UCL

COMP1008
Advice for the mini-project

UCL

Think Object-Oriented

- Identify classes and objects.
  - Objects encapsulate state or a representation.
  - And provide services or methods.
  - Classes describe how objects are implemented.
  - Classes are related by associations and inheritance.
  - Objects are linked by references, based on the associations.
- Tasks are performed by objects calling each others methods.

UCL

An example

- The London Underground Problem
  - Write a program that will find a route between any two stations on the London Underground network.

UCL

Getting Started

- Brainstorm!!!
- We need:
  - Data structure to store a representation of the underground.
  - An algorithm to find a route using the data structure.
- AND we want to take an object-oriented point of view.

UCL

Algorithm Ideas

- Try finding a route on an underground map:
  - Locate start station.
  - Follow line in one direction.
  - Do we go past end station?
  - If yes, then done, otherwise back-track to start and go in the other direction.
  - Can't find station on a line? Then change to a different line.
- People searching for a route will take short cuts – the program will need to do things step-by-step.

UCL

Algorithm Ideas (2)

- What if the station is on a different line?
  - Do a recursive search onto the new line.
    - At each station, if it is not the destination,
      - Pick each line in turn, search that line in both directions.

Algorithm Ideas (3)

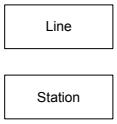
- Wait, can't this be done using graphs or adjacency matrices or something?
- Yes, and would be more efficient than recursive (depth-first) search.
- Then research the alternatives!
  - We will carry on with recursive search for now.
  - Because I want to talk about classes and objects.

Classes - Role Play (CRC method)

- Identify some initial classes/objects to work with:
  - Station, Line, Network
- Pick a task:
  - Find a route? – Too complicated to start with!
  - Try something simpler: What Line is a Station on.

Role Play (2)

- Walk through a scenario:
  - Get a station object. How?
  - Ask it what line it is on.
    Call a getLine method?
    - Need Line objects.
- Implications:
  - A station knows what line it is on. An instance variable is needed.
  - We need a collection of all stations to get a station from.



Role Play (3)

- Review
  - Realise a station can be on more that one line.
    - Needs a variable to store a collection of lines.
  - If we have a collection of stations, how is a given station identified?
    - By its name!
    - A station needs to store its name in a variable.
- (Note, the need for a name may be obvious but we should only add it once the need has been established.)

Station Class – 1st go

```
class Station
{
    private String name;
    private ArrayList<Line> lines;

    public ArrayList<Line> getLines()
    { return lines; }

    public String getName()
    { return name; }
}
```


Next Scenario

- What is the next station on the line?
  - In fact, what is a line?
    - A sequence of stations,
    - and a line has a name.
  - Could provide a getNextStation method?

Line

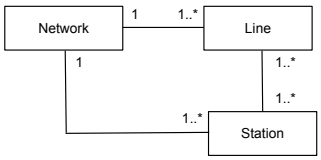
```
class Line
{
  private ArrayList stations;
  private String name;

  public Station getNextStation(String currentStation,
                                boolean forward)
 { … }
}
```

Returns null if no station?

The Data Structure

- Starting to look like:
  - A collection of lines,
  - where each line is a sequence of stations,
  - and each station has a collection of lines it is on.
  - Also want a collection of stations to make it easy to find a station to start with.

Data Structure



Searching

- Could Line implement searching?
  - find (Station destination) ;
- Yes – the recursive algorithm could be implemented quite easily.
- Searching would then take place by the Line and Station objects calling each others methods.
- But would a separate RouteFinder class work better?

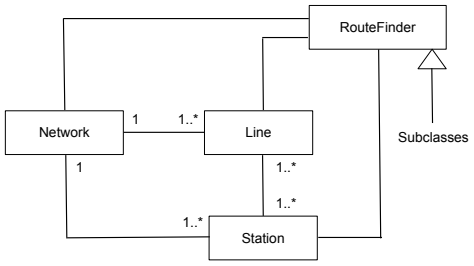
RouteFinder?

- An alternative is to encapsulate the algorithm in an object.
- Advantages:
  - We can change algorithm by using different route finder objects.
  - The algorithm can be implemented independently of how lines and stations are implemented.

RouteFinder (2)

- Disadvantages:
  - The encapsulation of Line and Station may need to be reduced to allow the RouteFinder access.
    - However, there are more advanced ways of structuring the program to avoid the problem.
    - And it may not be a dig deal anyway.
- On balance a RouteFinder is a better solution but requires more sophistication.

Data Structure



Progressing

- Now have enough to start coding a working prototype.
- New issues will be encountered:
  - How are all the lines and stations created and initialised?
  - Input/Output
- But a feasible solution is emerging.

Progressing (2)

- Prototyping involves experimenting.
- Some things won't work – throw them out!
- You'll get new ideas and perhaps find better solutions – use them!
- Don't let things get messy – spend time cleaning up and throwing out.
- Test your code!

Class Checklist

- What instance variables, what types?
  - Are they all private?
- Constructors and initialisation
  - How is an object initialised?
- What public methods?
  - What services do objects provide?

Review Classes

- What can be eliminated to keep the class as simple as possible but no simpler?
- What have we discovered that may matter in the future?
- Are the public methods reducing encapsulation unnecessarily?

Other Advice

- Use nouns for class names
  - Verb indicates that your class has no instance variables or is just a collection of methods.
  - InterestCalculator not CalculatingInterest
- Don't use plural class names
  - BookCollection not Books
- Keep a class cohesive
  - Focus on doing/representing closely related things.
  - Otherwise split into 2 classes.

Other Advice (2)

- If several classes have just one method
  - You are probably writing a procedural program.
  - Just wrapping methods in a classes.
  - Rethink design.
- Keep methods short and cohesive.
- Instance variables always private (unless static final).

Summary

- Think Object-Oriented!
- Had a taste of the thinking and designing process.
- All the time you are searching for viable solutions and balancing the conflicting issues.
- Simplify, Simplify, Simplify.
- BUT Simple != Trivial