Software Engineering and Testing

Session 4

Lecturer: Dr. Simon McLoughlin

Discovering Objects and Classes Revisited

Remember from last week:

- Examine each use case in turn
- Identify the objects that collaborate to achieve the use case functionality
- Describe the objects attributes and their responsibilities
- Repeat for other use cases and build overall analysis model iteratively
- Candidate objects and object attributes are revealed by examining the nouns (things) in the use case description
- Candidate object behaviours may be revealed by examining the verbs or the actions objects do in the use case description.
- These actions or verbs may also reveal object associations, e.g. the customer lodges cheque to their account, the action " lodging a cheque" reveals an association between Customer and Account

Promotion and Demotion of Classes

- As you iterate through the use case descriptions you may find that some "things" do not merit a full class description
- Perhaps they have no significant behaviour or are very simple types
- You may reveal synonyms
- You may find that some objects identified in one use case are actually attributes of another object in a different use case
- Attributes may need to be promoted to a class of their own if they are complex enough
- Other classes may be added to model also as you iterate through the use cases
- You may find that hidden classes or helper classes emerge from associations.

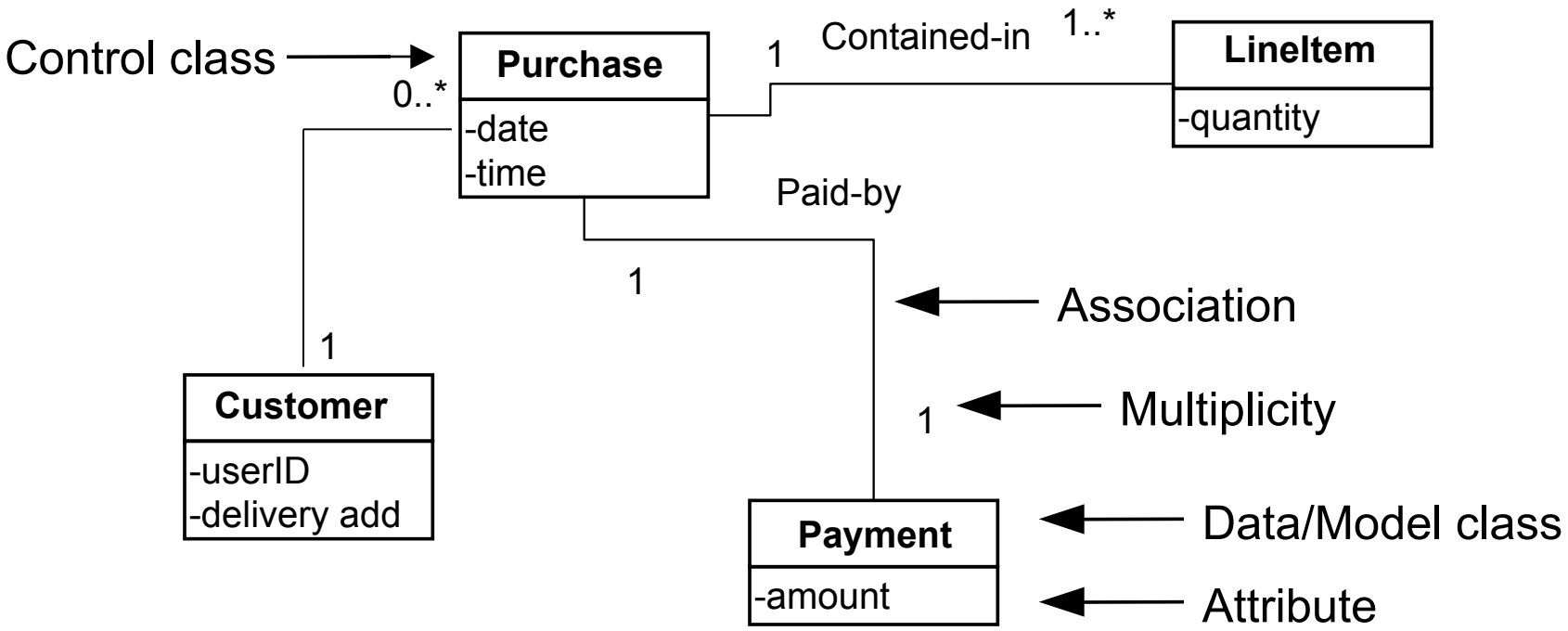
Types of classes

- It sometimes helps to place candidate classes into categories:
- Data classes:
  - Primarily responsible for storing data.
  - Often represent business concepts
  - Often participate in several use cases
- Control classes:
  - Manage interacting objects within the system and usually control a single use case (these may be actually be derived from actions so be careful, an action can be a behaviour, an association or a class)
- Interface classes: handle interaction with actors outside the system may represent physical devices or logical I/O

- There are also times when it makes sense to combine the data class and the control class but it depends on the application and the preference of the analyst
- Remember that 100 system analysts in a room working indepently on the same problem may not produce the exact same solution

Conceptual class diagram

- This can be a good place to start for your class diagrams.
- The conceptual class diagram will show classes with their attributes and basic associations.
- There is no mention of behaviours and the associations are basic without worrying about aggregation etc.



Analogy to ERD

ERD has	*Class Diagram has*
Entities	Classes
Relationships	Associations
Cardinality	Multiplicity
Data elements or attributes	Attributes

Although there are clearly similarities remember they are not the same. Classes exist for data that will be stored temporarily in memory to perform some functionality.

Identifying classes

- Here is an example of a use case describing a withdraw money transaction in a banking system:

Withdraw Money Use Case

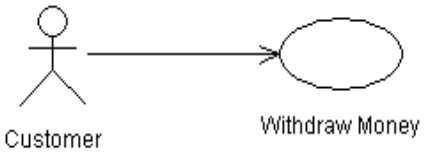
Simplified sequence of actions:

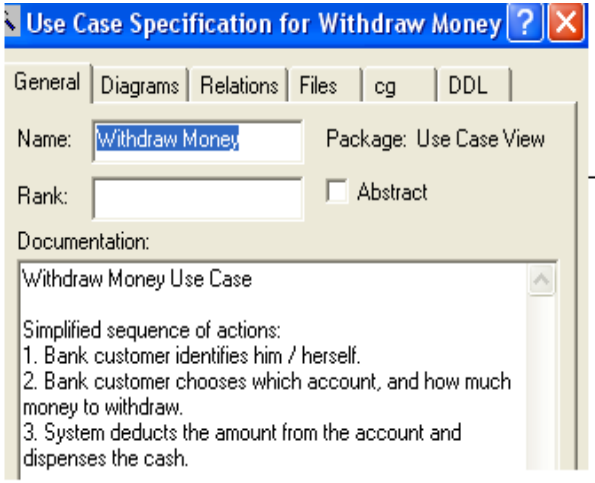
1. Bank customer identifies him / herself.
2. Bank customer chooses which account, and how much money to withdraw.
3. System deducts the amount from the account and dispenses the cash.

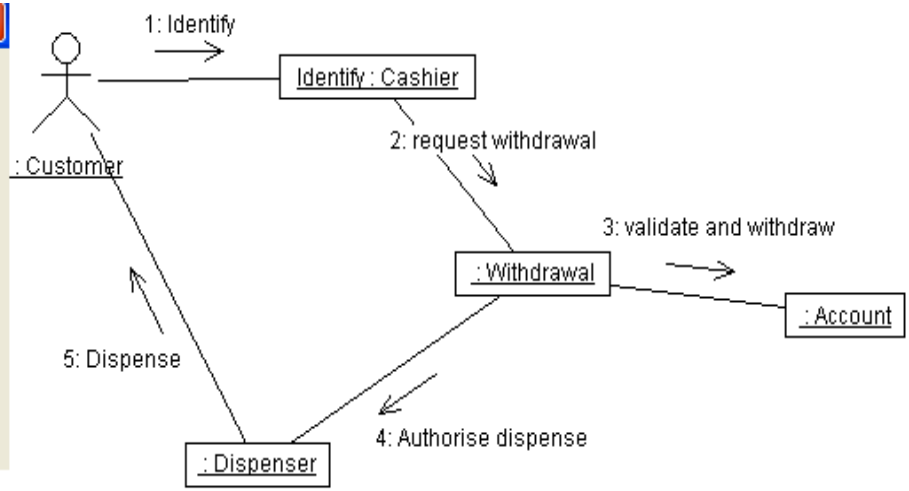
Identifying classes

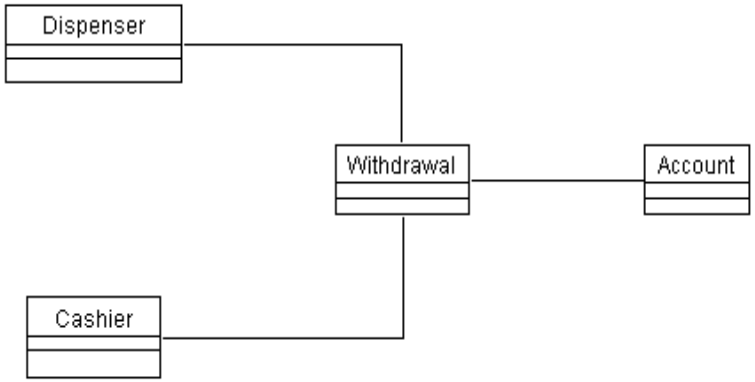
- A number of classes can be identified from the previous use case.
- An Account class is an easy one, this is obviously a data or passive class. You could argue the same for the Customer, although customer details could be part of the account class.
- There is also some interface classes, a Cashier interface where the customer identifies him/herself and a Dispenser interface that dispenses the money.
- Lastly we can introduce the idea of a control class in the shape of a Withdrawal class. This class could control the interaction of the objects in the use case.
- You could also argue that the control could be handled within the data class, i.e. the account class in this example and this approach is also sometimes taken.

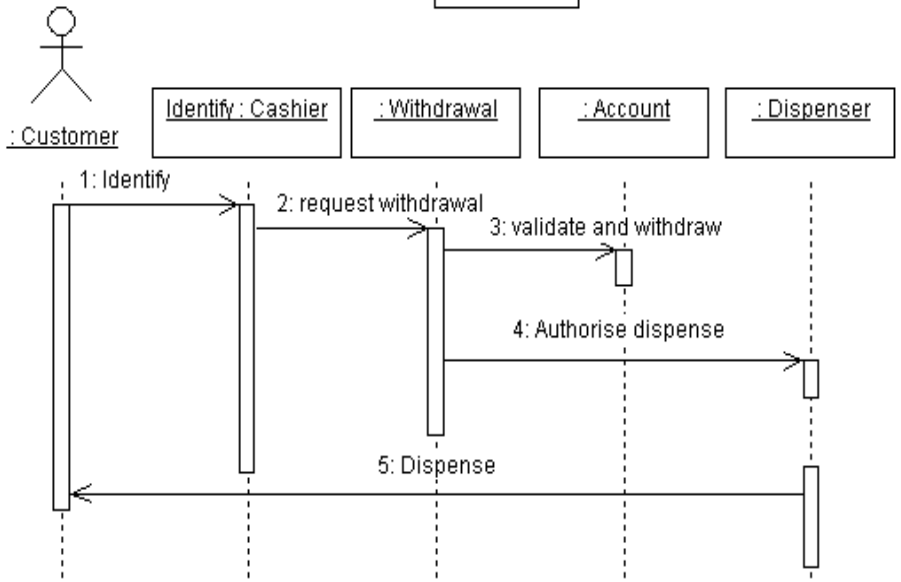
Withdraw Money Example











More on Class Relationships

- The class relationships that we mentioned last week (association, aggregation and generalisation) can be difficult to understand when seeing them for the first time.
- Lets revisit each and see how each is actually represented in Java code.
- Note that relationships like association and aggregation may be conceptually different from one another but in practice they may be implemented exactly the same.
- For this reason some system analysts do not distinguish between them in the model.

Association

- Association is the ability of one class instance to send a message to another class instance. This can actually happen in a number of ways in Java.

```
class RoadVehicle {
    private Passenger [] occupants; //passenger array
    …..//association where Passenger is a Vehicle instance variable

    public void refuel(Fuel fuel) {
    …..//as association where a Fuel object is a method Parameter
    }

    public Label getSpeedasLabel() { //returns car speed
            Label b = new Label();
            b.setText(Integer.toString(speed));
            …..// 2 associations here, Label as local variable and
            …..// Integer as a static method call
}
```

- You can also have self referential associations like in a linked list

```
Class Node {
    private Node next;
    …..// association between node class with itself
```

Aggregation

- Aggregation is like a subset of association where a part-whole relationship exists. For example a car is made up of wheels, an engine, a body…. and so on. When mapped to Java:

```
class RoadVehicle {
    private Wheel [] wheels; //wheel array
    private Engine eng;
    private Body bod;
```

- Note that the implementation of aggregation is the very same as one of the association cases, i.e. we have objects that are class instance variables

- The only difference is that instances cannot have cyclic aggregation relationships (i.e. a part cannot contain its whole).

Generalisation

- The generalisation relationship is one where one object is a generic version of others. For example RoadVehicle is a generic form of Car, Truck, Motobike etc.
- This is the inheritance or the "is-a" relationship that you are familiar with.
- In Java this is implemented using the keyword "extends".

```
class Car extends RoadVehicle {……..}
```

- Using interfaces in Java is another representation of the generalisation relationship (one class implements an interface)

```
class MyGUI extends YourGUI implements ActionListener {……}
```

User Interface Design

- Practically all of you applications will have a user interface and this part is really important as it defines the users perception of the software.
- An application is only as good or bad as its user interface

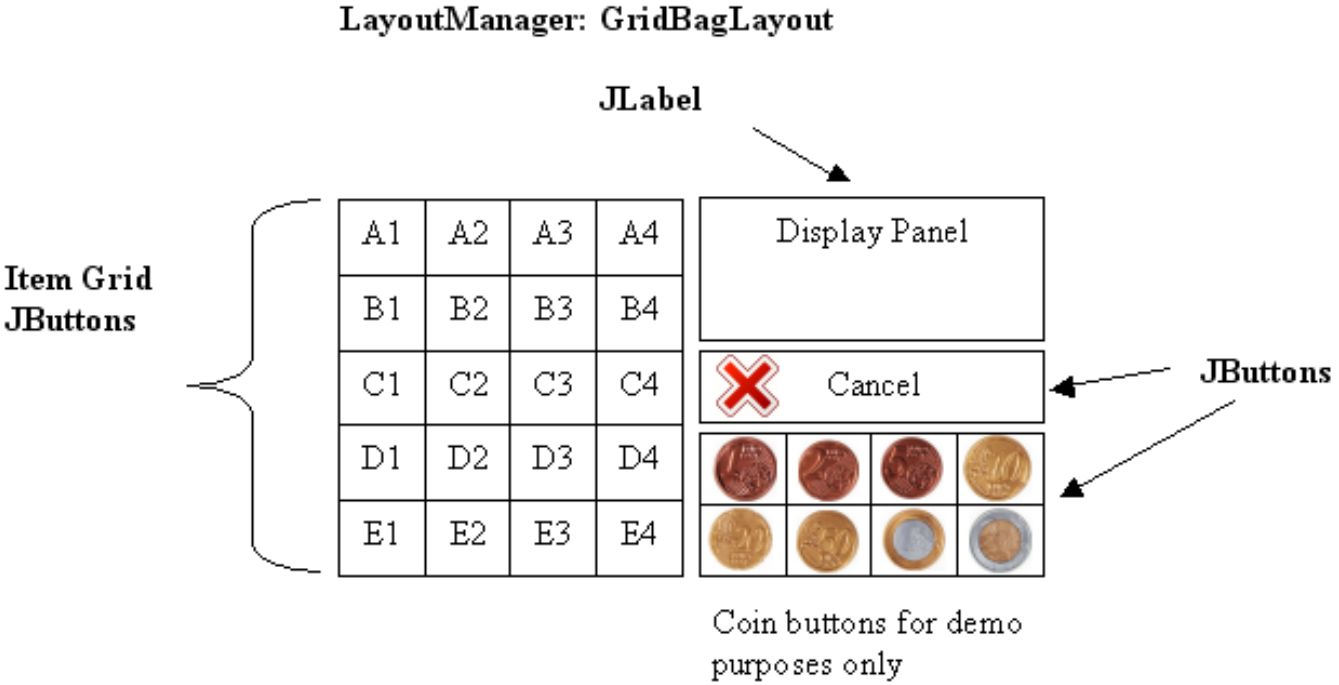
- Bear in mind the following principles when designing the GUI

  - Be consistent: Choose a look and feel and stick to it, this goes for colour schemes, fonts, widget placement, user interaction etc.
  - If your application involves navigation between different screens then the screen sequence should match the users business practice.
  - Word messages and labels effectively, use phrases or sentences as opposed to codes or abbreviations. Make instructions positive.
  - Use colour sparingly and remember that some users may be colour blind, also if using colours you need to use contrast appropriately, e.g. dark text on a bright background.

User Interface Design

- Align fields properly – your user interfaces should have textfields and labels aligned properly for a symmetrical finish.
- Expect users to make mistakes: your interface should enable the user to recover from mistakes e.g. through the use of an “undo’
- Don’t create busy user interfaces as this confuses the user. If there is too much going on one one screen, break it up into multiple screens.
- Group things effectively - if things are related or part of the same group well then they should be placed together. If not then they should be seperated.
- Understand user interface widgets. Always choose th right widget for the task and identify which widgets will be used during design and not implementation.
- Identify the layout you are going to use and the layout manager if one exists to support it.

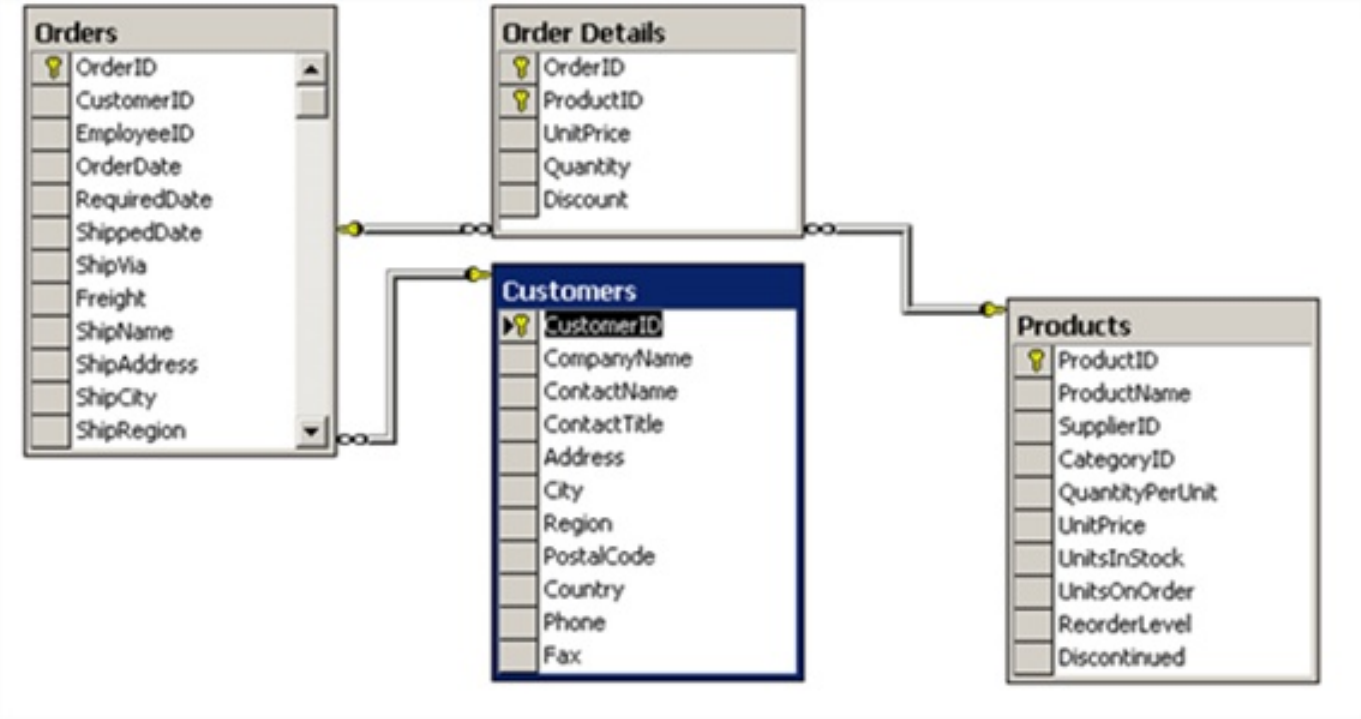
User Interface Design: Vending Machine



Display Panel: This will display messages to the customer such as:
Use correct change
The price of a selected item
Out of Order
Collect Change
Item Grid: this is the selection panel for the customer to select an item based on its code which will be displayed beside the item
Cancel Button: This will allow the customer to cancel the transaction at any time before vending
Coin Buttons: These are for demonstration purposes only and will simulate the coin mechanism
GridBagLayout: This is the layout manager chosen which gives maximum freedom in terms of widget placement in a row-column format.

Database Design

- We will not go into too much detail on database design as you have seen this already in database modules.
- Essentially what I will be looking for is the database schema
- The schema is normally represented as a diagram and shows all tables, all fields within a table and also the relationship between tables (cardinality) and fields (primary keys, foreign keys etc.).



Database Schema for Vending Machine

- Why Bother? For paedological reasons only ☺

Algorithm Design

- There may be some operations that your application has to do that may be relatively complex in terms of coding, is there any searching or sorting for example? If so which algorithm will be used.
- There is obviously classical algorithms for doing searching and sorting but your application may have its own unique operations that are complex and need to be thought out before coding.
- Consider for example for the vending machine operation "returnchange()". How will this be implemented? What coin denominations will be returned will be returned to the user?
- You have some options when designing algotithms: e.g. program design languages (like pseudocode or tight english) or flowcharts, formal methods, stepwise refinement etc.
- All class methods should be examined and summarised in design doc and if complex the algorithm should be designed in detail.

Algorithm Design – return change

```
returnChange(amount)

coin_array = {200, 100, 50, 20, 10, 5, 2, 1}
index = 0
//Begin with highest denomination (2 euro)
coin = coin_array[index]
while (amount > 0)
        if amount > coin
                if coin available (involves coin mechanism communication)
                        dispense coin (involves coin mechanism communication)
                        amount = amount – coin
                end if
        else
                index = index +1
                coin = coin_array[index]
        end if
end while
```

Trace for 72 cent

Pass	index	Coin	amount
1	0->1	200->100	72
2	1->2	100->50	72
3	2	50	72->22
4	2->3	50->20	22
5	3	20	22->2
6	3->4	20->10	2
7	4->5	10->5	2
8	5->6	5->2	2
9	6	2	0
10	Finished		

The analysis document (25%)

- Short Project Description (From feasibilty Study)
- UML Use case diagrams
- Use case specifications (as ledgible screen dumps, typed listings or activity diagrams)
- List of identified object/classes along with their attributes and behaviours
- UML Class diagram showing classes, attributes, methods, associations
- UML interaction diagram for each use case (sequence or collaboration)
- System deployment diagram for distributed applications
- User interface design
- Database schema
- Algorithm designs

Will grant extension if so required until Friday the 7th March