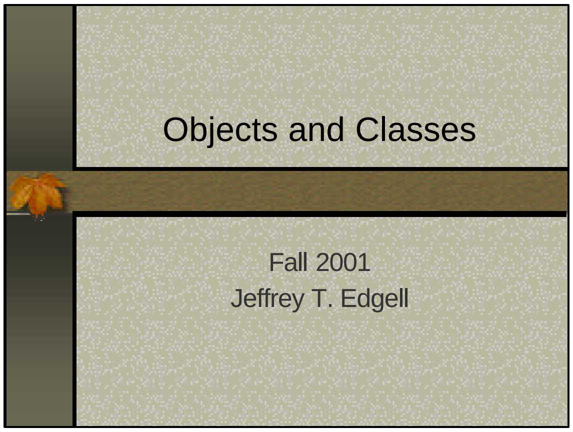
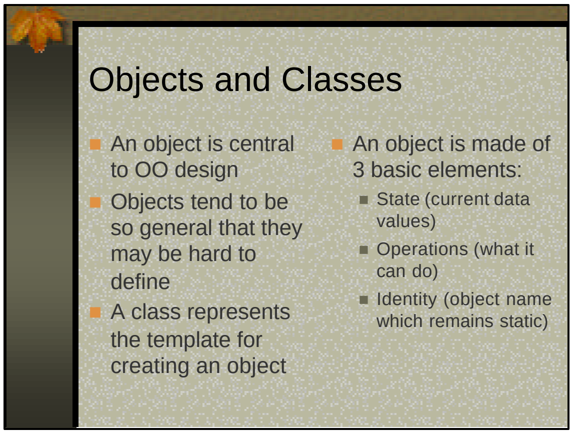


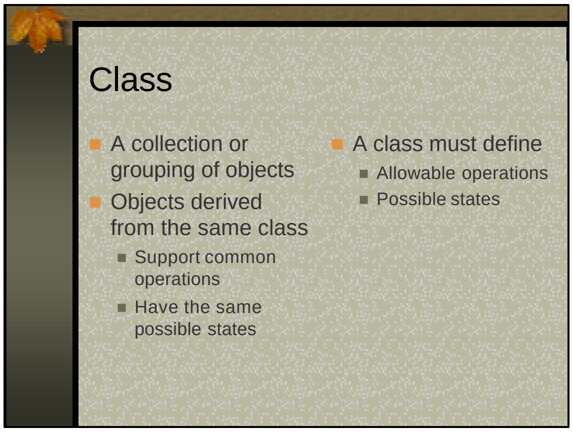
Objects and Classes

Fall 2001
Jeffrey T. Edgell



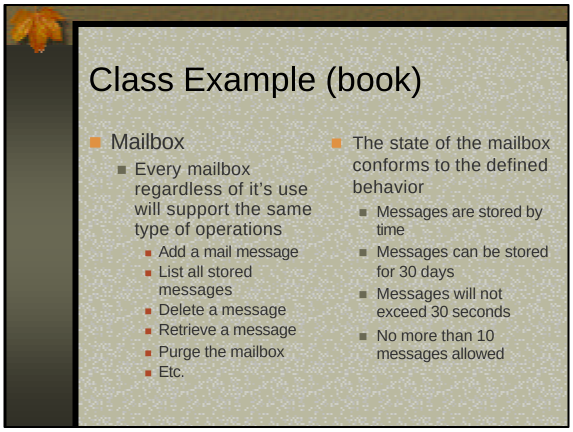
Objects and Classes

- An object is central to OO design
- Objects tend to be so general that they may be hard to define
- A class represents the template for creating an object
- An object is made of 3 basic elements:
  - State (current data values)
  - Operations (what it can do)
  - Identity (object name which remains static)



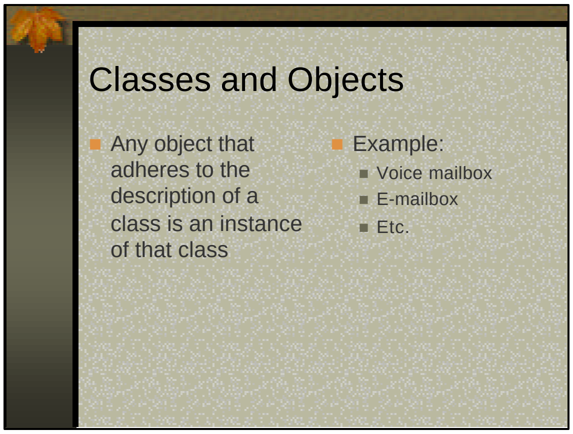
Class

- A collection or grouping of objects
- Objects derived from the same class
  - Support common operations
  - Have the same possible states
- A class must define
  - Allowable operations
  - Possible states



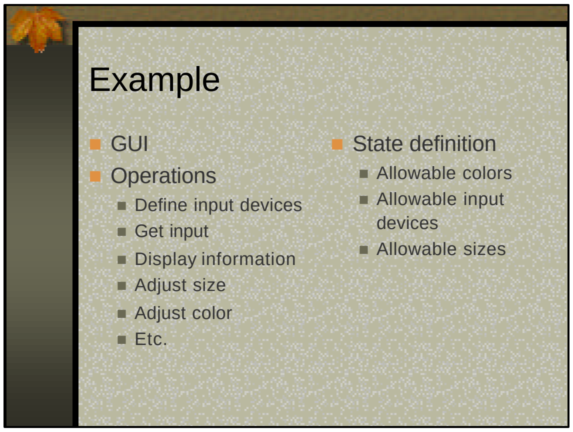
Class Example (book)

- Mailbox
  - Every mailbox regardless of it's use will support the same type of operations
    - Add a mail message
    - List all stored messages
    - Delete a message
    - Retrieve a message
    - Purge the mailbox
    - Etc.
- The state of the mailbox conforms to the defined behavior
  - Messages are stored by time
  - Messages can be stored for 30 days
  - Messages will not exceed 30 seconds
  - No more than 10 messages allowed



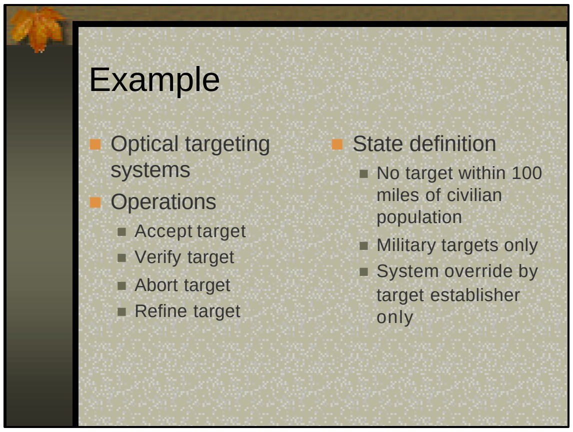
Classes and Objects

- Any object that adheres to the description of a class is an instance of that class
- Example:
  - Voice mailbox
  - E-mailbox
  - Etc.



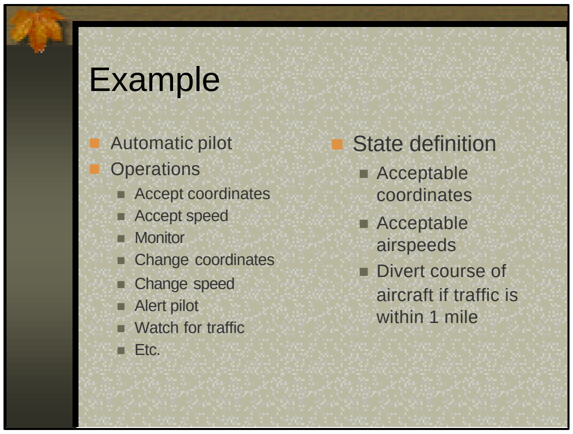
Example

- GUI
- Operations
  - Define input devices
  - Get input
  - Display information
  - Adjust size
  - Adjust color
  - Etc.
- State definition
  - Allowable colors
  - Allowable input devices
  - Allowable sizes



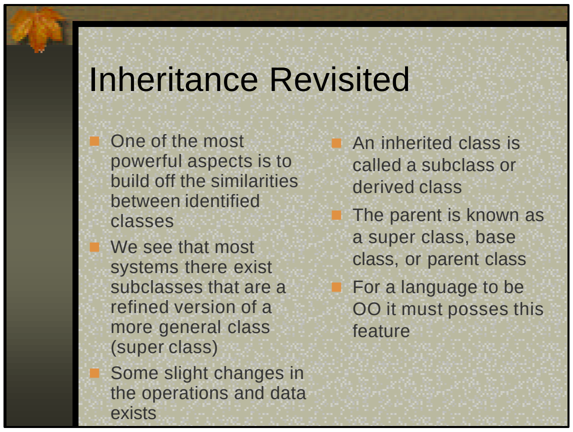
Example

- Optical targeting systems
- Operations
  - Accept target
  - Verify target
  - Abort target
  - Refine target
- State definition
  - No target within 100 miles of civilian population
  - Military targets only
  - System override by target establisher only



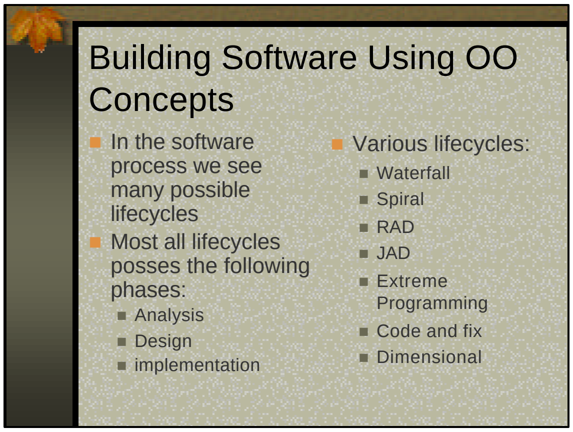
Example

- Automatic pilot
- Operations
  - Accept coordinates
  - Accept speed
  - Monitor
  - Change coordinates
  - Change speed
  - Alert pilot
  - Watch for traffic
  - Etc.
- State definition
  - Acceptable coordinates
  - Acceptable airspeeds
  - Divert course of aircraft if traffic is within 1 mile



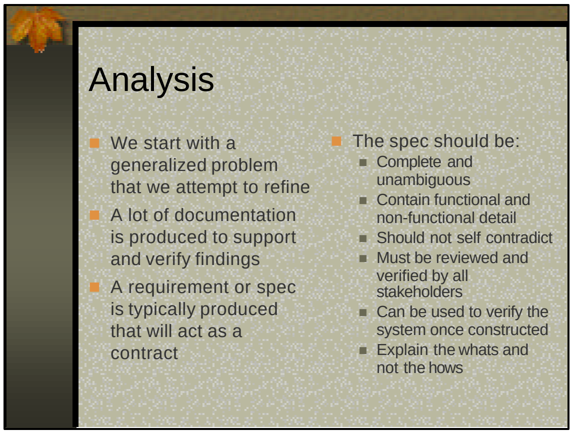
Inheritance Revisited

- One of the most powerful aspects is to build off the similarities between identified classes
- We see that most systems there exist subclasses that are a refined version of a more general class (super class)
- Some slight changes in the operations and data exists
- An inherited class is called a subclass or derived class
- The parent is known as a super class, base class, or parent class
- For a language to be OO it must posses this feature



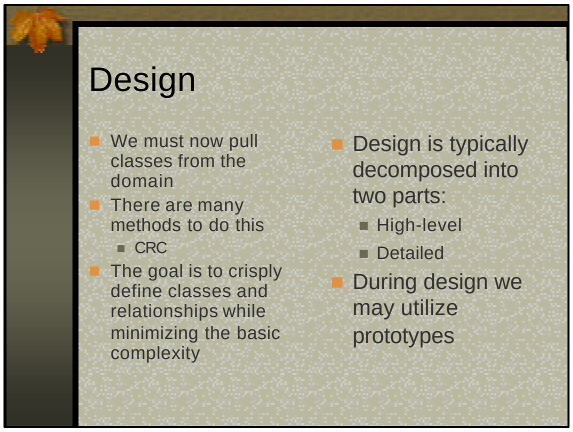
Building Software Using OO Concepts

- In the software process we see many possible lifecycles
- Most all lifecycles posses the following phases:
  - Analysis
  - Design
  - implementation
- Various lifecycles:
  - Waterfall
  - Spiral
  - RAD
  - JAD
  - Extreme Programming
  - Code and fix
  - Dimensional



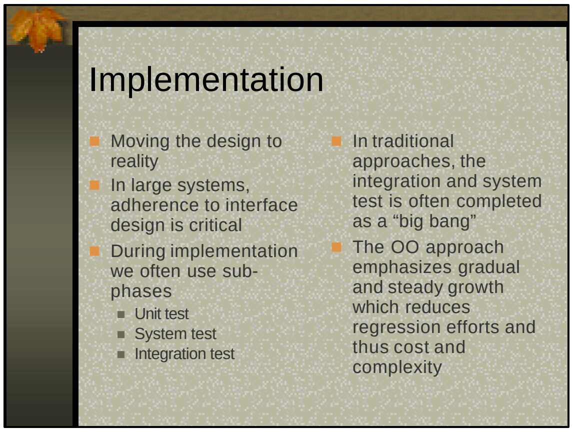
Analysis

- We start with a generalized problem that we attempt to refine
- A lot of documentation is produced to support and verify findings
- A requirement or spec is typically produced that will act as a contract
- The spec should be:
  - Complete and unambiguous
  - Contain functional and non-functional detail
  - Should not self contradict
  - Must be reviewed and verified by all stakeholders
  - Can be used to verify the system once constructed
  - Explain the whats and not the hows



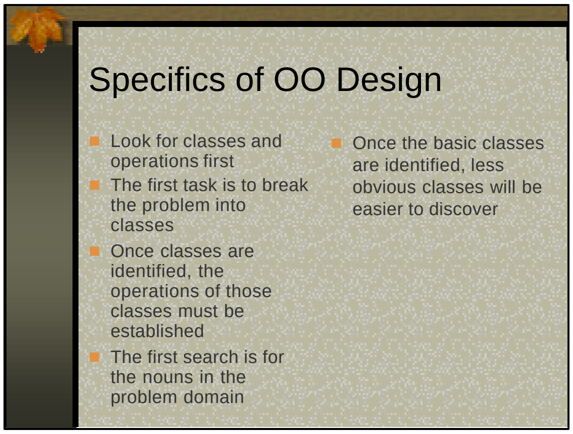
Design

- We must now pull classes from the domain
- There are many methods to do this
  - CRC
- The goal is to crisply define classes and relationships while minimizing the basic complexity
- Design is typically decomposed into two parts:
  - High-level
  - Detailed
- During design we may utilize prototypes



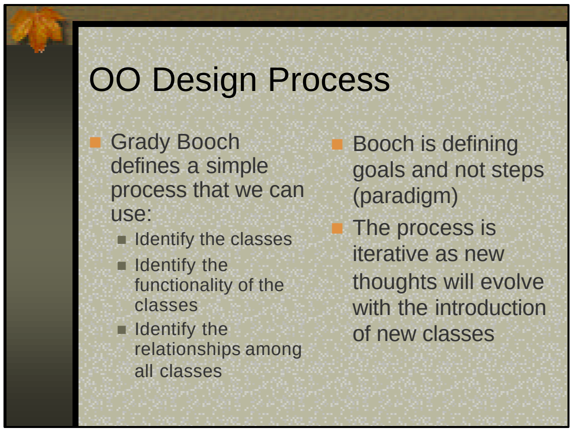
Implementation

- Moving the design to reality
- In large systems, adherence to interface design is critical
- During implementation we often use sub-phases
  - Unit test
  - System test
  - Integration test
- In traditional approaches, the integration and system test is often completed as a “big bang”
- The OO approach emphasizes gradual and steady growth which reduces regression efforts and thus cost and complexity



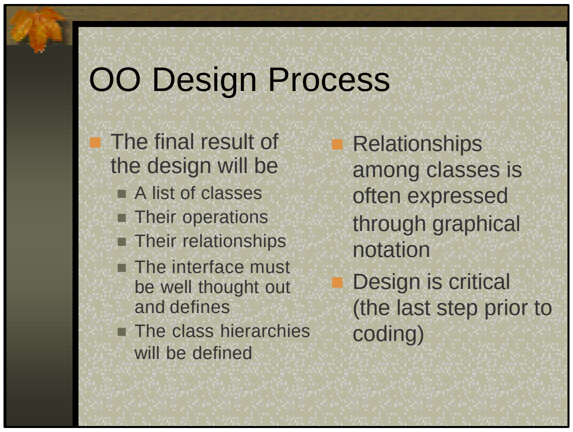
Specifics of OO Design

- Look for classes and operations first
- The first task is to break the problem into classes
- Once classes are identified, the operations of those classes must be established
- The first search is for the nouns in the problem domain
- Once the basic classes are identified, less obvious classes will be easier to discover



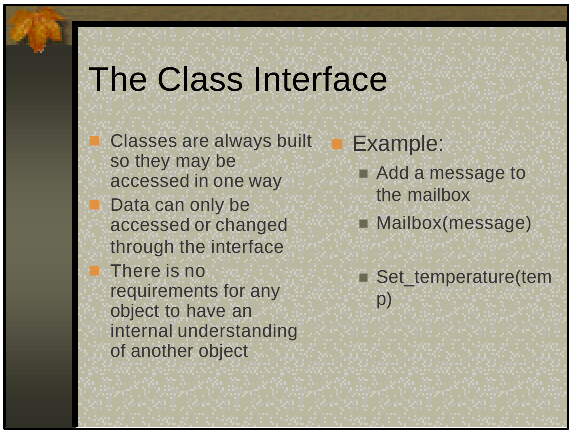
OO Design Process

- Grady Booch defines a simple process that we can use:
  - Identify the classes
  - Identify the functionality of the classes
  - Identify the relationships among all classes
- Booch is defining goals and not steps (paradigm)
- The process is iterative as new thoughts will evolve with the introduction of new classes



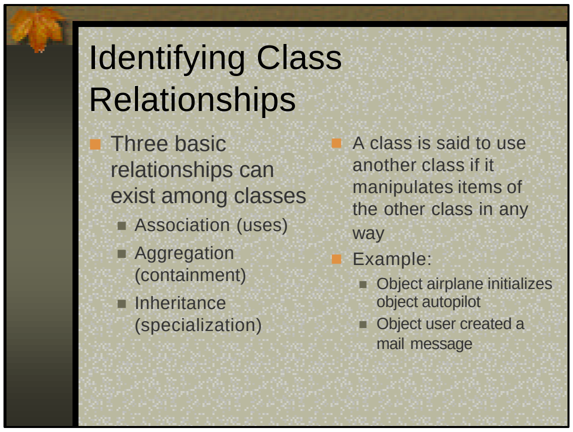
OO Design Process

- The final result of the design will be
  - A list of classes
  - Their operations
  - Their relationships
  - The interface must be well thought out and defines
  - The class hierarchies will be defined
- Relationships among classes is often expressed through graphical notation
- Design is critical (the last step prior to coding)



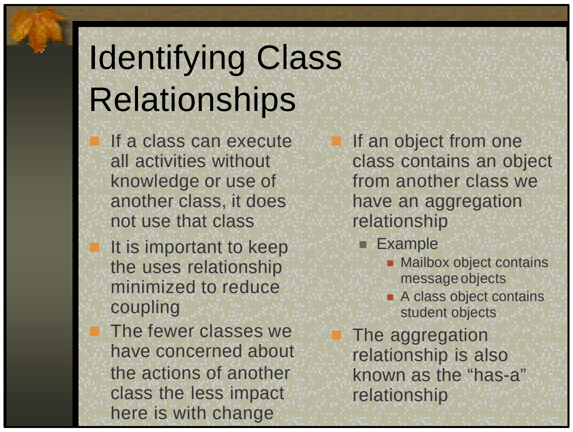
The Class Interface

- Classes are always built so they may be accessed in one way
- Data can only be accessed or changed through the interface
- There is no requirements for any object to have an internal understanding of another object
- Example:
  - Add a message to the mailbox
  - Mailbox(message)
  - Set_temperature(temp)



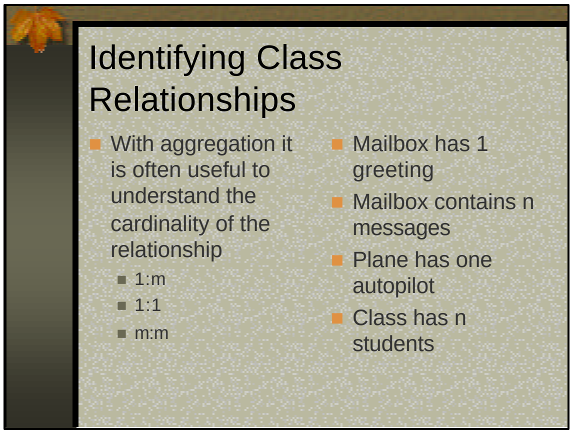
Identifying Class Relationships

- Three basic relationships can exist among classes
  - Association (uses)
  - Aggregation (containment)
  - Inheritance (specialization)
- A class is said to use another class if it manipulates items of the other class in any way
- Example:
  - Object airplane initializes object autopilot
  - Object user created a mail message



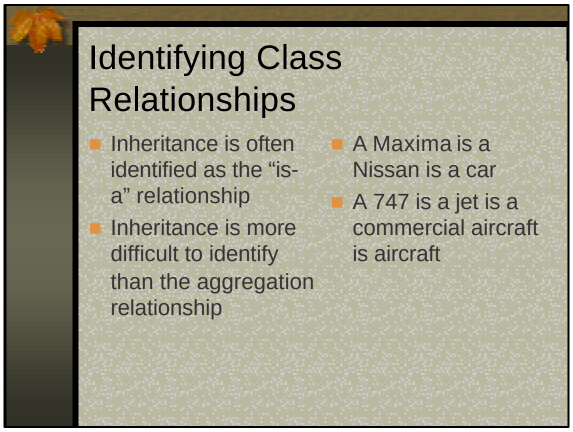
Identifying Class Relationships

- If a class can execute all activities without knowledge or use of another class, it does not use that class
- It is important to keep the uses relationship minimized to reduce coupling
- The fewer classes we have concerned about the actions of another class the less impact here is with change
- If an object from one class contains an object from another class we have an aggregation relationship
  - Example
    - Mailbox object contains message objects
    - A class object contains student objects
- The aggregation relationship is also known as the “has-a” relationship



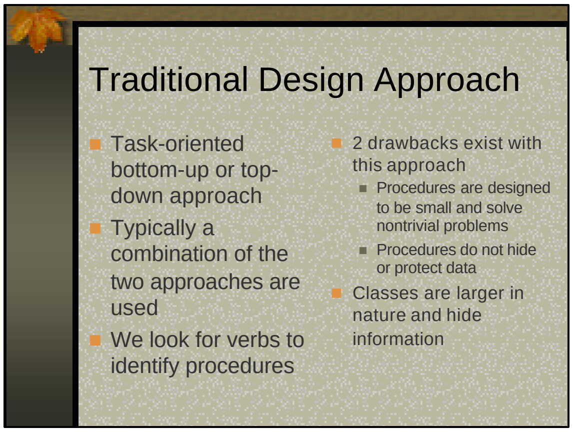
Identifying Class Relationships

- With aggregation it is often useful to understand the cardinality of the relationship
  - 1:m
  - 1:1
  - m:m
- Mailbox has 1 greeting
- Mailbox contains n messages
- Plane has one autopilot
- Class has n students



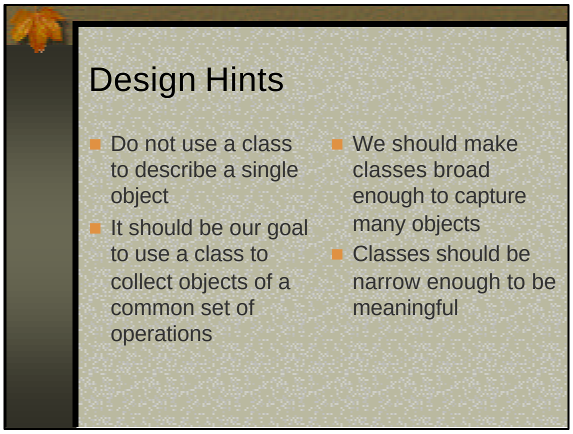
Identifying Class Relationships

- Inheritance is often identified as the “is-a” relationship
- Inheritance is more difficult to identify than the aggregation relationship
- A Maxima is a Nissan is a car
- A 747 is a jet is a commercial aircraft is aircraft



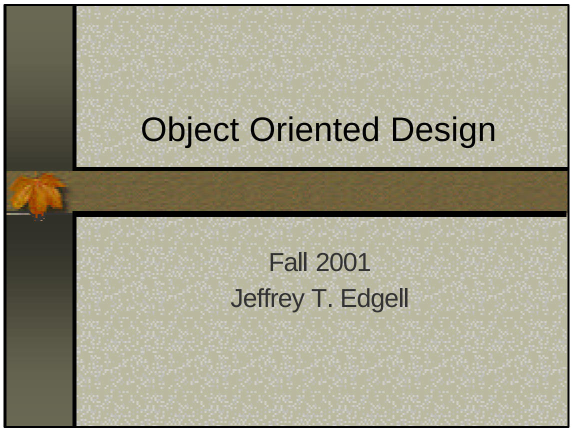
Traditional Design Approach

- Task-oriented bottom-up or top-down approach
- Typically a combination of the two approaches are used
- We look for verbs to identify procedures
- 2 drawbacks exist with this approach
  - Procedures are designed to be small and solve nontrivial problems
  - Procedures do not hide or protect data
- Classes are larger in nature and hide information



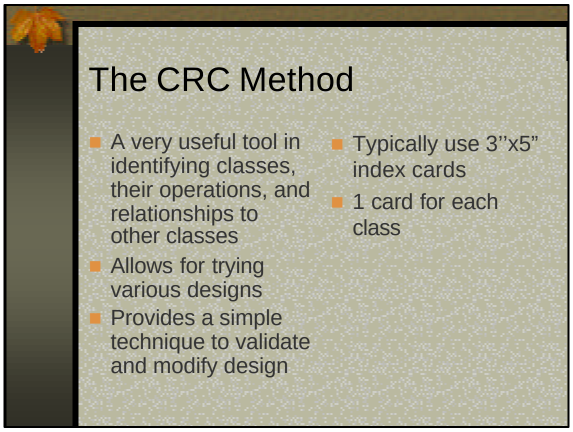
Design Hints

- Do not use a class to describe a single object
- It should be our goal to use a class to collect objects of a common set of operations
- We should make classes broad enough to capture many objects
- Classes should be narrow enough to be meaningful



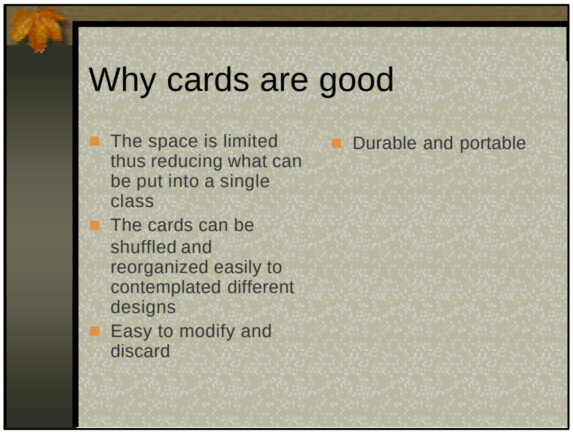
Object Oriented Design

Fall 2001
Jeffrey T. Edgell



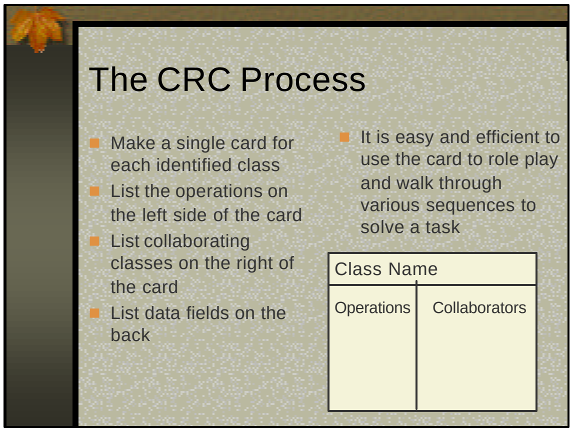
The CRC Method

- A very useful tool in identifying classes, their operations, and relationships to other classes
- Allows for trying various designs
- Provides a simple technique to validate and modify design
- Typically use 3''x5" index cards
- 1 card for each class



Why cards are good

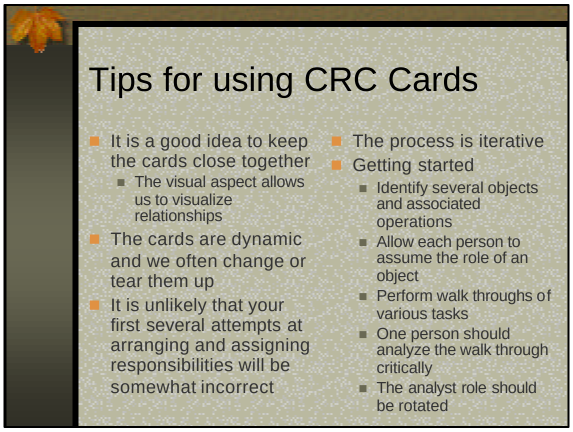
- The space is limited thus reducing what can be put into a single class
- The cards can be shuffled and reorganized easily to contemplated different designs
- Easy to modify and discard
- Durable and portable



The CRC Process

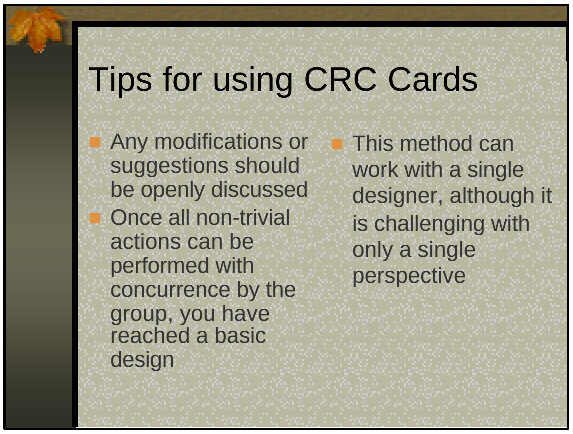
- Make a single card for each identified class
- List the operations on the left side of the card
- List collaborating classes on the right of the card
- List data fields on the back
- It is easy and efficient to use the card to role play and walk through various sequences to solve a task

Class Name	
Operations	Collaborators



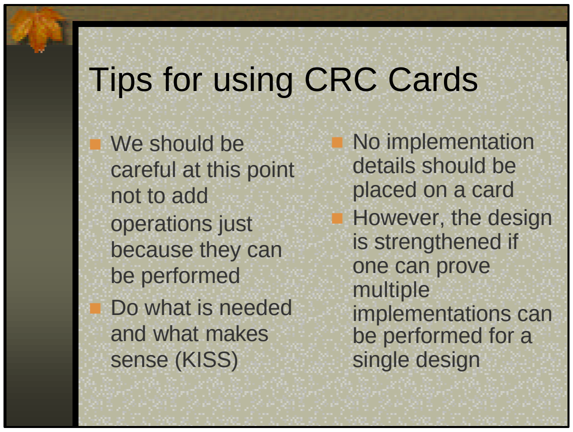
Tips for using CRC Cards

- It is a good idea to keep the cards close together
  - The visual aspect allows us to visualize relationships
- The cards are dynamic and we often change or tear them up
- It is unlikely that your first several attempts at arranging and assigning responsibilities will be somewhat incorrect
- The process is iterative
- Getting started
  - Identify several objects and associated operations
  - Allow each person to assume the role of an object
  - Perform walk throughs of various tasks
  - One person should analyze the walk through critically
  - The analyst role should be rotated



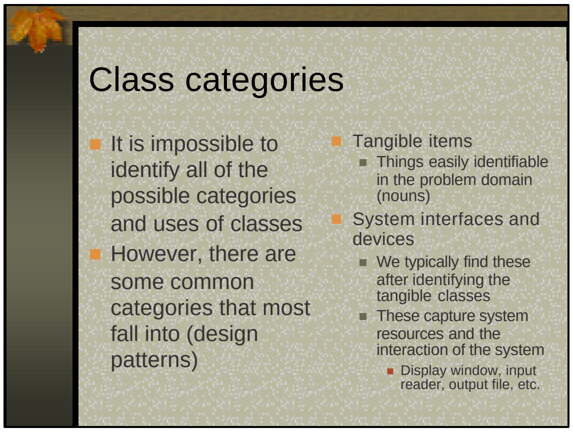
Tips for using CRC Cards

- Any modifications or suggestions should be openly discussed
- Once all non-trivial actions can be performed with concurrence by the group, you have reached a basic design
- This method can work with a single designer, although it is challenging with only a single perspective



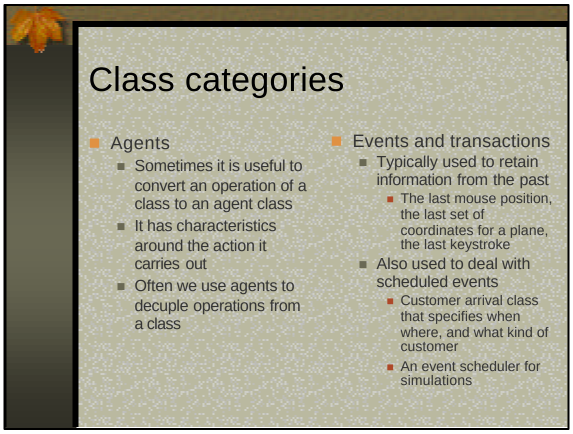
Tips for using CRC Cards

- We should be careful at this point not to add operations just because they can be performed
- Do what is needed and what makes sense (KISS)
- No implementation details should be placed on a card
- However, the design is strengthened if one can prove multiple implementations can be performed for a single design



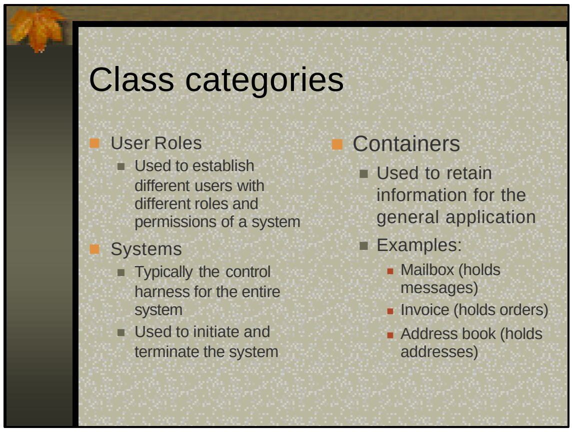
Class categories

- It is impossible to identify all of the possible categories and uses of classes
- However, there are some common categories that most fall into (design patterns)
- Tangible items
  - Things easily identifiable in the problem domain (nouns)
- System interfaces and devices
  - We typically find these after identifying the tangible classes
  - These capture system resources and the interaction of the system
    - Display window, input reader, output file, etc.



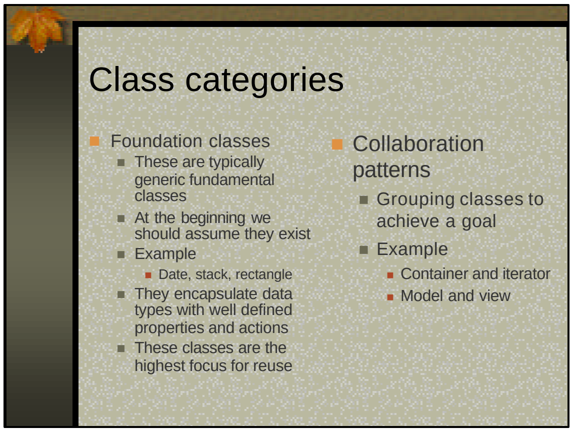
Class categories

- Agents
  - Sometimes it is useful to convert an operation of a class to an agent class
  - It has characteristics around the action it carries out
  - Often we use agents to decuple operations from a class
- Events and transactions
  - Typically used to retain information from the past
    - The last mouse position, the last set of coordinates for a plane, the last keystroke
  - Also used to deal with scheduled events
    - Customer arrival class that specifies when where, and what kind of customer
    - An event scheduler for simulations



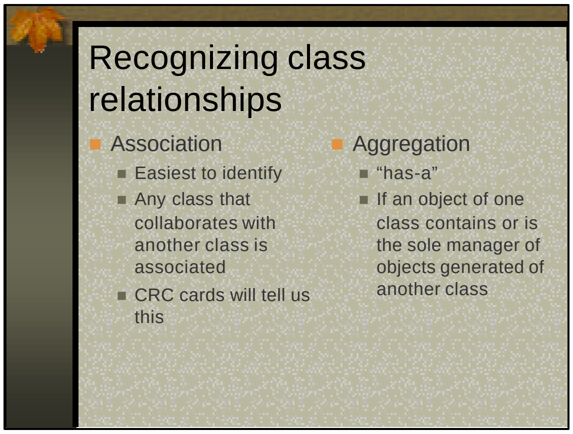
Class categories

- User Roles
  - Used to establish different users with different roles and permissions of a system
- Systems
  - Typically the control harness for the entire system
  - Used to initiate and terminate the system
- Containers
  - Used to retain information for the general application
  - Examples:
    - Mailbox (holds messages)
    - Invoice (holds orders)
    - Address book (holds addresses)



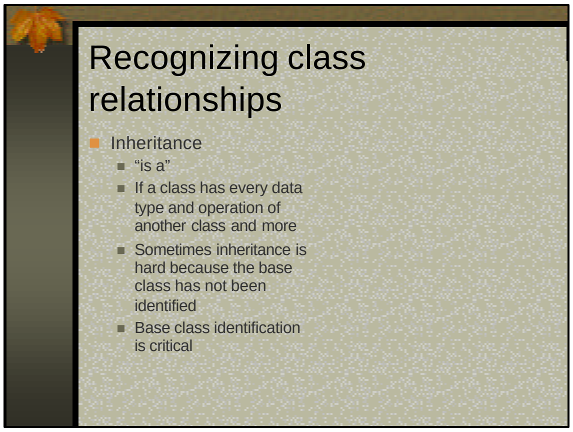
Class categories

- Foundation classes
  - These are typically generic fundamental classes
  - At the beginning we should assume they exist
  - Example
    - Date, stack, rectangle
  - They encapsulate data types with well defined properties and actions
  - These classes are the highest focus for reuse
- Collaboration patterns
  - Grouping classes to achieve a goal
  - Example
    - Container and iterator
    - Model and view



Recognizing class relationships

- Association
  - Easiest to identify
  - Any class that collaborates with another class is associated
  - CRC cards will tell us this
- Aggregation
  - "has-a"
  - If an object of one class contains or is the sole manager of objects generated of another class



Recognizing class relationships

- Inheritance
  - "is a"
  - If a class has every data type and operation of another class and more
  - Sometimes inheritance is hard because the base class has not been identified
  - Base class identification is critical