

# CMSC 132: Object-Oriented Programming II

---



## Problem Specification & Design

Department of Computer Science  
University of Maryland, College Park

### Overview

- Problem specification
  - Obstacles
- Program design
  - How to divide work
  - Interface & conditions

## Problem Specification

### ■ Goal

- Create complete, accurate, and unambiguous statement of problem to be solved

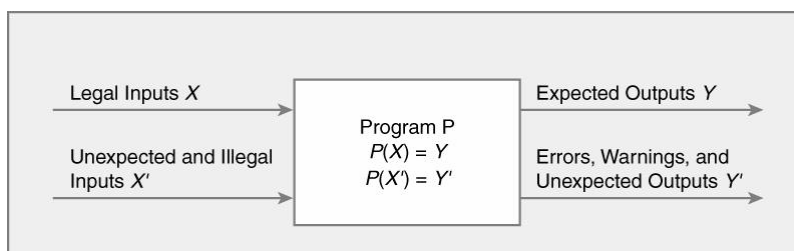
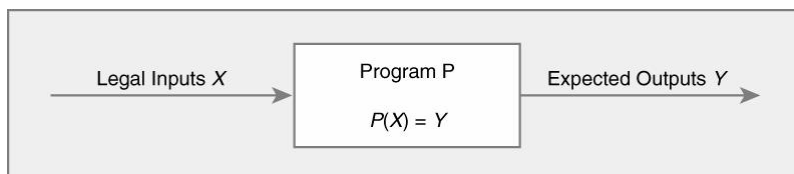
### ■ Problems

- Description may not be accurate
- Description may change over time
- Difficult to specify behavior for all inputs
- Natural language description is imprecise
- Formal specification languages limited and difficult to understand

## Problem Specification

### ■ Example

- Specification of input & output for program



## Problem Specification Problems

- **Description may not be accurate**
  - Problem not understood by customer
- **Description may change over time**
  - Customer changes their mind
- **Difficult to specify behavior for all inputs**
  - Usually only covers common cases
  - Hard to consider all inputs (may be impossible)
  - Example
    - Most UNIX utilities crash with random inputs

## Problem Specification Problems

- **Description may be ambiguous**
  - Natural language description is imprecise
    - Why lawyers use legalese for contracts
  - Formal specification languages are limited and may be difficult to understand
  - Examples
    - Find sum of all values in N-element list L between 1 and 100
    - $\sum_{i=0}^{N-1} L_i \ni (L_i \geq 1) \wedge (L_i \leq 100)$
- **Difficult to write specifications that are both readable and precise**

## Program Design

### ■ Goal

- Break software into integrated set of **components** that work together to solve problem specification

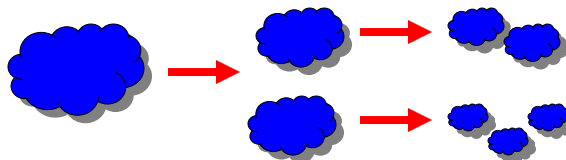
### ■ Problems

- Methods for decomposing problem
  - How to divide work
  - What work to divide
- How components work together

## Design – How To Divide Work

### ■ Decomposing problem

- Break large problem into many smaller problems
  - Cannot solve large problems directly
- Divide and conquer
  1. Break problem up into simpler sub-problems
  2. Repeat for each sub-problem
  3. Stop when sub-problem can be solved easily

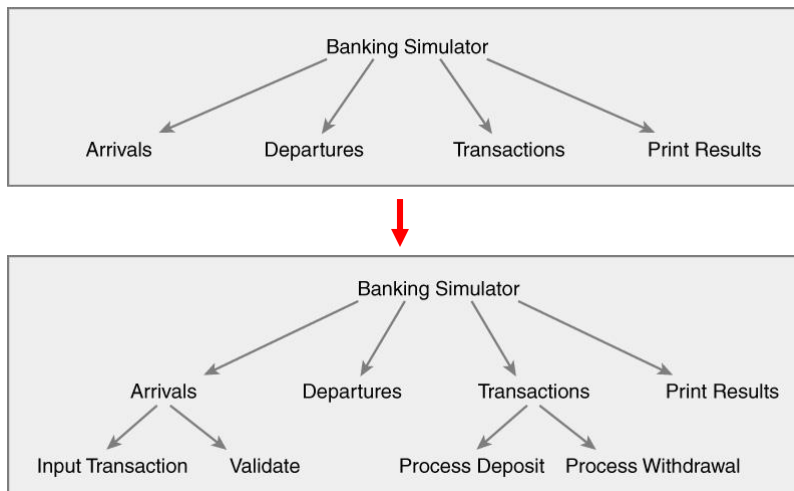


## Design – How To Divide Work

- **Functional approach**
  - Treat problem as a collection of **functions**
- **Techniques**
  - **Top-down design**
    - Successively split problem into smaller problems
  - **Bottom-up design**
    - Start from small tasks and combine

## Design – Decomposition Example

- **Top-down design of banking simulator**



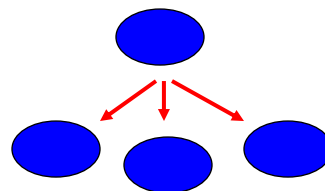
## Design – How To Divide Work

- **Object-oriented approach**
  - Treat problem as a collection of data **objects**
  - **Objects**
    - Entities that exist in problem
    - Contain data
    - Perform actions associated with data

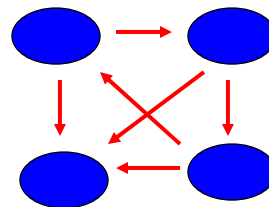
## Design – Comparison Example

### ■ **Bank simulation**

- **Functional programming**
  - Arrivals, departures, transactions



- **Object-oriented programming**
  - Customers, lines, tellers, transactions



## Design – Comparing Approaches

- **Functional approach**
  - Treat problem as a collection of **functions**
  - Functions perform actions
  - Think of functions as **verbs**
- **Object-oriented approach**
  - Treat problem as a collection of data **objects**
  - Objects are entities that exist in problem
  - Think of objects as **nouns**

## Design – Comparing Approaches

- **Advantages to object-oriented approach**
  - Helps to **abstract** problem
    - Simpler high-level view
  - Helps to **encapsulate** data
    - Hides details of internals of objects
    - Centralizes and protects all accesses to data
  - Seems to scale better for larger projects
- **In practice**
  - Tend to use a combination of all approaches

## Design – Components

- Components must work together easily
- Each component requires
  - Interface
    - Specifies **how** component is accessed & used
    - Specifies **what** functions (methods) are available
  - Pre-conditions
    - What conditions must be true **before** invocation
  - Post-conditions
    - What conditions will be true **after** invocation
- Pre & post conditions represent a **contract** between designer & programmer

## Design – Interface & Conditions

- Function positivePower()
  - Calculate  $x^n$  for positive values of  $x$  &  $n$
- Interface
  - `public static float positivePower(float x, int n)`
- Pre-conditions
  - $x$  has positive floating point value  $> 0.0$
  - $n$  has positive integer value  $\geq 0$
- Post-conditions
  - Returns  $x^n$  if preconditions are met
  - Returns  $-1.0$  otherwise