Object-Oriented Programming: An Operational Derivation

- If OOP weren't around, we'll invent it now.
- A design-driven approach to the derivation of OOP
  - starting from procedural design, what are the steps to arrive at OOP?
  - What motivates these steps?
- Show OOP as a uniform design method
  - OOP starts off as a design itself on top of procedural design.
  - It gets absorbed into the culture and begins being presented as a paradigm shift
  - Subsequent designs use OOP as the starting point

Stages

1. data structures
2. structured data
3. modular data hiding
4. object data hiding
   4.a. applied uniformly (an "aha!")
5. object references
6. object genericity
7. inheritance
8. polymorphism
   8.b. vtbls
   8.c. universal virtuals
9. Run-time type checking
10. Introspection

Stage 1: Data Structures

- The interface defines the data structures so that client code can use these data structures.

```
person.h:
#define MAX_PERSONS 100
extern char[] person_name[];
extern int person_age[];
extern int persons_num;
```

06 - OOP CSC407 3

Stage 1: Implementation

- The implementation is used to physically allocate the storage space in memory required to implement the data structures.

```
person.c:
#include "person.h"
char[30] person_name[MAX_PERSONS];
int person_age[MAX_PERSONS];
int persons_num = 0;
```

06 - OOP CSC407 4

Stage 1: Client

- Code that uses the data structure is scattered throughout the implementation (recall KWIC).

```
client.c:
#include “person.h”
void client() {
    strcpy(person_name[persons_num], “David”);
    person_age[persons_num] = 39;
    persons_num++;
}
```


Stage 1: Pros and Cons

- Pros
  - Grouped together into one place the interface and storage declarations for a data structure.
    - One place to go to see how a data structure is defined.
- Cons
  - Changes to the data structures required changes to code in many places.
    - Many requirements changes require changes to data structures
    - Therefore requirements changes require that we change a lot of code.
    - Changes are all tricky to make because they rely on detailed knowledge of how the data structures should be used

Stage 2: Structured Data

- Improvements on the interface to allow grouping of data into “structures”.

person.h:

```
#define MAX_PERSONS 100
typedef struct {
    char[30] name;
    int age;
} Person;
extern Person person[];
extern int persons_num;
```


Stage 2: Client

- Client code uses the data abstraction

client.c:

```
#include “person.h”
void client() {
    Person *pp = getPerson();
    memcpy(person[persons_num],pp,sizeof(Person));
    persons_num++;
}
```

Stage 2: Pros and Cons

- Pros
  - Can begin dealing with a domain-side abstraction directly.
  - Language-supported naming and grouping.
- Cons
  - Operations on data and data itself are still distinct.
  - Internal structure of data remains exposed for those desirous of by-passing the interface.
  - Conventional naming required

Stage 3: Modular Data Hiding

- Interface specifies operation on the data rather than showing the data itself.

person.h:

```
#define OK 0
#define BAD_NAME 1
#define BAD_AGE 2
#define OUT_OF_MEMORY 3
extern int addPerson(char* name, int age);
extern int numPersons();
extern char* namePerson(int p);
extern int agePerson(int p);
```

Stage 3: Implementation

```
person.c:

#define MAX_PERSONS 100
typedef struct {
    char[30] name;
    int age;
} Person;
static Person person[MAX_PERSONS];
static int persons_num = 0;

int agePerson(int p) {
    return person[p].age;
}

…
```


Stage 3: Client

- Client code uses the defined interface.

```
client.c:

#include "person.h"

void client() {

    int status = addPerson("David", 39);

}
```

Stage 3: Pros and Cons

- Pros
  - Hides the details of the data structure from clients.
  - Provides a small set of well-defined interfaces.
  - Isolates changes.
  - Prohibits any but allowed access
- Cons
  - Naming is conventional only
  - Singleton orientation.
  - (Parnas "Information Hiding")

Stage 4: Object Data Hiding

- Abstract and conventionalize the notion of data and operations on it.

person.h:

```
typedef struct {
    char[30] name;
    int age;
} Person;
extern Person* person_create(char* name, int age);
extern char* person_getName(Person*);
extern int person_getAge(Person*);
extern void person_delete(Person*);
```

Stage 4: Interface

- See that “Person” and “ListOfPerson” are actually different things.

```
personList.h:
#include “person.h”
extern int addPerson(Person*);
extern int numPersons();
extern Person* getPerson(int p);
```


Stage 4a: Objects Everywhere

- Realize there is no reason not to model all data/operations uniformly in the same manner.

```
personList.h:
#include “person.h”
typedef struct {
    int numPersons;
    Person* personList[];
} PersonList;
extern PersonList* personList_create(int maxNumPeople);
extern int personList_addPerson(PersonList*, Person*);
extern int personList_getNumPersons(PersonList*);
extern Person* personList_getPerson(PersonList*);
extern void personList_destroy(PersonList*);
```

Stage 4: A Pattern Emerges

X.h:

```
#include …other class interface files…
typedef struct {
    …data definitons…
} X;
extern X* X_create( …creation arguments… );
extern type X_op1(X*, …op1args… );
extern type X_op2(X*, …op2args… );
…
extern void X_destroy(X*);
```


Stage 4: Pros and Cons

- Pros:
  - A universal, conventional way of dealing with data/operations
  - Hides implementation details
  - uses well-defined interfaces
  - data creation/deletion handled uniformly
  - Allows the application of OOA to problem solving
- Cons:
  - relies on convention
  - details of data layout remain visible

Stage 5: Object References

- Make the data structures entirely opaque.
- Handle all objects uniformly as references

person.h:

```
typedef void* Person;
extern Person person_create(char* name, int age);
extern char* person_getName(Person);
extern int person_getAge(Person);
extern void person_delete(Person);
```


Stage 5: Implementation

- Make the data structures entirely opaque, handle all objects uniformly as references

person.c:

```
#include "person.h"
typedef struct {
    char[30] name;
    int age;
} _Person;
Person person_create(char* name, int age) {
    _Person *p = (_Person*)malloc(sizeof(_Person));
    strcpy(p->name, name);
    p->age = age;
    return (void*)p;
}
int person_getAge(Person p) {return ((_Person*)p)->age;}
```

Stage 5: Pros and Cons

- Pros:
  - No (straightforward/natural/safe) way for clients of the class to get around the interfaces provided.
    - In practice: no subversion of the interface
  - An entirely uniform approach to memory management.
  - All objects are the same size (a void* pointer size).

- Cons:
  - relies on convention
  - language allows a Person to be passed as a Shape
    - both are void* and hence in C are compatible types.
    - error prone & confusing errors at that
  - no notion of inheritance/polymorphism

Stage 6: Genericity

- Reuse the same code for different types of data
- Relies on the fact that all data is represented uniformly as void* pointers.

List.h:

```
typedef void* List;
extern List list_create(int maxItems);
extern void list_addItem(List, void*);
extern int list_getNum(List);
extern void* list_getItem(int n);
extern void list_delete(List);
```

Stage 6: Pros and Cons

- Pros
  - Allows for reuse of code
    - don’t need to write the same list code more than once.
- Cons
  - Can’t specify the frequent case:
    - a homogeneous list of one thing
    - either at declaration time for documentation and compile-time checking purposes
    - or even at run-time for run-time checking
  - Dangerous to make heterogeneous lists
    - can’t distinguish the object types easily
      - can’t perform different operations to different types.

Compile-Time versus Run-Time Checking

- Principle
  - You will only find some fraction of the errors in your code
    - compiler checks
    - code reviews
    - testing
  - the rest will make it past your QA controls and into the field
- Implication 1:
  - The more errors you can eliminate at compile time, the fewer will make it out into the field.
  - If an error that could be caught at compile time is left in the run-time code, it may make it past testing
- Implication 2:
  - if an error that could be caught with a run-time assertion at testing-time is not caught, it could make it all the way into the field.

Compile-Time versus Run-Time

- In practice:
  - Compile-time is best
  - Run-Time assertions are not far behind
  - Lots of compile-time checks wind up complicating an implementation.
  - Many programmers will attempt to avoid them by not using the language facilities properly.
    - You wind up having no compile-time checks AND no run-time assertion checks
  - Templates (Generic programming) (won't cover)
  - Run-time type checks (will discuss)

Stage 7: Inheritance

Shape.h:

```
typedef void* Shape;
extern void Shape_move(Shape, float dx,dy);
```

Circle.h:

```
typedef void* Circle;
extern Circle Circle_create(float x,y,r);
extern void Circle_move(Circle c, float dx,dy);
```

Rectangle.h:

```
typedef void* Rectangle;
extern Rectangle Rectangle_create(float x,y,h,w);
extern void Rectangle_move(Rectangle r, float dx,dy);
```

Stage 7: Implementation

```
_Shape.h:
#include "Shape.h"
typedef struct {
    float x;
    float y;
} _Shape;
extern void _Shape_init (Shape s, float,x,y);
```


Stage 7: Implementation

```
Shape.c:
#include "_Shape.h"
void _Shape_init (Shape s, float x,y) {
  _Shape* sp = (_Shape*)s;
  sp->x = x;
  sp->y = y;
}
void Shape_move(Shape s, float x,y) {
    _Shape* sp = (_Shape*)s;
    sp->x += dx;
    sp->y += dy;
}
```

Stage 7: Implementation

_Circle.h:

```
#include "_Shape.h"
#include "Circle.h"
typedef struct {
    _Shape shape;
    float r;
} _Circle;
extern void _Circle_init(Circle, float x,y,r);
```


Stage 7: Implementation

Circle.c:

```
#include "_Circle.h"
void _Circle_init(Circle c, float x,y,r) {
    _Shape_init((Shape)c,x,y);
    _Circle* cp = (_Circle*)c;
    cp->r = r;
}
Circle Circle_create(float x,y,r) {
      Circle c = malloc(sizeof(_Circle));
      _Circle_init(c,x,y,r);
      return c;
}
void Circle_move(Circle c, float dx,dy) {
    Shape_move((Shape)c, dx,dy);
}
```

Stage 7: Client

```
client.c:
#include "List.h"
#include "Circle.h"
#include "Rectangle.h"
void client() {
    List shapes = List_create(100);
    List_addItem(shapes, Circle_create(3,3, 1.5));
    List_addItem(shapes, Rectangle_create(0,0,1,1));
    for(int i = 0; i < List_getNum(shapes); i++) {
        Shape s = (Shape) List_getItem(shapes,i);
        Shape_move(s,5,5);
    }
}
```


Stage 7: Pros and Cons

- Pros
  - Can now factor common operations into a baseclass.
  - Code reuse
    - only need to make sure it works once
    - when it works, it works for everything
- Cons
  - No way to distinguish the items in the list
    - Suppose we wanted to `move()` only the circles?
  - No way to apply the same conceptual operation, but implemented differently, to different types in the list
    - Suppose we wanted to implement
      - `Shape_grow(Shape s, float growth_factor)`

Stage 7: Implementation recap

- Header file: X.h
  - defines the public interface for objects of class X
    - generic type
    - creation/operatios/destruction interfaces
- Protected header file: _X.h
  - defines the protected interface for objects of class X
  - classes that inherit from X must have access to this header
  - includes protected headers of sub-classes
  - implementation must have access to this header
    - declares storage layout
      - subsumes subclass storage
    - declares protected initialization (and destruction) routines
- Private implementation file: X.c
  - implements all the operations
  - performs storage allocation
    - separation of storage allocation and initialization
      - required for inheritance

Stage 8: Polymorphism

- Go back to the question:
  - How do we implement a common interface differently in different subclasses?

Stage 8: Polymorphism: Common Interface

Shape.h:

```
typedef void* Shape;
extern void Shape_move(Shape s, float dx,dy);
extern void Shape_grow(Shape s, float f);
```

Circle.h:

```
typedef void* Circle;
extern Circle Circle_create(float x,y,r);
extern void Circle_move(Circle c, float dx,dy);
extern void Circle_grow(Circle c, float f);
```

Rectangle.h:

```
typedef void* Rectangle;
extern Rectangle Rectangle_create(float x,y,h,w);
extern void Rectangle_move(Rectangle r, float dx,dy);
extern void Rectangle_grow(Rectangle r, float f);
```


Stage 8: Distinct Implementations

Circle.c:

```
…
void Circle_grow(Circle c, float f) {
    _Circle* cp = (_Circle*)c;
    cp->r *= f;
}
```

Rectangle.c:

```
…
void Rectangle_grow(Rectangle r, float f) {
    _Rectangle* rp = (_Rectangle*)r;
    rp->w *= f;
    rp->h *= f;
}
```

Stage 8: Client Code

client.c:

```
#include "List.h"
#include "Circle.h"
#include "Rectangle.h"

void client() {
    List shapes = List_create(100);

    List_addItem(shapes, Circle_create(3,3, 1.5));

    List_addItem(shapes, Rectangle_create(0,0,1,1));

    for(int i = 0; i < List_getNum(shapes); i++) {
        Shape s = (Shape) List_getItem(shapes,i);
        Shape_grow(s,2.0);
    }
}
```

How do we implement `Shape_grow()`?

Stage 8: Simple, but Inefficient Approach

_Shape.h:

```
typedef struct {
    float x,y;
    void (*grow_fp)(float,float);
} _Shape;
```

Shape.c:

```
void _Shape_init(Shape s, float x,y,
void(*f)(float,float)){
    _Shape* sp = (_Shape*)s;
    sp->x = x; sp->y = y; sp->grow_fp = f
}
```

Circle.c:

```
…
void _Circle_init(Circle c, float x,y,r) {
    _Circle* cp = (_Circle*)c;
    _Shape_init((Shape)c, x,y, &Circle_grow);
    cp->radius = r;
}
```

Stage 8a: Simple Implementation

```
_Shape.h:
…
void Shape_grow(Shape s, float f) {
    _Shape* sp = (_Shape*)s;
    (*(sp->grow_fp))(s,f);
}
```

- Inefficient:
  - 20 virtual functions -> each object is 80 bytes larger!
  - initialization would take 10x longer

Stage 8b: vtbl implementation

```
Object.h:
typedef void* Object;
```

```
_Object.h:
typedef struct {
    void* vtbl[];
} _Object;
```

```
_Shape.h:
typedef struct {
    _Object object;
    float x;
    float y;
} _Shape;
```

Stage 8b: vtbl implementation

```
Circle.c:
#include "_Circle.h"

void* _circle_vtbl[1] = {(void*)(&Circle_grow)};

void _Circle_init(Circle c, float x,y,r) {
    _Circle* cp = (_Circle*)c;
    // Initialize baseclass
    Shape_init((Shape)c, x,y);

    // install vtbl
    cp->shape.object.vtbl = _circle_vtbl;
}
```


Stage 8b: vtbl implementation

```
Shape.c:
typedef (*grow_signature)(Shape,float);
void Shape_grow(Shape s, float f) {
    _Shape* sp = (_Shape*)s;
    (*((grow_signature)sp->object.vtbl[0]))(s,f);
}
```

Stage 8: Implementation Recap

- Added an ultimate baseclass "Object"
  - All classes inherit from Object
  - Object has a data member called "vtbl"
    - pointer to an array of function pointers
  - vtbl is installed in the initialization routine
    - after baseclasses are initialized
    - before class itself is initialized
      - N.B. Java/Smalltalk is different here than C++
- All objects therefore have a vtbl
  - "virtual" functions are accessed indirectly off the vtbl
  - non-virtuals accessed in the regular manner

Stage 8c: Universal virtuals

- The Problem:
  - Logically, it is the decision of the derived class whether a function should be overridden or not.
  - Implementation just shown has the decision made in the baseclass
    - If virtual then can overide
    - If not then stuck with the static implementation provided in the baseclass
- The Solution:
  - Make all operations virtual all the time
  - Allows also for "disinheritance"
    - overriding of an operation that has no meaning in the derived class with a routine that throws an error

Stage 8: Pros and Cons

- Pros:
  - Can efficiently handle polymorphism
- Cons:
  - Highly conventional
  - Not type-safe

Stage 9: Run-Time Type Checking

- Can use vtbl as a type identifier.

Stage 9: Interface

Object.h:

```
typedef void* Object;
extern void* classOf(Object);
```

Circle.h:

```
extern void* Circle_class();
```

Rectangle.h:

```
extern void* Rectangle_class();
```


Stage 9: Implementation

Object.c:

```
void* classOf(Object o) {
    return (void*)((_Object*)o->vtbl);
}
```

Circle.c:

```
void* Circle_class() {
    return (void*) _circle_vtbl;
}
```

Rectangle.h:

```
void* Rectangle_class() {
    return (void*) _rectangle_vtbl;
}
```

Stage 9: Client

```
client.c:
#include “List.h”
#include “Circle.h”
#include “Rectangle.h”
void client() {
    List shapes = List_create(100);
    List_addItem(shapes, Circle_create(3,3, 1.5));
    List_addItem(shapes, Rectangle_create(0,0,1,1));
    for(int i = 0; i < List_getNum(shapes); i++) {
        Shape s = (Shape) List_getItem(shapes,i);
        if( classOf(s) == Rectangle_class() )
            Shape_move(s,5,5);
    }
}
```

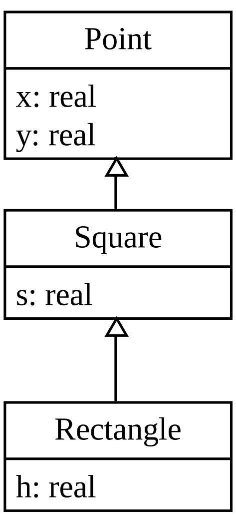

Stage 10: Introspection

- Instead of pointing to a simple vtbl, point to an object of class `Class`.
- The Class object can store information about the class, including
  - the name of the class
  - the virtual table
  - a reference to its parent class
  - the names of all the methods
  - the argument types and return types of the methods
    - each expressed as Class objects themselves

Stages

1. data structures
2. structured data
3. modular data hiding
4. object data hiding
   4.a. applied uniformly (an “aha!”)
5. object references
6. object genericity
7. inheritance
8. polymorphism
   8.b. vtbls
   8.c. universal virtuals
9. Run-time type checking
10. Introspection

Implementation Inheritance



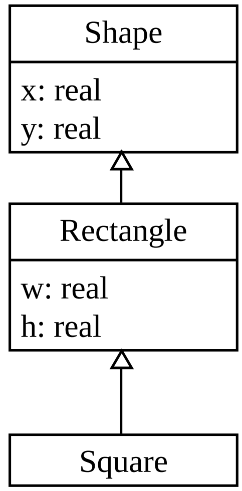
• A point defines an x and a y.

• Handy! A square need only add a side dimension.

• Handy again! A rectangle need only add a height dimension!

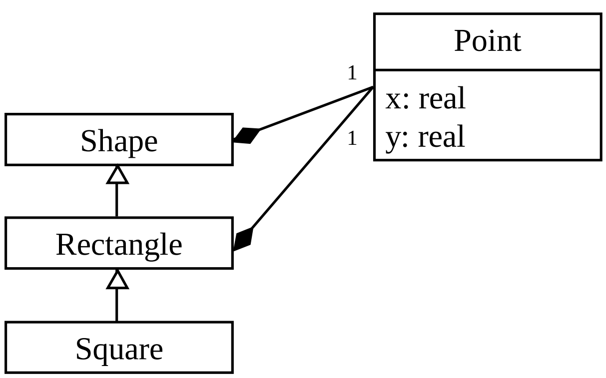
Proper Use of Inheritance

- The preceding works in the language but is a silly use of inheritance
  - get none of the benefits of OOA-OOD
    - locality of change
- This follows the “is-a” (generalization/specialization) hierarchy, and arrives naturally from an application of OOA.



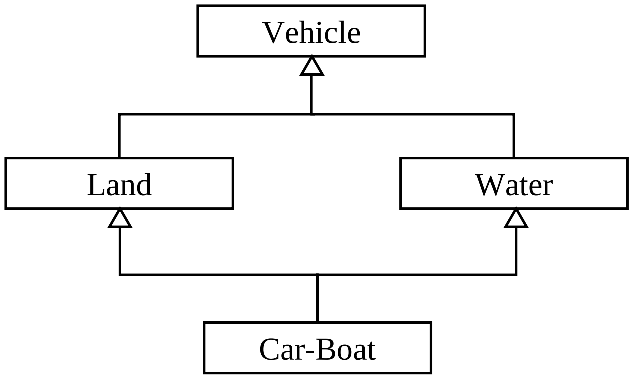
Has-A

- Often, implementation inheritance is a “has-a” aching to hatch out.



Multiple Inheritance

- "MI" is problematic
  - one copy of baseclass or two?
    - 2 is easy to implement
    - 1 is usually what you want
  - complexity of designs
    - "mixin" fever



Interface Inheritance

- Legitimate use of MI that we can't do without:
  - implementing an interface
  - us not an "is-a" relationship, but is still good OOD