Software components in software development

Dr. Harri Laine

Department of Computer Science, University of Helsinki

P.O. Box 26, 00014 University of Helsinki, Finland

E-mail: laine@cs.helsinki.fi

Abstract

Reuse of software has been proposed as a means to reduce the costs of software production and as a means to fulfill the ever–growing need for new software. This paper discusses binary software components as elements of reusability. Requirements for good software components are discussed. How to design and interconnect components is also discussed. Finding out suitable components relies on the metadata that is used in describing components. Requirements, structure and transfer of the metadata are examined.

1 Introduction

Reuse of software has been a goal in software engineering for years. It is claimed to lead into higher productivity and even better systems. However reuse of software is not as common as might be expected [4]. There are many reasons for this situation. One of the main reasons is the inability to find proper pieces of software. If suitable pieces are found they may be hard to adopt and adjust into the current case. There are many reasons for this situation: design methods that provide inadequate support for reuse, attitudes of programmers, insufficient and hard to get information about available components. Also the components that are available might not be as reusable as they were intended to be.

In the following chapters we discuss about binary software components, how they can be reused and the prerequisites for their reuse.

2 Reusing available software elements

The traditional form of software reuse is the copy–paste technique. A piece of code is copied out of one program and then pasted in another program possibly with minor modifications. Typically this type of reuse is mainly restricted to the programmer's own code, but sometimes also the software archives of the company are used this way. The programmer knows that he has done something similar before and tries to utilize this work with minimal effort. This type of reuse causes maintainability problems and relies heavily on the memory of the programmer.

Function libraries with or without source code are a more advanced form of reuse. The problem with these libraries is that the traditional way of design does not easily lead to the functions that are provided in the library. Another problem is that, if one wants to make adjustments on the functions, the source code must be available.

Reuse of object classes and frameworks makes the adoption and adjustment of code possible without source code. Inheritance and method overriding are the main techniques in reusing classes. Source code is not usually needed. However the classes are programming language dependent.

Binary software components are pieces of software that are supposed to be used as they are without program modifications. Ideally they should be platform and programming language independent. The behaviour of the components can usually be adjusted using predefined means build within the components.

3 Design process

Developing of an information system starts by the specification of the requirements for the new system. These requirements are problem oriented. This problem oriented specification is then transformed into a technical solution. In traditional top down design the functional requirements for the system are stepwise refined to finally obtain the software modules [7]. The resulting modules depend on the original problem and on how the designer performs the refinement. The result is thus a problem and designer specific decomposition of functionality. Even the same programmer at different times may produce different decompositions for the same

problem. Thus it is hard to find proper components for reuse.

The same applies at least partially to traditional object oriented design. Object oriented design methods [9], [3] propose that we should first find out the essential classes to model the 'reality'. Then we should add functionality to these classes and complete our design with technical classes. This also results into problem specific kernel classes and problem specific functionalities for these classes. They might be reusable within the same application domain or in other systems within the same company. Problem specific methods cannot be used as they are in other systems. Thus new methods must be included in the subclasses of the original class. A consequence might be a complex dependency tree of classes and their subclasses, that provide methods with partial or even full overlapping of functionality. Picking up a proper class to reuse becomes difficult.

On the other hand, it is actually the technical classes like buttons, windows, text fields, data stores, etc, that are usually provided for reuse. The way to separate the application specific functionality and the functionality provided by these reusable components is then the key issue.

4 What are binary software components

Binary software components are defined as modular, reusable, atomic and compiled software units that enable cross–language and cross–platform application development. Components with similar specifications should be interchangeable and independently upgradable [5]. Thus, we should be able to use the same binary components in a Visual Basic program in MS–Windows environment and in C++ program in UNIX environment. How can this be accomplished? The same binary code does not work in all current platforms. Thus, the components should either have multiple platform dependent binary representations to select from, or the binary representations should not be 'real' ones, as is the case in Java bytecode.

Currently there are two main component architectures [10] the Microsoft's COM architecture (ActiveX) [6] and Sun's Java Beans architecture [1]. COM architecture strives for cross–language development within Microsoft Windows environments. Java Beans are intended for cross–platform development using Java programming language. COM components are compiled into real binary code whereas Java Beans are compiled

to higher level Java bytecode. Ideal components according to the definition do not exist.

Components are objects and like objects they provide their services through their public interface. Component's public interface may introduce properties, methods and events. Properties expose component's public attributes. These attributes cannot be referred directly. Instead there must be accessor methods available for querying and setting the values of these attributes.

Public methods provide the services of the component. Methods are intended to be called by the program that uses the component. This program could be called the container of the component.

Events are provided to make it possible for the component user to react on the internal conditions within the component. Components use events also to inform the environment about some external stimuli the component is exposed on, for example about mouse clicks on the component's screen image. Events can be considered as signals from the component to the environment. If the environment wants to do something related to the signaled condition, it must provide an event handler for the situation. This event handler contains the code that will be executed in reaction to the event. Events are thus a means to separate the detection of a condition, which is carried out by the component, and the reaction to the condition, which is carried out by an external software.

In practice, events are usually implemented as automatic calls for the handler functions. The signatures for event handlers are specified in the component's interface. If the programming language provides function pointers, the calls for handlers can be implemented using pointers to the handler functions stored within each instance of the component. In languages like Java, that do not support function pointers, a different way to activate the handlers must be used [1]. Anyhow, the idea is to have component instance specific event handlers, not only type specific methods. Typically some kind of registration is needed to connect the handler to the event (Figure 1).

5 Design issues

The guidelines applying to the design of objects and classes apply also to the design of components. To obtain good reusability component should

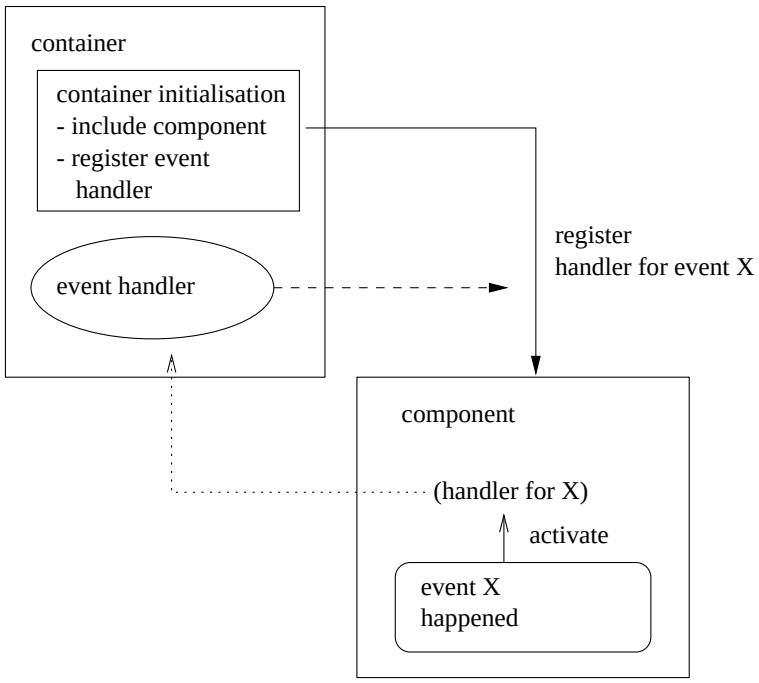

Figure 1. Use of event handlers

be simple and easy to comprehend. Reusing the guidelines for module decomposition [7] we may conclude that a component should have only one clearly specified responsibility.

A component should be designed so that it is not dependent on any object listening to its events and reacting on them in some particular fashion. This makes it possible to reuse the same component in situations, where similar events are produced, but the behaviour should be different. For each event the component user should be aware, if there is supposed to be only one event handler for the event, or can there be many handlers. If there can be many handlers, the order of their activation should be specified. It should also be clearly specified, how the component itself

reacts on the condition that caused the event and how this reaction is carried out with respect to the event handlers.

In component design events should be defined for all cases that might require attention of the container. This results to many events but provides good reusability. Use of events and event handlers is a fundamental architectural level design decision in software development. If we want to use components in our software we must already in the early design take into account that components use event based protocols in communicating with their containers. If we have not taken this into account, introducing it in a late phase of design may cause major iterations of the design and re–positioning of functionality.

Components should be general enough to be usable in many situations. This indicates that they should provide means for customization. Customization usually takes place during program development possibly using a dedicated customization software. In customization the programmer assigns values for the attributes of the component instance. He may also make connections between components. When we design components we must also design how it can be customized. For complex components the design of customization software might be needed.

6 Containers and Frameworks

Components exist and operate within containers. These containers provide the environment for the components to work in. They also provide the means for the co–operation of the components. A container can also be a component.

Frameworks are collections of classes that provide skeletons for applications. Frameworks usually implement some design patterns, enriched with application type specific services. Thus we could have for example drawing software frameworks, case tool frameworks, order processing frameworks and bank accounting frameworks. Also general window based application frameworks are available. Frameworks suit well as containers for components. As such their task is to provide the connectors to plug–in the components. An in–build customization facility is also needed for connecting the components to the connectors. In constructing container components one must decide which member component events are trapped within the container component and which are delivered outside. Best

adaptability is obtained if all the member events are passed through the container. A side effect is increased complexity, and complexity is the main hindrance for reuse. Thus, for simplicity it is better to block all of the member component events within the container and if necessary cause new higher level container specific events instead to inform the outside world.

All components should be designed in connection to some framework. This binds the components to an architectural design, shows how they are used and how they interoperate with other components.

7 Metadata and tools

Locating and using components relies on metadata about the components. There are different types of metadata: technical, design oriented and business oriented.

We call the metadata needed run–time and during compilation as technical metadata. They include type identifier and type descriptor, description of the interface and signatures for methods. Components themselves or the component environment should provide facilities for querying technical metadata at run–time.

Design oriented metadata is used in programming and in design of the software. They include all the information intended for designers in order to be able to include the components in their systems. This metadata should contain

- a descriptive name for the component,
- a general description of the purpose of the component,
- detailed descriptions of methods and properties,
- detailed descriptions of events and their triggering,
- instructions how to customize the component,
- description of how the component is related to other components,
- working environment,
- keywords describing the component,

- classification of component into some predefined categories,
- examples of the use and
- bug reports.

Business oriented metadata contains information concerning the acquiring, distribution and management of the components. Information about license conditions, availability and costs should also be included as well as references, recommendations and comments on the use.

Metadata about components should be stored in a repository. A company may build components for itself, but in addition it usually acquires them from various sources. Components should thus accompany with metadata in a format that can be easily loaded in the repository. This presupposes a standard for representing and transferring metadata. Such standards have been developed, for example the CDIF for CASE–data transfer [2] and UML for representing object oriented designs [8]. However, component related metadata in such a format is not commonly available. Companies store information about components in their own repositories. There is however also demand for global component repositories, kind of component catalogues, that collect information about available components. The information contents of such a catalogue is outlined in Figure 2. Local repositories might have the same information contents.

Designers use metadata about components in designing the software. In addition, computer aided design tools (CASE tools) and programming environments should be able to use the metadata. Currently there are programming environments, for example Delphi, JBuilder and Visual Basic, for including components in the software. A programming environment that supports components should provide

- palettes to displaying the available components,
- containers to host the components,
- facilities (drag and drop) to insert components in containers,
- facilities for connecting event handlers to the components,
- property and script editors for customizing the components,
- tools for constructing new components,

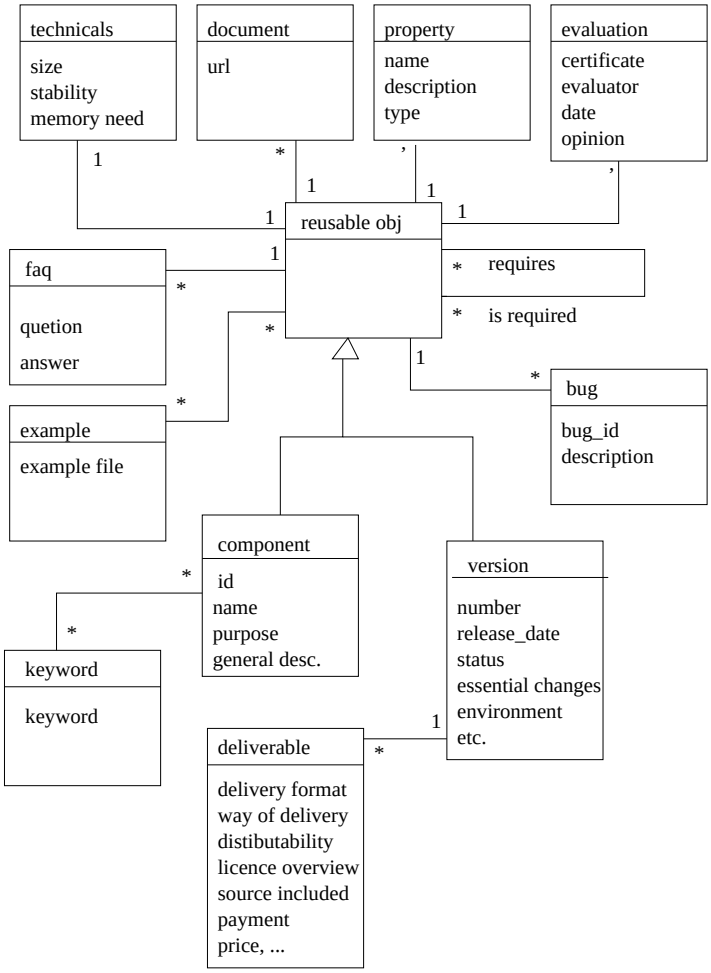

Figure 2. Outline class diagram for component metadata

- component repository and browser to locate components by matching user search criteria.

The customizing facilities may be included in the environment. In Java Beans architecture each component may have its own customizing tools.

When designing the software we should also be able to include components, design patterns and frameworks in our design with similar ease the modern programming environments make it possible to include components in program code. CASE tools to enable this are not however commonly available. These tools should support the use of components, frameworks, design patterns and architecture models on a higher abstraction level than what they are supported in a programming environment. The main interest on design time is to find out the proper components and to specify how these components will co–operate within the system. An essential part of component oriented design software is thus an advanced search facility.

8 Conclusion

Reuse of software is one of the factors in order to increase the productivity in software engineering. Properly designed software components promote reuse. Components must however be easy to locate and users must be able to easily include them in their designs. This presupposed that standardized metadata about components and facilities to utilize this metadata both within a company and also globally are available.

References

[1] *Englander R.* Developing Java Beans, O'Reilly, 1997.

[2] *Ernst J.* Introduction to CDIF July 1998, Online version, continuously updated: *http://www.cdif.org/intro.html* .

[3] *Jacobson I. et al* Object–oriented software engineering, a use case driven approach, ACM Press/Addison–Wesley, 1992

[4] *Jacobson I., Griss M. and Jonsson P.* Making the reuse business work, Computer, 30, 10, October 1997, pp. 36–42.

[5] *Krieger D. and Adler R. M.* The Emergence of Distributed Component Platforms. Computer 31 (3): 43–53 (1998)

[6] *Microsoft Corp.* DCOM Technical Overview, Microsoft Corp., 1996.

[7] *Pressman R.* Software engineering, a practioneer's approach, 4th ed., McGraw–Hill, 1997

[8] *Rational Software* UML home page, 1998, Online version: *http://www.rational.com/uml/*

[9] *Rumbaugh J. et al* Object–oriented modeling and design, Prentice–Hall, 1991.

[10] *Spitzer T.* Component architectures, *DBMS Magazine*, September 1997, Online version: *http://www.dbmsmag.com/9709d13.html*