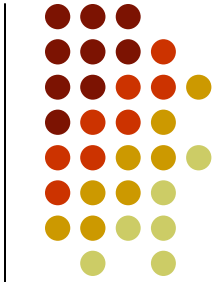
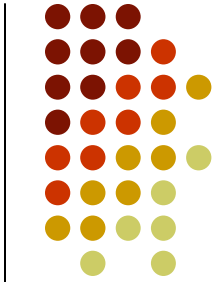
# Object Oriented Design



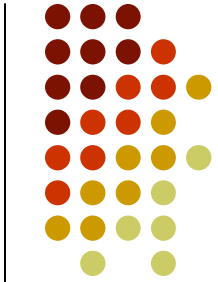
- An object combines data and operations on that data (object is an instance of class)
  - data: class variables
  - operations: methods
- Three principles of Object Oriented Design
  - Encapsulation – discussed earlier
  - Inheritance – discussed later in the course
  - Polymorphism – discussed later in the course

# Object Design – Identify Objects



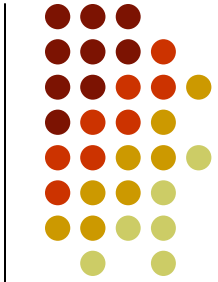
- Identify objects
  - Identify objects that exist in the problem statement and requirements
  - Typically, select nouns, ignoring irrelevant ones, such as synonyms
- Look for relationships amongst the (real-world) objects that were identified
  - Generalization – relates to inheritance
  - Containment – where one object contains another
  - Multiplicity – determine the quantity relationships between objects (e.g. one bank can have many accounts)

# Object Design – Identify Operations



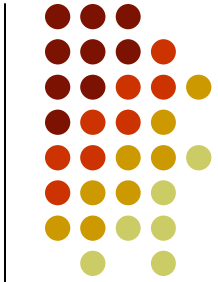
- Identify the operations of objects
- Typically, select verbs, ignoring irrelevant ones, such as actions performed by the user
- Associate each operation with the object that is responsible for providing the behaviour
- Note that an object should be responsible for modifying its own data

# Object Design – Create Interface



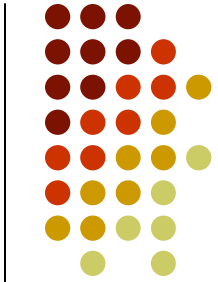
- An interface should be created for each object that is to be represented by a class (rather than a variable)
  - The interface describes how the class can be used, by specifying its public operations
- There are a number of ways of creating interfaces, e.g.
  - C++ Header files
  - Java Interface
- An interface should include:
  - Class invariants (conditions that must be true for an object)
  - Public methods, for each such define:
    - Parameter lists
    - Return type
    - Purpose (i.e. a description)
    - Pre and post conditions

# Class Invariants



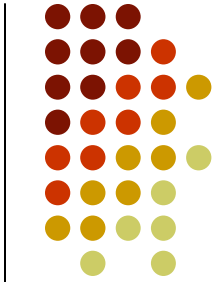
- A class invariant is an invariant on the values of the variables of an object,  For example:
  - Account balance is always >= 0
  - Account ID numbers are unique and cannot be modified
- All object constructors and mutators should respect class invariants
  - That is, they should always make sure that class invariants are true

# Pre-Conditions



- A pre-condition is an assertion about conditions at the beginning of a method
  - An assertion is a statement about a condition
- It is part of the "contract" implicit in a method
  - If the pre-conditions are not true then the method is not guaranteed to produce the desired results
  - e.g. for binary search, there is a pre-condition that the array or list being searched is sorted, it is not, there is not guarantee that the correct result will be returned
  - Or, to put it another way, that the post-conditions will hold

# Post-Conditions

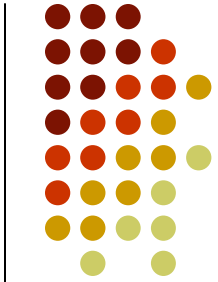


- A post-condition is an assertion about conditions at the end of a method
  - The post-conditions describe the state after the method has been run
  - They therefore describe the desired output of a method
- To prove that an algorithm is correct we need to prove that, if the pre-conditions are correct, following the steps of the algorithm will lead to the post-conditions
  - We should be able to show how the each step of an algorithm leads to the post-conditions

# Example (Object Oriented Design)

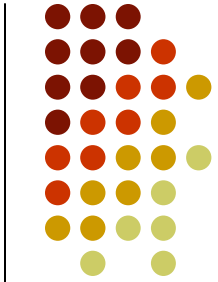
- You have been hired to design an application for a company to record information about its employees. You have been given the following information about the application's requirements.
- The company has a number of branches. It is necessary to record the address of the building occupied by each branch, as well as the manager (who is an employee). The company also needs to record the unique employee ID, first name, last name and annual salary of each employee. Each employee works for one and only one branch.
- The application will be used to perform the following tasks:
- Insert and delete (fire) employees, and transfer them between branches
- Change any employee information (with exception of the which employee's IDs cannot be changed, also note that an employee's salary cannot go below zero)
- Retrieve employee information
- Insert and delete branches; branches can only be deleted if they have no employees
- Change branch data
- Retrieve branch data, including the monthly payroll costs (the sum of the annual salaries of all employees who work for the branch divided by 12)

# Object Interaction Design



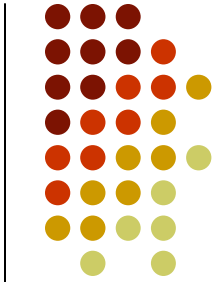
- Design the entire application by describing how the objects are to interact with each other
- One approach is to write a high level algorithm and then decompose it into several methods
  - Each such method should have one specific purpose

# Operation Design



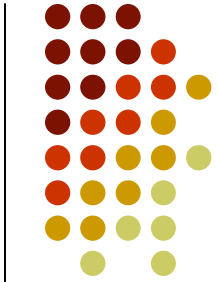
- Each of the operations identified in the object design should be designed
- If possible, re-use previously written methods or functions
- Otherwise design algorithms to implement each operation

# Test Design



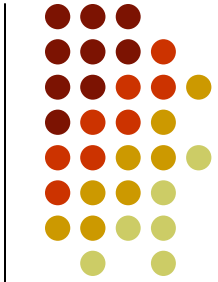
- Determine how each class (or unit, or module) will be tested before writing the application
- Write unit test cases that test the entire range of input that can be given to a method
  - Expected values
  - Boundary values
  - Invalid values
- For each input specify the expected result and when testing is performed compare this to the actual result
  - Debug where necessary!

# What makes a good program?



- Modularity
  - Favorable impact on program development
- Modifiability
  - Use of methods and named constants
- Ease of Use
  - Considerations for the user interface
    - Program should prompt the user for input
    - A program should always echo its input
    - The output should be well labeled and easy to read

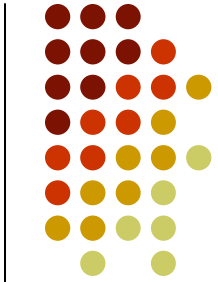
# What makes a good program?



- Efficient

- Fail-Safe Programming
  - Fail-safe program
    - A program that will perform reasonably no matter how anyone uses it
  - Types of errors:
    - Errors in input data
    - Errors in the program logic

# What makes a good program?



- Style
  - Five issues of style
    - Extensive use of methods
    - Use of private data fields
    - Error handling
    - Readability
    - Documentation