Object Oriented Analysis

- **Background**
  - Model the requirements in terms of objects and the services they provide.
  - Grew out of object oriented design.
  - Applied to modelling the application domain rather than the program.

- **Motivation**
  - OO is (claimed to be) more ‘natural’
  - As a system evolves, the functions it performs need to be changed more often than the objects on which they operate.
  - A model based on objects (rather than functions) will be more stable over time.
  - Hence the claim that object-oriented designs are more maintainable.
  - OO emphasizes importance of well-defined interfaces between objects.
  - Compared to ambiguities of dataflow relationships.

**NOTE:** OO applies to requirements engineering because it is a modeling tool. But we are modeling domain objects, not the design of the new system.

Requirements & Domain Models

- **Object Oriented Analysis**
  - **Rationale**
    - Identifying Classes
  - **Attributes and Operations**
  - **Class Diagrams**
    - Associations
    - Multiplicity
    - Composition
    - Generalization

- **Class Diagrams**
  - **Associations**
  - **Multiplicity**
  - **Aggregation**
  - **Composition**
  - **Generalization**

- **Our analysis models should...**
  - Represent people, physical things and concepts important to the analyst’s understanding of what is going on in the application domain.
  - Show connections and interactions among these people, things and relevant concepts.
  - Show the business situation in enough detail to evaluate possible designs.
  - Be organized to be useful later, during design and implementation of the software.
  - Allow us to check whether the functions we will include in the specification will satisfy the requirements.
  - Test our understanding of how the new system will interact with the world.

Nearly anything can be an object...

- **External Entities**
  - That interact with the system being modeled.
  - Examples: people, devices, other systems.

- **Things**
  - That are part of the domain being modeled.
  - Examples: reports, displays, signals, etc.

- **Occurrences or Events**
  - That occur in the context of the system.
  - Examples: transfer of resources, a control action, etc.

- **Roles**
  - Played by people who interact with the system.

- **Organizational Units**
  - That are relevant to the application.
  - Examples: division, group, team, etc.

- **Places**
  - That establish the context of the problem being modeled.
  - Examples: manufacturing floor, loading dock, etc.

- **Structures**
  - That define a class or assembly of objects.
  - Examples: sensors, four-wheeled vehicles, computers, etc.

Some things cannot be objects:

- Procedures (e.g. print, invert, etc)
- Attributes (e.g. blue, 50Mb, etc.)
What are classes?

- A class describes a group of objects with:
  - similar properties (attributes),
  - common behaviour (operations),
  - common relationships to other objects,
  - and common meaning ("semantics").

Examples:
- Employee: has a name, employee#, and department; an employee is hired, and fired; an employee works in one or more projects.

<table>
<thead>
<tr>
<th>Attributes (optional)</th>
<th>Name (mandatory)</th>
<th>Operations (optional)</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>employee#</td>
<td>department</td>
</tr>
<tr>
<td>hire()</td>
<td>fire()</td>
<td>assignproject()</td>
</tr>
</tbody>
</table>

Finding Classes

- Finding classes source data:
  - Look for nouns and noun phrases in stakeholders' descriptions of the problem.
  - Include in the model if they explain the nature or structure of information in the application.

- Finding classes from other sources:
  - Reviewing background information;
  - Users and other stakeholders;
  - Analysis patterns;

It's better to include many candidate classes at first
- You can always eliminate them later if they turn out not to be useful.
- Explicitly deciding to discard classes is better than just not thinking about them.

Selecting Classes

- Discard classes for concepts which:
  - Are beyond the scope of the analysis;
  - Refer to the system as a whole;
  - Duplicate other classes;
  - Are too vague or too specific
    - e.g. have too many or too few instances
  - Coad & Yourdon's criteria:
    - Retained information: Will the system need to remember information about this class of objects?
    - Needed Services: Do objects in this class have identifiable operations that change the values of their attributes?
    - Multiple Attributes: If the class only has one attribute, it may be better represented as an attribute of another class
    - Common Attributes: Does the class have attributes that are shared with all instances of its objects?
    - Common Operations: Does the class have operations that are shared with all instances of its objects?
  - External entities that produce or consume information essential to the system should be included as classes.

Objects vs. Classes

- The instances of a class are called objects.
  - Objects are represented as:
    - Fred_Bloggs:Employee
      - name: Fred Bloggs
      - Employee #: 234609234
      - Department: Marketing

- Two different objects may have identical attribute values (like two people with identical name and address).
- Objects have associations with other objects
  - E.g. Fred_Bloggs:employee is associated with the KillerApp:project object
  - But we will capture these relationships at the class level (why?)
  - Note: Make sure attributes are associated with the right class
    - E.g. you don't want both managerName and manager#: as attributes of Project! (.Why??)
Associations

Objects do not exist in isolation from one another

- A relationship represents a connection among things.
- In UML, there are different types of relationships:
  - Association
  - Aggregation and Composition
  - Generalization
  - Dependency
  - Realization

Note: The last two are not useful during requirements analysis

Class diagrams show classes and their relationships

Association Multiplicity

- Ask questions about the associations:
  - Can a campaign exist without a member of staff to manage it?
    - If yes, then the association is optional at the Staff end - zero or one
  - If a campaign cannot exist without a member of staff to manage it
    - then it is not optional
  - If it must be managed by one and only one member of staff then we show it like this - exactly one
  - What about the other end of the association?
  - Does every member of staff have to manage exactly one campaign?
    - No. So the correct multiplicity is zero or more.

Some examples of specifying multiplicity:

- Optional (0 or 1) 0..1
- Exactly one 1 = 1..1
- Zero or more 0..* = *
- One or more 1..* = *
- A range of values 1..6
- A set of ranges 1..3,7..10,15,19..*

Class associations

- A client has exactly one staff member as a contact person
- A staff member has zero or more clients on his/her clientList

Direction

The "liaises with" association should be read in this direction

Role

The staff member's role in this association is as a contact person

Role

The client's role in this association is as a clientList

More Examples
### Association Classes

- Sometimes the association is itself a class
  - Because we need to retain information about the association
  - And that information doesn't naturally live in the classes at the ends of the association
  - E.g. a "title" is an object that represents information about the relationship between an owner and her car

Diagram:

- :car
  - VIN: Vehicle Identification Number
  - Year Made
  - Mileage
- :title
  - Year Bought
  - Initial Mileage
  - Price Paid
  - Licence Plate #
- :person
  - Name
  - Address
  - Driver's Licence Number
  - Permitted Vehicles

### Aggregation and Composition

- **Aggregation**
  - This is the "Has-a" or "Whole/part" relationship

- **Composition**
  - Strong form of aggregation that implies ownership:
    - If the whole is removed from the model, so is the part.
    - The whole is responsible for the disposition of its parts

Diagram:

- :Car
  - VIN
  - Year Made
  - Mileage
  - :person
    - Name
    - Address
    - Driver's Licence Number
    - Permitted Vehicles
  - Composition
  - Aggregation

### Generalization

- **Notes:**
  - Subclasses inherit attributes, associations, & operations from the superclass
  - A subclass may override an inherited aspect
  - E.g. AdminStaff & CreativeStaff have different methods for calculating bonuses
  - Superclasses may be declared (abstract), meaning they have no instances
    - Implies that the subclasses cover all possibilities
    - E.g. there are no other staff than AdminStaff and CreativeStaff

Diagram:

- Grade
  - Position
  -Allocated
- Staff Number
  - Name
  - Department
  - Salary
  - Superclasses are inherited by subclasses

### More on Generalization

- **Usefulness of generalization**
  - Can easily add new subclasses if the organization changes

- **Look for generalizations in two ways:**
  - **Top Down**
    - You have a class, and discover it can be subdivided
    - Or you have an association that expresses a "kind of" relationship
    - E.g. "Most of our work is on advertising for the press, that's newspapers and magazines, also for advertising hoardings, as well as for videos"
  - **Bottom Up**
    - You notice similarities between classes you have identified
    - E.g. "We have books and we have CDs in the collection, but they are all filed using the Dewey system, and they can all be lent out and reserved"

- **But don't generalize just for the sake of it**
  - Be sure that everything about the superclass applies to the subclasses
  - Be sure that the superclass is useful as a class in its own right
    - E.g. not just that we would discard using our tests for useful classes
  - Don't add subclasses or superclasses that are not relevant to your analysis
Evaluation of OOA

- **Advantages of OO analysis for RE**
  - Fits well with the use of OO for design and implementation
  - Transition from OOA to OOD 'smoother' (but is it?)
  - Removes emphasis on functions as a way of structuring the analysis
  - Avoids the fragmentary nature of structured analysis
  - Object-orientation is a coherent way of understanding the world

- **Disadvantages**
  - Emphasis on objects brings an emphasis on static modeling
  - Although later variants have introduced dynamic models
  - Not clear that the modeling primitives are appropriate
    - Are objects, services and relationships really the things we need to model in RE?
  - Strong temptation to do design rather than problem analysis
  - Fragmentation of the analysis
    - E.g. reliance on use-cases means there is no "big picture" of the user's needs
  - Too much marketing hype!
    - And false claims - e.g. no evidence that objects are a more natural way to think