Composition of Classes

Composition: an organized collection of components interacting to achieve a coherent, common behavior.

Why compose classes?

Permits a "lego block" approach to design and implementation:

Each object captures one reusable concept.

Composition conveys design intent clearly.

Improves readability of code.

Promotes reuse of existing implementation components.

Simplifies propagation of change throughout a design or an implementation.

©2001 McQuain WD & Keller BJ

Composition by Aggregation

Aggregation (containment)

Example: an `Employee` object contains an `Address` object which encapsulates related information within a useful package.

The objects do not have independent existence; one object is a component or sub-part of the other object.

Neither object has "meaning" without the other.

Aggregation is generally established within the class definition. However, the connection may be established by pointers whose values are not determined until run-time. (Physical containment vs linked containment.)

Sometimes referred to as the “has-a” relationship.

©2001 McQuain WD & Keller BJ

Advantages of Aggregation

Simplicity – client can deal directly with the containing object (the aggregating object or aggregation) instead of dealing with the individual pieces.

Safety – sub-objects are encapsulated.

Specialized interface – general objects may be used together with an interface that is specialized to the problem at hand.

Structure indicates the designer's intention.

Can substitute implementations.

©2001 McQuain WD & Keller BJ

Static *vs* Dynamic Aggregation

Static – the number of sub-objects does not vary.

- a person has a name and an address
- a rectangle has a NW corner and a height and a width

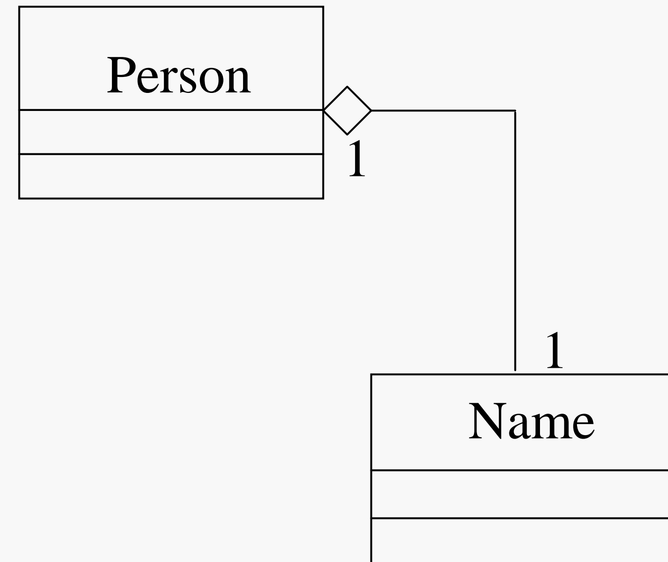
Dynamic – the number of sub-objects may vary.

- a catalog may have many items, and they may be added/deleted
- a host list has a changing list of entries

©2001 McQuain WD & Keller BJ

Aggregation in Class Diagrams

This is similar to the representation of an association relationship except that the arrow is rooted in a diamond instead of a circle.



Cardinality is indicated in the same manner. For a dynamic aggregation, the cardinality for the aggregated type (Name here) would be either a range, such as 0..n or an asterisk.

©2001 McQuain WD & Keller BJ

Simple Aggregations

An `Address` object physically contains a number of constituent objects:

```
class Address {
private:
   string Street1,
          Street2;
   string City;
   string State;
public:
   Address();
   // irrelevant for now
   ~Address();
};
```

```
class Name {
private:
   string First,
          Middle,
          Last;
public:
   Name();
   // irrelevant for now
   ~Name();
};
```

For instance, the object `City` is created when an `Address` object is created and destroyed when that object is destroyed. For our purpose, the `City` object has no meaning aside from its contribution to the `Address` object.

 ©2001 McQuain WD & Keller BJ

A More Interesting Aggregation

A `Person` object physically contains an `Address` object and a `Name` object:

```
enum Gender {MALE, FEMALE, GENDERUNKNOWN};

class Person {
private:
   Name    Nom;       // sub-object
   Address Addr;      // sub-object
   Person* Spouse;    // association link
   Gender  Gen;       // simple data member
public:
   Person();
 // mostly irrelevant for now
   ~Person();
};
```

There is also a provision in the `Person` object for an association with another `Person` object.

©2001 McQuain WD & Keller BJ

Typical Aggregation Construction/Destruction

In a typical aggregation, where the sub-objects are data members (not allocated dynamically), the following rules hold for constructor and destructor sequencing:

Construction: the default constructor is invoked for each sub-object, then the constructor for the containing object is invoked.

So, aggregates are constructed from the inside-out.

Destruction: the destructor is invoked for the containing object first, and then the destructor for each sub-object is invoked.

So, aggregates are destructed from the outside-in.

There is no default initialization for simple data members. Those should be handled explicitly in the constructor for the "enclosing" object.

 ©2001 McQuain WD & Keller BJ

`Person` Constructor

The `Person` constructors must manage sensible initialization of the simple data members:

```
Person::Person() {
   Spouse = NULL;
   Gen    = GENDERUNKNOWN;
}

Person::Person(Name N, Address A, Gender G) {
   Nom    = N;
   Addr   = A;
   Spouse = NULL;
   Gen    = G;
}
```

It is not necessary for the default constructor to manage initializing the sub-objects `Nom` and `Addr` since the default constructors for their types will be invoked automatically.

©2001 McQuain WD & Keller BJ

Example

Consider the trivial program below:

```
int main() {
    Person P;
}
```

```
Constructing default Name
Constructing default Address
Constructing default Person
Destructing Person
Destructing Address
Destructing Name
```

The constructors and destructors were instrumented so that we can see when they are invoked.

Obviously, this is consistent with the stated rules for aggregate construction and destruction.

©2001 McQuain WD & Keller BJ

Aggregation Example: Simple Array of Objects


```
#include <iostream>
using namespace std;
#include "DisplayableNumber.h"
const int Digits = 5;

void main() {
   DisplayableNumber* LCD = new DisplayableNumber[Digits];

   for (int Idx = 0; Idx < Digits; Idx++) {
        LCD[Idx].Show();
   }
   delete [] LCD;
}
```

```
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
0
0
0
0
0
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
```

If the constructors and destructors are instrumented, this program produces the output shown at right.

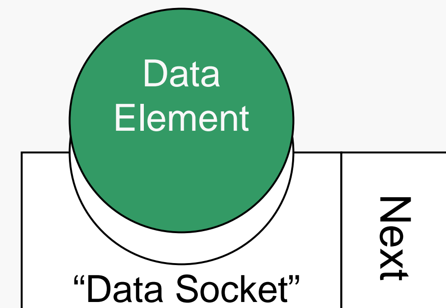
©2001 McQuain WD & Keller BJ

Composition for Flexibility

The use of composition promotes the reuse of existing implementations, and provides for more flexible implementations and improved encapsulation:

Here we have a design for a list node object that:

- separates the structural components (list pointers) from the data values
- allows the list node to store ANY type of data element…
- without needing any knowledge of that type




```
class LinkNode {
private:
   Item      Data;      // data "capsule"
   LinkNode* Next;      // pointer to next node

public:
   LinkNode();
   LinkNode(Item newData);
   void setData(Item newData);
   void setNext(LinkNode* newNext);
   Item getData() const;
   LinkNode* getNext() const;
};
```


©2001 McQuain WD & Keller BJ

Aggregation via Containers

One of the most common scenarios for aggregation is the use of some sort of container object to organize a collection of objects of some type.

The following slides present the interface for a simple array class that can be used with any type of data element.

To achieve that, we assume that the name of the type to be stored has been mapped to a dummy type name, which is then used in the class implementation.

For example,

```
typedef Person Item;
```


©2001 McQuain WD & Keller BJ

A Simple Container Class

```
class Array {          // static-sized array encapsulation
private:
   int   Capacity;    // maximum number of elements list can hold
   int   Usage;       // number of elements list currently holds
   Item* List;        // the list

   void ShiftTailUp(int Start);
   void ShiftTailDown(int Start);

public:
   Array();                                 // empty list of size zero
   Array(int initCapacity);                 // empty list of size initCapacity
   Array(int initCapacity, Item Value);     // list of size initCapacity,
                                            //    each cell stores Value
   Array(const Array& oldArray);            // copy constructor

   int  getCapacity() const;                // retrieve Capacity
   int  getUsage() const;                   //          Usage
   bool isFull() const;                     // ask if list is full
   bool isEmpty() const;                    //             or empty
// continues . . .
```


©2001 McQuain WD & Keller BJ

A Simple Container Class


```
// . . . continued
   bool InsertAtTail(Item newValue);      // insert newValue at tail of list
   bool InsertAtIndex(Item newValue, int Idx);// insert newValue at specified
                                              //    position in List

   bool DeleteAtIndex(int Idx);        // delete element at given index
   bool DeleteValue(Item Value);       // delete all copies of Value in list

   Item Retrieve(int Idx) const;       // retrieve value at given index
   int  FindValue(Item Value) const;   // find index of first occurrence of
                                       //    given value

   void Clear();                       // clear list to be empty, size zero
   ~Array();                           // destroy list (deallocate memory)
};
```


©2001 McQuain WD & Keller BJ

Aggregation Example: Fancy Array of Objects


```
#include <iostream>
using namespace std;

#include "Array.h"
const int Digits = 5;

void main() {
   ofstream Out("LCD.out");
   Array LCD(Digits, Item(0, &Out));

   for (int Idx = 0; Idx < Digits; Idx++) {
      LCD.Retrieve(Idx).Show();
   }
}
```

Note use of a nameless, or "anonymous" object.

Note use of member function of the returned `DisplayableNumber` object.

This "chaining" is made possible by having `Retrieve()` return an object.

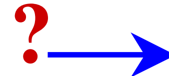
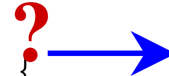
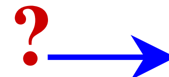
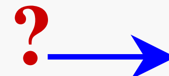
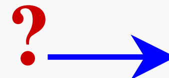
`Array.h` #includes `Item.h`, which contains:

```
#include "DisplayableNumber.h"

typedef DisplayableNumber Item;
```


©2001 McQuain WD & Keller BJ

Aggregation Example: Fancy Array of Objects


```
#include <iostream>
using namespace std;

#include "Array.h"
const int Digits = 5;

void main() {
   Array LCD(Digits, Item(0, &cout));

   for (int Idx = 0; Idx < Digits; Idx++) {
      LCD.Retrieve(Idx).Show();
   }
}
```

```
Constructing: DisplayableNumber
Constructing: Array
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
Destructing: Array
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
```

If the constructors and destructors are instrumented, this program produces the output shown.

There are a few subtleties illustrated here…

©2001 McQuain WD & Keller BJ

Aggregation and Constructor Sequencing

```
const int Digits = 5;
. . .
Array LCD(Digits, Item(0, &cout));
```

```
Constructing: DisplayableNumber
Constructing: Array
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
Constructing: DisplayableNumber
```

The sub-objects are constructed AFTER the `Array` object is constructed. The reason is clear if the `Array` constructor is examined:

```
Array::Array(int initCapacity, Item Value) {

   Capacity = initCapacity;
   Usage    =  Current = 0;
   List     = new Item[initCapacity];

   for (int Idx = 0; Idx < Capacity; Idx++)
      List[Idx] = Value;

   Usage = Capacity;
}
```

Here, the array elements don't exist until the `Array` constructor creates the array dynamically.

©2001 McQuain WD & Keller BJ

Aggregation and Destructor Sequencing


```
#include <iostream>
using namespace std;

#include "Array.h"
const int Digits = 5;

void main() {
   Array LCD(Digits, Item(0, &cout));

   . . .
}
```

```
. . .
Destructing: DisplayableNumber
Destructing: DisplayableNumber

. . .

Destructing: Array
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
Destructing: DisplayableNumber
```

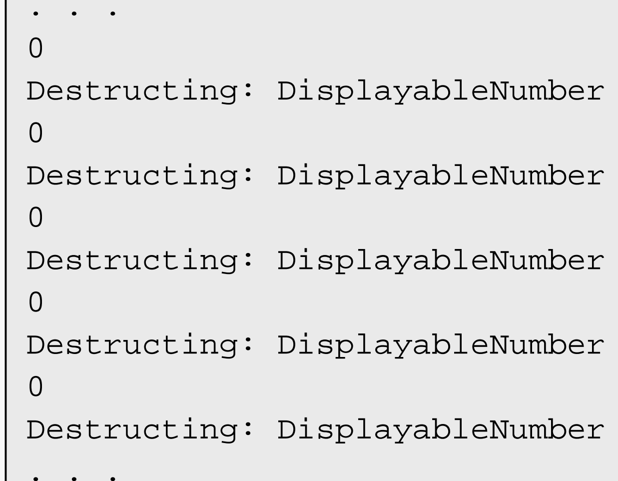
Why are there two `DisplayableNumber` destructor invocations associated with the creation of the `Array` object `LCD`?

Why and when do the destructor calls at the bottom occur?

©2001 McQuain WD & Keller BJ

Anonymous Objects and Returned Objects

```
. . .
for (int Idx = 0; Idx < Digits; Idx++) {
   LCD.Retrieve(Idx).Show();
}
. . .
```



```
. . .
0
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
0
Destructing: DisplayableNumber
. . .
```

On each pass through the `for` loop, an anonymous object is created and then destructed (when its lifetime ends at the end of the loop body).

But, why are no constructor calls shown?

```
Item Array::Retrieve(int Idx) const {

   if (Idx >= Usage)
      return Item();
   else
      return List[Idx];
}
```

Because the return value is created by a copy constructor (not instrumented).

©2001 McQuain WD & Keller BJ

One Last Aggregation Puzzler


```
class Foo {
private:
   DisplayableNumber DN;
public:
   Foo();
   Foo(DisplayableNumber DN);
   ~Foo();
};
```

```
Foo::Foo() {
   cout << "Constructing: Foo" << endl;
   DN = DisplayableNumber(0, &cout);
}

Foo::Foo(DisplayableNumber DN) {
   cout << "Constructing: Foo" << endl;
   this->DN = DN;
}

Foo::~Foo() {
   cout << "Destructing: Foo" << endl;
}
```

Now, the `DisplayableNumber` is a data member; it is created when a `Foo` object is created.

(physical aggregation)

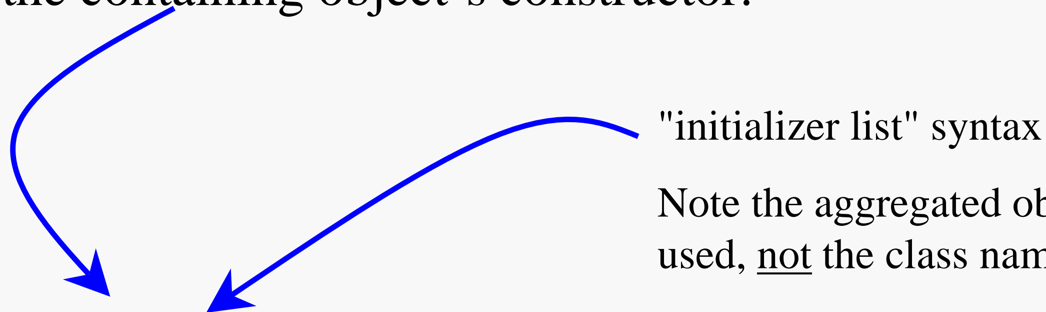
```
DisplayableNumber myDN(17, &cout);

Foo urFoo(myDN);
```

```
Constructing: DisplayableNumber
. . .
Constructing: DisplayableNumber
Constructing: Foo
Destructing: DisplayableNumber
. . .
Destructing: Foo
Destructing: DisplayableNumber
Destructing: DisplayableNumber
```


©2001 McQuain WD & Keller BJ

Controlling Construction of Aggregated Objects

Non-default constructors for sub-objects may be explicitly invoked BEFORE the body of the containing object's constructor:



"initializer list" syntax

Note the aggregated object name is used, not the class name.

```
Foo::Foo() : DN(0, &cout) {

   cout << "Constructing: Foo" << endl;
}
```

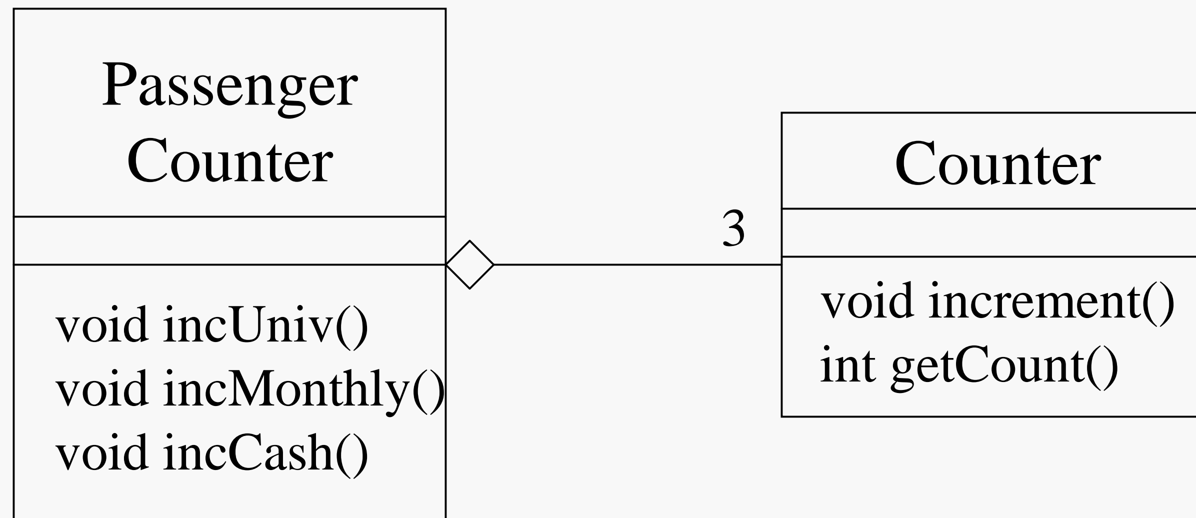
Here, a `DisplayableNumber` constructor is invoked and passed two parameters, which could have been parameters to the `Foo` constructor if that had been appropriate:

```
Foo::Foo(int Init) : DN(Init, &cout) {

   cout << "Constructing: Foo" << endl;
}
```


©2001 McQuain WD & Keller BJ

Extended Example

Consider a system for keeping track of passengers in a bus system, keeping a counter for bus passengers using each of several payment methods:



©2001 McQuain WD & Keller BJ

`Counter` Class

We will employ a `Counter` class:

```
class Counter {
private:
   int Cnt;
public:
   Counter() : Cnt(0) {}
   Counter(const Counter& C) : Cnt(C.Cnt) {}
   Counter(int iCnt) : Cnt(iCnt) {}
   void Increment() { Cnt++; }
   int  getCount() const { return Cnt; }
   ~Counter() {}
};
```

Implicit inlining is used, mainly for simplicity of presentation here.

The integer data member is initialized using the initializer list.

The second constructor is a copy constructor.

©2001 McQuain WD & Keller BJ

PassengerCounter Class

We will employ a PassengerCounter class:

```
class PassengerCounter {
private:
  Counter UnivID, Monthly, Cash;
public:
  PassengerCounter();
  PassengerCounter(const PassengerCounter& P);
  void incUnivID();
  void incMonthly();
  void incCash();
  int  getUnivIDCount() const;
  int  getMonthlyCount() const;
  int  getCashCount() const;
  void Summarize(ostream& Out) const;
  ~PassengerCounter();
};
```


©2001 McQuain WD & Keller BJ

PassengerCounter Implementation

Constructors:

```
PassengerCounter::PassengerCounter() :
       UnivID(), Monthly(), Cash() {}

PassengerCounter::PassengerCounter(const PassengerCounter& P) :
       UnivID(P.UnivID), Monthly(P.Monthly),
       Cash(P.Cash) {}
```

Mutators:

```
void PassengerCounter::incUnivID() {
       UnivID.Increment();
}
void PassengerCounter::incMonthly() {
       Monthly.Increment();
}
void PassengerCounter::incCash() {
       Cash.Increment();
}
```


©2001 McQuain WD & Keller BJ

PassengerCounter Implementation

Accessors:

```
int PassengerCounter::getUnivIDCount() const {
        return UnivID.getCount();
}
int PassengerCounter::getMonthlyCount() const {
        return Monthly.getCount();
}
int PassengerCounter::getCashCount() const {
        return Cash.getCount();
}
```

Display function:

```
void PassengerCounter::Summarize(ostream& Out) const {

   Out << "Payment summary:" << endl << endl;
   Out << "University ID  |" << setw(5)
                             << getUnivIDCount() << endl;
   Out << "Monthly pass   |" << setw(5)
                             << getMonthlyCount() << endl;
   Out << "Cash           |" << setw(5)
                             << getCashCount() << endl;
}
```


©2001 McQuain WD & Keller BJ

PassengerCounter Driver

Driver to test the PassengerCounter class:

```
int main() {
   PassengerCounter RiderStats;
   srand( (unsigned)time( NULL ) );

   for (int i = 0; i < 100; i++) {

      int payType = rand() % 3;
      switch (payType) {
      case 0:   RiderStats.incUnivID();
                break;
      case 1:   RiderStats.incMonthly();
                break;
      case 2:   RiderStats.incCash();
                break;
      default:  break;
      };
   }
   RiderStats.Summarize(cout);
   return 0;
}
```

```
Payment summary:

University ID  |   42
Monthly pass   |   31
Cash           |   27
```


©2001 McQuain WD & Keller BJ

Aside: Equality

The design of an equality operation poses some questions which require careful consideration. The same applies to any of the relational operators.

When are two `Counter` objects equal

When are same object?

When have same value?

Which is more appropriate depends on class and how it is used.

Whichever is appropriate, you can provide an overloaded equality operator for the class.

The point is that the meaning of equality requires a formal definition, and it may well not mean that the objects have identical content.

©2001 McQuain WD & Keller BJ

Equality: Implementations

If equality means the objects are the same object, compare the addresses of the objects:

```
bool Counter::operator==(const Counter& Other) {
  return (this == &Other);
}
```

This would be silly unless you are using pointers to `Counter` objects.

If equality means the objects store the same counter value, compare the `Cnt` members of the objects:

```
bool Counter::operator==(const Counter& Other) {
  return (Cnt == Other.Cnt);
}
```


©2001 McQuain WD & Keller BJ

Relational Operators in General

If objects of a class will routinely be stored in a container, the class should provide overloadings for at least some of the relational operators.

In order to perform searches and sorts, the container object must be able to compare the stored objects. There are several approaches:

- use accessor members of the stored objects and compare data members directly
- use comparison member functions of the stored objects, as opposed to operators, to compare the data members
- use overloaded relational operators provided by the stored objects

The first requires the container to know something about the types of the data members being compared.

The second requires the stored objects to provide member functions with constrained interfaces.

The third allows natural, independent design on both sides.

©2001 McQuain WD & Keller BJ

< (less than): Implementation

There is no particular uncertainty about the correct definition of a less-than operator for the `Counter` class:

```
bool Counter::operator<(const Counter& Other) {
  return (Cnt < Other.Cnt);
}
```

Other relational operators can be overloaded as easily.

It is insignificantly more expensive to overload all six relational operators than to overload only one or two.

©2001 McQuain WD & Keller BJ