Building Independent Black Box Components in C++

Mark Addesso
Software AG

Building Independent Black Box Components in C++

Mark Addesso
Software AG of North America

Abstract

When using Object Oriented techniques to build systems, the relationships between objects creates dependencies which inhibit reuse, unit testing and reliability. By using black box components, these relationships can be factored out of the object to create truly independent components. Techniques to design and build systems in this manner are presented. A fairly complex system built with black box components is also described.

Introduction

The basic idea with object technology was to create packaged units of data and behavior. You could put your hands around an object, design and code it separately, test it as a unit. Once completed, that package of functionality could be reused and would never have to be coded again.

It sounds good in theory, but in practice it is very difficult to create such independent packages. The difficulty is that any complex behavior requires a group of collaborating objects. The boundaries of these packages get blurred as one object requires the services of another, or needs to query another object, or starts to control another object. When the boundaries get blurred all the advantages of the packaged behavior and data blur as well. Objects which depend on others cannot be built and tested independently, and cannot be reused without including all the dependent objects.

These dependencies are inherent in current object design techniques. Most popular techniques let you model interactions between objects. Wirfs-Brock uses Collaboration Graphs [1], Rumbaugh uses Event Trace Diagrams [2], Jacobson uses Interaction Diagrams [3], etc. These interaction may be described as stimuli to another object, a contract between two objects, etc. Any direct interaction between objects creates a dependency. The object sending the message must contain a reference to the receiving object, and must know the exact protocol of the receiving message.

To reuse the sending object in another application, the receiver must also be included, and the application must correctly establish the relationship between the two. Even with only two objects, reuse becomes complicated, but we were only considering one message send. Given all message sends of an object, it may be dependent on many objects. Even worse, the dependent objects, can be dependent on others, etc., etc. Thus, the entire web of connected objects would have to be included and correctly initialized to be reused in another application. This is why currently only primitive classes (lists, widget, etc.) or large subsystems (OLE components) are good candidates for reuse.

Removing Dependencies

In trying to find a solution to the dependency problem I imagined what the ideal way to build and use objects would be. I went back to an early concept in object technology called Software ICs. A software IC is like a hardware IC, it has inputs and outputs which are connected to other ICs. If we could build systems by connecting black box object outputs to inputs, we would have a very loosely coupled system which should have the properties I was looking for.

The first decision was to decide what were the black boxes: objects or methods? If a black box was an object, then an input would be a method and an output would be a message send to another object's method. If the black box was a method, then the inputs would be the method parameters and the output would be the method's output parameters. Methods as black boxes is similar to a data flow approach. Morrison[4] has done some interesting work with "Flow Based" programming which uses this approach. However, I was more interested in building and reusing objects, not methods, so I opted for the first approach - using objects as black boxes.

Objects as independent black box components have the following properties:

1) They maintains their own state. A component cannot rely on any other component for state information, as that would create a dependency.

2) A component cannot query another component. When an output of a component is connected to the

input of another, the data flow is strictly in one direction.

3) A component can contain other components, but can only have one parent. This is consistent with the clear boundaries of a black box.

The API of a component has three sections: inputs, outputs, and controls. Component inputs set the state of the component or pass data to the object to process. Outputs of a component are generated when the state of a component changes or as the result of an operation of the component. Controls are used by the parent object to set, query and control the sub component. Since the parent's behavior is implemented by its sub components, parents can have intimate knowledge of the sub components behavior.

Building Systems with Components

Given this definition of components, systems are built by parent components connecting the outputs and inputs of its sub components.

A parent can connect sub components for one of three purposes: to implement a sequence of messages, to broadcast a state change, or to pump the output stream of one component to the input of another.

o Sequence of Messages

This usage is similar to a data flow style of programming. Each component transforms its inputs to outputs and sends them to the next component in the chain.

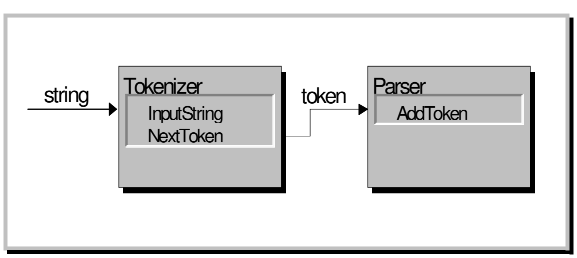

Figure 1 - Parser Example

This example is a parser which must tokenize its input and generate a parse tree. Here, a string is sent to the Tokenizer component's input_string port. It creates a token for the string and sends it out its next_token output port which gets sent to the ParseTree component.

o Broadcast of a State Change

This usage is similar to the MVC (Model-View-Controller) architecture developed in Smalltalk. A change in a model (or business) object is broadcast to all its views which are then updated.

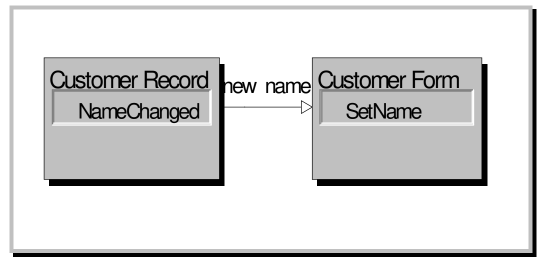

Figure 2 - Customer Entry Example

This example is from an application which lets the user look at and change customer records. Here, when the name is changed in the CustomerRecord component, the name_changed output is triggered which sends the new name to the CustomerForm, which updates the screen. Note that the CustomerRecord would have a number of output ports for various state changes. Also note that this use is different from a typical MVC setup, because here the "View" object does not have to query the model for the data, the data is always passed. See [5] for more information on MVC.

o Pumping an Output Stream to Another Component

This usage covers cases where a set of data is processed by a component and the results are sent to another component.

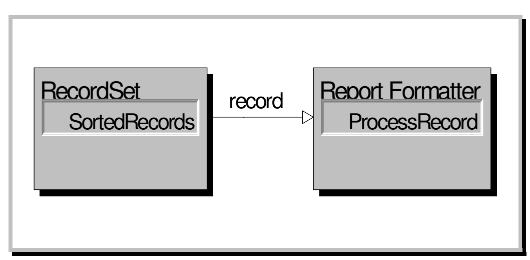

Figure 3 - Report Processing Example

This example is from an application which produces reports given a list of records. Here, the RecordSet component contains a list of records which will be formatted by the ReportFormatter component. The sorted_records output port pumps all the records in sorted order to the process_record input port of the ReportFormatter component.

Given these three types of interactions, and the fact that a parent component can control its sub components, the question is: Can we build complex systems with just these constructs? The case study presented in the next section describes a fairly complex system which was successfully implemented with these techniques.