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.

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.

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 and system abstraction.

Can substitute implementations.

Static *vs* Dynamic Aggregation

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

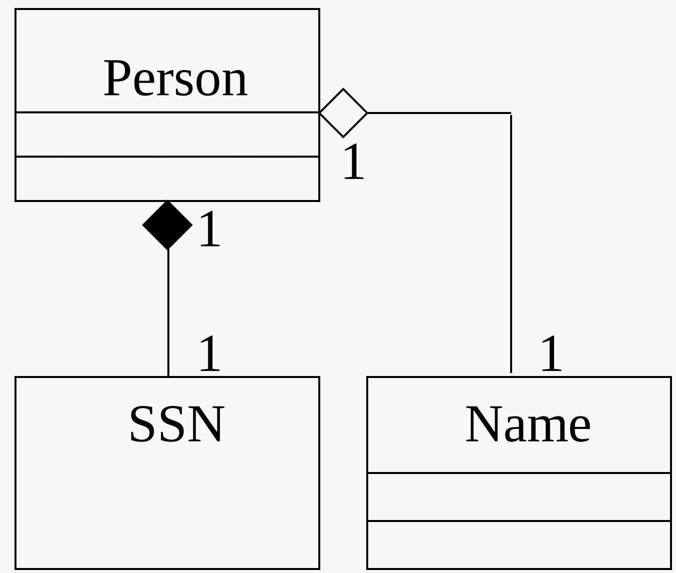
- a person has a name and an address
- a rectangle has a NW corner and a height and a width

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

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.

Simple Aggregations

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

```
public class Address {
 private String street;
 private String city;
 private String state;

}
```

```
public class Name {
 private String first;
 private String middle;
 private String last;

}
```

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.

A More Interesting Aggregation

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

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

public class Person {
  private Name    nom;       // sub-object
  private Address addr;      // sub-object
  private Person  spouse;    // association link
  private Gender  gen;       // simple data member

// . . .

}
```

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

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:

<u>Construction</u> 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.

<u>Destruction</u> 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.

Person Constructor

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

```
Person() {
   Spouse = null;
   Gen    = Gender.GENDERUNKNOWN;
}

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

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.

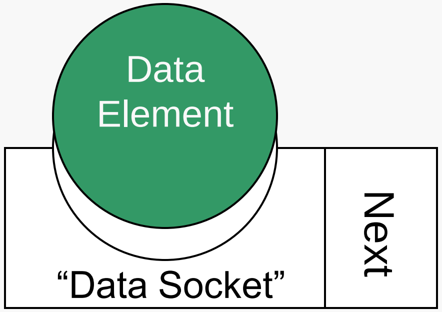
Obviously, this is consistent with the stated rules for aggregate construction and destruction.

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




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


// . . .

}
```

Containers

Container objects which hold a collection of objects/references of some type.

The use of containers require one to address possibly two relationships:

1. The container object itself.
2. The container and the objects it holds.

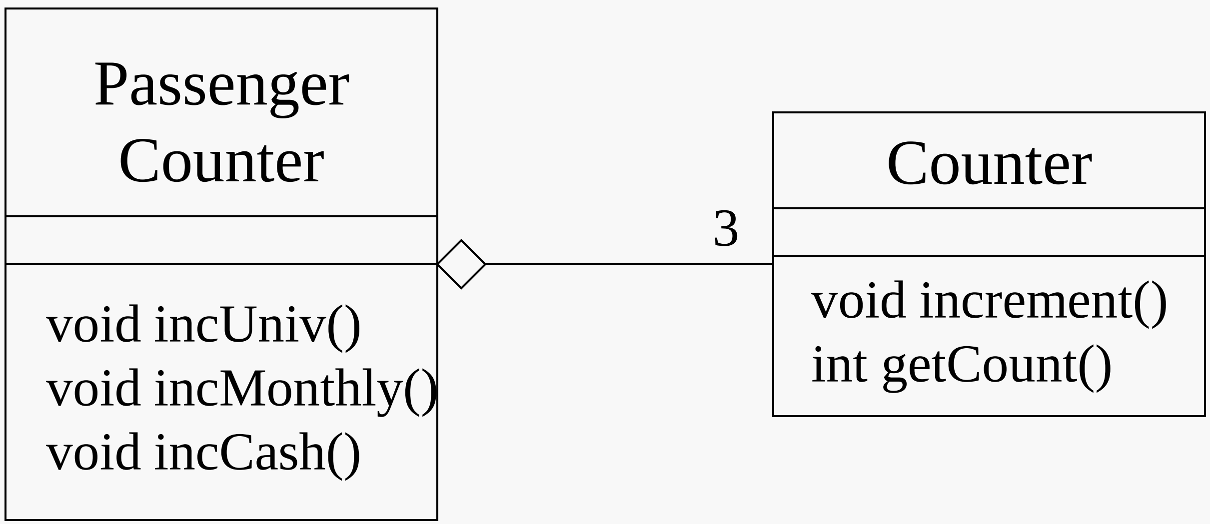
First: is the container object inside another object? Does the container object have a lifetime of its own?

Second: Are the contained objects instantiated (and destroyed) by the container?

Most type two relationships are association.

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:



Counter Class

We will employ a `Counter` class:

```
public class Counter
{
  private int cnt = 0;

  public Counter(Counter c)   {this.cnt = c.cnt;}
  public Counter(int iCnt)    { cnt = iCnt;}
  public void Increment()     { cnt++; }
  public int  getCount()      { return cnt; }

}
```

PassengerCounter Class

We will employ a PassengerCounter class:

```
public class PassengerCounter {
  private Counter UnivID;
  private Counter Monthly
  private Counter Cash;

 public PassengerCounter() { }

 public void incUnivID() { }
 public void incMonthly() { }
 public void incCash() { }
 public int  getUnivIDCount() { }
 public int  getMonthlyCount() { }
 public int  getCashCount() { }
 public void summarize(PrintWriter out) { }

}
```


PassengerCounter Implementation

Constructors:

```
public PassengerCounter() {
        UnivID   = new Counter();
        Monthly = new Counter();
        Cash    = new Counter();
}
```

Mutators:

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

`PassengerCounter` Implementation

Accessors:

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

Display function:

```
void summarize(PrintWriter out) {

   out.println("Payment summary:" );
   out.println("University ID  |" + getUnivIDCount());
   out.println("Monthly pass   |" + getMonthlyCount());
   out.println("Cash           |" + getCashCount());
}
```

PassengerCounter Driver

Driver to test the PassengerCounter class:

```
void driver() {
   PassengerCounter RiderStats = new PassengerCounter();
   Random rand = new Random( System.currentTimeMillis() );

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

      int payType = rand.nextInt(3);
      switch (payType) {
      case 0:   RiderStats.incUnivID();
                break;
      case 1:   RiderStats.incMonthly();
                break;
      case 2:   RiderStats.incCash();
                break;
      default:  break;
      };
   }
   RiderStats.Summarize(new PrintWriter( System.out ));
}
```

```
Payment summary:

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