Tutorial for the Java version

If you are reading this document for the first time, we recommend you to continue straight through. You can also jump directly to the section of your interest:

- **Bottom-up design style**
- **Top-down design style**
- **The library of data structures**
  - **- adding a new class to the library**
  - **- deriving a new data structure from an existing class**
  - **- as a wrapper of a Java container**
  - **- expanding Java container to a bi-directional association**
  - **- slow but easier way to develop new associations**

Except for a few minor syntax differences, the association libraries in C++ and in Java are quite similar. If you already read the C++ tutorial, you could probably use the Java version without much thinking about it. This tutorial will show you that, just like the C++ version, this library supports two styles of software development:

- **In the bottom-up approach, you evolve the code right from the beginning, but your data structures (relations, associations) are controlled by a short textual description (schema) which is inside your code. With every compilation, you automatically get a UML class diagram.**
- **In the top-down approach, you first evolve the UML model. The actual coding begins only later when the entire architecture has been well thought through. However, you do not(!) manipulate the UML diagram in a graphical environment. Instead, you edit a short textual UML description (schema), and the diagram is re-drawn automatically. The advantage of using the schema is that, even at the early stages of planning the architecture, it automatically gives you a functional code skeleton which already compiles and which you can instantly evolve.**

Whether you use the bottom-up or top-down approach, you eventually reach the stage when both the code and the model evolve simultaneously, and the IN_CODE modelling supports that in the most elegant manner.

Note that your design is always model driven (MDA=Model Driven Architecture). In contrast to other tools which assume that the model is an independent entity outside of your source, here the model (the schema) is always an integral part of your code.

The idea which lead to this tight integration is to implement all data structures and relations as *associations* and not as *containers (collections)* supplied with Java or provided by other class libraries. There is a big difference between associations and containers. Associations control mutual cooperation of two or more classes, while in containers just one class controls objects or another class. Associations naturally support intrusive data structures including graphs, many-to-many, and various design patterns which cannot be implemented as containers. Intrusive data structures are important because they are better protected against errors, are generally faster, use less space than containers, and do not trigger heap access when working with the data structures.

As you will see in the last section (Adding a new data structure to the library), any container can be treated as a simple association, and entire libraries such as Java collections can be easily included in the association library.

PART 1: Example of the bottom-up approach

Let's assume that we have a company with Employees who are organized into a hierarchy of Departments. Each Employee belongs to exactly one Department, and each Department has one Manager. Departments have ID numbers, and Employees have names.

You start with a skeleton of your classes, without inserting any code related to the relations among them. The name is not treated as an attribute, but rather as a relation, therefore it does not appear in this code yet.

```
public class Employee {
    public ZZ_Employee ZZds;
    public Employee(){ZZds=new ZZ_Employee();}
}

public class Manager extends Employee {
    public ZZ_Manager ZZds;
    public Manager(){ZZds=new ZZ_Manager();}
```

```
}

public class Department {
    private int deptNo;

    public ZZ_Department ZZds;  // add mechanically to every class
    public Department(int dNo){
        ZZds=new ZZ_Department(); // add mechanically to every class
        deptNo=dNo;
    }
}
```

Then, separately and usually in one block of code, you declare the relations (associations) among the classes. The association library (jlib) gives you a multitude of choices but, as you will see, the initial choice isn't critical. It is easy to switch and experiment with different choices. Let's implement the data organization with these associations:

Note that since the purpose of jlib is to eliminate reference from your application classes, we strongly recommend to use data organization Name for all variable length strings (names):

and you declare the associations in a separate file (e.g. called *ds.def* for 'data structure definitions):

```
// declare the relations (schema) one association per line
Association LinkeList1&ltDepartment,Employee> empl;
Association LinkedList1&ltDepartment,Department> dHier;
Association SingleLink&ltDepartment,Manager> boss;
Association Name&ltEmployee> eName;
```

Note that the syntax is identical with how you declare associations in the C++ version (alib).

Using the methods which jlib gives you for the associations, you can build and manipulate your data organization. Most associations dealing with multiple objects have iterators. For example, here is a simple program which builds a tree with a few Departments and populates it with Employees. Note how each association (data structure) has a name (here empl, dHier, boss, eName) which is then used as an identifier when operating on the data structure.

```
// relations (schema) in file ds.def, one association per line
Association LinkeList1&ltDepartment,Employee> empl;
Association LinkedList1&ltDepartment,Department> dHier;
Association SingleLink&ltDepartment,Manager> boss;
Association Name&ltEmployee> eName;

package test1;
import jlibGen.*;
import java.io.*;
import java.util.*;

public class Department {
    private int deptNo;
    public ZZ_Department ZZds;
    public Department(int dNo){ZZds=new ZZ_Department(); deptNo=dNo;}
    public Department(){ZZds=new ZZ_Department(); deptNo=0;}

    // recursive print of departments and employees
    public void prt(int layer){
        // iterators are automatically provided
        empl_Iterator eIter=new empl_Iterator();
        dHier_Iterator hIter=new dHier_Iterator();
        Department d;
        Employee e;

        prtSpace(layer);
        System.out.print("dept=" + deptNo + " manager=");
        boss.target(this).prt();

        for(e=eIter.fromHead(this); e!=null; e=eIter.next()){
            prtSpace(layer+1);
            e.prt();
        }

        for(d=hIter.fromHead(this); d!=null; d=hIter.next()){
            d.prt(layer+1);
        }
    }

    public void prtSpace(int layer){
```

```
            for(int i=0; i&ltlayer; i++)System.out.print("   ");
        }

        public void debPrt(){
            int i; Department d;
            System.out.print(deptNo+" tail=");
            d=dHier.tail(this);
            if(d==null)i=0; else i=d.deptNo;
            System.out.print(i+" nextRing=");
            d=dHier.nextRing(this);
            if(d==null)i=0; else i=d.deptNo;
            System.out.println(i);
        }
    }

    package test1;
    import jlibGen.*;
    import java.io.*;
    import java.util.*;

    public class Employee {
        public ZZ_Employee ZZds;
        public Employee(String name){
            ZZds=new ZZ_Employee();
            eName.add(this,name);
        }
        public Employee(){
            ZZds=new ZZ_Employee();
        }
        public void prt(){System.out.println(eName.get(this));}
    }

    package test1;
    import jlibGen.*;
    import java.io.*;
    import java.util.*;

    public class Manager extends Employee {
        public ZZ_Manager ZZds;
        public Manager(String name){
            super(name);
            ZZds=new ZZ_Manager();
        }
        public Manager(){
            ZZds=new ZZ_Manager();
        }
    }

    package test1;
    import jlibGen.*;
    import java.io.*;
    import java.util.*;

    public class test1 {
        // main program can already create and manipulate the data
        public static void main(String[] args){
            Employee e; Manager m; Department d1,d2,d3,d4,d5;

            d1=new Department(100);
            m =new Manager("C.Black"); boss.add(d1,m);

            d2=new Department(110);    dHier.addTail(d1,d2);
            m =new Manager("A.Green"); boss.add(d2,m);
            e =new Employee("J.Fox");  empl.addTail(e,d2); // error, should be (d2,e)
            e =new Employee("K.Doe");  empl.addTail(d2,e);

            d3=new Department(120);    dHier.addTail(d1,d3);
            m =new Manager("B.White"); boss.add(d3,m);

            d4=new Department(111);        dHier.addTail(d2,d4);
            m =new Manager("B.Brown");     boss.add(d4,m);
            e =new Employee("S.Winter");   empl.addTail(d4,e);
            e =new Employee("I.Springer"); empl.addTail(d4,e);
            e =new Employee("B.Summers");  empl.addTail(d4,e);


            d5=new Department(112);        dHier.addTail(d2,d5);
            m =new Manager("G.Gray");      boss.add(d2,m); // error, should be (d5,m)
            e =new Employee("F.Beech");    empl.addTail(d5,e);
            e =new Employee("H.Oats");     empl.addTail(d5,e);

            // print the entire data by accessing its root, d1
            d1.prt(0);
        }
    }
```

In this case, the application classes form package *test1*. They also must import

automatically generated jlib classes from package *jlibGen*.

For example, if directory *jtut* has two subdirectories *test1* and *jlibGen*, where jtut\test1 stores the application classes (Department.java, Employee.java, Manager.java, test1.java) plus file ds.def, while jtut\jlibGen is reserved for the generated association classes, you can compile and run this program from directory jtut like this (file test1\tt.bat):

```
cd jtut
java -cp c:\incode\jlib  src.codegen test1\ds.def c:\incode\jlib\lib  jlibGen test1\import
javac -classpath .;test1;jlibGen test1\test1.java
java test1.test1
```

THE FIRST LINE calls the code generator, codegen. Codegen does not mangle your code, it only generates additional *.java files with the customized associations as you requested them in ds.def. Codegen needs 4 parameters:

- **file which declares all your associations (here ds.def),**
- **path to the library of association templates (always jlib\lib),**
- **directory to deposit the classes for requested associations,**
- **file describing the package/import statements for the generated classes.**

In this example, file *import* has only one line:

```
import test1.*;
```

THE SECOND LINE compiles all *.java files in directories jtut\jlibGen and jtut\test1 with the main() in file test1\test1.java.

THE THIRD LINE runs the test1 program and deposits the results into file test1\res.

If you call codegen with the -u option everything is the same, except that beside generating the requested associations codegen also generates the logic of the corresponding UML diagram and deposits it into file layout.inp (fixed name). Program *layout* then uses this file as input and generates file display.svg with a properly layed out UML class diagram. File display.svg can be viewed with most Internet browsers or using special utilities. On the accompanying CD, this is all prepared as file test1\ss.bat:

```
cd jtut\test1
dir *.java > srcList
cd ..
java -cp c:\incode\jlib  src.codegen -u test1\ds.def c:\incode\jlib\lib  jlibGen test1\import test1\srcList test1_UML
javac -classpath .;test1;jlibGen test1\test1.java
c:\incode\layout\layout -s param.txt layout.inp
java test1.test1
```

The UML class diagram includes not only associations but also inheritance. In order to recover this information, codegen must browse through all the source (all *.java files) and search for the syntax indicating inheritance. For this reason, when you call codegen with the -u option, you must provide a file which provides the location of all the source files (here srcList). You can create this file under DOS or Windows by using: dir *.java or, under UNIX, by: ll *.java. When option -u is used, the last parameter provides the title text for the UML diagram.

Program *layout* is an independent executable so it really does not matter in what language it has been written. The existing version is in C++ but, as a conversion exercise, we are also planning its Java version. Option -s is for generating the svg display file. The program requires two parameters:

- **File describing the screen size in pixels and the default fonts for your UML diagrams. This file can be either coded manually or generated automatically and then used for all displays on your computer (param.txt).**
- **File describing the logic of the UML diagram, usually file layout.inp generated by codegen.**

The generation of the UML layout is not a simple task because it does not have a clean, mathematical objective -- the result should be simple and esthetically pleasing -- and we had to combine various tricks from the electronic CAD to achieve this goal which, to a human mind, appears deceivingly simple.

In order to demonstrate how associations and intrusive data structures increase the safety of data structures, program test1 includes two typical errors. The first one is caught by the compiler, the second is caught at the run-time.

On the first attempt to compile by invoking tt.bat, you get a compiler error indicating that, on line 16, addTail(e,d2) has wrong parameter types. The method has types addTail(Department,Employee) and not addTail(Employee,Department). Note that

similar type checking is NOT available for Java language containers which compile and then mysteriously crash under similar circumstances.

After you correct the error, the program compiles, but when it runs it prints

```
boss.add() error: already has a link
```

then it continues running and eventually crashes. You don't have to search for the reason of the crash. The message tells you that you attempted to add a boss twice to the same department. This type of error checking is NOT available in any existing Java or C++ container library.

After correcting the second line marked with the 'error' comment, the program runs and prints correct results:

```
dept=100 manager=C.Black
   dept=110 manager=A.Green
      J.Fox
      K.Doe
      dept=111 manager=B.Brown
         S.Winter
         I.Springer
         B.Summers
      dept=112 manager=G.Gray
         F.Beech
         H.Oats
   dept=120 manager=B.White
```

WARNINGS:

(1) If the first error were empl.addTail(m,e), a good Java compiler would catch the type incompatibility. Unfortunately, Microsoft J++ does not catch that one and then the program crashes without giving you any clues what is wrong. The problem is in the compiler, not in the jlib library.

(2) If you forward the results into a file, for example

```
java test1.test1 > test1\res
```

you do not get the run-time error message. The message is printed into file res which is then scrapped when the program crashes.

Note that even if the program does not compile, you already get the UML diagram which you can view by invoking ss.bat and then going to display.svg with your Internet browser:

Introducing changes

As the next exercise, let's see how easy it is to change the data organization. The existing organization has two disadvantages: (a) For a given emloyee, we do not have an easy way to find all his/her superiors. (b) When looking for the employee with a given name, we have to traverse the tree of all the departments, which for a company with thousands of employees may take a long time.

Let's make the following changes: We'll introduce a class Company representing the entire company, and in addition to departments holding employees, we'll make the Company to keep a hash table of employees. We will also replace both linked lists by aggregates. The current version of aLib has only doubly-linked Aggregate2. This will allow us to search up through the tree of the departments:

The implementation of these changes leads to only a few simple modifications of the code (program test2.cpp):

```
public class Company {
    public ZZ_Company ZZds;
    public Company(){ZZds=new ZZ_Company();};
};

Association Aggregate2&ltDepartment,Employee> empl;
Association Aggregate2&ltDepartment,Department> dHier;
Association DoubleLink&ltDepartment,Manager> boss;
Association Name&ltEmployee> eName;
Association Hash&ltCompany,Employee> eHash;
Association SingleLink&ltCompany,Department> root;
```

If you invoke test2\tt.bat, the program compiles and runs with the same results as before. Since the class Company and the hash table of the the Employees are not used, it does not matter that we have not specified the functions required for hashing. The new UML diagram is produced regardless whether the compiler finds errors or not:

Let's now evolve our program test2 by connecting the root of all the Departments to a Company object and by storing Employees in the hash table supporting fast searches for employees by their names. We also want to be able to find, for a given employee, all his or her superiors. When writing this code, I discovered that the data must support traversal of Departments and Employees in both directions, which means that the two LinkedList1 data structures must be replaced by Aggregate2 where each child knows its parent. Also, I found that it would be handy to replace the SingleLink 'boss' by a DoubleLink with the same name.

Below is the new code as stored in directory test3. The relatively few blue changes are the result of changing the data organization -- they were easy to implement because the compiler told me what had to be modified and why. The green changes implement the new features, in other words they are add-ons to the original program.

Note the two new methods for the Employee class, eHash_hash() and eHash_equals() as required for the hash table. Unless you want to use your own hashing algorithm, these funtions just pick up the default from class eHash:

```
// File ds.def declares the relations (schema), one association per line

Association Aggregate2&ltDepartment,Employee> empl;
Association Aggregate2&ltDepartment,Department> dHier;
Association DoubleLink&ltDepartment,Manager> boss;
Association Name&ltEmployee> eName;
Association Hash&ltCompany,Employee> eHash;
Association SingleLink&ltCompany,Department> root;
```

```
package test3;
import jlibGen.*;
import java.io.*;
import java.util.*;

public class Company
    {
    public ZZ_Company ZZds;
    Employee eTemp; // temporary Employee object for algorithms
    public Company(){
        ZZds=new ZZ_Company();
        eHash.form(this,1000); // form hash table with given num.of buckets
        eTemp=new Employee();
    }

    public void prtSuperiors(String emplName){
        Employee ee,e; Department d;

        eName.add(eTemp,emplName); // pass name through a temporary object
        ee=eHash.get(this,eTemp);
        eName.remove(eTemp); // re-initialize temporary object
        if(ee==null){
            System.out.println("employee=" + emplName + " not in the company");
            return;
        }
        d=ee.myDept();
        if(d==null){
            System.out.println("employee=" + emplName + " not assigned to a department");
            return;
        }
        if(boss.target(d) == ee)System.out.print("\nmanager ");
        else                    System.out.print("\nemployee ");
        System.out.print(emplName + " superiors: ");
        for(; d!=null; d=dHier.parent(d)){
            e=boss.target(d);
            if(e==ee) continue;
            System.out.print(eName.get(e) + "(" + d.getDeptNo() + ") ");
        }
        System.out.println();
    }
}
```

```
package test3;
import jlibGen.*;
import java.io.*;
import java.util.*;

public class Department {
    private int deptNo;
```

```
    public ZZ_Department ZZds;
    public Department(int dNo){ZZds=new ZZ_Department(); deptNo=dNo;}
    public Department(){ZZds=new ZZ_Department(); deptNo=0;}

    // recursive print of departments and employees
    public void prt(int layer){
        // iterators are automatically provided
        empl_Iterator eIter=new empl_Iterator();
        dHier_Iterator hIter=new dHier_Iterator();
        Department d;
        Employee e;

        prtSpace(layer);
        System.out.print("dept=" + deptNo + " manager=");
        boss.target(this).prt();

        for(e=eIter.fromHead(this); e!=null; e=eIter.next()){
            prtSpace(layer+1);
           e.prt();
        }

        for(d=hIter.fromHead(this); d!=null; d=hIter.next()){
            d.prt(layer+1);
        }
    }

    public void prtSpace(int layer){
        for(int i=0; i&ltlayer; i++)System.out.print("   ");
    }

    public void debPrt(){
        int i; Department d;
        System.out.print(deptNo+" tail=");
        d=dHier.tail(this);
        if(d==null)i=0; else i=d.deptNo;
        System.out.print(i+" nextRing=");
        d=dHier.nextRing(this);
        if(d==null)i=0; else i=d.deptNo;
        System.out.println(i);
    }

    public int getDeptNo(){return deptNo;}
}

package test3;
import jlibGen.*;
import java.io.*;
import java.util.*;

public class Employee {
    public ZZ_Employee ZZds; // add mechanically to every class
    public Employee(Company co,String name){
        ZZds=new ZZ_Employee();
        eName.add(this,name);
        eHash.add(co,this);
    }

    public Employee(){ // employee without a name
        ZZds=new ZZ_Employee();
    }


    public void prt(){System.out.println(eName.get(this));}

    public Department myDept(){
        return empl.parent(this);
    }

    // Methods required for the hashing
    // -------------------------------------
    public int eHash_hash(int hashSz){
        String s;
        s=eName.get(this);
        return eHash.hashString(s,hashSz); // pick up default
    }
    public boolean eHash_equals(Employee e){
        String s1=eName.get(this);
        String s2=eName.get(e);
        return s1.equals(s2); // compare the two names
    }
}

package test3;
import jlibGen.*;
import java.io.*;
import java.util.*;

public class Manager extends Employee {
    public ZZ_Manager ZZds;
```

```
    public Manager(Company co,String name){
        super(co,name);
        ZZds=new ZZ_Manager();
    }
}

package test3;
import jlibGen.*;
import java.io.*;
import java.util.*;

public class test3 {
    // main program can already create and manipulate the data
    public static void main(String[] args){
        Employee e; Manager m; Company co;
        Department d1,d2,d3,d4,d5;

        co=new Company();
        d1=new Department(100);       root.add(co,d1);
        m =new Manager(co,"C.Black"); boss.add(d1,m);

        d2=new Department(110);       dHier.addTail(d1,d2);
        m =new Manager(co,"A.Green"); boss.add(d2,m);
        e =new Employee(co,"J.Fox");  empl.addTail(d2,e); // (e,d2) is wrong, should be (d2,e)
        e =new Employee(co,"K.Doe");  empl.addTail(d2,e);

        d3=new Department(120);       dHier.addTail(d1,d3);
        m =new Manager(co,"B.White"); boss.add(d3,m);

        d4=new Department(111);           dHier.addTail(d2,d4);
        m =new Manager(co,"B.Brown");     boss.add(d4,m);
        e =new Employee(co,"S.Winter");   empl.addTail(d4,e);
        e =new Employee(co,"I.Springer"); empl.addTail(d4,e);
        e =new Employee(co,"B.Summers");  empl.addTail(d4,e);

        d5=new Department(112);           dHier.addTail(d2,d5);
        m =new Manager(co,"G.Gray");      boss.add(d5,m); // (d2,m) is wrong, should be (d5,m)
        e =new Employee(co,"F.Beech");    empl.addTail(d5,e);
        e =new Employee(co,"H.Oats");     empl.addTail(d5,e);

        // print the entire data by accessing its root
        root.target(co).prt(0);

        // print the superiors of H.Oats
        co.prtSuperiors("H.Oats");
    }
}
```

The result of running this program is:

```
dept=100 manager=C.Black
   dept=110 manager=A.Green
      J.Fox
     K.Doe
      dept=111 manager=B.Brown
         S.Winter
         I.Springer
         B.Summers
      dept=112 manager=G.Gray
         F.Beech
         H.Oats
   dept=120 manager=B.White

employee H.Oats superiors: G.Gray(112) A.Green(110) C.Black(100)
```

PART 2: Example of the top-down approach

Designing software top-down means that we start with vague ideas and work with a model such as the UML class diagram without writing much code. Only when we think that our model (the architecture) is more or less right, we gradually begin to fill in the code.

This of course does not mean that the initial model remains without changes. As the implementation proceeds, new conditions and problems pop up, and the model must change, often quite significantly.

The existing UML tools give you a graphical environment in which you can design and change UML models. They also give you code generators which, from a given UML diagram, generate code which "implements" the architecture. I used quotes for the word *implements* because they only generate a rough skeleton which you often must modify by hand.

The big problem with the existing tools is that they try to match UML associations with

container based data structures such as provided by Java language or C++ class libraries such as STL. Since the two concepts - associations and containers - do not match, the tools can generate only a rough code and cannot properly support model evolution. In particular, in some situations, it is impossible to retrieve the UML information automatically.

For example, assume that programmers who are implementing the software add 3 pointers and 2 collections to the model. Unless you know their intentions it is impossible to guess whether this is a new, complex association or just 5 simple ones, or perhaps only expansion (change) of some existing associations. For more explanation see the book *Next Software Revolution*.

Here is an example of how IN_CODE modelling eliminates these problems. Let's assume that we have a warehouse which stores parts required for the manufacturing of several different products. The parts are identified by their ID number (their bar code), the products are identified by their names.

Right away, we see that we need 3 basic entities which should be represented as classes: Warehouse, Part, Product - with one-to-many relations between Warehouse and Part and between Product and Part. Instead of wasting time on playing with a graphical tool, you simply describe this model in a few lines of text.

Note that at this stage of the design we do not care much about how the associations are implemented, and we use general associations such as *uni-directional one-to-many*" called Uni1toX in jlib.

If you have read Part 1 (bottom-up approach), you already know what the various parts of this code mean:

```
// File ds.def declares the relations (schema), one association per line

Association Uni1toX&ltWarehouse,Part> stored;
Association Uni1toX&ltProduct,Part> needed;
Association Name&ltProduct> prodName;
```

```
package test4;
import jlibGen.*;
import java.io.*;
import java.util.*;

public class Warehouse {
    public ZZ_Warehouse ZZds;
    public Warehouse(){ZZds=new ZZ_Warehouse();}
}
```

```
package test4;
import jlibGen.*;
import java.io.*;
import java.util.*;

public class Part {
    public ZZ_Part ZZds;
    public Part(){ZZds=new ZZ_Part();}
}
```

```
package test4;
import jlibGen.*;
import java.io.*;
import java.util.*;

public class Product {
    public ZZ_Product ZZds;
    public Product(){ZZds=new ZZ_Product();}
}
```

```
package test4;
import jlibGen.*;
import java.io.*;
import java.util.*;

public class test4 {
    public static void main(String[] args){
    }
}
```

Then you invoke this little batch file (tt.bat)

```
dir *.java > srcList
cd ..
del jlibGen\*.java
del jlibGen\*.class
del test4\*.class
```

```
java -cp c:\incode\jlib  src.codegen -u test4\ds.def c:\incode\jlib\lib  jlibGen test4\import test4\srcList test4_UML
copy layout.inp test4
javac -classpath .;test4;jlibGen test4\test4.java
java test4.test4
cd test4
```

Besides compiling the program which so far does nothing, tt.bat generates file layout.inp which can be instantly converted into the UML class diagram by calling ss.bat and then looking at layout.svg by your favourite browser:

Note that even if you do not have *.java files for your classes yet, you still can derive the UML just from file ds.def without compiling the software (file test4\ttt.bat):

```
dir *.java > srcList
cd ..
del jlibGen\*.java
del jlibGen\*.class
del test4\*.class
java -cp c:\incode\jlib  src.codegen -u test4\ds.def c:\incode\jlib\lib jlibGen test4\import test4\srcList test4_UML
REM copy layout.inp test4
REM javac -classpath .;test4;jlibGen test4\test4.java
REM java test4.test4
cd test4
```

and then you generate the UML diagram by invoking test4\ss.bat (the diagram is the same).

If your design involves inheritance, you have to tell *codegen* about it by supplying *.java files at least for those classes that use inheritance. Without that, *codegen* does not know about the inheritance and cannot display it in the UML diagram.

Let's continue evolving our original design. After you discuss the UML diagram with your client, several issues come up:

- **The warehouse should keep the count of the parts currently in stock. Class Part should be renamed PartType and have an int member *count*.**
- **There should not be just parts, but also assemblies which combine parts and other, simpler assemblies.**
- **The current model does not provide access to individual products.**
- **The client forgot to tell you there is not just one warehouse but several of them.**
- **It would make sense to add one more class, Company, in order to encapsulate the entire problem.**

All this results in only a few changes (shown in red):

```
Association Uni1toX&ltCompany,Warehouse> warehouses;
Association Uni1toX&ltCompany,Product> products;
Association Uni1toX&ltWarehouse,PartType> stored;
Association Uni1toX&ltAssembly,PartType> assemble;
Association Uni1toX&ltProduct,PartType> needed;
Association Name&ltProduct> prodName;

public class PartType { // replaces class Part
    public ZZ_PartType ZZds;
    private int count;
    public PartType(int cnt){ZZds=new ZZ_PartType(); count=cnt;}
}

public class Company { // new class
    public ZZ_Company ZZds;
    public Company(){ZZds=new ZZ_Company();}
}

public class Assembly extends PartType { // new class
    public ZZ_Assembly ZZds;
    public Assembly(int cnt){ZZds=new ZZ_Assembly();}
}
```

Invoking ss.bat will give you the new UML diagram:

Note that what we have is already a solid code which we can compile and run. For example, if someone asks you how much memory is needed to store data for 3 warehouses, 60 products and 7,500 parts with 800 parts per product, you can easily create such data organization and measure the change in the Java free memory.

Note that this method of estimating the size of data is different from what you typically do in C++ (see the C++ tutorial) where you can simply multiply *sizeof(className)* by

the number of the objects of the given class.

Let's follow the Java approach:

```
public class test5 {
    public static void main(String[] args){
        int i,j,k; long m1,m2,spaceUsed;
        Company co; Warehouse wh; Product pd; PartType pt,pt0;

        Runtime rt=Runtime.getRuntime();
        rt.gc();
        m1=rt.freeMemory(); // collect garbage and get free memory

        co=new Company();
        pt0=new PartType(); // the same PartType reference for the space estimate

        for(i=0; i<3;i++){ // 3 warehouses
            wh=new Warehouse();
            warehouses.add(co,wh);
            for(k=0; k<7500/3; k++){ // the 7500 PartTypes equally divided
                pt=new PartType();
                stored.add(wh,pt);
            }
        }
        for(k=0; k<60; k++){ // 600 products
            pd=new Product("Product No.12");
            products.add(co,pd);
            for(j=0; j<800; j++){ // 800 PartType references per product
                needed.add(pd,pt0);
            }
        }

        rt.gc();
        m2=rt.freeMemory(); // collect garbage and get free memory
        spaceUsed=m1-m2;
        System.out.println("spaceUsed=" + spaceUsed);
    }
}
```

The result is 724840 bytes, very close to 717896 bytes required for the C++ version - see the C++ tutorial. Both results are only first rough estimates anyway. The program will evolve and with every new member or data structure objects will grow in size .

Let's assume that after you have done all that, the new requirements come in:

- **One of your colleagues points out that you should also monitor the suppliers and how many parts they deliver to individual warehouses. This means another class, Supplier.**
- **The client says that the company needs fast direct access to the number of individual parts, using the PartType ID as a key - in other words the Company must also keep a hash table of PartTypes.**

All this results in the following additions of our code. Note how compact and efficient our textual representation is.

```
...
public class Supplier { // new class
    public ZZ_Supplier ZZds;
    public Supplier(){ZZds=new ZZ_Supplier();}
}
Association Uni1toX&ltCompany,Supplier> suppliers;
Association Uni1toX&ltSupplier,PartType> supply;
Association Hash&ltCompany,PartType> partHash;
...
```

and when you invoke ss.bat in directory jtut\test6, you get the new UML diagram:

At this point one of the programmers who is present at the meeting asks: "... and could we calculate the orders to individual suppliers if we want to build n products of type x? "

The logic of this calculation is simple but it requires, for a given PartType, to know who is the supplier. This means that the association 'supply' must be re-defined as bi-directional:

```
Association Uni1toX&ltSupplier,PartType> supply; // old model
Association Bi1toX&ltSupplier,PartType> supply;  // new model
```

And the new UML diagram has a small change (no arrow) on the 'supply' link:

At this point you may want to proceed with the implementation and at any time, as you evolve the software, you can compile and test the features you just implemented. The progress is fast (rapid development), yet you are getting a solid, production-grade code. The UML diagram always perfectly matches the code, and major changes of the architecture are easily absorbed even on large volume of code. The association classes have been designed so that the compiler tells you where your code requires changes -- usually only in surprisingly few places.

Note that, from this moment on, there is no difference between the top-down and bottom-up approaches. You evolve both the code and the architecture. However, when using associations, they both are controlled by the textual UML model which is a part of your code (file ds.def).

Gradually, you will replace the generic Associations (Uni1toX, Bi1toX,..) by more specific data structure such as Bag, LinkList2, Aggregate2 etc. Again this will result in no or only a few changes of your code. For example, if you re-define the data organization like this (see directory jtut\test6 files ds.alt and ttt.bat)

```
Association LinkedList1 warehouses;
Association LinkedList1 products;
Association Uni1toX stored;
Association Bag needed;
Association Bag assemble;
Association Name prodName;
Association LinkedList2 suppliers;
Association Aggregate2 supply;
Association Hash partHash;
```

The Java compiler tells you that you cannot use methods warehouses.add() and products.add() on lines 24 and 33 of test6.java. The reason is that for LinkedList1 you must use either addHead() or addTail() instead. That's all.

Regardless of the compiler errors, you still get the new UML diagram. Note that the association *stored* is still left as generic, without any instruction about its implementation:

PART 3: The library of data structures

For a detailed description of the currently available classes, see incode\jlib\doc\jClasses.doc.

IMPORTANT: Note that this library is well protected against a wide range of errors, and it has been designed for the maximum performance and ease of use. The penalty for all this is a bit of additional work when building the library itself.

3.1 Adding a new data structure to the library

This Chapter describes how to design a new library class (data structure or association) and add it either to your own library (myLib) or to the standard jlib library (jlib/lib). The library classes use parameters $$,$0,$1,$2 in a style similar to C++ templates or macros. These parameters allow the code generator (codegen) to create associations which are customized to the participating classes, just like the C++ compiler which expands C++ templates. However, codegen does a bit more -- it also generates special classes that bind together classes that form the association. For example in

```
public class Company {
    public ZZ_Company ZZds;
    Company(){ZZds=new ZZ_Company();}
}
```

class ZZ_Company is one of these transparently generated classes.

If you have experience with data structures and C++ templates, you may intuitively understand why parameters $$, $0, $1, $2 are needed and how you should use them. If your background is different and you find these parameters confusing, follow the slow route first. It introduces these parameters in a gradual, more logical way. After this detour. you can then return to this spot and continue reading. For additional help, look at the book "Next Software Revolution" by Jiri Soukup.

Note that general association describes a cooperation among several classes. However, the existing code generator (codegen) supports only associations which connect at most two classes. This restriction is not conceptual and will soon be removed.

As elsewhere in this tutorial, we will explain everything on a practical example. One of the most useful data structures which is missing in most existing class libraries is the LinkedList2 shown in the following diagram. It is an intrusive implementation of the one-to-many relation which also can be used as a uni-directional set. It is based on the doubly-linked list for which the operation remove() is very fast without changing the order of children. We will show how to code this data structure and add it to the library as MyLinkedList2.

Note that jlib has associations Ring2 and LinkedList2. Ring2 is a simple ring of children. LinkedList2 is derived from Ring2 and adds a parent to the ring. Association MyLinkedList2 which we are going to code will have the same functionality as LinkedList2, but its internal design will be different. It will not be derived from another class, it will be coded from scratch.

Before we start to code, remind yourself how the data structure will be used. Here are several examples.

```
Association MyLinkedList2&ltCompany,Product> products;
Association MyLinkedList2&ltCompany,Employee> employees;
Association MyLinkedList2&ltProduct,Component> assembly;
                          $1      $2           $$
```

The last line shows the three parameters which we will need for coding the generic form of the association:

- **$$ is the name of the association**
- **$1 is the name of the first class participating in the association, usually the more important one, called "parent" or "holder"**
- **$2 is the second class participating in the association, usually called "element" or "child".**

The Java code for MyLinkedList2 will include 4 classes:

- **MyLinkedList2, the data structure itself (its controls),**
- **MyLinkedList2Iterator which will help the user to traverse the list,**
- **MyLinkedList2Parent which will provide the data and references needed in the parent class,**
- **MyLinkedList2Child which will provide the data and references needed in the child class.**

The files with these classes will have the type *.jt and not *.java (for example MyLinkedList2.jt) because they are not regular Java files but special Java templates.

After you read the entire PART 3, you may learn more by analyzing other associations from jlib/lib. File *registry* is in particular useful when browsing through this directory.

Let's return to our MyLinkedList2, and start with MyLinkedList2Parent will provides the parent's reference to the tail of the children list.

FILE MyLinkedList2Parent.jt

```
package jlibGen;

public class $$_MyLinkedList2Parent {
    public $2 tail;
    public $$_MyLinkedList2Parent(){ tail=null; }
}
```

This code is easy to read and understand. The next class, MyLinkedList2Child, has pointers *next* and *prev* which implement the doubly linked ring

FILE MyLinkedList2Child.jt

```
package jlibGen;

public class $$_MyLinkedList2Child {
    public $2 next;
    public $2 prev;
    public $$_MyLinkedList2Child(){ next=prev=null; }
}
```

The previous two classes did not contain any logic, only the references implementing the data structure. On the other hand, class MyLinkedList2 includes no references, only methods which control the association:

FILE MyLinkedList2.jt

```
package jlibGen;

public class $$_MyLinkedList2 {

    public static void addHead($1 p, $2 c){
        $2 h, t;
        if(c.$0.next!=null){ ... error, child already linked ... }
        t=p.$0.tail;
        if(t==null){p.$0.tail=c.
                $0.next=c.$0.prev=c;}
        else {
            h=t.$0.next;
            c.$0.next=h; t.$0.next=c;
            c.$0.prev=t; h.$0.prev=c;
        }
    }

    public static void addTail($1 p, $2 c){
        if(c.$0.next!=null){ ... error, child already linked ... }
        addHead(p,c);
        p.$0,tail=c;
    }

    // insert element c1 before element c2, the tail does not change
    public static void insert($2 c1, $2 c2){
        $2 prv;
        if(c1.$0.next!=null || c1.$0.prev!=null){ ... error ... }
        if(c2.$0.next==null || c2.$0.prev==null){ ... error ... }
        prv=c2.$0.prev;
        prv.$0.next=c1; c1.$0.next=c2;
        c2.$0.prev=c1; c1.$0.prev=prv;
    }

    // Remove c from the list. This may change the tail.
    public static void remove($1 p, $2 c){
        $2 prv,nxt,t;
        if(c.$0.next==null || c.$0.prev==null){ ...error... }
        nxt=c.$0.next;
        prv=c.$0.prev;
        if(c==nxt)p.$0.tail=null;
        else {
            t=p.$0.tail;
            if(c==t)p.$0.tail=prv;
            prv.$0.next=nxt;
            nxt.$0.prev=prv;
        }
        c.$0.next=c.$0.prev=null;
    }

    // ...
}
```

For this associaiton, the iterator may allow to traverse the children in both directions but, for the sake of simplicity, let's implement only the forward traversal:

```
Association MyLinkedList2&gtProduct,Component< assembly;
Product p; Component c;
assembly_Iterator it=new assembly_Iterator();
...
for(c=it.fromHead(p); c!=null; c=it.next()){ ... } // forward traversal
```

Note that iterators usually store some temporary data and their methods are not(!) static:

FILE MyLinkedList2Iterator.jt

```
package jlibGen;

class $$_MyLinkedList2Iterator  {
    private $2 iTail; // null when loop finished
    private $2 nxt;  // null when starting a new loop

    public $$_MyLinkedList2Iterator() {iTail=nxt=null;}

    public $2 fromHead($1 p) {
        $2 ret;
        iTail=p.$0.tail;
        if(iTail==null)return null;
        ret=iTail.$0.next;
        if(ret==iTail)nxt=iTail=null;
        else nxt=ret.$0.next;
        return ret;
    }

    public $2 next(){
        $2 ret=nxt;
```

```
        if(ret==iTail)nxt=iTail=null;
        else nxt=ret.$0.next;
        return ret;
    }
```

By now, you probably see the pattern of how the $-parameters are used:

- **$$ is used only as a prefix for all the classes,**
- **S1 is used as the parent or the first participating class,**
- **$2 is used as the child or the second participating class.**
- **$0 is used as a prefix to all references in the parent or child classes.**

You probably agree with me that, with these parameters, the code is still quite readable and definitely less cluttered than C++ templates. Prefix $$ prevents the collision of names when two classes are connected by several associations of the same type:

```
Association Aggregate2&ltProduct,Component> assembly1;
Association Aggregate2&ltProduct,Component> assembly2;
Association Aggregate2&ltProduct,Component> assembly3;
```

This is similar to using multiple Java containers. For example:

```
public class Product {
    private Vector assembly1; // used as a vector of Components
    private Vector assembly2; // used as a vector of Components
    private Vector assembly3; // used as a vector of Components
    // ...
}
```

This situation causes no complications for simple containers but becomes an issue in a library like jlib which is type protected and supports bi-directional and intrusive associations.

The next step is to move the new classes to our new library, directory myLib. If you wanted to add the new association to the jlib library, you would add these files to directory jlib\lib. This would be rather pointless though since, as we explained, jlib/lib already contains association LinkedList2 which is more sophisticated than MyLinkedList2 in this simple example.

Note that, just like C++, Java will soon have templates. These templates will allow parametrization by type and could potentially replace jlib parameters $1 and $2. However, they will not replace parameters $$ and $0, and will not eliminate the need for the code generator (codegen).

After you move the four files to myLib, you also have to register the new association by creating file myLib\registry with the following line. (If you already have other classes in myLib, simply add this line anywhere in myLib\registry which is already there -- the order of the lines is not important):

```
u1-* MyLinkedList2&ltMyLinkedList2Parent,MyLinkedList2Child> Iterator;
```

In this example, the meaning of the first 4 characters is:

- **u = uni-directional association,**
- *** = the multiplicity of the source end is 'many',**
- **- = association of two classes,**
- *** = the multiplicity of the target end is 'many'.**

The first character can be 'u' or 'b' (b or bi-directional), or 'U' or 'B' if this association is a default for this association type.

If the association connects more then 2 classes, the registry record has two additional characters for each additional target class. For example

```
u14*u*u1 FSM&ltFSMholder,State,Input,TableElem>
```

where one FSMholder has access to many States and to many Inputs but only to one TableElem, yet neither of these know in which FSM they are used (u = uni-directions access for all the targets).

NOTE: In jlib Ver.2.0, the FSM class is not included yet. This example is based on class FSM from the Pattern Template Library (PTL), the library with classes that are generally easy to port to jlib.

A special registration code is used for commonly occuring many-to-many associations. This code has always only 4 characters:

```
R*n*
```

where *n* is the number of participating classes excluding(!) the relation class which is always listed as first. For example for

```
class Student{...};
class Course {...};
class IsTaking{
    int lastMark;
    ...
}

Association 2XtoX(IsTaking,Student,Course) isTaking;
```

the registration code is

```
        R*2* 2XtoX&ltIsTaking,Student,Course> Iterator;
```

The associations which we just discussed would be then displayed like this:

Let's now return to your new association and how we can use it in a program (this code is also available from directory doc\jtut\test7):

```
// file ds.def declares the relations (schema), one association per line

Association MyLinkedList2&ltProduct,Component> assembly;
```

```
package test7;
import jlibGen.*;
import java.io.*;
import java.util.*;

public class Component {
    private String cName;
    public ZZ_Component ZZds;
    public Component(String name){
        ZZds=new ZZ_Component();
        cName=name;
    }
    public void prt(){System.out.println("   "+cName);}
}
```

```
package test7;
import jlibGen.*;
import java.io.*;
import java.util.*;

public class Product {
    private String pName;
    public ZZ_Product ZZds;
    public Product(String name){
        ZZds=new ZZ_Product();
        pName=name;
    }
    public void prt(){System.out.println(pName);}
}
```

```
package test7;
import jlibGen.*;
import java.io.*;
import java.util.*;

public class test7 {
    public static void main(String[] args){
        Product p; Component c,cMot;
        assembly_Iterator it=new assembly_Iterator();

        p=new Product("bicycle");
        // load several colour to the list
        c=new Component("wheel");     assembly.addTail(p,c);
        c=new Component("pedal");     assembly.addTail(p,c);
        c=new Component("handlebar"); assembly.addTail(p,c);
        c=new Component("motor");     assembly.addTail(p,c);
        cMot=c; // remember the 'motor' component
        c=new Component("chain");     assembly.addTail(p,c);
        c=new Component("frame");     assembly.addTail(p,c);

        // remove the 'motor' from the component list
        assembly.remove(p,cMot);

        // print the resulting list
        System.out.print("Product: "); p.prt();
        for(c=it.fromHead(p); c!=null; c=it.next()){
            c.prt();
        }
   }
}
```

NOTE: The recommended way to deal with variable length names is to use the jlib association Name and not the String references (as used here, see cName and pName). The objective of jlib is to eliminate all user controlled references from the application classes. As an exercise, you may try to replace these two members by the association Name.

This code assumes that the application classes Product.java, Component.java, and test7.java are in directory jtut\test7 as package test7, while your new library with classes representing MyLinkedList2 are in jtut\myLib. and codegen is called to deposit the requested files to jtut\jlibGen. The following batch file compiles and runs this test from jtut (see jtut\test7\tt.bat):

```
del jlibGen\*.java
del jlibGen\*.class
del test7\*.class
java -cp c:\incode\jlib  src.codegen test7\ds.def myLib jlibGen test7\import
javac -classpath .;test7;jlibGen test7\test7.java
java test7.test7
```

If your new library class had an error -- and who can claim that all his/her code always works without debugging -- the compiler message will refer to generated *.java class in directory jlibGen and not to its generic *.jt form from directory myLib. For example, if file MyLinkedList2Iterator.jt variable ret has a wrong type

```
package jlibGen;

class $$_MyLinkedList2Iterator  {
    private $2 iTail; // null when loop finished
    private $2 nxt;  // null when starting a new loop

    public $$_MyLinkedList2Iterator() {iTail=nxt=null;}

    public $2 fromHead($1 p) {
        $1 ret; // error: $1 instead of $2
        iTail=p.$0.tail;
        if(iTail==null)return null;
        ret=iTail.$0.next;
        if(ret==iTail)nxt=iTail=null;
        else nxt=ret.$0.next;
        return ret;
    }

    // ...
}
```

The compiler tells you that in file jlibGen\assembly_MyLinkedList2Iterator.java on line 15 there is a mismatch of types between ret and next.

```
ret=iTail.ZZds.assembly.next;
                        ^
```

Without looking at directory jlibGen, you can go straight to myLib\MyLinkedList2Iterator.jt, and you will see that 'ret' should be declared as $2 and not as $1. If you want to avoid such indirect debugging, you can take the slow route. You first fully design and debug the association with specific classes, then replace them with parameters $$,$0,$1,$2, and the remaining errors are only a few and easy to find.

3.2 Deriving a new association from a simpler existing association

You guessed it correctly, now we are going to expand LinkedList2 into something else, and it will be Aggregate2, which is LinkedList2 where each child knows its parent. This is exactly how it is done in jlib\lib, and the code listings shown below are taken from there. (If you want to expand MyLinkedList2 to MyAggregate2, just change the names accordingly.)

We will have again four new files: Aggregate2Parent.jt, Aggregate2Child.jt, Aggregate2.jt, and Aggregate2Iterator.jt, plus one more file Aggregate2ParentAggregate2Child.jt for the situations when the same class is used both as the parent and as the child. We did not discuss this situation for MyLinkedList2, but LinkedList2 from jlib\lib uses such a class.

Note that LinkedList2 methods which work without a change do not have to be re-coded or even listed in Aggregate2.jt. This applies in particular to Aggregate2Iterator

which inherits all its methods from LinkedList2. Some methods of Aggregate2, for example remove(), have fewer calling parameters because each child knows its parent now.

FILE Aggregate2Parent.jt:

```
package jlibGen;

public class $$_Aggregate2Parent extends $$_LinkedList2Parent {}
```

FILE Aggregate2Child.jt:

```
package jlibGen;

public class $$_Aggregate2Child extends $$_LinkedList2Child {
    public $1 parent;
    public $$_Aggregate2Child() { parent=null; }
}
```

FILE Aggregate2ParentAggregate2Child.jt:

```
package jlibGen;

// the following class is used when Parent==Child
public class $$_Aggregate2ParentAggregate2Child
            extends $$_LinkedList2ParentLinkedList2Child {
    public $1 parent;
    public $$_Aggregate2ParentAggregate2Child() { parent=null; }
}
```

FILE Aggregate2.jt:

```
package jlibGen;

public class $$_Aggregate2 extends $$_LinkedList2 {
    public static $1 addHead($1 p, $2 c){
        if(c.$0.parent!=null){
            System.out.println("$$.addHead() error: c already in $$");
            return p;
        }
        c.$0.parent=p;
        $$_LinkedList2.addHead(p,c);
        return p;
    }

    // append Child c2 after Child c1 - different syntax from LinkedList2
    public static $1 append($2 c1, $2 c2){
        $1 p=c1.$0.parent;
        if(p==null){
            System.out.println("$$.append() error: c1=%d not in $$");
            return p;
        }
        if(c2.$0.parent!=null){
            System.out.println("$$.append() error: c2 is already in $$");
            return p;
        }
        $$_LinkedList2.append(p,c1,c2);
        return p;
    }

    // public static void insert($2 c1, $2 c2) ... derived from LinkedList2

    public static void remove($2 c){         // has a different syntax
        $1 p=c.$0.parent;
        if(p!=null) $$_LinkedList2.remove(p,c);
        else        System.out.println("WARNING: $$.remove(): c not in $$");
    }

    // ... and so on: some methods changed number of parameters,
    // some are internally modified, some are inherited without any modification.
}
```

The line which we have to add to the registry file is more complicated than it was for the association coded from scratch. It has to describe how Aggregate2 (and its parameters) are derived from the LinkedList2 (and its parameters). Character ':' is used to record inheritance, and base class parameters coded with $1,$2 refer to the parameters of the derived class.

```
b1-* Aggregate2 :LinkedList2<$1,$2> Iterator;
```

Note that if a class is passive (no references or data used by this association gets inserted into it) the parameter must have the '-' sign. Also, in some situations, the base class may be listed with one of the basic types such as int, void, etc. Here is two

examples from the existing library (jlib/lib/registry):

```
u1-* Array ;
u1-* Bag :Array<$1,$2> Iterator;
```

Here the Array refers only to ArrayElements, no data or references are injected into class ArrayElements. Bag inherits from Array but its second parameter, class BagElement is still passive (no injected data).

3.3 Converting a Java container to an association

Since containers are only special (simple) associations, converting a Java container to an association is only a matter of re-writing the interface. For example, class Vector1 from jlib/lib is just the Java Vector class with a slightly different interface and additional type protection:

```
// Traditional Java Vector
// ----------------------
class B {
    ...
};

class A {
public:
    vector&ltB> vec;
};

A a; B b;
  ...
a.vec.push_back(b);
```

```
Vector1 as an association
-------------------------
class B {
   ...
};

class A {
   ...
};
  Association Vector1&ltA,B> vec;

 A a; B b;
   ...
vec.push_back(a,b);
```

The three critical parts of the push_back() call are: a,b,vec. *Their order reflects your way of thinking depending on which approach you use.* When using the traditional Java style, a is first on your mind, then you think about vec as a part of A and, finally, you mentally add b to it. When treating the relation as an association, you first think about the model and the association vec and only then you decide which a and b will be involved. Here are the two classes which represent Vector1 in jlib\lib:

FILE Vector1Parent.jt:

```
package jlibGen;
import java.util.*;

class $$_Vector1Parent {
    public Vector vect; // Java Vector as a member
    public $$_Vector1Parent(){vect=null; }
}
```

FILE Vector1.jt:

```
package jlibGen;
import java.io.*;
import java.util.*;

class $$_Vector1 {

    static public void form($1 p){
        p.$0.vect=new Vector();
    }

    // Construct a vector containing the elements of the specified collection,
    // in the order they are returned by the collection's iterator.
    // ---------------------------------------------------------------------
    static public void form($1 p, Collection c){
        p.$0.vect=new Vector(c);
    }

    // Construct an empty vector with the specified initial capacity and
    // with its capacity increment equal to zero.
    // ---------------------------------------------------------------------
    static public void form($1 p, int initialCapacity){
        p.$0.vect=new Vector(initialCapacity);
    }

    // Construct an empty vector with the specified initial capacity and
    // capacity increment.
    // ---------------------------------------------------------------------
    static public void form($1 p, int initialCapacity, int capacityIncrement){
        p.$0.vect=new Vector(initialCapacity, capacityIncrement);
   }

    // Insert the specified element at the specified position in this Vector.
```

```
    // ------------------------------------------------------------------
    static public void add($1 p, int index, Object element) {
        p.$0.vect.add(index, element);
    }

    // Append the specified element to the end of this Vector.
    // ------------------------------------------------------------------
    static public boolean add($1 p, Object o) {
        return p.$0.vect.add(o);
    }

    // ... most methods of Vector are converted in the same style

    // Return a string representation of this Vector, containing
    // the String representation of each element.
    // ------------------------------------------------------------------
    static public String toString($1 p) {
        return p.$0.vect.toString();
    }

    // Trim the capacity of this vector to be the vector's current size.
    // ------------------------------------------------------------------
    static public void trimToSize($1 p) {
        p.$0.vect.trimToSize();
    }
}
```

WARNING: The jlib association Vector1 was mechanically derived from Java Vector without properly re-testing individual methods. The conversion was so simple and straightforward that the resulting class can be considered reasonably safe.

3.4 Expanding the Java Vector to a bi-directional association

Since we have converted Java Vector to an association (Vector1), we can derive the bi-directional association (Vector2) from Vector1. The main addition is that each element of the vector array must keep a reference to to the object which holds the array. Because of its intrusive nature, this data organization cannot be implemented as a Java container.

FILE Vector2Parent.jt:

```
package jlibGen;
class $$_Vector2Parent extends $$_Vector1Parent {}
```

FILE Vector2Child.jt:

```
package jlibGen;

class $$_Vector2Child extends $$_Vector1Child {
    public $1 parent;
    public $$_Vector2Child(){parent=null;}
}
```

FILE Vector2.jt:

```
package jlibGen;

class $$_Vector2 extends $$_Vector1 {
    public static $1 getParent($2 c){return c.$0.parent;}
    public static void addElement($1 p,$2 c){
        c.$0.parent=p;
        $$_Vector1.addElement(p,c);
    }

    // .. all other methods through the Vector1 (Java vector) interface
}
```