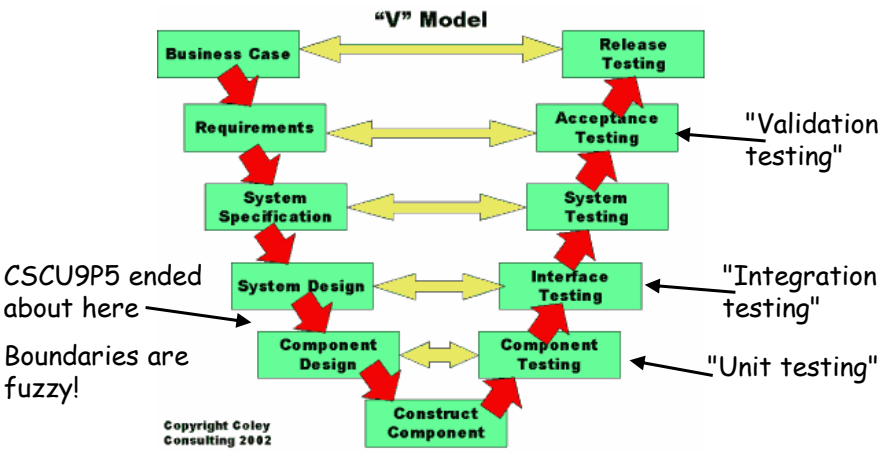
Implementation issues
Introduction, and Implementing associations

- 11 lectures, 4 practicals, Group project
- Will assume Java knowledge, Together, Eclipse
- The lecture plan:
  1. Associations
  2. Use cases, sequence diagrams
  3. State diagrams
  4. Refactoring
  5. Design patterns
  6. Implementing the MVC architecture
  7. Testing overview, debugging
  8. Integration testing
  9. JUnit testing
  10. Configurations, build control, Ant and Maven
  11. Collaborative working, version control, Subversion, Git

The V model

- A useful view of the relationship between analysis, design, coding and testing



February 2006: Diagram from http://www.coleyconsulting.co.uk/testtype.htm,

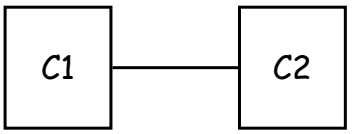
Detailed design activities

- Reviewing/refining the object model:
  - Adding/removing classes
  - Reviewing attributes & operations
  - Reviewing associations
  - ➢ All a natural part of iterative design
- "Refactoring" the object model:
  - Altering the internal details of the model without altering its "visible behaviour"
  - Adding/removing/splitting/merging classes/attributes/operations
  - ➢ At the modelling stage *to improve it as a model*
  - ➢ At the implementation stage perhaps *to improve manageability, to increase re-use, or for optimization, or to fit an architectural framework*

- Selecting an architectural framework:
  - A software architecture defines the system decomposition, global control flow patterns, inter-subsystem communication protocols
  - ➢ A *standard framework* can make design and implementation easier (eg MVC)
- Concretizing associations
  - Fixing details
  - Choosing an implementation
- Implementing the classes
  - Coding the public services
  - Introducing private attributes and operations
  - ➢ Basically *programming* rather than design
- The *boundaries* between these activities may be fuzzy – and remember *iteration!*

Refining associations

- An *association* represents a *conceptual relationship* between *instances* of classes
  - Initially perhaps only a recording of thoughts during analysis: "C1 and C2 have something to do with each other"



Note: We will always be thinking about *instances* of C1 and C2

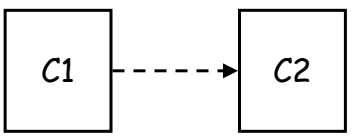
- As our understanding of the design improves, and as implementation progresses we *refine* the association:
  - Maybe it is just a *dependency*
  - Or perhaps one class holds/refers to instances of the other
  - Decide whether it is conceptual *aggregation* or *composition*
  - Identify its *navigability/directionality*
  - Identify its *multiplicity*

Dependencies

- Definition of a "dependency":
  - C1 depends on C2 if C1 has an operation parameter or result of type C2, or a local (temporary) variable of type C2
  - C1 has no persistent "ownership" of, or access to, an object of class C2 – no attribute – *but C1's code might need to change if C2 changes*
- Example: In class C1:

```
public void m(C2 c) { ... c.n(...) ... }
```

- Diagrammatically:



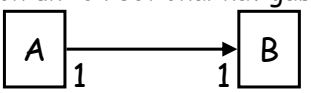
- Nothing special is required in implementing a dependency
- *But we need to remember to review C1 if we later change C2's public interface*

Attribute based associations

- If an association is *not* a simple dependency
  - C1, say, has *persistent* ownership of, or access to a C2
  - C1 may call on the services (public operations) of C2
  - So, the association will be *navigable from* "client" C1 *to* "supplier" C2
- To implement this C1 will have an *attribute* holding an instance of C2
  - In Java: a "global"/"member"/"instance" variable holding a *reference* to a C2
- *The actual details may vary depending on multiplicities, and decisions about aggregation and composition*
- So, for an initial simple association between C1 and C2 we need to decide on
  - Multiplicities
  - Direction of navigability
  - Aggregation/composition

Implementing associations

- 1 - 1 association uni-directional navigability from A to B



  - Easy: A has an attribute (instance variable) holding a (reference to an instance of) B

```
private B theB;
```

  - The attribute must be initialized somehow:

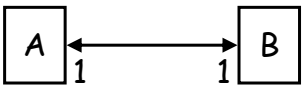
    Possibly `private B theB = new B(…);`

    Or via A's constructor

```
public A(B someB) {
    theB = someB;
}
```

    Or via a call of `public void setTheB(B someB)`

- 1 - 1 association *bi-directional* navigability from A to B (Rare?)



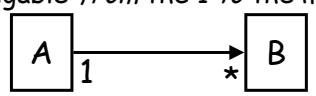
  - Each class has an attribute holding a reference to (an instance of) the other:

```
private B theB;        private A theA;
```

  - Setting up the references similar to the uni-directional case
  - Needs care if mutual references between two objects are required! E.g in **A:**

```
private B theB = new B(...);
...
theB.setTheA(this);
```

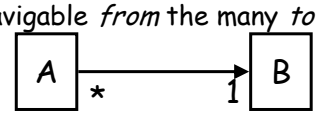

- 1 – many, navigable *from* the 1 *to* the many:



  - For example: Library 1 → * Book
  - A must have an attribute that is a *collection of references to instances of B*: eg array, **Vector**, **HashTable**…
    Perhaps:

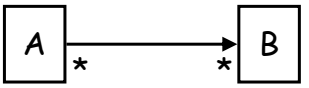
```
private B[] theBs = new B[100];
```

- 1 – many, navigable *from* the many *to* the 1:



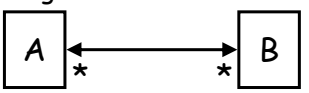
  - For example: Document * → 1 Template
  - Easy: A simply needs an attribute **private B theB** (as for 1-1)
    (and the * is not represented in the code! It records design intention/understanding)

- Many – many, navigable *in one direction only*:



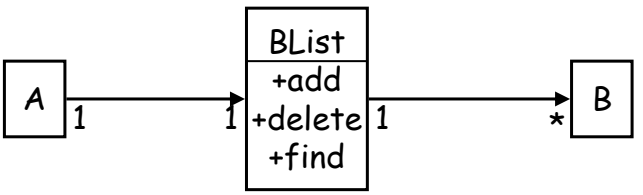
  - For example: Students * → * Courses
  - As for 1 – many
  - (And the * at A only records design intention/understanding)

- Many – many , navigable *in both directions*:



  - For example: Students * ↔ * Courses
  - Both A and B must have attributes that hold a collection of references to the other

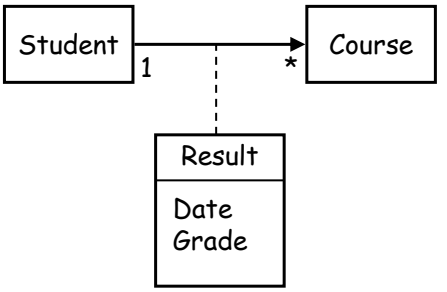
- An important alternative approach for implementing 1-many navigable from A to B: e.g. Library 1 → * Book
  - A might contain many attributes & operations concerned with its "A-ness"
  - Its collection of references to instances of B might also need a number of operations (add, delete, find…)
  - So it may be better/neater/easier to separate out a collection class, eg:



  - And perhaps BList contains an array of Bs, etc

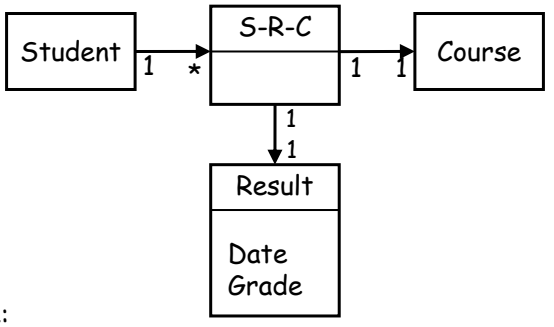
Implementing "association classes"

- An association class is information attached to an association
- For example:



- Result is not naturally a part of Student nor Course
- The meaning is: one instance of Result attached to *each* link from a Student to an instance of Course

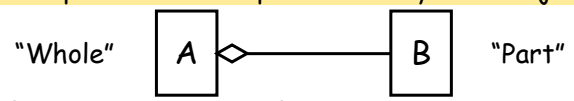
- This can be *implemented* in the following way:



- Note:
  - This version is an *implementation* refinement, not necessarily a good *design* representation
  - The detailed coding has to ensure that links are created consistent with the original intention

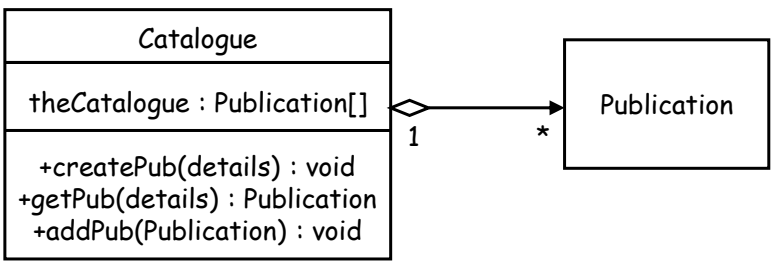
Aggregation

- *Aggregation* indicates a *conceptual* hierarchical structure – a whole-part relationship between *separate* objects



- Often 1 – 1 or 1 – many from A to B
- Navigability is often from A to B
  - For example Catalogue 1 → * Publication
- The designer is indicating that although collected within A, the object(s) B have a "separate existence" from A:
  - They may be created outside A
  - They may be passed to A as parameters, or returned by A as results
  - So *other objects may have references to them*
  - A *must* hold *references* to Bs to enable this (the only option in Java)

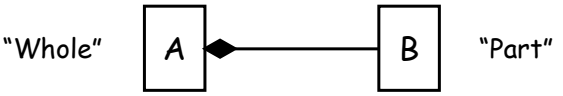
- Example:



- It is reasonable that Publications can be used independently of the Catalogue
- The Catalogue can receive and return references to Publications
- So, aggregation is correct here
- In Java, aggregation is implemented in the same way as, for example, a 1 – many association

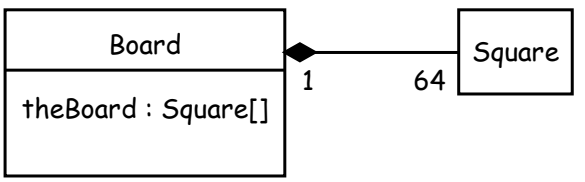
Composition

- *Composition* is a strong form of aggregation – the designer is indicating that the "parts" have *no separate conceptual existence from the whole*



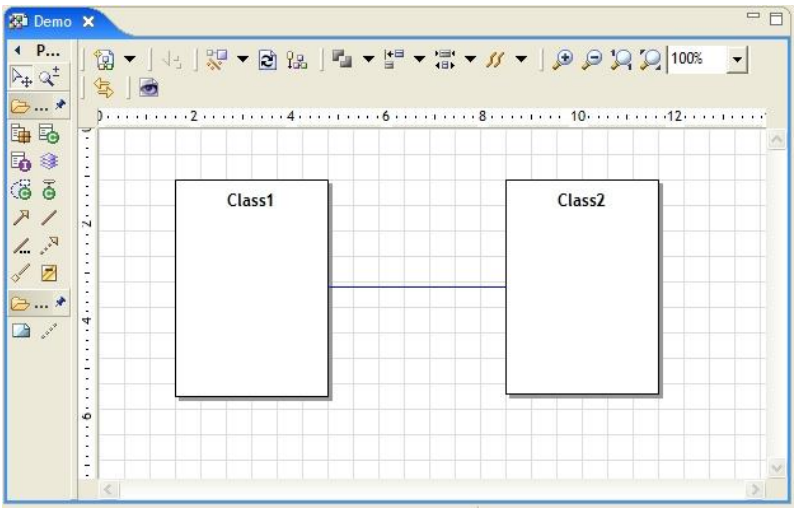
- Objects of class B are created and destroyed by A
  - Each instance of B belongs to one and only one instance of A
  - And A never "gives away" references to its Bs
- So, when an instance of A is destroyed, all its Bs are too

- Example:

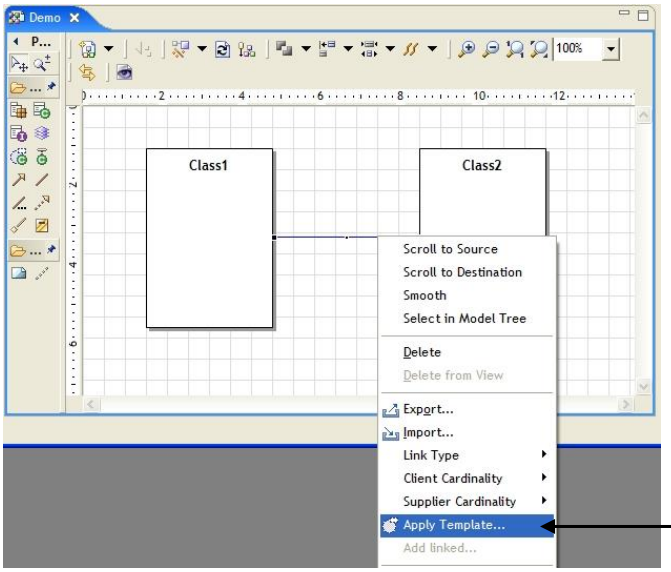


- It would be unusual if any public Board operation received or returned a Square object!
- The Board itself must create the Squares – maybe in its constructor
- So, composition is correct here
- In Java, composition is implemented *in the same way as aggregation*! There is no special mechanism
  - In C++ instances of Square could be held *directly* in Board (not as separate objects)

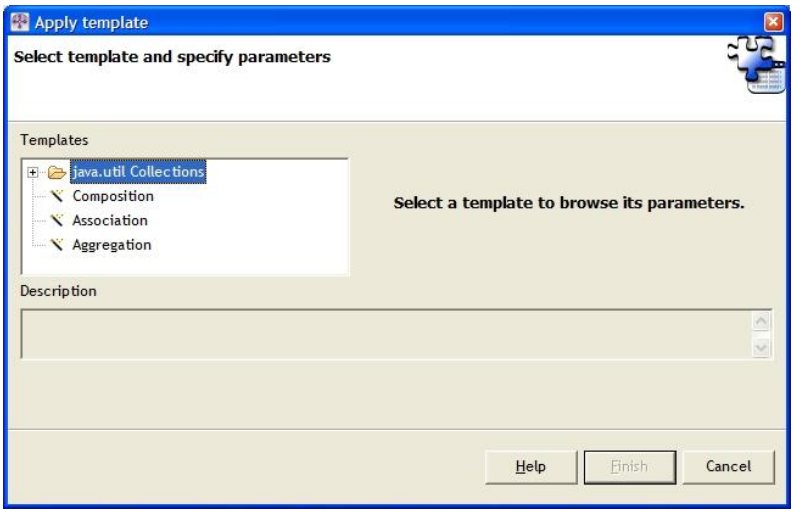
How Together can help



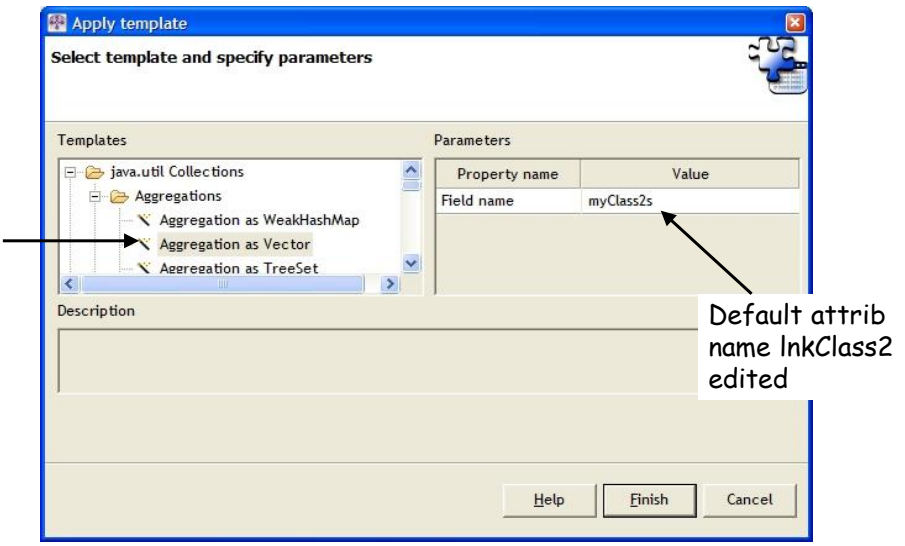
- Right-mouse click on association link and select Apply Template



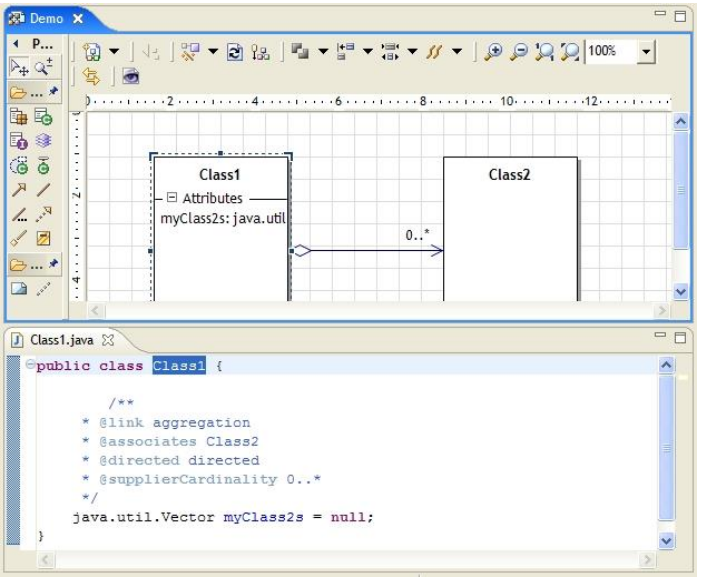
- Template dialogue appears



- Select, eg, Aggregation as Vector



- The resulting class diagram and code



End of lecture