Modularity and Object-Oriented Programming

Vitaly Shmatikov

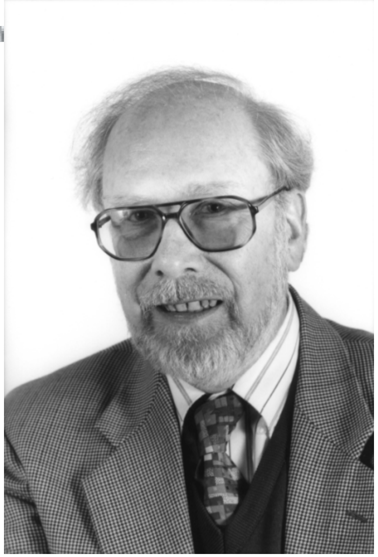
Reading Assignment

- Mitchell, Chapters 9 and 10

Topics

- Modular program development
  - Stepwise refinement
  - Interface, specification, and implementation
- Language support for modularity
  - Procedural abstraction
  - Abstract data types
  - Packages and modules
  - Generic abstractions
    - Functions and modules with type parameters

Stepwise Refinement



- "... program ... gradually developed in a sequence of refinement steps ... In each step, instructions ... are decomposed into more detailed instructions."
  - Niklaus Wirth, 1971

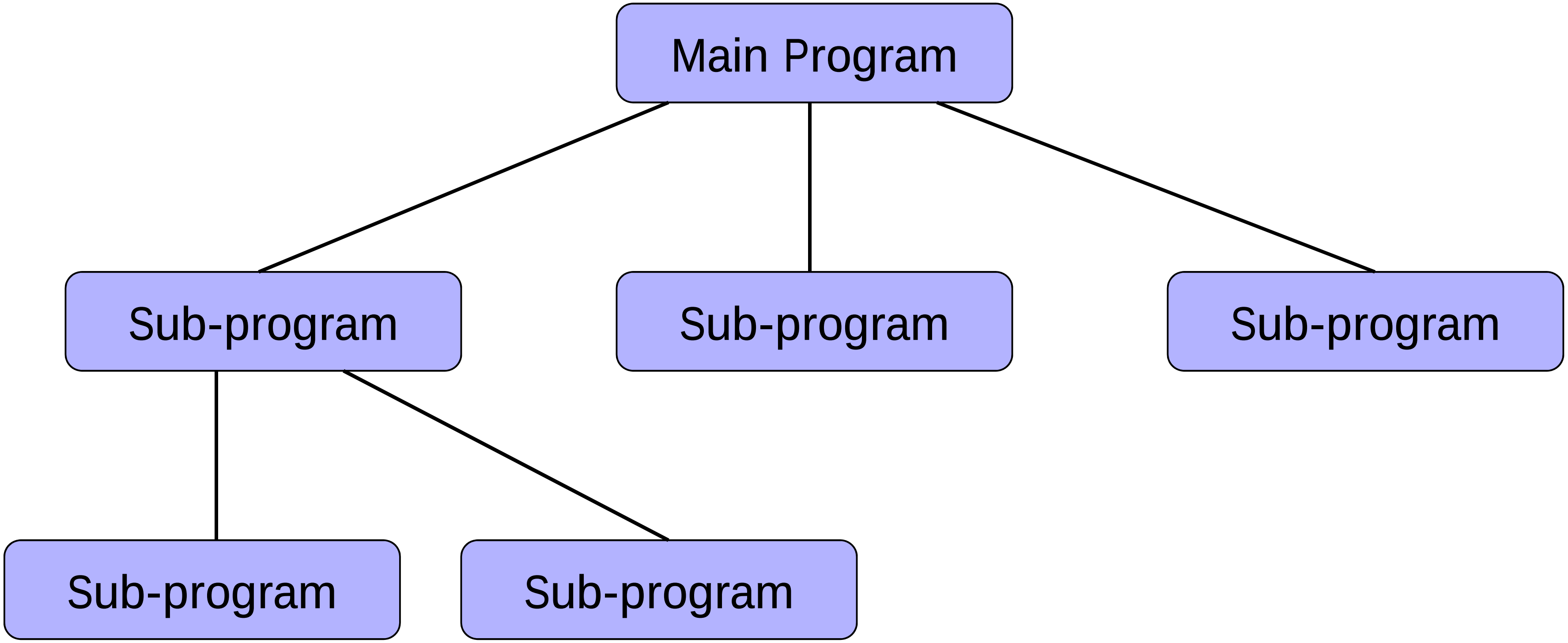
Dijkstra's Example (1969)

```
begin
    print first 1000 primes
end
```

```
begin
    variable table p
    fill table p with first 1000
            primes
    print table p
end
```

```
begin
    int array p[1:1000]
    make for k from 1 to 1000
            p[k] equal to k-th prime
    print p[k] for k from 1 to 1000
end
```

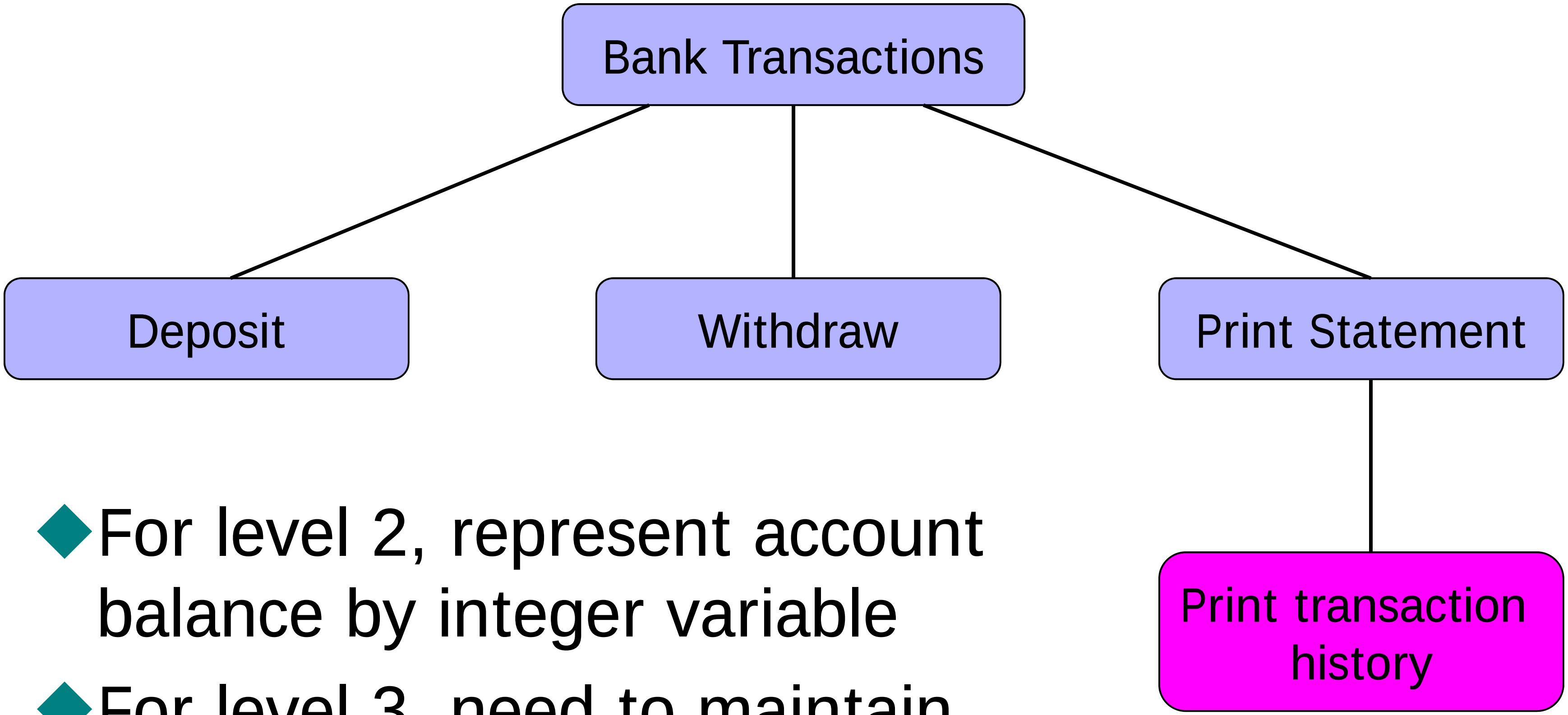
Program Structure



Data Refinement

- "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"
  - Wirth, 1971

Example



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

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 adds to set of stored elements
  - Deletemax returns max elt and pq of remaining elts

Using Priority Queue Data Type

- Priority queue: structure with three operations

```
empty       : pq
insert      : elt * pq → pq
deletemax   : pq → elt * pq
```

- Algorithm using priority queue (heap sort)

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

Abstract Data Types (ADT)

- Prominent language development of 1970s
- Idea 1: Separate interface from implementation
  - Example:

    Sets have operations empty, insert, union,
    is_member?, ...

    Sets are implemented as ... linked list ...
- Idea 2: Use type checking to enforce separation
  - Client program only has access to operations in the interface
  - Implementation <u>encapsulated</u> inside ADT construct

Modules

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

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
- Modules are more general
  - Several related types and operations
- Some languages separate interface & implementation
  - One interface can have multiple implementations

Generic Abstractions

- Parameterize modules by types, other modules
- Create general implementations
  - Can be instantiated in many ways
  - Same implementation for multiple types
- Language examples:
  - Ada generic packages, C++ templates (especially STL – Standard Template Library), ML functors, ...

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
- Remember swap function?
  - See lecture notes on overloading and polymorphism

C++ Standard Template Library

- Many generic abstractions
  - Polymorphic abstract types and operations
  - Excellent example of generic programming
- Efficient running time

  (but not always space)
- Written in C++
  - Uses template mechanism and overloading
  - Does <u>not</u> 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 (concept)
  - merge : range(s) $\times$ range(t) $\times$ comparison(u)
    $\to$ range(u)
  - 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

(This is not STL syntax, but illustrates the concept)

Merge 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)
```

STL vs. “Raw” C and C++

- 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() );
```

Object-Oriented Programming

- Several important language concepts
- Dynamic lookup
- Encapsulation
- Inheritance
- Subtyping

Objects

- An object consists of ...
  - Hidden data
    - Instance variables (member data)
    - Hidden functions also possible
  - Public operations
    - Methods (member functions)
    - Can have public variables in some languages
- Object-oriented program:
  - Send messages to objects

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

<u>Universal</u> encapsulation construct

(can be used for data structures, file systems, databases, windows, etc.)

Dynamic Lookup

- In conventional programming,

  operation (operands)

  meaning of operation is always the same
- In object-oriented programming,

  object → message (arguments)

  code depends on object and message

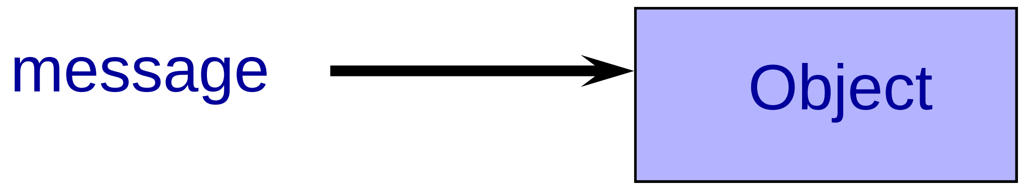
Fundamental difference between
abstract data types and objects!

Overloading vs. Dynamic Lookup

- Conventional programming add (x, y)
  function add has fixed meaning
- Add two numbers x → add (y)
  different add if x is integer, complex
- Similar to overloading, but critical difference: overloading is resolved at compile time, dynamic lookup at run time

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



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

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 as 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

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 reusing point implementation
  - Technique used by implementer of classes

Object-Oriented 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

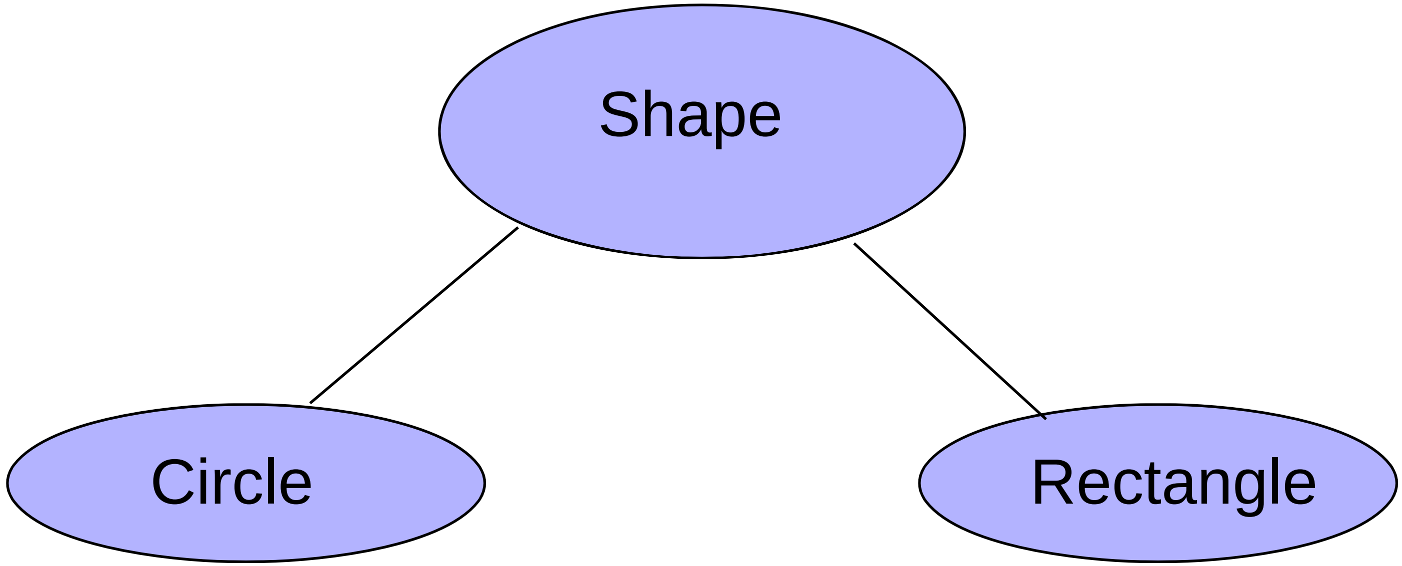
Example: Geometry Library

- Define general concept shape
- Implement two shapes: circle, rectangle
- Functions on 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

Usage Example: Processing Loop

Remove shape from work queue

Perform action

Control loop does not know the type of each shape

Subtyping ≠ Inheritance