Data Structures, Patterns, and UML

3

Abstract

An essential part of every persistent system are persistent class libraries. Existing class libraries have two flaws: They cannot store bi-directional associations, and they do not treat associations (relations) as first class entities. We need a new paradigm for the proper design of these libraries. We will treat data structures as a database, and implement databases as data structures. The architecture will be controlled by a textual schema, not by the UML class diagram. However, this diagram will be automatically generated from the textual schema. This is just the opposite to what probably expect.

Keywords

Dataless class • Data structures • Class libraries • Associations • Relations • Design patterns • UML • Class diagram • Implementation • Intrusive • Array-based • Pointer-based • Composite • Flywheel • Finite state machine

Why do we include generic data structures in a book on persistent objects? Because, in both cases, the key issue is the safe and transparent handling of pointers.[1] If we design class libraries in the right way, persistency can be more efficient and easier to use. Also, the fact that existing "standard" libraries do not support bi-directional data structures is a disgrace—it complicates programming, makes it more error prone, contradicts UML thinking and often leads to code with inferior performance.

The idea that with the speed and storage capacity of modern computers we do not have to worry about performance is an urban myth. Look at the farms of computers Google is running, or at the problems with the human genome.

Let's establish a few basic facts about building data structures.

[1] In this chapter, we use the term *pointer* for both pointers and references.

J. Soukup and P. Macháček, *Serialization and Persistent Objects*,
DOI 10.1007/978-3-642-39323-5_3,

3.1 Basic Facts About Data Structures

Most of currently used generic containers are based on arrays, while pointer based data structures have been neglected. This section discusses the differences between the two approaches, and it shows what you can do with pointer chains, including sorting, merging, and splitting them.

We can build data structures with array or pointers.[2] Array-based data structures are essentially the 40-year-old Fortran technology which surprisingly still survives in relational databases. Its advantages are:

1. If the data do not change, they takes smaller space—no pointers are needed to create a list.
2. Such data are persistent, indexes are valid even when moving arrays do different addresses.

The disadvantages are severe, especially when the data structures change or grow:

3. If the arrays grow, we have to allocate a larger space than required, which takes away the advantage of the original smaller space.
4. Removing an element from the middle of the array has a major performance penalty.
5. When working with indexes, it is very easy to make an error. Data structures coded in this style are less reliable and harder to debug and maintain.

Object oriented programming combines functions (control) with data, adds inheritance, emphasizing individual objects and their access by pointers. It removes all the disadvantages of arrays but, at the same time, we lose the persistence.[3]

Pointer-based data structures often use lists, either singly or doubly linked. These lists can be either NULL-ended or form a ring. In Chap. 1 (Fig. 1.3) we established that rings are better because they permit inexpensive yet efficient integrity checking, but the for() loop traversing a ring is slightly more complex than the common `for(p=start; p; p=p->next){...}`

With a few exceptions, in this book we are always assuming that rings are used.

Many programmers are not aware that linked lists can be sorted with an efficiency comparable to qsort, and that the same algorithm can merge or split lists. We will explain the algorithms with examples. For full running code, look at bk\alib\lib\llist1.cpp (C++), bk\jlib\lib\llist1.jt (for Java), or bk\benchmark\objcLib\lib\llist1.m (Objective-C).

[2] Or with a combination of both, but let's keep it simple for now.

[3] Here we have another connection between persistence and data structures.

Algorithm: Sorting a List (Example)
27 3 2 3 5 8 12 7 19 30 6 3 80 79 13 22 40 1 11 2 41 31 32 39

Walking through the list and reversing descending sections[4] gives us the starting set of sorted sublists:

2 3 27, 3 5 8 12, 7 19 30, 3 6, 79 80, 13 22 40, 1 11, 2 41, 31 32 39

When neighbours decrease, it implies a new boundary section. There may be fewer sections now, for example 3 6 becomes automatically one section.

Walk through and merge subsequent pairs of sections

2 3 3 5 8 12 27, 3 6 7 19 30 79 80, 1 11 13 22 40, 2 31 32 39 41

Merging of two sections A and B is a linear process—a parallel walk through them and always selecting the smaller number. For example, for the first two sublists of the starting set

```
A: 2 3 27 result: 2  3 27 result: 2 3  27 result: 2 3 3  27 result: 2 3 3 5
B: 3 5 8 12          3 5 8 12          3 5 8 12          5 8 12

A: 27 result: 2 3 3 5 8  27 result: 2 3 3 5 8 12  27 result: 2 3 3 5 8 12 27
B: 8 12                  12
```

Repeat this until only one section is left

2 3 3 3 5 6 7 8 12 19 27 30 79 80, 1 2 11 13 22 31 32 39 41 49

1 2 2 3 3 3 5 6 7 8 11 12 13 19 22 27 30 31 32 39 41 49 79 80

Notes: Sort works perfectly for singly-linked list. We can sort doubly-linked list using only the `next` pointer. When finished, in one pass set the `prev` pointer—see Fig. 3.1.

Algorithm: Splitting or Merging Singly-Linked Rings The same algorithm can be used to merge or split rings—see Fig. 3.1. When a and b are in the same ring, the algorithm splits the ring into two. When a and b are in different rings, it splices (merges) them together.

```
// a and b are given elements
c=a->next;
d=b->next;
a->next=d;
b->next=c;
```

3.1.1 Working with Lists

It is important to understand the inside workings of lists in various libraries, as shown in Fig. 3.2.

Pointer Array corresponds to the use of STL Vector

```
class D {
    std::vector<L*> ptrArray; // declaration line
};

D* dp=new D;
...
L* lp=dp->ptrArr[17];
int sz=dp->ptrArr.size(); // size of the array
```

[4] The breaks between sections are shown as a comma here but not recorded during the calculation.

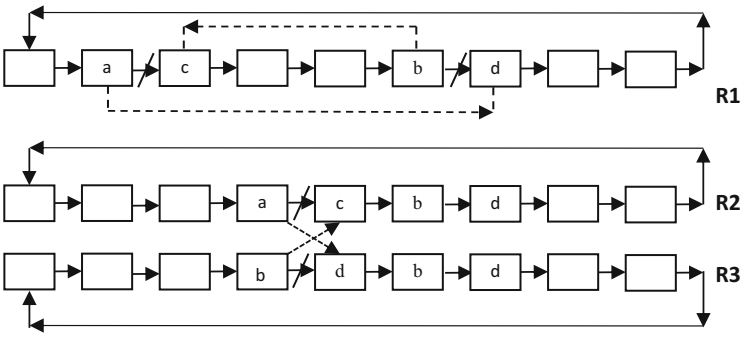


Fig. 3.1 The same reconnection of two pointers splits the ring into two when both objects are on the same ring (R1), or it combines two rings into one (R1 + R2) when each object is on a different ring

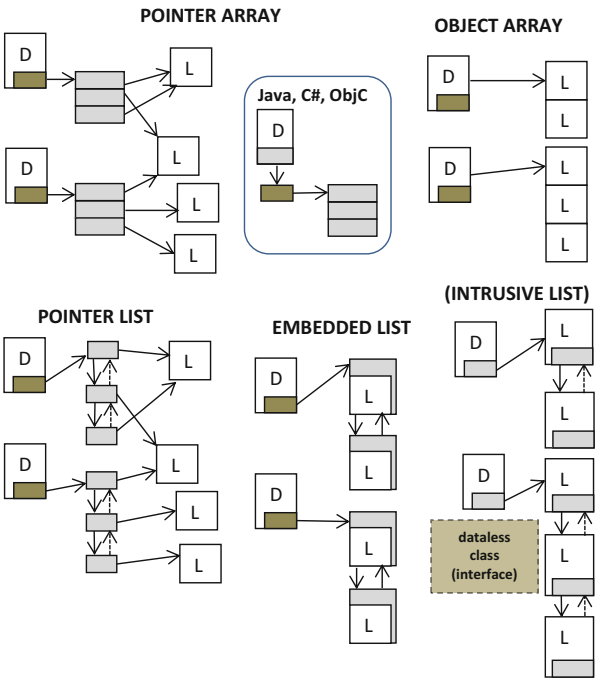


Fig. 3.2 Different implementations of lists. *White boxes* D and L are objects of application classes, *shaded boxes* are objects of library classes. *Lightly shaded boxes* store pointers and other data required for the implementation. *Darker boxes* provide the implementation of the List interface. Intrusive List has its interface implemented in a separate, dataless class. This data structure representation is used only in Code Farms libraries (DOL and InCode)

Here ptrArray is a member of D and contains both data (pointer to the array, its size, etc.) and the implementation of the data structure interface. Nothing is inserted into L. The array (shaded in Fig. 3.2) is transparent to the user.

In other languages, the declaration line would be

```
L[] ptrArray;        // in Java
ArrayList<L> ptrArray; // a better Java implementation
NSMutableArray *ptrArray; //  in ObjectiveC
L[] ptrArray;       // in C#
List<L> ptrArray; // a better C# implementation
```

but those languages do not include an instance of the ptrArray object, only a reference. Both the data and the controls become a separately allocated object—see Fig. 3.2.

Object Array corresponds to the use of STL Vector when the array stores entire L objects:

```
class D {
    std::vector<L> ptrArray; // declaration line
};

D* dp=new D;
...
L* lp= &(dp->ptrArray[17]);
int sz=dp->ptrArray.size(); // size of the array
```

This style of array is not allowed in Java or Objective-C. It could exist in C#, but only if L is a structure, not a class.

Pointer List corresponds to STL List using a reference

```
class D {
    std::list<L*> myList; // declaration line
};

D* dp=new D;
...
L* lp=dp->myList.front(); // head of the list
```

Objective-C library does not have a List class.[5] In Java and C# the declaration line would be:

```
LinkedList<L> myList; // in Java
LinkedList<L> myList; // in C#
```

[5] GitHub website offers one https://github.com/mschettler/NSLinkedList

Embeded List corresponds to STL List using an instance

```
class D {
    std::list<L> myList; // declaration line
};

D* dp=new D;
...
L* lp= &(dp->myList.front()); // head of the list
```

This is not a recommended type of list because the existence of the application object L is controlled by a transparent object that cannot be accessed by the application.

Intrusive List is often needed in real-life projects, but standard libraries do not provide it. Here is how you invoke it if you use the InCode library:

```
Association LinkedList2<D, L> myList; // declaration line
class D {
    ZZ_D ZZds;
};
class L {
    ZZ_L ZZds;
};

D* dp=new D;
...
L* lp= myList.head()); // head of the list
```

We will be discussing this library later in this chapter. The data structure is controlled by a self-standing dataless class, myList, which also inserts[6] pointers or other variables into participating classes D and L. The declaration line is the same in C++, Java, and in Objective-C.

Why do we claim that this type of list is often needed in real life projects? Look again at Fig. 3.2—Object Array and Embedded List are not suitable for practical use, and both Pointer Array and Pointer List are *collections*. L objects may appear several times under the same D or under several different Ds.

In real life, however, we more often encounter a *set*, where any L can appear only once: Department has Employees, and one Employee normally cannot be member of two Departments. Two customers cannot purchase the same TV screen with the same serial number. A train has ten cars. These cars cannot be, at the same time, in another train. A paragraph in a text document consist of lines. The same line cannot be there twice—it would need a different line number.[7]

Standard libraries have a *set* class, but because they only work with collections they have to do a lot of calculations to make sure that nobody can enter an object more than once. A set is either implemented as a balanced tree which requires O(log N) search for each insertion, or a hash table which needs a constant time but still has an overhead both in the space and time.

[6] How to perform this insertion is the prime subject of his chapter.

[7] If the same text appears twice, one of the lines is a copy of the other.

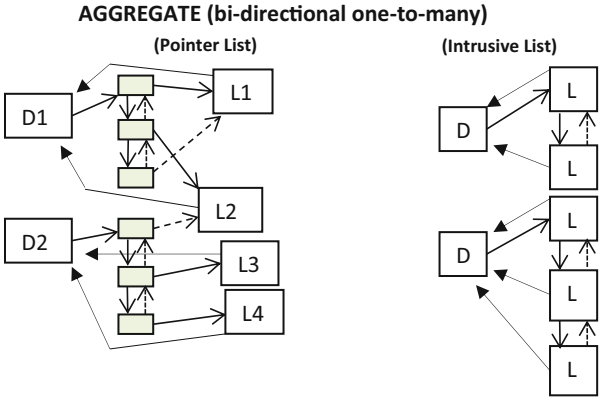


Fig. 3.3 Aggregate is a bi-directional *association*, even when built with a *collection* which is uni-directional. This mae disadvantage of this implementation is a possibility of an error: D2 thinks that L2 belongs to it, but L2 and D1 think that L2 belongs to D1. In Aggregate, a child can belong to only one parent. Intrusive Aggregate (on the right) is Intrusive List with an additional parent pointer

On the other hand, *Intrusive List* implemented with rings is a natural *set*, where a simple check whether a pointer is 0 tells you whether the object is free to be inserted.

If you find it surprising how the standard *set* is usually implemented, this is only a beginning. All the lists in Fig. 3.2 are *uni-directional associations*: from any D object you can reach its L objects, but these L objects do not know to what D object they have been assigned. In real life application about half the time we need associations that are bi-directional. Teachers have students, and they know who their teachers are. A person has several accounts, and the bank knows, for each account, to whom it belongs. The book may have several authors, and each author knows[8] what books he wrote.

And no standard library supports bi-directional associations! Is that possible, and why?

The reason is that the existing object-oriented languages do not provide a mechanism that would automatically insert the required pointers into more than one class. By definition, class L needs a pointer which would directly or indirectly lead to a D object—see Fig. 3.3.

The left part of Fig. 3.3 shows how using a *collection* (which is not a *set*) we can build *Aggregate* (which is a *set*). When adding an L-object, we check whether its parent pointer is NULL. If it is not, it is already member of this or another *Aggregate*, and must not be added second time.

Imagine that in Fig. 3.3, we are adding L-objects to D1 and D2 from the top down. When adding L1 the second time to D1 (dashed line), the operation is not accepted. Addition of L2 to D2 is also not accepted because L2 is already under D1.

[8] Unless he is very very old.

3.1.2 Separating Data and Interface

This section will lead you to a new style of representing data structures and associations, a style which is much better than the existing containers: Participating classes will store the required pointers or arrays, but the overall control—methods such as add() or sort()—will be in a separate dataless class.

Let's see what happens in the code when we implement Aggregate in the style shown on the left side of Fig. 3.3.

Listing 3.1 Implementing Aggregate with `std::list<T>`

```
#include <stdio.h>
#include <list>

template<class P,class C> class Aggregate : public std::list<C*> {
public:
     void add(P *p,C *c){
          if(c->myPar)return;
          std::list<C*>::push_back(c);
          c->myPar=p;
     }
        ...
};
// ------ library classes above this line, application below ------
class Lecturer;

class Department {
public:
   Aggregate<Department,Lecturer> lecturers; // <<<<<<<<
};

class Lecturer{
friend class Aggregate<Department,Lecturer>; // <<<<<<<<<
     Department *myPar; // <<<<<<<<<<
public:
     Lecturer():myPar(NULL){}
};

int main(){
    Department* dp=new Department;
    Lecturer* lp=new Lecturer;
    dp->lecturers.add(dp,lp);
    return 0;
}
```

Here Aggregate inherits std::list<> and with it both its interface and data—the pointer to the beginning of the list plus possibly the size of the list and other numbers. It also expects that pointer `myPar` has been inserted into Lecturer. Maintaining such data for possibly many data structures is a recipe for disaster.

Fortunately, if the application inserts `myPar` into a wrong class or uses a wrong name, the Aggregate will not compile, and the compiler tells you where the problem is. That makes this design style relatively safe though a bit tedious.

Besides the Aggregate being a public member of Department, this design has three flaws:

1. The fixed name `myPar` for the member inserted into Lecturer can cause a name collision. For example, if Lecturer can also be a Union member, we have

```
Aggregate<Department,Lecturer> lecturers;
Aggregate<Union,Lecturer> members;

class Lecturer {
    Department *myPar;
    Union      *myPar;
};
```

2. The call to the add() method at the bottom of Listing 3.1 requires `dp` to be mentioned twice, with a potential for introducing an error.
3. Declarations of data structures are spread through the classes, buried in them, and, especially with every class having a separate *.h file, there is no central place where you can see the overall architecture.

We will now address each of these issues separately.

Case 1. We need to use different names for `myPar`, or parametrize it by something typical for each case. For example, we could have

```
class Lecturer {
      Department *Department_Lecturer_myPar;
      Union *Union_Lecturer_myPar;
};
```

but that still may lead to a collision as in this situation

```
Aggregate(Company,Employee) employees;
Aggregate(Company,Employee) retirees;
```

The best parameter to use is the instance name of the data structures, such as

```
Department *lecturers_myPar;
Union *unionMembers_
Company *employees_myPar;
Company *retirees_myPar;
```

That sounds like a good idea, but templates (or generics) do not allow us to parameterize variable and member names. In C based languages we can use macros; in other languages we can use a preprocessor which would provide the substitution:

Listing 3.2 Implementing Aggregate with `std::list<T>` and using a macro for additional parameterization

```
#include <stdio.h>
#include <list>

#define Aggregate(P,C,X)                          \
class X##_Aggregate : public std::list<C*> {  \
public:                                           \
    void add(P *p,C *c);                          \
}

#define AggregateImplement(P,C,X)     \
void X##_Aggregate::add(P *p,C *c){  \
   if(c->X##_myPar)return;           \
   std::list<C*>::push_back(c);      \
   c->X##_myPar=p;                   \
}

// ------ library include above this line, application below ------
class Lecturer;
class Department;
Aggregate(Department,Lecturer,lecturers);

class Department {
public:
    lecturers_Aggregate lecturers;
};

class Lecturer {
friend class lecturers_Aggregate;
    Department *lecturers_myPar;
};

int main(){
   Department* dp=new Department;
   Lecturer* lp=new Lecturer;
   dp->lecturers.add(dp,lp);
   return 0;
}

AggregateImplement(Department,Lecturer,lecturers);
```

Note how the name `lecturers` becomes the ID of the data structure. This is still not the ideal way of doing it, only the first step.

In Listing 3.2, two characters ## are used to concatenate names. For example:

`A##B` creates `AB`.

Cases 2 and 3. The problem with relations spread through the classes is typical for the current way of designing software with STL and other container libraries, but it is still a major problem. Relations (associations, data structures) should have the same visibility in the code as classes. We need associations (relations) to become first class entities, as we have them in the UML class diagrams.

Both this and the problem with `dp` being mentioned twice can be fixed by separating data and interface. The data will reside in the application classes, and the interface will be implemented in a special, dataless class.

Rather than evolving the example based on `std:list<>`, this is easier to explain on the design of the intrusive Aggregate. Let's start with the Aggregate designed in the same style as the popular container classes, i.e. inserting an instance of Aggregate into class Department.[9] In order to make Aggregate generic, we will use a macro, because that is the only way to prevent collision of names.

Listing 3.3 is not easy to read, but it would be useful if you grasp its essence.

Note what happens in the application code. In the beginning you declare what Aggregate you are going to need. This statement creates two classes, `lecturers_AggregateChild` and `lecturers_Aggregate`. These are the types of members you add to classes Department and Lecturer. In both cases you name the member `lecturers`, which is the common name for everything associated with this Aggregate; see the call to add() in the main.

Statement `AggregateImplement(...)` which is at the end of the code contains implementation of all the methods the Aggregate needs. You could also compile it separately.

This implementation is already usable, and it solves two of the problems that we previously mentioned: It avoids the collision of names, and it declares Aggregate as a separate entity,[10] which is just as visible as the application classes.

The disadvantages are the massive use of macros which are always difficult to debug, the need to repeat the relation ID (here lecturers) many times, and the fact that `dp` occurs twice in the call to `add()`. Also we have not touched upon the issue that a library of data structures should derive more complex data structures from simpler ones.

The next improvement[11] of this code is in Listing 3.4. It removes the problem with `dp` and it makes the implementation much more logical. There are three classes now:

1. Aggregate—a dataless class which implements the Aggregate's interface, essentially providing the control of the data structure.
2. AggregateParent—which has no methods (it could even be just a structure) which must be inserted into the parent of the Aggregate, in this example into Department.
3. AggregateChild—which inserts the data into the child of the Aggregate, in this example into Lecturer.

Even though the library file aggregate.h is quite different, the application code remains practically the same (only the bold sections changed), and the problem with the double use of `dp` is solved. Note that this time we named the inserted members differently (_lecturers instead of lecturers). Since lecturers now has a global visibility, using the same name may lead to a collision.[12]

[9] This is not the aimplementation style we recommend for intrusive Aggregate, but watch what will happen.

[10] See the line marked with // <<<<.

[11] This is the implementation style we prefer and highly recommend.

[12] We tested that, in this example, using the same name works, but it a general case it may cause problems.

Listing 3.3 Intrusive Aggregate in the style of existing containers

FILE: aggregate.h

```
#define Aggregate(P,C,X)              \
class X##_AggregateChild {            \
friend class X##_Aggregate;           \
     P *par;                          \
     C *next;                         \
     C *prev;                         \
public:                               \
     X##_AggregateChild(){            \
          par=NULL; next=prev=NULL; \
     }                                \
};                                    \
class X##_Aggregate {                 \
     C *first;                        \
public:                               \
     void add(P *p,C *c);             \
     X##_Aggregate(){first=NULL;}     \
}

// list implemented as doubly-linked ring
#define AggregateImplement(P,C,X)          \
    void X##_Aggregate::add(P *p,C *c){    \
        C* f=first;                        \
         first=c;                          \
        c->_##X.par=p;                     \
        if(f){                             \
             c->_##X.prev=f->_##X.prev; c-_##_X.next=f; \
             f->_##X.prev->_##X.next=c; f->_##X.prev=c; \
        }                                              \
    else {first=c->_##X.next=c->_##X.prev=c;}          \
    }
\
```

APPLICATION:

```
#include <aggregate.h>
class Department;
class Lecturer;
Aggregate(Department,Lecturer,lecturers); // <<<<<<<<

class Department {
public:
     lecturers_Aggregate lecturers;
};

class Lecturer {
public:
     lecturers_AggregateChild lecturers;
};

int main(){
    Department* dp=new Department;
    Lecturer* lp=new Lecturer;
    dp->lecturers.add(dp,lp);
    return 0;
}

AggregateImplement(Department,Lecturer,lecturers);
```

Listing 3.4 Intrusive Aggregate with the separation of separated data and interface

```
FILE: aggregate.h

#define Aggregate(P,C,X)                    \
class X##_AggregateChild {                  \
friend class X;                             \
     P *par;                                \
     C *next;                               \
     C *prev;                               \
public:                                     \
     X##_AggregateChild(){                  \
          par=NULL; next=prev=NULL;         \
     }                                      \
};                                          \
class X##_AggregateParent {                 \
friend class X;                             \
     C *first;                              \
public:                                     \
     X##_AggregateParent(){first=NULL;}     \
};                                          \
class X {                                   \
public:                                     \
     static void add(P *p,C *c);            \
}

// list implemented as doubly-linked ring
#define AggregateImplement(P,C,X)           \
    void X::add(P *p,C *c){                 \
        C* f=p->X.first;                    \
        p->X.first=c;                       \
        c->X.par=p;                         \
        if(f){                              \
             c->X.prev=f->X.prev; c->X.next=f;   \
             f->X.prev->X.next=c; f->X.prev=c;   \
        }                                   \
        else {c->X.next=c->X.prev=c;}       \
    }

APPLICATION:

class Department;
class Lecturer;
Aggregate(Department,Lecturer,lecturers); // <<<<<<<

class Department {
public:
     lecturers_Aggregate Parent _lecturers;
};

class Lecturer {
public:
     lecturers_AggregateChild _lecturers;
};

int main(){
    Department* dp=new Department;
    Lecturer* lp=new Lecturer;
    lecturers::add(dp,lp);
    return 0;
}

   AggregateImplement(Department,Lecturer,lecturers);
```

3.1.3 Generalized Templates—Code Generator

Most of the parameterization we need for our data structures can be done with templates, but we also need to parameterize names of certain members—something that templates cannot do but a code generator can. However, if we already use the code generator, we can let it also to expand the templates, and that takes us to new, more general type of templates.

The main problem with Listing 3.4 is the use of macros. They are difficult to understand and debug. Even the simple examples in Listings 3.3 and 3.4 took us a while to debug.

Think what we need to do and what templates and macros provide. We want to parameterize the library classes with types of participating classes just as when you use templates. The only exception is that we also want to manipulate class and member names using one more additional parameter.

Think then what is the compiler doing with templates. Compilers first find for what parameters the templates are instantiated, and expand the templates. For example, if you have an error in

```
Template < class P,class C> class Aggregate {...}
```

and you are using this class for P = Department and C = Lecturer, the compiler tells you that you have an error in class

```
Department_Lecturer_Aggregate
```

After this, the compiler proceeds with the normal compilation.

We can do the same thing, but we can make it simpler and faster by tuning it to what we really need. We can use our expanded templates, and prepare the code for the compiler in the same way as the compiler prepares it with the normal templates. We will use a code generator but will not change the existing code. We will only create files with the expanded templates that can be compiled separately and linked with the application code. This is the method used in the InCode library today.

It uses the special keyword *Association*, which has the same effect as the // comment—the remaining code on this line is removed. However, the code which follows is the instruction for parameterization of templates. We will add this keyword to the lines that invoke the data structure. In Listings 3.3 and 3.4 those are the lines marked with // <<<<<<. Because these lines will be ignored by the compiler, we can change their syntax to look more like templates.

For example, for this original line

```
Aggregate(Department,Lecturer,lecturers);
```

we can use syntax

```
Association Aggregate<Department,Lecturer> lecturers;
```

which better portrays the meaning of this expression, except that `lecturers` is still the id of the interface, and method add() is static

```
lecturers::add(dp,lp)
```

see Listing 3.4.

The next question is how the code generator finds these special lines. You can feed it all the application code, but we recommend placing all these lines into a special small file, which in the InCode library is called ds.def (data structure definitions). The advantage of having them in a single file is not only simple processing. This file becomes a textual form of the UML diagram, the central place that stores the architecture. It further elevates the visibility of relations.

Inside the library, the templates are coded with parameters $$, $0, $1, $2, . . .

where $1, $2, . . . are the types of the participating classes, and $$ is the name of the association and $0 is the same as _$$. For example, in line

```
Association Aggregate<Department,Lecturer> lecturers;
```

we have $$ = lecturers, $1 = Department, $2 = Lecturer. As a shortcut, internally, $0 is used for _$$ or, in this example, for _lecturers. This makes the library encoding simple and readable. For example, the macros from Listing 3.4 become what are in Listing 3.5. The application code remains the same, except for the line starting with the *Association* keyword.

Listing 3.5 Intrusive Aggregate with generalized templates

FILE: aggregate.h

```
class $$_AggregateChild {
public:
    $1 *par;
    $2 *next;
    $2 *prev;
    $$_AggregateChild(){
        par=NULL; next=prev=NULL;
    }
};

class $$_AggregateParent {
public:
    $2 *first;
    $$_AggregateParent(){first=NULL;}
};

class $$ {
public:
    static void add($1 *p,$2 *c);
};

typedef $$_Aggregate $$;
#define Association /##/
```

FILE: aggregate.cpp

```
// list implemented as doubly-linked ring
void $$::add($1 *p,$2 *c){
    $2* f=p->$0.first;
    p->$0.first=c;
    c->$0.par=p;
    if(f){
        c->$0.prev=f->$0.prev; c->$0.next=f;
        f->$0.prev->$0.next=c; f->$0.prev=c;
    }
    else {c->$0.next=c->$0.prev=c;}
}
```

APPLICATION:

```
Association Aggregate<Department,Lecturer> lecturers;

// <<<< marks automatically inserted lines
class Department {
friend lecturers;                          // <<<<
   lecturers_AggregateParent _lecturers;   // <<<<
};

class Lecturer{
friend lecturer;                           // <<<<
   lecturers_AggregateChild _lecturers;    // <<<<
};

int main(){
    Department *dp; Lecturer *lp;
    lecturers::add(dp,lp);
```

Generated code is the normal C++ source which can be debugged as usual. The generated code (lecturers.h and lecturers.cpp) has the same number of lines as the library files (Aggregate.h and Aggregate.cpp) in this example. You can either debug lecturers.cpp and transfer each correction to Aggregate.cpp, or you can debug directly Aggregate.cpp.

Note that there is a simple mechanism which allows a quick manual conversion of DOL data structures to InCode.[13]

3.1.4 Transparent Insertion

Now when you are familiar with the Association statements, it is much easier to explain some things. For example, from this

```
Association Aggregate<Faculty,Student> students;
Association Aggregate<Faculty,Teacher> teachers;
Association Aggregate<Teacher,Course> courses;
Association Aggregate<Teacher,Student> advisorOf;
Association Aggregate<Student,Book> booksOnLoan;
Association ManyToMany<Student,Takes,Course> takes;
Association Name<Student> studentName;
```

you immediately see that we have classes Faculty, Student, Teacher, Course, Book and Takes, and you also see the data structures (Associations) that connect them. The only thing we have to add is that even a text string is represented as a special association, called Name.

This block of Association statements is a good example of how complex it would be to insert manually all the required parts. In this case, for example, class Student would look like this:

Listing 3.6 Insertion required for class Student[14]

```
class Student {
friend class students;
friend class advisorOf;
friend class booksOnLoan;
friend class takes;
friend class studentName;
      students_AggregateChild _students;
      advisorOf_AggregateChild _advisorOf;
      booksOnLoad_AggregateParent _booksOnLoan;
      takes_ManyToManySource _takes;
      studentName_Parent _studentName;
          ...    whatever else
};
```

[13] This was the way the InCode library was populated.

[14] This is just a code snippet, with no code on the website.

To handle this on large projects would be a nightmare, but don't worry. These insertions can be performed automatically and transparently. Chapter 5 will show a proposal how, by adding one keyword to existing object oriented languages, this could be done in a simple command. Until this proposal is accepted and implemented, however, we have to find some other way.

Since we were already forced to accept the use of code generator, and the code generator already analyzes the Association statements, we have a great opportunity there. The code generator can easily assemble these statements, and set them up so that Listing 3.6 is reduced to

```
class Student {
     ZZ_Student ZZds;
        ...    whatever else
};
```

This is exactly what the InCode generator does. ZZ prefix was chosen for historical reasons,[15] and *ds* stands for *data structure*.

The library expects that when entering a new association (data structure) you register it in a special registry file. For each association, there is one line which describes its design—see Fig. 3.4.

This record has a little more information than we discussed so far. InCode has two aggregates: Aggregate1 derived from the singly-linked list and Aggregate2 which is derived from the doubly-linked list. Otherwise, Fig. 3.4 is self-explanatory. The directionality and multiplicity record will be needed later for automatic generation of the UML class diagram.

The purpose of $1 and $2 parameters in this line is to connect parameters of the base and derived classes. For example, line

```
Association Aggregate2<Faculty,Student> students;
```

will trigger generation of files for `Aggregate2<Faculty,Student>`, but also for `List2<Faculty,Student>`, and the $.. parameters refer to the Aggregate2 definition. In this case $1 = Faculty, $2 = Student. There are situations where the base class has a different order or fewer parameters than the derived class.

If this appears too laborious, consider that this is done only once when entering the association into the library, and it is well worth it for the simple user interface, the prevention of errors in the application and the ability to generate the UML class diagram about which we will be talking later on.

Besides expanding the classes coded with the $ codes, the code generator also creates files gen.h and gen.cpp. File gen.h provides a mechanism which transparently inserts all the required members. This can have two forms: a macro or involving another level of indirection.

[15] All library-related expressions in DOL have prefix ZZ.

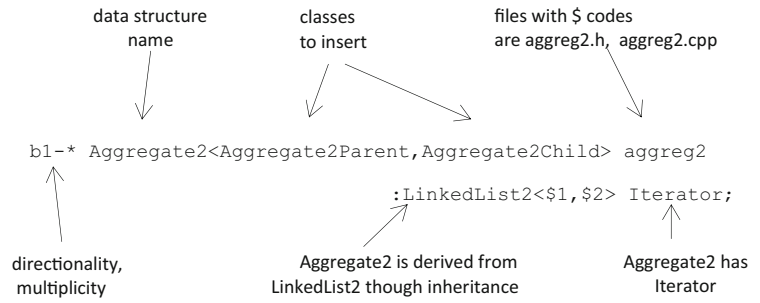


Fig. 3.4 Record of the Aggregate2 class in the InCode registry file

The DOL library uses a macro:

```
// FILE: gen.h
#define ZZ_EXT_Student
fri class students;                          \
friend class advisorOf;                      \
friend class booksOnLoan;                    \
friend class takes;                          \
friend class studentName;                    \
students_AggregateChild _students;           \
advisorOf_AggregateChild _advisorOf;         \
booksOnLoad_AggregateParent _booksOnLoan; \
takes_ManyToManySource _takes;               \
studentName_Parent _studentName;

// Application
class Student {
     ZZ_EXT_Student
        ... anything else
};
```

The InCode library uses an intermediary class:

```
// FILE: gen.h
class ZZ_Student {
    friend class students;
    friend class advisorOf;
    friend class booksOnLoan;
    friend class takes;
    friend class studentName;
    students_AggregateChild _students;
    advisorOf_AggregateChild _advisorOf;
    booksOnLoad_AggregateParent _booksOnLoan;
    takes_ManyToManySource _takes;
    studentName_Parent _studentName;
};

// Application
class Student {
public:
     ZZ_Student ZZds;
        ... anything else
};
```

In this case, the $0 code is not converted to

```
_associationName,
```

but to

```
ZZds._associationName
```

3.1.5 Big and Small, STL

Examples of how the data structures coded in this style are useful for projects of any size, and a discussion on how to convert STL classes into this representation.

Using this style, you can build data structures of any complexity and involving any number of classes in this style - each as easy to use as Aggregate or List that we just discussed. You can create these data structures using pointers and arrays, or you can use simpler data structures to build the more complicated ones. Details of how you do that are beyond the scope of this book, but you can grasp the main idea by analyzing classes from the InCode library.[16]

Since our objective is to remove all pointer members from the application classes, the library includes
SingleLink = equivalent of a single pointer, or uni-directional 1to1,
DoubleLink = two objects mutually linked via pointer, bi-directional 1to1,
Name = Null ending String attached to the object, an equivalent of char*.

Listing 3.7 shows how these classes are used.

Listing 3.7 Company–Manager–Employee example, InCode style

```
class Employee {
     ZZ_Employee ZZds;
     int phone;
     int salary;
};
class Manager : public Employee {
     ZZ_Manager ZZds;
};
class Company {
     ZZ_Company ZZds;
};

Association Aggregate<Company,Employee> employees;
Association Aggregate<Manager,Employee> subordinates;
Association Name<Employee> employeeName;
Association SingleLink<Manager,Employee> secretary;
```

[16] On the website at incode/alib/lib for C++, incode/jlib/lib for Java.

The important question is whether the InCode style library can include STL classes with their original interface, so that programmers who are used to them could still work with them while enjoying the benefits of the additional intrusive data structures.

As an example, let's look at how to represent `std::list<>` in the InCode style while keeping its original interface.

The standard way to use `std::list<>` to store Books in a Library is

```
class Library {
     std::list<Book*> myList;
};
```

In order to use the same class with our new interface, we will place `myList` in the same place, but we will get it there indirectly:

```
class NewListParent {
     std::list<Book*> myList;
};
class Library {
     NewListParent myPar;
};
```

This inserts not only the data that we want to be in Library, but also the STL interface—quite a bit of code which we don't want there but can tolerate it in this special case. Then we code the new dataless class—see Listing 3.8—which consists of short, usually one-line conversions of the old method to the new interface. Note that, compared to the STL interface, the new methods have typically one more parameter—a pointer to the class which holds the STL container. We can use the same method names as in STL, or replace them by new names.

This is more than just the Adaptor Design Pattern, because it combines insertion of data with the conversion of the interface.

Listing 3.8 stl_list coded in the InCode style, the concept of keeping the same interface (general idea, not a generic implementation yet)

```
class NewList { // stl_list, new style
     public:
          static void push_back(Library *lp, Book *bp)
               {lp->myPar.myList.push_back(bp);
          }
           ... // all other methods
     };
     int main(){
         Library* lp=new Library;
         Book* bp=new Book;
           NewList::push_back(lp,bp); // or add()
```

The generic $-encoding of `NewList` is simple and clean:

```
using namespace std;
class $$_NewListParent {
    std::list<$2*> oldList;
};
class $$_NewList {
public:
    static void push_back($1 *lp, $2 *bp)
                        {lp->$$.oldList.push_back(bp);}
    ... // all other methods
};
```

This conversion is safe[17] and works fine, but it is tedious because STL containers have quite many of methods.

Example: Airlines, Flights and Airports It's time to show a complete, more realistic example coded in this style. The code is in `bk\chap3\list_3-9.cpp`, `tt9.bat` compiles it, `rr9.bat` runs it.

[17] We did not make any changes in the code of stl_list.h.

Listing 3.9 Flights of different Airlines connect Airports, while distinguishing between the arrival and departure flights

```
#include "gen.h" // file generated by incode/codegen.exe
class Flight {
     ZZ_Flight ZZds;
     int flightNo;
};
class Airline {
     ZZ_Airline ZZds;
}; // attached name treated as a data structure
class Arrivals {
     ZZ_Arrivals ZZds;
};
class Departures {
     ZZ_Departures ZZds;
};
class Airport {
     ZZ_Airport ZZds;
     char code[4]; // 3-letter airport code
     Airport(){Departures* d=new Departures; toDept::add(this,d);
               Arrivals*   a=new Arrivals;   toArr::add(this,a);
     }
};

/* ++++++++++ next lines stored in file ds9.def ++++++++++++++++++
Association 3XtoX<Flight,Airline,Departures,Arrivals> flights;
Association DoubleLink<Airport,Arrivals> toArr;
Association DoubleLink<Airport,Departures> toDept;
Association Name<Airline> airlineName;
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++  */
int main(){
    Flight *fg; Airline *line; Arrivals *arr; Departures *dpt;
    Airport *dPort,*aPort;
    flights_Iterator it;
      ...
    // print all flights that depart airport 'dPort'
    dpt=toDept::fwd(dPort);
    for(fg=it.from2(dpt); fg; fg=it.next2()){
        line=flights::entity1(fg);
        arr =flights::entity3(fg);
        aPort=toArr::bwd(arr);
        printf("%s %d departs at %2d:%2d for %s\n",
             airlineName::get(line), fg->flightNo,fg->depTime/100,
             fg->depTime%100,aPort->code);
    }
    return 0;
}
Airport::Airport(){
     Departures* d=new Departures;  toDept::add(this,d);
     Arrivals*   a=new Arrivals;    toArr::add(this,a);
}
#include "gen.cpp" // generated by incode/codegen.exe
```

3.1.6 Code Generator and IDE

> Today, programming with a code generator is not considered a bad practice as it was a decade ago, but if you don't know how to set up your IDE properly, it can be a serious deterrent. This chapter describes how to set up a project so that the code generator is called automatically any time you recompile.

It used to be that using a preprocessor or a code generator was considered inappropriate. There were two practical reasons:

1. If preprocessor changed your code, debugging was difficult, especially when line numbers changed.
2. Integrated environments, such as Microsoft's Visual C++, did not integrate well with preprocessors and code generators.

Note also the difference between preprocessing and code generation. Preprocessor changes your original code, while code generator creates additional source files which compile separately and link to your original code.

Code generators are clearly better and they are frequently used today. Many programs quoted in this book use code generators but never a preprocessor.[18]

One of the reasons why code generators are not frowned upon any more is that development environments allow the programmer to register a code generator in such a way that you compile as if there were no code generation. The code generator is invoked automatically and transparently whenever you compile.

If you have never worked with a code generator, here is instruction on how to integrate the code generator for InCode library into VS2010.[19] Don't be discouraged by the fact that ten steps are required. We are trying to explain every detail so that even a complete beginner could do it. Also, remember that once you register the code generator, you compile as if it wasn't there.

1. We assume that the InCode library is stored in c:\InCode , with the code generator in c:\InCode\alib\codegen.exe[20] and the library in c:\InCode\alib\lib
2. Create a project, with or without any source files
3. Create file codeg.bat:

```
mkdir tmp
c:\InCode\alib\codegen.exe ds.def c:\InCode\alib\lib gen
```

   and move it to the directory where you have your new project.
4. In the same directory, create file environ.h, which may even be empty. You don't have to add this file to the project, but the generated files may need it.
5. At the top of main() you need

[18] With the exception of the built-in C preprocessor (macros).

[19] The steps are similar in VS2008, and probably in VS2012.

[20] Subdirectory alib is for C++, c:\InCode\jlib is for Java.

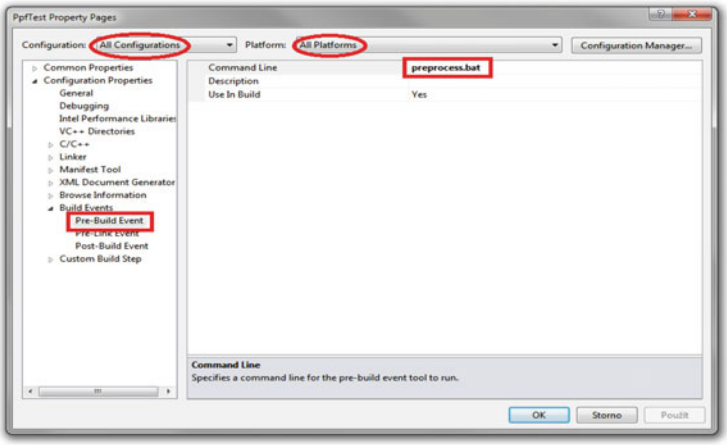


Fig. 3.5 Setting up VS2010 for a project using a code generator

#include "gen.h".
This file still does not exist; the code generator will create it.

6. Display *Properties* of your project by Alt + Enter, or right click in *the Solution Explorer* on your project, and then *Properties*.
7. Using the screen from Fig. 3.5, select *Configuration, All Configurations,* then *Debug* or *Release*.
8. Select *All Platforms*, as in Fig. 3.5.
9. Select Configuration Properties, Build Events, Pre-Built Event.
10. Select Command Line, type the name of the bat file you created, codeg.bat, and click OK.

From now on, you can compile as usual, but before every compilation, codeg.bat will be executed.

As we explained earlier, the full source for most of the examples is available as one large zip file. It unzips into directory bk, which is organized by chapter. More complex examples may be stored in separate subdirectories, with their own readme.txt and files tt.bat to compile and rr.bat to run it in the black CMD window.

As an example, let's take Listing 3.9 which uses the InCode code generator and library. The entire program is in `list3_9.cpp`, and the block of associations is in file `ds9.def`:

```
File tt.bat:
mkdir tmp
c:\incode\alib\codegen ds9.def c:\incode\alib\lib gen
cl list3_9.cpp

File rr.bat
list3_9
```

If you are working with Unix or Linux, you may prepare a makefile or bash file which invokes the code generator only if file `ds.def` has changed, e.g. if `ds.def` is younger than `get.h`. For example[21]

```
#!/bin/bash
#if tmp dir doesn't exist then create it
if [ ! -d tmp ]; then
   mkdir tmp
fi
run_cg=0
#if gen.h exists
if [ -f 'gen.h' ]; then
#if ds.def is newer then gen.h
   if [ ds.def -nt gen.h ]; then
     run_cg=1
   fi
else
   run_cg=1
fi
if [ $run_cg -eq 1 ]; then
 /opt/incode/alib/codegen ds.def /opt/incode/alib/lib gen
fi

gcc myprogram.cpp
```

3.1.7 Arrays (Vectors)

This section compares arrays of pointers with arrays of objects, and show how to build an Array class in the new style.

Arrays are important in many data structures, for example when building hash tables, and so far we have completely avoided them. As explained in Sect. 2.1.4, in order to make an array persistent, it has to be implemented through a special library class. In the DOL, InCode and PTL libraries this class is called Array and in the STL library it is called Vector. PPF includes class Vector as an example of how to make a STL class persistent. The array class usually represents a dynamic array,

[21] Full source bk\chap3\cg.sh.

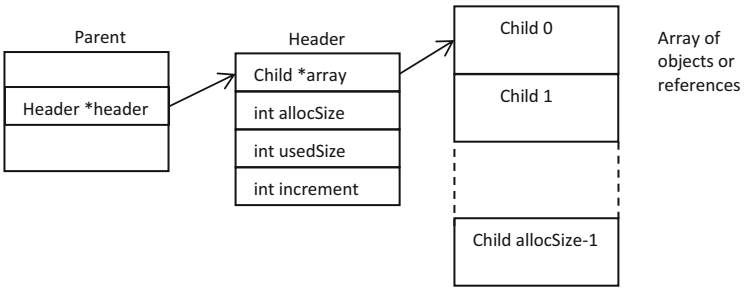


Fig. 3.6 Internal implementation of the Array class. In Java, arrays of objects are not allowed

which automatically increases its size when needed. In Objective-C, it is called MutableArray.

The InCode Array can be invoked in the same way as the pointer-based data structures:

```
Association Array<Parent,Child> name;
```

where Parent is the class to which the array is attached, and the Child is the type of object which forms the array. Child can be an entire object or just a pointer to it, for example:

```
Association Array<Library,Book> books; // array of Books
Association Array<Library,Book*> bookPtrs; // Book pointers
```

This association does not insert anything into the Child class, but it inserts a pointer to a special Header (or a pointer to such Header) into the Parent class— see Fig. 3.4. The Header stores a pointer to the array itself, and the size of the array. When making an array persistent, the Header object is stored to disk in the same way as the instances of the application classes.

The advantage of this arrangement is that, if there is no array attached to the Parent object, only one NULL pointer is wasted—see Fig. 3.6.

Here is an example of using the Array class. Method `form()` forms the initial array:

```
Array::form(Parent *par, int initialSize, int increment);
```

where increment $= 0$ specifies an array of fixed size, increment > 0 indicates how many items to add to the array if its size is not sufficient and increment < 0 gives a multiplication factor[22] to increase the size.

```
class Library {
     ZZ_Library ZZds;
     ...
};
class Book {
     ZZ_Book;
     int ID;
     ...
};
Association Array<Library,Book> books;
int main(){
     Library *lib; Book bk, *bp; int i;
     lib=new Library;
     // form array with initial size 10, increase 2x when needed
     books.form(lib,10,-2);
     for(i=0; i<32; i++){
          bk=books.get(lib,i); // equivalent of bk=a[i]
          bk.ID=300-i; // some number
          books.set(lib,i,bk); // equivalent of a[i]=bk
     }
     books.sort(lib);
```

It is much faster to get a pointer to the object inside the array and change the ID directly there, using function ind() which, for a given index, returns the pointer to the object- for example, DOL Array class has such a function. However, this function must be used with extreme care, and only before any other command accesses the array. If the array automatically reallocates, bp becomes invalid:

```
for(i=0; i<32; i++){
     bp=books.ind(lib,i); // equivalent of bk= &(a[i])
     bp->ID=300-i; // immediate use is OK
}
```

Can you figure out what happens with the `allocSize` and `usedSize` in this loop? When reaching i $= 10$ the array reallocates to 20, because the form() method specifies the multiplication factor of 2. Then when i $= 20$ it reallocates to 40. When the loop is finished, allocSize $= 40$ and usedSize $= 32$. When storing the array to disk, only 32 objects are stored.

[22] After discarding the negative sign.

3.1.8 Make Them Persistent

Data structures designed in the new style are easy to make persistent, and by moving all the pointers from application classes to library classes, the application classes also become persistent.

There are three important factors when thinking about persistence:

1. When we stick to the rule that application classes must not have any explicit pointer members, the application data are persistent as long as the library classes[23] are persistent.
2. Each data structure is represented by a dataless class. Since these classes keep no data, it is guaranteed that they do not keep any pointers, and these classes do not need any conversion to become persistent.
3. Remaining library classes are easy to make persistent because we know the pointers they store.

Classes storing the data to be inserted into the application classes are simple, because they typically include a few values and a constructor, no additional methods. As an example, to make Aggregate from Listing 3.5 persistent is to add default constructors to AggregateParent and AggregateChild as shown in Listing 3.10.

Listing 3.10 Making class Aggregate persistent

```
class $$_AggregateParent {
    $2 *first;
public:
    $$_AggregateParent(){PTR(first,$2);}
};
class $$_AggregateChild {
    $1 *par;
    $2 *next;
    $2 *prev;
    $$_AggregateChild(){PTR(par,$1); PTR(next,$2); PTR(prev,$2);}
};
```

When creating a new object with new(), unless other constructors are explicitly called, C++ calls default constructors for all base classes and their members. This guarantees that the PTR statements mark properly the position of all these pointers even in very complex composite objects—see example in Listing 3.11. This is specific in C++, but it does not work in Objective-C.

[23] Classes that represent data structures in the style described here.

Listing 3.11 Automatic invocation of default constructors in C++

```
#include <stdio.h>
class A {
    int a;
public:
    A(){printf("A\n");}
};
class B {
    A b;
public:
    B(){printf("B\n");}
};
class C : public B {
    int c;
public:
    C(){printf("C\n");}
};

int main(){
    C *c=new C;
    A *a=new A[5];
    return 0;
}
// It prints ABC and them AAAAA
```

For arrays of pointers we need to use macro `ARP()` as explained in Listing 2.17.

3.2 Inserting Pointers with Inheritance

Until now, the pointers and other variables that formed the data structures were alway inserted as members of participating classes. Interestingly, we can achieve the same objective with inheritance, at least in C++.

The basic idea of what we did so far was to have dataless class to represent the data structure and to implement its interface. Fpr example

```
template<class P,class C> Aggregate { ... };
```

implemented interface, and classes `AggregateParent` and `AggregateChild` stored the data to be inserted into the application classes as their members. We used macros for additional parameterization, but this is still the essence of what we did:

```
template<class P,class C> class AggregateParent{
    C *first;
};
template<class P,class C> class AggregateChild{
    P *parent;
    C *next;
    C *prev;
};
```

However, there is another way to insert data into a class—using inheritance.[24] The main idea is instead of coding as we did so far:

```
class Library {
        AggregateParent<Library,Book> _books;
        ...
    };
    class Book {
        AggregateChild<Library,Book> _books;
        ...
    };
```

to do this:

```
class Library : public AggregateParent<Library,Book> {
    ...
};
class Book : public AggregateChild<Library,Book> {
    ...
};
```

So far, it is not clear how we will access the data, but let's continue. It is clear that this approach will lead to a massive use of multiple inheritance, so it will be possible only in C++, not in Java, C# or Objective-C. For example, consider class Book which participates in three Aggregates—see Listing 3.12.

Listing 3.12 Class Book participating in three Aggregates

```
Association Aggregate<Library,Book> books;
Association Aggregate<Author,Book> published;
Association Aggregate<Book,Page> pages;

// that implies Book has to inherit from three classes:
class Book : public AggregateChild<Library,Book>,
             public AggregateChild<Author,Book>,
             public AggregateParent<Book,Page> {
    ...
};
```

[24] This is how the Code Farms' Pattern Template Library (PTL) works.

Considering that aggregate may be derived from LinkedList, and Link can be derived from Ring2, and Ring2 from Ring1, the use of inheritance is truly massive, we believe beyond what the creators of the language expected.

The advantage of this entire approach is that we don't need any parameterized names. The parameterization is by type. For example, the InCode style aggregate from Listing 3.5

```
class $$_Aggregate {
    void add($1 *p,$2 *c){
        C* f=p->$0.first;
        p->$0.first=c; c->$0.par=p;
        if(f){
            c->$0.next=f; c->$0.prev=NULL;
            f->$0.prev=c;
        }
        else {c->$0.next=c->$0.prev=NULL;}
    }
};
```

now becomes

```
template<class P,class C> class Aggregate {
    typedef AggregateParent<P,C>* pType;
    typedef AggregateChild<P,C>* cType;
    void add(P *p,P *c){
        C* f=(cType)p->first;
        (pType)p->first=c; (cType)c->par=p;
        if(f){
            (cType)c->next=f; (cType)c->prev=NULL;
            (cType)f->prev=c;
        }
        else {(cType)c->next=(cType)c->prev=NULL;}
    }
};
```

Without using the `typedef` statements, the long templates would make this code unreadable, but in this form it is crisp and manageable.

The code would work without casting with cType and pType, but if we don't cast, we open door to a potential error of using a member with the right name but from a wrong class.

There is still one situation though in which this design fails. If there are two associations of the same type, for example aggregates, between the same two classes, then the casting cannot differentiate between the two aggregates:

```
Association Aggregate<Company,Employee> employed;
Association Aggregate<Company,Employee> onVacation;
```

Fortunately, C++ templates allow an `int` parameter which can be used in such situations. The template is declared as

```
template<class P,class C,int i> class Aggregate { ...
```

and is normally used without the last parameter which is 0 by default:

```
Association Aggregate<Library,Book> books;
Association Aggregate<Author,Book> published;
Association Aggregate<Book,Page> pages;
```

but when two Aggregates connect the same classes, we use either

```
Association Aggregate<Company,Employee> employed;
Association Aggregate<Company,Employee,1> onVacation;
```

or

```
Association Aggregate<Company,Employee,1> employed;
Association Aggregate<Company,Employee,2> onVacation;
```

If we assemble manually the multiple inheritance statements such as in Listing 3.12, this library can be used without any code generator which is a definite advantage compared to the approach from Sect. 3.1. However, programming with the library is much easier if, in the fashion similar to the InCode approach, we place all Association declarations into one little file and let the code generator read it and create macros with the inheritance statements.[25] The application code from Listing 3.7 then looks like Listing 3.13:

Listing 3.13 Company–Manager–Employee example, PTL style (compare with Listing 3.7, which is in the InCode style)

```
class Employee : ZZ_Employee {
    int phone;
    int salary;
};
class Manager : ZZ_Manager, public Employee {
};
class Company : ZZ_Company {
};

// ----- either here or in a separate file ds.def ------------
Association Aggregate<Company,Employee> employees;
Association Aggregate<Manager,Employee> subordinates;
Association Name<Employee> employeeName;
Association SingleLink<Manager,Employee> secretary;
// -----------------------------------------------------------

int main(){
    Employee *e=new Employee;
    Manager *m=new Manager;
    secretary::add(m,e);
```

[25] The use of code generator in PTL is optional.

As we were writing this chapter, we realized that Code Farms missed a great opportunity to make PTL persistent. All that is needed is to include `PTR()`, `STR()` and `ARP()` statements into the default constructor. For example, adding the two bold lines shown in the following code makes the class Aggregate persistent in the PPF environment:

```
template<class P,class C, int i> class AggregateParent {
     C *first;
public:
     AggregateParent(){ PTR(first,C); }
};
template<class P,class C, int i> class AggregateChild {
     P *parent;
     C *next;
     C *prev;
public:
    AggregateParent(){ PTR(parent,P); PTR(next,C); PTR(prev,C); }
};
```

The only known library using this approach is Pattern Template Library[26] (PTL), which was coded as a proof that a generic library of intrusive data structures can be implemented without a code generator. The library has been available on Code Farms website since 1996, but was the only library never used on a serious, real-life project. It was originally designed as a framework for generic design patterns, and it has the following classes, some of them quite unusual and unique: *Aggregate, Array, Pointer Array, Collection,* pattern *Composite,* pattern *Flyweight* and *Finite State Machine* which can reset its settings while it is running. It would be relatively simple to transfer remaining classes from the InCode and DOL libraries to PTL, because these libraries are coded in the same style.

3.3 Library of Design Patterns

Structural design patterns are data structures which, besides pointers and arrays, also involve inheritance. This section shows how such patterns can be stored in a class library just like containers or other associations.

Christopher Wolfgang Alexander is an architect noted for over 200 building projects around the world. He was born in Austria, grew up in England and studied at Cambridge and Harvard. For many years, he taught at the UC Berkeley, and he is now retired in Switzerland.

[26] www.codefarms.com/ptl

In his search[27] for "quality without a name", he began to record patterns that made buildings pleasant to live in or around. Things such a layout of rooms, doors, windows and stairs, and their specifics depend on the climate and culture and interaction with objects around them. For example, when discussing a street café, we should consider the possible desires of the guests, the working environment of the café owner, but also the people who just walk by.

Alexander's patterns are catalogued in a uniform fashion. They consist of a short name, a rating, a sensitizing picture, the context description, the problem statement, a longer part of text with examples and explanations, a solution statement, a sketch and further references. Patterns recorded in this style are a great communication and teaching tool, and Alexander used them successfully when discussing his projects with the future occupants of his buildings.

Around 1994 a group of software designers began to develop software patterns that would be independent of the programming language and the application domain. The first conference on Pattern Languages of Program Design (PLoP) was in 1994,[28] and the "bible"[29] of this movement by the "Gang of four" was published the same year.

For example, the methodology we are using for bi-directional generic associations in this chapter could be considered a design pattern, and the following example shows the categories that should be recorded. In most situations, individual categories would be much longer; here we assume you understand the subject.

Name: Separation of interface from the attributes that form the data structure (or association).

Motivation: Containers buried in the application classes confuse the architecture, and do not allow the building of generic bi-directional associations as single entities.

Forces: Associations should have the same visibility and importance as application classes. Programming languages limit the implementation. Simple code generation is a practical solution, but is frown upon by purists.

Applicability: Any data structures, structural design patterns.[30]

Participants: The dataless class representing the data structure, and several application classes that store data. Instances of these classes get inserted into appropriate application classes.

[27] See his book "The timeless way of building", published by Oxford University Press in 1979.

[28] Proceedings edited by J.O. Coplien and D.C. Schmidt, and published by Addison-Wesley in 1995.

[29] Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns: elements of reusable object-oriented software. Addison-Wesley.

[30] The pattern part will be explained later in this chapter.

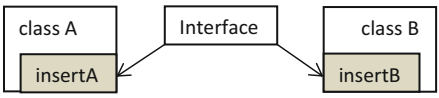


Fig. 3.7 Pattern "Separate interface and data of generic associations", where Interface has the same visibility as application classes A and B manages inserted data, usually pointers or references

Description: . . . <detailed description>
Diagram: see Fig. 3.7
Dynamic behaviour: This is a static pattern, but may be applied to dynamic data organizations such as FSM.
Implementation: . . . <detailed description>
Variants: Insertion can be either as members or through inheritance.
Consequences: Significant improvement in the clarity and quality of the software. Easier to maintain and evolve. UML class diagram matching implementation.
Limitations: Simple code generator is required.
See also: Separate interface pattern, Reflection in certain OO languages.
Sample code: Listing 3.10.
Known uses: This approach has been supported by Code Farms Inc. since 1989 and was successfully applied to hundreds of projects, some over 100,000 lines of code.

Structural design patterns are a special category of software patterns that can be considered an extension of the classical data structures by adding inheritance to the usual network of pointers and arrays. We have shown already in 1994[31] that these patterns can be implemented in a generic form and stored in a library with other data structures such as Aggregate or HashTable.

For an advanced reader, Listing 3.14 shows the complete implementation of the PTL pattern Composite. Composite is the mechanism that allows one to build a system from bigger and bigger parts. Listing 3.15 demonstrates how to create graphics from lines, text, pictures and smaller sub-designs, we build mechanical designs from plates, bolts, nuts and pre-built parts, and we design silicon chips from transistors, wires, and contacts that connect different layers of wiring. In these and many other applications we design hierarchically, creating larger and more complex designs from smaller and simpler ones—see Listing 3.14.

[31] Soukup J (1994) Implementing patterns. PLoP conference, pp. 395–412.

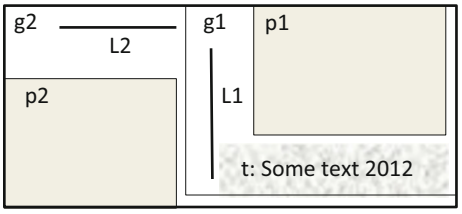
Listing 3.14 Internal implementation of class `Composite` in the PTL library. `Composite` is derived from `Collection`, which is equivalent to `Intrusive List`[32] from Fig. 3.2

```
// file composite.h in directory ptl\lib
// =====================================
template<class P,class C,int i> class CompositeChild :
                            public CollectionChild<P,C,i>{
};
template<class P,class C, int i> class CompositeParent :
                            public CollectionParent<P,C,i>{
};
template<class P,class C, int i> class Composite :
                            public Collection<P,C,i>{
    // all methods of Collection are inherited
};

#define CompositeInherit1(id,par,chi,i)                          \
    public chi, public CompositeParent<par,chi,i>

#define CompositeMember1(id,par,chi,i)                           \
    virtual int isComposite(Composite<par,chi,i> *c){ return 1;}

#define CompositeInherit2(id,par,chi,i)                          \
                                public CompositeChild<par,chi,i>

#define CompositeMember2(id,par,chi,i)                           \
    virtual int isComposite(Composite<par,chi,i> *c){ return 0;}

// file pattern.h produced by code generator ptl\mgr\mgr.exe
// =========================================================
#define pattern_Part \
  CompositeInherit2(comp,Graphics,Part,1) { \
  CompositeMember2(comp,Graphics,Part,1) PTL_COMMENT

#define pattern_Graphics \
  CompositeInherit1(comp,Graphics,Part,1) { \
  CompositeMember1(comp,Graphics,Part,1) PTL_COMMENT

//file mgr.h from ptl\lib that all application source includes
// =========================================================
#define Pattern(A) pattern_##A
#define PTL_COMMENT /##/
```

[32] If you want to traverse Composition both up and down, then deriving it from Aggregate would make more sense.

Listing 3.15 Applying Composite to a graphics design

```
class Part : Pattern(Part) {
     int x1,y1,x2,y2; // overall dimensions
};
class Graphics : Pattern(Graphics) {
};
class Line : public Part{ // see footnote[33]
};
class Text : public Part { // see footnote
     char *txt;
};
class Picture : public Part { // see footnote
  char *fileName;
};
Association Composite<Graphics,Part> parts;
int main(){
    int main(){
    Graphics *g1,*g1; Line *L1,*L2; Text *t; Picture *p1,*p2;

        ...
    parts.add(g1,L1);
    parts.add(g1,t);
    parts.add(g1,p1);
    parts.add(g2,g1);
    parts.add(g1,L2);
    parts.add(g1,p2);
```



Useful Trick No. 7

Macro `Pattern(Graphics)` **on line**

```
class Graphics : Pattern(Graphics) {
```

adds not only a base class; it also adds a virtual function to class Graphics(!)

```
#define Pattern(T) \
public Part{ZZ_##T; virtual int isComposite(...){...}; //
```

where the end comment wipes out the brace at the end of the original line:

```
class Graphics : public Part{ZZ_##T; virtual int isCompsite(...)
{...};//{
```

which really is

```
class Graphics : public Part{
ZZ_##T; virtual int isCompsite(...){...};
```

[33] If class Line participates in some other data structures or patterns, the statement would take this form:
`class Line : public Part, public Pattern(Line){`

WARNING:
Listing 3.15 can mislead you, because it does not show the key idea of how PTL can access its subparts. This is easier to explain on the Composite[34] class designed from scratch. Note that the types of internal pointers are CT* and PT*, and not C* and P* as one could expect:

```
typedef CompositeChild<P,C> CT;
typedef CompositeParent<P,C> PT;
template<class P,class C,int i> class CompositeChild{
    CT *next;
    PT *parent;
};
template<class P,class C, int i> class CompositeParent{
    CT *first;
};
template<class P,class C, int i> class Composite}
    C *nxt(CT *cp){return (C*)(cp->next); } // <<<<<<<<<<<<<
    ... other methods
};
Association Composite<Graphics,Part> parts;
int main(){
    Part *p; Text *t;
    p=parts.nxt(t);
```

The function finds the right subpart through automatic casting. It is as if writing

```
cp=(CT*)(Part*)t;
```

3.4 Complexity and Errors

This section explains why the new style of data structures reduces complexity, eliminates errors, and allows rapid software development from a simple prototype that already works like a production-quality product.

The prime purpose of the new approach to building data structures is to increase the productivity of software development and simplify its maintenance, while producing code with the ultimate performance.

These are hard and controversial things to measure. The abilities of programmers vary widely, so letting two people do the same project in different styles may not tell us much. Letting one programmer to do the same project twice

[34] This example is for the singly-linked, bi-directional Composite.

does not give meaningful results either, because the programmer learns from the first exercise and is then more efficient the second time.

However, over 2 decades of using this approach on many complex projects, our users reported a two to four times faster development and maintenance for projects without persistence, and three to ten times for projects with persistence. Small groups of developers often outperformed large departments of prestige companies. The larger and more complex the project, the greater was the productivity improvement.

Another personal experience. The author experienced exasperating frustrations when forced to develop without these libraries. Projects that he expected to take a few days, went on for weeks and required extensive debugging.

Everybody offering software tools claims improvements of productivity, and our numbers may appear exaggerated. Why such a large improvement?

The secret is in reducing the code complexity, letting the compiler find errors that we are now debugging in the run time and eliminating hard-to-find run-time errors. It makes it fun to develop software in this new style; it is less stressful.

Let's examine the individual features that, together, have this remarkable effect.

3.4.1 Reducing Complexity

It is now generally accepted that class libraries reduce code complexity and improve productivity. However, about half the data structures needed in real-life applications are bi-directional, and thow are not supported by the existing container libraries. Having generic classes for bi-directional data structures makes a big difference.

The complexity is also reduced, because the Association statements provide a concise description of the entire data organization, especially if they are together, as a block of code or in a special file (ds.def). You can also say that this block of statements defines your *framework*, or that it is a textual form of the UML class diagram.[35] If you get a program written by someone else, and you look at its Association statements, you know instantly what it is all about. Try yourself:

Listing 3.16 Can you see what data is used in this project

```
Association Collection<Library,CD> cds;
Association Hash<Library,Composer> composers;
Association Hash<Library,Performer> performers;
Association Aggregate<CD,Track> tracks;
Association Aggregate<Composer,Work> works;
Association Aggregate<Work,Track> tracksOn;
Association ManyToMany<Track,Link,Performer> playedBy;
```

[35] Section 3.5 expands on this subject.

This is a non-trivial organization—a library of CDs that can be efficiently searched by Composer or Performer. Composer composed Works (songs). CD has Tracks and several Performers may participate on one Track. A Work can be recorded several times, with different Performers and on tracks of different CDs.

The block of Associations is also extremely useful when analyzing a section of the code. For example, if you see this line:

```
works_Iterator wit;
```

you know immediately that `wit` will iterate over the Works of a Composer. Or

```
w=trackOn.parent(t);
```

tells you that `w` is a Work that is on Track `t`, even if you are not sure what are the types of `w` and `t`.

The block of Associations is also useful, when you want to see which sections of code are using certain data structures. For example, if looking for the ManyToMany relation between Track and Performer, simply search the code for "playedBy".

Reduced complexity increases the size of the problem you can keep in your mind with all its details, without keeping written records and pictures to guide you when you revisit the program. This is the mode of operation when you are most efficient. Once you reach the point when you don't remember all the parts, the project suddenly takes much more time, and the probability of making a mistake dramatically increases.

When working in a team, a clear communication is essential and, again, the block of Association statements is invaluable: it instantly clears any possible confusion related to the data organization.

3.4.2 Leaving More Work to the Compiler

All the data structures in the new libraries are strictly typed, so mistakes such as placing objects into a wrong data structure are caught by the compiler.[36] For example,

```
Composer *c; Track *t;
performers.add(t,c); //compiler error
```

You can also change, remove or add data structure without analyzing your old code, and the compiler will tell you precisely which lines will need a modification.

[36] For InCode, this is true not only in C++, but also in Java and Objective-C.

The Association statements in Listing 3.16 describe the Library of popular music, where each Work is really a song that is always recorded as a CD track. Let's assume that we already have a program running with this data organization, and we want to expand it so it would also support recordings of classical music, where a Work is a composition which usually has several Parts (movements) that are recorded on separate tracks. A CD may have tracks with only some Parts of the Work.

So in Listing 3.16, we replace

```
Association Aggregate<Work,Track> tracksOn;
```

by

```
Association Aggregate<Work,Part> parts;
Association Aggregate<Part,Track> onTracks;
```

Without even looking at the code, you attempt to compile, and the compiler tells you about all places where *tracksOn* was used and which have to be redesigned manually. The new data organizations *parts* and *onTrack* will pass the compilation. They are now empty, and you will likely need them when redesigning the places that compiler picked up.

When replacing an organization it is always safer to use different names, as we did here with *tracksOn* and *onTrack*. However, even if we used the same name, `onTrack`, the compiler would produce the same errors because of the type differences: `<Work,Track>` in the original source and `<Part,Track>` in the new version. For example the original source

```
onTrack::add(w,t);
```

would not compile in the new version because `w` is not `(Part*)`.

3.4.3 Preventing and Catching Runtime Errors

Pointer (or reference) errors are a potential source of treacherous errors, and we have eliminated all pointer members from the application. These pointers are in libraries that were carefully designed and extensively tested. All pointer chains in our libraries are coded as rings, and the basic rule is that unused pointers are always NULL.

Therefore if an object has a pointer-member which is not NULL, it indicates that the object is connected in some data structure. That provides a protection in two situations:

1. If an object is already in a pointer chain, you cannot move it by mistake to another chain. For example

```
Association Aggregate<A,B> aggr;
A *a1,*a2; B *b;
...
aggr.add(a1,b);
aggr.add(a2,b); // error message, will not execute
```

If you really want to move the object, you have to disconnect it from the old chain and then add it to the new one:

```
aggr.add(a1,b);
aggr.del(b); // dow not need del(a1,b),aggregate knows parent
addr.add(a2,b);
```

2. An object cannot be destroyed until it is completely disconnected

```
aggr.add(a1,b);
delete b; // error message
```

In this case the program may still crash later, but you will know exactly where and which pointer (and organization) was the culprit.

A better solution would be to prevent the destruction and continue in the program run. Unfortunately, once you are in the destructor, you cannot prevent the destruction. Or can you?

We could `throw` an exception, but then making a `try{ }` block around every `delete` call would make an ugly code. However, if we hide all this in a macro, we can use `safeDelete(b);` instead `delete b;` for any class; see Listing 3.17.

But wait a minute! If we are replacing delete by another call, wouldn't this be simpler:

```
class Book {
     Book *next;
public:
     Book(){next=NULL;}
     void safeDelete(){if(next!=NULL)delete this
};
int main() {
    Book *b=new Book;
    b->safeDelete();
```

The difference is that using exception works for all classes, while method `safeDelete()` has to be coded for every class.

Listing 3.17 Bypassing destruction when object is not disconnected

```
#define safeDelete(x)                    \
try {                                    \
     delete x;                           \
}                                        \
catch (BypassDestruction& bd){           \
     printf("bypassed destruction\n"); \

}

class BypassDestruction {
};

class B;

class A {
public:
     B *toB;
     A(){toB=NULL;}
     ~A(){if(toB)throw BypassDestruction();}
};

class B {
public:
     A *toA;
     B(){toA=NULL;}
     ~B(){if(toA)throw BypassDestruction();}
};

int main(){
     A *a=new A;
     B *b=new B;
     b->toA=a;
     a->toB=b;

     safeDelete(a);
     safeDelete(b);
```

3.4.4 Interface: Less May Be More

In order to benefit fully from a data structure stored in a library, its interfaces must be simple enough to remember without constantly searching the documentation. For example, InCode and DOL libraries use a much shorter list of commands than STL, where class *list* has more than a page of methods.[37] Listing 3.18 shows the methods of the InCode class comparable to `stl::list`.

[37] Plauger PJ, Stepanov AA, Lee M, Musser DR (2000) The C++ standard template Library. Prentice Hall, pp 290–292.

Listing 3.18 DOL and InCode class similar to stl_list needs fewer methods

```
Child* tail(Parent *p); // get the tail of the list
Child* head(Parent *p); // get the head of the list
void addHead(Parent *p, Child *c); // add c as the head
void addTail(Parent *p, Child *c); // add c as tail
void append(Parent *p,Child *c1, Child *c2); // c2 after c1
void insert(Child *c1, Child *c2); // insert c2 before c1
void remove(Parent *p, Child *c); // remove c from the list
Child* next(Parent *p, Child *c); // returns NULL at the end
Child* prev(Parent *p, Child *c); // returns NULL at beginning
void sort(ZZsortFun cmpFun, Parent *p); // efficient merge sort
void merge(Child *s,Child *t,Parent *p); // merge two sublists

// special commands for ring control, infrequently used
Child* nextRing(Child *c); // wrap around at the end
Child* prevRing(Child *c); // wrap around beginning
void setTail(Parent* p,Child* c,int check); // set c as tail
```

plus there is an iterator which you use like this:

```
Association LinkedList2<A,B> myList;
A *ap; B *bp;
myList_Iterator mit;
...
mit.start(ap);
ITERATE(mit,bp){ // bp traverses B objects
    ...
}
mit.start(ap);
RETRACE(mit,bp){ // reverse traversal
    ...
}
```

The iterators are smart enough to permit removal and destruction of objects while traversing the list without causing a crash or other malfunction.

3.4.5 True Rapid and Agile Development

With data structures designed in this style, you can start with skeleton classes and the Association statements, and your "program" can already compile and run. You plan in code. You evolve and experiment, and with every compilation you can print the new UML class diagram. You can change, remove or add Associations statements, and the compiler guides you as to what changes are needed. All this time you are working with a safe, running code unless, of course, you make an error in your algorithms. There should be no chasing of pointers, no mysterious low-level errors. It takes longer to clear the compiler errors, but then the program runs solid.

3.5 DB Schema and UML Class Diagram

> We are treating data structures as a memory-resident database, where the block of Association statements works like a database schema. This block of statements directly maps to/from the UML class diagram, and program called Layout can read this schema and generate the UML diagram.

When working with persistent objects you may treat your data structures as a simple but highly efficient object-oriented database. And if you implemented the data structures as we described it, then the block of Associations statements becomes a schema of this database.

You can also look at the block of Associations from a different angle. Except for the inheritance, it contains the same information as the UML class diagram. You can consider this block as a textual form of the UML class diagram, and that leads to interesting ideas.

Today, tools like Rational Rose allow one to create, in graphics, the UML class diagram. Then a code generator creates a skeleton of classes and relations you entered in graphics. If the libraries of data structures such as we recommend were commonly used, the UML tool would not need a code generator—it could simply generate the block of Association statements!

Another idea is whether it wouldn't be better to have the UML class diagram directly in the code, in a textual form, which would be an integral part of the code. It would be easy and safe to introduce changes, and the diagram and the code would not need any synchronization. It is also much faster to change a few words in the block of the Association statements than to manipulate graphics on the screen. On the other hand, most of us like a diagram when it comes to relations that form a complex network. Note that the idea of entering the information in the textual form is also used by Timothy Lethbridge in the programming environment called UMPLE—see UMPLE (2012).

We prefer to use Association statements to enter and control the relations. At the same time, perhaps with every compilation, we can automatically produce the UML class diagram in a graphical form. This reverses the control flow. Until now, the UML class diagram controlled the data organization. Now the data organization is controlled by the Association statements, and the graphical diagram is demoted to a visual aid.

This arrangement has several advantages: fast and easy initial entry, easy modifications, code and the diagram tightly synchronized, control of the data organization directly in the code—independent of any outside tools.

All[38] Code Farms libraries use a block of Association statements, each library with a slightly different syntax, and they can invoke program called *Layout*, which generates the UML class diagram.

[38] DOL, InCode, PTL and the PPF/InCode combination.

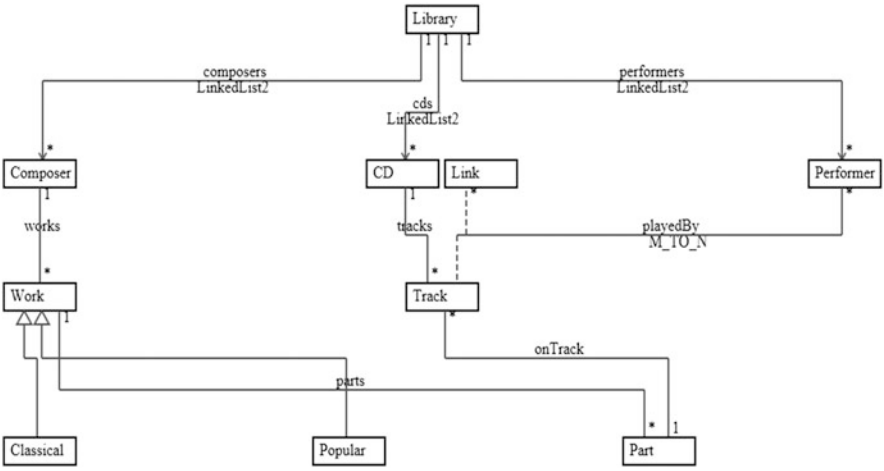


Fig. 3.8 UML class diagram generated automatically from Listing 3.19

For example, Association statements from Listing 3.19 create Fig. 3.8. Listing 3.19 is Listing 3.16 expanded for classical music. Also, in order to make the example more interesting, we added two classes derived from class *Work*, *Popular* and *Classical*.

Listing 3.19 Example for UML class diagram

```
Association Collection<Library,CD> cds;
Association Collection<Library,Composer> composers;
Association Collection<Library,Performer> performers;
Association Aggregate<CD,Track> tracks;
Association Aggregate<Composer,Work> works;
Association Aggregate<Work,Part> parts;
Association Aggregate<Part,Track> onTracks;
Association ManyToMany<Track,Link,Performer> playedBy;
class Classical : public Work
class Popular : public Work
```

How does the Layout program work? In order to draw the diagram, all data structures in the library must be registered in a special *registry* file, which was described earlier in Fig. 3.4. For example, the data structures which we are using in this example are listed with these codes:

```
u1-* LinkedList2    = uni-directional 1 to many
b1-* Aggregate2     = bi-directional  1 to many
R*2* 2XtoX          = bi-directional ManyToMany
```

In addition to the code required for the data structures and persistence, the code generators of the Code Farms libraries[39] combine the Association statements with the registry file, add the information about inheritance and, as a byproduct, generate file layout.inp, which is the input required for the Layout program; see Listing 3.20.

[39] InCode, DOL, and PTL.

Listing 3.20 Input file for program Layout which generates the UML diagram on the screen[40]

```
Inherits Popular Work ;
Inherits Classical Work ;
u1-* LinkedList2 Library CD cds ;
u1-* LinkedList2 Library Composer composers ;
u1-* LinkedList2 Library Performer performers ;
B1-* Aggregate2 CD Track tracks ;
B1-* Aggregate2 Composer Work works ;
B1-* Aggregate2 Work Part parts ;
B1-* Aggregate2 Part Track onTrack ;
R*2* M_TO_N Track Link Performer playedBy ;
```

Because C++ does not have reflection, the code generator must retrieve the information about inheritance by searching through all the *.h files for the following test pattern: `"class something :"` outside the scope of all (), [] and { } brackets, and after removing comments and lines starting with #, which is straightforward and fast.

The Layout program applies algorithms traditionally used in the design of silicon circuits, which is mathematically a similar problem to the one we have here—boxes connected with lines. However, we modified the original objective of the smallest overall area and wires not crossing each other to getting a display which would be pleasing to the human eye. The diagram must reflect the flow of the relations from the root class which is automatically detected. The program first places the boxes in rows, connecting each row before proceeding to the next row. Two labels are added to each line: name of the organization (e.g. *works*) above the line, and the name of the data structure (e.g. *Aggregate2*) below the line.

3.6 Intrusive Data Structures with Aspects

> We have discussed data structures where pointers were inserted as members or with inheritance, but we did not mention Aspects. The key idea behind Aspects is a controlled insertion of code or members, and they can be used for the data structure design we are exploring.

In 2007–2009, two workshops discussed how to implement associations as first class entities:

1. OOPSLA 2007 in Montreal, workshop "Implementing Reusable Associations/ Relationships[41]".

[40] Certain permutations of rows in this listing may cause a crash of the Layout program. This is a reported bug scheduled for repair.

[41] Sometimes referred to by the former title "The Popularity Cycle of Graphical Tools, UML, and Libraries of Associations."

2. ECOOP2009 in Genova, workshop "Relationships and Associations in Object-Oriented Languages" (RAOOL'09).

About half of the papers in these workshops were based on Aspects.[42]

Aspects provide another language layer above Java or C++, with their own compiler—AspectJ for Java programs, and AspectC++ for C++ programs.

An aspect is similar to a breakpoint in the debugger. It interrupts the program run at chosen points, allows you to examine or change application data and call a function, and then it returns to the program run. As in the debugger, no code is added to the program. The definition of where to stop and what to do is written on the side in a form resembling a class definition, and that is called *aspect*. Unlike the debugger, the aspect does not stop when interrupting the program. It executes the required actions and returns to the program run.

This type of aspect is called *dynamic aspect*, and its obvious application is a debugging layer with many printouts and checks which may be invoked in a single command but which are transparent to the ordinary user. There are also *static aspects*, which modify the structural part of the program—they can add inheritance or members to application classes.

In general, aspects can simplify programs with objects that combine several independent concerns. For example, an Employee object may store information about the employee, be persistent, and participate in an Aggregate between classes Department and Employee. If the Aggregate is implemented as an aspect, the participation of the Employee is completely transparent. The application code is very similar to what we have been doing, but there isn't even the `ZZ_Employee ds` statement which we have used. In case you are not familiar with aspects, we will analyze Listing 3.21 line by line, and explain how it works.

The overall approach is surprisingly similar to the PTL library described in Sect. 3.2 (inserting pointers with inheritance) and, because aspects can insert inheritance, this approach can also be used to implement a library of design patterns as described in Sect. 3.3.

001: Aggregate is designed as an abstract aspect, because it does not have any data, only methods that control the use of this association.

002–003: The data (references implementing the aggregate) is stored in classes AggregateParent and AggregateChild just as we did before. Java does not support multiple inheritance, but it supports multiple inheritance of interfaces. This is critical—otherwise Employee could not participate in more than one association.

004–005: If classes Department and Employee form an aggregate, then Employee must inherit AggregateChild, and Department must inherit AggregateParent, just as in Sect. 3.2.

006–008: Definition of references that form the Aggregate. This is the most tricky part of the Aggregate design. From the syntax of these lines, one would think that they insert head, next and parent into interfaces AggregateChild and AggregateParent, and thus head, next and parent must be

[42] Aspect implementation for Java; different implementation for C++ is also available.

static—which is not what we need. However, these lines perform *inter-type member insertion*,[43] which inserts head, next and parent into the classes which inherit from AggregateChild and AggregateParent, in this case into classes Employee and Department. As a result, head, next and parent are not static.

009–016: The code of method addHead () is as you would expect.
020–022: The same definition of Associations as we have been using.
023–024: Application coded in the style we have been recommending.

Listing 3.21 Implementing Aggregate with AspectJ (code obtained from the Victoria University, New Zealand)[44]

```
      // Aggregate itself is an aspect.
001  public abstract aspect Aggregate<Parent,Child> {
002     public static interface AggregateParent {}
003     public static interface AggregateChild {}

004     declare parents : Child implements AggregateChild;
005     declare parents : Parent implements AggregateParent;

006     private Child AggregateParent.head = null;
007     private Child AggregateChild.next = null;
008     private Parent AggregateChild.parent = null;

009     public static void addHead(Parent p, Child c) {
010          if(p.AggregateParent.head!=null){
011              c.AggregateChild.next=p.AggregateParent.head;
012          }
013          else c.AggregateChild.next=NULL;
014          c.AggregateChild.parent=p;
015          p.AggregateParent.head=c;
016     }
017     ...
018  }

      // Application using the Aggregate
019  public class Department {...}// same as if not using Aggregate

      // Declaration of data structures, just like our Associations
020  aspect departments extends Aggregate<Company,Department> {};
021  aspect employees extends Aggregate<Department,Employee> {};
022  aspect boss extends OneToOne<Department,Employee> {};

      // Using the Aggregate
023  Department d; Employee e;
024  employees.addHead(d,e);
```

[43] http://www.eclipse.org/aspectj/doc/next/progguide/language-interType.html

[44] Stephen Nelson, David J. Pearce and James Noble.

The drawback of this code is that if the class Employee was a Child in two Aggregates, for example

```
aspect employees extends Aggregate<Department,Employee> {};
aspect inUnion extends Aggregate<Union,Employee> {};
```

then Aggregate::addHead() would not know how to access each part. AggregateChild and AggregateParent really should be generics just like Aggregate:

```
001  public abstract aspect Aggregate<Parent,Child> {
002      public static interface AggregateParent<Parent,Child> {}
003      public static interface AggregateChild<Parent,Child> {}
```

but this version has not been tested.

Nelson, Pearce and Noble developed a library (Nelson et al. 2007; Pearce and Noble 2006) of generic associations with AspectJ, but it did not work in some special cases due to the bugs in 2007 versions of AspectJ, and these authors have not continued in this research since 2009. We also discussed with Olaf Spinczyk (Spinczyk and Lohmann 2007), author of AspectC++, how to implement Aggregate with AspectC++; see Listing 3.22, where AggregateChild, AggregateParent and Aggregate are generic classes, not interfaces. Because AspectC++ does not have inter-type insertion, the declaration of the Aggregate aspect takes four lines, where the line *advice—slice—*... inserts the required pointers into participating classes. However, we can hide these four lines under a macro—see Listing 3.22, and then the invocation of the Aggregate is the same as it was in AspectJ, or all the other libraries we have discussed throughout Chap. 3

Listing 3.22 Aggregate implemented with AspectC++

```
// reusable part
//------------------
template <typename Aggregation> class AggregateChild {
public:
     typename Aggregation::Child *next;
     typename Aggregation::Parent *parent;
     AggregateChild () : next (0), parent (0) {}
};

template <typename Aggregation> class AggregateParent {
public:
     typename Aggregation::Child *head;
     AggregateParent () : head (0) {}
};

template <typename Aspect, typename _Parent, typename _Child> class
Aggregation {
public:
     typedef _Parent Parent;
     typedef _Child Child;

     static void addHead (Parent *p, Child *c) {
          typedef AggregateParent<Aspect> P;
          typedef AggregateChild<Aspect> C;
          if(p->P::head) c->C::next = p->P::head;
          else c->C::next = 0;
          c->C::parent = p;
          p->P::head = c;
     }
};

#define AGGREGATION(Name, Parent, Child)                          \
   aspect Name : public Aggregation<Name, Parent, Child> {        \
   advice #Child : slice class : public AggregateChild<Name>;     \
   advice #Parent : slice class : public AggregateParent<Name>; \
}

// application specific part
/ ----------------------------
class Employee;
class Department;
class Union;

AGGREGATION(inDepartment, Department, Employee);
AGGREGATION(inUnion, Union, Employee);

class Employee {};
class Department {};
class Union {};

int main () {
     Department *dp; Union *up; Employee *ep1,*ep2;
     dp=new Department; up=new Union;
     ep1=new Employee; ep2=new Employee;
     ...
     inDepartment::addHead (dp,ep1);
     inDepartment::addHead (dp,ep2);
     inUnion::addHead (up, ep2);
      }
```

3.7 Conclusion

Aspects lead to the same simple use of associations as we obtained through other methods, but we are not in favour of adding a complex programming layer just for implementing associations. In our opinion, associations are part of the core programming and we believe they should be supported from within that environment.