Software Engineering III

Managing/Implementing one-to-one & one-to-many Associations

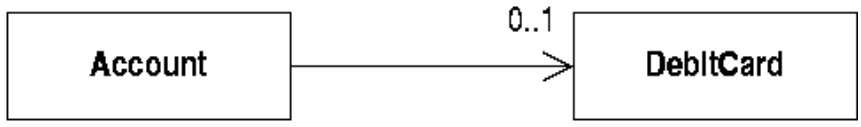
Ref: Priestly Ch.13

Issues with Associations

- Simple associations can be implemented with reference data members
- References support only one direction of navigation
  - it is often worth trying to restrict direction of navigation to make implementation easier
- Give one class the responsibility for maintaining an association
  - References should not be manipulated explicitly by other classes
- Other classes gain access to the reference and hence the linked objects through an interface

One to one 0..1 Optional Associations

- A major distinction between associations is their multiplicity
- Reference variables can hold
  - a reference to another object
  - or the null reference
- So the 'default' multiplicity is '0..1'



Defining the 0..1 Association

- The association is defined by a field in the 'Account' class holding a reference to an instance of the 'DebitCard' class

```
public class Account
{
     private DebitCard theCard ;
     …
}
```

5

Maintaining the 0..1 Association

The 'Account' class is responsible for maintaining this association since it holds the attribute in question.

Add methods as required, e.g.:

```
public class Account {
  private DebitCard theCard ;

  public DebitCard getCard() {
        return theCard ;
  }
  public void setCard(DebitCard card) {
        theCard = card
  }
  public void removeCard() {
        theCard = null
  }
}
```

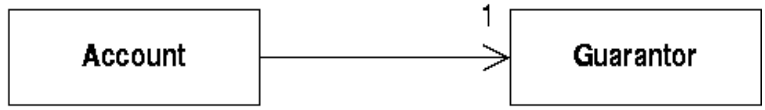
6

Immutable 0..1 Association

```
public class Account
{
  private DebitCard theCard;

  public DebitCard getCard() {
    return theCard ; }

  public void setCard(DebitCard card) {
    if (theCard != null) {
    // raise ImmutableAssociationError
    }
    theCard = card;
  }
}
```

One-to-one 1..1 Mandatory Association

- Suppose every account has exactly one *guarantor*
- ***Guarantor is mandatory***
- We must *not allow null references to Guarantor* to be stored in the 'Account' class
  - this must be checked explicitly in code



One-to-one 1..1 Association

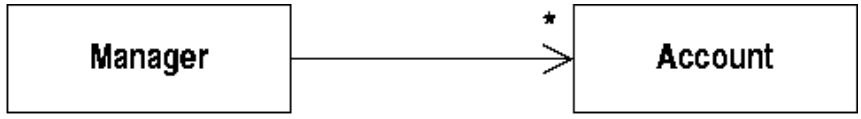
Check that a non-null link value is provided in the constructor for Account

```
public class Account {
  private Guarantor theGuarantor ;

  public Account(Guarantor g) {
    if (g == null) {
    // throw NullLinkError
    }
    theGuarantor = g;
  }
  public Guarantor getGuarantor() {
    return theGuarantor;
  }
}
```

Implementing Associations
One to Many

One to Many 1..n Associations

- The model shows that it is necessary to identify each account that a manager is responsible for.
  - A manager object can be linked to many account objects.
- It is not required to identify the particular manager for a specific account.
- Manager object will require multiple pointer/references (an array of references) to each Account object.



One to Many 1..n Associations

The Java Vector class solution provides a host of methods for adding, deleting and traversing the account objects. The methods in Manager need only deal with its business processing.

However the addAccount method needs to check for duplicate Accounts

Note the use of the *Vector* collection class

```
public class Manager{
  private Vector theAccounts;

  public void addAccount(Account acc){
    theAccounts.addElement(acc) ;
  }
  public void removeAccount(Account acc) {
    theAccounts.removeElement(acc)
  }
}
```

Vector Fields

- protected int capacityIncrement
  - The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity.
- protected int elementCount
  - The number of valid components in this Vector object.
- protected Object[] elementData
  - The array buffer into which the components of the vector are stored.

Vector Constructor Summary

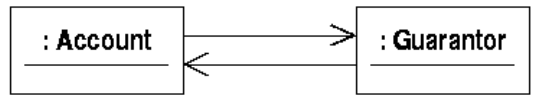
- Vector()
  - Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.
- Vector(Collection c)
  - Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.
- Vector(int initialCapacity)
  - Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
- Vector(int initialCapacity, int capacityIncrement)
  - Constructs an empty vector with the specified initial capacity and capacity increment.

Vector Method Examples

- void add(int index, Object element)
  - Inserts the specified element at the specified position in this Vector.
- boolean add(Object o)
  - Appends the specified element to the end of this Vector.
- boolean addAll(Collection c)
  - Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
- void addElement(Object obj)
  - Adds the specified component to the end of this vector, increasing its size by one.
- int capacity()
  - Returns the current capacity of this vector.

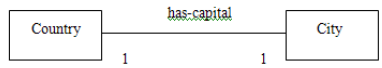
Bidirectional Links

- Some associations need to be navigated in both directions
  - because references are unidirectional, it will take two references to implement a link
  - the association can be implemented by including a suitable reference field in each of the associated classes



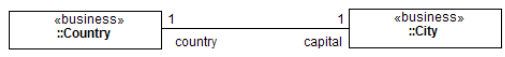
One-to-One Association traversed in both directions, i.e. bi-directional.

Conceptual diagram.



The conceptual diagram shows the associations between the classes without any consideration of how the association might be implemented.

Specification diagram.



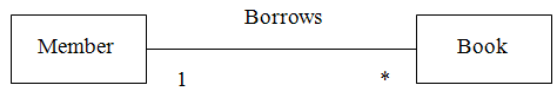
One-to-One Association traversed in both directions, i.e. bi-directional.

```
public class  Country
{
        private String name ;
        private City capital ;
        // etc.
        void setCapital(City  aCapital)
        {
                capital = aCapital ;
        }
}
```

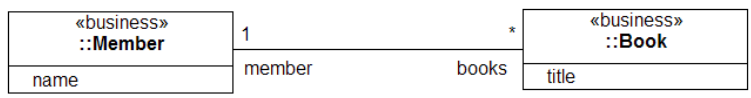
```
public class  City
{
        private String name ;
        private Country country ;
        // etc.
        public void setCountry(Country aCountry)
        {
                country = aCountry ;
        }
}
```


Bi-directional implementation of one-to-many association

Conceptual Diagram



Specification Diagram



Bi-directional implementation of one-to-many association

- For a bi-directional association - as well as a *Member* object containing a *Set* of *Book*s, a *Book* object will contain a reference to the *Member* object that contains it within its *Set*.

```
public class Book
{
   …
   private Member member ;

   public Book(String aTitle) {
      …
      …
      member = null ;
   }
   …
   …
}
```


Referential Integrity

- With **bi-directional associations**, we need to consider how an operation of one class may effect the other associated class.
- E.g.
  - if we had a ***member*** object with a set of ***book*** objects – each *book* object will have a reference to the *member* object. If we removed a *book* from the *Set* of *books* in the *member* object's *Set*, then we should also set the reference to the *member* object in the *book* object to null to maintain **referential integrity**.

Referential Integrity Example – Member Class

```
public boolean borrowBook(Book aBook)
{
    if (aBook == null)
        return false ;

    if (books.contains(aBook) )
        return false ;

    if (aBook.checkOut() )
    {
        books.add(aBook) ;
        aBook.setMember(this) ;
        return true ;
    }
    else
        return false ;
}
```

When borrowing a book, we will create the **bi-directional** *link* between the *Member* object and the *Book* object

As well as adding the *book* to its *Set*, it sets the *member* attribute of the *book* object to itself.

Referential Integrity Example – Member Class

```
public boolean returnBook(Book aBook)
{
    if (aBook == null)
        return false ;

    if (! books.contains(aBook) )
        return false ;

    if (aBook.checkIn() )
    {
        books.remove(aBook) ;
        aBook.removeMember() ;
        return true ;
    }
    else
        return false ;
}
```

When returning a book, we will remove the *link* between the *Member* object and the *Book* object (**in both directions**)

As well as removing the *book* from its *Set*, it removes the *member* reference of the *book* object.

Referential Integrity Example – Book Class

```
public boolean setMember(Member aMember)
{
    if (aMember == null)
        return false ;

    if (member != null)
        return false ;

    member = aMember ;
    member.borrowBook(this) ;

    return true ;
}
```

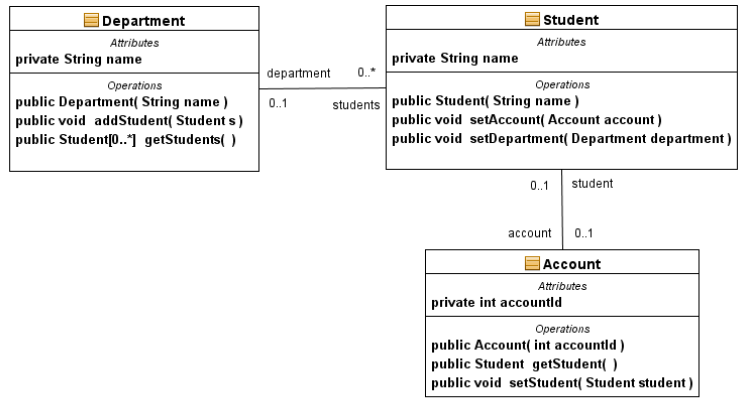
As well as setting the *member* reference, it adds the *book* to the member object's *Set* of *books*

Referential Integrity Example – Book Class

```
public boolean removeMember()
{
    if (member != null)
    {
        member.returnBook(this);
        member = null ;
        return true ;
    }
    else
        return false ;
}
```

As well as removing the *member* reference it removes the *book* reference from the *member* object's list

Example

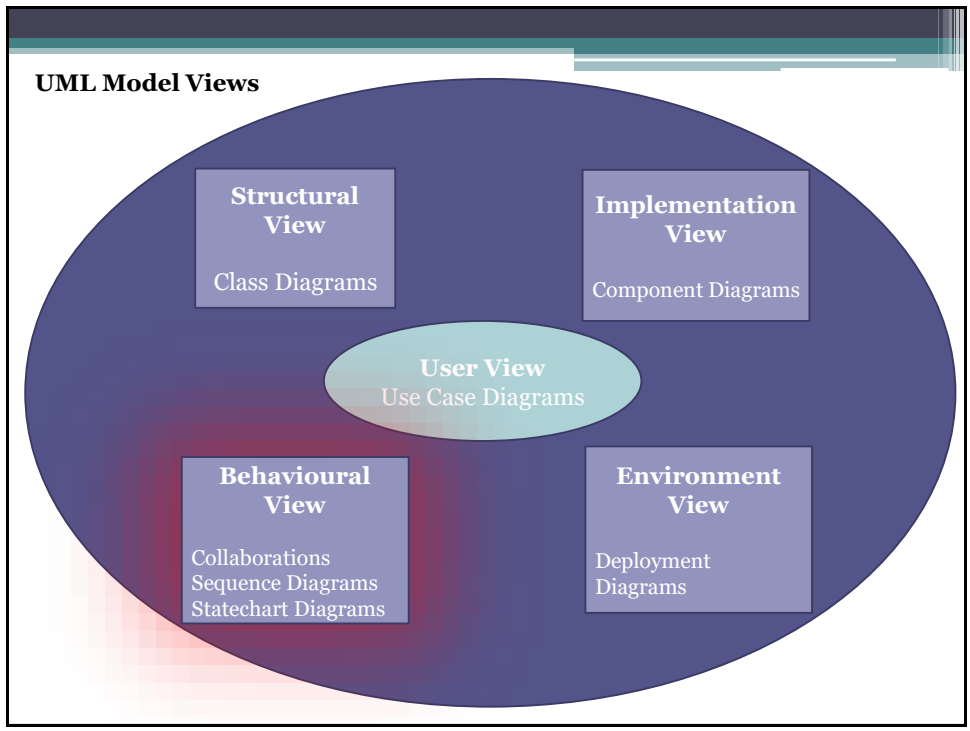



```java
public class Department {
    private String name;
    private List<Student> students = new ArrayList<Student>();

    public Department(String name) {
        this.name = name;
    }
}

public class Student {
    private String name;
    private Department department = null;
    private Account account = null;

    public Student(String name) {
        this.name = name;
    }
}

public class Account {
    private int accountId;
    private Student student = null;

    public Account(int accountId) {
        this.accountId = accountId;
    }
}
```

```
//Student Class
public void setAccount(Account account) {
  if (account == null)
      return;
  if (this.account == null) {
      this.account = account;
      this.account.setStudent(this);
  }
}

//Account Class
public void setStudent(Student student) {
  if (student == null)
      return;
  if (this.student == null) {
      this.student = student;
      this.student.setAccount(this);
  }
}
```

```
//Department Class
public void addStudent(Student s) {
  if (s == null)
      return;
  if ( ! students.contains(s)) {
      students.add(s);
      s.setDepartment(this);
  }
}

//Student Class
public void setDepartment(Department department) {
  if (department == null)
      return;
  if (this.department == null) {
     this.department = department;
     this.department.addStudent(this);
  }
}
```

Sequence Diagrams Revisited



Recall from previous class - Object Links

```
public class SomeClient {

    SomeClient (Circle circleA,
                Circle circleB)
    {
        circleA.setRadius(10);
        circleB.setRadius(12);
    }
}
```

```
public class Circle {

  private int radius;

  public int getRadius() {
        return radius;
  }

  public void setRadius(int new_radius)
  {
        radius = new_radius;
  }

}
```

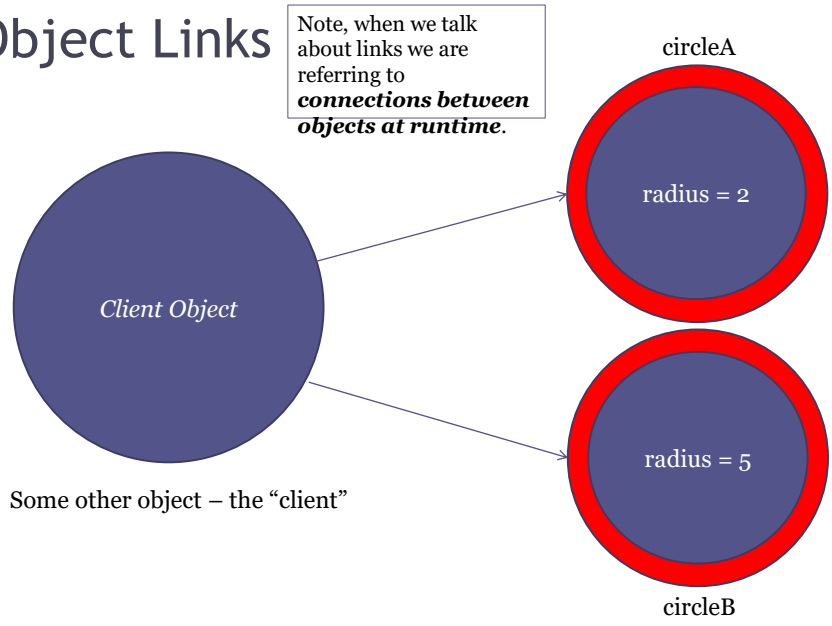
Consider an instance of the above class - it will contain two *references* to two Circle objects. This means that at runtime it will be *linked* to those objects

Vice versa, the two instances of this class (circleA and circleB) are *linked* to the client object

The actual *link* is provided by the references (*circleA* and *circleB*) within the client object
– interestingly, this means that only the client object "knows" about the link. In this sense, the link is *uni-directional.*

Object Links

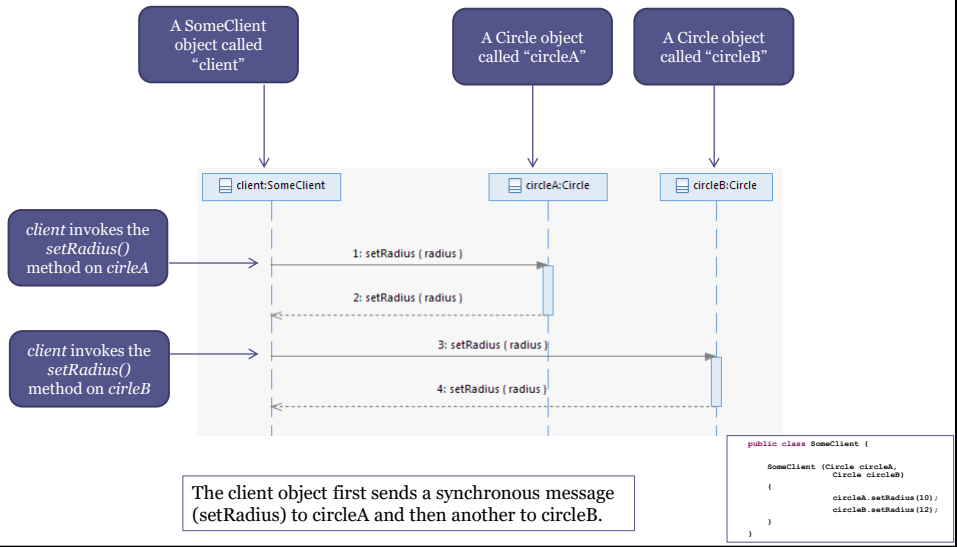
Note, when we talk about links we are referring to ***connections between objects at runtime***.



Some other object – the "client"

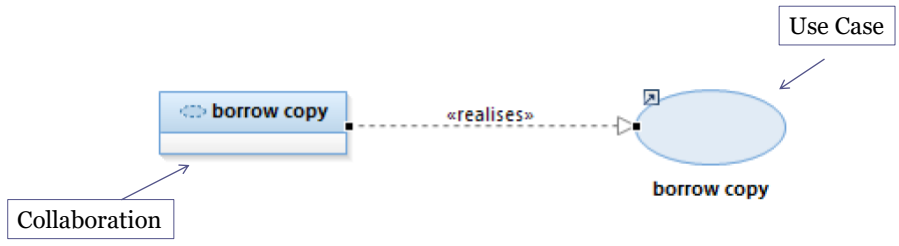
Sequence Diagram

Depicting how the code executes



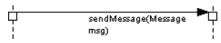
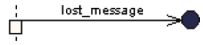
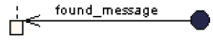
The client object first sends a synchronous message (setRadius) to circleA and then another to circleB.

Collaborations



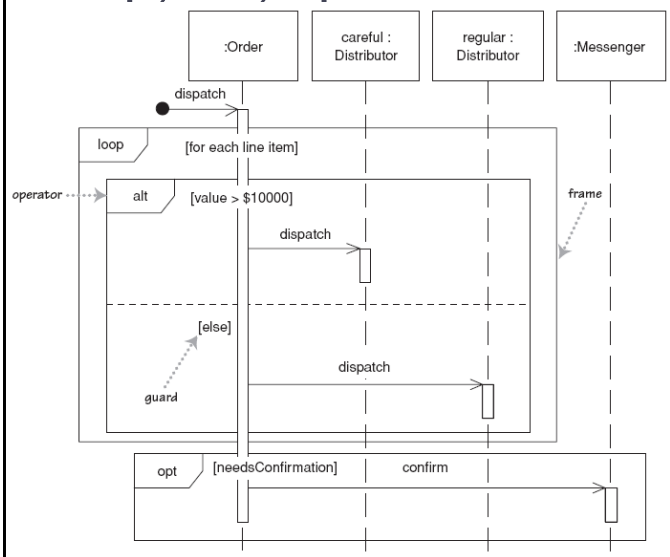
- Model a set of classes that are capable of interacting for a set of purposes.
  - Collaboration / Communication Diagrams
  - **Sequence Diagrams**

35

Entity	
Synchronous 	A message with a solid arrowhead at the end. If the message is a return message it appears as a dashed line rather than solid.
Asynchronous 	A message with a line arrowhead at the end. If the message is a return message it appears as a dashed line rather than solid.
Lost 	A lost message is one that gets sent to an object that is not modeled in the diagram. The destination for this message is a black dot.
Found 	A found message is one that arrives from an unknown sender, or from an object that is not modeled in the diagram. The unknown part is modeled as a black dot.
Self Message	A self message is usually a recursive call, or a call to another method belonging to the same object.

36

loop, alt, opt



alt models if /else / switch alternatives

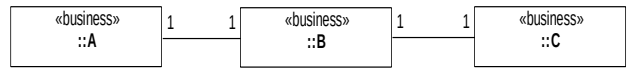
opt models if conditions

Dependencies and Associations

Principle of Least Knowledge

Dependencies & Associations

- It is useful to think of the associations between classes in a class diagram as ***paths of communication*** between objects of the classes.
- In the diagram below, objects of class A may talk to (i.e. send messages to) objects of class B and objects of class B may talk to objects of class A.
- Likewise, objects of classes B and C may also communicate. As there is no direct line of communication between A and C, objects of these classes ***may not communicate directly***. If they do communicate, then there is a bug in the model.



Law of Demeter / Principle of Least Knowledge

- In response to a message M, an object O should send messages only to the following objects:

1. O itself
2. Objects which are sent as arguments to the message M
3. Objects which O creates as part of its reaction to M
4. Objects which are directly accessible from O, that is, using values of attributes of O.

In principle:
- *Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.*
- *Each unit should only talk to its friends; don't talk to strangers.*
- *Only talk to your immediate friends.*

Law of Demeter 1

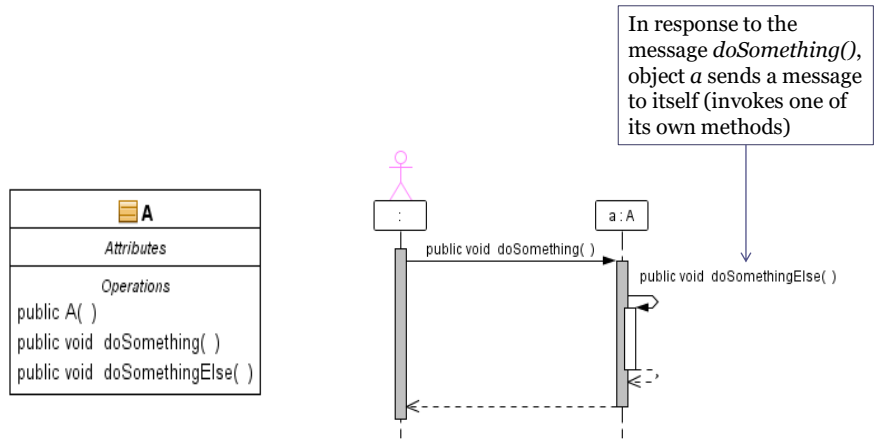
In response to a message M, an object O should send messages only to O itself

In response to the message, the object sends a message to itself, i.e. it calls one of its own methods.

```
public class A {

   public A() {
   }

   public void doSomething() {
      System.out.println("This is method doSomething() of class " +  this.getClass().getName());
      this.doSomethingElse();                    // Send message to self
   }

   public void doSomethingElse() {
      System.out.println("This is method DoSomethingElse() of class "  + this.getClass().getName());
   }

   public static void main(String[] args) {
      A a = new A();                             // a is object of class A
      a.doSomething();                                  // message sent to a
   }
}
```

Law of Demeter 1 - in UML notation

In response to a message M, an object O should send messages only to O itself



Next - Consider the following Class

```
public class B {

   public B() {
   }

   public void doSomethingElse() {
      System.out.println("This is method doSomethingElse() of
   class “                                          +
   this.getClass().getName());
   }
}
```

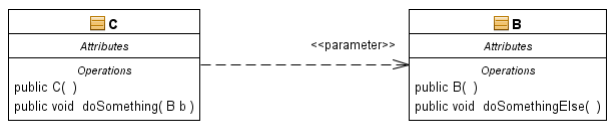
Law of Demeter 2

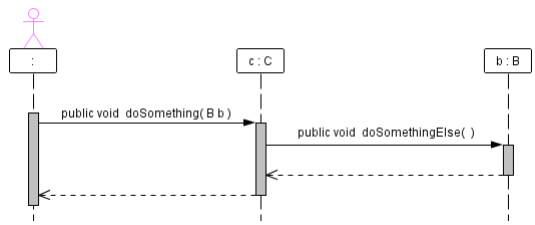
In response to a message M, an object O should send messages to objects which are sent as arguments to the message M

```
public class C {

   public C() {
   }

   public void doSomething(B b) {
      System.out.println("This is method doSomething() of class " + this.getClass().getName());
      b.doSomethingElse();
   }

   public static void main(String[] args) {
      C c = new C();                    // c is object of class C
      B b = new B();                    // b is object of class B
      c.doSomething(b);                 // message sent to c
   }
}
```


Law of Demeter 2 - in UML notation

In response to a message M, an object O should send messages to objects which are sent as arguments to the message M





Note the dependency (dashed) arrow, stereotyped by <<parameter>>. Recall that a dependency means the objects may be linked at runtime but an explicit association is not modelled at design time.

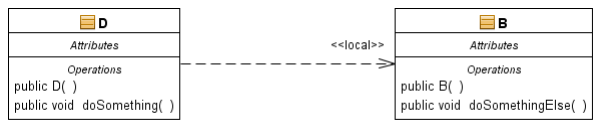
Law of Demeter 3

In response to a message M, an object O should send messages to objects which O creates as part of its reaction to M

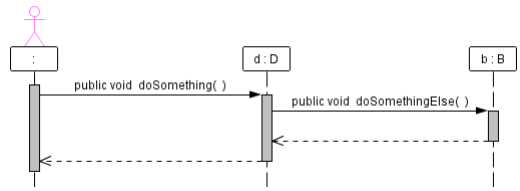
```
public class D {

   public D() {
   }

   public void doSomething() {
      System.out.println("This is method doSomething() of class "  + this.getClass().getName());
      B b = new B();
      b.doSomethingElse();
   }

   public static void main(String[] args) {
      D d = new D();           // d is object of class D
      d.doSomething();      // message sent to d
   }
}
```


Law of Demeter 3

In response to a message M, an object O should send messages to objects which O creates as part of its reaction to M



Again, note the dependency (dashed) arrow, stereotyped by <<local>>.
Here, the local denotes a locally scoped reference to an object of type B.



Law of Demeter 4

In response to a message M, an object O should send messages to objects which are directly accessible from O, that is, using values of attributes of O.

```
public class E {
    private B b;                    // b is reference to object of class B ;

    public E() {
       b = new B();
    }

    public void doSomething() {
       System.out.println("This is method doSomething() of class " +this.getClass().getName());
       b.doSomethingElse();
    }

    public static void main(String[] args) {
       E e = new E();               // e is object of class E
       e.doSomething();             // send message to e to doSomething()
    }
}
```


Law of Demeter 4

In response to a message M, an object O should send messages to objects which are directly accessible from O, that is, using values of attributes of O.