JERUSALEM COLLEGE OF TECHNOLOGY
LEV ACADEMIC CENTER

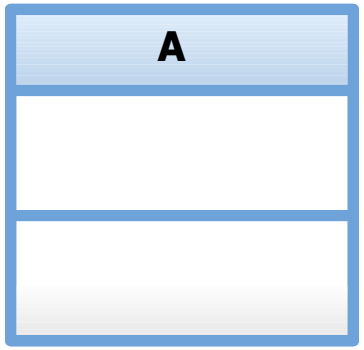
Object Oriented Programming and Design for Engineering
Course No.: 157109

Lab 2: From UML to code

Today's agenda

- Translating a class diagram to Java code.
- UML is not restricted to a single language. A diagram may have different translations in different languages.
- Some relationships may not have a direct translation. Keeping the contract is the programmer's responsibility.

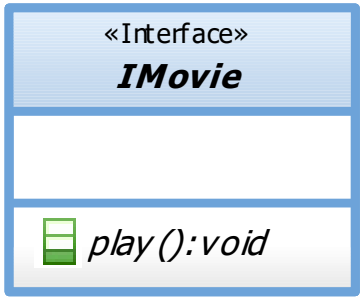
Class



A.java

```
class A {
}
```

Interface



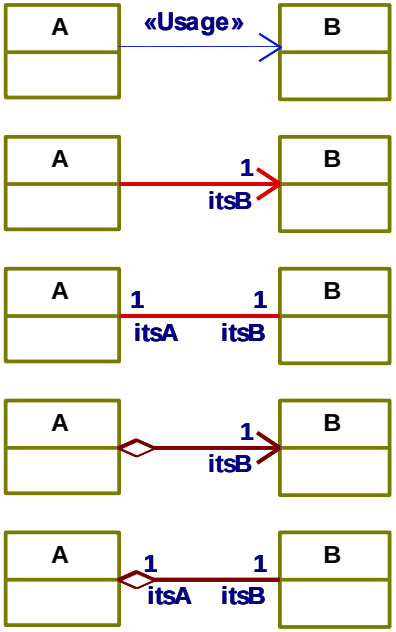
IMovie.java

```
interface IMovie {
    void play();
}
```

Some languages do not have an explicit representation for interfaces. In C++ for example, the equivalent is an abstract class where all the functions are pure virtual.

JERUSALEM COLLEGE OF TECHNOLOGY
LEV ACADEMIC CENTER

Types of connections (1)



Dependency

Directed Association

Bidirectional Association

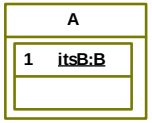
Directed Aggregation

Bidirectional Aggregation

Types of connections (2)

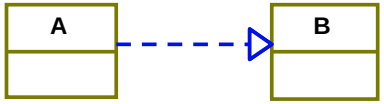
Directed Composition

Bidirectional Composition

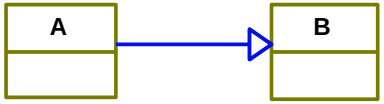


Composite

Link (direction = the direction of underlying association))



Realization



Generalization

Dependency


```
import B;
class A {
   f(B b){
      …
   }
}
```


```
import B;
class A {
   public A(B b){
      …
   }
}
```


```
import B;
class A {
   public B f(…){
      B b = new b();
      …
   }
}
```

Directed Association:
with End name and Multiplicity displayed


```
class A {
    private B itsB;
    ….
    public void setItsB(B itsB) {
        this.itsB = itsB;
    }

    public B getItsB() {
        return itsB;
    }
}
```

Directed Association: What else can be defined?

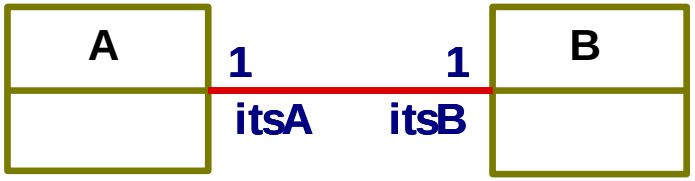
A UML diagram doesn't define all the properties which define the link:

- Visibility: private? public?
- Setters? Getters?

Each code generator / programmer can choose an implementation according to a different default setting

For this course we will assume that the underlying code includes setters/getters, keeping fields encapsulated.

Association (Bidirectional)




```
public class A{
    private B itsB;
}
```

```
public class B{
    private A itsA;
}
```

- A bidirectional relationship defines not only structure, but also behavior!
  - Each time an instance of A is connected to an instance of B, the connection applies in the opposite direction as well
  - Each time an instance of A is disconnected from an instance of B, the disconnection applies in the opposite direction as well

Connecting objects with bidirectional association

Given the following class diagram

We want to connect the object *a* of type *A* to the object *b* of type *B*, However, these object are already connected to different objects.

We first need to disconnect the existing connections and connect the objects bidirectionally. We will see in the next slides how this is done.

Connecting objects with bidirectional association

1

a.setItsB(b)

In Class A

```
public void setItsB(B b) {
    if(b != null) {
        b._setItsA(this);
    }
    _setItsB(b);
}
```

Connecting objects with bidirectional association

2

b._setItsA(a)

In Class B

```
public void _setItsA(A a) {
    if(itsA != null) {
        itsA.__setItsB(null);
    }
    __setItsA(a);
}
```

Connecting objects with bidirectional association

3

a1.__setItsB(null)

b.itsA.itsB=null

Connecting objects with bidirectional association

4

b.__setItsA(a)

b.itsA=a

Connecting objects with bidirectional association

5

a._setItsB(b)

In Class A

```
public void _setItsB(B b) {
    if(itsB != null) {
        itsB.__setItsA(null);
    }
    __setItsB(b);
}
```

Connecting objects with bidirectional association

6

b1.__setItsA(null)

a.itsB.itsA=null

Connecting objects with bidirectional association

7

a.__setItsB(b)

a.itsB=b

Connecting objects with bidirectional association

8

Done

Connecting objects with bidirectional association

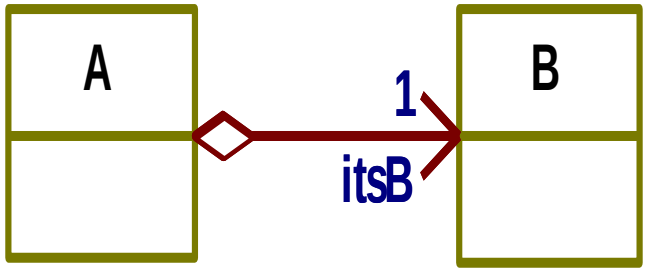
In class *A*

```
public void setItsB(B b) {
    if(b != null) {
        b._setItsA(this);
    }
    _setItsB(b);
}
public void _setItsB(B b) {
    if(itsB != null) {
        itsB.__setItsA(null);
    }
    __setItsB(b);
}
public void __setItsB(B b) {
    itsB = b;
}
```

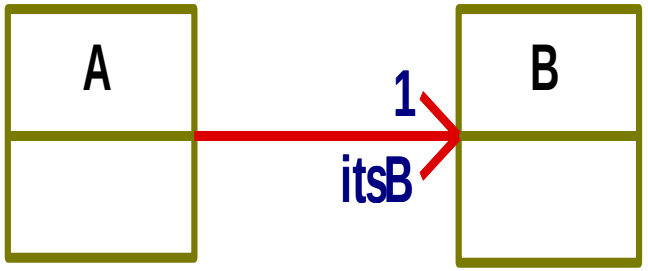
In class *B*

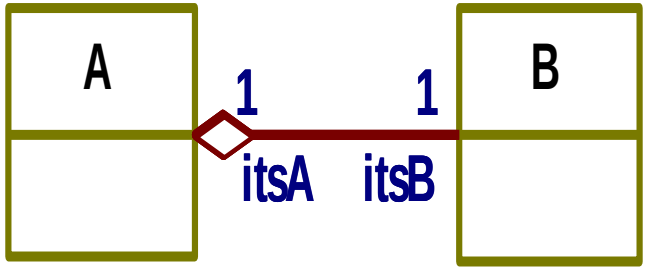
```
public void setItsA(A a) {
    if(a != null) {
        a._setItsB(this);
    }
    _setItsA(a);
}
public void _setItsA(A a) {
    if(itsA != null) {
        itsA.__setItsB(null);
    }
    __setItsA(a);
}
public void __setItsA(A a) {
    itsA = a;
}
```

Regarding the code generated in Java: Association=Aggregation

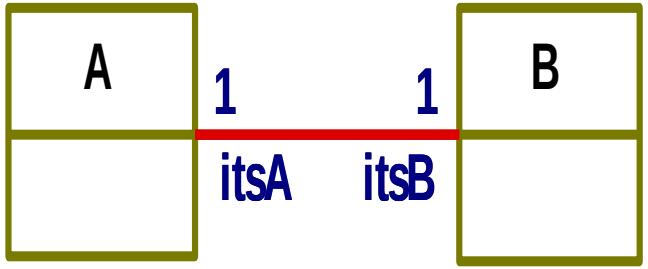


Same code generated as

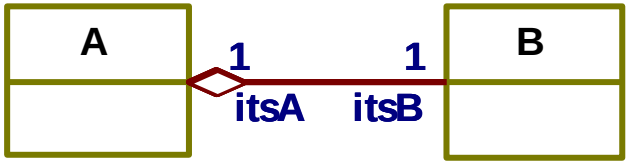


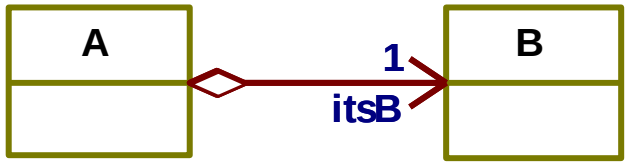


Same code generated as

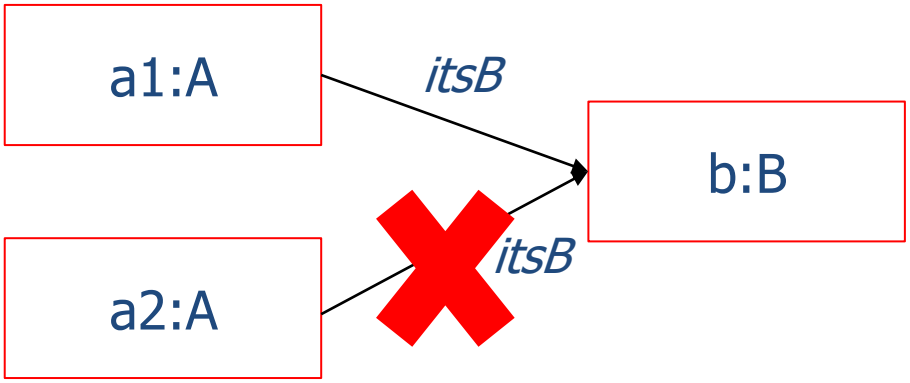


Aggregation





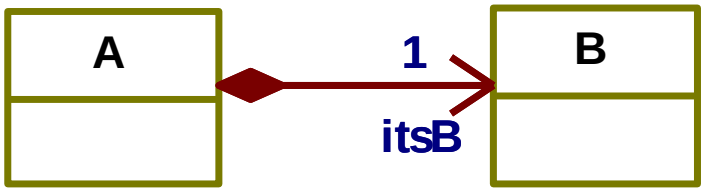
- Aggregation defines not only structure, but also behavior!
- In runtime, for every object, *b*, of type *B*, only one *A* object may have an *itsB* pointer to *b*. This implies ownership.



Important note: Generated code ≠ semantics

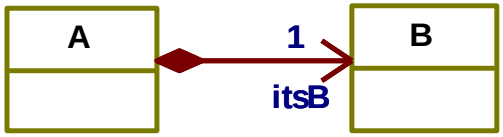
- Each programming language has unique properties
- The default generated Java code can't express concepts such as ownership and dependence
- The responsibility for matching between the diagram and implementation falls on the programmer

Directed Composition



- Composition defines not only structure, but also behavior!
- In runtime, every time we initialize an object, *a*, of type *A*, the object *a.itsB* is also initialized.
- When *a* is deleted, *a.itsB* is also deleted.

Implementing Composition in Java



A.java

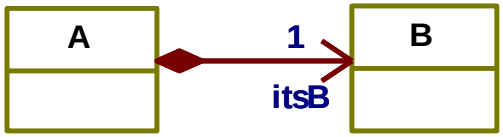
```
class A {
    private B itsB=new B();



}
```

For bidirectional:

B.java

```
class B {
    private  A itsA;
}
```

Implementing Composition in C++



A.h

```
#include "B.h"
class A {
    private B itsB;



}
```

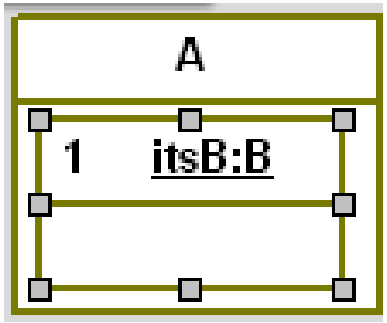
For bidirectional:

B.h

```
class A;
class B {
    private  A *itsA;
}
```

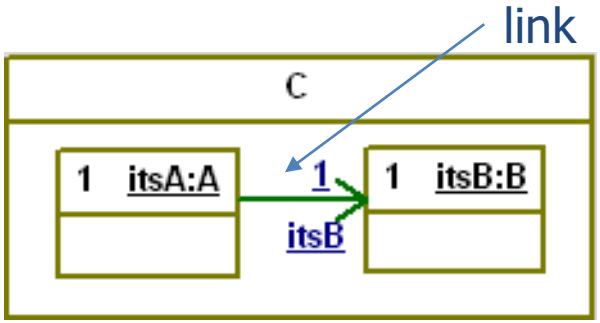
In C++, B is defined as an instance and not as a reference. The concept of dependence can be expressed. When A is deleted, B will stop existing immediately.

Directional/Bidirectional Composite

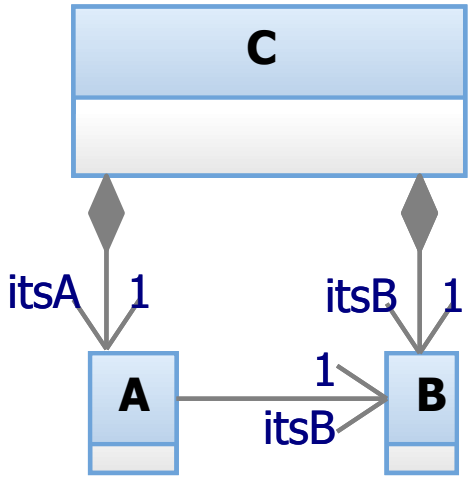


- The code and semantics are the same as Directional/Bidirectional Composition
- The diagram doesn't include the information whether it's a directional/bidirectional composition

Composite with links ≠ Composition

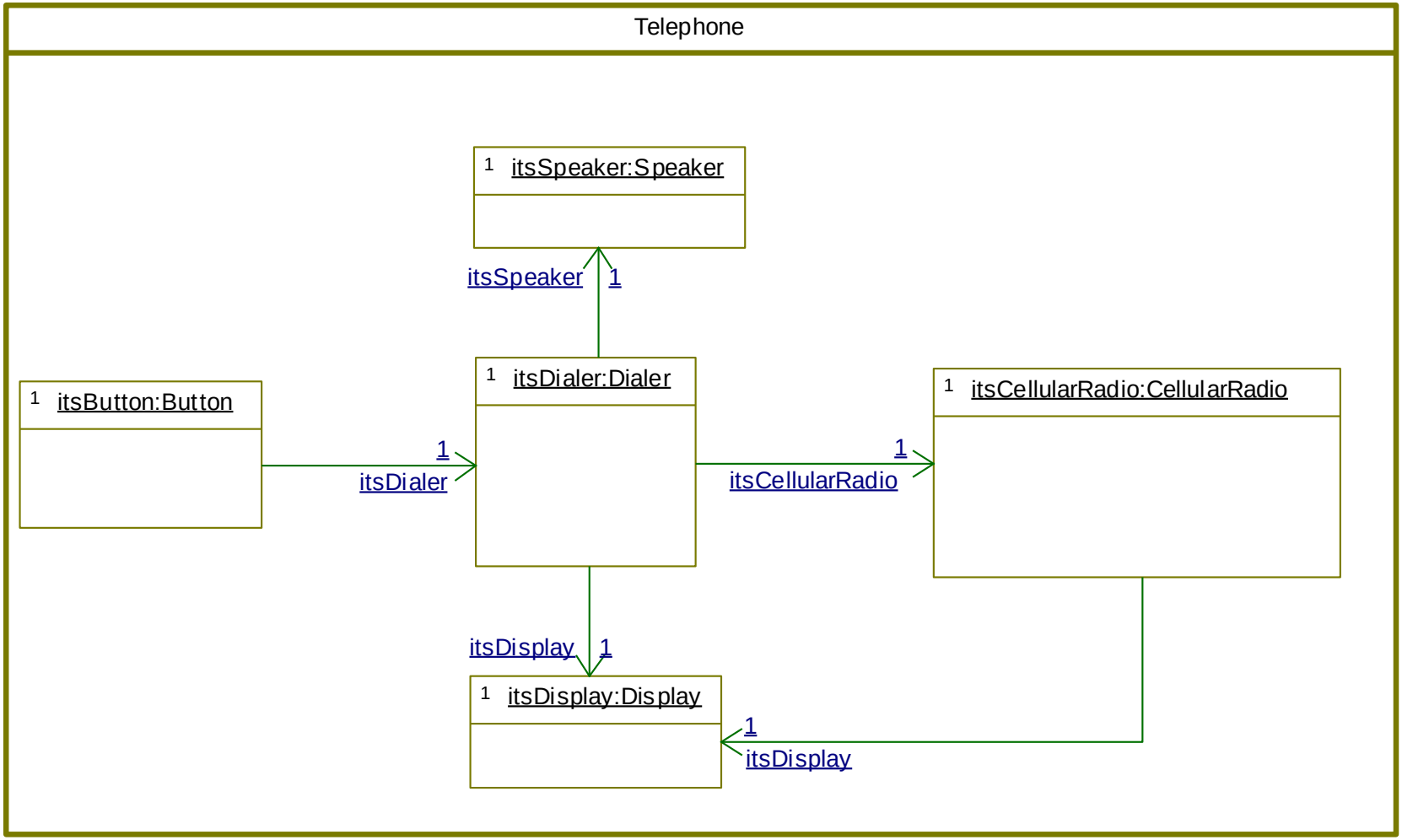



```
class C {
  private A itsA;
  private B itsB;
  public C(){
      itsA = new A();
      itsB = new B();
      initRelations();
  }
  private void initRelations(){
      itsA.setItsB(itsB);
  }
}
```




```
class C {
  private A itsA;
  private B itsB;
  public C(){
      itsA = new A();
      itsB = new B();
  }
}
```

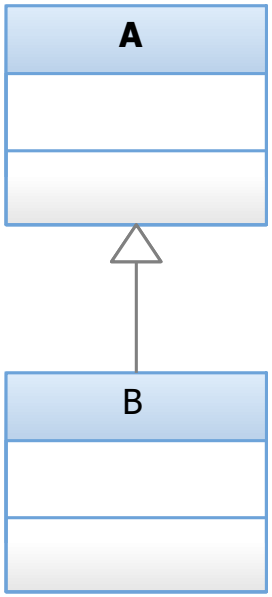
Composite with links



Composite with links: initRelations()

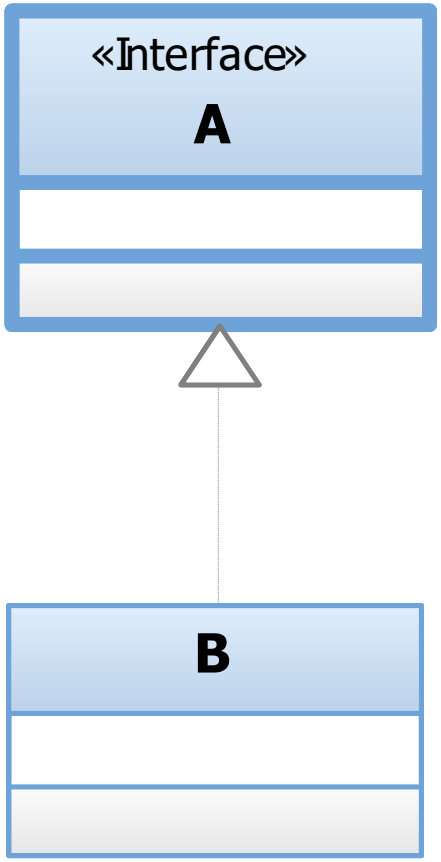
```
void initRelations() {
    itsButton.setItsDialer(itsDialer);
    itsDialer.setItsCellularRadio(itsCellularRadio);
    itsDialer.setItsDisplay(itsDisplay);
    itsDialer.setItsSpeaker(itsSpeaker);
    itsCellularRadio.setItsDisplay(itsDisplay);
}
```

Generalization



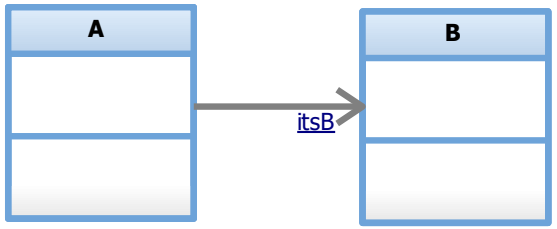

```
public class B extends A{
    public B(){
        super();
    }
}
```

Realization



Public class B implements A

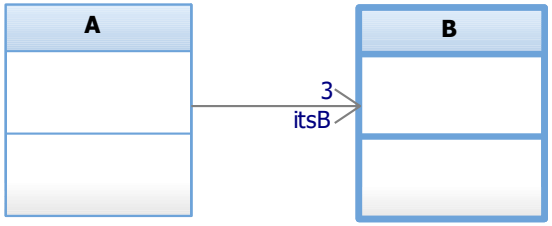
Multiplicity: default=1



Default=1

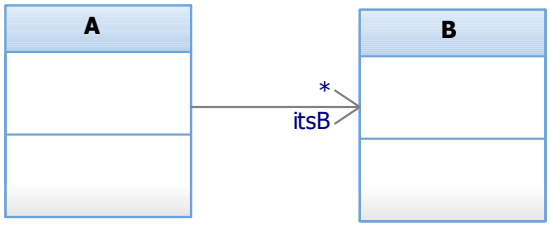
```
B itsB;
...
B getItsB(){
  return itsB;
}
void setItsB(B b){
  itsB = b;
}
```

Multiplicity: 3



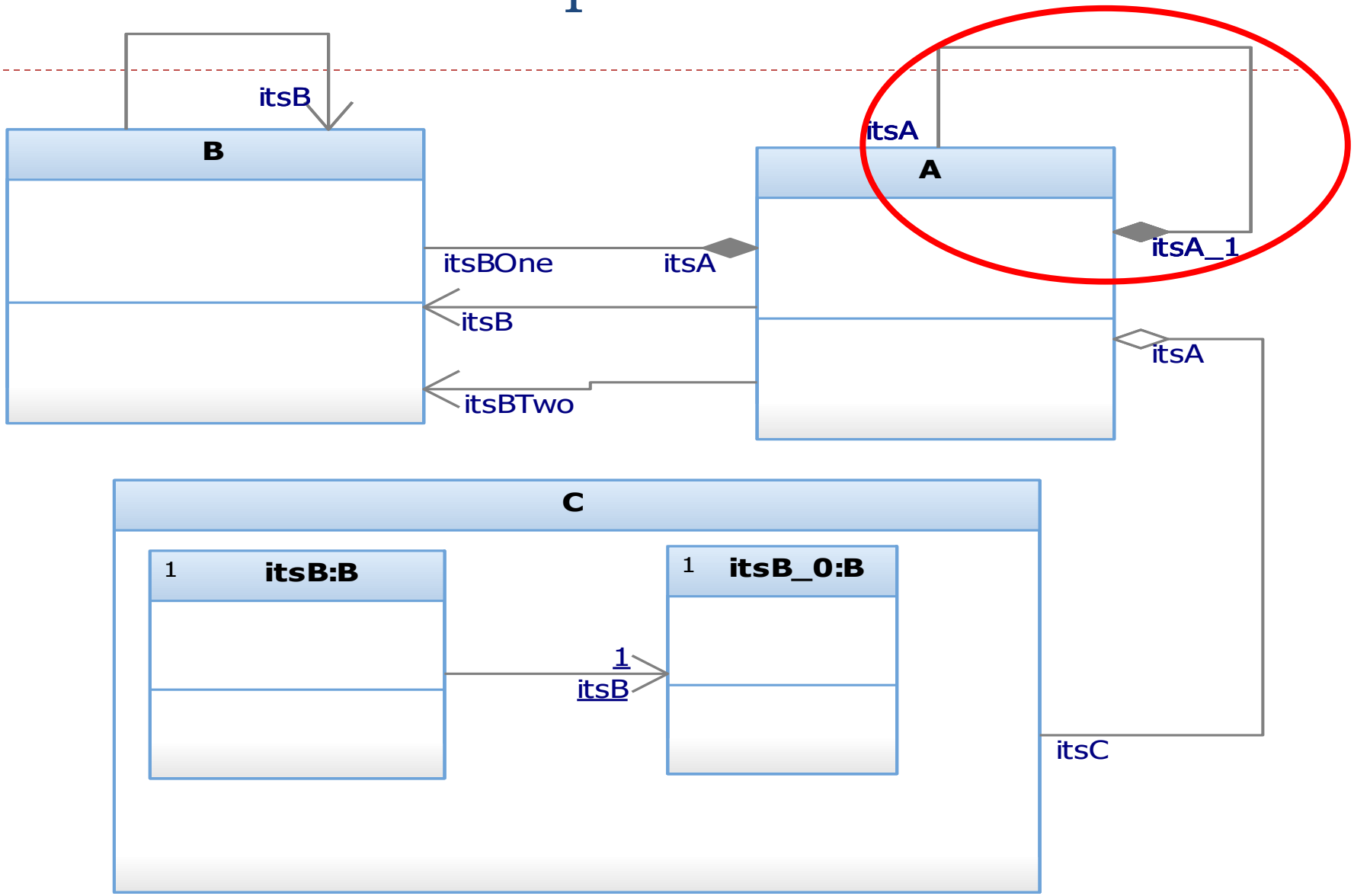

```
public class A{
    B[]  itsB;
    public A(){
        itsB = new B[3];
    }
    public void addB(B b){
        int pos;
        for (pos = 0; pos < 3 && itsB[pos] !=null ; pos++) ;
        if (pos != 3) itsB[pos] = b;
    }
    public void removeB(B b) {
        for (int pos = 0; pos < 3; pos++)
            if (itsB[pos] == b)
                for(int i = pos; i<2; i++)
                    itsB[i] = itsB[i+1];
                itsB[2] = null
                break;
    }
    public B getB (int i){
        return itsB[i];
    }
    public void clearB(){
        for (int pos = 0; pos < 3; pos++)
            itsB[pos] = null;
    }
}
```

Multiplicity: *




```
public class A{
    List<B> itsB;
    public A(){
        itsB=new ArrayList<B>();
    }
    public void addB(B b){
        itsB.add(b);
    }
    public void removeB(B b){
        itsB.remove(b);
    }
    public B getB (int i){
        return itsB.get(i);
    }
    public void clearB(){
        itsB.clear();
    }
}
```

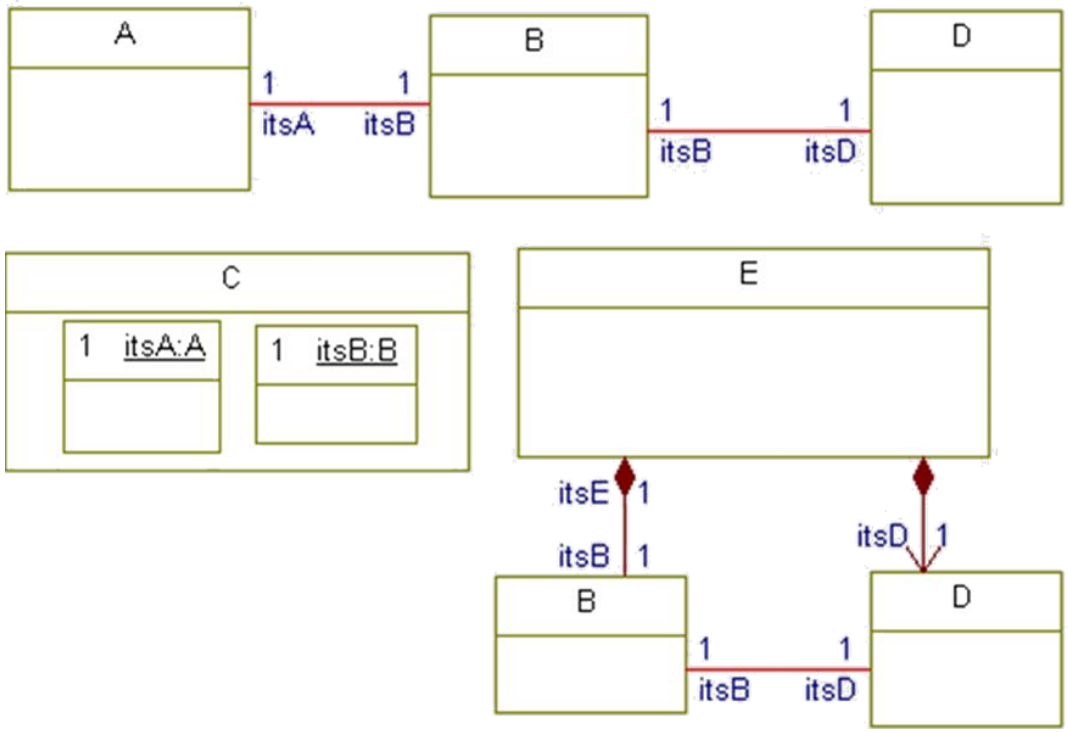
Is there a problem here?



How many instances from each class will be initialized?

```
A a = new A();
B b = new B();
C c = new C();
E e = new E();
```

class	#instances
A	2
B	3
C	1
D	1
E	1

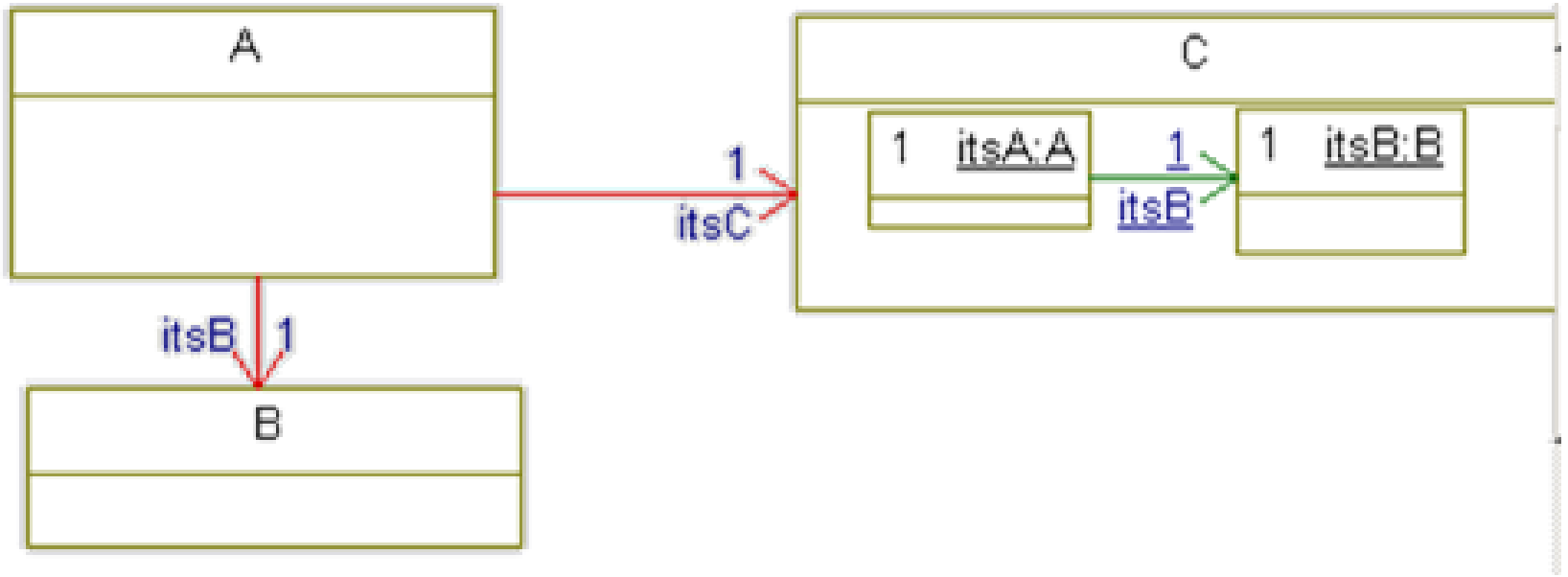


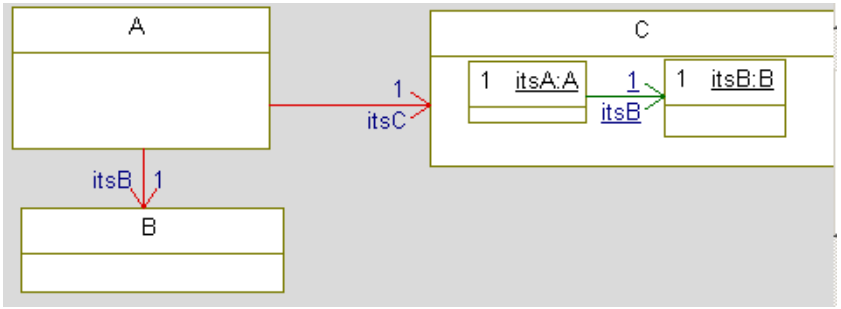
Describe the instances and relations

Assume that the Composite is bidirectional. Given the following code:

```
A a = new A();
a.setItsC(new C());
```

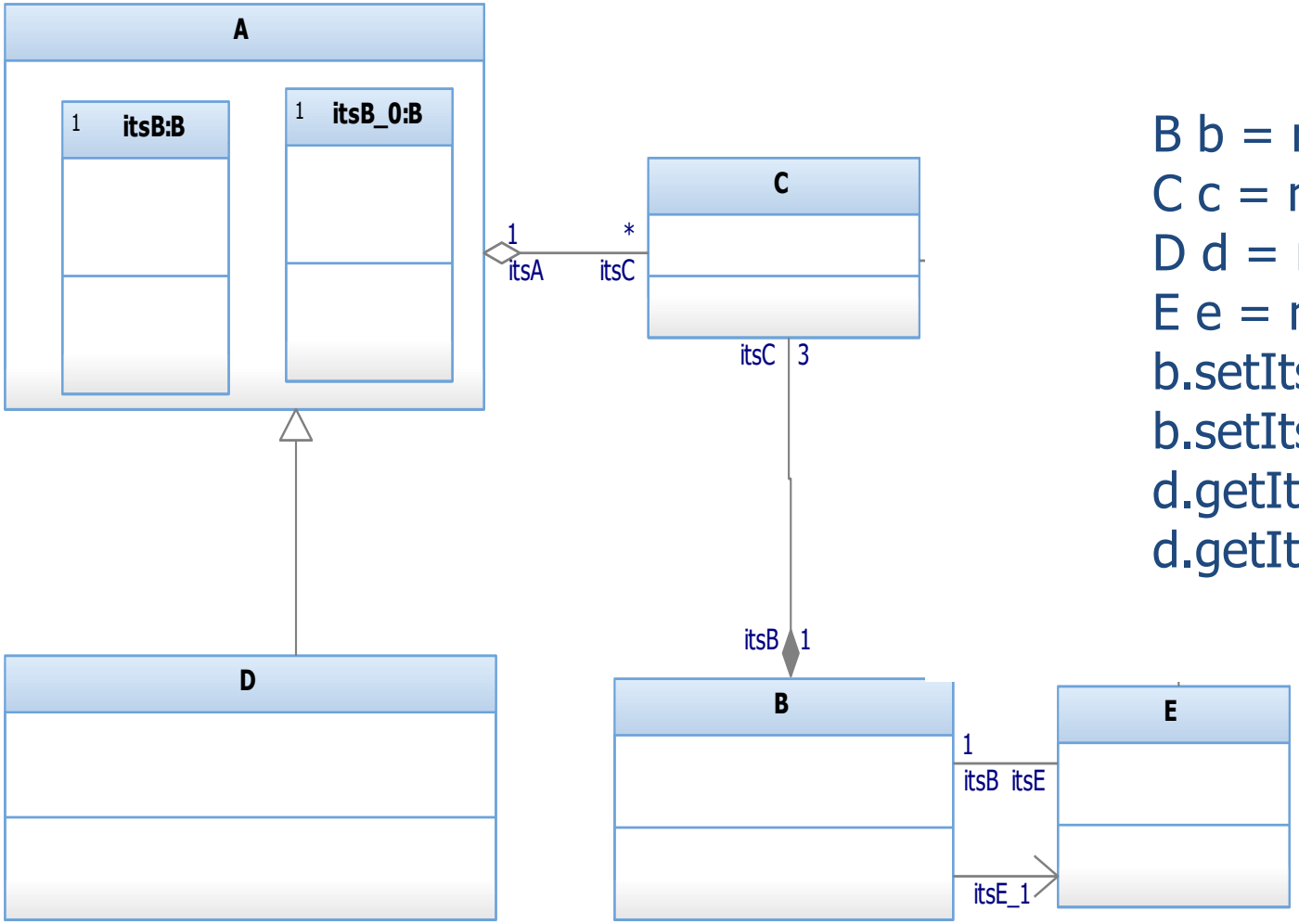
For every instance of A, provide its name and its relations





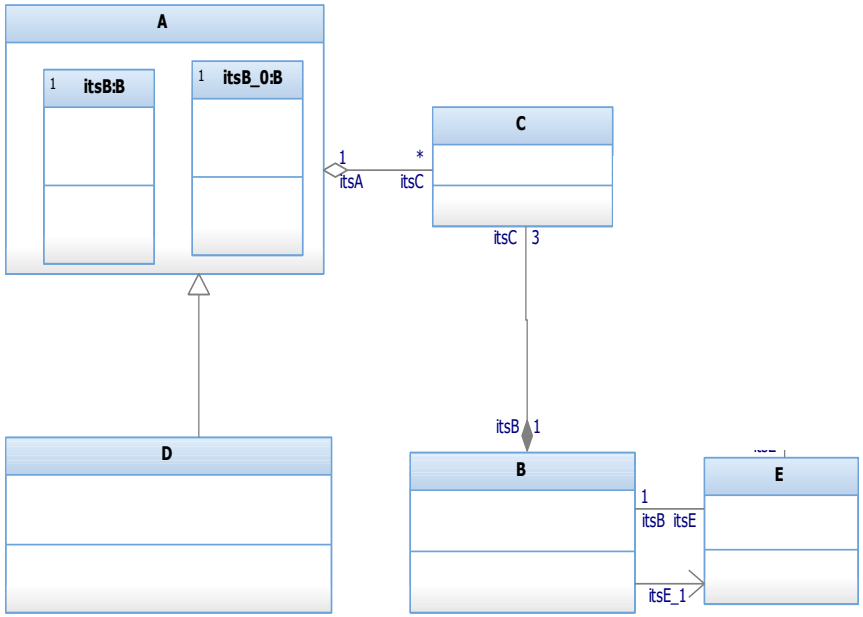
Instance	Relations
A[0]	A[0].itsC = C[0] A[0].itsB = null A[0].itsC_1 = null
C[0].itsA	C[0].itsA.itsC = null C[0].itsA.itsB = C[0].itsB C[0].itsA.itsC_1 = C[0]

Describe the instances and their relationships




```
B b = new B();
C c = new C();
D d = new D();
E e = new E();
b.setItsE(e);
b.setItsE_1(e);
d.getItsB.setItsE(e);
d.getItsB.setItsE_1(e);
```

Describe the instances and their relationships




```
B b = new B();
C c = new C();
D d = new D();
E e = new E();
b.setItsE(e);
c.setItsE_1(e);
d.getItsB().setItsE(e);
e.getItsB().setItsE_1(e);
```

Tools for auto code generation

- GenMyModel (Cloud service)
- Rhapsody (Application)
- Intellij (IDE)