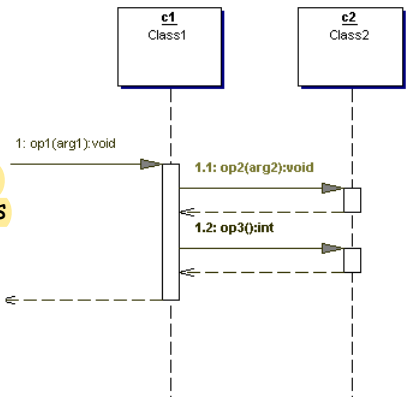
Implementation issues
Use case and sequence diagrams

- Use cases and sequence diagrams are closely related to each other:
  - A use case describes an interaction that an *actor* will have with *the system*
  - Use cases are not "implemented" directly (they can provide guidance for validation/acceptance testing)
  - Typically there may be one sequence diagram for each use case
  - The sequence diagram will show how an input "stimulus" from an actor causes interaction *between a number of objects* within the system
  - The "stimulus" could be an event (in the GUI, say), a message, or an operation/method call
    (bottom line: all these are really the same thing!)
- We will concentrate on implementing sequence diagrams

Implementation from sequence diagrams

- We do not really "implement a sequence diagram"
  - A sequence diagram guides us in implementing different parts of the system usually scattered over a collection of classes
- The core component of a sequence diagram is a message sent from one object to another:
- Sequence diagrams emphasize the *time ordering* of messages sent between collaborating objects



- Each message from a "client" *to* a "supplier" (eg from c1 to c2 in previous diagram):
  - *Arises from* a statement in a *method* of the client object *sending* the message (c1)
  - (or possibly initialization code, for example in instance variable declarations),
  - Is a *call* of a *public operation* in the *supplier object (c2) that the message is being sent to*
  - The client will use the returned value (if non-void)
  - Examples: In c1:

```
c2.doAction(...);
int info = c2.getInfo();
```

- So, there is a close correspondence between
  - The *sequence(s)* of messages *sent by* an object,
  - and the *body(ies)* of the operations in that object's class

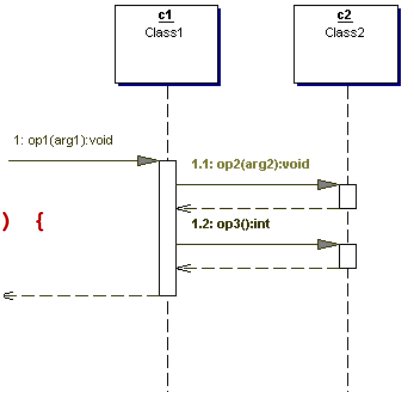
- In the "supplier", each message corresponds to
  - A call of a *public operation* in the supplier object the message is *being sent to*
  - And there may be a return message if the called operation returns a result
- So, there is a close correspondence between the kinds of messages *sent to* an object, and the *public operations* in that object's class
  - Examples: In Class2

```
public void doAction(...) {...}
public int getInfo() {...}
```

- If a sequence diagram contains enough information, Together can generate outline code from sequence diagrams!

Example

- The sequence diagram fragment again:



- Class1 might contain:
  - Attribute:

```
private Class2 c2;
```

  - Operation:

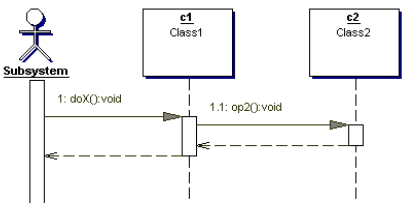
```
public void op1(arg1) {
  arg2 = ...;
  c2.op2(arg2);
  int a = c2.op3();
}
```

- Class2 might contain:
  - Operations:

```
public void op2(...) {...}
public int op3() {...}
```


Actors and boundary classes in sequence diagrams

- The initial message in a sequence diagram is (frequently) from an *actor* to a *boundary class object*
- This could simply be a call of a public operation from some other (part of the) system
  - For example a diary/calendar sub-system might "prompt" the core system to carry out an action at certain times



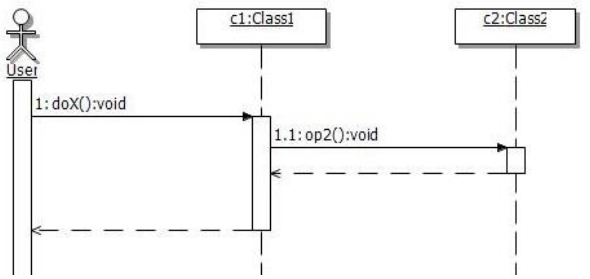
  - Class1 would need

```
public void doX() {...}
```

  - And Subsystem would contain:

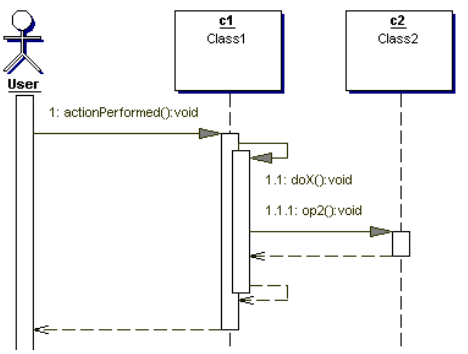
```
c1.doX();
```

- Or, perhaps more often, the actor is a *user* causing an action through a GUI:
  - For example, we might *model* the action like this, where Class1 is the *boundary class*:



  - But this is one step away from implementation through the GUI...

- In reality this will usually be handled through a call of an event handling method
  - For example, if the event is actually caused by a button click:



  - The JVM calls `c1.actionPerformed` automatically
  - Class1 contains `actionPerformed`, which calls Class1's `doX` to handle the action, which then calls `c2.op2`

- Class1 might contain:

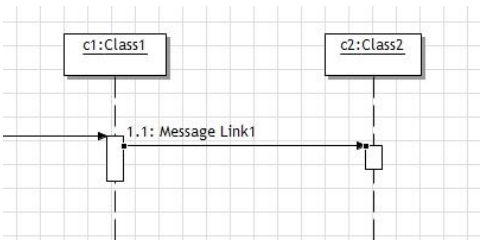
```
private Class2 c2;
public void actionPerformed(… e) {
  if (e.getSource() == …)
    doX(…);
  …
}
private void doX(…) {
  c2.op2(…);
}
```

- Class2 would contain:

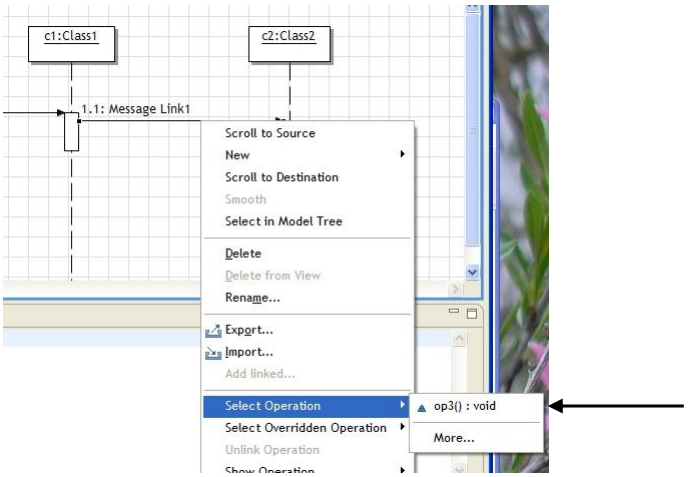
```
public void op2(...) {...}
```


Help from Together

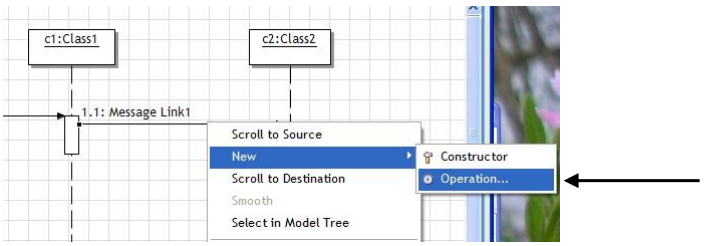
- Two kinds of implementation assistance are available from Together:
- If we create a message, Together can
  - Offer help linking it to an existing operation in the target class,
  - Or can create an appropriately named method in the target class
- Step 1:
  Create message:



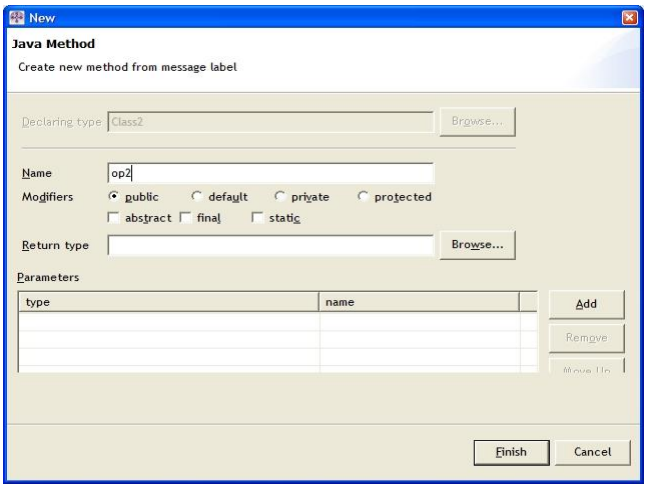
- Step 2: Choose an existing operation:
  - Right-click on the message and choose Select Operation
  - Operations available in the target class are listed



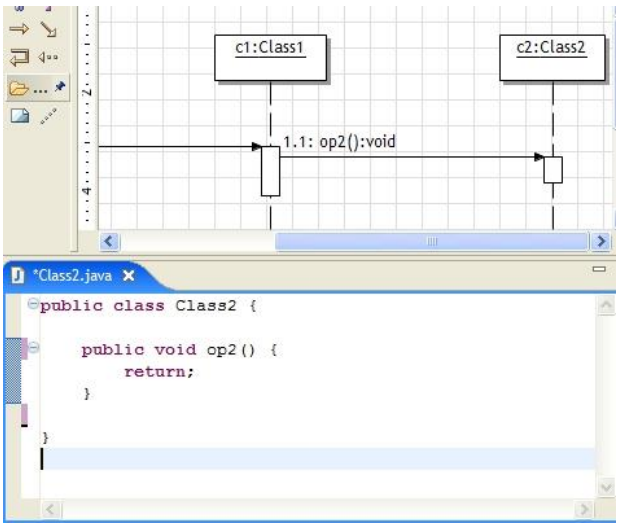
- Or Step 2: Create a new operation:
  - Right-click on the message and choose New / Operation



- Step 3: New method dialogue box appears:
  - Right-click on the message and choose New / Operation
  - Enter name and any other details



- Step 4: Method is implemented:



End of lecture