UCL

COMP1008
Developing Classes

UCL

Agenda

- For the next block of lectures we will be looking at designing and writing an OO program requiring a small number of classes.
- Along the way we will:
  - Get an overview of the development process.
  - Learn more about being Object-Oriented.
  - Look at the details of writing classes in Java.
  - Look at more of the Java language.
  - Look at testing.

UCL

Development Activities

- Requirements Gathering
- Requirements Analysis
- Analysis
- *Design*
- *Implementation*
- *Testing*
- Delivery, Deployment
- Maintenance

Requirements List
Use Cases

OOA/OOD

OOP

Process - Organising the Activities

- Could work through each activity one after the other?

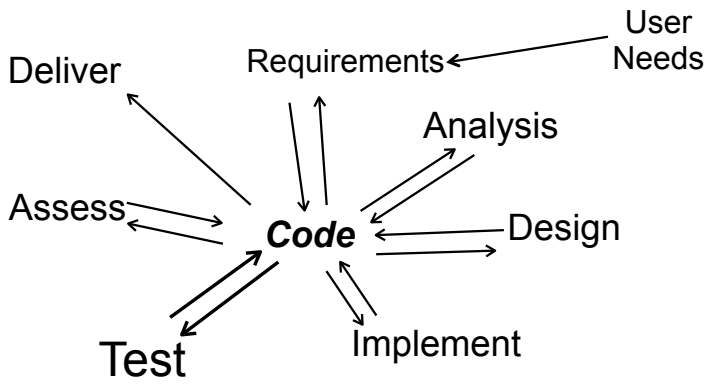
Won't work!

- Need iteration, exploration and experimentation:

"Analyse a little,
Design a little,
Code a little,
Test a lot"

- I'll be taking a code-centred approach.
  - In the 2nd & 3rd year Software Engineering courses you will look at process in much greater depth.

Iterative Programming



5

Everything is about design

- At different levels of abstraction.
- Getting all the details correct.
- Building the correct thing.
- Confirming it does work.
- Confirming it does what users actually want/need.
- Making it work well.

Good Design/Programming

… is hard to do!

Trying to predict the future

- It will work (won't it?)
- I can program this (can't I?)
- I don't make mistakes (do I?)
- I do good quality work (don't I?)

So,

- How do you go about designing and implementing an object-oriented program?
- How do you know the program code you are writing is:
  - Correct?
  - Good quality?
  - Robust?
  - Reliable?
- We want to start addressing these issues.
  - You will never stop learning about the answers throughout your professional career.
  - Always expect rapid change.

Key Issues

- Knowing how to test your program.
- Really knowing how to test your program.
- Knowing how to design and implement your program.
- Knowing how to judge the quality of your work (including self assessment).

All are hard!

The Simple Order System

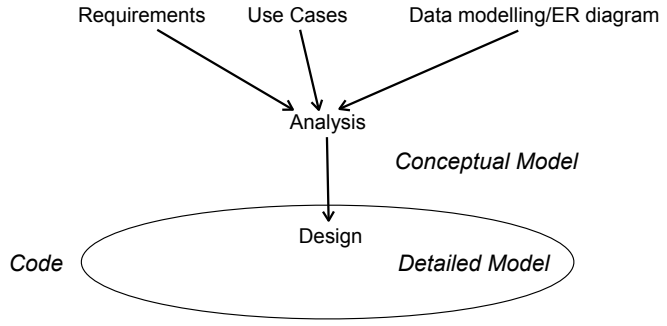
- Maintains a list of Customers.
  - Each Customer has zero or more orders.
  - Each Order consists of one or more LineItems
  - Each LineItem is for a quantity of 1 or more of a specific Product.
- Has a simple user interface.
  - New Customer can be added.
  - New Order can be entered for customer.
  - New Product can be added.
- Stores data in files.

Is that all?

- Yes, for this iteration.
  - Get a basic program working, then extend it.
- Is this realistic?
  - To a limited extent, this is an example, focussed on learning Java!
    - This is a programming course.
  - For a "real" program we would do a lot more requirements gathering and data modelling.
  - You will be looking at requirements/analysis modelling and data modelling in a lot more detail in other courses.
    - But will give an overview here.

Modelling

- Construct representations of required behaviour and structure of application.
- Developed in iterations.



Requirements

- List, number, organise requirements.

R1: A customer has a first name, family name, postal address, phone number, email address.

R2: A product has a code, description and price.

...

R10: A new customer can be added by inputting customer details.

R11: A new order can be created for a customer (who must already exist).

...

R22: A product can be added to an order as a line item.

R23: Product quantity can be one or more, combined into the same line item.

...

Use Cases (Stories)

- A Use Case describes a task an *actor* (a user in this example) can perform with the program.
- Describes what the application should do, not how the code works.
- Lists steps carried out by actor and responses made by program.

Use Case Example

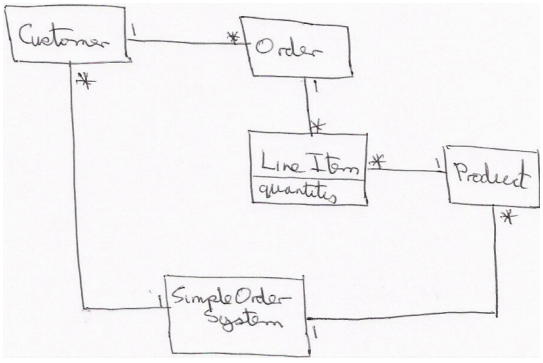
U1: Add a new customer
Actor: User
Pre-conditions: main menu displayed
Sequence:
1. User selects add customer from menu.
2. Program prompts for user name.
3. User enters name.
4. Program prompts for postal address.
... etc.
10. Program confirms that new customer created.
Post-conditions: New customer added.
Alternatives: None
Errors: 10a: Program reports customer already exists and makes no change.

Analysis

- For this example:
  - Identify key classes and relationships.
  - Following an Object-Oriented approach.
  - Most classes will represent a data item.
  - Construct a Conceptual Model first and then progressively refine.
  - Add methods by identifying how a task is performed by objects.

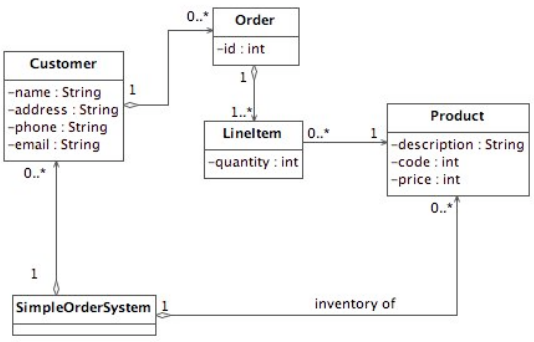
The Object-Oriented Perspective

- Conceptual Diagram, First sketch:

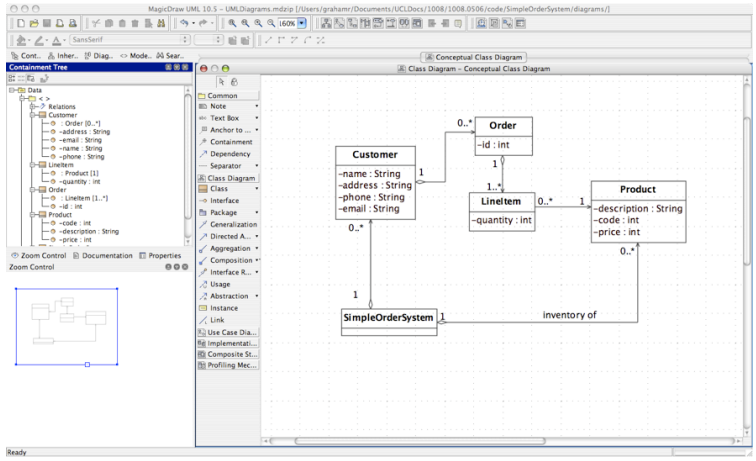


Refinement

- Starting to fill in details by working through requirements and use cases.
- Used an UML modelling tool to create a basic model.



What the tool looks like



Use Cases and Methods

- A use case (story) must be implemented via a sequence of method calls between objects of the proposed classes.
- Consider adding a customer
  - Input name and details
  - Create customer object
  - Store object somewhere

Fill in details

- Input name and details
  - Need an input method.
- Create customer object
  - What are customer attributes?
    - Refer back to requirements/use cases: first name, last name, address, phone, email
- Store object somewhere
  - Need a data structure
  - Could use Customer[] or ArrayList<Customer>
    - Don't know how many customers there will be, so ArrayList is the obvious choice.

Allocate methods

- How is behaviour split between methods?
- Which class does each method belong to?

Try:

- Input method in class SimpleOrderSystem.
- Store ArrayList<Customer> as instance variable in same class.
- Constructor for class Customer.

What if this is wrong? Backtrack and learn from experience.

Adding an order

- Steps
  - Find customer (can only create order for existing customer)
  - Create Order object
    - For each item added to an order
      - Select Product (can only order products that exist)
      - Create a LineItem object given product and quantity
      - Add LineItem to Order
  - Calculate total price and confirm order
    - LineItem can calculate cost of n items.
    - Order can calculate cost of all LineItems
  - Add Order to customer
    - Implies customer has list of orders.

Allocate Methods

Can also use Class-Responsibility-Collaboration (CRC) approach - role play method class between objects.

- Find customer
  - method in class SimpleOrderSystem
- Input order - sequence of product, amount
  - also method in class SimpleOrderSystem
- Create LineItem
  - method in class SimpleOrderSystem
  - addItem method needed for class Order
- Calculate total price
  - Methods in class Order and LineItem
- Order confirmation
  - method in class SimpleOrderSystem
- Add order to customer

Class SimpleOrderSystem seems to be getting quite a few methods. Monitor this for now but may need to take action later.

Wrong Classes?

- What if classes and associations don't match with methods?
  - Missing class
  - Unused class
  - Missing association
    - Remember an object of one class can only call an object of another class if it has a reference (association) to it.
- Modify proposed classes and try again.
- Iterative process.

Wrong code?

- Writing the code validates the design.
- If the code won't fit together, or is awkward,
- then iterate and modify design.
  - Don't be afraid to throw away code.
    - If it's wrong, it's wrong.
- Don't expect to get design/code right first time (or second time...).

Class Customer (initial version)

```
public class Customer {
  private String firstName;
  private String lastName;
  ...
public Customer(String firstName, String lastName, String address,
                String phone, String email) {
   this.firstName = firstName;
   this.lastName = lastName;
   ...
  }
  public String getFirstName()  {
   return firstName;
  }
  public String getLastName()  {
   return lastName;
  }
 ...
```

Omitted other instance variables and methods due to space.

Constructor method.
Customer c = new Customer("Arthur", "Dent", ...)

Constructor method called automatically when object is created. Parameters *must* be supplied. New object must be initialised using constructor to represent a customer.

If constructor omitted, then instance variables initialised by default to null.
Object would not be initialised to represent a specific customer.

Constructor Method Summary

- Same name as class name.
- Cannot return a value, no return type declaration.
  - Beware public *void* Customer(...) is a valid instance method but not a constructor.
- Is called when an object is created using new.
  - Parameter list must be provided to match declaration.
  - new returns a reference to the new *initialised* object.
- Cannot be called elsewhere, like an instance method.
- Is used to properly initialise a new object to its correct state, before the object is used.

No constructor?

- If no constructor declared, compiler adds a zero parameter constructor automatically:
  - public Customer() { }
  - Only want this for very simple classes.
  - Normally you provide a proper constructor.
- Instance variables initialised to default values
  - int, 0
  - double, 0.0
  - Any class type (e.g., String), null

Instance Variable Initialisation *Idiom*

```
public Customer(String firstName, String lastName, ...)  {
    this.firstName = firstName;
    this.lastName = lastName;
```

- Instance and parameter variables have same names.
- Use this.name, to refer to instance variable.
  - Avoids need to invent additional names.
  - But possibly more open to mistakes/confusion.

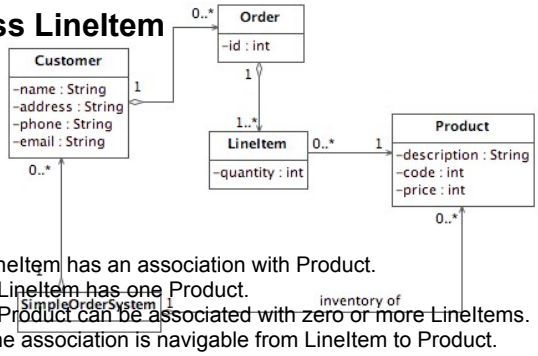
An idiom is a style or convention a programmer uses when writing code.

class Product

```
public class Product {
  private int code;
  private int price;
  private String description;
  public Product(int code, String description, int price)  {
    this.code = code;
    this.price = price;
    this.description = description;
  }
  public int getPrice()  {  return price;  }
  public String getDescription()  { return description;  }
  public int getCode()  { return code; }
}
```

Very straightforward. What if price changes? Will worry about that in a later iteration.

class LineItem



- LineItem has an association with Product.
- A LineItem has one Product.
- A Product can be associated with zero or more LineItems.
- The association is navigable from LineItem to Product.
  - A LineItem knows its Product.
  - A Product is associated with a LineItem but does not know it.
- LineItem has two attributes, quantity and product.
  - The set of attributes of a class is the combination of those shown inside the icon *and* the associations.
  - Class LineItem will need *two* instance variables.

class LineItem (2)

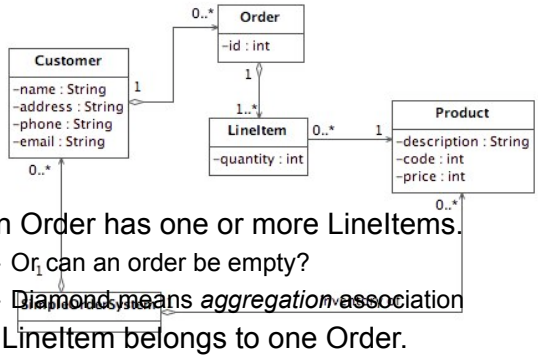
```
public class LineItem {
  private int quantity;
  private Product product;
  public LineItem(int quantity, Product product)  {
    this.quantity = quantity;
    this.product = product;
  }
  public int getQuantity()  { return quantity; }
  public Product getProduct() { return product; }
  public int getSubTotal() {
    return product.getPrice() * quantity;
  }
}
```

A new LineItem needs a Product and quantity. Hence, constructor has two arguments.

class LineItem (3)

- NOTE
  - LineItem is given a Product and quantity.
  - It does *not* create a Product (that happens elsewhere).
  - It does *not* do any input.

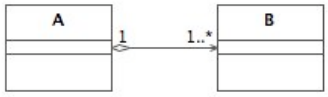
cla



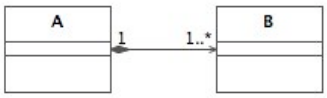
- An Order has one or more LineItems.
  - Or can an order be empty?
  - Diamond means *aggregation* association
- A LineItem belongs to one Order.
- Order has two attributes, id and collection of LineItems (via association).

Aggregation and Composition

- Both show a one to many relationship between 2 classes.
- Object of class A holds collection of references to objects of class B.



Aggregation
Open diamond
Implies A object holds changeable collection of B objects.



Composition
Closed diamond
Implies A object owns and controls lifetime of B objects.

For our purposes, we are not really interested in the distinction.
Can omit diamond entirely.

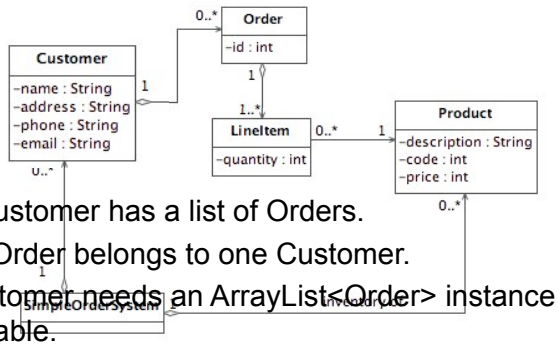
class Order

```
import java.util.ArrayList;
public class Order {
  private ArrayList<LineItem> lineItems;
  public Order()  {
    lineItems = new ArrayList<LineItem>();
  }
  public int getLineItemCount()   { return lineItems.size(); }
  public void add(LineItem item)   { lineItems.add(item); }
  public int getTotal()   {
    int total = 0;
    for (LineItem item : lineItems)  { total += item.getSubTotal(); }
    return total;
  }
}
```


class Order (2)

- NOTE (again)
  - LineItems are added to an Order by passing a reference to an already existing LineItem object.
  - Order does *not* create LineItems (that happens elsewhere).
  - Order does *not* do any input.

Back to class Customer



- A Customer has a list of Orders.
- An Order belongs to one Customer.
- Customer needs an ArrayList<Order> instance variable.

Back to class Customer

- Add methods to add an order, get list of orders and get total for all orders (assume this is in the specification).

```
public void addOrder(Order order)  { orders.add(order); }
public ArrayList<Order> getOrders() {
  return new ArrayList<Order>(orders);
}
public int getTotalForAllOrders()  {
  int total = 0;
  for (Order order : orders)  {
    total += order.getTotal();
  }
  return total;
}
```

Customer does not create Orders. Or do any input.

Copying

```
return new ArrayList<Order>(orders);
```

- Remember that “return an object” really means “return a reference to an object”.
- Code the reference is returned to can change the object by calling its methods.
- The ArrayList is copied but what about the Orders, LineItems and Products?
  - Don’t copy - rely on programmer using objects properly?
  - Copy ArrayList only?
  - Copy everything?

class SimpleOrderSystem

- Has gained quite a lot of responsibility
  - Does input/output via terminal
  - Implements high level control
    - Display menu
    - Manage adding products, orders and customers
- Have written a basic working version (see listing).
- Serves as a framework in which to use classes.
- BUT
  - Needs a lot of work before being “finished”.

public static final

- Constant values are declared like this:

  public static final int ADD_CUSTOMER = 1;
- static denotes a *class variable*
  - belongs to class, one copy shared by all instance objects
- final denotes the variable cannot be changed by assignment
  - it can be directly initialised (as above)
  - or assigned to once if not directly initialised
- public denotes that the variable can be accessed from other classes using: SimpleOrderSystem.ADD_CUSTOMER

Using null

```
private Product getProduct(int code)  {
  for (Product product : products) {
    if (product.getCode() == code) { return product;  }
  }
  return null;
}
```

- Either finds Product object and returns a reference to it,
- Or, returns null if no Product found with given code.
- Hence, code calling getProduct (the *client code*), must check to see if null is returned.

Using null (2)

```
Product product = getProduct(code);
if (product == null) { ... }
```

- null can be compared using == and !=
- If reference == null then no Product object found.
  - Trying to call method will cause NullPointerException.
- If reference != null, Product object found and methods can be called on it.
- The getProduct method has a “contract of use” that must be followed.

Exercises 2

- Take SimpleOrderSystem code and work with it.
- Add/modify methods.
- Find bugs!
  - Don't assume the code is correct.
  - Read through it and understand how it works.
- Use BlueJ.

Compiling the .java files

- Each class is stored in a *separate* .java file.
- The .java files are located in the working directory.
- Use javac *.java to compile, or can compile individually.
  - In fact, the compiler will automatically compile dependent .java files, i.e., if class A uses class B, then compiling A.java will also compile B.java, if no B.class.
- BUT
  - It would be much better to use BlueJ.
  - Here is a demo...

Summary

- Looked at the basic process of designing and writing a small program.
- Simple classes in Java.
- Lots of details.
  - Still many design decisions to be resolved before program is finished.