore, we distinguish conceptual and contentual contexts.

We also introduce the need for contexts for *relationships* between concepts and contents in this paper. The original 3C model does not include a context of this kind.

Layering

When reuse efforts succeed, new software systems will be largely built by assembling existing artifacts. The layered nature of components in a software system built from reusable components is illustrated in Figure 1. In the figure, a concept is denoted by a circle, and a content is represented by a rectangle. Arrows denote relationships between concepts and contents. Contexts are shown as shaded ovals enclosing concepts, contents, or relationships. Contexts can be visualized to exist at an orthogonal dimension to the plane of the figure.

In Figure 1, an arrow from a concept/abstraction A1 to a content/realization R11 denotes that R11 implements A1. There may be several contents for realizing the same concept, each probably satisfying some different constraints (e.g., memory or real-time bounds). The same content may also realize multiple concepts, at least in principle.

An arrow from a content (e.g., R11) to a concept (e.g., A2) denotes that R11 uses A2. A content, of course, may use more than one concept.

The description of a concept usually refers to other external entities. These entities form its context. The environment in which a content exists forms its context. A context, for example, may include formal mathematical theories, citations, operating systems, or hardware architectures. The context of an arrow from a concept A1 to a content R11 specifies the conditions under which R11 realizes A1. The context of an arrow from a content R11 to a concept A2 specifies the conditions that must be satisfied by a content of A2 in order for it to be used by R11.

It is important to emphasize that the elements of Figure 1 need not all be executable. Some may be executables and some others may be documents in English or some other (possibly more formal) language.

What can be reused?

In this setting, what can be reused and how? A concept is reused in two ways. A concept is reused when more than one content is designed for the same concept. For example, contents R11 and R12 reuse the concept A1 in the figure. Another way in which a concept is used is when another content uses this concept. In this case, the concept as well as one or more of its contents are reused. For example, contents R12 and R21 use the same concept A3 in the figure.

A content can be reused only through its concept. The same content may be associated with multiple uses of a concept and thus the content will be reused.

A concept or content may also be used in multiple contexts; however, some portability issues will have to addressed to achieve this goal.

When the same context applies to multiple concepts, contents, or concept-content relations, then that context is reused. In the figure, the same context has been placed on the relationships between the Client and A1, R11 and A2, and R11 and A3. Contexts are also reused whenever a concept or content is reused. As noted earlier, the context of a concept may refer to other concepts and the

context of a content may refer to other concepts and contents. In this case, concepts and contents will be reused when these contexts are reused. In general, reuse is possible directly or indirectly (i.e., reuse of some entity M leads to reuse of all entities reused by M).

It is important to note that the framework makes no assumptions on how the various pieces came into existence. A concept, for example, may have been created starting from scratch or may have been constructed by inheriting several other concepts. The same is the case for contexts. A content may have been generated mechanically from a concept or may have been created manually. The techniques for building concepts and contents are important. However, they do not influence the structures presented in this paper.

3.2 Examples

In this section, we discuss a series of examples to illustrate various possible interpretations of the extended 3C model. The discussion here is intentionally left at a high level. Related technical details can be found in [Sitaraman 92a, Sitaraman 92b, Weide 91].

In Figure 2, Stack_Template is an Ada package specification (possibly annotated with formal specifications). List_Based is a body that must satisfy the constraint that all stack operations execute in constant time. This constraint is the context on the relationship between Stack_Template and the body named List_Based. This constraint in turn places constraints on the components chosen for implementing the List_Based body. Time constraints, of course, may be expressed more formally and more precisely.

In Figure 3, the concept "Sort" stands for a formally or informally specified idea of sorting. Constraints are not restricted to time or space, as can be seen from this example.

Figure 4 illustrates how different life cycle artifacts can be presented in the same framework. Some concepts (including the concept at the top-most level) found in this figure may have been realized previously, in which case those concepts and their realizations can be reused. Others may have to be implemented from scratch.

3.3 Glossary

This section presents a glossary of some common software engineering terms from the perspective of the extended 3C model.

Requirements definition document Informal definition of a concept under a specific context.

Formal specification A formal statement of a concept.

Correctness of a concept A concept is correct if it matches the intuition of the people involved in creating the document that describes the concept. Clearly, there can be no mechanical procedure to ensure that a concept is correct. However, a concept can be "inspected" to ensure that it indeed captures the intended idea and that it is consistent and sufficiently complete. Prototyping is another method of validating that a concept meets its concept.

Design document Informal (or semi-formal) explanation of how a content satisfies a given concept under a specific context.

Code component A content presented in a (formal) programming language.

Correctness of a content A content is behaviorally correct if it satisfies the corresponding concept. A content is correct with respect to its constraints (i.e., the context on the relationship between the content and its concept) if it satisfies the corresponding constraints. Performance (e.g., execution time) is an important type of constraint. Real-time components must, for example, satisfy time-related performance constraints.

Goal of inspections, walkthroughs, formal verification and testing To ensure that a given content is correct (with respect to its concept); typically, to ensure a design document is correct, inspections and walkthroughs are useful. To ensure code components are correct, verification and testing may be appropriate. While black box testing relies on the concept to generate test cases, white box testing is based on the content.

Goal of portability To move a concept or content from one context to another.

Goal of automatic programming To mechanically generate a content from a given concept (and constraints). Such a concept is often called an executable specification.

A goal of domain analysis Identification of basic concepts in a domain that will form the basis for development of most contents in that domain.

4 References

[Biggerstaff 89] T. Biggerstaff and A. J. Perlis, *Software Reusability*, Volumes 1 and 2, Addison-Wesley, 1989.

[Edwards 90] Edwards, S., "The 3C Model of Reusable Software Components," *Third Annual Workshop: Methods and Tools for Reuse*, Syracuse, 1990.

[Latour 90] Latour, L., Wheeler, T., and Frakes, W., "Descriptive and Predictive Aspects of the 3C Model: SETA1 Working Group Summary," *Third Annual Workshop: Methods and Tools for Reuse*, Syracuse, 1990.

[Prieto-Diaz 90] Prieto-Diaz, R., "Domain Analysis: An Introduction," *ACM SIGSOFT 15*, 2, April 1990, 47-54.

[Sitaraman 92a] Sitaraman, M., "A Class of Programming Language Mechanisms to Facilitate Multiple Implementations of the Same Specification," *IEEE Computer Society 1992 International Conference on Computer Languages*, Oakland, CA, 1992, 172-181.

[Sitaraman 92b] Sitaraman, M., "Performance-Parameterized Reusable Software Components," *International Journal of Software Engineering and Knowledge Engineering 2*, 4, World Scientific, 1992.

[Tracz 89] Tracz, W. J., and Edwards, S., "Implementation Working Group Report," *Reuse in Practice Workshop*, Pittsburgh, PA, 1989.

[Weide 91] B. W. Weide, W. Ogden and S. H. Zweben, "Reusable Software Components," *Advances in Computers*, M. C. Yovits, ed. Academic Press, New York, NY, 1991.

[Wing 90] J. M. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer 23*, 9, September 1990, 8-24.

5 Biography

Sitaraman is an assistant professor in computer science at the West Virginia University. He has a Ph.D. from The Ohio State Univresity (1990). His research focuses on various aspects of software reuse including specification of abstract functionality and performance, formal verification and testing, language design, portability, distributed and real-time computing, and education. area of research centers around verification and validation of reusable software components/systems. Sitaraman's research is currently funded by a 3-year NSF grant and a 2-year DAAPA grant. Sitaraman has authored several technical papers on related topics in software engineering. He is a member of the ACM and IEEE Computer Society.