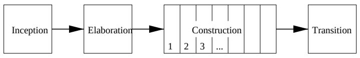
A lightweight, low-ceremony software development process



- *Inception* Establish business rationale for the project; decide on its scope; state the purpose

- *Elaboration* Do the following hand-in-hand:
  - Build a model of the problem domain: describes the world, not the software; use Class Diagrams and Sequence Diagrams
  - Collect more detailed requirements by identifying *use cases* (typical interactions between users and the system that achieve user goals)

Our simple process

- *Construction* Comprises many iterations; each is a mini-project that builds production-quality software to satisfy a subset of the use cases:
  - *analysis:* specify the software; use Class Diagrams and Sequence Diagrams
  - *design:* decide on classes that will do the work and how they will do it; use Class Diagrams, Sequence Diagrams and GUI mock-ups
  - *implementation:* write code for each class and do 'unit testing' (see `Fraction.java`)
  - *integration:* 'integration testing'

  Each concludes with a demo to the user

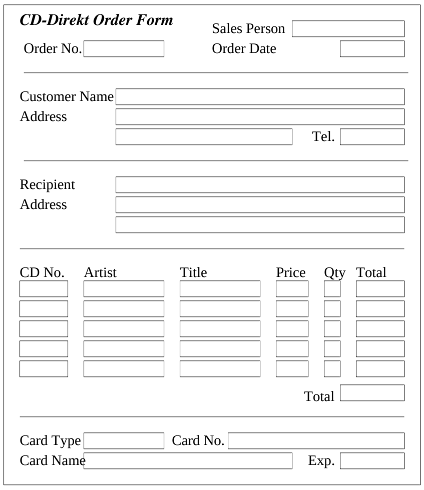
- *Transition* Beta testing, performance tuning, user training, etc.

Simple case study

CD-Direkt is a CD tele-sales company. It advertises exclusive CDs on television. Customers ring up to place orders for CDs. The sales person uses an order form to record customer details, order details (including the recipient of the order if different from the customer, and credit card payment details) and order lines (to record which CDs the customer wants to purchase). Information about CDs is kept in a ring-binder, containing one A4 sheet per CD. A credit check is made with the credit card company, while the customer is still on the line.

Completed orders are passed to the warehouse staff for despatch. The warehouse staff keep the outstanding order forms in a box near the kettle.

Sometimes customers ring up to find out the status of their orders. These calls presently get forwarded to the warehouse staff who can tell customers whether an order has been despatched or is waiting for stock.


CD-Direkt Order Form

Sales Person
Order No.
Order Date

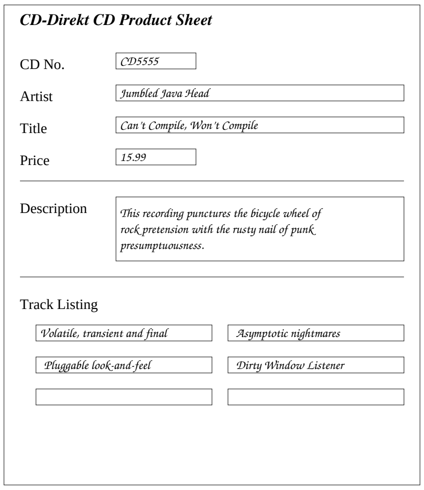
Customer Name
Address
Tel.

Recipient
Address

CD No.	Artist	Title	Price	Qty	Total

Total

Card Type
Card No.
Card Name
Exp.


CD-Direkt CD Product Sheet

CD No. CD5555
Artist Jumbled Java Head
Title Can't Compile, Won't Compile
Price 15.99

Description This recording punctures the bicycle wheel of rock pretension with the rusty nail of punk presumptuousness.

Track Listing
Volatile, transient and final
Asymptotic nightmares
Pluggable look-and-feel
Dirty Window Listener

Inception

- It takes a long time to record an order (re-recording details of even those customers who have purchased from the company before; finding CD details in the ring-binder; calling the credit card company). This implies lost sales (people who can't get thro' give up.)

- Enquiries about orders are handled inefficiently. This implies lost sales (people don't re-order if they get a bad impression).

- *The Managing Director wants us to automate order entry and tracking to provide faster call processing.*

Elaboration

We model the problem domain and determine use cases.

Problem domain class identification

- "Underline the nouns in your requirements document"
  - easy for teaching; irrelevant in practice
- Problem domain classes will be revealed
  - in discussion with domain experts (recurring terms, terms they specially define)
  - by seeing physical manifestations (seeing customers, order forms, etc.)
  - from existing software systems (the contents of files & data structures)
  - from refining your diagrams, e.g. classes sometimes lurk behind relationships (especially many-to-many ones).

Use case identification

- Use cases satisfy user goals:
  - set-up
  - conducting the business
  - reporting business results
- Completeness at this stage is impossible; change is inevitable; good software development processes anticipate such change
- Use cases specify *functional requirements* (desired system functionality); it might be possible to attach *non-functional requirements* to uses cases (performance/ reliability requirements)

Construction: analysis

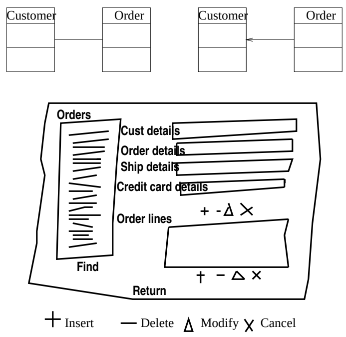
- Draw a Class Diagram, but now the classes describe objects that will be represented in the software. (In Elaboration, they were objects in the business (problem domain).)
  - Problem domain classes might have corresponding software classes
  - Boundary classes that mediate between the software and its users and other external systems
  - Control classes that coördinate workflow between different objects. (Ignore these until you're more experienced)
- Draw Sequence Diagrams for each use case using objects whose classes are in your Class Diagram

Software class identification

- A software class is an abstraction around data
  - state: attributes/instance variables
  - behaviour (the services): operations/instance methods
- Candidate classes that aren't data abstractions should come under suspicion
  - Classes whose names are actions (ProcessX, PerformX, ReadX, etc.) are suspect (unless they're control classes)
  - Classes that have attributes only or operations only are suspect
- But, before throwing classes away, consider whether candidate classes that don't measure up now might measure up in the future, when the system evolves

Design: refining the models

- Add more data structures & GUI classes to the Class Diagrams; redraw the Sequence Diagrams accordingly. How to identify what classes to add:
  - by knowing design patterns (e.g. Model-View-Controller)
  - by knowing code libraries
  - through courses on data structures, GUIs, etc.
  - through experience
  - by discussion with other programmers
- Add low-level detail to the diagrams, e.g. navigability arrows on associations
- Draw GUI mock-ups



Implementation

- It should be easy to code up each class and the relationships between classes (IDEs even automate the production of skeletal code from UML Class Diagrams)
  - The interface is the list of operations (plus perhaps some getters/setters that were too obvious to include until now)
  - The attributes suggest what instance variables you need
  - The Sequence Diagrams help you to write the bodies of the methods
  - Generalisation is typically implemented by using super/subclasses (`extends`)
  - Association is typically implemented by storing references (pointers) in instance variables

Implementing associations: methods needed

- Where max. multiplicity is 1, you need a getter & a setter

```
public class Order
{  public Customer getCustomer()
   public void setCustomer(
                     Customer theCust)
}
```

- When max. multiplicity > 1, you need an add method, a remove method and a getter that returns an array/Vector/other data structure/Enumeration

```
public class Customer
{  public Vector getOrders()
   public void addOrder(Order anOrder)
   public void removeOrder(Order anOrder)
}
```


Implementing associations: multiplicities

- Constructors and the methods above must enforce other multiplicity constraints

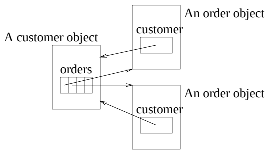
- E.g. if min. multiplicity is > 0, maybe the constructor should insist on a non-null argument

```
public Order(Customer theCust)
{  if (theCust == null)
   {  // error
   }
   ...
}
```

- E.g. if min. is 1, then error if setter tries to set to null, or if add method tries to remove the last one

- E.g. if max. is 5, then error if add method tries to add a 6th

Implementing associations: instance variables at both ends



- Efficient navigation in both directions

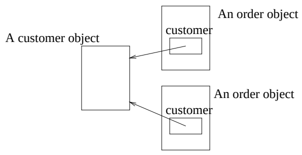
- Uses more space

- Slower update because updates to one must be reflected in the other. We can leave it to the programmer to remember to call both methods from the client code but what if s/he forgets? (But trying to propagate updates automatically has either circularity or visibility problems.)


```
public class Order
{  public Customer getCustomer()
   {  return customer;
   }
   public void setCustomer(Customer theCust)
   {  customer = theCust;
   }
   private Customer customer;
}

public class Customer
{  public Vector getOrders()
   {  return orders;
   }
   public void addOrder(Order anOrder)
   {  orders.addElement(anOrder)
   }
   public void removeOrder(Order anOrder)
   {  orders.remove(anOrder);
   }
   private Vectors orders;
}
```


Implementing associations: instance variables at one end only



- Efficient navigation in only direction (order to customer). To navigate in the other direction (customer to order), visit *all* orders and return those that reference the customer you wanted to navigate from. (Implies that all orders are stored in some data structure)

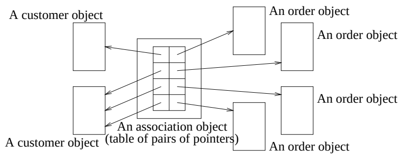
- Uses less space

- Faster, safer update because no consistency problem

```
public class Order
{  // As before
}

public class Customer
{  public Vector getOrders()
   {  /* Walk through all orders and put
         those that reference this
         customer into a Vector.
       */
   }
   /* Could omit these next methods or
      define them as follows:
    */
   public void addOrder(Order anOrder)
   {  anOrder.setCustomer(this);
   }
   public void removeOrder(Order anOrder)
   {  anOrder.setCustomer(null);
   }
}
```


Implementing associations: using an association object



- Customer & Order need no instance variables to 'point' at each other. There are as many entries in the table as there are associations
- Navigation requires search of the table. The speed of this depends on the way the table is stored. (E.g. hashing could improve performance)
- There are no update consistency problems, e.g. removing an entry from the table removes both 'pointers'


```
import java.util.*;

public class CustomerOrderAssociations
{
   public CustomerOrderAssociations()
   {  assocs = new Vector();
   }

   public Customer getCustomerFor(
                             Order anOrder)
   {  Enumeration e = assocs.elements();
      while (e.hasMoreElements())
      {  Pair pair = (Pair) e.nextElement();
         if (anOrder == pair.order)
         {  return pair.customer;
         }
      }
      return null;
   }
```



```
   public Vector getOrdersFor(Customer aCust
   {  Vector results = new Vector();
      Enumeration e = assocs.elements();
      while (e.hasMoreElements())
      {  Pair pair = (Pair) e.nextElement();
         if (aCust == pair.customer)
         {  results.addElement(pair.order);
         }
      }
      return results;
   }

   public void addAssoc(Customer aCust,
                             Order anOrder)
   {  // Should also test multiplicity
      // constraints!
      Pair pair = new Pair(aCust, anOrder);
      assocs.addElement(pair);
   }
```



```
   public void removeAssoc(Customer aCust,
                             Order anOrder)
   {  // Should also test multiplicity
      // constraints!
      Pair pair = new Pair(aCust, anOrder);
      assocs.removeElement(pair);
   }

   private class Pair
   {
      Pair(Customer aCust, Order anOrder)
      {  customer = aCust;
         order = anOrder;
      }

      public boolean equals(Object anObj)
      {  if (anObj == null ||
            ! (anObj instanceof Pair))
         {  return false;
         }
         Pair pair = (Pair) anObj;
         return customer == pair.customer &&
                  order == pair.order;
      }
```



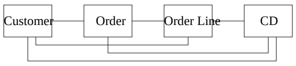
```
      Customer customer;
      Order order;
   }

   private Vector assocs;
}
```

Example client code:

```
CustomerOrderAssociations mappings =
   new CustomerOrderAssociations();
mappings.addAssoc(c1, o1);
mappings.addAssoc(c2, o2);
mappings.addAssoc(c2, o3);
System.out.println(
   mappings.getCustomerFor(o1));
System.out.println(
   mappings.getOrdersFor(c2));
mappings.removeAssoc(c2, o3);
```

Implementing associations: derived associations

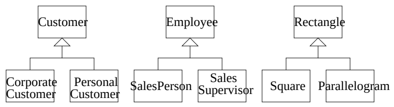


- Some associations can be derived from others; you may have excluded them from earlier class diagrams
- If they're still there and you want to implement them, you can use any of the first three approaches but this gives additional consistency problems
- Or you can compute them 'on demand' using the associations from which they are derived

Generalisation: are you sure it's generalisation?

- Every B is a special kind of A
- An object belonging to a subclass will forever remain in that subclass
- It should inherit attributes/operations, extend with new attributes/operations, preferably not override operations (unless they were abstract), and definitely not ignore or otherwise try to nullify what is inherited

Are these cases of generalisation?



Implementing generalisation: inheritance

```
public class A          | public class B
{                       |   extends A
  protected int var1;   | {
}                       |   protected int var2
                        | }
```

- Simple & clear
- Efficient
- Gives polymorphism: variables of type A can reference objects of class B
- But quite high coupling: the implementation of A is exposed to B, so changes in A may have knock-on effects in B

Implementing generalisation: duplication

```
public class A          | public class B
{                       | {
  protected int var1;   |   protected int var1
}                       |   protected int var2
                        | }
```

- Simple & clear
- Efficient
- No polymorphism
- Duplication gives a maintenance problem

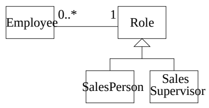
Implementing generalisation: optional attributes

```
public class A
{
   private int var1;
   private int var2;
   private int whichSubclass;

   public static final int B = 0, C = 1;
}
```

- Collapsing subclasses into a single class gives a loss of cohesion
- There is wasted space and methods will be more complicated (and less efficient), e.g. testing the flag & making sure certain variables are null when the flag has certain values
- No real polymorphism (although there's now only one class anyway)

Implementing generalisation: using association & inheritance



- `Employee` contains a variable of type `Role`. This is used polymorphically to store whichever role subclass the employee currently plays.
- This approach has huge flexibility — see dynamic classification, multiple classification, the adapter design pattern

```
public class Employee
{  // Stuff common to all employees goes her
   // plus:

   private Role role;
}

public abstract class Role
{  // Stuff common to all roles would go her
}

public SalesPerson extends Role
{  // Stuff specific to sales people goes he
}

public Supervisor extends Role
{  // Stuff specific to supervisors goes her
}
```


Dynamic classification

- Suppose an object can change which subclass it belongs to (dynamic classification)
- If you use inheritance
  - create a new object from the new subclass
  - copy as much data as possible from the existing object
  - 'delete' the existing object

  You lose information about the object; you have no sense of history
- Similar problems if you use duplication (above)
- A bit easier if you use optional attributes (above), but this has too many other failings

Dynamic classification, cont'd

- If you use association, dynamic change of role can easily be implemented:

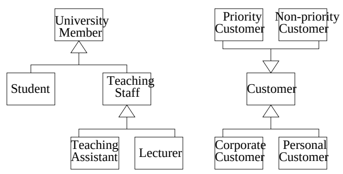
```
Employee e = new Employee(...);
Role r1 = new SalesPerson(...);
e.setRole(r1);
// After promotion...
Role r2 = new Supervisor(...);
e.setRole(r2);
```

- Keeping 'history' is easy too (when the employee changes role, put the old role into the vector and the new role into `currentRole`):

```
public class Employee
{  private Role currentRole;
   private Vector pastRoles;
}
```


Multiple classification

- Suppose an object can simultaneously belong to more than one subclass (multiple classification). Examples of how this might happen (see also multiple inheritance):

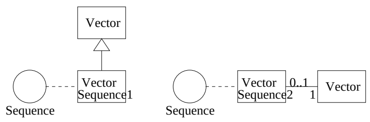


- Easy to implement using association with roles:

```
public class Employee
{  private Vector currentRoles;
}
```


Reuse and the Adapter Design Pattern, revisited

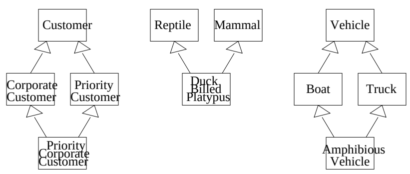
- Compare `VectorSequence1` and `VectorSequence2` from earlier in the course
- `VectorSequence2` achieves reuse using the Adapter pattern; effectively, this uses association instead of inheritance



- So this is another place where association is winning over inheritance

Multiple inheritance

- Suppose one class is a direct subclass of more than one superclass; it inherits from more than one direct superclass. Examples:



Note how multiple inheritance implies multiple classification (but multiple classification does not imply multiple inheritance)

Multiple inheritance: programming language support

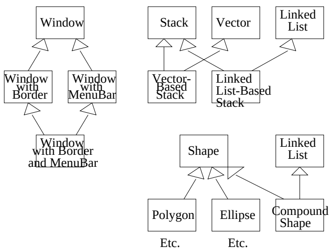
- C++ & Eiffel directly support multiple inheritance from classes; so implementation is easy
- They have ways of resolving name clash problems where you inherit different variables/methods with the same name from different parents
- They also have ways of resolving the problem of inheriting the same variable by more than one route: does the subclass get one copy or two?
- Java, Simula67, Ada & Smalltalk do not support multiple inheritance from classes (Java allows multiple inheritance from interfaces but only single inheritance from classes.)
- For these languages, an alternative implementation is needed.

Multiple inheritance: how to implement in languages with only single inheritance

- Use duplication (above) for at least one parent
- Use optional attributes (above) for at least one parent
- Use association for at least one parent
- A final approach is to question whether your class diagram is correct to use multiple inheritance at all. . .

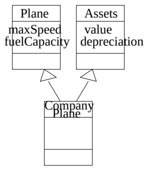
Multiple inheritance: is it really there (part I)?

Are the tests passed (Every B is a special kind of A; an object doesn't change subclasses; B extends A and overrides a little but ignores/nullifies not at all) (In particular, would one relationship be better modelled by association?)



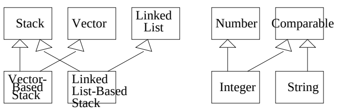
Multiple inheritance: is it really there (part II)?

Are both parents really part of your problem domain/software solution (Many examples of multiple inheritance are contrived and wouldn't arise if irrelevant classes were excluded)



Multiple inheritance: is it really there (part III)?

Is one of the parents an abstract class? And does it have no concrete methods? If so, in Java you can use a Java interface instead.



Course recap

- *Introduction:* Cohesion; coupling
- *Case study I (fractions):* revision; starting with an interface; information hiding; methods inherited from `Object`; recording method preconditions in `javadoc` comments; testing
- *Case study II (data structures package):* packages & visibility; inner classes (member classes); Java interfaces (their polymorphism & dynamic method binding); programming to an interface; reuse by inheritance; reuse by association; the adapter design pattern; design patterns; the iterator design pattern

Course recap

- *Case study III (low-grade web browsing):* UML class diagrams; bad & good menu implementations; the command design pattern; defensive programming; defensive programming using Java exception handling; principles of good exception handling; precondition checking; design-by-contract

- *Case study IV (Derek's burger bar):* event-driven programs; architectures for simple programs with GUIs; UML sequence diagrams; model-view-controller; the observer design pattern

- *Case study V (CD-Direkt order entry):* the software development process; mapping UML class diagrams to Java (association & generalisation)