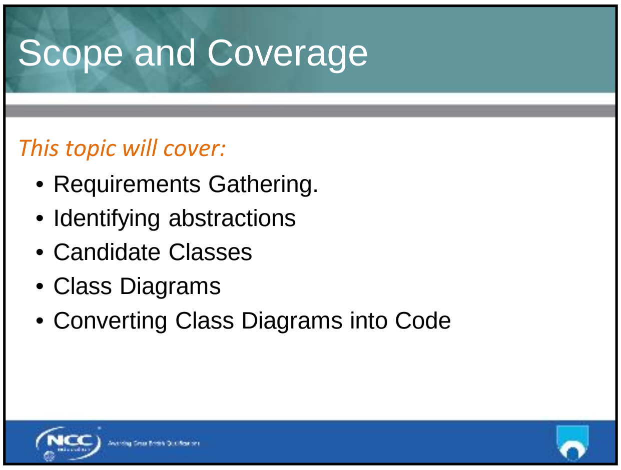
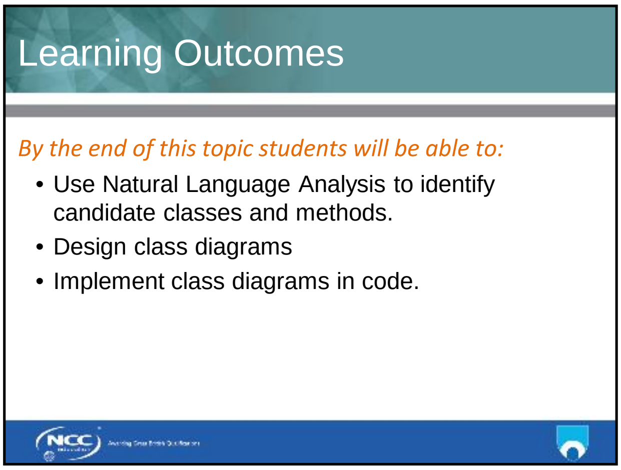
Analysis, Design and Implementation
Topic 5:
Static Analysis and Design



Scope and Coverage

This topic will cover:

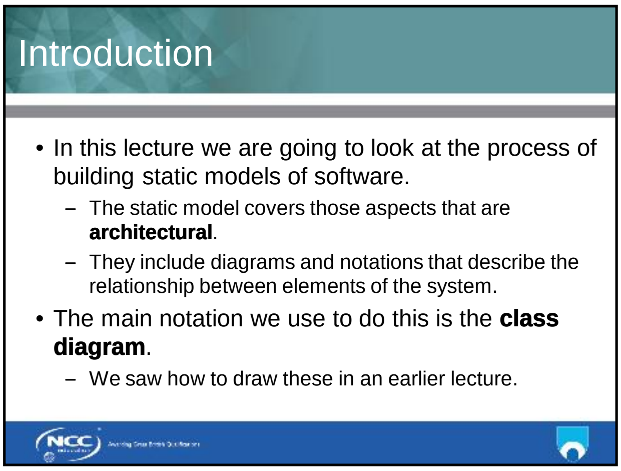
- Requirements Gathering.
- Identifying abstractions
- Candidate Classes
- Class Diagrams
- Converting Class Diagrams into Code



Learning Outcomes

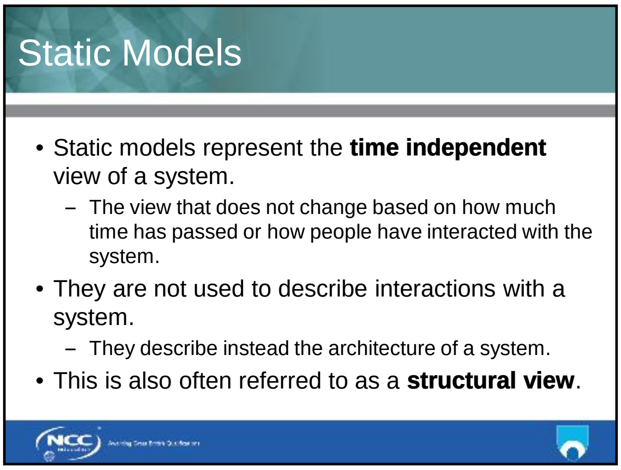
By the end of this topic students will be able to:

- Use Natural Language Analysis to identify candidate classes and methods.
- Design class diagrams
- Implement class diagrams in code.



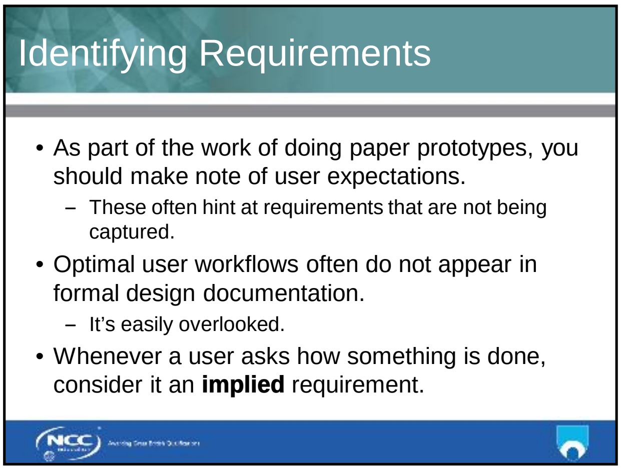
Introduction

- In this lecture we are going to look at the process of building static models of software.
  - The static model covers those aspects that are **architectural**.
  - They include diagrams and notations that describe the relationship between elements of the system.
- The main notation we use to do this is the **class diagram**.
  - We saw how to draw these in an earlier lecture.



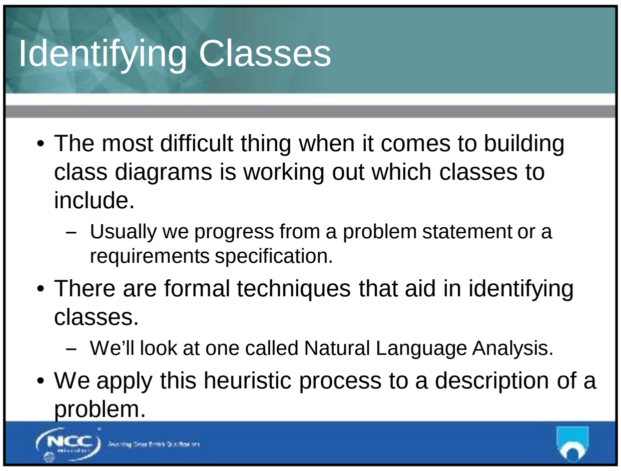
Static Models

- Static models represent the **time independent** view of a system.
  - The view that does not change based on how much time has passed or how people have interacted with the system.
- They are not used to describe interactions with a system.
  - They describe instead the architecture of a system.
- This is also often referred to as a **structural view**.



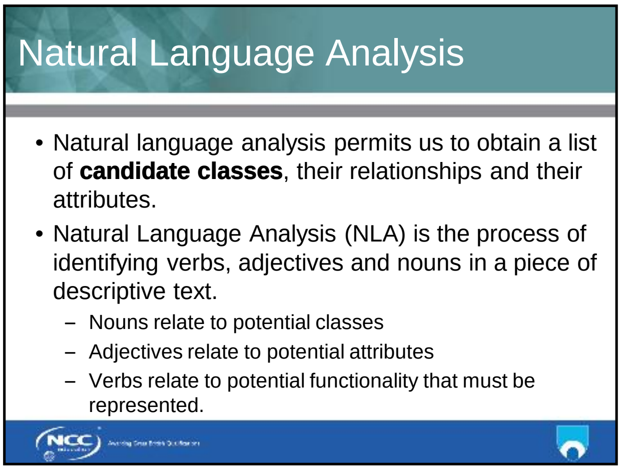
Identifying Requirements

- As part of the work of doing paper prototypes, you should make note of user expectations.
  - These often hint at requirements that are not being captured.
- Optimal user workflows often do not appear in formal design documentation.
  - It's easily overlooked.
- Whenever a user asks how something is done, consider it an **implied** requirement.



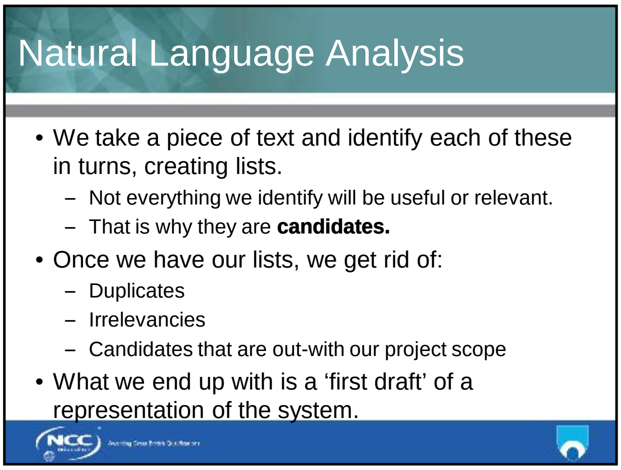
Identifying Classes

- The most difficult thing when it comes to building class diagrams is working out which classes to include.
  - Usually we progress from a problem statement or a requirements specification.
- There are formal techniques that aid in identifying classes.
  - We’ll look at one called Natural Language Analysis.
- We apply this heuristic process to a description of a problem.



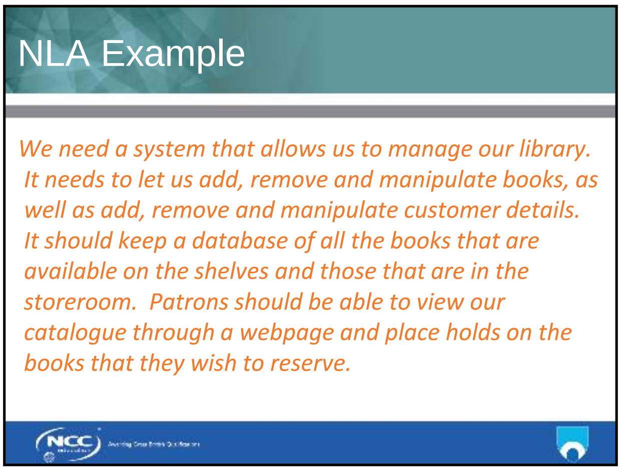
Natural Language Analysis

- Natural language analysis permits us to obtain a list of **candidate classes**, their relationships and their attributes.
- Natural Language Analysis (NLA) is the process of identifying verbs, adjectives and nouns in a piece of descriptive text.
  - Nouns relate to potential classes
  - Adjectives relate to potential attributes
  - Verbs relate to potential functionality that must be represented.



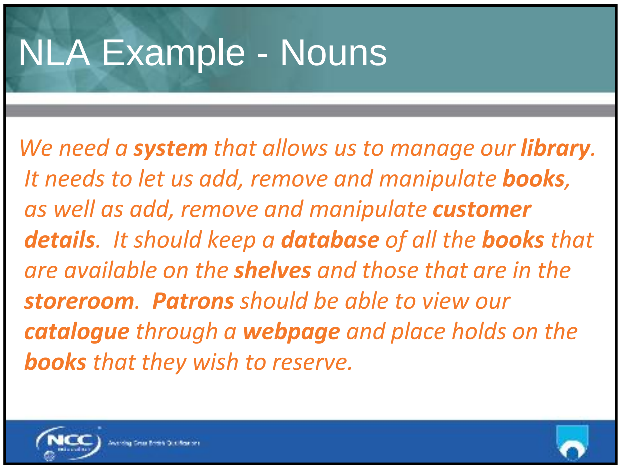
Natural Language Analysis

- We take a piece of text and identify each of these in turns, creating lists.
  - Not everything we identify will be useful or relevant.
  - That is why they are **candidates.**
- Once we have our lists, we get rid of:
  - Duplicates
  - Irrelevancies
  - Candidates that are out-with our project scope
- What we end up with is a ‘first draft’ of a representation of the system.



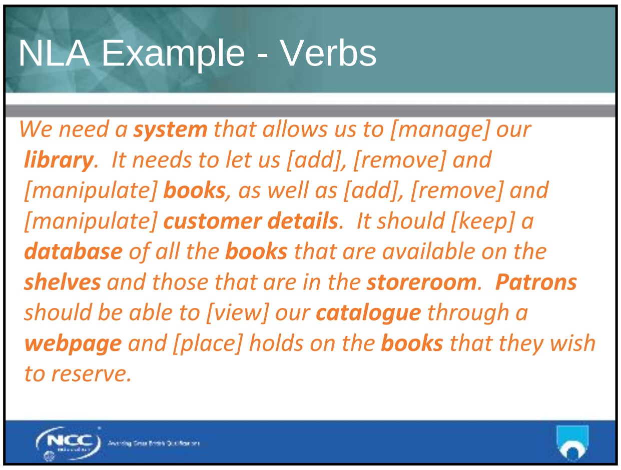
NLA Example

We need a system that allows us to manage our library. It needs to let us add, remove and manipulate books, as well as add, remove and manipulate customer details. It should keep a database of all the books that are available on the shelves and those that are in the storeroom. Patrons should be able to view our catalogue through a webpage and place holds on the books that they wish to reserve.



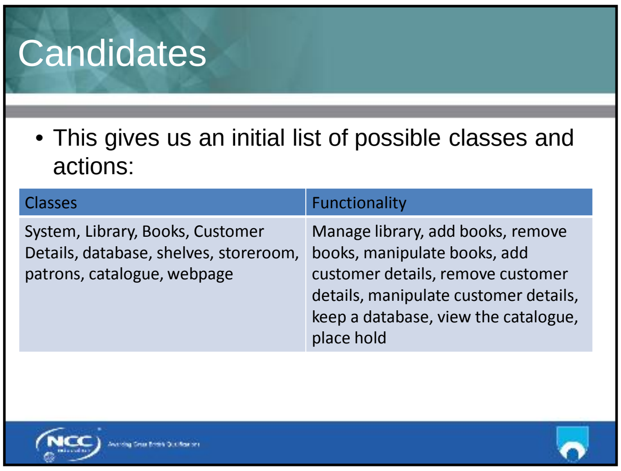
NLA Example - Nouns

We need a ***system*** *that allows us to manage our* ***library****. It needs to let us add, remove and manipulate* ***books****, as well as add, remove and manipulate* ***customer details****. It should keep a* ***database*** *of all the* ***books*** *that are available on the* ***shelves*** *and those that are in the* ***storeroom****.* ***Patrons*** *should be able to view our* ***catalogue*** *through a* ***webpage*** *and place holds on the* ***books*** *that they wish to reserve.*



NLA Example - Verbs

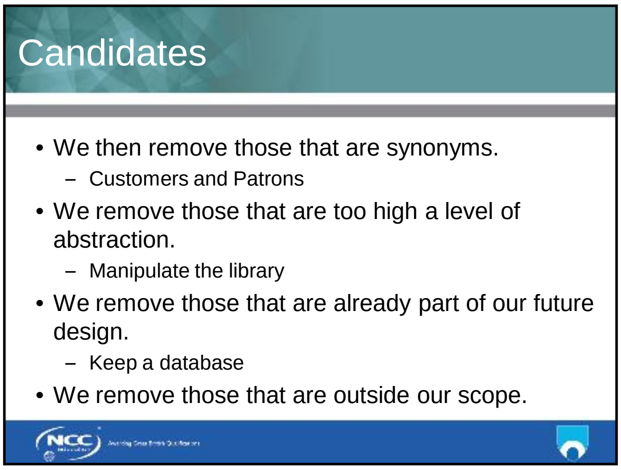
We need a ***system*** *that allows us to [manage] our* ***library****. It needs to let us [add], [remove] and [manipulate]* ***books****, as well as [add], [remove] and [manipulate]* ***customer details****. It should [keep] a* ***database*** *of all the* ***books*** *that are available on the* ***shelves*** *and those that are in the* ***storeroom****.* ***Patrons*** *should be able to [view] our* ***catalogue*** *through a* ***webpage*** *and [place] holds on the* ***books*** *that they wish to reserve.*



Candidates

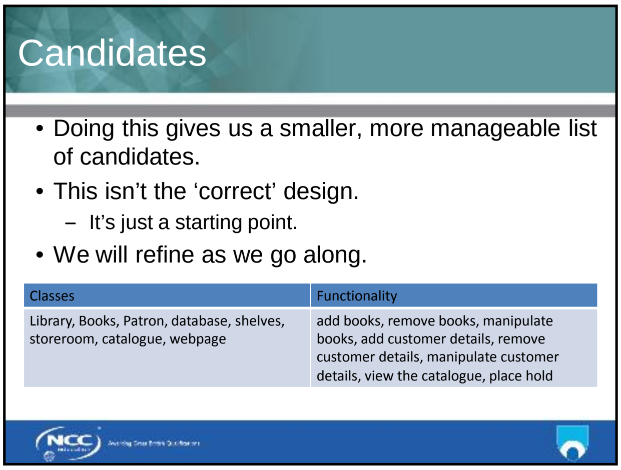
- This gives us an initial list of possible classes and actions:

Classes	Functionality
System, Library, Books, Customer Details, database, shelves, storeroom, patrons, catalogue, webpage	Manage library, add books, remove books, manipulate books, add customer details, remove customer details, manipulate customer details, keep a database, view the catalogue, place hold



Candidates

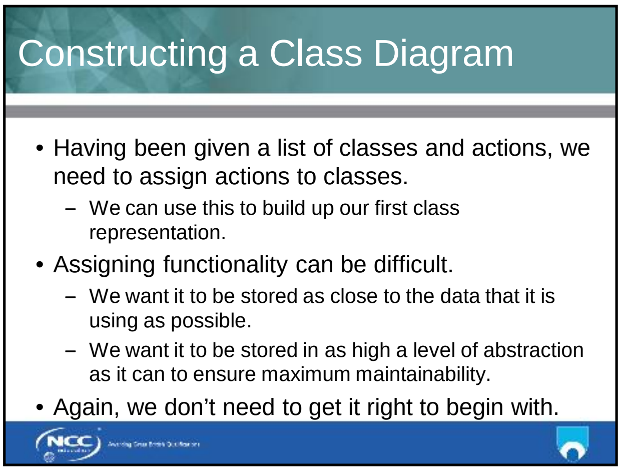
- We then remove those that are synonyms.
  - Customers and Patrons
- We remove those that are too high a level of abstraction.
  - Manipulate the library
- We remove those that are already part of our future design.
  - Keep a database
- We remove those that are outside our scope.



Candidates

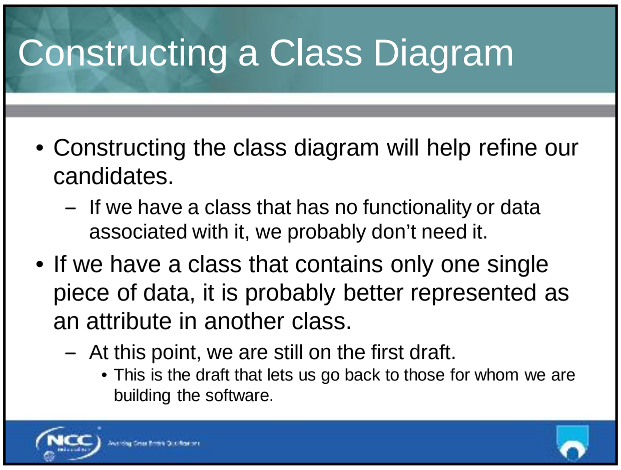
- Doing this gives us a smaller, more manageable list of candidates.
- This isn't the 'correct' design.
  - It's just a starting point.
- We will refine as we go along.

Classes	Functionality
Library, Books, Patron, database, shelves, storeroom, catalogue, webpage	add books, remove books, manipulate books, add customer details, remove customer details, manipulate customer details, view the catalogue, place hold



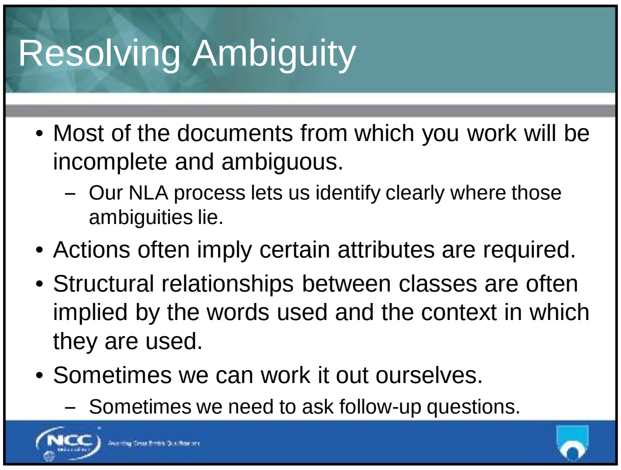
Constructing a Class Diagram

- Having been given a list of classes and actions, we need to assign actions to classes.
  - We can use this to build up our first class representation.
- Assigning functionality can be difficult.
  - We want it to be stored as close to the data that it is using as possible.
  - We want it to be stored in as high a level of abstraction as it can to ensure maximum maintainability.
- Again, we don't need to get it right to begin with.



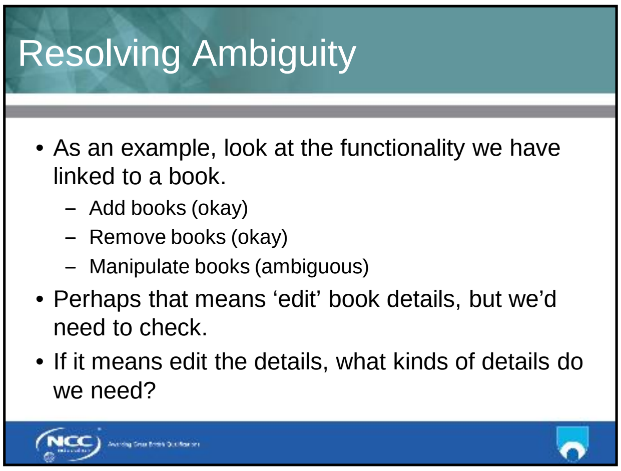
Constructing a Class Diagram

- Constructing the class diagram will help refine our candidates.
  - If we have a class that has no functionality or data associated with it, we probably don't need it.
- If we have a class that contains only one single piece of data, it is probably better represented as an attribute in another class.
  - At this point, we are still on the first draft.
    - This is the draft that lets us go back to those for whom we are building the software.



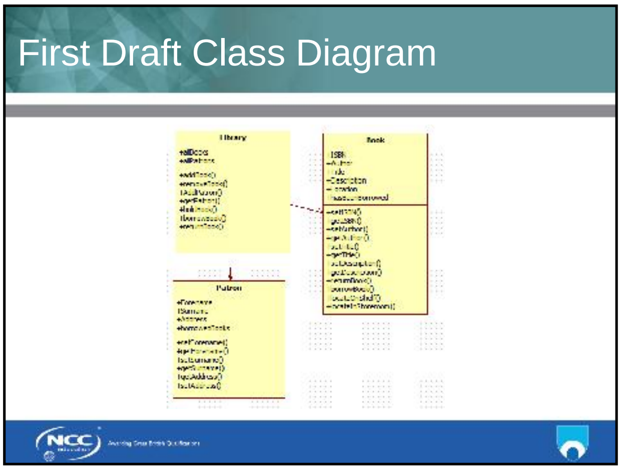
Resolving Ambiguity

- Most of the documents from which you work will be incomplete and ambiguous.
  - Our NLA process lets us identify clearly where those ambiguities lie.
- Actions often imply certain attributes are required.
- Structural relationships between classes are often implied by the words used and the context in which they are used.
- Sometimes we can work it out ourselves.
  - Sometimes we need to ask follow-up questions.

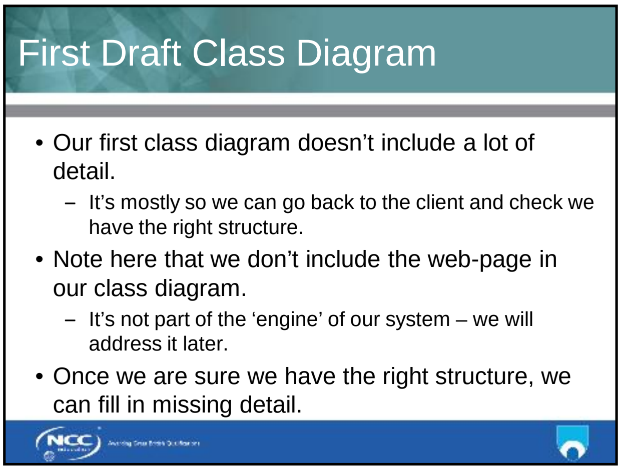


Resolving Ambiguity

- As an example, look at the functionality we have linked to a book.
  - Add books (okay)
  - Remove books (okay)
  - Manipulate books (ambiguous)
- Perhaps that means 'edit' book details, but we'd need to check.
- If it means edit the details, what kinds of details do we need?

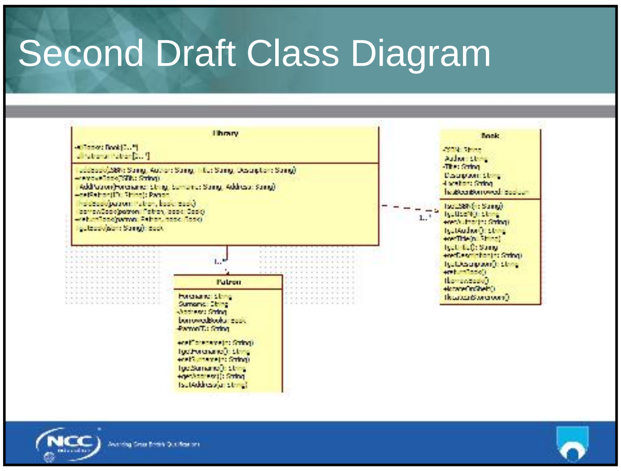


First Draft Class Diagram

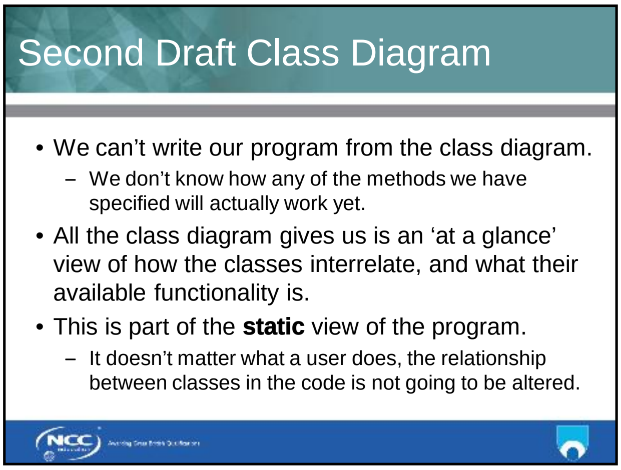


First Draft Class Diagram

- Our first class diagram doesn't include a lot of detail.
  - It's mostly so we can go back to the client and check we have the right structure.
- Note here that we don't include the web-page in our class diagram.
  - It's not part of the 'engine' of our system – we will address it later.
- Once we are sure we have the right structure, we can fill in missing detail.

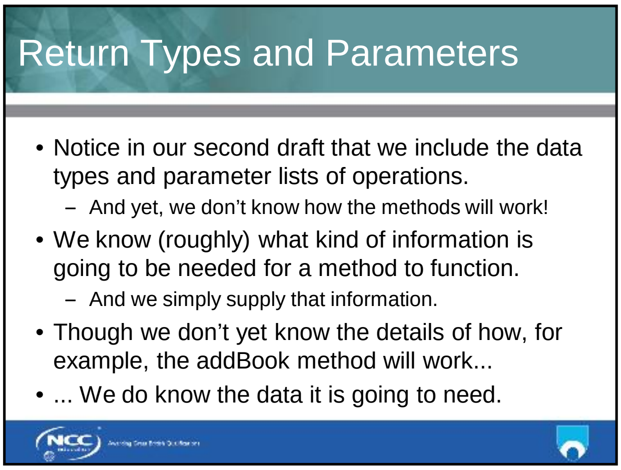


Second Draft Class Diagram



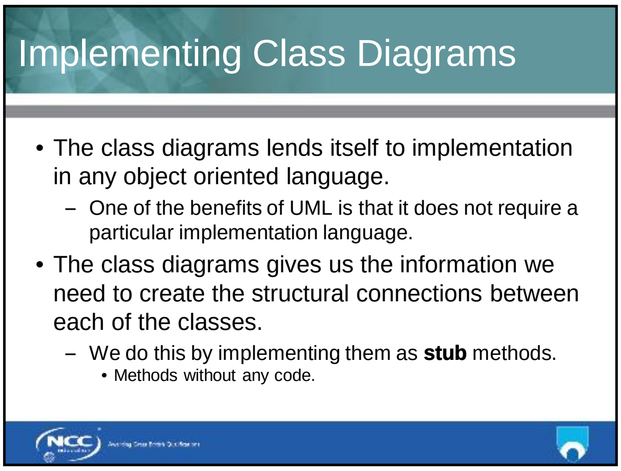
Second Draft Class Diagram

- We can’t write our program from the class diagram.
  - We don’t know how any of the methods we have specified will actually work yet.
- All the class diagram gives us is an ‘at a glance’ view of how the classes interrelate, and what their available functionality is.
- This is part of the **static** view of the program.
  - It doesn’t matter what a user does, the relationship between classes in the code is not going to be altered.



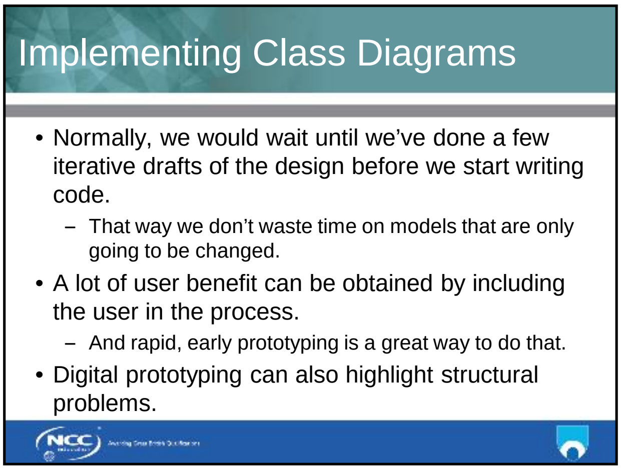
Return Types and Parameters

- Notice in our second draft that we include the data types and parameter lists of operations.
  - And yet, we don’t know how the methods will work!
- We know (roughly) what kind of information is going to be needed for a method to function.
  - And we simply supply that information.
- Though we don’t yet know the details of how, for example, the addBook method will work...
- ... We do know the data it is going to need.



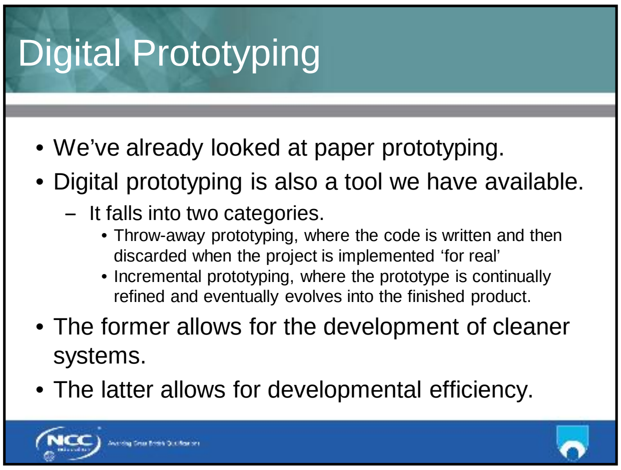
Implementing Class Diagrams

- The class diagrams lends itself to implementation in any object oriented language.
  - One of the benefits of UML is that it does not require a particular implementation language.
- The class diagrams gives us the information we need to create the structural connections between each of the classes.
  - We do this by implementing them as **stub** methods.
    - Methods without any code.



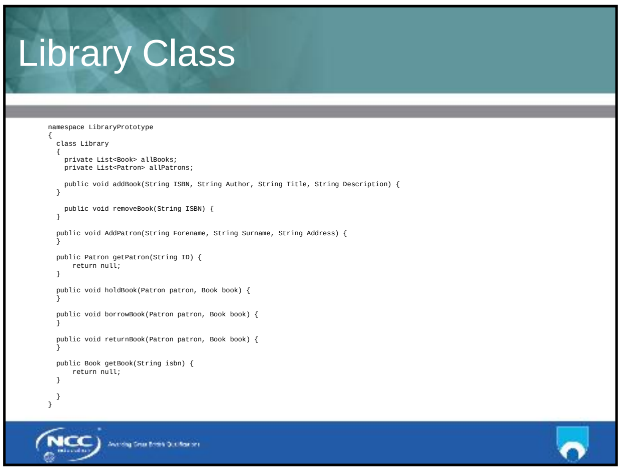
Implementing Class Diagrams

- Normally, we would wait until we've done a few iterative drafts of the design before we start writing code.
  - That way we don't waste time on models that are only going to be changed.
- A lot of user benefit can be obtained by including the user in the process.
  - And rapid, early prototyping is a great way to do that.
- Digital prototyping can also highlight structural problems.



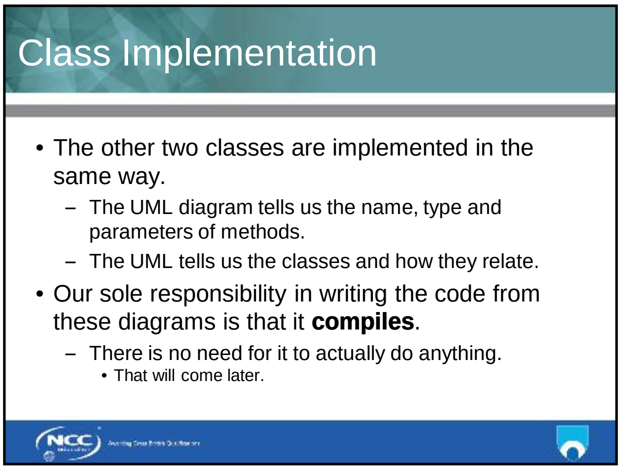
Digital Prototyping

- We've already looked at paper prototyping.
- Digital prototyping is also a tool we have available.
  - It falls into two categories.
    - Throw-away prototyping, where the code is written and then discarded when the project is implemented 'for real'
    - Incremental prototyping, where the prototype is continually refined and eventually evolves into the finished product.
- The former allows for the development of cleaner systems.
- The latter allows for developmental efficiency.



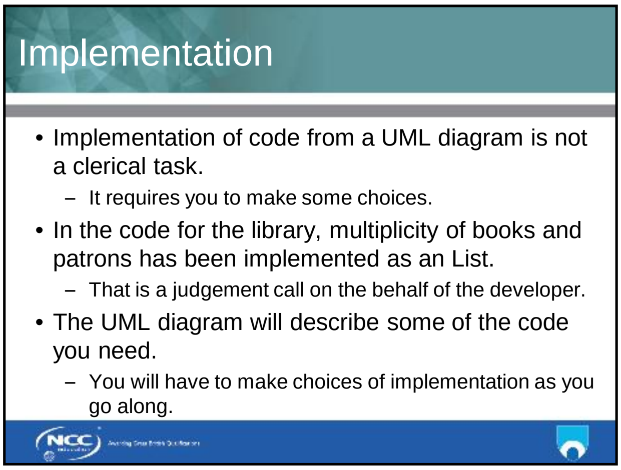
Library Class

```
namespace LibraryPrototype
{
  class Library
  {
    private List<Book> allBooks;
    private List<Patron> allPatrons;

    public void addBook(String ISBN, String Author, String Title, String Description) {
  }

    public void removeBook(String ISBN) {
  }

  public void AddPatron(String Forename, String Surname, String Address) {
  }

  public Patron getPatron(String ID) {
      return null;
  }

  public void holdBook(Patron patron, Book book) {
  }

  public void borrowBook(Patron patron, Book book) {
  }

  public void returnBook(Patron patron, Book book) {
  }

  public Book getBook(String isbn) {
      return null;
  }

  }
}
```



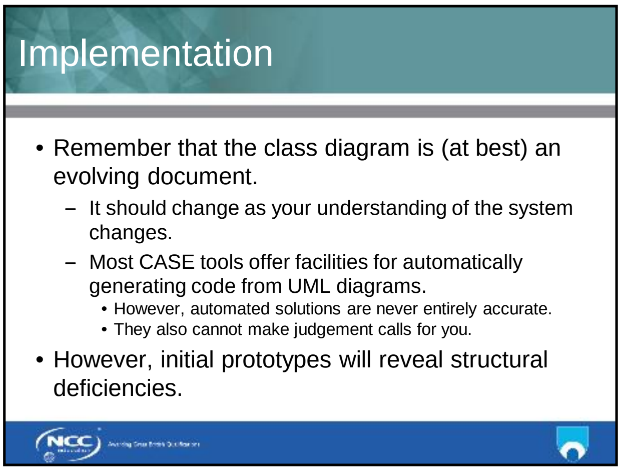
Class Implementation

- The other two classes are implemented in the same way.
  - The UML diagram tells us the name, type and parameters of methods.
  - The UML tells us the classes and how they relate.
- Our sole responsibility in writing the code from these diagrams is that it **compiles**.
  - There is no need for it to actually do anything.
    - That will come later.



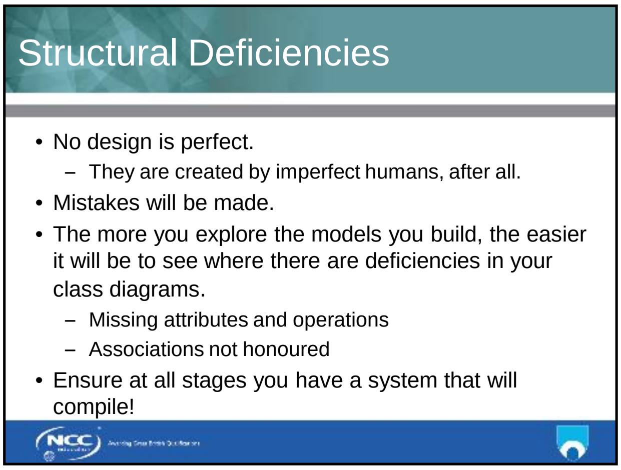
Implementation

- Implementation of code from a UML diagram is not a clerical task.
  - It requires you to make some choices.
- In the code for the library, multiplicity of books and patrons has been implemented as an List.
  - That is a judgement call on the behalf of the developer.
- The UML diagram will describe some of the code you need.
  - You will have to make choices of implementation as you go along.



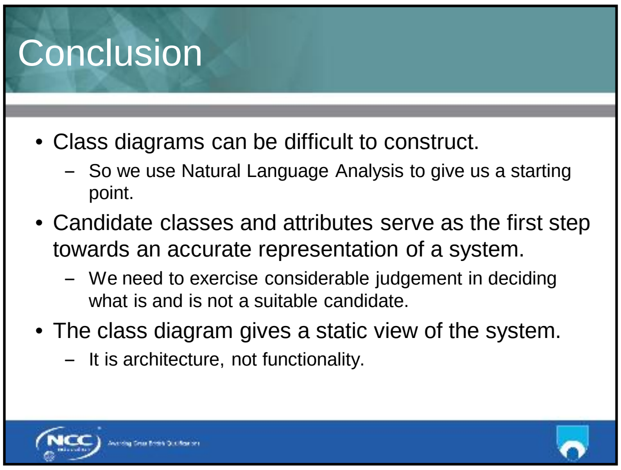
Implementation

- Remember that the class diagram is (at best) an evolving document.
  - It should change as your understanding of the system changes.
  - Most CASE tools offer facilities for automatically generating code from UML diagrams.
    - However, automated solutions are never entirely accurate.
    - They also cannot make judgement calls for you.
- However, initial prototypes will reveal structural deficiencies.



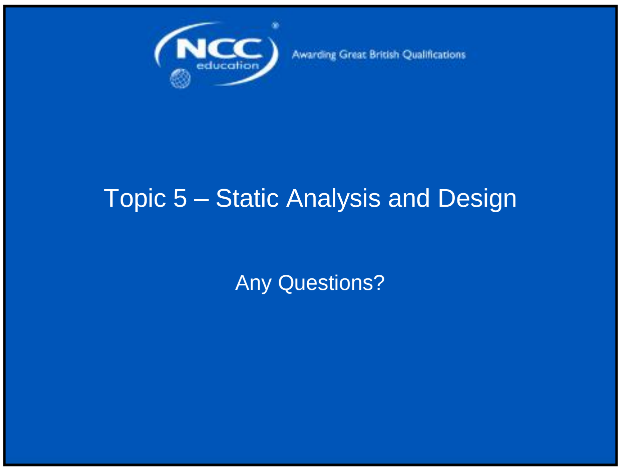
Structural Deficiencies

- No design is perfect.
  - They are created by imperfect humans, after all.
- Mistakes will be made.
- The more you explore the models you build, the easier it will be to see where there are deficiencies in your class diagrams.
  - Missing attributes and operations
  - Associations not honoured
- Ensure at all stages you have a system that will compile!



Conclusion

- Class diagrams can be difficult to construct.
  - So we use Natural Language Analysis to give us a starting point.
- Candidate classes and attributes serve as the first step towards an accurate representation of a system.
  - We need to exercise considerable judgement in deciding what is and is not a suitable candidate.
- The class diagram gives a static view of the system.
  - It is architecture, not functionality.



Awarding Great British Qualifications

Topic 5 – Static Analysis and Design

Any Questions?