Type Systems and Object-Oriented Programming (II)

John C. Mitchell
Stanford University

Outline

- ✓ Foundations; type-theoretic framework
- Principles of object-oriented programming
- Decomposition of OOP into parts
- Formal models of objects

Goals

- Understand constituents of object-oriented programming
- Insight may be useful in software design
- Trade-offs in program structure
- Possible research opportunities
  - » language design
  - » formal methods
  - » system development, reliability, security

Object-Oriented Programming

- An object consists of
  - » hidden data
  - » public operations

Hidden data	
msg 1	method 1
...	...
msg n	method n

- Program sends 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-oriented programming

- Programming methodology
  - » organize concepts into objects and classes
  - » build extensible systems
- Language concepts
  - » encapsulate data and functions into objects
  - » subtyping allows extensions of data types
  - » inheritance allows reuse of implementation

Incremental Methodology [Booch]

- Identify the objects at a given level of abstraction.
- Identify the semantics (intended behavior) of these objects.
- Identify the relationships among the objects.
- Implement these objects.

This Method

- An iterative process
- Based on associating objects with components or concepts in a system.
- Why iterative?
  - » An object is typically implemented using a number of constituent objects
  - » Apply same methodology to subsystems, underlying concpets

Compare to top-down programming:

- Similarity:
  - » a procedure is typically implemented by a number of finer-grained procedures.
- Difference:
  - » both functionality and data representation may be refined
  - » *working* OO program can be refined incrementally (prototyping)

Three Program Examples

- Generic object example
- Dijkstra top-down example
- Conventional vs. class structure for geometry program

Generic Example: Work Queue

- Remove task from work queue
- Process task:
  - » Perform specific operation
  - » Possibly place more tasks on queue
- Instances of this pattern
  - » event-based simulation
  - » process bank transactions
  - » print queue

Why Objects?

- Each task has specific operation
  - » perhaps additional operations
- Implementations may differ
  - » but this is hidden
- Commonality is used to advantage
  - » send same message to each object
  - » object implements this appropriately

Step-wise refinement

- Compose program in steps [Dijkstra 1969]
- In illustrative examples, data structures are simple, remaining invariant.
- In more complex systems, necessary to refine data structures.

"A first example ..." [Dijkstra 1969]

- begin
  print first thousand prime numbers
  end
- begin variable *table p*
  fill table p with first thousand primes
  print table p
  end

"A first example ..."

begin integer array p[1:1000]
make for k from 1 through 1000
p[k] equal to the kth prime number
print p[k] for k from 1 through 1000
end

At this point, the data structure that is common to both tasks has been determined.

What has changed since 1969?

- Program size and complexity
  - » "What I am really concerned about is the composition of large programs ... the same size as the whole of this chapter."

  (80 pages x 40 lines/page = 3200 lines)
- Lifespan of software systems
- Range of applications
  - » more complex systems, concepts

Geometry library

- Define and implement points
- Define structure of shapes
- Implement two shapes: circle, rectangle
- Functions on implemented shapes
  center, move, rotate, print
- Anticipate additions to library

Compare program structures

- Type-case ("function-oriented")
  - » data represented by C struct
  - » functions branch according to type of data
- Object-oriented
  - » data represented by C++ objects
  - » functions are members
  - » branch according to type is implicit

Points (Typecase version)

```
struct Pt { float x;  float y; };

struct Pt* newPt(float xval, float yval) {
  struct Pt* p = (struct Pt *)malloc(sizeof(struct Pt));
  p->x = xval;  p->y = yval;  return p; };

struct Pt* copyPt(struct Pt* p) {
  struct Pt* q = (struct Pt *)malloc(sizeof(struct Pt));
  q->x = p->x;   q->y = p->y; return q; };
```

Shape concept

```
enum ShapeTag {Circle, Rectangle};
struct Shape {
  enum ShapeTag tag;
};
```

First field of each shape stores kind of shape, represented using enum type

Circle structure

```
struct Circle {
  enum ShapeTag     tag;
           struct Pt*     center;
               float       radius;
};
```

Tag is common to all shapes

Data is specific to circles

Constructor and destructor

```
struct Circle* newCircle(struct Pt* cp, float r) {
  struct Circle* c = (struct Circle*)malloc(...Circle);
  c->center=copyPt(cp);  c->radius=r;  c->tag=Circle;
  return c; };

void deleteCircle(struct Circle* c) {
  free (c->center); free (c); }
```

Rectangle structure

```
struct Rect {
  enum ShapeTag  tag;
  struct Pt*     topleft;
  struct Pt*     botright;
};
```

Tag is common to all shapes

Data is specific to rectangles

Constructor and Destructor

```
struct Rect* newRect(struct Pt* tl, struct Pt* br)  {
  struct Rect * r = (struct Rect*)malloc(... Rect );
  r->topleft=copyPt(tl);  r->botright=copyPt(br);
  r->tag=Rect;  return r;   }

void deleteRect(struct Rect* r) {
  free (r->topleft);
  free (r->botright);
  free (r);   };
```

Center function

```
struct Pt* center (struct Shape* s) {
  switch (s->tag) {                          /* test tag  */
   case Circle: {
     struct Circle* c = (struct Circle*) s;  /* type cast */
     return copyPt(c->center);  };
   case Rect: {
     struct Rect * r = (struct Rect *) s;    /* type cast */
     return newPt((r->botright->x -  ... );
   };};};
```

Center function

- Must assume that the type tags are set correctly
- This cannot be detected at compile time

Move function

```
void move (struct Shape* s,float dx, float dy) {
  switch (s->tag) {
   case Circle: {
       struct Circle* c = (struct Circle*) s;
       c->center->x  +=  dx;    c->center->y  += dy;
   }; break;
   case Rectangle:  {
       ...    }; }; };
```

Same switch and casts as center function

Rotate function

```
void rotate (struct Shape* s) {
  switch (s->tag) {
   case Circle:   break;
   case Rect: {
       struct Rect * r = (struct Rect *)s;
       float d = ((r->botright->x - ... ) ... };
    break; }; };
```

Same switch and casts

Print function

```
void print (struct Shape* s) {
  switch (s->tag) {
   case Circle: {
       struct Circle* c = (struct Circle*) s;
       printf("circle at ..." ... c->center->x  ... )};
   break;
   case Rectangle: { ...  };
   break; };
```

Same switch and casts

Test Program

```
struct Pt* origin = newPt(0,0);  ...
struct Shape* s1 =(struct Shape*)newCircle(origin,2);
struct Shape* s2 =(struct Shape*)newRect(p1,p2);
  print(s1);    print(s2);    rotate(s1); rotate(s2);
  move(s1,1,1); move(s2,1,1); print(s1);  print(s2);
  deleteCircle((struct Circle*)s1);
  deleteRect((struct Rect *)s2);
  free(origin); free(p1); free(p2);
```

Need explicit casts for uniform program

Object-oriented shapes

```
class Pt {
 public:
  Pt (float xval, float yval) {
       x = xval;  y=yval; };
  Pt (Pt* p) {
       x = p->x;  y = p->y; };
  float x;
  float y;   };
```

Overloaded constructor for new and copy

In-line functions for readability

Shape concept

```
class Shape {
public:
  virtual Pt*     center()                    =0;
  virtual void   move(float dx, float dy)  =0;
  virtual void   rotate()                    =0;
  virtual void   print()                     =0;
};
```

Pure abstract base class

- Defines interface for derived classes
- Cannot create shape objects
- Can write functions that expect shape objects as arguments

Circle class

```
class Circle : public Shape {
 public:
   Circle(Pt* cn, float r) {                    /* constructor */
      center_ = new Pt(cn);  radius_ = r;   };

   virtual ~Circle()  { delete center_;  };  /* destructor */

   virtual Pt* center() { return new Pt(center_);  };
```

Circle class (cont'd)

```
void move(float dx, float dy) {
     center_->x += dx; center_->y += dy;  };
 void rotate() { };
 void print() {
        printf("circle at  ... ", ... center_->x, ); };

private:                    /* private data  */
   Pt*   center_;
   float radius_;
};
```

Rectangle class

```
class Rect : public Shape {
 public:
   Rect(Pt* tl, Pt* br) {
      topleft_  = new Pt(tl);
      botright_ = new Pt(br);     };
   virtual ~Rect() {
      delete topleft_;
      delete botright_;     };
```

Rectangle class (cont'd)

```
Pt* center() {
  return new Pt((botright_->x - ...,... );        };
void move(float dx,float dy) {
  topleft_->x  += dx;  topleft_->y  += dy; ...   };
void rotate() {
  float d = ((botright_->x - topleft_->x) -
             (topleft_->y - botright_->y))/2.0;
  topleft_->x  += d;  topleft_->y  += d;  .. .    };
```

Rectangle class (cont'd)

```
void print () {
  printf("rectangle coordinates  %.1f  ... \n",
         topleft_->x, topleft_->y,
         botright_->x, botright_->y);    };

 private:                  /* private data  */
  Pt* topleft_;
  Pt* botright_;
};
```

Test Program

```
Pt* origin = new Pt(0,0);   Pt* p1     =    ...
Shape* s1 = new Circle(origin, 2 );
Shape* s2 = new Rectangle(p1, p2);
   s1->print(); s2->print(); s1->rotate();   ...
   s1->move(1,1);  s2->move(1,1);  ...
delete s1;  delete s2;   delete origin;
delete p1;  delete p2;
```

Subtyping eliminates explicit conversion

Only *shape* operations;   no type case or casts

Criteria for Comparison

- Extensibility
- Encapsulation
- Type-checking / static analsysis
- Correspondence between structure of problem and structure of program
- Efficiency

Program Organization

class	function			
	center	move	rotate	print
circle	c_center	c_move	c_rotate	c_print
rect	r_center	r_move	r_rotate	r_print

Object-oriented: arrange by row

Function-oriented: arrange by column

Comparison

- Function-oriented
  - » easier to add a new operation
    - – code for new operation all goes in one place
- Object-oriented
  - » easier to add a new shape
    - – code for new shape all goes in one place
  - » can also add function that is not a method

Encapsulation

- object oriented
  - » representation of shape is hidden
  - » functions have access
    - – to shape they are associated with
    - – not to data of other shapes
- function oriented
  - » all data manipulated by functions must be publicly accessible

Static checking

- object oriented
  - » statically-typed in C++
- function oriented
  - » cannot be statically type-checked in C
    - – no guarantee that tag is actually the type
  - » could be in language with typecase
    - – need to test type of an object, not data field

Example: Simula *inspect*

```
class A;  A class B;      /*   B is a subclass of A   */
ref (A) a;
inspect a
      when B do   ...   /*   subclass B operations*/
      otherwise   ...   /*  superclass A operations */
```

This form is type safe.
Why do we seem to need this in an OOL?

Taxonomic hierarchy

- Many systems are hierarchical
  - » Animals: Phyla, genus, species
  - » Banks: different kinds of accounts
  - » Geometric shapes
- Object-oriented
  - » try to capture structure in class hierarchy

Function/typecase-oriented

- hierarchy is used in an ad hoc manner
- want extensible union types
  - » circle, rectangle coded with same field in first location.
  - » hack for tagged union
  - » could be avoided with safe disjoint union

Complexity

- Function/typecase
  - » space cost of tag field
  - » time cost,in each function, of branching according to type.
- Object oriented
  - » hidden class or *vtable* pointer per object
  - » one extra indirection in method invocation
  - » (for optimized C++; Smalltalk could be less efficient)

Comparison

- Extensibility — Tradeoff
  - » Objects: easier to add new kinds of data
  - » Functions: new functions onexisting data
- Encapsulation — Objects
- Type-checking — Objects
- System hierarchy — Objects
- Efficiency — Same

Principles of OO Design

- Open/Closed Principle
  - » Software entities should be
    - – open for extension
    - – closed for modification
- Principle of Dependency Inversion
  - » Details should depend upon abstractions
  - » Abstractions should not depend on details
- ... many others ...

Computer "Science" terms exposed

Structured programming = slow
Modular = bloated
Extensible = late
Reusable = buggy
Object-Oriented =
slow and bloated and late and buggy

[Haeberli and Karsh]

Haeberli and Karsh, cont'd

Top-Down Design = hard to use
Standard Compliant = slow and late and hard to use
Automatic = manual
Configurable = unfinished
Fully Configurable = completely unfinished
Type Safe = imponderable
Hack = useful
Little Hack = very useful