

# Abstraction

CIS\*2520 Summer 2006

## Abstraction

- Most difficult concept to master to do well in this course
- Unifying theme for everything we will learn and do
- Fundamental concept in computing science and problem solving in general

CIS\*2520 - Summer 2006

D. McCaughan

## Abstraction

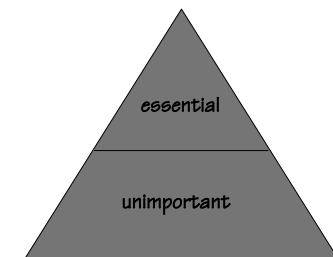
- Recall definition
  - Abstraction: reasoning about the essential properties of an object, ignoring unimportant details
- Consider going to a movie
  - what do you really need to know in order to “solve” this problem?
  - at what level of abstraction are these details?

CIS\*2520 - Summer 2006

D. McCaughan

## Abstraction

- Steps
  - 1) *examine complex problem*
    - distinguish between high-level information and low-level information
  - 2) *pull out the high level information*
  - 3) *solve less complex problem*



CIS\*2520 - Summer 2006

D. McCaughan

## Abstraction Example

- Four kitchen devices
  - cook must understand interface of a device before applying it
  - to what extent has this interface been abstracted?
    - abstract devices: interface understood in terms of input/output; cook can ignore internal details of the device
    - non-abstract devices: require understanding of all details
  - note: abstract devices can be used even if the method used to accomplish their task is a mystery

CIS\*2520 - Summer 2006

D. McCaughan

## Abstraction Example

- Apple Corer
  - "a round metal devices that you place on top of an apple and push downward, splitting the apple into its core and several slices"
  - no abstraction - must understand the process to make it work
- Microwave
  - completely abstract interface
  - *input*: food to be heated & setting --- *output*: hot food
- Barbeque
  - would be convenient to have an abstract interface, however the reality is that the cook must understand the process in detail
- Sink/Tap
  - completely abstract
  - *input*: turn tap --- *output*: water

CIS\*2520 - Summer 2006

D. McCaughan

## Abstraction Example

- Apple corer and BBQ much less abstract than the microwave and sink/tap
- There is still some additional order to be imposed
  - i.e. BBQ more abstract than sink/tap
- Ordered by increasing level of abstraction:
  - corer -> bbq -> sink/tap -> microwave
- Aside: Generality
  - recall: applicable to many tasks rather than a specific few
  - ranking: corer -> sink/tap -> bbq -> microwave

CIS\*2520 - Summer 2006

D. McCaughan

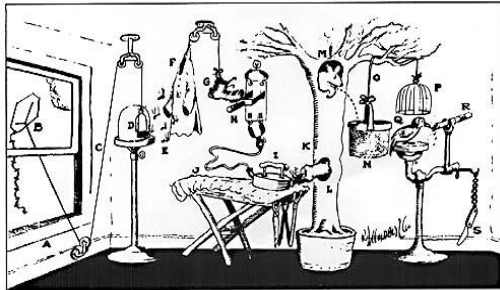
## Implementation

- Are all implementations created equal?
- Rube-Goldberg machines
  - goal: perform a simple task in as complex a way as possible
- Endless different implementations of a specification
  - not all are equal: differ in complexity / efficiency / cost
  - you will build better, more robust and bug-free systems if you keep this in mind and make intelligent decisions

CIS\*2520 - Summer 2006

D. McCaughan

## Rube-Goldberg Pencil Sharpener



Pencil Sharpener RUBE GOLDBERG (pm) KGI 038

- Open window(A) and fly kite(B). String(C) lifts small door(D) allowing moths(E) to escape and eat flannel shirt(F). As weight of shirt becomes less, shoe(G) steps on switch(H) which heats electric iron(I) and burns hole in pants(J). Smoke(K) enters hole in tree(L), smoking out opossum(M) which jumps into basket(N), pulling rope(O) and lifting cage(P), allowing woodpecker(Q) to chew wood from pencil(R), exposing lead. Emergency knife(S) is always handy in case opossum or the woodpecker gets sick and can't work.

Image and text used with permission. Rube Goldberg is the ® and © of Rube Goldberg Inc.

CIS\*2520 - Summer 2006

D. McCaughan

## Abstracting Data

- Recall: *specification* (i.e. “what”) and *implementation* (i.e. “how”)
- For subroutines in code:
  - specification: method header + pre/post conditions
  - implementation: local variables + body of subroutine
- Extend this to data types:
  - specification: definition of data type + operations defined on that type
  - Implementation: in C - a header file containing type and function declarations together with a .c file in which they are implemented

CIS\*2520 - Summer 2006

D. McCaughan

## Abstracting Data

- General Idea:
  - data types/algorithms organized into modules
  - module contains data structure + operations
  - module exports (makes public)
    - a type (in an OO language this would be the class itself)
    - methods that operate on instances of this type
  - user of the module can declare variables of this type, and manipulate them by calling the ADT's operations
  - **NOTE: no need to know implementation details**

CIS\*2520 - Summer 2006

D. McCaughan

## ADT Module Conventions

- Some operations are common to many ADT specifications
  - these should be called once for each variable (at least conceptually)
    - ADT lifecycle: declare -> create -> use -> destroy
  - `create`
    - initialization of internal variables/allocation of dynamic structures
  - `destroy`
    - management of de-allocation of dynamic resources
  - others are related to the gross type of the ADT:
    - `typeRead`, `typeWrite`, `typeIsEqual`, `typeIsEmpty`, etc.

CIS\*2520 - Summer 2006

D. McCaughan

## Fraction ADT

- Consider: for integers and floats we can do:
  - $a = b+c$ ;  $a = b-c$ ;  $a = b*c$ ;  $a = b/c$ ;
  - i.e. assignment and arithmetic
- We would like to do something similar for fractions:
  - $a = \text{op}(b, '+', c)$ ;
  - alternatively  $\text{fSum}(a, b, c)$ ;  $\text{fDiff}(a, b, c)$ ;
    - or in a more Java/C++ way:  $a.\text{sumOf}(b, c)$ ,  
 $a.\text{diffOf}(b, c)$ , etc.
- What do we need in order to specify a Fraction ADT?

CIS\*2520 - Summer 2006

D. McCaughan

## Fraction ADT

- DEF'N:
  - A fraction is a number  $a/b$  where  $a, b$  are integers,  $b$  is non-zero and  $\text{gcd}(a, b) = 1$
- OPERATIONS:
  - **create**
    - creates a new Fraction object with a provided numerator and denominator represented in reduced form
    - PRE: denominator is not 0
    - POST: fraction has been reduced and stored
  - **destroy**
    - deletes the provided Fraction object
    - PRE: n/a
    - POST: fraction is destroyed

CIS\*2520 - Summer 2006

D. McCaughan

## Fraction ADT

- OPERATIONS (cont.):
  - **op**
    - performs arithmetic operation from  $\{+, -, *, /\}$  on the provided operands (with assignment)
    - PRE: operator is in  $\{+, -, *, /\}$ , second operand is not 0 where operator is  $/$
    - POST: fraction is assigned the reduced result of applying the operation to the operands
  - **read**
    - formatted input of fraction with format " $a / b$ ", where  $a, b$  ints
    - PRE: input contains a valid representation of a fraction
    - POST: fraction converted is reduced and stored
  - **write**
    - formatted output of a fraction object with format " $a / b$ "
    - PRE: void
    - POST: fraction is converted to formatted output

CIS\*2520 - Summer 2006

D. McCaughan

## Fraction ADT

- Note: it is an implicit precondition that the objects exist
- see also: Fraction ADT example (in C)
- see also: DisplayFractions application
- Food for thought...
  - what about a String ADT?, DeckOfCards ADT?

CIS\*2520 - Summer 2006

D. McCaughan

## About Data Hiding and ADTs

- You must respect that *interface* == “the wall”
- Consider a String ADT (where length is stored as an integer in the ADT implementation)
  - what’s the big deal with getting the length?
    - `size = word.length;` vs.
    - `size = strGetLength(word);`
  - what if the underlying implementation changes?
    - your code breaks!
- Another view of information hiding:
  - a (conceptual) wall separating the implementation of the type from its use in higher level code

CIS\*2520 - Summer 2006

D. McCaughan

## About Data Hiding and ADTs

- NOTE: this all encourages a “layered approach”, even to modules built from, or extending, other modules
  - i.e. the ADT that layers itself on top of an existing ADT is the “higher level code” in this case (not necessarily the main application)
  - extension and encapsulation of ADTs!
- Recall that DeckOfCards ADT
  - how might you implement that?
- An object-oriented “drawing program”
  - Shape ADT -> { Circle ADT, Square ADT, Triangle ADT }
- The modules we construct represent a *Spectrum of Generality*
  - program to manipulate a deck of cards uses a DeckOfCards ADT which extends a Queue ADT which encapsulates a List ADT...

CIS\*2520 - Summer 2006

D. McCaughan

## Refinement of an ADT

- Abstract level
  - relation between data and operations needed
- Data Structures level
  - specify enough detail to analyze the behaviour of our operations and make appropriate decisions as dictated by our problem (e.g. linked list vs. linear list depending on expected use)
- Implementation level
  - high-level implementation detail (e.g. linked list using pointers/references or indirection via indices in an array)

CIS\*2520 - Summer 2006

D. McCaughan

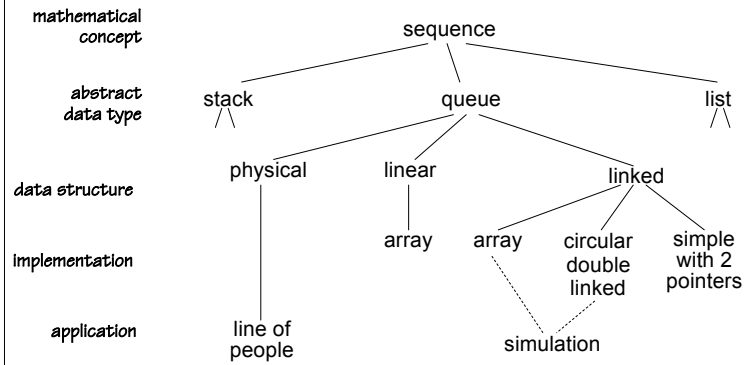
## Refinement of an ADT (cont.)

- Application level
  - low-level implementation detail (e.g. variable names and special requirements/limitations imposed by the application)
- Each level strives to specify enough detail to permit problem solving and reasoning at that level, but ignores unimportant considerations
  - these issues are deferred to the more refined levels below

CIS\*2520 - Summer 2006

D. McCaughan

# Refinement



Adapted from *Data Structures and Program Design in C*, Kruse & Leung

CIS\*2520 - Summer 2006

D. McCaughan