Modularity and Object-Oriented Programming. Exceptions.

CS2104 – Lecture 10

Topics

- Modular program development
  - Step-wise 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

- 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*

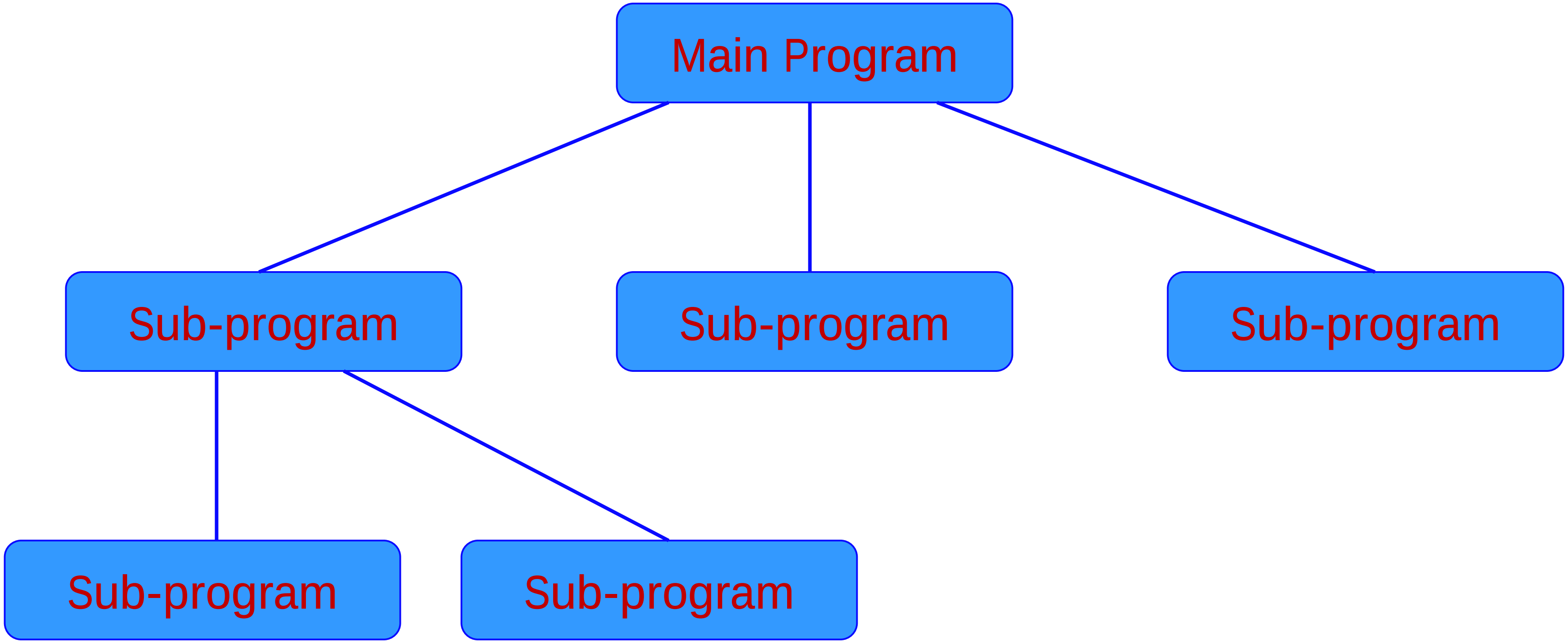
Dijkstra's Example (1969)

```
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
```

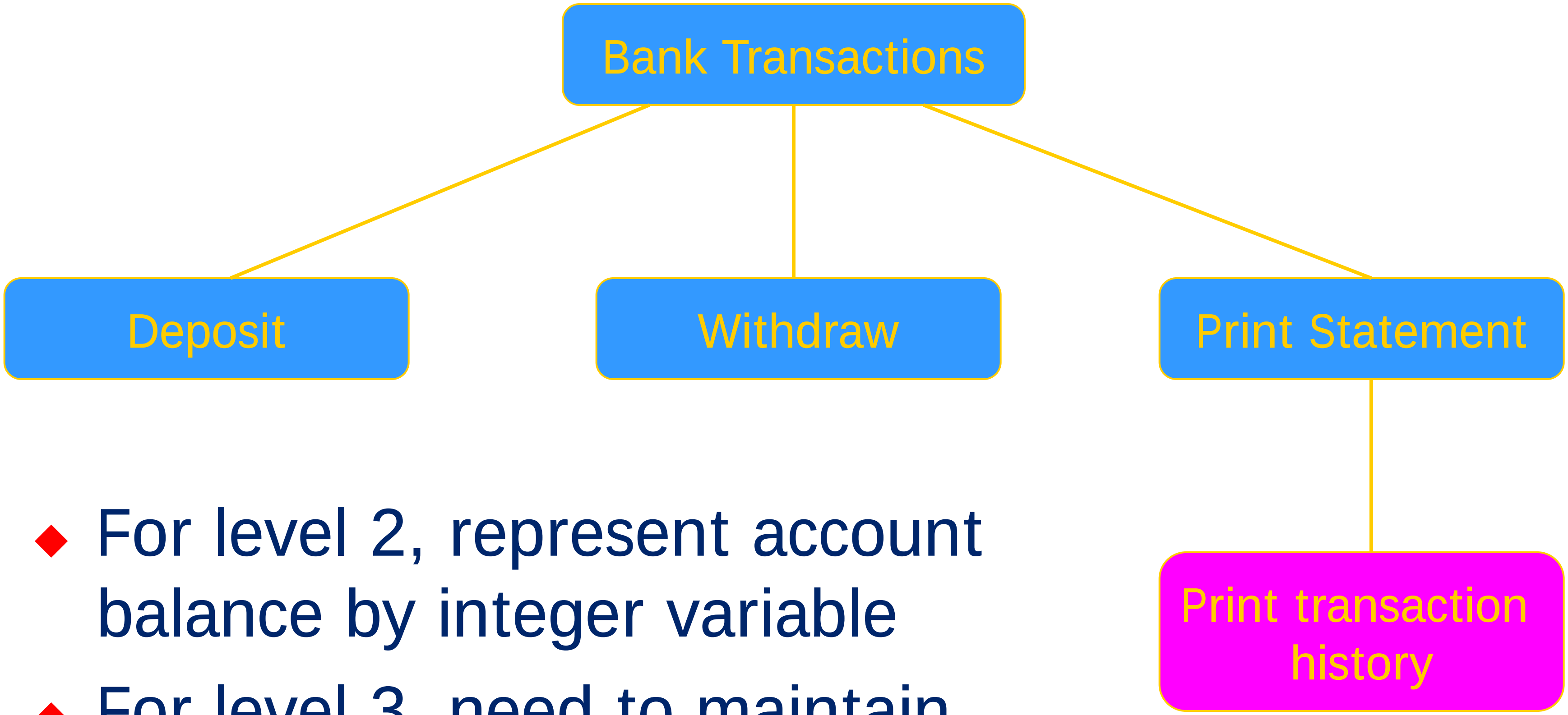
Program Structure



Data Refinement

- 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

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 $\rightarrow$ pq
    deletemax : pq $\rightarrow$ 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 $\rightarrow$ pq

  deletemax : pq $\rightarrow$ elt * pq

- Algorithm using priority queue (heap sort)

  begin

    create 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)

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 book
  - 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

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)

    $\rightarrow$ 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<int> 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...

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 $x \rightarrow$ add (y)
  different add if x is integer, complex

- Conventional programming add (x, y)
  function add has fixed meaning

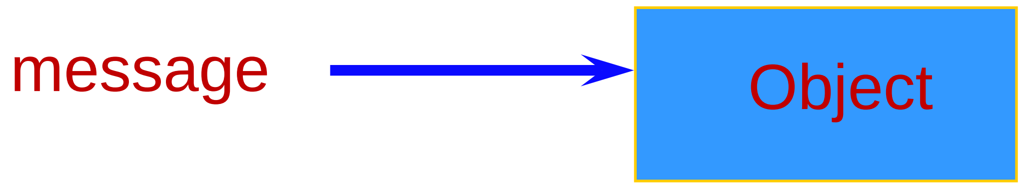
Very important distinction:
Overloading is resolved at compile time,
Dynamic lookup at run time

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



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 contains all of interface , then objects can also be used objects.

Point
- x-coord
- y-coord
- move

Colored_point
- x-coord
- y-coord
- 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 reusing point implementation
  - Technique used by implementer 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

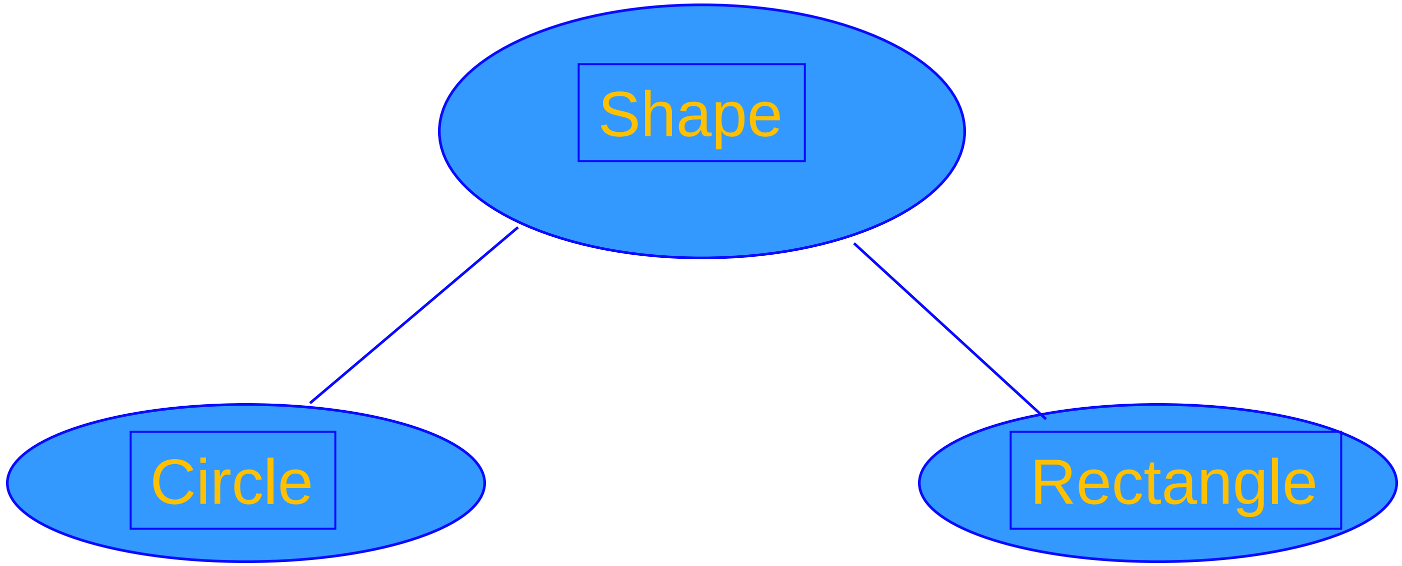
Example: Geometry Library

- 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

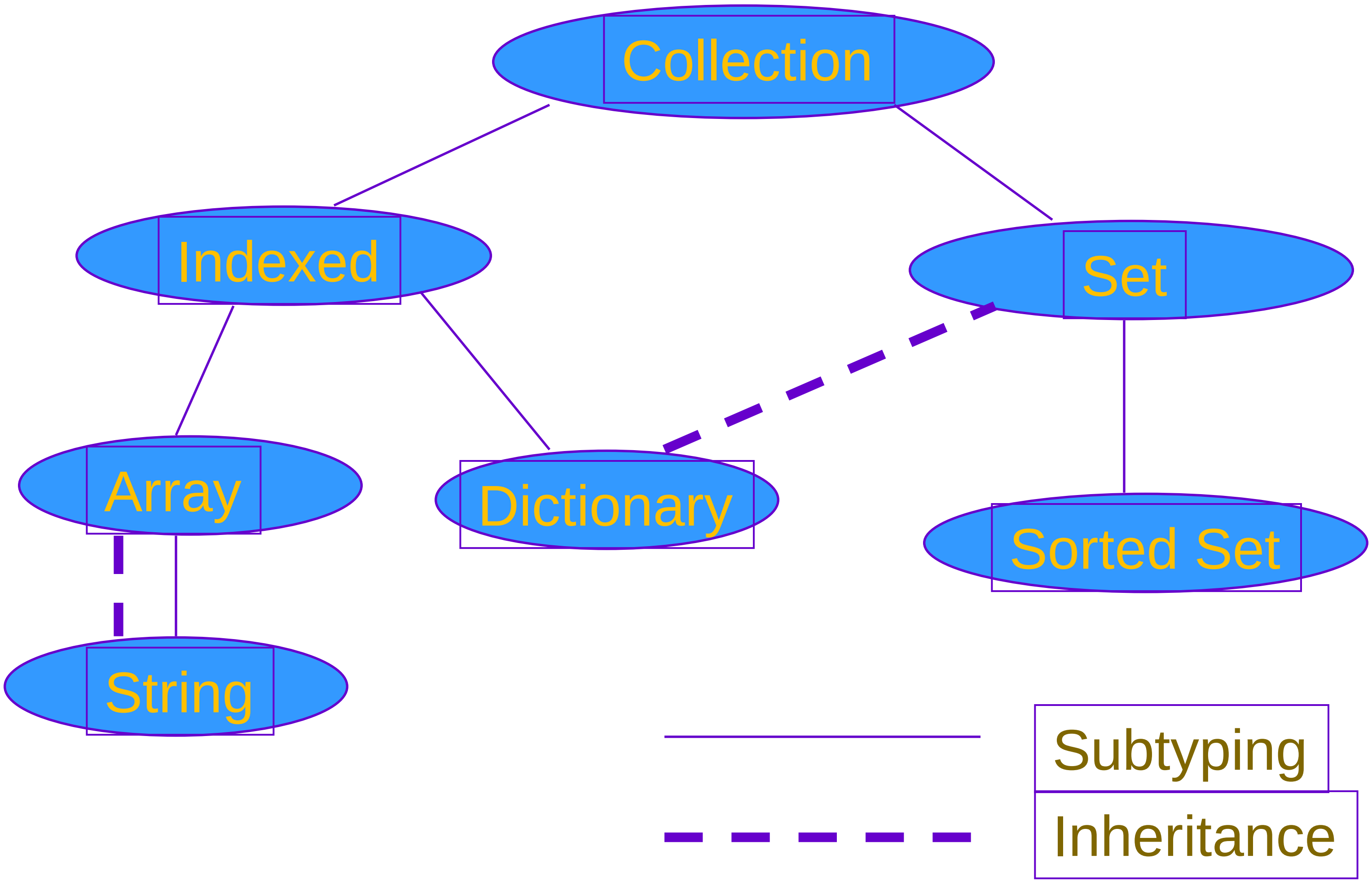
Example use: Processing Loop

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

Example Design Patterns

- Singleton pattern
  - There should only be one object of the given class
- Visitor design pattern
  - Apply an operation to all parts of structure
  - Generalization of maplist, related functions
  - Standard programming solution:
    - Each element classes has *accept* method that takes a visitor object as an argument
    - *Visitor* is interfaced with *visit()* method for each element class
    - The *accept()* method of an element class calls the *visit()* method for its class

Example Code

```
class Car {
   CarElement[] elements;
   public CarElement [] getElements(){ return elements.clone(); }
   public Car() {
      this.elements = new CarElement[] {
      new Wheel("front left"), new Wheel("front right"),
      new Wheel("back left") , new Wheel("back right"),
      new Body(), new Engine()};
    }
}
interface CarElement{
   void accept(Visitor visitor);
}
```

Each relevant class supports interface that accepts visitors

Example Code

```
class Wheel implements CarElement{
    private String name;
    Wheel(String name) { this.name = name; }
    String getName() { return this.name; }
    public void accept(Visitor visitor) { visitor.visit(this); }
}
```

Each accept calls visitor.visit(this), where this has type of class

```
class Engine implements CarElement{
    public void accept(Visitor visitor) { visitor.visit(this); }
}
```

Visitor classes implements visit method for each component class

```
interface Visitor {
    void visit(Wheel wheel);     void visit(Engine engine);
    void visit(Body body);       void visitCar(Car car);
}
class PrintVisitor implements Visitor {
    public void visit(Wheel wheel) {
        System.out.println("Visiting "+ wheel.getName() + " wheel"); }
    public void visit(Engine engine) {
        System.out.println("Visiting engine"); }
    public void visit(Body body) {  ... }
   public void visitCar(Car car) {
         ...  for(CarElement element : car.getElements()) {
                    element.accept(this); }
    ... }
```

Essentially, adds a new “method” to each component class

Visitor classes implements visit method for each component class

```
interface Visitor {
    void visit(Wheel wheel);     void visit(Engine engine);
    void visit(Body body);       void visitCar(Car car);
}
class DoVisitor implements Visitor {
   public void visit(Wheel wheel) {
         System.out.println("Kicking my "+ wheel.getName()); }
   public void visit(Engine engine) {
         System.out.println("Starting my engine"); }
   public void visit(Body body) { …}
   public void visitCar(Car car) {
      System.out.println("\nStarting my car");
      for(CarElement carElement : car.getElements()) {
            carElement.accept(this); } …
  }}
```

An OO Example

```
class Speaker {
  void say(String msg) { System.out.println(msg) ; }
}

class Lecturer extends Speaker {
  void lecture(String msg) {
    say(msg) ;
    say("You should be taking notes") ;
  }
}

class ArrogantLecturer extends Lecturer {
  void say(String msg) { super.say("It is obvious that" + msg) ; }
}

...

(new ArrogantLecturer).lecture("The sky is blue") ;
```

Speaker: C Equivalent

```
struct speaker {
       void (*say)(struct speaker * self, char* msg) ;
} ;

void speaker_say(struct speaker * self, char* msg) {
     printf("%s\n",msg) ;
}

void init_speaker(struct speaker *p) {
       p->say = speaker_say ;
}

struct speaker * make_speaker () {
       struct speaker * retVal = malloc(sizeof(struct speaker));
       init_speaker(retVal) ;
       return retVal ;
}
```

Lecturer: C Equivalent

```
struct lecturer {
       void (*say)(struct speaker * self, char* msg) ;
       void (*lecture)(struct lecturer * self, char* msg) ;
} ;

void lecturer_lecture(struct lecturer * self, char * msg){
     self->say(self,msg) ;
     self->say(self,"You should be taking notes") ;
}

void init_lecturer(struct lecturer *p) {
       init_speaker(p) ;
       p->lecture = lecturer_lecture ;
}

struct lecturer * make_lecturer() {
       struct lecturer * retVal = malloc(sizeof(struct lecturer));
       init_lecturer(retVal) ;
       return retVal ;
}
```

Arrogant Lecturer: C Equivalent

```
struct alecturer {
       void (*say)(struct speaker * self, char* msg) ;
       void (*lecture)(struct lecturer * self, char* msg) ;
       void (*super_say)(struct speaker * self, char* msg) ;
};

void alecturer_say(struct alecturer * self, char * msg){
     char * p = malloc(200) ;
     *p = '\0' ;
     strcat(p,"It is obvious that " ) ;
     strcat(p,msg) ;
     self->super_say(self,p) ;
}

void init_alecturer(struct alecturer *p) {
       init_lecturer(p) ;
       p->super_say = p->say ;
       p->say = alecturer_say ;
}

struct alecturer * make_alecturer() {
       struct alecturer * retVal = malloc(sizeof(struct alecturer));
       init_alecturer(retVal) ;
       return retVal ;
}
```

Discussion

- Each class
  - compiled into a structure
  - data members remain the same
  - methods
    - pointer to function member
    - translated into function
- Each method
  - compiled into a function with a mangled name
  - first argument is pointer to self
  - always called via the corresponding pointer

Exceptions

- Useful for error handling
- Two parts:
  - `try` statement with `catch`/`finally` clauses
  - `throw`/`raise` statement: execution jumps to the `catch` clause that can handle the exception
- Without function calls: similar to a labeled `break`
- With function calls: non-local returns

Exception Example

```
try {
  …
  throw exc ;
  …
} catch (Exc1 e) {
  …
} catch (Exc2 e) {
  …
} finally {
  …
}
```

```
{ struct Exc { exc_type t ; … } E ;
  E.t = no_exception ;
  …
  E.t = exc_type ; // Exception to be thrown
  // Fill E with more relevant data
  goto catch_clauses ; // jump to ``catch``
  …
catch_clauses:
  switch ( E.t ) {
  case exc1_type:
    // catch Exc1 code
    goto finally_clause ;

  case exc2_type:
    // catch Exc2 code
    goto finally_clause ; // may be optimized

  case no_exception:
  finally_clause:
    // finally code
    break ;
  }
}
```

Unchecked Exceptions

- `try` statement saves processor "state", including the *current stack configuration*
- Keep a global variable whose type is a structure to record the currently thrown exception
- `throw` statement
  - fills up global variable with exception details
  - restores "state", performing a long return into the try statement
- global variable helps select correct `catch` clause

Unchecked Exception - Refinement

- `try` statements may be nested
- saved state encoded as element to be placed on *exception stack*
- each `try` pushes current state on exception stack
- `throw`: restores state at top of exception stack
- if restored `try` cannot handle current exception, re-`throw`
- before re-throwing, execute `finally`, if one exists

Example

- C++ code to be translated into C

```
int f() {
  …
  try {
    …
    rg = g() ;
    …
  } catch (E1 e) {
    …
  }
  …
  return … ;
}

int g() {
  …
  rh = h() ;
  …
  return … ;
}
```

```
int h() {
  …
  try {
    …
    E1 e1 ;
    …
    throw e1 ;
    …
    E2 e2 ;
    …
    throw e2 ;
    …
  } catch (E2 e2) {
    …
  } finally {
    …
  }
  …
  return … ;
}
```

`setjmp/longjmp` in C

- `int setjmp(jmp_buf env)`
  - Sets up the local `jmp_buf` buffer and initializes it for the jump.
  - Saves the program's calling environment in the environment buffer `env`.
  - Direct invocation: `setjmp` returns 0.
  - Return from call to `longjmp`: returns nonzero.
- `void longjmp(jmp_buf env, int value)`
  - Restores context of environment buffer `env`.
  - The value specified by value is passed from `longjmp` to `setjmp`.
  - Program execution continues as if the corresponding invocation of `setjmp` had just returned.
  - `value != 0` -> `setjmp` returns 1 ; otherwise returns value.

Example

```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void second(void) {
    printf("second\n");         // prints
    longjmp(buf,1);             // jumps back to where setjmp was called -
making setjmp now return 1
}

void first(void) {
    second();
    printf("first\n");          // does not print
}

int main() {
    if ( ! setjmp(buf) ) {
        first();                // when executed, setjmp returns 0
    } else {                    // when longjmp jumps back, setjmp returns 1
        printf("main\n");       // prints
    }

    return 0;
}
```

Unchecked Exceptions: f

Original code:

```
int f() {
  …
  try {
    …
    g() ;
    …
  } catch (E1 e) {
    …
  }
  …
  return R ;
}
```

g remains the same!

```
extern struct E exception ;
extern jmp_buf push(); //create empty exc slot on stack
extern jmp_buf pop() ; //remove exc slot frm stack
                       // but return old top
int f() {
  …
  if ( !setjmp(push()) ) { // try
    …
    g() ;
    …
    pop();
  } else {
      switch ( exception.T ) {
         case E1 :
           …
           exception.T = NOEXCEPTION ;
           goto finally ;
         default:
         finally:
           … // may be empty
           if ( exception.T != NOEXCEPTION )
             longjmp(pop(),1) ;
      }
  }
  …
  return R ;
}
```

C translation

Unchecked Exceptions: h

Original code:

```
int h() {
  …
  try {
    …
    E1 e1 ;
    …
    throw e1 ;
    …
    E2 e2 ;
    …
    throw e2 ;
    …
  } catch (E2 e2) {
    …
  } finally {
    …
  }
  …
  return R ;
}
```

C translation

```
int h() {
  …
  if (setjmp(push())) {
    …
    exception.T = E1 ;
    exception.V = e1 ;
    longjmp(pop(),1) ;
    …
    exception.T = E2 ;
    exception.V = e2 ;
    longjmp(pop(),1) ;
    …
    pop();
  } else {
      switch ( exception.T ) {
         case E2 : // handle E2
           …
           exception.T = NOEXCEPTION ;
           goto finally ;
         default:
         finally:
           …
           if ( exception.T != NOEXCEPTION )
             longjmp(pop(),1) ;
      }
  }
  …
  return R ;
}
```