Implementation issues
Refactoring object oriented designs

- Definition (from the Together User Guide):

  "Rewriting existing source code, with the intent of improving its design rather than changing its external behaviour"

  - Applicable at the *design level* too
- Together provides built-in help with *low-level* refactoring options – see next slide
- We will also look at some higher level refactoring that Together does not yet support…

'Low level' refactoring options in Together

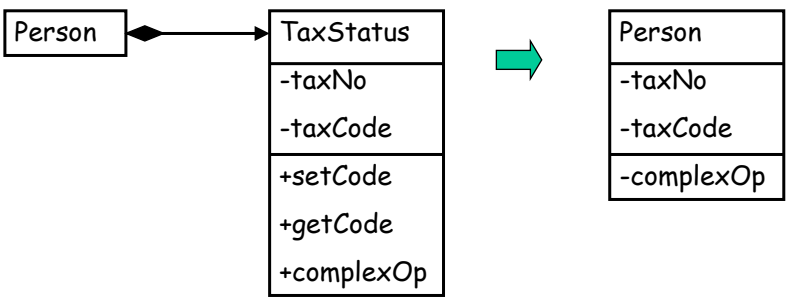
- The following options
  - Are accessed mainly via the right-click context menu on the item of interest
  - Have built-in intelligence about OO properties/constraints
- Options:
  - "Pushing down" an operation or attribute: super to sub class
  - "Pulling up" an operation or attribute: sub to super class
  - Renaming a class, interface, operation or attribute *throughout a whole project*
  - Renaming a parameter or local variable *throughout a method*
  - "Encapsulating" an attribute: making it private and creating get/set methods
  - Extracting new superclasses from single classes
  - Extracting code fragments to make new methods

'Higher level' refactoring operations

Collapsing objects

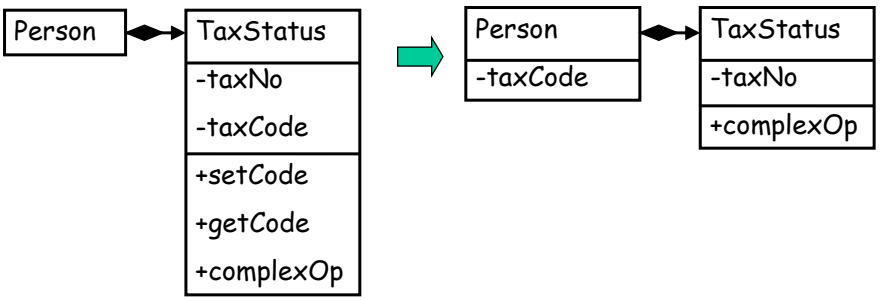
- If class B is a part of class A by *aggregation* or *composition*
  - And Class B is associated with no other classes
  - And we are happy conceptually with losing B as a separate modelled item (probably OK if B is simple)
- Then we may be able to remove class B:
  - Absorbing its attributes into class A, probably as *private*
  - And also its operations, which will become private, or *may disappear altogether*
- Example on next slide

- Simple example of collapsing:



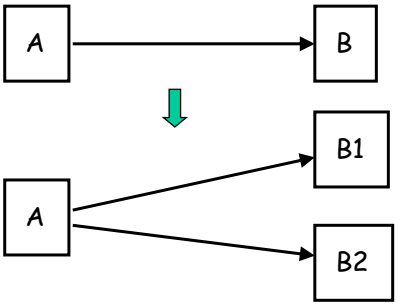
  - Note that **`setCode`** and **`getCode`** have vanished: their bodies have been "in-lined" in Person (optional)
  - Note that **`complexOp`** has *not* been inlined, but has become private – only needed by Person

- B&D suggest the following "partial collapsing" rule:
  - Attributes of a class that are *only used in getters and setters* are candidates for *moving* to the client class
  - The getters and setters are moved too – or in-lined and deleted
  - In this case the supplier class might not disappear
  - For example:



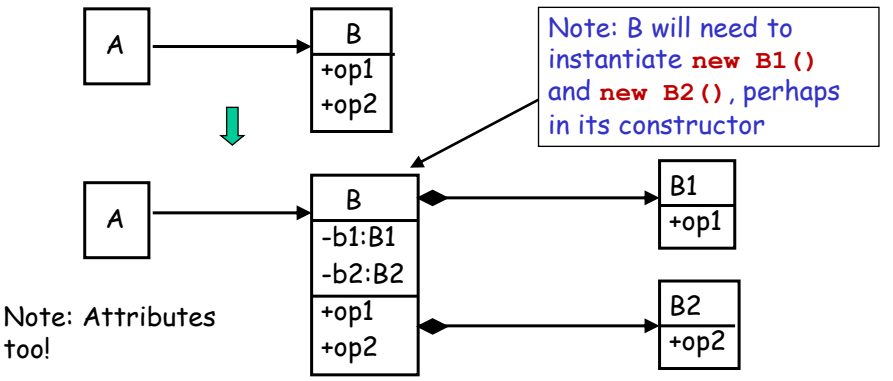
Splitting classes

- There may be benefits in *splitting* complex classes into two or more new independent classes
  - The new classes have a *more focussed identity*
  - Only really possible if the complex class partitions neatly into independent groups of attributes and operations (the class is *poorly cohesive*)
- It may be possible to achieve a complete decomposition of complex class B:



But this would imply that A's coding detail would also have to change
- better if we could avoid that

- In general it may be best to retain B as a *container* for instances of the new classes:
  - B would keep the same public operations
  - But their bodies would *delegate* the calls to appropriate instances of the new classes
  - A would *not have to change* – good!



- Introducing "delegating" methods:
  - Suppose that initially we have in B:

```
public int op1(String s) {
  … some action …
  return …some expression…;
}
```

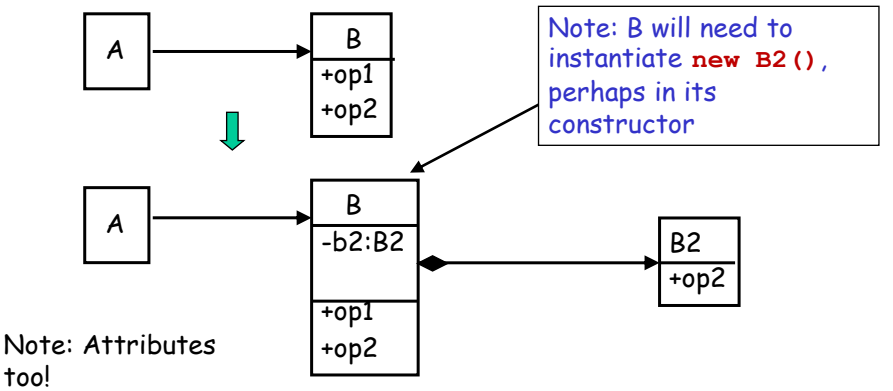
  - We move the whole operation to new class, say B1
  - But B *must still offer* `op1` *as a public service*
  - So we retain a method `op1`, but with a simple body that forwards the call the moved `op1`:

```
public int op1(String s) {
  int result = b1.op1(s);
  return result;
}
```

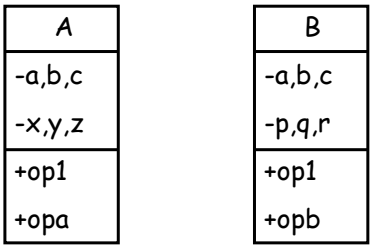
  - (And note that it returns the result from the forwarded call)
  - This can be adapted to no/more parameters, and `void` results

- The class splitting technique could also be used for the extraction of *cohesive parts* of a class to produce a new class
  - The new class would (hopefully) have a more focussed identity
  - An instance of the new class would be aggregated/composed into what remains of the original class
  - This is effectively the inverse of collapsing: neither is uniquely the best – in each case we must evaluate the (proposed) refactoring to determine whether it gives a better model or implementation
  - Example on next slide

- Example: Extracting a cohesive part of a class into a new class:



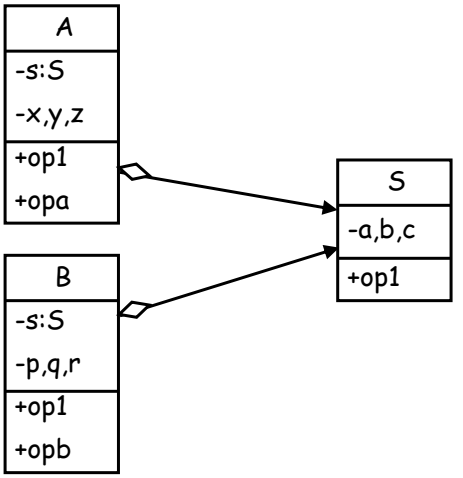
- The class splitting technique can also be used where several classes have common components, but extraction of those components to a new superclass would introduce undesirable *implementation inheritance*
  - For example:



in which the a, b, c, op1 are *identical* in A and B (possibly after renaming)

  - Refactored form: next slide

  - Refactored form: split/combine/aggregate/delegate



Introducing new superclasses: Refactoring to increase re-use within a system

- During development we may discover classes that share attributes and operations but with specialized components
- Introducing superclasses (possibly abstract) and inheriting from them may enable more and beneficial code re-use
- Note: The benefits may be short term if this gives implementation inheritance
  - If it does then using class splitting + delegation may work better

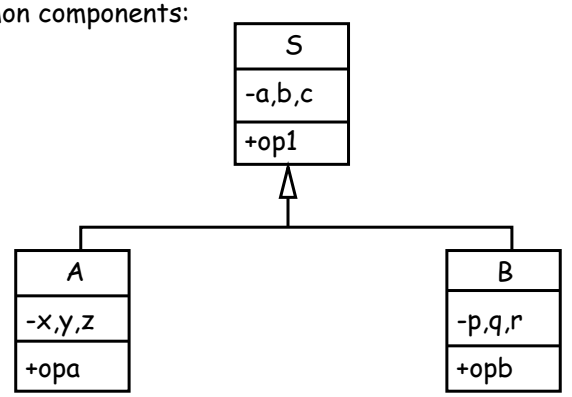
- Suppose that we have classes A and B:

A
-a,b,c -x,y,z
+op1 +opa

B
-a,b,c -p,q,r
+op1 +opb

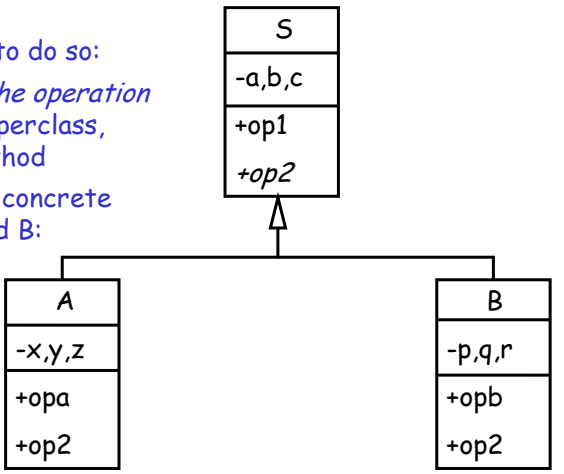
in which the a, b, c, op1 are *identical* in A and B

- Then we can introduce a superclass S by extracting the common components:



  - Only advisable if conceptually "A is-an S", "B is-an S", and so S is meaningful *in the model* – behavioural inheritance

- Variant: if A and B have an operation with the same name, formal parameters and result, say op2, but different *bodies* in A and B:
  - If it makes sense to do so:
  - Can extract *just the operation header* into the superclass, as an *abstract* method
  - Leaving overriding concrete definitions in A and B:



End of lecture