3. Object-Orientation

**Objects**
- OO Programs built from Objects
- Objects communicate to solve task
- Hard to define
  - short int — number between ±32768
  - long int — number between ±$2^{31}$

**Objects in EMail system**
- What objects would you find in an automatic teller machine (ATM) system?

**Object State**
- Objects have data
- Only altered by the result of operations
- Determines the result of operations
- Mailbox may accept new messages when empty but not when full

**Object Operations**
- Objects have a set of operations they can perform
- OO Programs ask objects to carry out operations
- Not all objects support all operations (Can a mailbox solve linear equations?)
Object Identity

- State + Operations ≠ Object
- Objects need identity

Objects

- Mother Clanger is not the type of Objects we mean
- Computer representation of her in a game...
- Best to use experience and intuition to find them

Class

- Class describes a collection of objects that:
  - Support the same operations
  - Support the same possible states
  - Class definition must define both these

Mailbox Class

- Can:
  - Add a Message
  - Delete Message
  - List Messages
  - etc…

Mailbox Class

- State not arbitrary e.g:
  - Messages sorted by arrival time
- Objects that conform to a class are called instances of that class
From Problem to Code

- Class design
- Class decomposition
- Could be simple, could be complicated

From Problem to Code

- Can break this down into three phases
  - Analysis
  - Design
  - Implementation
  - This is not a linear-ordering (no waterfalls here!)
  - Consider 2 views – internal/external

From Problem to Code

- Implementation experience may lead to a better design
- New requirements mean the project design and implementation change
- OO software generally stands up better to these changes than procedural

Analysis

- Problem turned into a precise description
- Tasks completely defined
- No contradictions
- Readable by experts and developers
- Reviewable
- Testable against reality

Analysis

- ‘What’ not ‘How’...
- Specialist techniques for Large Projects, beyond the scope of the course

Design

- Structure the task into classes and class clusters
- Precise definitions of Class
  - Operations
  - Relationship to other classes
Design

• "What' not 'How'
• Although our "What's are now more specific
• Look at this in detail

Implementation

• Implementing the design in a programming language
• OO lends itself to gradual development
• Common to build a rapid prototype
• This can then inform the design and analysis phase

OO Design

• Goal to break the task into classes
• Define functionality of the classes

Email Example

• Consider an email system
• Likely to have several Mailboxes
• But what can we do with a mailbox?

Mailboxes

• What Operations?
  • Add email to it
  • View the current email
  • Delete the current email
  • Implies state (list of email, current email)
  • Implies other objects (email)

OO Design Process

• Identify the classes
• Identify the functionality of these classes
• Identify the relationships among these classes
OO Design Process

- Goals, not steps
- Iterative process
  - Find some classes
  - Define functionality
  - Leads to more classes

End Result

- List of Class Descriptions
- Overview of Class Relationships
- Foundation for Implementation

Finding Classes

- Look for nouns in the problem analysis
- Tend to be good choices for classes
- Others will become apparent

Finding Operations

- Verbs tend to point to operations
- Each operation should have exactly one class that is responsible for carrying it out
- Put operations on classes that make sense

Finding Relationships

- Three common relationships:
  - Association or use
  - Aggregation or containment
  - Inheritance or specialization

Association

- Class uses Objects of another class
- Important to minimize coupling
Aggregation

- Concrete relationship
- Pointers or actual objects
- Could be 1:1 or 1:n

Inheritance

- is-a relationship
- Specializes a class
- Add extra operations
- Enhance inherited operations

Inheritance

- Important to exploit similarities between classes
- Often one class (subclass) is a specialized version of another class (superclass)
- Subclass must be valid whenever the superclass is