CS 242

Modularity and Object-Oriented Programming

John Mitchell

Reading: Chapter 10 and parts of Chapter 9

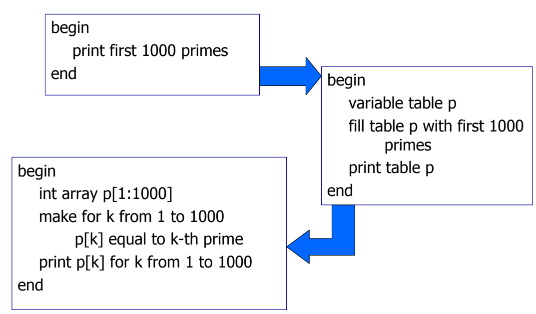
Topics

- Modular program development
  - Step-wise refinement
  - Interface, specification, and implementation
- Language support for modularity
  - Procedural abstraction
  - Abstract data types
    - Representation independence
    - Datatype induction
  - Packages and modules
  - Generic abstractions
    - Functions and modules with type parameters

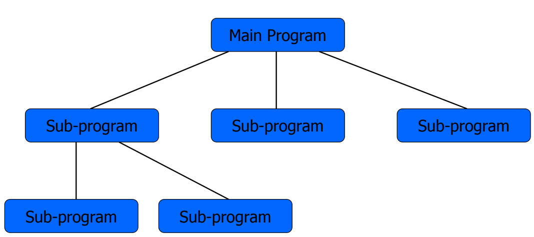
Stepwise Refinement

- Wirth, 1971
  - "... program ... gradually developed in a sequence of refinement steps"
  - In each step, instructions … are decomposed into more detailed instructions.
- Historical reading on web (CS242 Reading page)
  - N. Wirth, Program development by stepwise refinement, *Communications of the ACM,* 1971
  - D. Parnas, On the criteria to be used in decomposing systems into modules, *Comm ACM,* 1972
  - Both *ACM Classics of the Month*

Dijkstra's Example (1969)



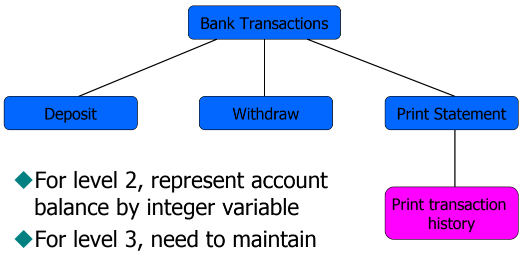
Program Structure



Data Refinement

- Wirth, 1971 again:
  - As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine program and data specifications in parallel

Example



- For level 2, represent account balance by integer variable
- For level 3, need to maintain list of past transactions

Modular program design

- Top-down design
  - Begin with main tasks, successively refine
- Bottom-up design
  - Implement basic concepts, then combine
- Prototyping
  - Build coarse approximation of entire system
  - Successively add functionality

Modularity: Basic Concepts

- Component
  - Meaningful program unit
    - Function, data structure, module, …
- Interface
  - Types and operations defined within a component that are visible outside the component
- Specification
  - Intended behavior of component, expressed as property observable through interface
- Implementation
  - Data structures and functions inside component

Example: Function Component

- Component
  - Function to compute square root
- Interface
  - float sqroot (float x)
- Specification
  - If x>1, then sqrt(x)*sqrt(x) ≈ x.
- Implementation

```
float sqroot (float x){
  float y = x/2; float step=x/4; int i;
  for (i=0; i<20; i++){if ((y*y)<x) y=y+step; else y=y-step; step = step/2;}
  return y;
}
```

Example: Data Type

- Component
  - Priority queue: data structure that returns elements in order of decreasing priority
- Interface
  - Type pq
  - Operations empty : pq
    - insert : elt * pq → pq
    - deletemax : pq → elt * pq
- Specification
  - Insert add to set of stored elements
  - Deletemax returns max elt and pq of remaining elts

Heap sort using library data structure

- Priority queue: structure with three operations
  - empty : pq
  - insert : elt * pq → pq
  - deletemax : pq → elt * pq
- Algorithm using priority queue (heap sort)

```
begin
  empty pq s
  insert each element from array into s
  remove elements in decreasing order and place in array
end
```

This gives us an O(n log n) sorting algorithm (see HW)

Language support for info hiding

- Procedural abstraction
  - Hide functionality in procedure or function
- Data abstraction
  - Hide decision about representation of data structure and implementation of operations
  - Example: priority queue can be binary search tree or partially-sorted array

In procedural languages, refine a procedure or data type by rewriting it. Incremental reuse later with objects.

Abstract Data Types

- Prominent language development of 1970's
- Main ideas:
  - Separate interface from implementation
    - Example:
      - Sets have empty, insert, union, is_member?, …
      - Sets implemented as … linked list …
  - Use type checking to enforce separation
    - Client program only has access to operations in interface
    - Implementation encapsulated inside ADT construct

Modules

- General construct for information hiding
- Two parts
  - Interface:
    A set of names and their types
  - Implementation:
    Declaration for every entry in the interface
    Additional declarations that are hidden
- Examples:
  - Modula modules, Ada packages, ML structures, ...

Modules and Data Abstraction

```
module Set
  interface
    type set
    val empty : set
    fun insert : elt * set -> set
    fun union : set * set -> set
    fun isMember : elt * set -> bool
  implementation
    type set = elt list
    val empty = nil
    fun insert(x, elts) = ...
    fun union(...) = ...
    ...
end Set
```

- Can define ADT
  - Private type
  - Public operations
- More general
  - Several related types and operations
- Some languages
  - Separate interface and implementation
  - One interface can have multiple implementations

Generic Abstractions

- Parameterize modules by types, other modules
- Create general implementations
  - Can be instantiated in many ways
- Language examples:
  - Ada generic packages, C++ templates, ML functors, …
  - ML geometry modules in course reader
  - C++ Standard Template Library (STL) provides extensive examples

C++ Templates

- Type parameterization mechanism
  - template<class T> … indicates type parameter T
  - C++ has class templates and function templates
- Instantiation at link time
  - Separate copy of template generated for each type
  - Why code duplication?
    - Size of local variables in activation record
    - Link to operations on parameter type

Example (discussed in earlier lecture)

- Monomorphic swap function

```
void swap(int& x, int& y){
    int tmp = x;  x = y;  y = tmp;
}
```

- Polymorphic function template

```
template<class T>
void swap(T& x, T& y){
   T tmp = x;  x = y;  y = tmp;
}
```

- Call like ordinary function

```
float a, b;  …   ;  swap(a,b); …
```

Standard Template Library for C++

- Many generic abstractions
  - Polymorphic abstract types and operations
- Useful for many purposes
  - Excellent example of *generic programming*
- Efficient running time (but not always space)
- Written in C++
  - Uses template mechanism and overloading
  - Does *not* rely on objects – No virtual functions

Architect: Alex Stepanov

Main entities in STL

- Container: Collection of typed objects
  - Examples: array, list, associative dictionary, ...
- Iterator: Generalization of pointer or address
- Algorithm
- Adapter: Convert from one form to another
  - Example: produce iterator from updatable container
- Function object: Form of closure ("by hand")
- Allocator: encapsulation of a memory pool
  - Example: GC memory, ref count memory, ...

Example of STL approach

- Function to merge two sorted lists
  - merge : range(s) $\times$ range(t) $\times$ comparison(u) $\to$ range(u)

  This is conceptually right, but not STL syntax.
- Basic concepts used
  - range(s) - ordered "list" of elements of type s, given by pointers to first and last elements
  - comparison(u) - boolean-valued function on type u
  - subtyping - s and t must be subtypes of u

How merge appears in STL

- Ranges represented by iterators
  - iterator is generalization of pointer
  - supports ++ (move to next element)
- Comparison operator is object of class Compare
- Polymorphism expressed using template

```
template < class InputIterator1, class InputIterator2,
              class OutputIterator, class Compare >
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2, InputIterator1 last2,
              OutputIterator result, Compare comp)
```

Comparing STL with other libraries

- C:

```
qsort( (void*)v, N, sizeof(v[0]), compare_int );
```

- C++, using raw C arrays:

```
int v[N];
sort( v, v+N );
```

- C++, using a vector class:

```
vector v(N);
sort( v.begin(), v.end() );
```

Efficiency of STL

◆Running time for sort

	N = 50000	N = 500000
C	1.4215	18.166
C++ (raw arrays)	0.2895	3.844
C++ (vector class)	0.2735	3.802

◆Main point

- Generic abstractions can be convenient and efficient !
- But watch out for code size if using C++ templates…

Boost Lambda Library

◆C++ library providing lambda expressions

◆Example

function expression

for_each(a.begin(), a.end(), std::cout << _1 << ' ');

◆Function expression

- Function arguments _1, _2, _3, … , _9
- Call using usual syntax (std::cout << _1 << ' ')(5)
  - – Too many arguments – OK
  - – Too few arguments – Compile-time error
- Implementation uses overloading
  - – cout << _1 is a function expression because << is overloaded so that if one operand is a function expression, the result is a function expression

Object-oriented programming

◆Primary object-oriented language concepts

- dynamic lookup
- encapsulation
- inheritance
- subtyping

◆Program organization

- Work queue, geometry program, design patterns

◆Comparison

- Objects as closures?

Objects

◆An object consists of

- hidden data
  - instance variables, also called member data
  - hidden functions also possible
- public operations
  - methods or member functions
  - can also have public variables in some languages

hidden data	
msg_1	$method_1$
. . .	. . .
msg_n	$method_n$

◆Object-oriented program:

- Send messages to objects

What’s interesting about this?

◆Universal encapsulation construct

- Data structure
- File system
- Database
- Window
- Integer

◆Metaphor usefully ambiguous

- sequential or concurrent computation
- distributed, sync. or async. communication

Object-Orientation

◆Programming methodology

- organize concepts into objects and classes
- build extensible systems

◆Language concepts

- dynamic lookup
- encapsulation
- subtyping allows extensions of concepts
- inheritance allows reuse of implementation

Dynamic Lookup

- In object-oriented programming,
  object → message (arguments)
  code depends on object and message
- In conventional programming,
  operation (operands)
  meaning of operation is always the same

Fundamental difference between abstract data types and objects

Example

- Add two numbers x → add (y)
  different add if x is integer, complex
- Conventional programming add (x, y)
  function add has fixed meaning

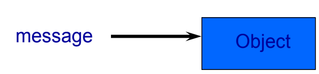
Very important distinction:
Overloading is resolved at compile time,
Dynamic lookup at run time

Language concepts

- "dynamic lookup"
  - different code for different object
  - integer "+" different from real "+"
- encapsulation
- subtyping
- inheritance

Encapsulation

- Builder of a concept has detailed view
- User of a concept has "abstract" view
- Encapsulation separates these two views
  - Implementation code: operate on representation
  - Client code: operate by applying fixed set of operations provided by implementer of abstraction



Comparison with Abstract Data Types

- Traditional (non-OO) approach to encapsulation is through abstract data types
- Advantage
  - Separate interface from implementation
- Disadvantage
  - Not extensible in the way that OOP is

We will look at ADT's example to see what problem is
Better: some HW involving C++ classes w/o virt fctn

Abstract Data Types

- Guarantee invariants of data structure
  - only functions of the data type have access to the internal representation of data
- Limited "reuse"
  - Cannot apply queue code to pqueue, except by explicit parameterization, even though signatures identical
  - Cannot form list of points, colored points
- Data abstraction is important part of OOP, innovation is that it occurs in an *extensible* form

Language concepts

- "Dynamic lookup"
  - different code for different object
  - integer "+" different from real "+"
- Encapsulation
  - Implementer of a concept has detailed view
  - User has "abstract" view
  - Encapsulation separates these two views
- Subtyping
- Inheritance

Subtyping and Inheritance

- Interface
  - The external view of an object
- Subtyping
  - Relation between interfaces
- Implementation
  - The internal representation of an object
- Inheritance
  - Relation between implementations

Object Interfaces

- Interface
  - The messages understood by an object
- Example: point
  - x-coord : returns x-coordinate of a point
  - y-coord : returns y-coordinate of a point
  - move : method for changing location
- The interface of an object is its *type*.

Subtyping

- If interface A contains all of interface B, then A objects can also be used B objects.

Point	Colored_point
x-coord	x-coord
y-coord	y-coord
move	color
	move
	change_color

- Colored_point interface contains Point
  - Colored_point is a *subtype* of Point

Inheritance

- Implementation mechanism
- New objects may be defined by reusing implementations of other objects

Example

```
class Point
    private
        float x, y
    public
        point move (float dx, float dy);
class Colored_point
    private
        float x, y; color c
    public
        point move(float dx, float dy);
        point change_color(color newc);
```

- Subtyping
  - Colored points can be used in place of points
  - Property used by client program
- Inheritance
  - Colored points can be implemented by resuing point implementation
  - Property used by implementor of classes

OO Program Structure

- Group data and functions
- Class
  - Defines behavior of all objects that are instances of the class
- Subtyping
  - Place similar data in related classes
- Inheritance
  - Avoid reimplementing functions that are already defined

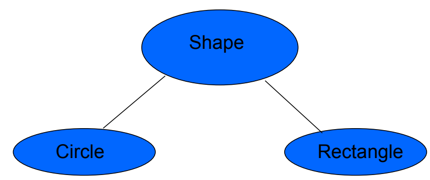
Example: Geometry Library

- Define general concept shape
- Implement two shapes: circle, rectangle
- Functions on implemented shapes
  center, move, rotate, print
- Anticipate additions to library

Shapes

- Interface of every shape must include
  center, move, rotate, print
- Different kinds of shapes are implemented differently
  - Square: four points, representing corners
  - Circle: center point and radius

Subtype hierarchy



- General interface defined in the shape class
- Implementations defined in circle, rectangle
- Extend hierarchy with additional shapes

Code placed in classes

	center	move	rotate	print
Circle	c_center	c_move	c_rotate	c_print
Rectangle	r_center	r_move	r_rotate	r_print

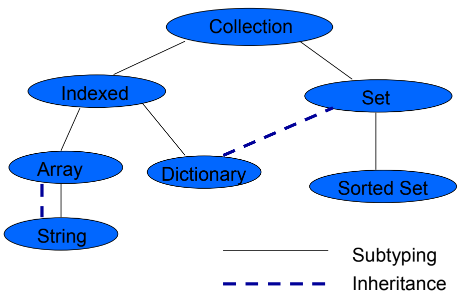
- Dynamic lookup
  - circle → move(x,y) calls function c_move
- Conventional organization
  - Place c_move, r_move in move function

Example use: Processing Loop

Remove shape from work queue
Perform action

Control loop does not know the type of each shape

Subtyping differs from inheritance



Design Patterns

- Classes and objects are useful organizing concepts
- Culture of *design patterns* has developed around object-oriented programming
  - Shows value of OOP for program organization and problem solving

What is a design pattern?

- General solution that has developed from repeatedly addressing similar problems.
- Example: singleton
  - Restrict programs so that only one instance of a class can be created
  - Singleton design pattern provides standard solution
- Not a class template
  - Using most patterns will require some thought
  - Pattern is meant to capture experience in useful form

Standard reference: Gamma, Helm, Johnson, Vlissides

OOP in Conventional Language

- Records provide "dynamic lookup"
- Scoping provides another form of encapsulation

Try object-oriented programming in ML.

Will it work? Let's see what's fundamental to OOP

Dynamic Lookup (again)

receiver → operation (arguments)

code depends on receiver and operation

This is may be achieved in conventional languages using record with function components

Stacks as closures

```
fun create_stack(x) =
  let val store = ref [x] in
    {push = fn (y) =>
          store := y::(!store),
     pop  = fn () =>
          case !store of
             nil => raise Empty |
             y::m => (store := m; y)
    } end;

val stk = create_stack(1);
    stk = {pop=fn,push=fn} : {pop:unit -> int, push:int -> unit}
```

Does this work ???

- Depends on what you mean by "work"
- Provides
  - encapsulation of private data
  - dynamic lookup
- But
  - cannot substitute extended stacks for stacks
  - only weak form of inheritance
    - can add new operations to stack
    - not mutually recursive with old operations

Varieties of OO languages

- class-based languages
  - behavior of object determined by its class
- object-based
  - objects defined directly
- multi-methods
  - operation depends on all operands

This course: class-based languages

History

- Simula 1960's
  - Object concept used in simulation
- Smalltalk 1970's
  - Object-oriented design, systems
- C++ 1980's
  - Adapted Simula ideas to C
- Java 1990's
  - Distributed programming, internet

Next lectures

- Simula and Smalltalk
- C++
- Java

Summary

- Object-oriented design
- Primary object-oriented language concepts
  - dynamic lookup
  - encapsulation
  - inheritance
  - subtyping
- Program organization
  - Work queue, geometry program, design patterns
- Comparison
  - Objects as closures?