Interaction Schemata: Compiling Interactions to Code

Neeraj Sangal[*], Edward Farrell[*], Karl Lieberherr[†], David Lorenz[†]
[*]*Tendril Software, Inc, Westford, MA 01886-4133*
[†]*Northeastern University, Boston, MA 02115-9959*

Abstract

Programming object interactions is at the heart of object-oriented programming. To improve reusability of the interactions, it is important to program object interactions generically. We present two tools that facilitate programming of object interactions. StructureBuilder, a commercial tool, achieves genericity with respect to data structure implementations for collections, following ideas from generic programming, but focussing only on the four most important actions add, delete, iterate and find that are used to translate UML interaction diagrams into code. The focus of StructureBuilder is to generate efficient code from interaction schemata that are an improved form of interaction diagrams. DJ, a new research prototype intended for fast prototyping, achieves genericity with respect to the UML class diagram by dynamic creation of collections based on traversal specifications.

1 Introduction

The Unified Modeling Language (UML [BRJ96]) defines 9 kinds of diagrams, listed in Figure 1, to help in the construction, analysis and comprehension of object-oriented programs. Of those diagrams, this paper focuses on one important kind: *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 instances of these classes to perform specific functions. Design tools (like *Rational Rose* [S98], *StructureBuilder* [SB], etc.) make it possible to generate code from class diagrams; and visualization tools make it possible to construct object-interaction diagrams by tracing the execution of the program (e.g., *Program Explorer* [LN95]).

Performing the translations in the opposite direction is possible albeit more complex. For class diagrams it is not that hard. By parsing the class code, or using reflection capabilities, a class diagram can be produced by means of reverse engineering (e.g., the on-going work at MIT on generating object models from Java [J99]). Indeed, with Java and other object-oriented languages, it has become possible to map class diagrams to code and vice versa. For object-interaction diagrams, however, there is a key difficulty. There is not enough information in the interaction diagram to do the job. One can try to overcome this difficulty and construct code by abstracting over *execution patterns* [DLWV98] in object-interaction diagrams. But this would require a working program to begin with. In this paper, we show an incremental direct technique for moving from an interaction diagram to code.

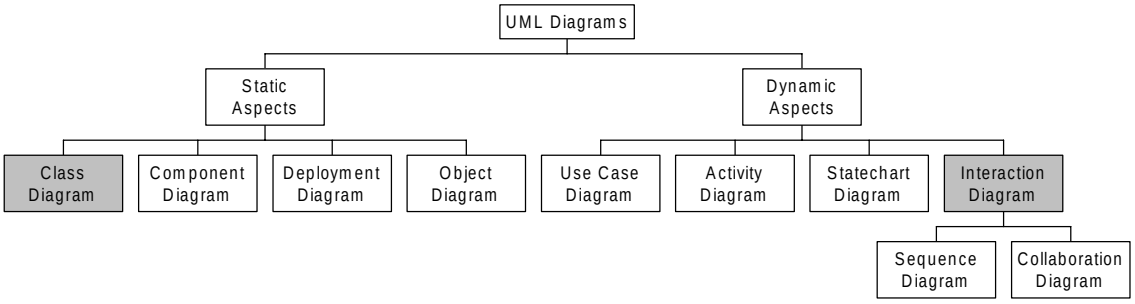


Figure 1

1.1 Motivation

Diagrammatic notations must be very precise to express the design accurately and yet imprecise enough to permit different implementations. Interaction diagrams are no exception. What makes interaction diagrams useful is that they contain enough information to embody the essential aspects of the object interaction but not too much to become identical to code. The challenge addressed in this work is in specifying interaction diagrams in sufficient detail so that code can be generated, while maintaining the essential simplicity, which is necessary for human communication.

The technique presented in this paper lets you start from an object-interaction diagram and generate actual Java code from it. In working towards this goal, we use the following guidelines:

- The generated code captures just the sequence diagram. The user is expected (and enjoys the freedom) to add additional code which embodies the application logic not captured within the sequence diagram.
- We made a simple generalization of messages. Instead of thinking of messages as just method calls, we treated them as code fragments. This allows us to treat iterations and conditionals as messages. It also allows us to capture common data structure manipulations in actions, that we call: *add*, *delete*, *iterate* and *find*. By parameterizing these actions with properties we found that we were able to capture most of the common usage patterns. These actions embody and convey what they do at a high level, and fit very well into the sequence diagram paradigm. On the other hand, the properties associated with these actions contain enough detail to generate the code. We call descriptions comprising of such actions: *interaction schemata.*

What is so striking about this approach is that an interaction schema conveys an overview of the function in a manner 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 contain the details of what it takes to implement the interaction schema. You can experiment with different class diagrams and different data structures. Interaction diagrams facilitate communication between software developers. Most software development will be done in teams and these teams are going to change over time. Using class and interaction diagrams that are guaranteed to be current is an excellent way to communicate what the program does.

In the rest of this paper, we identify the missing information required to convert an interaction diagram to code. We explain why it is so difficult to generate code from a sequence diagram, and list what we believe the programmer does *inside his head* when converting an interaction diagram into code. We demonstrate two approaches implemented in two tools. The first tool generates large parts of your program for you. Furthermore the parts that are generated relate to object interaction and tend to be more tedious and error prone. This approach can eliminate many errors in these parts. Therefore, it opens up the possibility of writing highly reliable programs. The process of incremental development is critical to good software development. The second tool uses the Java Generic Library and a traverse action in combination with a domain specific language for object traversal.

2 A Library System Example

Consider a *library system* whose design is given by the class diagram in Figure 2 using the UML notation. The edges represent associations. The **Library** class is associated with the **Book** and **User** classes, which in turn are associated with the **Copy** class. An edge marked with * indicates zero-to-many relationships, but the implementer is free to choose which collection type to use to realize the association. The roles *books* and *users* (the labels on the arrows from **Library** to **Book** and **User**) suggest that a **Library** instance contains multiple **Book** instances and multiple **User** instances. The arrow directions indicate the direction of the references. However, one can imagine an implementation in which, e.g., the **Book** or the **User** instances point to the **Library** instance, or an implementation in which external objects model the associations.

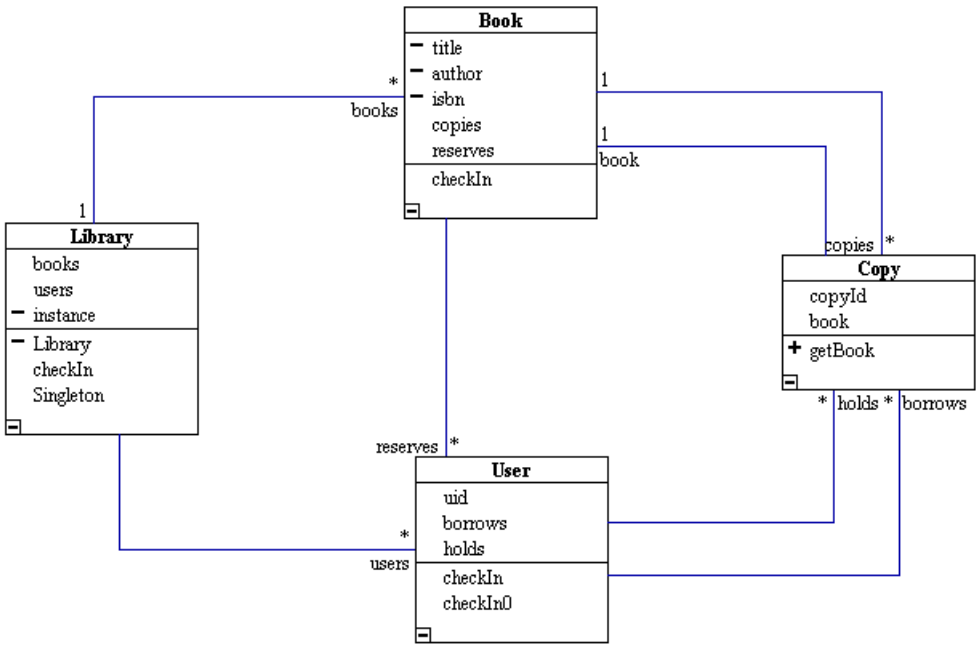


Figure 2

For the purpose of this illustration, nevertheless, all of the associations are assumed to be implemented as in memory structures. Since the library may have multiple copies of a book, each book contains multiple **Copy** instances**.** Each **User** instance contains multiple instances of **Copy,** one for each (copy of a) book that is checked out by that user (the *borrows* role). Each **User** also holds a collection of copies: the ones that are ready to be picked up (the *holds* role). Each **Copy** instance references a single **Book** instance. The zero-to-many relationships are implemented using the basic Java collections: Vector and Hashtable. Note that the thrust of this paper would be unaffected whether different collection types were used or whether a persistence mechanism such as a database is employed.

2.1 Returning a Book

Consider now a sequence diagram for the library system. For clarity, we shall concentrate only on sequence diagrams, but the technique described is applicable also to collaboration diagrams and other kinds of object-interaction diagrams. Sequence diagrams illustrate how objects of 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.

The sequence diagram in Figure 3 shows the details of checking in (i.e., returning) a book by a user. First we find a user with the specified **uid**. Then we remove the copy from the user's list of borrowed books. Next we access the book of the removed copy, and remove the first user from the reservation queue. Finally, we add the removed copy to the **holds** list of the reserver and notify him.

Note that a number of data structure operations are hidden behind several of the messages. For instance, the message **find** will operate on the data member **library.users.** It will iterate through the collection looking for the appropriate user.

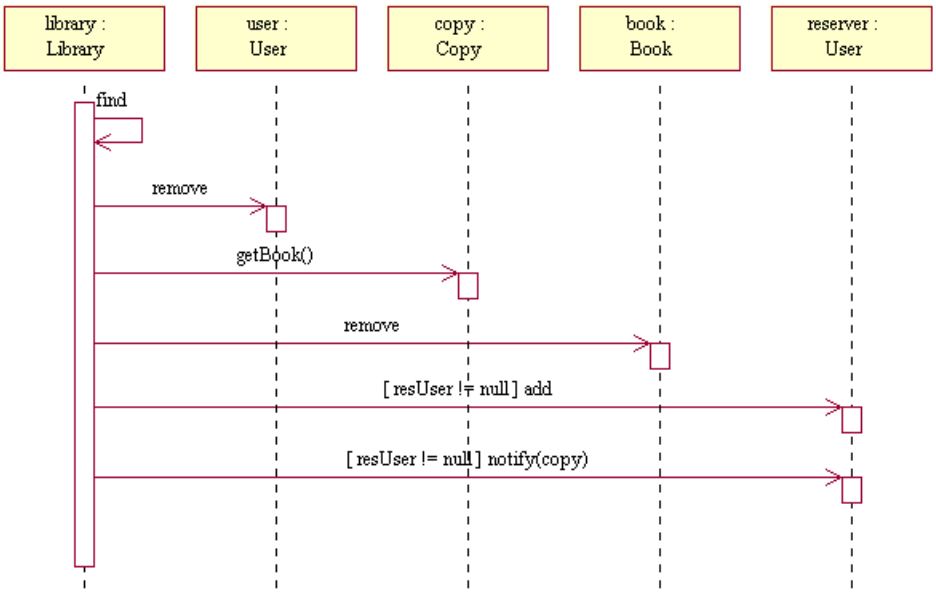


Figure 3

2.2 Going from Interaction Diagram to Code

First, we must figure out *from where* the objects come, and locate the origin of the objects **library**, **book**, **user**, **copy** and **reserver**. Looking at the diagram, we can deduce that **library** is the object which the method is called on and corresponds to the *this* object in the method. On the other hand, it is not immediately clear how **user**, **book** and **copy** objects were accessed. The creator of the interaction diagram knows that **user** is an object which is discovered by **find**, **book** is the object returned by **getBook**, and that **copy** is the object which is returned by *remove*. But we don't.

Second, we must figure out how objects are *transported*. It isn't clear from the sequence diagram how these objects are passed around between different methods. This is a tedious task for the programmer.

However, the problem is even harder than just being an issue of a programmer not being diligent in maintaining interaction diagrams. Interaction diagrams support the notion of *iteration* and *conditionals*; therefore, objects of an interaction diagram are subject to the same *scoping* and *visibility* rules that programmers encounter in programming languages. Indeed, it is easy to construct interaction diagrams, which violate these rules and therefore cannot be used to generate correct code.

Iterations and conditionals limit the visibility of new objects within their scope. This is true when writing code and remains true within interaction diagrams. Any access to such objects outside the scope is illegal. Since sequence diagrams describe collaboration of multiple objects that may be of different types, the code for implementing a collaboration spans multiple classes. Therefore, method parameters and return variables enable those variables to be visible within the appropriate method.

Third, we must fill in the missing details. There are many details that may be missing from sequence diagrams:

- The iteration specified in an interaction diagram does not contain enough information. Frequently, iterations are over a collection of objects. Interaction diagrams generally do not specify what that object is. Often iterations are subject to conditions, e.g., iterate over all elements in a collection that meets certain constraints.
- The method calls need to have parameters and return types. Conditionals need boolean expressions.

- Interaction diagrams typically show the method call stack. Generally there is additional code that a user needs to write. Often sequence diagrams contain the overall structure of the method calls but do not contain all the code. It is up to the user to decide how much of the method logic he wants to show in the sequence diagram.

3 Formalizing Interactions

We introduce *interaction schemata*. An interaction schema is a textual description of object-interaction. From interaction schemata you can generate complete executable code.

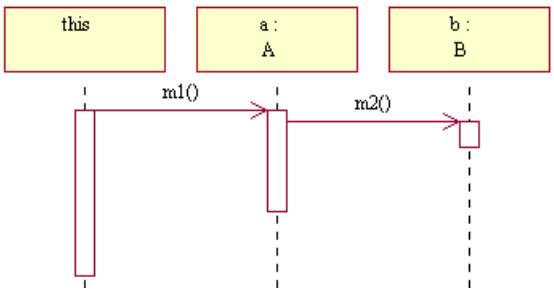
Now we represent sequence diagrams as *interaction schemata*. An interaction schema contains enough details to deal with variable scoping, variable transportation and to generate the complete code.

Interaction schemata are represented as a list of actions. Each message of an interaction-diagram can be translated mechanically into one or more actions. Each action is either of the following form:

```
[interactor → interactor → ...].actionName(exp1,exp2,...)
    return(type1 retexp1, type2 retexp2,...) {...}
```

or a conditional or a looping action. A regular method call is the simplest form of an action. Its arguments and return types correspond to the method signature. The interaction path is of the form [interactor$_1$ → interactor$_2$ → ...]where interactor$_1$ is an object in the interaction diagram and interactor$_2$ is an instance variable of interactor$_1$, and so on. Note that some actions such as conditional and looping actions can contain other actions. The scope of each returned variable is limited to being inside the innermost enclosing conditional or iterative action.

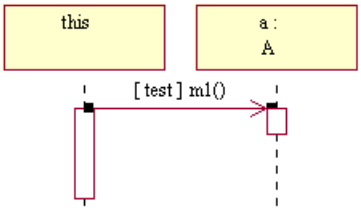
For example, if we have a sequence diagram of the form:



it would be represented in terms of actions as:

```
a.m1() {
       b.m2()
}
```

A conditional message of the form:



would be represented in terms of actions as:

```
if (test) {
       a.m1();
}
```

As we alluded to earlier we have defined additional actions which allow us to capture certain common data structure manipulations. These actions are *iterate*, *find*, *add*, and *remove*. These actions are defined on collection types and the generated code is appropriate for the type of collection.

Now let us look at the action description of the **checkIn** method:

```
Library.checkIn(UID uid, Copy copyId)
```

It takes as inputs a **uid** and a **copyId**. The **uid** identifies the borrower and the **copyId** identifies the copy of the book being returned.

The following sequence of actions describes the program. Each of the actions takes as input a set of expressions, which are the properties associated with them, and which serve to parameterize the generated code. Properties for many of these actions are shown after the interaction schema. We have added comments to each of the actions to assist the reader in understanding the interaction schema.

```
Library.checkIn(UID uid, Copy copyId)
{
    // Find the user with the specified uid
     [library→users→uid].find(uid'current == uid)
         return (User users'current as theUser)

    // Remove the copy with the specified copyid
    [theUser→borrows→copyId].remove(copyId'current == copyId)
         return (Copy borrows'current as theCopy)

    // Call method on copy
    [theCopy].getBook()
         return (Book book as theBook)

    // Find the first user who is on the reserve collection
    [theBook→reserves].remove(reserves'index == 0)
         return (User reserves'current as theReserver)

    // Conditional action
    if (theReserver != null) {
           // Add this copy to the reservers hold list
           [theReserver→holds].add(theCopy)

           // Call method on reserver
           [theReserver].notify(theCopy);
    }
}
```

3.1 Object Transportation

In order to translate the schema into code, the issue of object transportation must be resolved. For example, the action

```
    [theUser→borrows→copyId].remove(copyId'current == copyId)
         return (Copy borrows'current as theCopy)
```

leads to the generation of the method *checkIn* in the classes **Library**, **User**, and **Copy** (Figures 4a, 4b, and 4c show the generated code.) The value **copyId** needs to be passed into the methods *checkIn* from the main method *checkIn* in **Library**. This is an example of an *external* transportation across actions, i.e., between methods which where generated from different actions. Transportation can also occur *internally*. Internal transportation refers to the passing of an object to several generated methods within a single action.

There are many delicate issues involved in object transportation. The object name may change. Multiple objects may need to be returned through a single method requiring the use of wrappers (if the language does not support multiple return values.) Conditionals within actions can lead to unexpected transportation.

There are two approaches to translating to code. StructureBuilder, a Java design tool, takes a code generation approach, in which code is generated for each instance of each action, and method signatures are updated to perform object transportation. DJ [DJ99], a research project at Northeastern University, takes a generic approach, in which traversal specifications are adapted to be used with predefined generic algorithms in the Java Generic Library [JGL], and reflection is used to customize the traversals at runtime. DJ also provides important traverse actions (traverse, gather, fetch, etc.) to support the Visitor Design pattern [GOF] without the problem of structure hardening.

Figure 4a: Method Generated in class Book

```
/** @SBGen Generated Method (2), created by Li-
brary.checkIn(UID, Copy) (Library.2,-5)   */
User checkIn()
{
  // SBgen: Action Remove User from reserves (5)
  User resUser = null;
  int size = reserves.size();
  if (size > 0) {
    resUser = reserves.elementAt(0);
    reserves.removeElementAt(0);
  }
  // SBgen: End Remove
  // SBgen: Return resUser
  return resUser;
}
```

Figure 4b: Method Generated in class Library

```
/**
 * @param uid
 * @param copyId
 * @SBGen Generated Method (2)
 */
void checkIn(UID uid, Copy copyId)
{
  // SBgen: Action Find User in users (2)
  User user = null;
  Object tmpKey;
  Enumeration i = users.keys();
    while(i.hasMoreElements()) {
      tmpKey = i.nextElement();
      user = (User)users.get(tmpKey);
      if (user.uid==uid)
        break;
    }
  // SBgen: End Find
  // SBgen: Call generated method on User (-3)
  Copy copy = user.checkIn(copyId);
  // SBgen: Action Execute method on Copy (4)
  Book book = copy.getBook();
  // SBgen: Call generated method on Book (-5)
  User resUser = book.checkIn();
  // SBgen: Action If (6)
  if (resUser != null) {
    // SBgen: Call generated method on User (-7)
    resUser.checkIn0(copy);
    // SBgen: Action Execute method on User (8)
    resUser.notify(copy);
  }
  // SBgen: End If (6)
}
```

3.2 StructureBuilder: A Code Generation Approach

When code is generated there are a number of issues to consider:

- The actions themselves can embody method calls because not all of the objects that they act upon are accessible in the method specified. In a interaction diagram, the programmer would explicitly specify the method necessary. StructureBuilder, on the other hand will automatically generate a method call if necessary.
- When methods are generated, it is necessary for objects to be transported correctly to the generated methods. It is also necessary for generated objects to be transported back. StructureBuilder will generate methods with the correct signature and return type.

Notice, however, that this is simply an implementation issue. Normally the programmer sets up his method signatures so that scoping issues are dealt with appropriately. Indeed, we could bypass the whole issue of object transportation by leaving it up to the programmer to specify the method signature completely.

Figure 4c: Methods Generated in class User

```
/** @SBGen Generated Method (2), created by Li-
brary.checkIn(UID, Copy) (Library.2,-3)   */

Copy checkIn(Copy copyId)
{
  // SBgen: Action Remove Copy from borrows (3)
  Copy copy = null;
  int i, size = borrows.size();
  for (i=size-1; i>=0; i--) {
    Copy tmpVar = (Copy)borrows.elementAt(i);
    if (tmpVar.getId() == copyId) {
      copy = tmpVar;
      borrows.removeElementAt(i);

      // SBgen: Begin actions (3)
      // SBgen: End actions (3)
      break;
  }
  // SBgen: End Remove
  // SBgen: Return copy
  return copy;
}

/** @SBGen Generated Method (3), created by Li-
brary.checkIn(UID, Copy) (Library.2,-7)   */

void checkIn0(Copy copy)
{
  // SBgen: Action Add Copy to holds (7)
  holds.addElement(copy);

  // SBgen: End Add
}
```

3.3 DJ: A Generic Approach

The DJ tool [DJ99] provides an alternative technique to implement actions by making them more generic. The first observation of DJ is to note that actions like *add*, *find*, *delete* etc. also appear in Generic Programming (GP) as generic algorithms or as methods of container interfaces [MS94, JGL]. Therefore DJ attempts to reuse those generic algorithms. The second observation of DJ is that traversal-visitor style programming is a frequently recurring pattern and therefore DJ offers a traverse action that simplifies traversal visitor style programming.

```
class ClassGraph {Object traverse(Object o, TravSpec s, Visitor v);}
```

ClassGraph-objects are constructed from the Java programs (either in compiled or source form). TravSpec-objects (traversal specifications [L96]) encapsulate a domain-specific language for navigation through object structures and visitor-objects define what needs to be done on top of the traversal. The third observation of DJ is that traversal specifications are an ideal ingredient for Generic Programming lifting the level of genericity by an order of magnitude.

Generic Programming (GP) is about expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible. A second goal of GP is to lift a concrete algorithm to as a general level as possible without losing efficiency. In GP, the algorithms are parameterized by iterators and data structures are connected to the algorithms using iterators as connectors. In Demeter [L96], algorithms are parameterized by traversal specifications and data structures are connected to the algorithms using traversal specifications as connectors. A good way to integrate GP and Demeter is to view traversal specifications as “superiterators” and to have a conversion function that translates a traversal into an iterator that gives access to collection methods (e.g., *find*, *select*, *findIf*) of some generic library such as the Java Generic Library. The goal is to reuse the useful work done in GP and not to reinvent many of the operations already provided by GP. Given a class graph *classGraph*, a traversal specification *sg* and an object *og*, we can create a container *TGC(classGraph,sg,og)* which can be used as argument for many generic algorithms such as *forEach*, *lexicographicalCompare*, *mismatch*, *accumulate*, *count*, *countIf*, *reject*, *select*, *adjacentFind*, *detect*, *every*, *find*, *findIf*, *some*, etc. The generic algorithms operate directly on *og* without creating a new collection object duplicating the information in *og*.

Figure 5: The checkIn method in DJ

```
void checkIn(UID uid, Copy copyId)
{
    // [library→users→uid].find(uid'current == uid)
    //   return (User users'current as user)
    Container LibraryToUserContainer = new TraversalGraph(
        Main.classGraph,
        new TravSpec("From Library to User").container(this));
    User user = Finding.findIf(
        LibraryToUserContainer, new FieldEquals("uid",uid));

    if (user == null)
        return;
    // [user→borrows→copyId].remove(copyId'current == copyId)
    //   return (Copy borrows'current as copy)
    Container UserToCopyContainer = new TraversalGraph(
        Main.classGraph,
        new TravSpec("From User through borrows to Copy").container(user));
    Copy copy = Finding.findIf(
        UserToCopyContainer, new FieldEquals("copyId",copyId));

    if (copy == null)
        return;
    Removing.remove(UserToCopyContainer,copy);
    // [copy→book→reserves].remove(reserves'index == 0)
    //   return (User reservers'current as reserver)
    Container CopyToUser = new TraversalGraph (…) ;

    Container UserToCopyHoldsCont = new TraversalGraph(
        Main.classGraph,
        new TravSpec("From User through holds to Copy").container(user));
    User reserver =
       (User)CopyToUser.remove(CopyToUser.elements());
    if (reserver != null) {
        //[reserver→holds].add(copy)
        UserToCopyHoldsCont.add(copy);

        //[reserver].notify(copy);
        reserver.notify(copy);
    }
}
```

In Figure 5, we demonstrate the DJ approach by rewriting the interaction schema for **checkIn** in DJ. Method **checkIn** is for class Library. We annotate the Java code with the interaction schema to show the correspondence between the two. FieldEquals extends UnaryPredicate from JGL.

The fourth and final observation of DJ is that sequence diagrams can be automatically generated from DJ code to facilitate the understanding of the code at various levels of details. In our current implementation, DJ does no generation or pre-processing of user code.

4 Related Work and Conclusions

This paper describes a new technique for object-oriented programming, which can lead to the production of much higher quality software, and to significantly quicker development. This technique uses *interaction schemata* and has been implemented in StructureBuilder, a development tool for Java Programming, from Tendril Software [SB]. DJ [DJ99] provides additional genericity and ease of maintenance but a slower implementation based on Java Reflection.

The origins of this technique are in a decade long research program at Northeastern University on Adaptive Programming called the Demeter Project [L96]. Like Demeter, StructureBuilder internally views the pro-

gramming process in terms of navigating through the object model. This view of thinking of objects as a network and providing for their transportation is of what a large part of the programming task consists. 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.

Generic actions address two important problems in software development. First, tangling of object collaboration code. We have shown how object collaborations can be more easily expressed using generic actions. Each generic action describes a multi-object collaboration in a succinct way. A generic action cross-cuts the class structure and it is easy to read because all relevant information is part of the generic action and not spread out across several classes and tangled with lots of other code [HL95, K+97]. Second, maintaining UML interaction diagrams is tedious during evolution of the class structure. Generic actions are structure-shy and fairly robust under changing class structures.

Acknowledgements

We thank Josh Marshall and Doug Orleans for designing and implementing DJ and their feedback.

References

[BRJ96] Grady Booch, James Rumbaugh, and Ivan Jacobson. The Unified Modeling Language for Object-Oriented Development, July 1996.

[DLWV98] Wim De Pauw, David Lorenz, Mark Wegman, and John Vlissides. Execution Patterns in Object-Oriented Visualization. In *Proceedings of The Fourth Conference on Object-Oriented Technologies and Systems*, pages 219-234, Santa Fe, New Mexico, April 27-30, 1998. USENIX.

[GOF] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[HL95] Walter L. Hürsch and Cristina Videira Lopes. Separation of Concerns, College of Computer Science, Northeastern University, 1995, February, NU-CCS-95-03, Boston, MA

[J99] Daniel Jackson and Allison Waingold, Lightweight Extraction of Object Models from Bytecode, International Conference on Software Engineering, 1999, May, Los Angeles, CA, http://sdg.lcs.mit.edu/womble/

[JGL] Object Space, Inc., Java Generic Library, 1997, http://www.objectspace.com/developers/jgl/

[K+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming, ECOOP'97, 220-242, Springer Verlag, 1997.

[L96] Karl J. Lieberherr, Adaptive Object-Oriented Software: The Demeter Method, PWS Boston, 1996.

[LN95] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In OOPSLA'95, pages 342-357, Austin, Texas. ACM SIGPLAN Notices 30(10) Oct. 1995.

[DJ99] Joshua Marshall, Doug Orleans, and Karl Lieberherr. DJ Home Page, Northeastern University, May, 1999. http://www.ccs.neu.edu/research/demeter/DJ/

[MS94] D. R. Musser and A. A. Stepanov, Algorithm-Oriented Generic Libraries, Software--Practice and Experience, 1994, July, 24(7)

[S98] Robon Schumacher. Products Hands-On Reviews: Rational Rose 98. DBMS Magazine 11(10), September 1998.

[SB] StructureBuilder 3.1 Tutorial. http://www.tendril.com/