On a Taxonomy for Software Components

Johannes Sametinger
Brown University
Department of Computer Science, Box 1910
Providence, RI 02912, USA
Email: js@cs.brown.edu

Abstract

Object-oriented programming has been an important step forward in increasing the quality of software systems and the productivity of software engineers. Objects have brought a radical change in the way software systems are being built. Objects can be regarded as components in that they facilitate to build software systems by putting various classes (objects) together. In order to facilitate component-oriented software engineering we must loosen the dependencies among components, attain higher abstraction levels, achieve vendor/platform neutrality, and facilitate the composition of different kinds of components.

Components that are integrated into a system do not run on their own. Functions and classes (objects) belong to this category. Document-oriented computing as supported by OPENDOC and OLE provides components of higher abstraction than classes and objects. Furthermore, the advances in distributed computing suggest the use of components that run on their own, have less coupling and more autonomy.

Component composition has proofed to be successful when a particular platform, i.e., operating system, run-time system, standard, has been propagated and used, e.g., UNIX pipes and filters, VISUALBASIC, CORBA. Yet, we have not defined a uniform model for component composition and there are many issues to be addressed.

Keywords: software components, software composition, component taxonomy, open systems, software reuse

Topics of Interest: anything related to the interoperability of components, discussion/definition of component/composition taxonomies

1 Introduction

The ideal scenario of software engineering is to build applications by putting high-level components together. If any required components are not available, they have to be built out of lower-level components. Finally, when even low-level components are not available, they eventually have to be implemented in a certain programming language.

Software components are prefabricated, pre-tested, self-contained, and reusable software modules. They bundle data and procedures that perform specific functions. Advantages of component software are distribution, reusability, economy, (user) modifiability, extensibility, and vendor neutrality [5].

Component-oriented software development means, that at the methodological level we design and develop software systems in a compositional way, i.e., we create a set of components that are supposed to work together in some way. The components are not designed in isolation but rather are meant to collaborate. From this point of view components can be macros, functions, modules, classes, templates, etc. Technically speaking, component-oriented software development is the integration of computational and compositional aspects of software development. Nierstrasz and Dami define software components as "static abstractions with plugs" [8].

2 Software Composition

Constructing software systems from software components is called software composition. Composable software has a higher degree of flexibility than monolithic software. Raising the level of abstraction helps us in dealing with increasing complexity. Today many examples of software components exist at various abstraction levels.

Different languages and environments realize software composition to different degrees. They support different notions of components and compositions. Component-oriented software development requires that we have a selection of reusable components that are plug-compatible. The higher the granularity of the components is, the higher the increase in software productivity can be. Putting objects or applications together is more effective and productive than putting functions together.

It is easier to recompose software in order to meet new requirements instead of modifying a monolithic creation. Examples of successful application of software composition exist in certain domains like user interfaces, application frameworks, programming environments, and fourth-generation languages. But a general model of software composition does not yet exist [7].

3 Component Reuse

Component-oriented software engineering has many benefits for software reuse. Components can be stored in repositories and, if properly classified, considerably increase the productivity of software engineers. Unfortunately, most of today's components can hardly be combined with each other. Getting information about a component's functionality is not sufficient. We have to define the typical characteristics of software components and their composition in order to find and select them for the reuse in various contexts. Example questions about component candidates for reuse are:

- What is the component's functionality?
- What kind of component is it?
- Can the component be interconnected with our components?
- Can the component be (re)used in our context? How?
- Can the component be customized? How? To what extent?
- What else is needed to reuse the component?
- Is the component's quality sufficient for our purposes?

In order to systematically exploit the reuse of software components we have to classify components and compositions, adhere to open standards, and raise the abstraction level from source code components to higher levels of abstraction.

Open Systems

Vendor transparent environments in which users can mix and match hardware and software from various vendors on networks from different vendors are called *open systems*. We eventually want to combine software components from different vendors on hardware *and* software from various vendors interoperating over networks from various vendors. Interoperability, portability, integration and standards are crucial for open systems as well as the success of component-oriented software [9, 10].

4 Component Taxonomies

Several attempts have been made to classify software components. To give an overview we will shortly sketch several of these.

Booch made a division into three major groups of abstractions, i.e., structures, tools, and subsystems [1]. Structures are components that denote objects or classes of objects (abstract data type). Tools are components that denote algorithmic abstractions targeted to structures. Finally, subsystems are components that denote logical collections of cooperating structures and tools. Additionally, forms have been introduced to classify time and space requirements.

Wegner provides a classification of software components of different languages by using state, inheritance, concurrency, and distribution as discriminating characteristics [11]. This yields to the following components: functions and subprograms, packages and modules, classes with single inheritance, classes with multiple inheritance, concurrent tasks with shared memory, distributed concurrent processes, and distributed sequential processes.

Dusink and van Katwijk distinguish between active and passive components [3]. Functions in libraries are considered passive components to be used as building block in a system. Executable programs are active components that require some kind of interprocess communication for composition. Care must be taken not to confuse active and passive components with *pro-active* and *re-active* components. Pro-active components become active on their own (e.g., timer components), whereas re-active components react only in response to certain events (e.g., when activated by another component).

In [6] a taxonomy of interface types of reusable software components is given. The following interface types are contributed: subprogram call, task invocation, memory sharing with subprogram or task, communication via shared file with or without simultaneous access, and communication via message passing or mailbox mechanism.

Bradford Kain uses the characteristics scope, purpose, granularity, and level of abstraction to categorize components [2]. Fernandez provides a taxonomy of coordination mechanisms in real-time software [4].

Many taxonomies concentrate on source code and do not take higher levels of abstractions into consideration. Programming languages provide the most common form of building reusable software components. Other means are the use of visual programming languages, command languages, module interconnection languages, interface definition languages, etc.

5 Workshop Goals

We are still far from the ideal scenario of composing our software systems of existing components. Many questions are still unanswered, and much research has to be done especially on the interoperability of components. But even if we somehow reach this goal or come near to it, of course, we won't get rid of all our problems. It remains to be seen whether we raise the abstraction levels of components and at the same time get high-performance systems by using these components. Performance is one of the key hindrances of clean software composition. Whenever abstraction levels have been raising lack of performance has yielded as well, and efforts have been made to circumvent these abstractions.

Evaluating the current situation of component-oriented programming by classifying existing components and composition techniques is suggested for the workshop. Based on such a classification future research efforts should be derived and discussed. Example topics of further discussion include (multiple) interface definitions for components, integration of legacy components, standards for component composition, the role of document-oriented component models, influences on the software engineering life-cycle and on software design models, etc.

References

[1] Booch G.: *Software Components with Ada: Structures, Tools, and Subsystems*, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

[2] Bradford Kain J.: "Components: The Basics: Enabling an Application or System to be the Sum of its Parts", *Object Magazine*, Vol. 6, No. 2, pp. 64–69, April 1996.

[3] Dusink E.M., van Katwijk J.: "Reflections on Reusable Software and Software Components", in Tafvelin Sven (Ed.): *Ada Components: Libraries and Tools*, Proceedings of the Ada-Europe International Conference, Stockholm, Cambridge University Press, pp. 113–126, 1987.

[4] Fernandez J.: "A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis", *Technical Report*

CMU/SEI-93-TR-34, Software Engineering Institute, Carnegie Mellon University, December 1993.

[5] Meta Group: "Component Software", Meta Group, Inc., White Paper, December 5, 1994.

[6] NATO: *Nato Standard for Management of a Reusable Software Component Library*, Vol. 2 (of 3 Documents), Nato Communications and Information Systems Agency, 19??.

[7] Nierstrasz Oscar, Meijler Theo Dirk: "Research Directions in Software Composition", *ACM Computing Surveys*, Vol. 27, No. 2, pp. 262–264, June 1995.

[8] Nierstrasz Oscar, Dami Laurent: "Component-Oriented Software Technology", in Nierstrasz Oscar, Tsichritzis Dennis (Eds.): *Object-Oriented Software Composition*, Prentice Hall International (UK), pp. 3–28, December 1995.

[9] Quarterman John S., Wilhelm Susanne: *UNIX, POSIX, and Open Systems*, Addison-Wesley, 1993.

[10] Umar Amjad: *Distributed Computing: A Practical Synthesis*, Prentice Hall, 1993.

[11] Wegner Peter: "Capital-Intensive Software Technology", in Biggerstaff Ted J., Perlis Alan J.: *Software Reusability, Vol. I: Concepts and Models*, ACM Press, 1989.