A Model for Flexible Communication Between Objects

Sten Minör

LU-CS-TR:88-35

Also published in *Proceedings of the 16th Simula User's Conference*, 1988

Department of Computer Science
Lund University

P.O. Box 118
S-221 00 Lund
Sweden

A Model for Flexible Communication Between Objects

Sten Minör

Department of Computer Science
Lund Institute of Technology
Lund University
Lund, Sweden

Abstract

This paper describes an approach for communication between objects in object-oriented languages. The goal is to separate the functionality and communication of objects in order to obtain systems that are extensible, flexible, and easy to maintain.

An example is given in order to identify the problems that motivate the model. The model is presented together with a description of how it can be realized in most object-oriented languages. The properties of the model is discussed and some experience from using it in a large application is given. The approach is also compared with other models designed to solve the same problem.

1. Introduction

Many software systems are at the top level described as a number of communicating components with different functionality. The various components take care of specific subtasks on request from other components. The collection of components together with their current interconnections, i.e. the references between components and their communication paths, form the current system configuration. In order to facilitate code reuse it is of great importance that the components are generic and that they may function in a number of different configurations. In other words a component must be able to work in a variety of environments.

In an object-oriented system the components typically are implemented as objects, described by classes, which communicate via messages or remote procedure calls. The class concept is known to facilitate code reuse and extensibility as a result of its data abstraction and inheritance mechanisms.

Despite the possibility to specialize the properties of a class in order to tailor it for a specific purpose some problems arise. One problem is the communication between components. The nature of communication in object-oriented languages, such as Simula [SIS 87] and Smalltalk [Goldberg 83], is imperative. One component requests another to perform certain actions. Another problem is the structure of references between components, i.e. the current component configuration. References from one component to others are an integral part of the component. The other components existence is a requisite for the component and thus for the system as a whole. In order to be able to build dynamic and flexible, "open-ended", systems where different configurations may be built from generic components these problems have to be tackled.

The key subject is to separate the functionality and the communication of components. The functionality describes the internal behavior of the component. It describes only the task the component is designed for and must only have knowledge of other components that are absolutely required for its function and components that not will be subject for extensive modifications or removal. Communication and references to components that may be subject for removal or modifications should thus not be part of the functionality of a component.

Some successful attempts in language design has been done in order to separate

functionality and communication. In Modula-2 [Wirth 82] and Ada [DoD 83] a component (module or package) consists of one definition and one implementation part. The description part defines the protocol of a component. It describes how other components access the component. The implementation part describes the internal functionality, i.e. how the protocol is realized in the component. These languages has separate parts for the input protocol and functionality. We claim in this paper that a separate part for the output protocol, i.e. how a component calls other components, is needed in order to obtain a flexible system that is straightforward to reconfigure.

In section 3 we will suggest a model for how this division may be done in an object-oriented language. Prior to that, in section 2, an example is given illustrating the needs for such a model. In section 4 some experiences of the model are given and some related models are discussed. Finally a summary and conclusions are given in section 5.

2. An example

We will here give an example which illustrates the problems described in the previous section. Suppose that we have a system for manipulation of graphical images as depicted in figure 1.

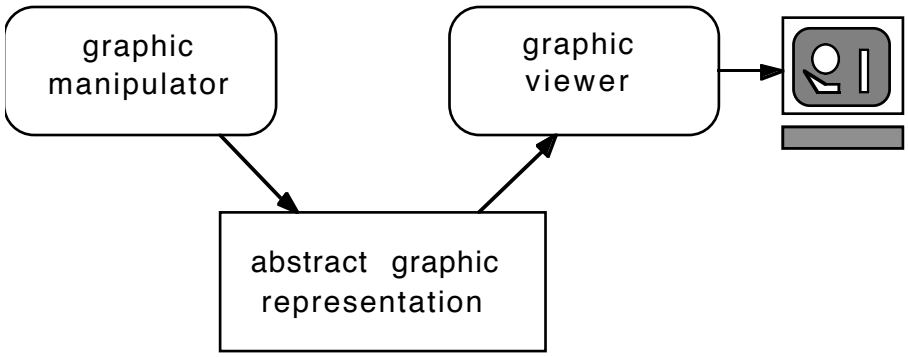

Figure 1
A hypothetical system for manipulation of graphical images

The system consists of three components, an abstract graphical representation of the image, a graphical manipulator and a graphical viewer. The manipulator modifies the abstract graphical representation. It could be an editor where the user edits the graphical image or some other tool that change the representation. The viewer is used for presentation of the abstract representation on a screen. After the manipulator has performed some modifications to the abstract representation it has to communicate with other components that are dependent of the modification. In this case the viewer has to update the image on the screen according to the modifications.

The components are of different nature. The abstract graphic representation is more basic than the others in the sense that it is essential for the function of all other components. We have no reason to expect that it will be radically modified

or removed. The other two components are, however, not crucial for the function of the other components. We can imagine a system without the graphic manipulator that is used for presentation of graphical images only, or a system without the graphical viewer that only manipulates abstract representations of graphical images. Furthermore we can imagine other components to be added to the system, e.g. a database manager for storage of graphical images, etc.

When the system is reconfigured, i.e. some components are added or removed, it is a significant advantage if the other components in the system is affected as little as possible. We would like to build the system from generic components that are able to perform their specific tasks in different environments without being forced to make any major modifications of the components. With such components we are able to obtain open-ended, non monolithic, systems that may be tailored for different purposes in a straightforward manner.

3. A communication model

The key point that may cause trouble in the previous example is references and communication from one components to other components that may be subject for removal or major modifications, as well as references and communication to new components that may appear in the future. Such references and communication must not be an integral part of the code describing the functionality of a component. It would in that case be scattered in the code of that component and thus troublesome to maintain.

As a solution to this problem we suggest a model where a component is separated into three parts: the interface part, the functional part, and the communication part, as showed in figure 2.

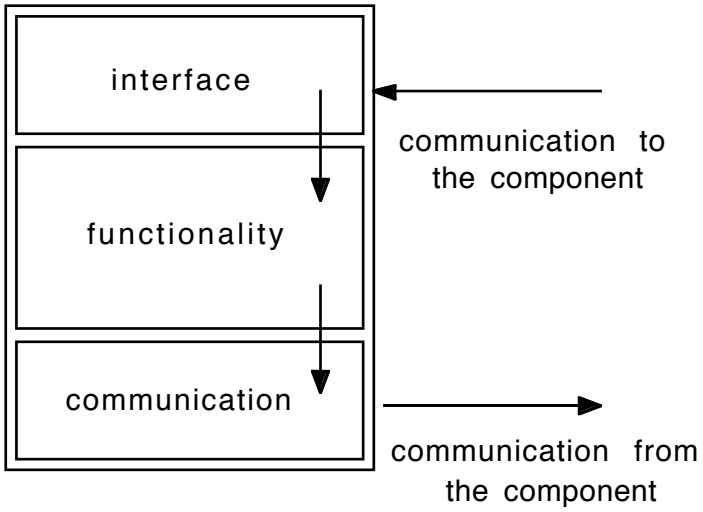

Figure 2
The division of a component and its communication

The interface part specifies how other components may utilize the component, i.e the protocol for the incoming communication. It is similar to the definition modules in Modula-2 and Ada as described in section 1. The functionality part describes

the implementation of the definition part. It realizes the intended functionality of the component. The communication part contains references to other components and the output communication protocol.

The presented model can be realized rather straightforward within object-oriented languages. A component may be described by a hierarchy of sub-classes with the interface part as the parent class. The class describing the functional part will inherit the interface class and extends thus the component with the functionality that implement the interface. The class describing the communication part in turn inherit the functionality class extending it with the output communication protocol.

Using inheritance for composing the components from the three parts gives a simple way for communication between the parts within the component. If the protocol of the interface part exactly matches the methods in the implementation part, the implementation methods may overwrite the interface methods. If there is no one-to-one matching the methods of the interface part may map to other methods in the functional part. This is done by calling the appropriate methods in the sub-class.

The functional part in turn utilizes methods of its sub-class, the communication part, whenever it whishes to notify other components that an event has occurred that might be of interest for them. The communication sub-class keeps track of what other components might be interested of that specific event, and perform the actual communication. The communication class may, as well as the interface class, contain a map that translate the methods in the communication class to methods according to the protocol of the other components it communicates with.

Figure 3 shows an example of the class hierarchy and methods in the graphical manipulator component from figure 1. The example is written in a pseudo language that hopefully would be reasonably easy to understand for persons with some knowledge of object-oriented languages. X class Y denotes that Y is a subclass to X, i.e. Y inherit the properties of X. O.m(p) denotes that the object O´s method m is called with the parameter p. "self" refers to the calling object itself. The methods of a class or class hierarchy are assumed to be late bound like all methods in Smalltalk, and virtual procedures in Simula, are.

```
class ManipulatorInterface;

  method  removeObject(object);

  method insertObject(object);


ManipulatorInterface class Manipulator;

   method  removeObject(object);
        - remove the graphical object in the abstract representation -
        self.notifyObjectRemoved(object),

   method insertObject(object);
        -  insert the graphical object in the abstract representation -
        self.notifyObjectInserted(object);


Manipulator class ManipulatorCommunication;

   instancevariables: reference to the Graphical Viewer

  method notifyObjectRemoved(object);
     GraphicalViewer. showObjectRemoved(object);

  method notifyObjectInserted(object);
     GraphicalViewer.showObjectInserted(object);
```

Figure 3
The three parts of the graphical manipulator component from figure 1

The protocol in interface part contains two methods intended for external use, "insertObject" for insertion of a new graphical object in the abstract representation and "removeObject" for removal of an object. In this example both are overwritten by the methods in the functional part, i.e. we have no translation mapping. The methods in the functional part implement the operations by actually adding or removing the objects from the abstract representation. In addition to that they notify the communication sub-class by calling the methods "notifyObjectInserted" and "notifyObjectRemoved". The communication sub-class has knowledge about the other components in the current environment, their protocol, and the translation map between its own methods and the components protocols.

As we can see in the figure the functional part is independent of the current environment. If the environment change the interface or communication maps may be modified. If, for instance, a new component is added that is dependent on the graphic manipulator we only have to change the communication part by by adding a reference to that component and add a map that maps the methods of the communication part to the protocol of the new component.

Furthermore the (low level) metaphor for communication is separated from the functionality. In a language, or language environment, where we can choose between different communication metaphors, such as message-passing, events, procedure calls, co-routines, or maybe rendezvous, the communication is localized in the interface and communication parts and does not interfere with the functionality.

Using this model we have separated the concerns of function and communication. If we want to reconfigure the system, or build a new one from a collection of components, we only have to supply new classes for communication. Systems written this way would thus be open-ended, straightforward to modify, and hopefully reasonably cheap to maintain.

4. Experiences and alternative solutions

The model has been utilized within the Mjølner project [DLMM 86]. Mjølner is a joint Nordic research project on object-oriented programming environments. In the Mjølner environment programs are represented by attributed abstract trees stored in a common database. Different components, or tools, operate on the common representation. The different tools works incrementally and take care of different tasks such as browsing, editing, static semantic analysis, code generation, etc. The environment is integrated in the sense that the tools cooperate and is presented as one tool to the user. The incrementality and integration puts great demands on the communication between the different components of the environment. Whenever a component has performed a modification to the program representation several other tools need to be informed about the kind of modification, and where it is done. Figure 4 shows one configuration of the Mjølner system using the model.

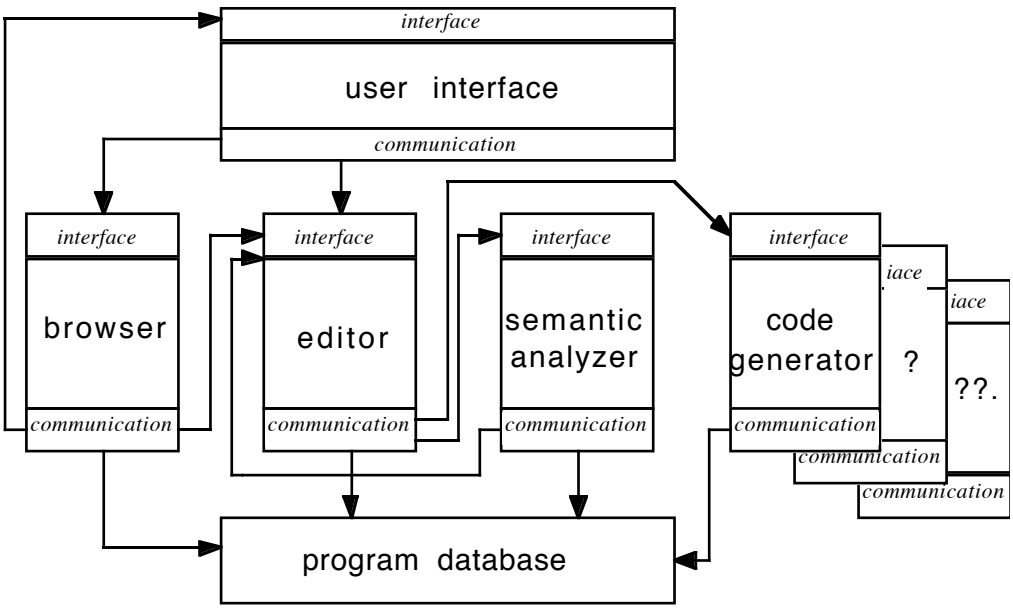

Figure 4
The Mjølner system built from communicating components

An important issue in the Mjølner project has been to experiment with the environment in an explorative fashion. It is thus important that the environment is flexible to reconfigure, e.g. put together specialized environments for different purposes or to compose partial environments for evaluation. It is also important that the environment can be extended in the future with new or alternative components as the system evolves. These goals has been met by using the

model for communication presented in this paper. The implementation language in the Swedish part of Mjølner is Simula, but the communication model is general and could be utilized in most object-oriented languages.

The use of inheritance for putting together the interface, functionality, and communication parts has appeared to have a minor disadvantage. If a generic component is specialized to a number of alternative tools, e.g. the class Manipulator of figure 3 is specialized to ColorManipulator and BlackAndWhiteManipulator, we are not able to use the same communication subclass. The problem arises because of that the inheritance graph (using single inheritance) is limited to be a tree. This is no major problem since different specializations of a component in many cases need different communication protocols. If the same communication protocol is wanted it can easily be solved syntactically by copying the communication subclass and renaming the copy, even if that solution is somewhat awkward. Using multiple inheritance for the composition is not discussed in this paper since this mechanism seems not to be well enough understood yet. No conceptual model for how to use and understand multiple inheritance has been presented at current, only ad hoc implementations.

Using the presented model for communication will cause some additional execution load. The execution speed is, however, not significantly slowed down. Firstly, not all objects in the system has to use the model. Components that not are likely to be removed or radically modified can be called using a communication mechanism supported by the language, such as method invocation or procedure call. Secondly, the communication between components in a system is often, to our experience, rather sparse. Finally the overhead caused by the model is in most cases limited to one additional method invocation.

Smalltalk-80 [Goldberg 83] offers a mechanism for expressing dependency relations in order to coordinate activities. The dependency mechanism is a part of the Object instance protocol and will thus be a property of all objects in the system. The mechanism is quite simple. Each object maintain a collection of dependent objects. Dependents can be added or removed dynamically. Whenever an object performs an action that may affect the dependent objects it is supposed to send the message "changed" to itself. The message may carry one parameter used for describing the kind of change. It will result in that all dependent objects will receive the message "update" with the same parameter. The dependent objects may then perform whatever they find appropriate in response to the change by redefining the "update" method.

The Smalltalk approach is only capable of describing one level of dependencies. When a change is signalled all dependents are notified unregarded what kind of change that has occurred. The model presented in this paper can define one level of dependencies for each type of change. Furthermore it does not force the dependent components to use a certain protocol such as the update method in Smalltalk. Any component with an arbitrary protocol including an arbitrary number of parameters can be used as receiver.

5. Conclusions

A model for communication that facilitates flexible system reconfigurations in object oriented systems has been described. The basic idea in the model is separation of concerns. The functionality of an object or a component should be

independent of the environment. Functionality should be separated from communication. Hence a component has been divided into three parts, the input interface, the functionality, and the output communication.

The style of communication is somewhat different than what is usual in object oriented languages. The functional part do not refer to other components in the environment requesting actions to be executed. Instead they notify the communication part what they have done. The communication part has knowledge of other components in the environment and can map the notifications into actions in other components. It can be viewed as a number of probes that monitor the events in a component and trigs other components when appropriate. New components can easily be connected to the probes and components that are removed may be disconnected in the same manner.

The model separates the communication between the components from the different communication primitives offered by the language or its environment, such as message passing, co-routines, events, etc. It is not restricted to be used in a certain language or environment. It can be employed in most object-oriented languages.

Furthermore it does not force all components or objects to use the model. Components using the model can call components that are not, and components that do not use the model can call components that do. Components that do use the model, however, are straightforward to adapt to different systems used for different purposes, and are easy to maintain if the system is modified.

Acknowledgements

This work was supported by the Mjølner project, partially funded by a grant from the Nordic Council of Ministers.

Ada is a trademark of U.S. Department of Defence. SIMULA is a trademark of SIMULA AS, Norway.

The author whishes to thank the colleagues in the Mjølner project for many valuable discussions and comments on the style and contents of this paper.

References

[DLMM 86] Dahle, H. P., Löfgren, M., Madsen, O. L., Magnusson, M., Mjølner - a Highly Efficient Programming Environment for Industrial Use, Mjølner report no 1, Dept. of Computer Science, Lund Institute of Technology, Sweden, 1986

[DoD 83] United States Department of Defence, Reference Manual for the Ada Programming Language, ANSI/MIL-std 1815-a, 1983

[Goldberg 83] Goldberg, A., Robson, D., Smalltalk-80 The Language and its Implementation, Addison-Wesley 1983.

[SIS 87] SIS Swedish Standard , Programming languages - SIMULA, SIS,

Stockholm, Sweden, 1987

[Wirth 82] Wirth, N., Programming in Modula-2, Springer-Verlag, 1982