Classification of Composition and Interoperation

Johannes Sametinger

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

Johannes Kepler University
CD Laboratory for Software Engineering
A-4040 Linz, Austria
sam@swe.uni-linz.ac.at

Poster Presentation at OOPSLA '96
San Jose, CA, October 1996

1 Introduction

Component-oriented software development is the design and development of software systems in a compositional way, i.e., the creation of a set of components that are supposed to work together in some way. Components are not designed in isolation but rather are meant to collaborate. Technically speaking, component-oriented software development is the integration of computational and compositional aspects of software development.

The scenario of compositional software reuse 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.

2 Software Reuse

Software reuse is the process of creating software systems from existing software rather than building them from scratch. Reusable software has many benefits, for examples see [2, 3].

Object-oriented programming has many benefits for software reuse. Objects, i.e., classes, can be stored in repositories and, if properly classified, considerably increase the productivity of software engineers. Unfortunately, many of today's objects can hardly be combined with each other. Getting information about an object's functionality is not sufficient to determine its reuse value in a certain context. We have to know characteristics of objects and their kind of interoperation in order to find and select them for reuse.

3 Software Components

In this context we regard objects as reusable software components. However, any kind of component like functions, tools, applications could be considered. Several attempts have been made to classify software components. For example, 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.

Wegner provides a classification of software components of different languages by using state, inheritance, concurrency and distribution as discriminating characteristics [5]. 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.

Programming languages provide the most common form of building reusable software components. Other means are, for example, the use of visual programming languages or filters (and pipes) as used with the UNIX operating system.

4 Software Composition

Constructing software systems from software components is called software composition. Composable software has a higher degree of flexibility and reusability than monolithic software.

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.

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 [4].

Depending on a component's interface that is used for reuse we suggest a classification of composition into three categories: *program interfaces*, *user interfaces*, and *data interfaces*. UNIX pipes and filters provide an example for components using data interfaces. Wrappers like pseudo ttys can be used to reuse components with command-line user interfaces. Program interfaces are the most common and flexible means for reuse. Interfaces range from pure textual to object models (like CORBA, OPENDOC, and OLE) and open platforms, which remain yet to be specified.

5 Software Interoperation

If two components interoperate we have a *sending* component (initiating the interoperation) and a *receiving* component. The sending component activates the receiving component; it gives *control* to this component. The receiving component *reacts* to the control input; it performs some action and, depending on synchronous or asynchronous communication, returns control to the sending component. Some amount of information is usually passed along with interoperation. If a more extensive data exchange is needed, components may use another component for that purpose.

The receiving component may or may not be known to the sending component. This has a major influence on the flexibility of compositions. We denote this with *static* and *dynamic* interoperability. Interconnections can be between two components (*peer-to-peer*), to a fixed set of components (*multicast*), and to a dynamic set of components (*broadcast*). Static interconnections are peer-to-peer. Dynamic interconnections can be either peer-to-peer, multicast, or broadcast. The data component also may or may not be known to both the sender and receiver of interoperation.

For software reuse it is essential that components can be composed without having to know each other. This allows component composition without modifying components. For example, a function calls a, let's say, *sort* function. In order to call a function *shellsort* instead, the program text in the calling function has to be modified. Object-oriented programming provides flexibility through dynamic binding. A calling object does not know the receiver of a call. This makes this object work with a variety of other objects without being modified.

Component composition is easiest and most flexible when interconnections among components are not *point-to-point*. Reusing components is easy in environments where each component can react to events generated by any other components and create new events without being aware of any recipients.

6 Posters

The posters show the proposed categories for composition and interoperation. Composition categories are based on interfaces, i.e., program, user, and data interfaces. Categories of interoperation are based on control and data integration.

References

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

[2] Braun Christine: "Reuse," in Marciniak John J. (Editor-in-Chief): *Encyclopedia of Software Engineering, Vol. 1*, John Wiley & Sons, pp. 1055–1069, 1994.

[3] Krueger Charles W.: "Software reuse," *ACM Computing Surveys*, Vol. 24, pp. 131–83, June 1992.

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

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