

Interaction Graphs: A System for Specifying and Generating Object Interactions

*Neeraj Sangal, Edward Farrell, Karl Lieberherr
Tendril Software, Inc.*

1.0 Introduction

This paper describes a new technique for Object Oriented Programming which can lead to the production of much higher quality software and for significantly quicker development. This technique uses Interaction Graphs and has been implemented in StructureBuilder, a development tool for Java Programming, from Tendril Software (www.tendril.com).

The origins of this technique are in a decade long research program at Northeastern University on Adaptive Programming called the Demeter Project. Like Demeter, StructureBuilder internally views the programming process in terms of navigating through the object model. This view of thinking of objects as a network and providing for their transportation is what a large part of the programming task consists of. Even though programmers don't always conceptualize their task in these terms – this is an essential aspect of virtually all programming. Indeed, object oriented programming is an attempt to organize this network of objects and to provide programmers with rules on how they might access the network.

Unified Modeling Language (UML) defines a number of different diagrams to help in the construction, analysis and comprehension of Object Oriented Programs. Of all those diagrams, in our view, two of the most important kinds are the Class Diagrams and the Object Interaction Diagrams. Class Diagrams give the static view of how classes relate to each other. Object Interaction Diagrams give the dynamic view of how a program organizes the interaction of these classes to perform specific functions.

However, there is a key difficulty. UML is the stuff we use to think and communicate with while we use a programming language, like Java, for actual implementation. Unfortunately, once you begin writing in Java it becomes nearly impossible to go back to UML. With Java and other object oriented languages, it has become possible to map class diagrams to code and vice versa. In this paper we will show a new technique of how you can start from an Object Interaction Diagram and generate actual Java code from it. Furthermore, it is possible to do round-trip-engineering with the generated code. *This ability to do round-trip engineering with Object Interaction Diagrams is new and significant.*

In this paper, we will show how an Object Interaction Diagram can be extended into an Interaction Graph. We will formalize the concept of Interaction Graphs which are composed from actions. Looking at a program in terms of action structure gives a useful overview abstracting away from data structure and code details. Additional key contributions are the untangling of actions and properties and the handling of object transportation based on the structure of the actions. By thinking of actions as parameterized code blocks which have inputs and outputs we were able to build a powerful system for visually representing and composing data structure operations and to ultimately create a general system for representing Object interactions and for generating code which implements those interactions.

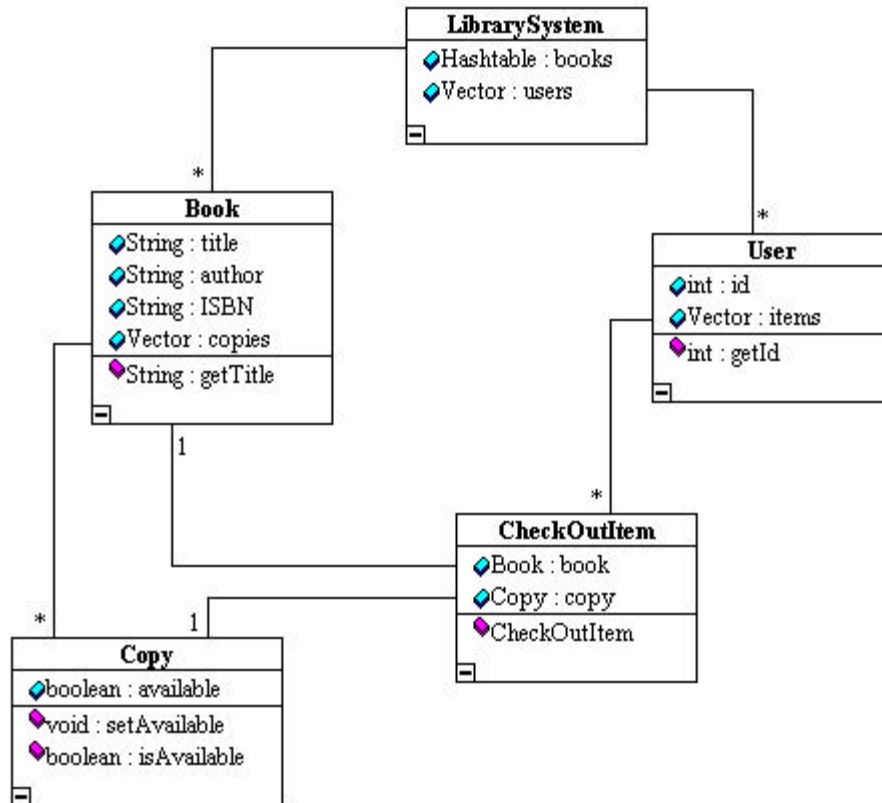
This paper starts with an example of a library system. We will first show a sequence diagram to represent a checkout process and then show how to build an interaction graph from it. After this example we will formally define Interaction Graphs and Actions and explain how objects are actually tracked and transported.

2.0 Example – Library System

Here is an example of a Library System. We first show its Class Diagram, then show a Sequence Diagram illustrating the interacting objects for a check out function. We then show how those functions are implemented using an Interaction Graph and how code is generated.

2.1 Class Diagram

Here is a class diagram for a simple library system in UML notation:

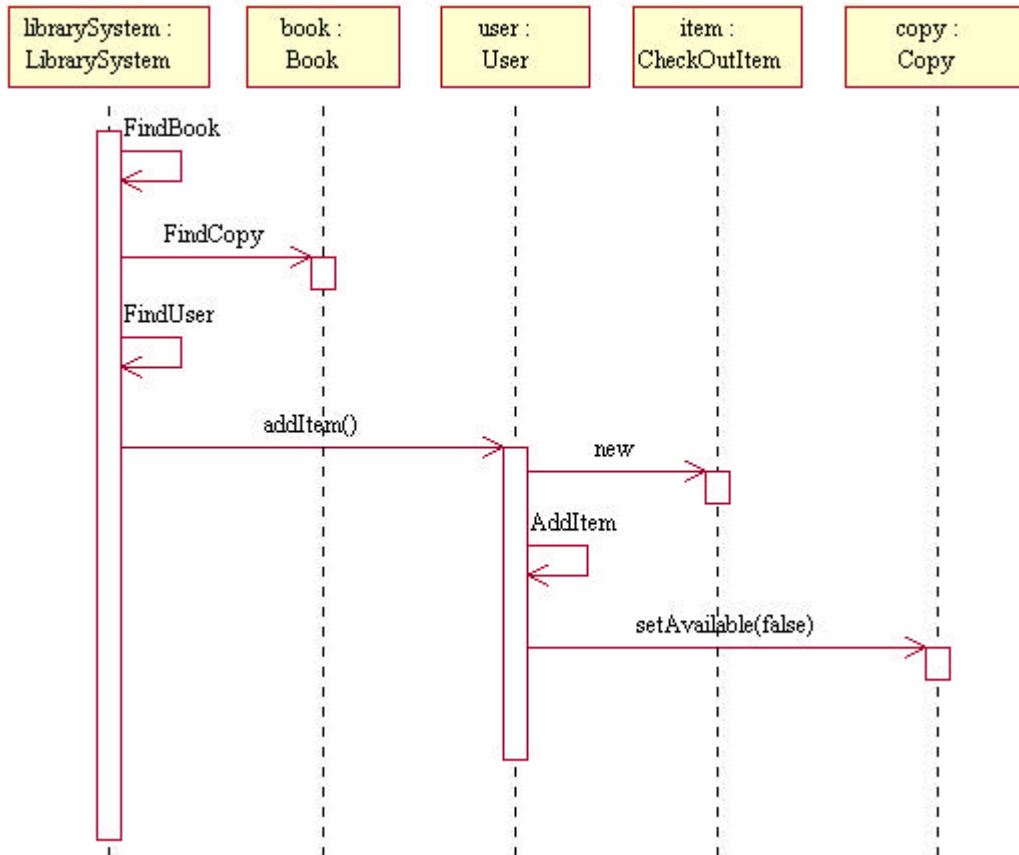


The class **LibrarySystem** contains multiple instances of the class **Book** and multiple instances of the class **User**. Since the library may have multiple copies of a book, each book contains multiple instances of the class **Copy**. Each book that is checked out is represented by a **CheckOutItem**. A **CheckOutItem** points to a **Book** and a **Copy**. Each user contains multiple instances of the class **CheckOutItem** one for each copy of a book that is checked out by that user.

For purposes of illustration, all of these are implemented as in memory structures. The one-many relationships are implemented using the basic Java collections: `Vector` and `Hashtable`. Note that the thrust of this paper would be unaffected whether different collections types were used or whether a persistence mechanism such as a data base is employed.

2.2 Sequence Diagram

Sequence Diagrams illustrate how these classes are used for specific functions. Note that it is not necessary to specify a Class Diagram prior to creating a Sequence Diagram. However, as you iterate over the design you will begin filling in the Class Diagram as you continue to refine your Sequence Diagram.



This sequence diagram shows what is involved in checking out of a book by a user. First we will try to find a book that the user is checking out. Then we will try to find a copy that is available. A book cannot be checked out if a copy is not available. Next we will try to find the User who is trying to check out the book. Based on the book and the copy we will construct a CheckOutItem, add this item to the user and mark the copy as not available.

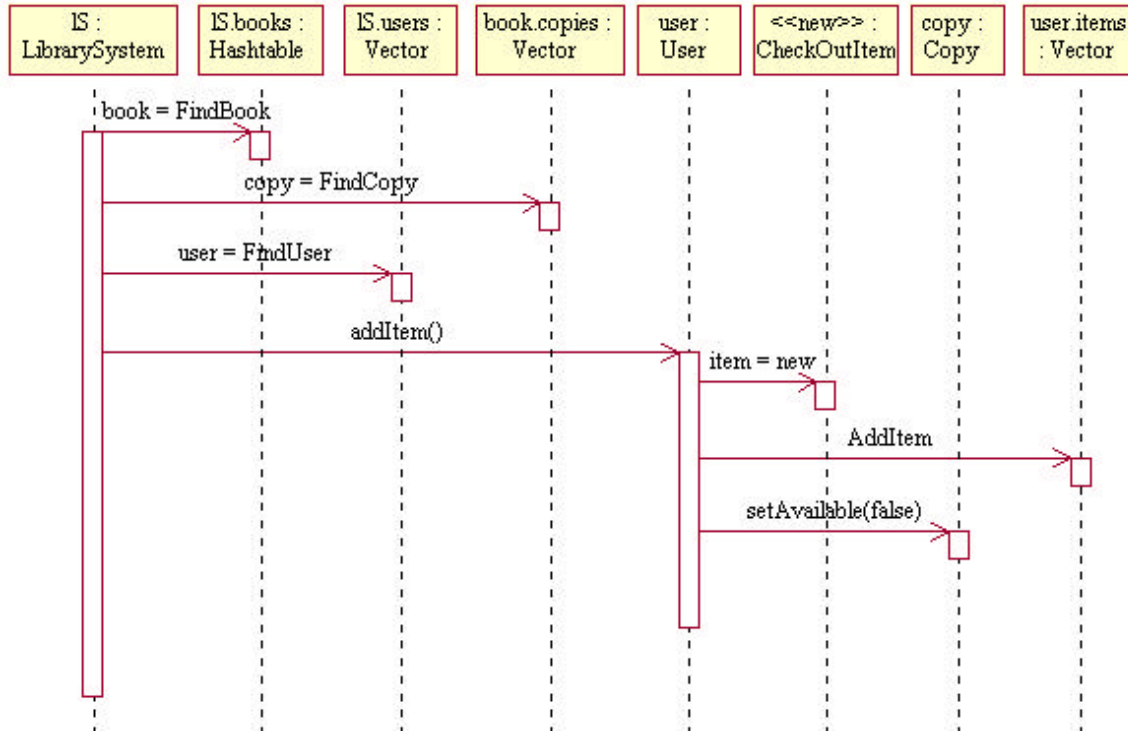
Note that a number of data structure operations are hidden behind several of the messages. For instance the message **FindBook** will operate on the data member **librarySystem.books**. It will iterate through the collections looking for the appropriate book. Similarly, **FindCopy** operates on the object **book.copies** and **FindUser** operates on the object **librarySystem.users**.

2.2 A Sequence Diagram with More Details

We will now, re-write this sequence diagram to show more of the objects involved. When sequence diagrams are used as pure diagramming aides many users frequently do not show this level of detail.

However, some authors have pointed out the importance of showing messages to multi-objects. [Applying UML and Patterns – In Introduction to Object-Oriented Analysis and Design by Craig Larman, Page 181].

The new sequence diagram shows the collection objects. Note that traditionally the messages in a sequence diagram represent method calls. As we pointed out earlier, we have chosen to think of these messages as actions which can be simple method calls. But they can also be code fragments useful for capturing many of the common data structure manipulation operations. Note that for purposes of presentation we have used the short form 'ls' to represent 'librarySystem' in this picture.



2.3 Interaction Graphs

Now we will represent this Sequence Diagram in an Interaction Graph. The Interaction Graph contains enough details to deal with variable scoping, variable transportation and to generate the complete code.

Method:

CheckOutItem LibrarySystem.checkout(bookName, userId)

It takes the name of a book and userId as input and returns a CheckOutItem.

The following sequence of actions describe the program. Each of the actions has a set of properties associated with them which serve to parameterize the generated code. Properties for many of these actions are shown after the interaction graph. We have added comments to each of the actions to assist the reader in understanding the interaction graph.

```

// Action find on the collection librarySystem.books.
// Searches the collection using bookName as input and outputs the object book.
librarySystem.books.find

// Action find on the collection book.copies.
// Searches the collection for an available element and outputs the object copy
book.copies.find

// Action find on the collection librarySystem.users.
// Searches the collection using userId as input and outputs the object user
librarySystem.users.find

// Action NewMethod on the object user.
// Creates a new method called addItem() and contains 3 sub-actions.
user.addItem()

    // Action Execute on the class CheckOutItem.
    // Calls the constructor with: item =new CheckOutItem(book,copy)
    CheckOutItem.new

    // Action Add on the collection user.items.
    // Adds the object item to the collection.
    user.items.Add

    // Action Execute on the object copy
    // Executes a method containing the expression: setAvailable(false)
    copy.unavailable

```

We have added comments to each of the action to explain what each action does. In our system each of the actions has a set of properties associated with them. The code generated by each of the actions is parameterized by properties which allows the programmer to specify further detail. For instance, the find action still needs properties to specify what find should be based on. Here are examples of properties associated with some of these actions:

Specifying librarySystem.books.Find

This is a “Find” action on the object “librarySystem.books”. In our example, this is organized as a Hashtable. We specified the following properties:

```

ResultType/ResultName:      Book/book
Find Criteria:               hash key is bookName

```

Specifying book.copies.Find

This is a “Find” action on the object “book.copies”. In our example, we organized it as a Vector. We specified the following properties for it:

```

ResultType/ResultName:      Copy/copy
Find Criteria:               $curobj.isAvailable()

```

Note that the criteria to find is to look for an available element in the “copies” collection.

2.4 Benefits

What is so striking about this approach is that the Interaction Graph conveys the overview of the function that is easy to understand. The details of the data structure are abstracted away from the user. The properties of each of these actions on the other hand contains the details of what it takes to implement the interaction graph.

3.0 Formalizing Interaction Graphs

An interaction graph is a directed labeled tree. The nodes of an interaction graph represent variables with their classes and are labeled by **[ObjectName]:ClassName**. **ObjectName** is a path in the class diagram following data members and denoting some data member. The first element of a path may be a local variable of the Interaction Graph or an instance variable. The edges represent actions and each edge is labeled with one action. An edge (**s.Source, Action, t.Target**) has the following intuitive meaning: somewhere in the code of class Source (in the method or statement currently generated) there is a call to Action for an object in variable t of class Target. Variable t must be available in class Source.

The actions are totally ordered and this order determines the order of the generated code and the order of how the actions are executed. An action is attached to a class and it has the following information associated with it:

1. An action name
2. Properties of the action. For example, the Execute action, which represents a call to a method, has properties which allow the user the name of output variable and to specify the expression to be used to make the method call.

Four pieces of information for an action are computed from the properties of the action:

1. Inputs
2. Outputs
3. A Boolean flag, **constrainScope**, which indicates whether the output objects of this action and the output objects of its sub-actions are visible outside its sub-actions.
4. A Boolean flag, **hasSubActions**, which indicates whether the action can have sub-actions. Some actions such as conditionals and new method always have sub-actions. Other actions, such as Add, never have sub-actions. For yet other actions such as Remove it depends upon their properties whether they have sub-actions.

The following rules are associated with variables in an Interaction Graph:

1. Method Parameters. These variables are visible everywhere in the interaction graph. Note that it includes the “this” variable for non-static methods.
2. Interaction Graph Global Variables. These variables are defined at the beginning of the interaction graph and are visible everywhere in the interaction graph.
3. Interaction Graph Local Variables. These variables are output by different actions. These variables are visible everywhere in the interaction graph except those variables whose scope is constrained by the constrainScope property of an action. Such variables are only visible in all sub-actions of the action which constrains them. As usual, a local variable may not be used before it is assigned.
4. Class Diagram Global Variables. These are class based data members. In Java these correspond to static variables in classes.

Interaction graphs support a very flexible approach to “broadcasting” variables across classes. Actions don’t have to specify explicitly what the inputs and the outputs are and the required inputs will be automatically delivered from potentially distant classes. The broadcasting is done based on variable names. Inside an interaction graph, the same variable name may be used in two methods in two different classes and the two are aliased. This improves the readability of collaboration specifications. In ordinary object-oriented programming the programmer has to decide for each collaboration where the objects need to be transported to. With interaction graphs issues of object transportation is automated based on the structure of the interaction graph. With interaction graphs code blocks are less coupled because all the glue code which does the transportation is left out.

3.1 Example of visibility of variables in an Interaction Graph

Consider the following Interaction Graph where output objects are shown on the right of the action and the actions are numbered in parenthesis:

```
Class.method(param1, param2) global variables: globalVar1, globalVar2
  obj2.find                →outObj2      (1)
  obj3.iterate             →outObj3      (2)
    obj4.find              →outObj4      (3)
    obj5.Execute           →outObj5      (4)
  obj6.Execute            →outObj6      (5)
```

Here is the how the visibility of the variables within this Interaction Graph is determined:

1. *param1, param2*: Visible at all actions.
2. *globalVar1, globalVar2*: Visible at all actions.
3. *outObj2*: Visible to actions (2), (3), (4), (5).
4. *outObj3*: Action (2) on obj3 constrains the scope of its outputs and the outputs of its sub-actions. Therefore, outObj3 is visible only to actions (3) and (4).
5. *outObj4*: Action (3) does not constrain its output but Action (2) does ; therefore it is available to Action (4).
6. *outObj5*: Action (4) does not constrain its output but Action (2) does; therefore it is not available to any of the actions displayed. Note that if there was another sub-action of Action (2) after Action (4) then it would have been available to that sub-action.
7. *outObj6*: Action (5) does not constrain its outputs but there are no other actions after Action (5) therefore this action is not available to any of the actions. Note that if there were actions after Action(5) then this object would have been available to those actions.

The key differences between Interaction Graphs and Interaction Diagrams are:

1. Interaction Graphs are computationally complete.
2. Interaction Graphs use Actions which have properties. This separation into Actions and Properties allows useful overview while abstracting away from data structures and code details.
3. Actions have input and output objects; Actions impose scoping rules on output objects. Code generation leads to automatic object transportation.

3.2 Type of Actions

There are three types of actions:

1. **Actions based on objects:** All objects support the **NewMethod**, **Execute** and **Get** actions. The **NewMethod** and **Execute** actions corresponds to a method call. **NewMethod** can have sub-actions while **Execute** cannot have sub-actions. The sub-actions of a **NewMethod** will be in a new method that will generated during code generation. Additional actions may be configured into the system and they are based on the type of the object. For instance, we have developed the **Add**, **Remove**, **Find** and **Iterate** actions on all Java collection types such as Vector and Hashtable.
2. **Actions based on class:** Actions based on class correspond to calls to static methods and constructors. The **Execute** action is the only action of this type. The target object for these actions is blank, the target class is the name of class on which the action is called.
3. **Actions based on program elements:** These actions are not associated with any object or class. Examples of these actions are **If**, **Elseif**, **Else** and **Iterate**. The target object is the same as the source object for these actions.

All actions have properties associated with them. These properties allow the programmer to refine what the action does. For instance, the **Find** action has properties which allows the programmer to specify the condition to be used to do a find. Therefore, these properties serve to parameterize the generated code fragment. Each action can be queried for the following:

Here are a few examples of actions:

Action:

objB.Execute

Properties:

| | |
|------------------------|--------------------|
| ResultType/ResultName: | ClassA/objA |
| Method to Execute: | method(objC, objD) |

The input objects are: **objB**, **objC**, **objD**

The output object is: **objA**

This action can have sub-actions

This action does not constrain the scope of output objects

Action:

objA.hashtableBs.Find

Properties:

| | |
|------------------------|----------------------|
| ResultType/ResultName: | ClassB/objb |
| Find Criteria: | hash key is bookName |

The input objects are: **objA.hashtableBs**, **bookName**

The output object is: **objB**

This action cannot have any sub-actions

This action does not constrain the scope of output objects

Each Action in an Interaction Graph has the “smarts” to know what its input and output objects are. It also knows what constraints it imposes on its outputs and the outputs of its sub-actions. Therefore, our system can track all objects that are available at any action. Furthermore, during code generation, our system can also automatically generate any necessary object transportation.

4.0 Transporting Objects

Actions generate new objects as well as make use of existing objects. Since the code generated by different actions is in different methods, it is essential that objects be transported correctly from one method to another.

4.1 Compute the Pairs of Actions Involved in Transportation

The basic approach is to look at the input and output actions for each object. Consider the object **obj** and assume the actions which either generate the object or use the object are organized in a list in order of the action call sequence. Here is a simplified example of how this is done:

Assume that for **obj**, these actions are { **(a1, output), (a2, input), (a3, output), (a4, input) ...**}. For a short form we will write this as {**s1, d2, s3, d4...**}.

For any destination **d**, let us define the following terms:

1. **s[d]** represents the immediate preceding source action.
2. For any source **s** of destination **d**, the conditionals which enclose **s** but does not enclose **d** are **c[s,d]_i**, where **i=0..N-1**
3. The transportation between two actions is defined by **T(d, s)**.

The total transportation required to transport **obj** to **d** is:

$$\mathbf{Tot(d)} = \mathbf{T(d, s[d])} + \mathbf{SUM(Tot(c[s,d]_i))}$$

Notice, how the presence of conditionals can lead to implicit transportation of objects.

4.2 Compute the Transportation for a Pair of Actions

Computing the transportation of an object between two actions **a1** and **a2** involves figuring out the downward and upward transportation through various methods. If both actions are in the same method then no transportation is necessary. A downward transportation occurs when **a2** is in the sub-tree of **a1**. An upward transportation occurs when **a2** is not in same method as **a1** and **a2** is not in the sub-tree of **a1**. Note that in the last case we may have to transport the object up to the highest common node in the Interaction Graph and then transport it down.

The exact Object Transportation technique is language dependent. In Java, downward transportation takes place by sending objects and arguments to methods. The “this” case is dealt as a special case of downward transportation. In Java, when methods need to return objects back to the caller they are returned through the return variable. However, in the general case, more than one object may be returned back; in that case one object is returned by a return variable while all the other objects are returned back through wrapped objects passed in as method arguments.

Here are some examples of object transportation. The last example shows how conditionals can affect object transportation.

4.3 Downward Transport

Assume that we have two methods `m1()` and `m2()`. Also assume that `m1()` contains Action1 and Action2. Action2 is a call to `m2()`. Now assume that Action1 generates **obj1**. Assume that `m2()` contains Action3 which uses **obj1**.

Method `m1` contains:

```
Action1      → obj1
Action2 (calls m2)
```

Method `m2` contains:

```
Action3      ← obj1
```

Code of the following form would be generated:

```
m1()
{
    code for Action 1
    m2(obj1);
}

m2(Object obj1)
{
    Code for Action 2
}
```

We generated the method `m2` with the appropriate signature and also generate the actual call to it with the correct right arguments.

4.4 Upward Transport

Consider the following:

Method `m1()` contains:

```
Action1 (calls m2)
Action2      ← obj1, obj2
```

Method `m2()` contains:

```
Action3      → obj1, obj2
```

Code of the following form would be generated:

```
m1()
{
    obj1 = m2(wrapObj2)
    obj2 = wrapObj2.getValue();

    Code for Action 2
}

m2(wrapObj2)
{
    Code for Action 3
}
```

```

        wrapObj2.setValue(obj2);
        return obj1;
    }

```

Note that the Java return mechanism allows us to pass back only one argument. Therefore, we pass back the second object in a wrapper. An alternative mechanism would be to put both obj1 and obj2 inside a single object and then pass that object back.

4.5 Transportation in the presence of Conditionals

Conditionals can lead to implicit transportation of variables. Here is an example where one of the actions which outputs an object is inside a conditional. Notice how, in this case, it leads to the same object being passed in as well as being passed back.

Method m1() contains the following:

```

Action 1           →obj1
Action 2 (calls m2)
Action 3           ← obj1

```

Method m2() contains the following:

```

If (condition)
    Action 4       →obj1

```

Code of the following form would be generated:

```

m1()
{
    Code for Action 1
    obj1 = m2(obj1);
    Code for Action 3
}

m2(obj1)
{
    if (condition)
        Code for Action 4
    return obj1;
}

```

6.0 Comparison to Demeter

As we stated at the outset, StructureBuilder and the Demeter System share the same philosophy in terms of thinking of the programming process as manipulation and navigation through a network of objects. While the Demeter System has been developed for research, StructureBuilder is a commercial product. Here is a brief summary of similarities and differences:

1. The concept of traversals through an object model is fundamental to Demeter. The edges of a Demeter traversal are linked together by the static relationship between classes. The concept of an Interaction Graph is fundamental to StructureBuilder. An Interaction Graph consists of actions linked together by objects moved around by the "Object Transporter".
2. Interaction Graphs and Demeter Traversals are distinct and can be combined successfully. A linear traversal in the class graph shows in the Interaction Graph as a consecutive sequence of actions. Hopefully, the Interaction Graph contains many sub-graphs that are matching pieces of the class graph. Such traversals can be abstracted into TRAVERSAL-VISITOR actions. The code parameters of a TRAVERSAL-VISITOR action are a traversal specification and a visitor class.
3. In a traversal, when going across the one-many boundary, the Demeter system iterates over the "many" objects. StructureBuilder, on the other hand, uses actions to bridge objects. Since these actions are general the user is not limited to just iterating for purposes of traversals. Thus, many data structure operations such as add, remove and find can be specified very succinctly in StructureBuilder.
4. A key idea behind Demeter traversals is adaptiveness. These traversals can be specified generally so that they can be applied to a whole family of class diagrams. StructureBuilder does not support adaptive traversal specification.
5. StructureBuilder does not have any built-in support for traversing parts of inheritance hierarchies. The user has to use a conditional action to choose selected inheritance paths. Demeter will automatically generate empty methods for classes which are not in the traversal.
6. StructureBuilder supports the powerful concept of actions. Actions have a set of properties which is intended to capture the common usage paradigms associated with that object type. Furthermore, additional actions can be programmed and dynamically loaded into the system. This means that additional data structure types and data base interfaces can be added in without having to do any work to the underlying StructureBuilder system.
7. StructureBuilder uses a visual point and click system for its user interface. It has support for creating UML diagrams and reverse engineering of Java source code. It also has support for creating Interaction Diagrams and for mapping Interaction Diagrams to Interaction Graphs.

7.0 Why is this better than writing code?

If you have an object model and know how to manipulate it then why not directly write code – why even bother going through a Sequence Diagram. There are a number of reasons why this is a much better approach:

1. Development is much faster. You can craft up how your classes are used fairly quickly. Even more importantly, you can experiment with different class diagrams and different data structures. The process of iteration is critical to good software development.

2. It is very easy to communicate what a method does using Interaction Diagrams and Interaction Methods. Most software development will be done in teams and these teams are going to change over time. Using Class Diagrams and Sequence Diagrams that are guaranteed to be current is an excellent way to communicate what the program does.
3. Large parts of your program are generated for you. Furthermore the parts that are generated relate to object interaction and tend to be more tedious and error prone. This approach can virtually eliminate errors in these parts. Therefore, it opens up the possibility of writing highly reliable and error free programs.