# Pattern Template Library

# User's Guide

www.codefarms.com

Code Farms Inc.
7214 Jock Trail
Richmond, Ontario, Canada
K0A 2Z0

Telephone: (613) 838-4829
Email: info@codefarms.com

August, 1998



If you are interested in the overall design philosophy of this library or in the comparision of this approach with container-based class libraries such as STL or Rogue Wave, read article [1].

Classes from this library can be mixed with classes from any other class library.

# Chap.1: INSTALLATION

---

This library comes in full source code, and all you have to do is to unpack it or copy the directory tree onto your hard disk. The instructions are provided with the software.

Under the root directory PTL, you will have the following directories:

- LIB ... the library itself, one *.h file for each class template.
- ENVIR ... directory which lets you convert the library easily to different operating systems and compilers, and contains compilation script files for the regression tests.
- MGR ... source and executable of the Template Manager, and of a special DIFF program which is used in regression testing.
- DOC ... the documentation (Users Guide) in HTML format.
- TTEST ... regression test suite using plain templates (not the Template Manager).
- MTEST ... regression test suite using the Template Manager.
- OUT ... correct results for all tests (identical for TTEST and MTEST).

After you have installed all of the files, recompile the library using these steps:

- ✎ Change to directory PTL\ENVIR, and select the batch file corresponding to your favorite compiler. For example, for Microsoft Visual C++, type *msft*, or for Borland C++ type *borland*. If there is no batch file for your compiler, modify one of these files.
- ✎ Change to directory MGR, and type *all*. This will recompile the Template Manager, and a special *diff* program used in the regression tests.
- ✓✎ There is no need to recompile the library itself. Directory LIB contains source code for all the templates which must be compiled again with each application.

All the documentation is in the directory DOC. The documentation is in HTML format, and you can conveniently browse through it with any WWW browser, such as Netscape Navigator/Communicator or Internet Explorer from Microsoft. Start with the file GUIDE.HTM. If you don't have a WWW browser, you can either ask a friend who has such a browser to print the documentation for you, or you can purchase the printed Users Guide from our company.

Directories TTEST and MTEST contain test programs for all patterns currently available from the library. These programs can verify that your installation is complete and correct, but they are also invaluable as examples and provide guidance when you are learning how to use a new class. To start a complete regression test, type *regr* in either directory.

If you run into difficulties, feel free to contact us:


- email (prefered)info@codefarms.com
- fax: 613-838-3316
- telephone (please use only as last resort): 613-838-4829
- Our address is: Code Farms Inc., 7214 Jock Trail, Richmond, Ontario, K0A 2Z0 Canada
- For information on other products and services, seehttp://www.CodeFarms.com


---

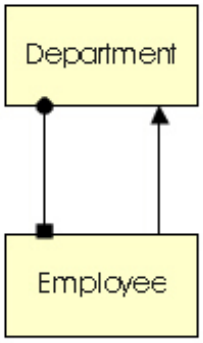
# Chap.2: BASIC CONCEPTS

---

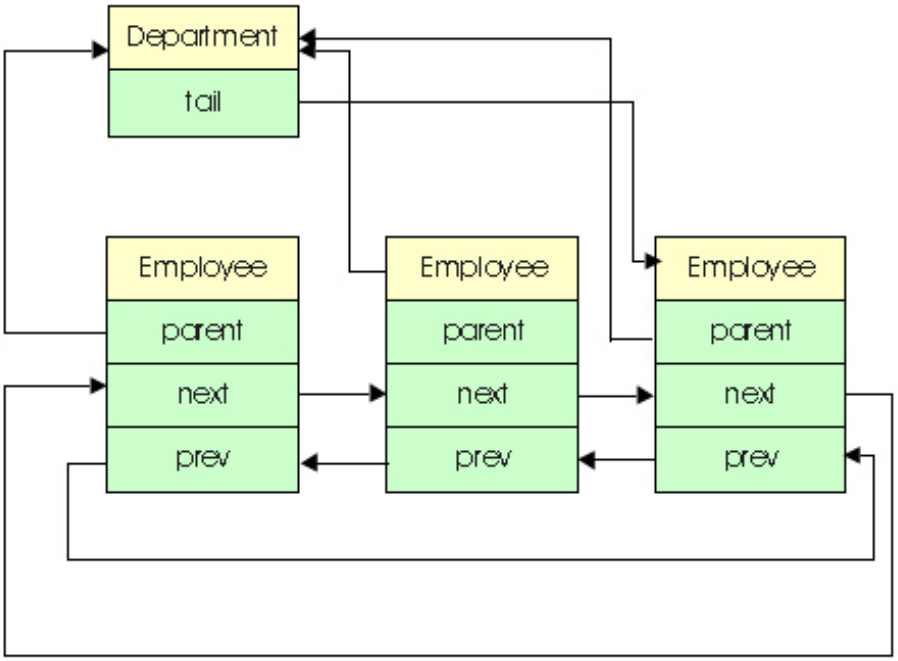
Let's start with an example, where every object of the class Department contains several objects of the class Employee, and each Employee knows its Department. This relation is often called an AGGREGATE, or 1-to-many relation, and it is a frequently occuring data structure in most applications.

If you code this aggregate from scratch, very likely you would code it with pointers embedded in classes Department and Employee:

## Class Diagram:




## Implementation:



There are several ways of implementing the list. Instead of the usual NULL-ending list, we used a circular list (ring). As explained in Chap.3, a ring allows fast run-time checking of data integrity. This library uses rings for all its classes.

The diagram above does not show the assignment of the functions that will manipulate the aggregate. Most programmers would assign these functions arbitrarily to classes Department or Employee, depending on where the function could be most naturally implemented. For example:

```
Employee* Employee::nextEmployee(){ return next;}
Department* Employee::myDepartment(){ return parent;}
void Department::addEmployee(Employee *e){ ... };
Employee* Department::firstEmployee(){
    Employee* e=NULL; if(tail)e=tail->next; return e;}
```

As explained in [1] and [2], if classes Department and Employee are not part of one aggregate, but participate in several data structures (a quite common situation), the result is spaghetti++ code which is difficult to debug and maintain. Also, classes Department and Employee (and possibly many other classes) could not be compiled independently. Any time you change one of the headers, the entire project must be recompiled.

The cure for this problem is to implement a special manager class for each data structure. This class typically contains no or little data, but it has all the functions that manipulate the data structure. To provide fast access to all the pointers, this class must be a friend of the classes that participate in the data structure. For example, in our example, we can introduce class DEaggregate:

```
class DEaggregate {
public:
    Employee *nextEmployee(Employee *e){return e->next;}
    Department *myDepartment(Employee *e){return e->parent;}
    void addEmployee(Employee *e){ ... }
    Employee *firstEmployee(Department *d){
         Employee* e=d->tail; if(e)e=e->next; return e;}
    ...
};
class Department {
friend class DEaggregate;
    Employee *tail;
    ...
public:
    ...
};
class Employee {
friend class DEaggregate;
    Department *parent;
    Employee *next;
    Employee *prev;
    ...
public:
    ...
};
```

We use manager classes to represent not only simple data structures such as aggregates, but also structural design patterns.

Instead of coding the aggregate again for every new application, it makes more sense to design a generic aggregate with C++ templates, and store it in a class library. The pointers that we need in the application classes such as Employee, can be injected through inheritance. In the following code, class P is the application class which will be the parent of the aggregate, and class C is the class which will be the child of the aggregate. The integer parameter i, with default 0, prevents conflict when using the same organization between the same classes more than once (e.g. for multiple lists):

The following classes will come from a library:

```
template<class P, class C, int i=0> class Aggregate {
public:
    C *next(C *c){return (AggregateChild<P,C,i>::c)->next;}
    P *parent(C *c){return (AggregateChild<P,C,i>::c)->parent;}
    void addTail(P *p,C *c); // add c as a tail under p
    void remove(C *c); // remove child c from its parent
    C *head(P *p){ C* c=(AggregateParent<P,C,i>::p)->tail;
                                        if(c)c=c->next; return c;}
    ...
};
template<class P, class C, int i=0> class AggregateParent {
friend class Aggregate<P,C,i>;
    C *tail;
public:
    AggregateParent(){tail=NULL;}
};
template<class P, class C, int i=0> class AggregateChild {
friend class Aggregate<P,C,i>;
    P *parent;
    C *next;
    C *prev;
public:
    AggregateChild(){parent=NULL; next=prev=NULL;}
};
```

The application uses the library classes. Since we have only one aggregate between Department and Employee, we can use the default value for paremeter i. In essence, this is the way the Pattern Template Library is coded and used:

```
class Department : public AggregateParent<Department,Employee> {
    ... // no members or functions related to the aggregate
};
class Employee : public AggregateChild<Department,Employee> {
    ... // no members or functions related to the aggregate
};
```

---

# Chap.3: SMART ITERATORS

---

A class library usually provides one or several iterators for each data structure. The iterator permits you to traverse the data using the operators ++ and --. However, if you try to destroy an aggregate by traversing it and destroying individual children, most libraries will crash:

```
class Department { ... };
class Employee { ... };
Employee *e;
Aggregate<Department,Employee> agg;
// create a department with 10 employees
Department* d=new Department;
for(i=0; i<10; i++){
    e=new Employee;
    aggr.addTail(d,e);
}
// disconnect and destroy all employees
AggregateIterator<Department,Employee> it;
for(e=it.start(d); e; e=++it){ // crash on the second pass
    aggr.remove(e);
    delete e;
};
```

The reason for this crash is that most iterators remember the current object, and use its *next* pointer to get to the next object on the list. If the current object is destroyed within the loop, the next call to ++ attempts to use pointer *next* on an invalid (already destroyed) object, which results in a crash.

The cure for this problem is to have the iterator remember not the current object, but the next object on the list. Then, even if the current object is destroyed, you still get the next object correctly. This is the method used througout the Pattern Template Library. If you look at the implementation of any of the iterators in lib/*.h, you will see that this only slightly complicates the logic of the operators ++ and --, and is certainly worth the qualitative improvement in the behavior of the iterator.

The improved iterator still is not bullet proof. For example, if you perform the following operation, it will crash:

```
Employee *e, *nxt;
for(e=it.start(d); e; e=++it){
    if(...){
        nxt=aggr.nextChild(e);
        aggr.remove(nxt);
        delete nxt;
    }
    ...
};
```

It is possible to design an iterator which permits one to remove or add objects while traversing the list. In this case, for example, the Aggreate class must keep the list of all active AggregateIterator instances. Functions *Aggregate::addTail()* and *Aggregate::remove()* must run through all active iterators and check whether the change in the list affects the next object stored in each iterator, and if it does, modify it.

The PTL does not use this type of iterator, because it would cause problems when running with multiple threads.

The PTL permits you to use three styles of traversal loops. The first style combines a *for(..)* loop with functions *start()* or *end()* which start the iterator from the beginning or the end of the list:

```
Department *d;
Employee *e;
```

```
AggregateIterator<Department,Employee> it;
...
for(e=it.start(d); e; e= ++it){
    ... // runs through all e under d
}
for(e=it.end(d); e; e= --it){
    ... // runs through all e under d
}
```

The second style uses the convenient macros ITERATE() or RETRACE(), which hide the underlying *for()* loops:

```
Department *d;
Employee *e;
AggregateIterator<Department,Employee> it;

...
ITERATE(it,d,e){
    ... // runs through all e under d
}
RETRACE(it,d,e){
    ... // runs through all e under d
}
```

The third style uses function next(), but you have to handle the end of the loop yourself. Don't forget, we are working with a ring:

```
for(e=aggr.head(d); e; e=aggr.next(e)){
    ... // runs through all e under d
    if(e==aggr.tail(d))break; // end condition
}
```

Multiple loops are permitted in both styles, ++ and -- can be used within the same loop. If you use the iterator, functions *next()* and *prev()* may be used, but must not be used to advance the loop variable. For example:

```
Department *d;
Employee *e,*ee;
AggregateIterator<Department,Employee> it;
Aggregate<Department,Employee> aggr;
...
ITERATE(it,d,e){
    ...
    e=aggr.next(e);  // has no effect on the next iteration
    ee=aggr.next(e); // no problem
    ...
}
```

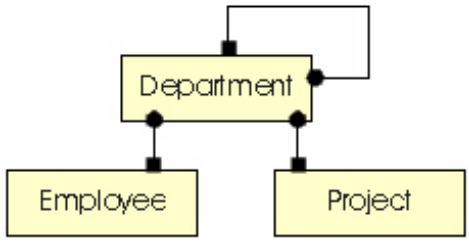
SUMMARY:

1. For basic loops, use ITERATE() or RETRACE(); otherwise, start each loop with a *begin()* or *end()* function.
2. Multiple loops are allowed.
3. Operators ++ and -- may be used within the same loop.
4. When using iterators, do not advance the loop variable using the functions *next()* or *prev()*.

---

# Chap.4: EXAMPLE WITH SEVERAL AGGREGATES

Let us go through a full, running sample problem which uses several aggregates. Below is the class diagram of the problem. In this simplified textual form, symbol A 0------x B represents an aggregate between A and B. This is the same as assigning many Bs to each A.




Each Department has several other Departments. In other words, Departments form a tree. Within this tree, each Department has a set of Employees and a set of Projects. The following program uses the class Aggregate from the Pattern Template Library. It creates a sample hierarchy of Departments, each with Employees and Projects, demonstrates how the data can be manipulated by removing one Employee and moving another Employee into a new position, and then prints the resulting data.

Watch for the following details:

- For each library class, either include *.h or *.cpp, but never both.
- We assume here that you compile with the I option set to the directory PTL/LIB, where all the *.h and *.cpp files are stored.
- The inheritance statements may look complicated at first, but if you follow the OO diagram, they make sense. Later on, you will learn how to avoid these statements.
- Function *Department::find()* is interesting because it shows how you can easily traverse the entire tree.
- Function *Department::prt()* demonstrates the use of iterators for various aggregates. Don't get confused by variable offset and function *prtOffset()*. Both are used only to generate a nice looking printout, with deeper offset for subordinated departments. - Note how the entire data structure is compressed into the three *Aggregate*<> statements. These three lines are almost like a database schema: they contain all the relations between the application classes.
- The creation of the data is a bit tedious to read, but we wanted to show you a meaningful example. The end of *main()*, where the data are modified, is more interesting.
- Note that before destroying the Boss, he must be first removed from his Department, and before moving Chapeau to another Department, she also must first be removed from her original Department. If this did not happen, the library would detect a run time error.

```
// ------------------------------------------------------
#include <iostream.h>
#include <string.h>
#include <mgr.h>  // always include
#include <aggregat.h>
class Department;
class Employee;
class Project;
class Department : public AggregateParent<Department,Department>,
                   public AggregateChild<Department,Department>,
                   public AggregateParent<Department,Employee>,
                   public AggregateParent<Department,Project>{
    int mDeptNo;
```

```
    void prtOffset(int i);
public:
    Department(int n){mDeptNo=n;}
    void prt(int offset); // print this and subordinate departments
    Employee* find(char *name); // recursively, find this employee
};
class Employee : public AggregateChild<Department,Employee>{
    char *mpName;
public:
    Employee(char *n){mpName=new char[strlen(n)+1]; strcpy(mpName,n);}
    void prt(void){cout << " " << mpName;}
    int checkName(char *n){return !strcmp(mpName,n);}
};
class Project : public AggregateChild<Department,Project>{
    char mProj;
public:
    Project(char p){mProj=p;}
    void prt(void){cout << " " << mProj;}
};
// ------------------------------------------------
// The following lines declare the data organization.
// These objects do not have to be global, but making them
// global often contributes to the clarity of the code
// ------------------------------------------------
Aggregate<Department,Department> departments;
Aggregate<Department,Employee> employees;
Aggregate<Department,Project> projects;
// ------------------------------------------------
void Department::prtOffset(int i){
    cout << "\n";
    for(int k=0; k<i; k++) cout << "    ";
}
void Department::prt(int offset){
    Employee *pEmpl;
    Project *pProj;
    Department *pDept;
    AggregateIterator<Department,Department> deptIt;
    AggregateIterator<Department,Employee>   emplIt;
    AggregateIterator<Department,Project>    projIt;
    prtOffset(offset);
    cout << "department= " << mDeptNo;
    if(employees.head(this)){
        prtOffset(offset);
        cout << "employees:";
        ITERATE(emplIt,this,pEmpl) pEmpl->prt();
    }
    if(projects.head(this)){
        prtOffset(offset);
        cout << "projects:";
        ITERATE(projIt,this,pProj) pProj->prt();
    }
    cout << "\n";
    // recursively, print all subordinated departments
    ITERATE(deptIt,this,pDept) pDept->prt(offset+1);
}

Employee* Department::find(char *name){
    Employee *pEmpl;
```

```
    Department *pDept;
    AggregateIterator<Department,Department> deptIt;
    AggregateIterator<Department,Employee>   emplIt;
    // traverse the employees of this department
    ITERATE(emplIt,this,pEmpl){
        if(pEmpl->checkName(name))return pEmpl;
    }
    // recursively, search subordinate departments
    ITERATE(deptIt,this,pDept){
        pEmpl=pDept->find(name);
        if(pEmpl)return pEmpl;
    }
    return NULL;
}

int main(void){
    Department *pDept0,*pDept1,*pDept2;
    Employee *pEmpl;
    Project *pProj;

    // create a small hierarchy of departments under pDept0
    // To each department, add some employees and projects
    pDept0=new Department(200);
        pEmpl=new Employee("Head F."); employees.addTail(pDept0,pEmpl);
    pDept1=new Department(220); departments.addTail(pDept0,pDept1);
        pEmpl=new Employee("Mann H."); employees.addTail(pDept1,pEmpl);
    pDept2=new Department(225); departments.addTail(pDept1,pDept2);
        pEmpl=new Employee("Brown J."); employees.addTail(pDept2,pEmpl);
        pEmpl=new Employee("Green D."); employees.addTail(pDept2,pEmpl);
        pEmpl=new Employee("Black S."); employees.addTail(pDept2,pEmpl);
        pProj=new Project('A'); projects.addTail(pDept2,pProj);
    pDept2=new Department(228); departments.addTail(pDept1,pDept2);
        pEmpl=new Employee("Little K."); employees.addTail(pDept2,pEmpl);
        pEmpl=new Employee("Grossman A."); employees.addTail(pDept2,pEmpl);
        pProj=new Project('B'); projects.addTail(pDept2,pProj);
        pProj=new Project('C'); projects.addTail(pDept2,pProj);
    pDept1=new Department(240); departments.addTail(pDept0,pDept1);
        pEmpl=new Employee("Boss I."); employees.addTail(pDept1,pEmpl);
    pDept2=new Department(243); departments.addTail(pDept1,pDept2);
        pEmpl=new Employee("Zeller K."); employees.addTail(pDept2,pEmpl);
        pEmpl=new Employee("Chapeau A."); employees.addTail(pDept2,pEmpl);
        pEmpl=new Employee("Krpec P."); employees.addTail(pDept2,pEmpl);
        pProj=new Project('D'); projects.addTail(pDept2,pProj);
        pProj=new Project('E'); projects.addTail(pDept2,pProj);
        pProj=new Project('F'); projects.addTail(pDept2,pProj);
    // Boss I. leaves the company
    pEmpl=pDept0->find("Boss I.");
    employees.remove(pEmpl);
    delete pEmpl;
    // Chapeau A. is promoted as head of department 240
    pEmpl=pDept2->find("Chapeau A.");
    employees.remove(pEmpl);
    employees.addTail(pDept1,pEmpl);
    pDept0->prt(0); // print the entire organization
    return(0);
}
/* ---------------------- results -----------------------
department= 200
```

```
employees: Head F.
    department= 220
    employees: Mann H.
        department= 225
        employees: Brown J. Green D. Black S.
        projects: A
        department= 228
        employees: Little K. Grossman A.
        projects: B C
    department= 240
    employees: Chapeau A.
        department= 243
        employees: Zeller K. Krpec P.
        projects: D E F
   ----------------------------------------------------*/
```

For your convenience, this program is stored in the documentation directory as PTL\DOC\TEST4.CPP, and the results are in file PTL\DOC\OUT4.

---

# Chap.5: TEMPLATE MANAGER

When going through the preceding example, you probably noticed the complex inheritance statements, especially in class Department. Imagine how ugly this can be in a real life application with several dozen classes connected by about the same number of relations.

Fortunately, you, as the user, do not have to worry about these statements; they can be made completely transparent. The reason is that the declaration of the relations already contains all the information:

```
// ------------------------------------------------
Aggregate<Department,Department> departments;
Aggregate<Department,Employee> employees;
Aggregate<Department,Project> projects;
// ------------------------------------------------
```

If we know that class Aggregate works with classes AggregateParent and AggregateChild (and that information is stored in the library), then we can see that

```
from line1: Department must inherit AggregateParent<Department,Department>
            Department must inherit AggregateChild<Department,Department>
from line2: Department must inherit AggregateParent<Department,Employee>
            Employee   must inherit AggregateChild<Department,Employee>
from line3: Department must inherit AggregateParent<Department,Project>
            Project    must inherit AggregateChild<Department,Project>
```

Instead of coding the inheritance statements by hand, we can write a simple template generator which reads the declarations of the relations, and derives the inheritance statements from them. The full source code of the template manager is enclosed in mgr/mgr.cpp, and has only 450 lines of code after stripping off comments and blank lines.

To make the code generation simple, we reserve two keywords: *pattern* and *Pattern.* The first keyword will mark all pattern or relation declarations so that the template manager can find them easily.

```
// ------------------------------------------------
pattern Aggregate<Department,Department> departments;
pattern Aggregate<Department,Employee> employees;
pattern Aggregate<Department,Project> projects;
// ------------------------------------------------
```

The second keyword will be a cover name for the combined inheritance statements required for the given class. The template manager will generate these Pattern(..) expressions in the form of a macro:

```
    class Department : Pattern(Department) {
      ... // everything as before
    };
    class Employee : Pattern(Employee) {
      ... // everything as before
    };
    class Project : Pattern(Project) {
      ... // everything as before
    };
```

If we have the Template Manager which already generates code, we can improve two more things:

(1) In the example, all the relations were applications of the same template, AGGREGATE. For this reason, we only had to include one file from the library, *aggregate.h*. In real life situations, there would be a variety of relations used, and when

coding everything by hand, you would have to control which library files were included. Since the Template Manager already knows which patterns/relations are going to be used, it can decide which files must be included. The user then includes only a single file, *pattern.h*, without worrying about which patterns are used.

(2) Instead of using the template form of the iterators, we can use simplified names derived from the instances of the pattern. These names just hide the proper template, and are created in *pattern.h*. For example, instead of *AggregateIterator<Department,Department> it*; you can write *pattern_iterator_departments it*;

With these features, the code for the example will be much simpler:

```
#include <iostream.h>
#include <string.h>
#define TEMPLATE_MANAGER
#include "pattern.h" // includes all patterns used here
class Department;
class Employee;
class Project;
class Department : Pattern(Department) {
    ... // everything as before
};
class Employee : Pattern(Employee) {
    ... // everything as before
};
class Project :  Pattern(Project) {
    ... // everything as before
};
// these statements are the roadmap to all patterns/relations
// ------------------------------------------------
pattern Aggregate<Department,Department,0> departments;
pattern Aggregate<Department,Employee,0> employees;
pattern Aggregate<Department,Project,0> projects;
// ------------------------------------------------
... // The remaining code is identical, except for the iterators
    // that either have to have the last parameter 0,
    // or must use the simplified form of iterators:
```

For example:

```
    AggregateIterator<Department,Department,0> deptIt;
    AggregateIterator<Department,Employee,0>   emplIt;
    AggregateIterator<Department,Project,0>    projIt;
```

or

```
    pattern_iterator_departments deptIt;
    pattern_iterator_employees   emplIt;
    pattern_iterator_projects    projIt;
```

Prior to compiling your program, you run the template manager with your program (or its header file) as the input:

```
    mgr\mgr mycode.c
```

You don't have to go through this code generation again and again while debugging the program. You must generate a new pattern.h only when you add or remote a pattern, or when one of the patterns changes. You must link to the Pattern Template library, for example for the Borland compiler:

```
    bcc -ml -I..\lib test.cpp
```

**RULES FOR USING THE Pattern(..) STATEMENTS:**

(a) For every pattern, you must have the full *pattern* statement.

(b) Every class listed in any of the 'pattern' statements must have a *Pattern* statement.

(c) This statement must be inserted just before '{' which starts the class description. '{' must be on the same line, and nothing except possibly a comment may follow. Examples:

```
class A : Pattern(A) {                    // OK
class B : Pattern(B) {};                  // not acceptable
class C : Pattern(C)
{                                         // not acceptable
class D : public A, public X, Pattern(D) { // OK
```

(d) There is one exception to this mechanical use of the statement 'Pattern'. Character ':' must not be used when the class does not inherit from any other application class and, at the same time, it participates passively as the second parameter in one or more statements pattern *Array<H,T,i>* ... or pattern PtrArray<H,T,i>. Examples:

```
class A : Pattern(A) {
    ...
};
class B Pattern(B) {  // Note that ':' is missing
    ...
};
pattern Array<A,B,0> myArray;
```

but if B is active in other patterns, the character ':' is needed:

```
class A : Pattern(A) {
    ...
};
class B : Pattern(B) {  // Note that ':' is used
    ...
};
pattern Array<A,B,0> myArr;
pattern Collection<A,B,0> myCol;
```

We do not like exceptions, but we could not figure out a programming trick which would make the use of Pattern() more uniform. The good thing is that even if you make a mistake in these rules, the compiler will tell you that you have an error on a particular line. Watch that you have always just one '{' after the Pattern() statement, and nothing more on the same line. If you violate this rule, the compiler message is usually obscure, and relates to the nesting of braces {}.

---

# Chap.6: AVAILABLE CLASSES

---

Note that for each class, the directories PTL/TTEST and PTL/MTEST provide a program which demonstrates the use of the class, and verifies that they work correctly. Tests in TTEST have explicitely coded multiple inheritance, tests in MTEST use the Template Manager. All iterators are smart iterators that permit one to remove() and delete() objects while traversing the data.

You can easily add your own classes to the library - see Chap.9. When you receive the library, it will include the following classes:

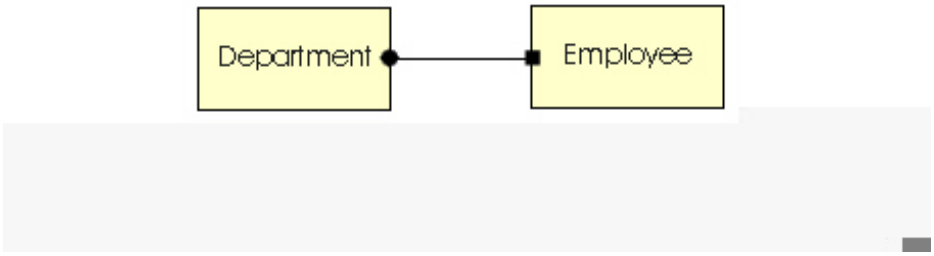


---

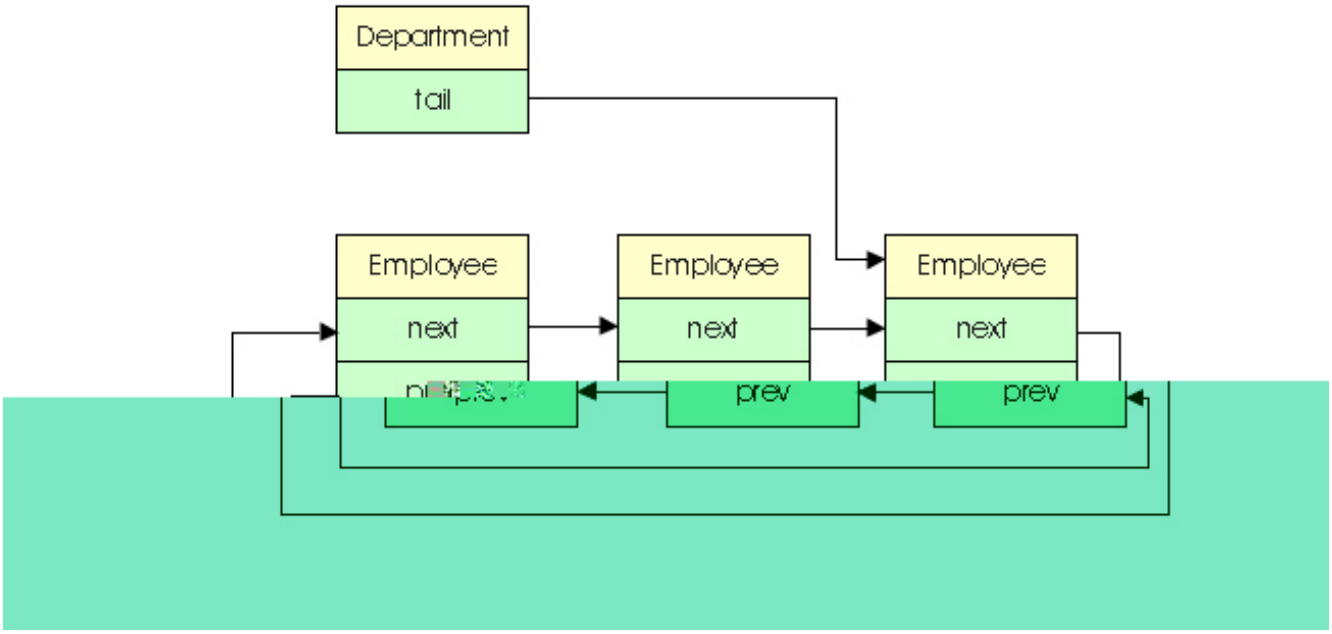
## (6.1) COLLECTION or LINKED LIST (file collecti.h)

In this data organization, each Parent keeps a doubly linked, intrusive, circular list of Children. Depending how you interpret this pattern, it can be used as a linked list, collection, sorted collection, or a set. It is one of the basic building blocks of numerous data structures and patterns.

Class diagram:




Implementation:




In the the following description, we will often use P for the Parent class (for example Department), C for the child class (for example Employee), and i for the integer index, which is 0 most of the time.

**CONDITIONS FOR USING THIS PATTERN:**

1. P must be a subclass of CollectionParent<P,C,i>
2. C must be a subclass of CollectionChild<P,C,i>

When you use the PatternManager, this happens automatically.

**AVAILABLE METHODS:**

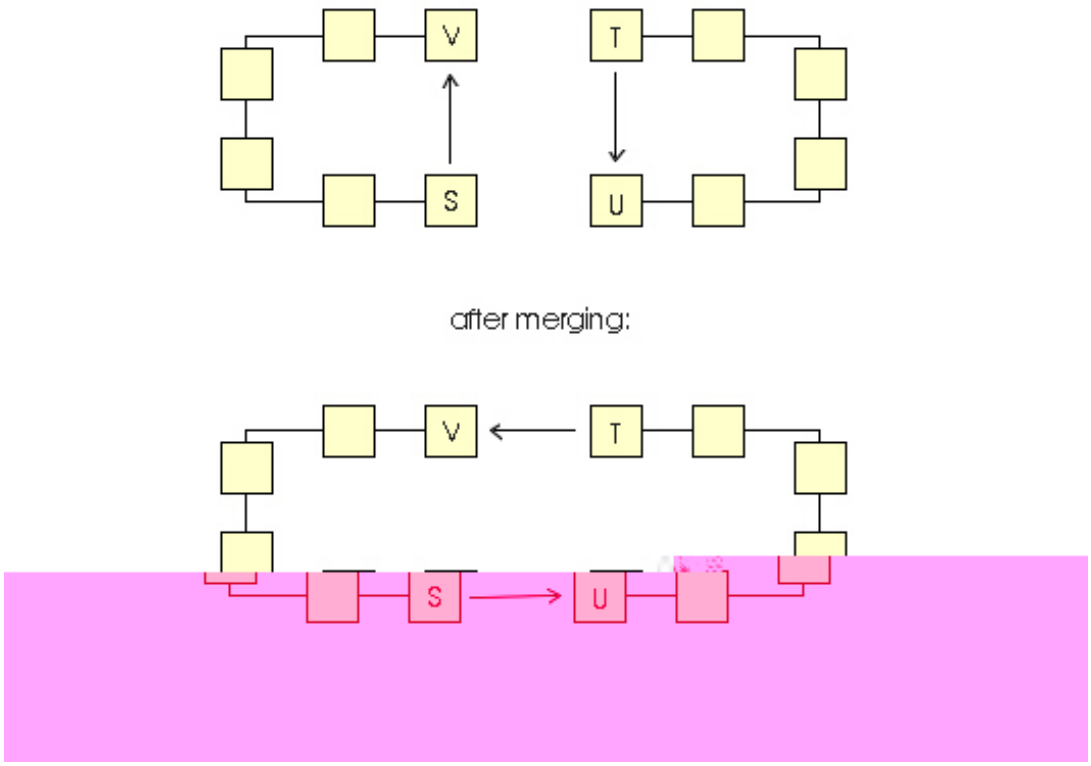
```
template<class P,class C, int i=0> class Collection {
    ...
public:
    cType *head(const pType *p) const; // head of the list
    cType *tail(const pType *p) const; // tail of the list
    cType *next(const cType *c) const; // next on the list
    cType *prev(const cType *c) const; // previous on the list
    void setTail(pType *p,cType *c) const; // set c to be the tail on p
    void setHead(pType *p,cType *c) const; // set c to be the head on p
    void addTail(pType *p,cType *c) const; // add c as new tail under p
    void addHead(pType *p,cType *c) const; // add c as new head under p
    void insert(pType *p,cType *c,cType *x) const; // insert x before c
    void append(pType *p,cType *c,cType *x) const; // append x after c
    void remove(pType *p,cType *c) const; // remove c from under p
    void merge(cType *s,cType *t,pType *ps,pType *pt) const; // see below
    void merge(pType *ps,pType *pt) const; // merge pt after ps
    void split(cType *s,cType *t,pType *ps,pType *pt) const; // see below
    void sort(pType *p,cmpType cmp) const; // sort using C::cmp()
    void addSorted(pType *p,cType *c,cmpType cmp) const; //add,keep sorted
    int count(const pType* p) const; // count of items
};
```

Situations which violate data integrity are detected. Examples: *remove(p,c)* or *setHead(p,c)* when c is not under p. For more

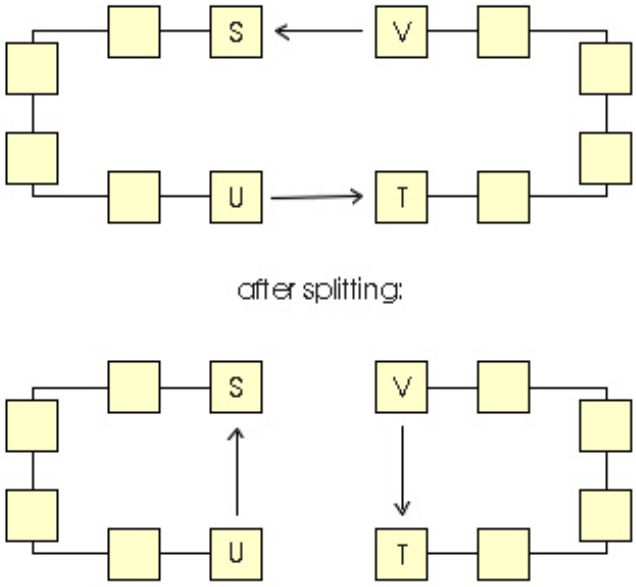
details on error handling, see Chap.10.

The function *split(s,t,ps,pt)* splits one collection into two, while the function *merge(s,t,ps,pt)* merges two collections into one. In both situations, two children (s and t) and their parents (*ps* and *pt*) are involved:

## Merging two collections:




## Splitting a collection:




In the case of merging, s and t must be in different collections. The function disconnects s and t from their next neighbors v and u, and connects t to v, and s to u - see the figure. The new collection is under ps, and pt is empty.

In the case of splitting, both s and t must be originally under ps, and pt must be an empty Parent. The function again disconnects v and t from their next neighbors v and u, and connects t to v and s to u. The result is two collections: one which contains s and u and remains under ps, and one which contains v and t and is assigned to pt.

In either case, if s=t, no action is taken. It may interest you that, internally, a single algorithm provides both services. To make the interface easy to use, this general function is hidden behind merge() and split().

There is also a second function for merging two collections, *void merge(pType *ps,pType *pt);* which is the most frequently occuring situation.

**ORDER OF THE ITEMS IN THE COLLECTION:**

When you add items using *addTail()*, the iterator will return them in the same order (FIFO queue). When you add items using *addHead()*, the items will be stored in the reverse order (LIFO queue). Function *sort()* sorts the collection by a fast merging algorithm, which has a performance comparable to *qsort()* for arrays. The sort is controlled by the compare function *int C::cmp(C *c1, C *c2)*, which is similar to the compare function required for *qsort()*. The result of *cmp(c1,c2)* must be < 0 when c1>c2, must be > 0 when c1>c2, and must be 0 when they are equal. We recommend that if you need sorting, you should make this function a static member function of class C.

Function *addSorted()* adds an item to the collection so that its order is maintained. Since Collection is based on a linked list, this operation is relatively inefficient (i.e., it must traverse the list). Rather then using repeatedly *addSorted()*, it is much faster to use *addTail()* or *addHead()* and then sort the collection by calling *sort()*.

Function *count()* returns the number of Children under Parent p. It is not efficient to store this value on each parent; therefore this value is calculated every time you ask for it, and it requires a full traversal of the list.

**ITERATOR:**

```
template<class P,class C, int i=0> class CollectionIterator {
    ...
public:
    CollectionIterator();     // creates an iterator
    C* begin(const pType *p); // starts the forward iteration (ITERATE)
    C* end(const pType *p);   // starts the reverse iteration (RETRACE)
    C* const operator++();    // next object in the forward direction
    C* const operator--();    // next object in the reverse direction
};
```

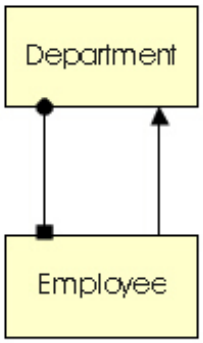
For examples of using this data structure, see *ttest/collecti.cpp* and *mtest/collecti.cpp*. The correct results are in *ptl/out/collecti.out.*

---

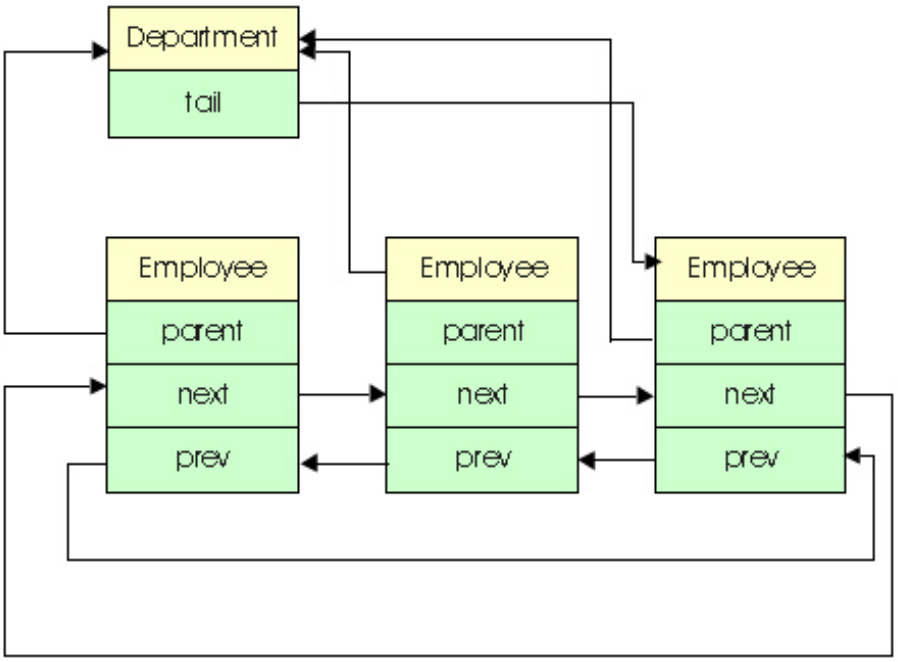
## (6.2) AGGREGATE or ONE_TO_MANY (file aggregat.h)

This data organization is sometimes called a ONE_TO_MANY relation. For each Parent, it keeps a doubly linked, intrusive, circular list of Children. This pattern is very similar to the COLLECTION, except that each Child also keeps a pointer to its Parent. Internally, class Aggregate<> is derived from class Collection<>. The AGGREGATE represents a hierarchy of objects, and it is one of the most useful and most frequently used data structures. Examples: Department and Employees, CircuitBlock and its Terminals, Account and its Transactions.

Class diagram:




Implementation:




In the the following description, we will often use P for the Parent class (for example, Department), C for the child class (for example, Employee), and i for the integer index, which is 0 most of the time.

**CONDITIONS FOR USING THIS PATTERN:**

1. P must be a subclass of CollectionParent<P,C,i>
2. C must be a subclass of CollectionChild<P,C,i>

When you use the PatternManager, this happens automatically.

**AVAILABLE METHODS:**

Note that even though Aggregate is quite similar to Collection, the syntax of some functions is different. The reason is that, in Aggregate, children always know their parents, therefore the parents do not have to be given in operations like *insert()* or *remove()*.

```
template<class P,class C, int i=0> class Aggregate {
    ...
public:
    // methods which are the same as for Collection
    cType *head(const pType *p) const; // head of the list
    cType *tail(const pType *p) const; // tail of the list
    cType *next(const cType *c) const; // next on the list
    cType *prev(const cType *c) const; // previous on the list
    void addTail(pType *p,cType *c) const; // add c as the new tail under p
    void addHead(pType *p,cType *c) const; // add c as the new head under p
    void setTail(cType* c) const; // set c to be the tail on p
    void setHead(cType* c) const; // set c to be the head on p
    void merge(pType *ps,pType *pt) const; // merge pt after ps
    void addSorted(pType *p,cType *c,cmpType cmp) const; // add, keep sorted
    void sort(pType *p,cmpType cmp) const; // sort collection using C::cmp()
    int count(const pType* p) const; // count of items
```

```
    // methods which are different from Collection
    pType *parent(cType *c) const; // the parent of c
    void insert(cType *c,cType *x) const; // insert x before c,under parent of c
    void append(cType *c,cType *x) const; // append x after c, under parent of c
    void remove(cType *c) const;          // removes c from its parent
    void merge(cType *s,cType *t) const;            // - see the text
    void split(cType *s,cType *t,pType *pt) const;  // - see the text
            // splits aggregate into two, pt must be a new, empty parent
};
```

For the description of the *merge()* and *split()* operations, and ordering of the items in the aggregate, see the equivalent functions in Collection, in Section 6.1. For Aggregate, the parents do not have to be given in some situations.

The iterator is exactly the same as for the Collection, and is also internally implemented like it:

```
template<class P,class C, int i=0> class AggregateIterator :
                                public CollectionIterator{};
```

The interface therefore appears to be like the following:

```
template<class P,class C, int i=0> class AggregateIterator {
    ...
public:
    // all same as for Collection
};
```

For examples of using this data structure, see *ttest/aggregat.cpp* and *mtest/aggregat.cpp.* The correct results are in *ptl/out/aggregat.out.*

---

## (6.3) ARRAY CONTAINER or BASIC DYNAMIC ARRAY (arraycon.h)

This is a dynamic array in its classical (standard) template form. It is not implemented as a pattern: the data and all functions that control it are in the same class. It has been included for completeness, and also because we will need it for some more sophisticated array implementations.

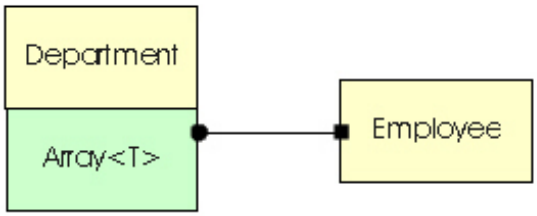
This array does many things; in contrast to STL and other libraries it is centered on implementation, and it lets YOU do what you want. You can control its order, sort it, use it as a collection or array of pointers or as a stack, and many other things.

This class keeps an array of objects, which you can access with the operator[] as if it was a real array. The array starts with a certain size (either default or specified by the constructor), and as you access it, it automatically increases its size as needed. You can also control the size directly with certain commands.

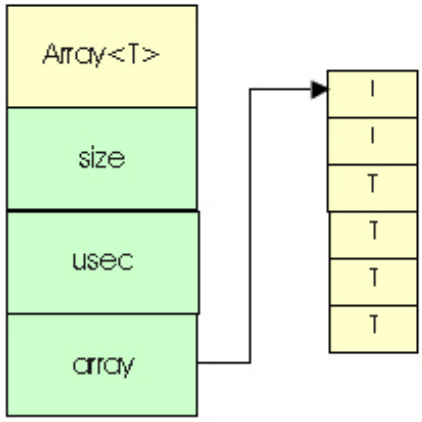
Normally, when a bigger size is needed, the array multiplies the current size by 2 until a sufficient size is reached. If you know the required size of the array, you can avoid excessive allocation by specifying it in the constructor, by calling grow(maxIndex), or accessing it like: *x=myArray[maxIndex];*

The array can be uninitialized, initialized to 0, or blank. This is applied not only to the original, starting array, but also to its unused portion as the array grows. You can specify the initialization method in one of the constructors.

### Class diagram:




Implementation:




where *size* is the currently allocated size of the array, and *used* is the number of items in the used part of the array. For example, if you accessed indexes 2,3, and 9, you will have used=10, based on the highest used index=9. Even though you never accessed items 0,1, etc., your current range is 0-9, therefore the array has 10 items.

In the its current form, we did not provide iterators, because they seem unnecessary. You can traverse the used part of the array like this:

```
for(i=0; i<myArray.used(); i++){ ...=myArray[i];}
```

or you can traverse any part without worrying about the size and allocation like this:

```
for(i=0; i<25000; i++){ myArray[i]=...;}
```

Sorting is based on *qsort,* and expects the same style of compare function. We recommend that you code this function as a static function on the class of the object that is stored in the array. See function *B::cmp()* in the test program *ttest/arraycon.cpp.*

When working with an array of pointers, declare it like this:

```
    class A;
    Array<A*> myArray;
```

**CONDITIONS FOR USING THIS PATTERN:**

This pattern does not involve inheritance. There is no need to use the *Pattern()* statement when you run with the Template Manager.

**AVAILABLE METHODS:**

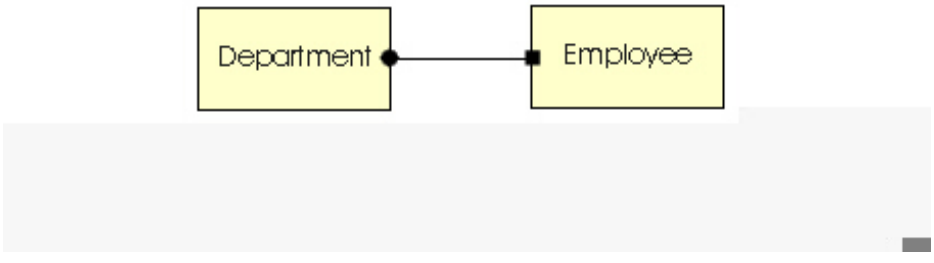
```
template<class T> class ArrayContainer {
typedef int (*cmpType)(const void *, const void *);
    ...
public:
    ArrayContainer(); // default starts with size=8, no initialization
    ArrayContainer(unsigned int const startSize,unsigned int const ini);
                     // user-specified starting size and initialization
                     // ini=0 no initialization,ini=1 by 0,ini=2 by blank
    virtual ~ArrayContainer();
    unsigned int size(void) const; // returns the currently allocated size
    unsigned int used(void) const; // returns the currently used range
    T& operator[](const unsigned int k); // normal access to the array
    void remove(const unsigned int k);   // remove, shrink remaining part
    void fastRemove(const unsigned int k);  // without maintaining order
    void insert(const unsigned int k,T *t); // insert, expand remainder
    void push(T* e); // add e to the end as if a stack, increment 'used'
    T* pop(void);    // return the last item, decrement 'used'
    int reduce();    // reduce the array to its used size
    int cut(const unsigned int k);     // cut the size up to index k
    int grow(const unsigned int k);    // grow to accomodate index k
    void sort(cmpType cmp);            // sort the array, using qsort
};
```

---

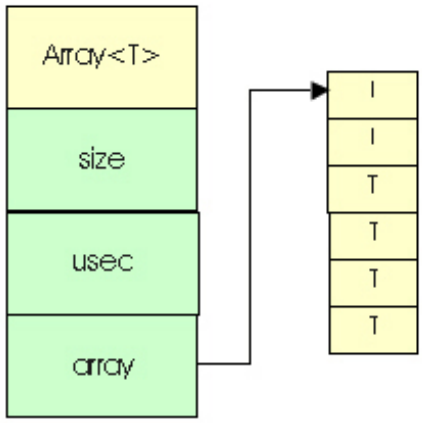
## (6.4) ARRAY attached to an application class (file array.h)

The dynamic array is one of the basic data organizations, like the linked list. Many data structures can be built with dynamic arrays. For example, most container class libraries such as STL, RogueWave tools.h++, and Microsoft MCF extensively use dynamic arrays.

Class diagram:




Implementation:




Class *Array<H,T,i>* is essentially *ArrayContainer<T>* which was described in the previous section, but with a style of interface that fits intrusive data structures and patterns such as *Collection<>* or *Aggregate<>.* The array is attached to application class H, contains an array of Ts, and is managed by the manager class *Array<H,T,i>*. For the internal organization, look at class *ArrayContainer<T>*.

The functions that control this data organization are similar to those available for *ArrayContainer<>*, except that the holder object must always be given. For example, to traverse the used part of the array, you do this:

```
class Holder;
class Element;
Holder *hp;
Element *e;
Array<Holder,Element> myArray; // default i=0 not needed
...
for(i=0; i<myArray.used(h); i++) {
    e = myArray.array(h)[i];
    ...
    myArray.array(h)[i] = e;
}
```

Function *initial()* sets up the default starting size and initialization for all arrays within the class *Array<H,T,i>*. The array grows in multiples of 2 whenever operator[] or push() access an index which overflows the existing size. However, even when a big jump in size occurs, the new size is determined first and then only one allocation is performed. You can increase/decrease the size to a specific value using the functions *grow()* or *cut().*

**CONDITIONS FOR USING THIS PATTERN:**

1. H is a subclass of *ArrayHolder<H,T,i>*
2. T is the type of the objects to store in the array

For arrays of pointers, use *Array<H,T*,i>*, or even better, class *PtrArray<H,T,i>*.

**ITERATORS**

Currently none; use the for() loop as explained above.

**AVAILABLE METHODS:**

```
template<class H,class T,int i=0> class Array {
```

```
typedef int (*cmpType)(const void *, const void *);
    ....
public:
    void initial(unsigned int const startSz,unsigned int const ini);
             // startSz=starting size for all arrays,
             // initialization: 0=no initialization,1=by NULL,2=by blank
    unsigned int size(H* h) const; // returns currently allocated size
    unsigned int used(H* h) const; // returns currently used size
    hType& array(H *h); // main access to the array, to be used with []
    void remove(H *h,const unsigned int k);
                // remove item in position k and shrink remaining part
    void fastRemove(H *h,const unsigned int k);
                // remove item in position k without maintaining order
    void insert(H *h,const unsigned int k,T *t);
                // insert t into position k and expand remaining part
    void push(H *h,T* e); // push e to the end, increment 'used'
    T* pop(H *h);         // pop the last item, decrement 'used'
    int reduce(H *h);     // reduce to the currently used size
    int cut(H *h,const unsigned int k);
                          // cut the size to accomodate index k
    int grow(H *h,const unsigned int k); // grow to accomodate index k
    void sort(H *h,cmpType cmp); // sort using qsort and cmp()
};
```

**Comments:**

- Use *reduce()* if your program has reached a situation where the allocated size is much bigger than you need, and you want to release the unused memory.
- Be careful when using *cut()*. The function will destroy any data above the specified index.

For examples of using this data structure, see *ttest/array.cpp* and *mtest/array.cpp*. The correct results are in *ptl/out/array.out.*

---

## (6.5) POINTER ARRAY (file ptrarray.h)

The previous chapter described DYNAMIC ARRAY, which also can store simple pointers (as an array of pointers). However, handling a pointer array through this generic class is not as easy as one would wish in the case of a simple item like a pointer. There are certain situations in which the number of '*' preceding some variables and parameters becomes confusing, or in which you simply want to pass a pointer in/out without going through a complicated interface. For this purpose, the PTL provides specially tuned class called *PtrArray<>.*

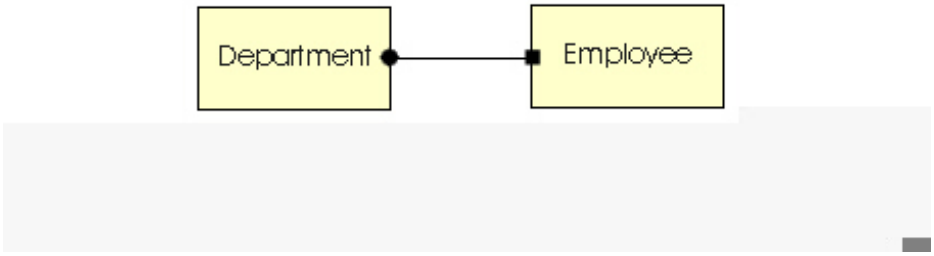
PtrArray is derived from *Array<>,* and reuses many of its functions. The main differences are:

- PtrArray provides functions *set()* and *get()* for simple loading or retrieval of a pointer into the array.
- PtrArray has function *count()*, which returns the number of entries that are not NULL. All access functions automatically maintain this count, except for *operator[]*, which allows the user to do with the object whatever he/she pleases. Any call to *operator[]* makes the count invalid, and on the next call to *count()*, a new count is recovered by a pass over the used part of the array. If you restrain yourself from using *operator[],* function *count()* is very fast.
- Function *remove()* returns the value of the pointer which was removed.
- Function *clean()* follows all the pointers through the array and destroys the objects. Potential crashes caused by attempting to destroy the same object twice are prevented by first sorting the array.
- *push()* and *pop()* work with values of pointers, not their addresses.
- The user cannot control initialization, unless explicitely invoking *Array::initilize()*. Function *PtrArray::initialize()* does nothing.

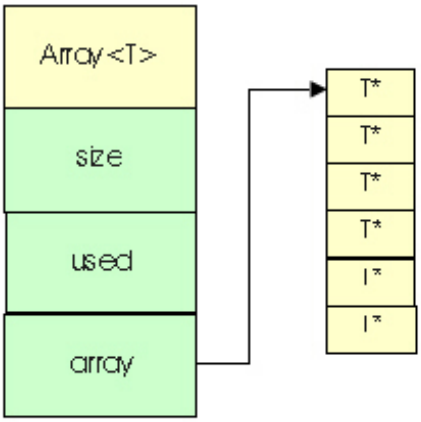
Internally, the PtrArray class is similar to Array, but note the difference in the parametrization - we have *PtrArray<T>,* but

each element of the array is T*

## Class diagram:




## Implementation:




**CONDITIONS FOR USING THIS PATTERN:**

None. When using an index other than *unsigned int*, for example *int* or *unsigned char*, the compiler provides conversion automatically.

**ALGORITHM AND TRAVERSING THE ARRAY**

Size management is the same as for Array. The array is always initialized to NULL pointers. Traversing is the same as for Array. Calling *set(i,NULL)* for a position where the array is not NULL will decrement the count. Function *sort()* is the same as for Array; you may want to sort the array either by the value of the pointers or by some keys which are members of the objects to which the pointers lead.

**AVAILABLE METHODS:**

```
template<class E> class PtrArray : public Array<E*> {
typedef int (*cmpType)(const void *,const void *); // for sort
    ....
public:
    // Methods which are new or different
```

```
    PtrArray():Array<E*>();              // same size default as Array
    PtrArray(const unsigned int s):Array<E*>(s); // given initial size
    virtual ~PtrArray();                 // because ~Array() is virtual
    unsigned int count();                // count of pointers != NULL
    E*& operator[](const unsigned int i); // use as you would intuitively
    E *get(const unsigned int i);        // returns pointer with index i
    void set(const unsigned int i,E *e);  // loads pointer e into index i
    void push(E *e);                     // pushes pointer e onto stack
    E *pop(void);                        // pops pointer from the stack
    void clean(void); // removes all objects to which the pointers lead
    E* remove(const unsigned int i);  // removes pointer at i, returns it
    void insert(const unsigned int i,E *e); // insert e to i, expand array
    void initialize(unsigned int const code); // automatic,control disabled
    // Same interface as for Array
    int reduce(); // reduces the size to fit the useful part of the array
    int cut(const unsigned int i);  // cuts the size to accomodate index i
    int grow(const unsigned int i); // grows the size to accomodate index i
    void sort(cmpType cmp);         // sorts the used part using qsort()
    unsigned int size(void) const; // currently allocated size
    unsigned int used(void) const; // highest index currently used}
};
```

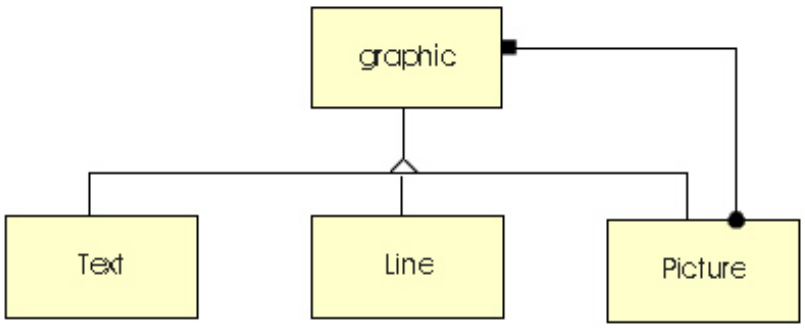
For examples using this PtrArray, see *ttest/ptrarray.cpp* and *mtest/ptrarray.cpp*. The correct results are in *ptl/out/ptrarray.out.*

---

## (6.6) Pattern COMPOSITE (file composit.h)

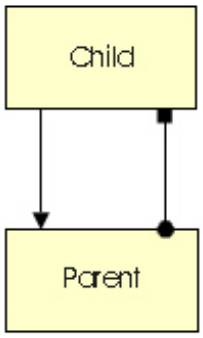
COMPOSITE is one of the most powerful design patterns. It implements an elegant and efficient hierarchy of objects, applicable to uncountable practical situations. For example, assume that you are designing a graphical editor which handles Pictures, Text, and Lines. Text and Line are basic units, but Pictures are composed of Text, Lines, and other Pictures. For example, a rectangle becomes a Picture which includes 4 lines, and a rectangle with text in it is a Picture which includes the Picture representing the rectangle, plus the text.

To apply Composite to this example, we introduce a base class common to all these objects, and call it Graphic, and we assign a collection of Graphic objects to each Picture:




Only after you attempt to draw a pointer diagram of this arrangement, will you appreciate its beauty and simplicity. The

essence of this pattern is




and its implementation depends only on how we implement the collection. Our template *Composite<class Parent, class Child, int i>* is based on the intrusive collection, *Collection<class Parent, class Child, int i>*, which is described in Section 6.1. If you prefer a container-based collection, you can design your own Composite class with *PtrArray<>* (see Section 6.5)

For more details on this pattern, see reference [4], Chap.4 - p.163.

**CONDITIONS FOR USING THIS PATTERN:**

1. P is a subclass of both C and *CompositeParent<P,C,i>*
2. C is a subclass of *CompositeChild<P,C,i>*
3. The user must provide the virtual function isComposite for both P and C.

```
virtual int P::isComposite(Composite<P,C,i> *c){c=c; return(1);}
virtual int C::isComposite(Composite<P,C,i> *c){c=c; return(0);}
```

When you use the Pattern Manager, all this happens automatically.

**AVAILABLE METHODS:**

COMPOSITE is derived from COLLECTION, and has all its methods. ITERATE and RETRACE traverse one level of the hierarchy. There are additional functions which traverse the entire hierarchy:

*depthFirst()* traverses depth first and applies function *f()* to every node;
*breadthFirst()* traverses breadth first and applies function *f()* to every node;
*dissolve()* disconnects the hierarchy under the given parent, but does not destroy the objects.

The function applied by these functions, *int Child::f(const void *v)*, permits you to pass one or more parameters (possibly as a structure). The function should normally return 0; return=1 exits the traversal loop.

```
template<class P,class C, int i=0> class Composite
                          : public Collection<P,C,i> {
    ...
public:
    // methods inherited from Collection<>
    cType *head(const pType *p) const; // head of the list
    cType *tail(const pType *p) const; // tail of the list
    cType *next(const cType *c) const; // next on the list
    cType *prev(const cType *c) const; // previous on the list
    void setTail(pType *p,cType *c) const; // set c to be the tail on p
    void setHead(pType *p,cType *c) const; // set c to be the head on p
```

```
    void addTail(pType *p,cType *c) const; // add c as new tail under p
    void addHead(pType *p,cType *c) const; // add c as new head under p
    void insert(pType *p,cType *c,cType *x) const; // insert x before c
    void append(pType *p,cType *c,cType *x) const; // append x after c
    void remove(pType *p,cType *c) const; // remove c from under p
    void merge(cType *s,cType *t,pType *ps,pType *pt) const; // see below
    void merge(pType *ps,pType *pt) const; // merge pt after ps
    void split(cType *s,cType *t,pType *ps,pType *pt) const; // see below
    void sort(pType *p,cmpType cmp)const; // sort one level using C::cmp()
    void addSorted(pType *p,cType *c,cmpType cmp) const; // add, keep sorted
    int count(const pType* p) const; // count children for this parent
typedef C* (*TravFun)(C*,void *);
    // additional methods that traverse entire hierarchy
    void dissolve(cType *c);
    C *depthFirst(cType *c,TravFun f,void *v);
};
```

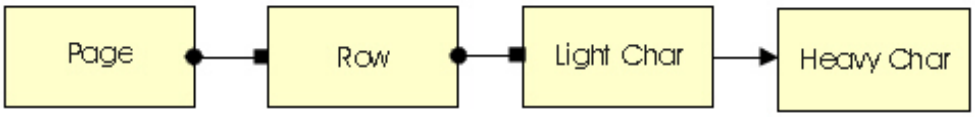
For examples using this data structure, see *ttest/composit.cpp* and *mtest/composit.cpp*. The correct results are in *ptl/out/composit.out.*

---

## (6.7) Pattern FLYWEIGHT (file flyweigh.h)

Despite our regard for the authors of [4], we think that by using a different terminology, the description of this pattern in [4] p.195, can be improved. We will discuss the example of this pattern presented in [4], but will introduce more meaningful names for the classes participating in the pattern.

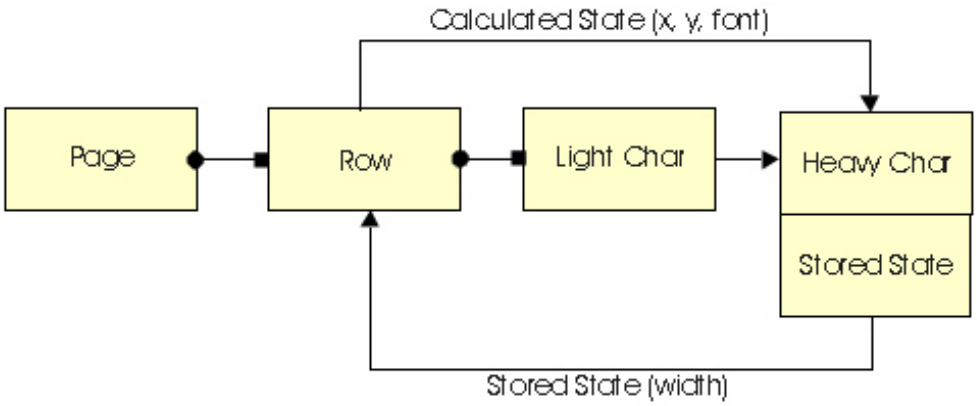
If you are designing a text editor, the text will appear on the screen as rows of characters, and it seems natural to store the data internally in the same manner. However, you must be careful to minimize the storage: even relatively small documents will include thousands of characters. The flyweight pattern achieves this objective with the following arrangement:




Each Page has a collection (or a list) of Rows, and each Row has a collection of LightChar objects. LightChar is designed to take the minimum possible space, and stores the main bulk of text. Each LightChar refers to a HeavyChar object, which is the master template of the character. HeavyChar knows how to draw the character, its width and height, and other details. Most likely, you would represent each row as an array of bytes, where the integer stored in each byte is used as an index into the array of HeavyChar objects, one HeavyChar array for each available font.

In this pattern, some of the important data are not stored but are calculated on the fly. For example, to draw a character, function *HeavyChar::draw()* needs to know the lower-left corner of the character, which is determined by its position in the row. The Row can calculate this information and pass it in some form (we will call this the "calculated state") to HeavyChar. On the other hand, when calculating the position, the Row needs to know the width of all the characters which are stored in

HeavyChar (we will call this the "stored state"). Pattern flyweight is this combination of light/heavy objects with the action of passing the states back and forth.

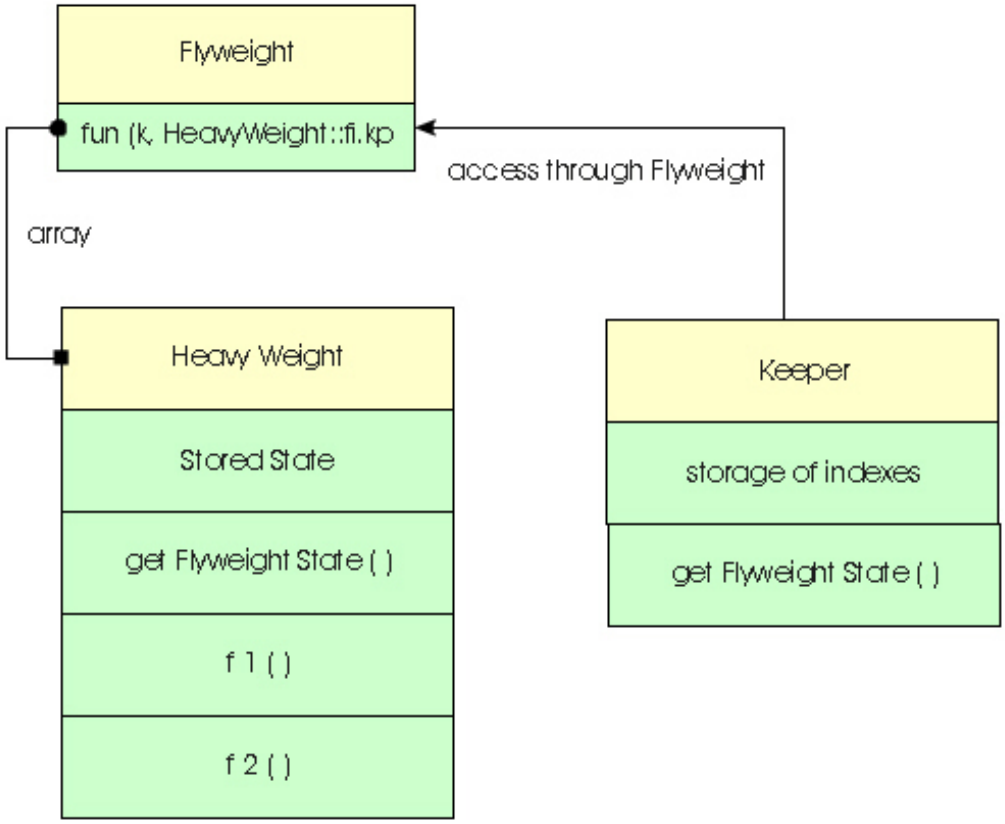



For example, in the test problem *ttest/flyweigh.cpp* or *mtest/flyweigh.cpp*, the calculated state also includes the font. The start of the text set in italics is recorded by character '{', and the change to the normal text by '}'.

The challenge of designing a library class for flyweight is the need to cover several different situations:

- LightChar may be stored not only in arrays, but in some more complex data structures such as a BTree ([4] shows an example of this).
- LightChar may be 1, 2, 4 or more bytes, depending on how many master objects we have.
- Handling multiple Flyweights for the same set of classes must be easy. For example, multiple fonts may be implemented as multiple Flyweights.

For this reason, our Flyweight has the following organization, which does not include an equivalent of class LightChar. It is simply assumed to be an integer of general size, and the size of integers it keeps depends on class Keeper (equivalent of Row), and whether they form an array, tree, or some other data structure.




Note that *HeavyWeight::getFlyweightState()* returns the StoredState, while *Keeper::getFlyweightState()* returns the CalculatedState. Keeper corresponds to Row from the previous example, and HeavyWeight corresponds to HeavyChar. StoredState and CalculatedState have the same meaning as before. Flyweight is the class representing the pattern itself, and it also keeps the array of HeavyWeights. As explained above, there is no equivalent of LightChar, which is assumed to be a general integer.

Function *fun(k,fi,kp)* is an essential part of this pattern, because classes Keeper and HeavyWeight may communicate only through class Flyweight. For more detailed description, see below - AVAILABLE METHODS.

**CONDITIONS FOR USING THIS PATTERN:**

- Class Keeper must be derived from *FlyweightKeeper<Keeper,HeavyWeight,CaculatedState,StoredState,i>*
- Class HEA must be derived from *FlyweightHeavy<Keeper,HeavyWeight,CaculatedState,StoredState,i>*
- Both Keeper and HeavyWeight must be friends of *Flyweight<Keeper,HeavyWeight,CalculatedState,StoredState,i>*

It is recommended that Keeper be a friend of CalculatedState, and HeavyWeight be a friend of StoredState.

When you use the Template Manager,all this happens automatically, including Keeper being a friend of CalculatedState, and HeavyWeight being a friend of StoredState.

**AVAILABLE METHODS:**

Abbreviations:
K=Keeper
H=HeavyWeight
C=CalculatedState
S=StoredState
i=integer index

```
#define kType FlyweightKeeper<K,H,C,S,i>
```

```
#define hType FlyweightHeavy<K,H,C,S,i>
#define fType Flyweight<K,H,C,S,i>
#define aType ArrayContainer<H*>
template<class K,class H,class C, class S,int i=0> class Flyweight
                                                        : public aType {
public:
    Flyweight(); // default starting size (=10) of the HeavyWeight array
    Flyweight(const unsigned int sz); // user-specified starting size
    virtual ~Flyweight(){};
    void add(H *const ip,const unsigned int k);
       // record ip as reference No.k, but only if the position is free
    void force(H *const ip,const unsigned int k){remove(k); add(ip,k);}
       // record ip as reference No.k; when k used, remove the old ref.
    void remove(const int k){(void)(((aType*)this)->remove(k));}
                                        // remove the reference at k
    void remove(const H *ip);
       // search for this entry and remove it. Involves a linear search
    void cleanHeavy(void); // destroy all HeavyWeight objects
    S *getState(const unsigned int k); // get StoredState at reference k
    int fun(const unsigned int k, int (H::*f)(C*), K *ep) const;
       // key function in this pattern - see below
};
```

Function *fun(k,fi,kp)* is pivotal for the cooperation between Keeper and HeavyWeight, which can communicate only through class Flyweight. *fun()* finds the HeavyWeight under reference k, and applies to it the method *HeavyWeight::f()*. The pointer to the keeper is provided as an additional reference which *fun()* can use to obtain the current CalculatedState. *fun()* passes the pointer returned by *f()*.

For examples using this data structure, see *ttest/flyweigh.cpp* and *mtest/flyweigh.cpp*. The correct results are in *ptl/out/flyweigh.out.*

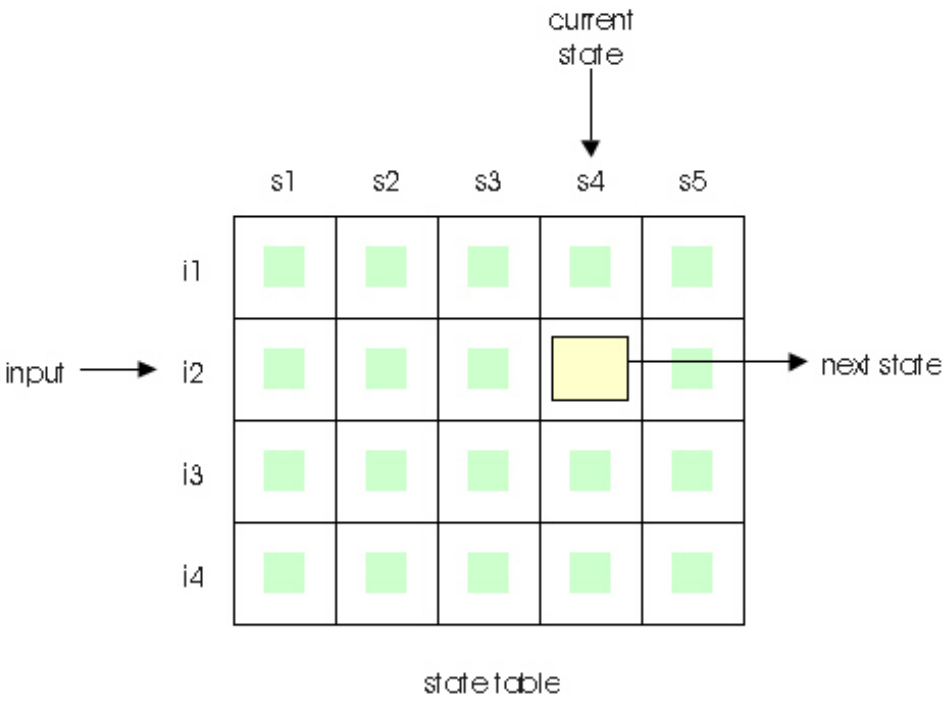
---

## (6.8) FINITE STATE MACHINE- FSM (file fsm.h)

A generic Finite State Machine was not included in the Pattern Book [4], and is not in any commonly used class library. And yet, it is a very important pattern, and many applications depend on its efficient implementation. Our class FSM allows you to create and dynamically modify your own finite state machines that are fast and efficient.

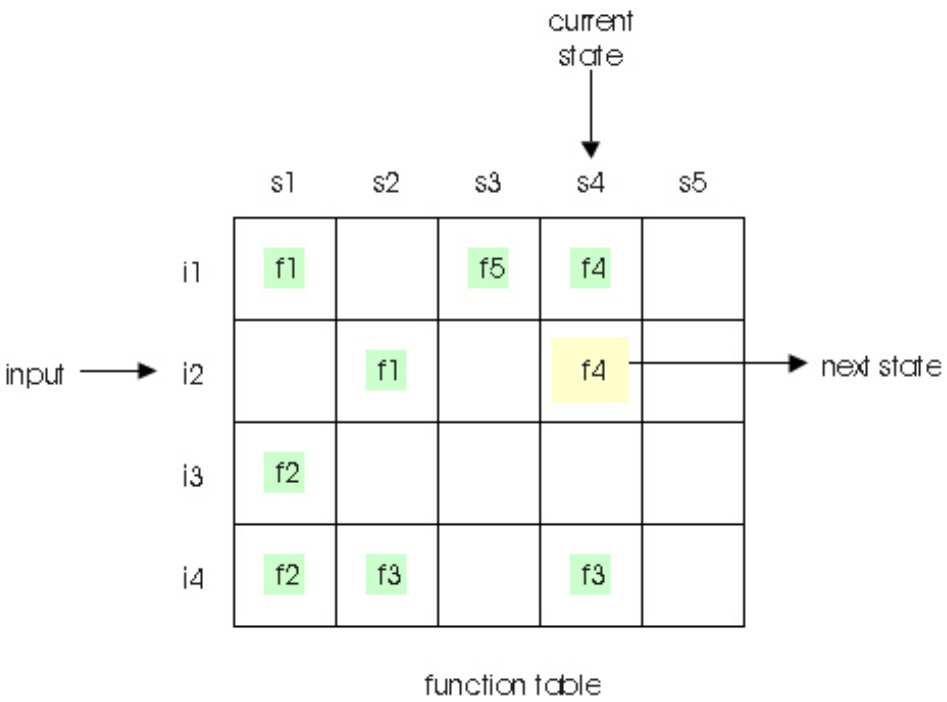
The FSM class assumes that you have a special class for each of your states, with a common base state class. Similarly, you must have a special class for each of the input stimulae, with one common input base class. In many applications, these two classes will really represent an integer or only a byte, but having them as general classes allows you to implement complex and yet highly efficient state machines.

State table:




state table

Function table:




function table

The FSM class is implementated as template *FSM<F,S,I,T,D>* with the following parameters:

```
F = your finite state machine class
S = base class for all your state classes
I = base class for all your input classes
T = type for the internal table index, must be unsigned and able to
    accomodate MAX(numInputs,numStates,numFuncts) entries.
D = integer id when using multiple parallel organizations (as for other
    patterns).
```

As pointed out by J. Coplien (see [5]) this representation is rather unusual because your FSM class will inherit from a template which uses this class as one of the parameters, for example:

```
class myFSM : public MSF<myFSM, myState, myInput, int, 0> { ... };
```

Your FSM class (myFSM) can have an arbitrary number of transition functions that look like this: *S* myFSM::f(S*,void**)*. One function can be assigned to several transitions. The assignment of both functions and of target states is performend dynamically, while executing your program.

Even though this concept is very general, you will find it simple to use once you go through the first example. Here is a state machine which describes the life of a typical university student. One night they go partying; the next day they are tired and study in the evening; they party the next day; and so on. The night before a test they do not go anywhere and try to learn everything at the last minute. When going to a party, they need some beer, and the night before the test, they drink a lot of coffee. If we describe this behavior as a finite state machine, we get the following classes (skipping some unimportant details):

```
    class Student : public FSM<Student,Party,Day,char> {
    public:
        static PartyOrNot *buyBeer(..);
        static PartyOrNot *buyCoffee(..);
    };
    class PartyOrNot : public FSMstate<...> { ... };
    class Day : public FSMinput<...> { ... };
    class GoParty : public PartyOrNot { ... } goParty;
    class StayHome : public PartyOrNot { ... } stayHome;
    class NormalDay : public Day { ... } normalDay;
    class BeforeTest : public Day { ... } beforeTest;
```

You configure the state machine dynamically in your program, using statements like this:

```
    addTrans(&normalDay, &goParty, &stayHome, NULL);
    addTrans(&normalDay, &stayHome, &goParty, Student::buyBeer);
    addTrans(&beforeTest, NULL, &stayHome, Student::buyCoffee);
```

The functions assigned to the Student class may change the state transition, even though they are not used in this way in this example. Function *S* fire(I*)* applies new input to the state machine, and returns its new state. Here is a more detailed description of the *FSM<>* class:

```
template<class F,class S,class I,class T,int D=0> class FSM {
typedef S* (*FSMfun)(S*,void**);
    ...
public:
    FSM(); // Creates a state machine with undefined state (NULL).
           // Internal tables are initialized to the default size:
           // 8 states, 8 inputs, 8 functions
    FSM(int ns,int ni,int nf);  // Same as the default constructor,
           // but internal tables are initialized to the given sizes.
    ~FSM(); // behaviour as expected
    // functions to maintain the state machine:
    void addTrans(I *i, S *s1, S *s2, FSMfun f); // add transition
```

```
        // to state s2, when in state s1 and receiving input i.
        // On this transition, f is executed first. If f returns
        // a nonzero (State*), this state overwrites s2 and is
        // used instead of s2. Function f can be used for other
        // purposes without influencing the next state, and is
        // always executed as a part of the transition, except when
        // f=NULL is given.
        // When i=NULL, this transition applies to all inputs.
        // When s1=NULL, this transition applies to all current states.
        // When s2=NULL, the transition will not change the state.
    void remState(S *s); // remove state and all associated transitions
    void remInput(I *i); // remove input and all associated transitions
    void remTrans(I *i, S *s); // remove some or all transitions
        // remTrans(NULL,NULL) will remove all recorded transitions
    // functions to operate the state machine
    void setState(S *s);  // force the state machine into the given state
    S *currentState();    // report the current state
    S *fire(I *i);        // apply input i, and transfer to the next state
};
```

Transition functions must either be free functions (not assigned to any class), or what we strongly recommend, static functions on your finite state machine class - just as we implemented functions buyBeer() and buyCoffee() above. In function

```
    S *f(S *s, void**);
```

the first parameter is always automatically the current state. The second parameter (void**) provides a convenient way of passing one or more (an array of) additional objects of parameters. Function *f()* may, but need not, return a new state.

For examples using this data structure, see *ttest/fsm.cpp* and *mtest/fsm.cpp*. The correct results are in *ptl/out/fsm.out.*

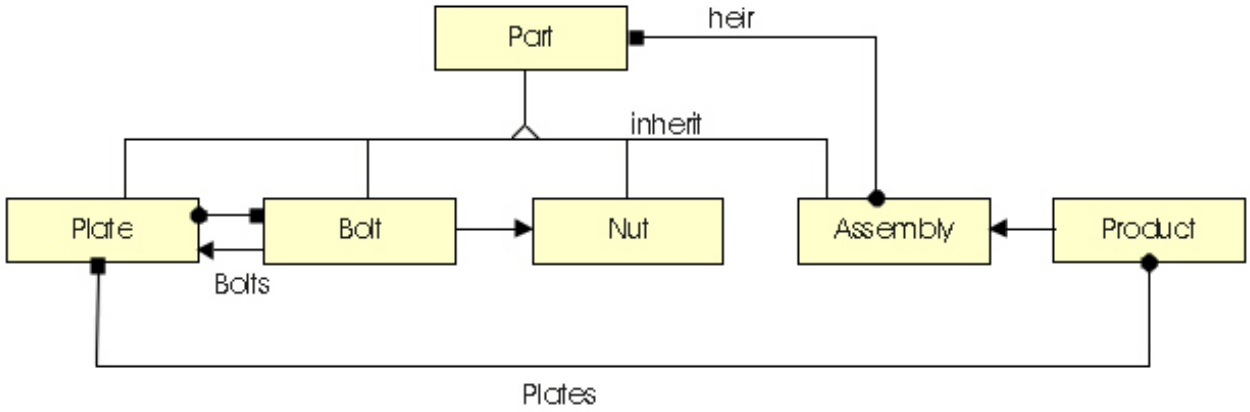
---

# Chap.7: EXAMPLE: SEVERAL PATTERNS AND USING THE TEMPLATE MANAGER

---

This example works with class Product, which represents a complex mechanical device built as a hierarchy of Assemblies. Each Assembly is composed of simple Parts and lower level Assemblies (pattern Composite). Parts can be: Plate, Bolt, or Nut. Each Bolt keeps a pointer to its corresponding Nut. Each Plate keeps a one-to-many relation with the associated Bolts (pattern Agregate), and the Product also keeps a list of all Plates (pattern Collection). As you can see, this is an example of a framework, which is like a small object-oriented database that sits in memory.

The standard class diagram, which does not have symbols for patterns, would look like this:




```
// ------------------------------------------------------------
#include <iostream.h>
#define TEMPLATE_MANAGER
#include "pattern.h"
class Part;
class Plate;
class Bolt;
class Nut;
class Assembly;
class Product;
class Part : Pattern(Part) {
protected:
    int mPartNo;
public:
    Part(int n){mPartNo=n;}
    virtual void prt(int k){};
};
class Plate : public Part, Pattern(Plate) {
                        // inheritance of Part not part of composite
    int mx,my;
public:
    Plate(int n,int x,int y):Part(n){mx=x; my=y;}
    void prt(int k);
};
class Bolt : public Part, Pattern(Bolt) {
                        // inheritance of Part not part of composite
```

```
    int mDiam;
    Nut *mpNut; // knows its own nut
public:
    Bolt(int n,int d,Nut *np):Part(n){mDiam=d; mpNut=np;}
    void prt(int k);
};
class Nut : public Part { // does not need Pattern(..), is not in any pattern
    int mDiam;
public:
    Nut(int n,int d):Part(n){mDiam=d;}
    void prt(int k);
};
class Assembly : Pattern(Assembly) { // covers even inheritance from Part
public:
    Assembly(int n):Part(n){}
    void prt(int k);
};
class Product : Pattern(Product) {
    char *mpName; // product name
    Assembly *mpAssembly; // to the root assembly
public:
    Product(char *name,Assembly *root){mpName=name; mpAssembly=root;}
    void prt(); // print the composition of the entire product
};
void Plate::prt(int k){
    for(int i=0; i<k; i++) cout << "    ";
    cout << "Plate No." << mPartNo << " x=" << mx << " y=" << my << "\n";
}
void Nut::prt(int k){
    for(int i=0; i<k; i++) cout << "    ";
    cout << "Nut No." << mPartNo << " diam=" << mDiam << "\n";
}
void Bolt::prt(int k){
    for(int i=0; i<k; i++) cout << "    ";
    cout << "Bolt No." << mPartNo << " diam=" << mDiam << " -> "; mpNut->prt(0);
}
void Assembly::prt(int k){
    pattern_iterator_hier it;
    Part *pPart;
    for(int i=0; i<k; i++) cout << "    ";
    cout << "Assembly No." << mPartNo << "\n";
    ITERATE(it,this,pPart){
        pPart->prt(k+1);
    }
}
void Product::prt(){
    Plate *pPlate;
    // first print the assembly hierarchy
    cout << "Product=" << mpName << "\n";
    mpAssembly->prt(1);
    cout << "list of all plates:\n";
    pattern_iterator_plates it;
    ITERATE(it,this,pPlate) pPlate->prt(1);
}
int main(void){
   pattern Composite<Assembly,Part,0> hier; // hierarchy of Assemblies
   pattern Aggregate<Plate,Bolt,0> bolts; // 1-to-many relation, both ways
   pattern Collection<Product,Plate,0> plates; // all plates in the product
```

```
   Product *pProduct;
   Assembly *pRootAssembly;
   // using simplified names which permit table-like code in this case
   Assembly *a1,*a2; Plate *p; Bolt *b; Nut *n;
   // create a complex product from plates, bolts, and nuts
   pRootAssembly=new Assembly(1000);
   pProduct=new Product("myProduct",pRootAssembly);
   a1=new Assembly(1100); // assembly one
   p=new Plate(1110,34,22); plates.addTail(pProduct,p); hier.addTail(a1,p);
   n=new Nut(1101,10);                                  hier.addTail(a1,n);
   b=new Bolt(1102,10,n); bolts.addTail(p,b);           hier.addTail(a1,b);
   n=new Nut(1103,10);                                  hier.addTail(a1,n);
   b=new Bolt(1104,10,n); bolts.addTail(p,b);           hier.addTail(a1,b);
   a2=new Assembly(1100); // assembly two
   p=new Plate(1120,72,12); plates.addTail(pProduct,p); hier.addTail(a2,p);
   n=new Nut(1105,8);                                   hier.addTail(a2,n);
   b=new Bolt(1106,8,n); bolts.addTail(p,b);            hier.addTail(a2,b);
   p=new Plate(1130,14,14); plates.addTail(pProduct,p); hier.addTail(a2,p);
   n=new Nut(1107,6);                                   hier.addTail(a2,n);
   b=new Bolt(1108,6,n); bolts.addTail(p,b);            hier.addTail(a2,b);

   // nut & bolt for the root assembly, not linked to any plate
   n=new Nut(1501,8);    hier.addTail(pRootAssembly,n);
   b=new Bolt(1502,8,n); hier.addTail(pRootAssembly,b);
   hier.addTail(pRootAssembly,a1);
   hier.addTail(pRootAssembly,a2);
   pProduct->prt();
   return 0;
}
```

Assuming that you have your environment properly set, you can compile and run it. The result is in file PAT\DOC\OUT7*:*

```
Product=myProduct
    Assembly No.1000
        Nut No.1501 diam=8
        Bolt No.1502 diam=8 -> Nut No.1501 diam=8
        Assembly No.1100
            Plate No.1110 x=34 y=22
            Nut No.1101 diam=10
            Bolt No.1102 diam=10 -> Nut No.1101 diam=10
            Nut No.1103 diam=10
            Bolt No.1104 diam=10 -> Nut No.1103 diam=10
        Assembly No.1100
            Plate No.1120 x=72 y=12
            Nut No.1105 diam=8
            Bolt No.1106 diam=8 -> Nut No.1105 diam=8
            Plate No.1130 x=14 y=14
            Nut No.1107 diam=6
            Bolt No.1108 diam=6 -> Nut No.1107 diam=6
list of all plates:
    Plate No.1110 x=34 y=22
    Plate No.1120 x=72 y=12
    Plate No.1130 x=14 y=14
```

---

# Chap.8: ADDING NEW PATTERNS TO THE LIBRARY

---

Internally, the library provides plenty of examples of how you can derive a new pattern from the existing ones. For example, Aggregate and Composite are derived from Collection; Array, PtrArray, and Flyweight are derived from ArrayContainer; and internally FSM uses five ArrayContainers. Each pattern may require a different coding technique, but if you want it to fit into the framework of this library, and especially if you plan to use the Template Manager, we suggest that you follow the format of the existing *.h files, paying attention to the following details:

(a) Enclose each file with *#ifndef* statements so that multiple includes of the same file do not cause compilation errors. For example:

```
#ifndef AGGREGATE_USED
  .... all code ....
#endif // AGGREGATE_USED
#define AGGREGATE_USED
```

(b) Include patterns which will be used as a part of your new pattern. For example:

```
#include <collecti.h>
#include <arraycon.h>
```

(c) In the comment part, explain the parameters your new pattern/template is using, and also what classes must be inherited by the application. For example:

```
// ---------- aggregat.h --------------------
// pattern Aggregate<P,C,i>
// where P is the parent class (for example Department),
//       C is the child class (for example Employee)
//       i is an integer index, usually 0
// ------------------------------------------------
// Example:
// Class Employee : public AggregateChild<Department,Employee,0> {
//     ...
//};
// Class Department : public AggregateParent<Department,Employee,0> {
//     ...
//};
// ------------------------------------------------
```

(d) Replace long templates by shorter symbols - it will make the code more readable. For example:

```
#define pType AggregateParent<P,C,i>
#define cType AggregateChild<P,C,i>
#define iType AggregateIterator<P,C,i>
#define oType Aggregate<P,C,i>

#define pTypeC CollectionParent<P,C,i>
#define cTypeC CollectionChild<P,C,i>
#define iTypeC CollectionIterator<P,C,i>
#define oTypeC Collection<P,C,i>
```

(e) Code the class definition and the implementation of the template methods in a single file, including the part which (if this was not a template) would be in the *.cpp file:

```
// part equivalent to *.h file
class XYZ {
```

```
   ...
    void prt();
};
// part equivalent to *.cpp file
void XYZ::prt(){
   ...
}
...
```

(f) Provide hooks for the Template Manager. To understand how this works, consider the following. If your application declares *pattern Aggregate<Department,Employee,0> myAggr;* class Department is given No.1, and class Employee is given No.2. As a result, the statement

```
class Department : Pattern(Department) {
    ... whatever
```

will be translated into

```
class Department :
            AggregateInherit1(myAggr,Department,Employee,0){
    AggregateMember1(myAggr,Department,Employee,0)
    ... whatever
```

This permits you to arrange that the Template Manager transparently implements inheritance, but can also add members or functions to the class. For example, the setup for pattern Aggregate which is based on class Collection looks like this:

```
#ifdef TEMPLATE_MANAGER
// Include sections - needed only when the Template Manager is used
#define AggregateInherit1(id,par,chi,i) \
                        public AggregateParent<par,chi,i>
#define AggregateMember1(id,par,chi,i)
#define AggregateInherit2(id,par,chi,i) \
                        public AggregateChild<par,chi,i>
#define AggregateMember2(id,par,chi,i)
// -----------------------------------------------
#endif  // TEMPLATE_MANAGER
```

(g) Cancel the shorthand for your classes; otherwise, there could be a conflict with other patterns:

```
#undef pType
#undef cType
#undef iType
#undef oType
#undef pTypeC
#undef cTypeC
#undef iTypeC
#undef oTypeC
```

(h) Write a test program for your new class. Test it both with and without the Template Manager, place the two copies (there should be only minor differences due to the style of using the patterns) in directories MTEST and TTEST, and the correct result in the directory OUT.

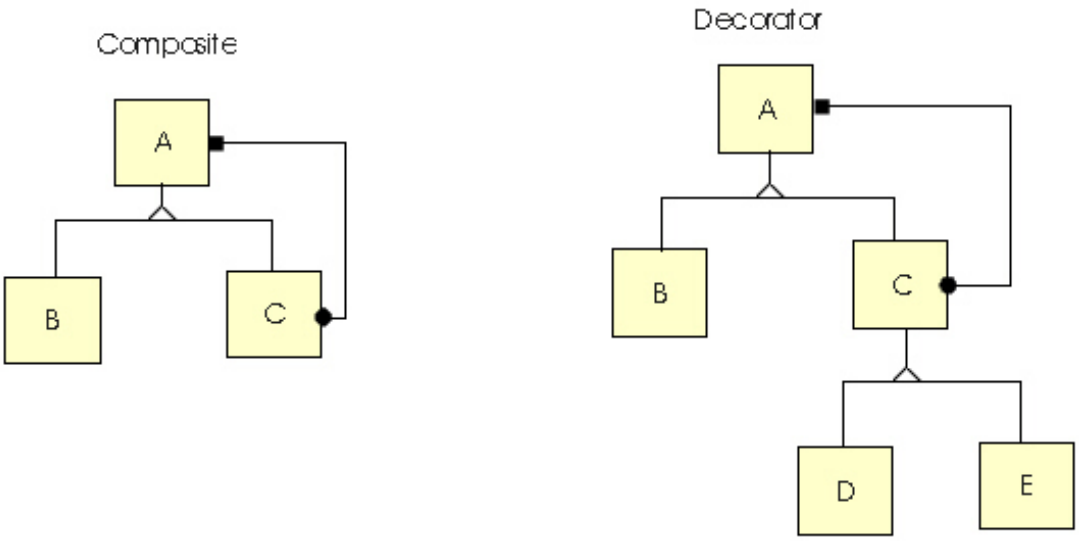
---

# Chap.9: MISSING PATTERNS

Reference [4] lists several structural patterns which are simple inheritance trees applied to some particular situations. Since there is no other structure involved (collections or pointers), these patterns are not good candidates for this library. When working with these patterns, the main work is in setting up application-specific functions, and this work must be done by the user whether or not there is a library class.

For this reason, the following patterns are not included in this library: **Adapter, Bridge.**

Another reason for not including some other patterns is that they can be implemented easily with patterns that are already available. For example, from the structural point of view, **Decorator** is just Composite except for a few additional subclasses:




When you need a Decorator, simply use Composite from the library, and derive classes D and E from C.

Our plan is to gradually expand this library depending on user needs and requirements. Probably, we will introduce the following classes in the near future:

- **CONTAINER**
  will be derived from *PtrArray*<>, and provide one class for all the containers typically provided in other libraries: collection, ordered collection, vector, set, bag, etc. It will also have an iterator which is not provided for Array and PtrArray.
- **TREE**
  will be derived from *Aggregate*<>, with iterators for depthFirst and breadthFirst search. The search function will execute a user coded function for each object in the tree, and abort when this function returns 1.
- **MANY_TO_MANY RELATION**
  will use 3 classes: Source, Relation, and Target, with aggregates between Source and Relations, and between Target and Relation.

# Chap.10: ERROR HANDLING

---

There are two types of error situations that can happen in this library:

1. The library detects that the requested operation would corrupt data integrity. A warning message is issued, and the operation is bypassed (not executed). For example, this happens if you want to insert object A into a Collection before object B, but B is not in any Collection.
2. The library detects data which is corrupted beyond repair, and the program will crash sooner or later. An error message is issued, and the library throws an exception using class PTLerrorHandler. For example, this happens if you attempt to destroy an object which still participates in a Collection or if, in an Aggregate, object A is one of the children under parent B, but the parent pointer on A points to C, where C!=B.

Each application has different requirements and style of error handling. For this reason, the library has centralized error handling which permits you to customize it by changing just a few lines in file LIB/MGR.H. The code here speaks better than a long explanation:

```
// ---------- ERROR HANDLING FOR THE ENTIRE LIBRARY -----------------
#include <iostream.h>

class PTLerrorHandler{
public:
    char *msg;
    void *ptr;
    PTLerrorHandler(char *m,void *p){msg=m; ptr=p;}
};

void PTLerror(char *msg,void *ptr){
    cout << "ERROR: " << msg << " " << (unsigned long)ptr << "\n";
    cout.flush();
    throw PTLerrorHandler(msg,ptr);
}

void PTLwarning(char *msg,void *ptr){
    cout << "WARNING: " << msg << " " << (unsigned long)ptr << "\n";
    cout.flush();
}
```

In the case of any error, the library calls either *PTLerror()* or *PTLwarning()*, and these functions print the given message and the given address or other value. For example:

```
MyObject *p; int i;
   ...
PTLerror("destroying object which participates in a collection, obj=",p);
   ...
PTLwarning("index already occupied, i=",(void*)i);
```

Function *PTLerror()* throws exception:

```
           throw PTLerrorHandler(msg,ptr);
```

It just depends on you how you will handle library errors. You can use the mechanism supplied with the library and catch the exceptions:

```
try {
   ... your normal code
}
catch(PTLerrorHandler& eh){
```

```
   ... eh.msg has the message
   ... eh.ptr has the pointer or other value
}
```

If you don't want to abort the run even the in case of a serious error, comment out the line which throws the exception. If you want the messages on a different I/O stream, replace *cout* in functions *PTLerror()* and *PTLwarning().* If you don't want to see any message, comment out the two lines that print them. If you want something completely different, supply new source for functions *PTLerror()* and *PTLwarning()*.

Note that some compilers require a special option for *stack unwinding*. For example Microsoft Visual C++ must be compiled with *-GX flag...*, while the Borland C++ compiler does not require any special option. If you plan to catch the exceptions, this option must be added to the appropriate file PTL\ENVIR*.cmp, for example file *msft.cmp.*

---

This is the end of the last Chapter.

# REFERENCES

---

[1] Soukup J.: Intrusive Data Structures, article published in the C++ Report, in three parts between May and October 1998. (This article describes how to design a template library for intrusive data structures, and compares this library with traditional, container based class libraries).

[2] Soukup J.: Taming C++, Pattern Classes and Persistence for Large Projects, Addison-Wesley 1994, ISBN 0-201-52826-6 (This book explains the unified coding style for data structures and patterns, using manager classes.)

[3] Soukup J.: Quality Patterns (Implementing Patterns), The C++ Report, Vol.8, No.9, Oct.1996, pp.34-47.

[4] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley 1995, ISBN 0-201-63361-2

[5] Coplien J.: "Curiously reccuring template patterns", The C++ Report, Vol.7, No.2, Feb.1995, pp.24-27.

---

Root of the Users Guide

# PATTERN TEMPLATE LIBRARY

# USER GUIDE

**Code Farms Inc.** March 1997

## IMPORTANT

The Pattern Template Library (PTL) is based on **intrusive data structures** (see [1]), and it is coded (and must be used) in a completely different way than any other class library you may have seen. This manual is very short, but it is important that you read it without skipping any parts. Otherwise, you are likely to run into difficulties when using the library.

The manual will take you through the following steps:



If you are interested in the overall design philosophy of this library or in the comparision of this approach with container-based class libraries such as STL or Rogue Wave, read article [1].

Classes from this library can be mixed with classes from any other class library.

**Note:** If you browse through the documentation and some figures seem to be missing, you just clicked too fast. If you want to see the figures, wait until the hourglass prompt disappears before you proceed to the next item.

---

continue ...

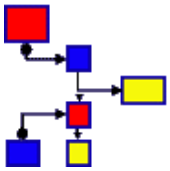
Code Farms Home Page

# Code Farms, Inc.

Visitors since January 1, 2000

We are the leading provider of components for complex software projects which must run fast and be efficient in memory and disk space. We also developed a methodology which allows to design and debug production quality software 5-times faster than using other methods. Our products include C++ class libraries with special emphasis on data integrity and automatic persistency, software for rapid design of customized OO databases and frameworks, intrusive data structures, and a library of design patterns. We were the first company to introduce a complete regression test suite, and we still ship it free of charge.

C++ Data Object Library includes the most extensive data structures on the market, plus three options for fast, automatic persistence. Ideal for large, complex C++ projects. It received the Jolt Award for the best library/language product. One reviewer called it 'the silver bullet'.



Pattern Template Library is a library which lets you use design patterns as if they were simple collections. It provides collections, aggregates, arrays, patterns Composite, Flyweight, and a fast, dynamically reconfigurable Finite State Machine - all coded with pure C++ templates.

Persistent Pointer Factory is a Persistent Pointer Class which, automatically, makes any program or library persistent. This entry lets you to download a free copy and preview this revolutionary product.


What's New

Free Downloads

Scope and Comparison of our Products

---

Contact us at info@codefarms.com or our address is:

7214 Jock Trail, Richmond, Ont., KOA 2ZO, Canada

Tel: (613) 838-4829
Fax: (613) 838-3316

This page is best viewed with


## Best Data Structures and Automatically Persistent Data

## Now Available Free!!