Graphics Class Design

Lecture 18

Hartmut Kaiser

https://teaching.hkaiser.org/spring2024/csc3380/

Software Development Notes

The SOLID Principles

- Single-responsibility Principle
  - A class should have one and only one reason to change, meaning that a class should have only one job.
- Open-closed Principle
  - Objects or entities should be open for extension but closed for modification.
- Liskov Substitution Principle
  - Let q(x) be a property provable about object x of type T. Then q(y) should be provable for object y of type S where S is a subtype (derived type) of T.
- Interface segregation principle
  - A client should never be forced to depend on an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use.
- Dependency inversion principle
  - Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on internal state of the low-level module,
  - They should depend on abstractions (functionalities exposed)

Kinds of Refactoring

- Move Member Function or Data
- Rename Member Function or Data
- Pull-up/push-down
- Extract class/method
- Encapsulate field
- Replacing code with patterns
- Functional Refactoring

Extract Class/Method

- Pull methods and classes out of another method or class
- Reduce the size of the larger class/method to improve cohesion
- Which SOLID principle relates to this?
  - Single-responsibility (A class should have one and only one reason to change, meaning that a class should have only one job.)

Example

```
class Car {
public:
    Car() {
        // extremely complicated state-dependent logic
    }
    void accelerate() { /*...*/ }
    void brake() { /*...*/ }
    void turn() { /*...*/ }
};
```

- Move creating a `Car` to a factory:

```
class CarFactory {
    public: Car create_car() { /*...*/ }

    // all the state
};
```

Encapsulate Field

- Take a field (i.e., member variable) from being public to being accessed by a "getter"
- In some languages, this is extremely easy:
  - C# has a feature where you can make a field secretly call a getter
- In other programming languages, such as Java and C++, it takes a bit of work:
  - Hide the field by making it private
  - Fix all the errors that appear by using the new getter method
- When should you do that?
  - Encapsulate members that may become inconsistent when one of those changes independently

The Fallacy of Encapsulation

- If you return a mutable object, you are not encapsulating anything
- Consider the following attempt at encapsulation:

```cpp
class OrderedCarList {
private:
    std::map<std::string, Car> list;

public:
    Car& get_car(std::string const& brand) {
        return list[brand];
    }
};
```

The Fallacy of Encapsulation

```
OrderedCarList car_list = { /*...*/ };
auto& car = car_list.get_car("Ford");
car.set_price(car.get_price() * 0.9);
```

- The encapsulated map in the `car_list` object is changed outside of the `OrderedCarList` class!

Real Encapsulation

```
class OrderedCarList {
private:
    std::map<std::string, Car> list;

public:
    void change_price(std::string const& brand, double price) {
        return list[brand].set_price(price);
    }
};
```

Why bother Encapsulating?

- Encapsulation has three purposes:
  - Reduce state-based errors
    - If an operation needs to occur before or after a state change, your setter can do this so the caller won't forget
  - Reduce coupling
    - Dependency on an inner object means it can't change to a different class without breaking your build
  - Maintain data integrity
    - The class can run checks to data changes to make sure data states remain valid and stable

In Conclusion

- Refactor between sprints to reduce technical debt
- Remember these simple, common refactoring techniques in future technical interviews
- Use them on your own code if it's getting unmanageable
- Remember, refactoring properly won't break anything
  - Testing, testing, testing…!

Graphics Class Design

Abstract

- We have discussed classes in previous lectures
- Here, we discuss design of classes
  - Library design considerations
  - Class hierarchies (object-oriented programming)
  - Data hiding

Ideals

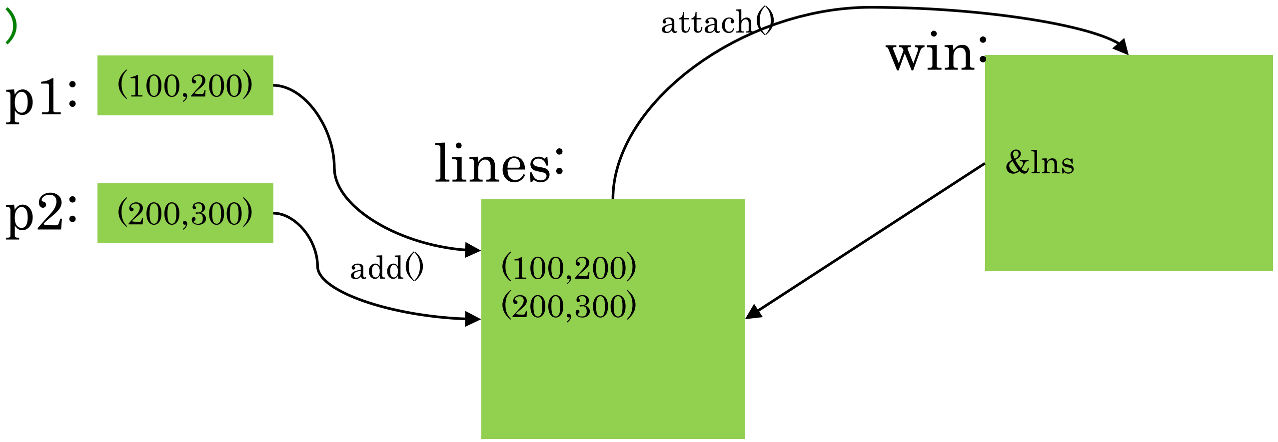
- Our ideal of program design is to represent the concepts of the application domain directly in code.
  - If you understand the application domain, you understand the code, and vice versa. For example:
    - `Window` – a window as presented by the operating system
    - `Point` – a coordinate point
    - `Line` – a line as you see it on the screen
    - `Color` – as you see it on the screen
    - `Shape` – what's common for all shapes in our Graph/GUI view of the world
- The last example, Shape, is different from the rest in that it is a generalization.
  - You can't make an object that's "just a Shape"

Logically identical Operations have the same Name

- For every class,
  - `draw_lines()` does the drawing
  - `move(dx, dy)` does the moving
  - `s.add(x)` adds some `x` (e.g., a point) to a shape `s`.
- For every property `x` of a `Shape`,
  - `x()` returns its current value and
  - `set_x()` gives it a new value
  - e.g.,
    - `Color c = s.color();`
    - `s.set_color(Color::blue);`

Logically different Operations have different Names

```
Lines lines;
Point p1(100, 200);
Point p2(200, 300);
// add points to lines (make copies)
lines.add(p1, p2);
// attach lines to window
win.attach(ln);
```



- Why not win.add(ln)?
  - add() copies information; attach() just creates a reference
  - we can change a displayed object after attaching it, but not after adding it

Possible pitfall

```
void add_line(Simple_window& win)
{
    Graph_lib::Lines x;
    x.add(Graph_lib::Point(100, 100), Graph_lib::Point(200, 100));
    x.add(Graph_lib::Point(150, 50), Graph_lib::Point(150, 150));
    win.attach(x);
}   // oops, lifetime of x ends here

void main()
{
    Simple_window win(Graph_lib::Point(100, 100), 600, 400, "Canvas");

    add_line(win);        // asking for trouble

    win.wait_for_button();
}
```

Expose Things Uniformly

- Data should be private
  - Data hiding – so it will not be changed inadvertently
  - Use private data, and pairs of public access functions to get and set the data

```
c.set_radius(12);                   // set radius to 12
c.set_radius(c.radius() * 2);       // double the radius (fine)
c.set_radius(-9);                   // set_radius() could check for negative,

double r = c.radius();    // returns value of radius
c.radius = -9;            // error: radius is a function (good!)
c.r = -9;                 // error: radius is private (good!)
```

- Our functions can be private or public
  - Public for interface
  - Private for functions used only internally to a class

What does “private” buy us?

- We can change our implementation after release
- We don’t expose FLTK types used in implementation to our users
  - We could replace FLTK with another library without affecting user code
- We could provide ‘checking’ in access functions
  - But we haven’t done so systematically (later?)
- Functional interfaces can be nicer to read and use
  - E.g., s.add(x) rather than s.points.push_back(x)
- We enforce immutability of shape
  - Only color and style change; not the relative position of points
  - const member functions
- The value of this “encapsulation” varies with application domains
  - Is often most valuable
  - Is the ideal
    - i.e., hide representation unless you have a good reason not to

What is a Library?

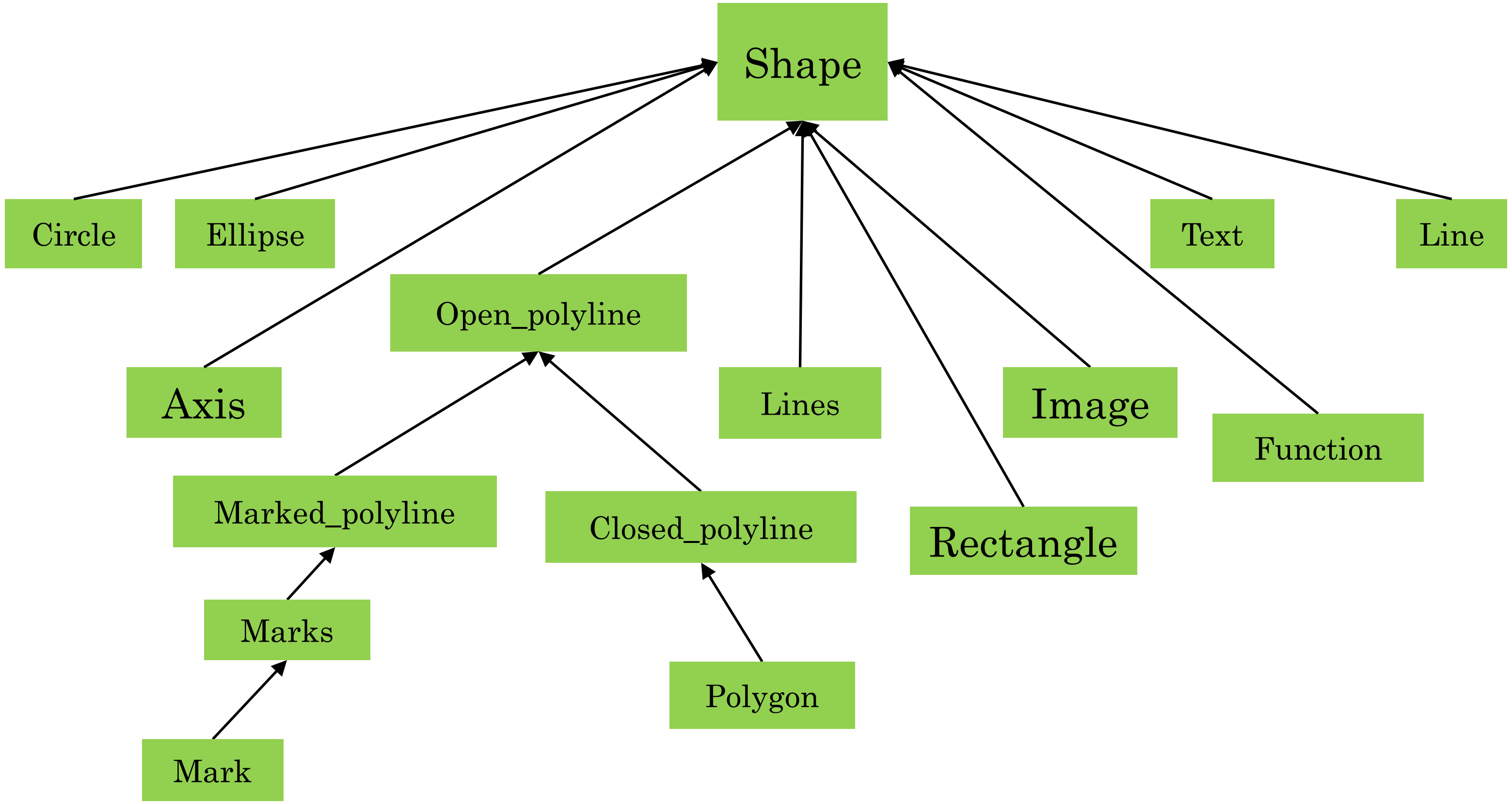
- A collection of classes and functions meant to be used together
  - As building blocks for applications
  - To build more such "building blocks"
- A good library models some aspect of a domain
  - It doesn't try to do everything
  - Our library aims at simplicity and small size for graphing data and for very simple GUI
- We can't define each library class and function in isolation
  - A good library exhibits a uniform style ("regularity")

“Regular” Interfaces

```
Line ln(Point(100, 200), Point(300, 400));
Mark m(Point(100, 200), 'x');    // display a single point as an 'x'
Circle c(Point(200, 200), 250);

// Alternative (not supported):
Line ln2(x1, y1, x2, y2);    // from (x1, y1) to (x2, y2)

// How about? (not supported):
Square s1(Point(100, 200), 200, 300);           // width==200 height==300
Square s2(Point(100, 200), Point(200, 300));    // width==100 height==100

Square s3(100, 200, 200, 300);
// is 200, 300 a point or a width plus a height?
```

Class Shape

- All our shapes are "based on" the Shape class
  - E.g., a Polygon is a kind of Shape



Class Shape

- Shape ties our graphics objects to "the screen"
  - Window "knows about" Shapes
  - All our graphics objects are kinds of Shapes
- Shape is the class that deals with color and style
  - It has Color and Line_style members
- Shape can hold Points
- Shape has a basic notion of how to draw lines
  - It just connects its Points

Class Shape – is abstract

- You can't make a "plain" Shape

```
protected:
    Shape();    // protected to make class Shape abstract
```

  - For example:

```
    Shape ss;   // error: cannot construct Shape
```

  - Protected means "can only be used from a derived class"

- Instead, we use Shape as a base class

```
struct Circle : Shape {    // "a Circle is a Shape"
    // ...
};
```

- An abstract class is a user defined data type, which can be used as a base class only

Class Shape

- Shape deals with color and style
  - It keeps its data private and provides access functions

```
public:
    void set_color(Color col);
    Color color() const;
    void set_style(Line_style sty);
    Line_style style() const;
    // ...
private:
    // ...
    Color line_color = fl_color();
    Line_style ls = 0;
```

Class Shape

- Shape stores Points
  - It keeps its data private and provides access functions

```
public:
    Point point(int i) const;    // read-only access to points
    int number_of_points() const;
    // ...
protected:
    void add(Point p);           // add p to points
    // ...
private:
    std::vector<Point> points;  // not used by all shapes
```

Class Shape

- Shape itself can access `points` directly:

```
void Shape::draw_lines() const    // draw connecting lines
{
    if (color().visible() && points.size() > 1)
        for (int i = 1; i < points.size(); ++i)
            fl_line(points[i - 1].x, points[i - 1].y, points[i].x, points[i].y);
}
```

- Others (incl. derived classes) use `point()` and `number_of_points()`
  - why?

```
void Lines::draw_lines() const    // draw a line for each pair of points
{
    for (int i = 1; i < number_of_points(); i += 2)
        fl_line(point(i - 1).x, point(i - 1).y, point(i).x, point(i).y);
}
```

Class Shape (Basic Idea of Drawing)

```
// The real heart of class Shape (and of our graphics interface system)
// called by Window (only)
void Shape::draw() const
{
    // ... save old color and style
    // ... set color and style for this shape

    // ... draw what is specific for this particular shape
    // ... Note: this varies dramatically depending on the type of shape
    // ... e.g. Text, Circle, Closed_polyline

    // ... reset the color and style to their old values
}
```

Class Shape (Implementation of Drawing)

```
// The real heart of class Shape (and of our graphics interface system)
// called by Window (only)
void Shape::draw() const
{
    Fl_Color oldc = fl_color();    // save old color
    // there is no good portable way of retrieving the current style (sigh!)
    fl_color(line_color.as_int());    // set color and style
    fl_line_style(ls.style(), ls.width());

    // here is what is specific for a "derived class" is done
    draw_lines();    // call the appropriate draw_lines()
                     // a "virtual call"

    fl_color(oldc);      // reset color to previous
    fl_line_style(0);    // (re)set style to default
}
```

Note!

Class Shape

- In class Shape

```
virtual void draw_lines() const;     // draw the appropriate lines
```

- In class Circle

```
void draw_lines() const { /* draw the Circle */ }
```

- In class Text

```
void draw_lines() const { /* draw the Text */ }
```

- Circle, Text, and other classes
  - "Derive from" Shape
  - May "override" draw_lines()

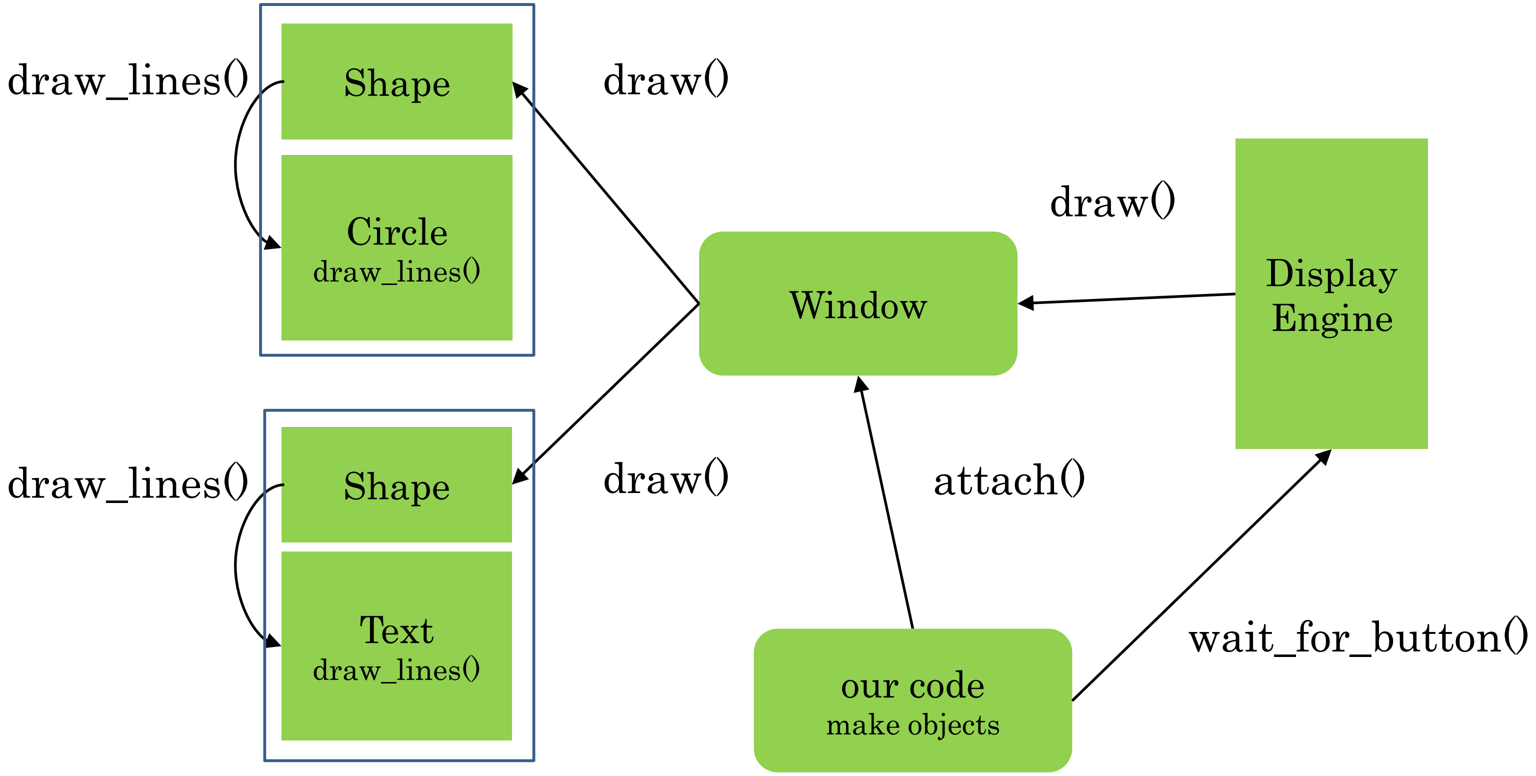
Class Shape

```
// deals with color and style, and holds a sequence of lines
class Shape {
public:
    void draw() const;    // deal with color and call draw_lines()
    virtual void move(int dx, int dy);    // move the shape += dx and += dy

    void set_color(Color col);    // color access
    int color() const;
    // ... style and fill_color access functions

    Point point(int i) const;    // (read-only) access to points
    int number_of_points() const;
protected:
    // ...
};
```

Class Shape

```
// deals with color and style, and holds a sequence of lines
class Shape {
protected:
    // ...

    Shape();    // protected to make class Shape abstract
    // ... prevent copying
    void add(Point p);                   // add p to points
    virtual void draw_lines() const;     // simply draw the appropriate lines
private:
    std::vector<Point> points;    // not used by all shapes
    Color lcolor;                 // line color
    Line_style ls;                // line style
    Color fcolor;                 // fill color
};
```

Display Model Completed

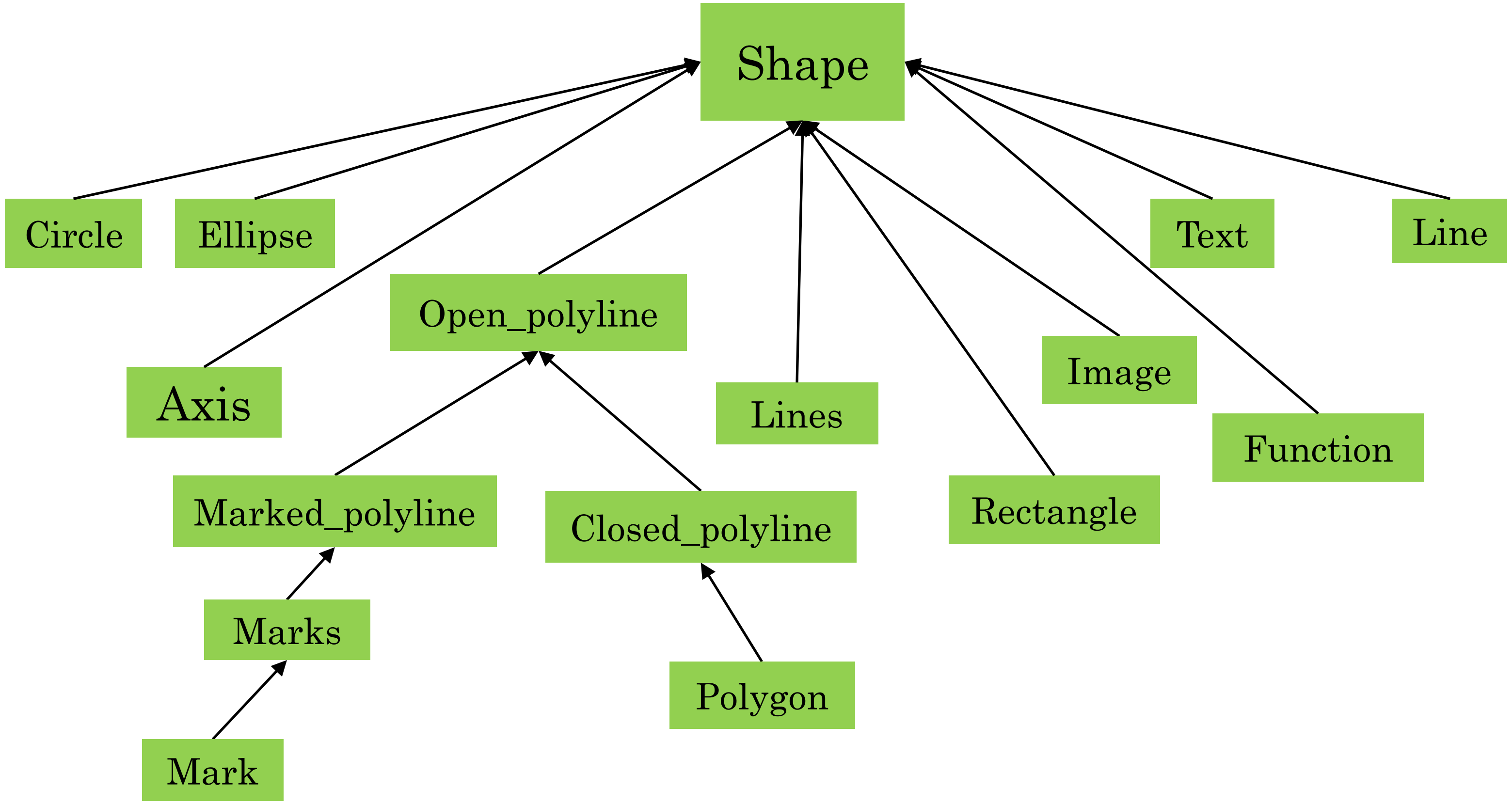


Language Mechanisms

- Most popular definition of object-oriented programming:
  - OOP == inheritance + polymorphism + encapsulation
- Base and derived classes // inheritance
  - `struct Circle : Shape { … };`
  - Also called "inheritance"
- Virtual functions // polymorphism
  - `virtual void draw_lines() const;`
  - Also called "run-time polymorphism" or "dynamic dispatch"
- Private and protected // encapsulation
  - `protected: Shape();`
  - `private: std::vector<Point> points;`

A simple Class Hierarchy

- We chose to use a simple (and mostly shallow) class hierarchy
  - Based on Shape



Object Layout

- The data members of a derived class are simply added at the end of its base class (a Circle is a Shape with a radius)

Shape:

points line_color ls

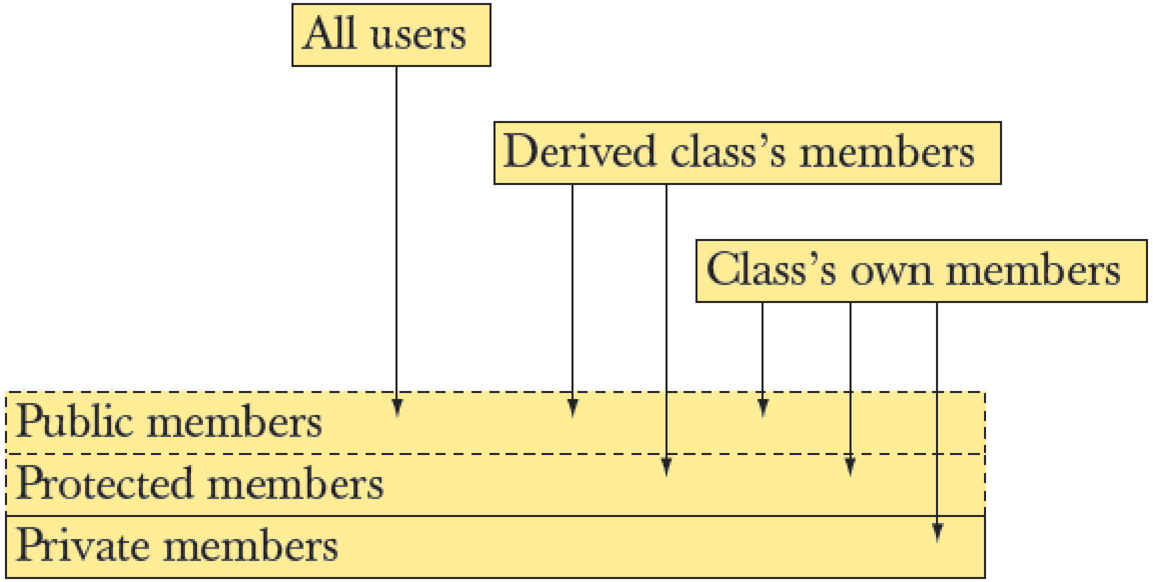
Circle:

points line_color ls ---------------------- r

Benefits of Inheritance

- Interface inheritance
  - A function expecting a shape (a Shape&) can accept any object of a class derived from Shape.
  - Simplifies use
    - sometimes dramatically
  - We can add classes derived from Shape to a program without rewriting user code
    - Adding without touching old code is one of the “holy grails” of programming
- Implementation inheritance
  - Simplifies implementation of derived classes
    - Common functionality can be provided in one place
    - Changes can be done in one place and have universal effect
      - Another “holy grail”

Access Model



- A member (data, function, or type member) or a base can be
  - Private, protected, or public

Pure virtual functions

- Often, a function in an interface can't be implemented
  - E.g. the data needed is "hidden" in the derived class
  - We must ensure that a derived class implements that function
  - Make it a "pure virtual function" (=0)
- This is how we define truly abstract interfaces ("pure interfaces")

```
// interface to electric motors
struct Engine
{
    // no data
    // (usually) no constructor
    virtual double increase(int i) = 0;    // must be defined in a
                                           // derived class
    // ...
    virtual ~Engine();    // (usually) a virtual destructor
};
Engine eee;    // error: Engine is an abstract class
```

Pure virtual functions

- A pure interface can then be used as a base class
  - We talked about Constructors and destructors before

```
// engine model M123
class M123 : public Engine {
     // representation
public:
    M123();    // constructor:  initialization, acquire resources

    // overrides Engine::increase
    double increase(int i) { /* ... */ }

    // ...
    ~M123();    // destructor: cleanup, release resources
};

M123 window3_control;    // OK
```

Technicality: Copying

- If you don't know how to copy an object, prevent copying
  - Abstract classes typically should not be copied

```
class Shape
{
    // ...
    Shape(Shape const&) = delete;                    // don't copy construct
    Shape& operator=(Shape const&) = delete;     // don't copy assign
};

void f(Shape& a)
{
    Shape s2 = a;    // error: no Shape copy constructor (it's deleted)
    a = s2;          // error: no Shape copy assignment (it's deleted)
}
```

Technicality: Overriding

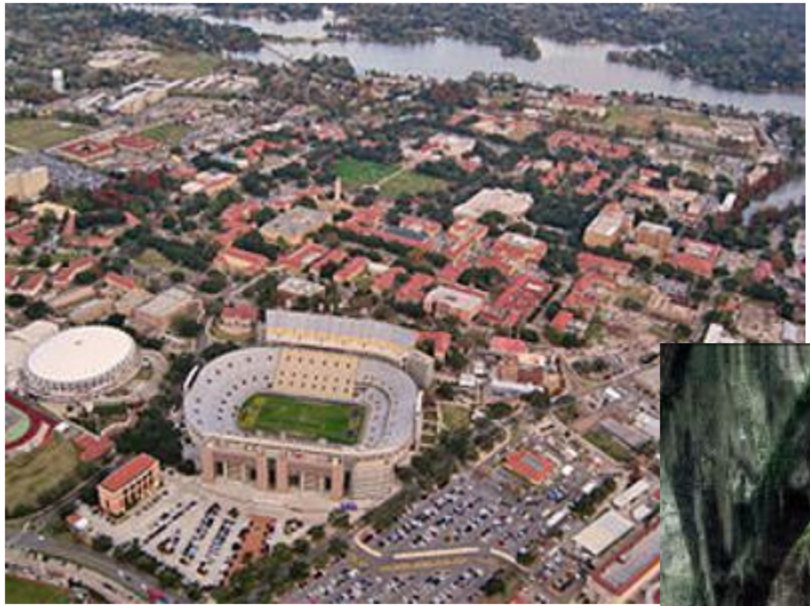
- To override a virtual function, you need
  - A virtual function
  - Exactly the same name
  - Exactly the same type

```
struct B
{
    void f1();    // not virtual
    virtual void f2(char);
    virtual void f3(char) const;
    virtual void f4(int);
};
```

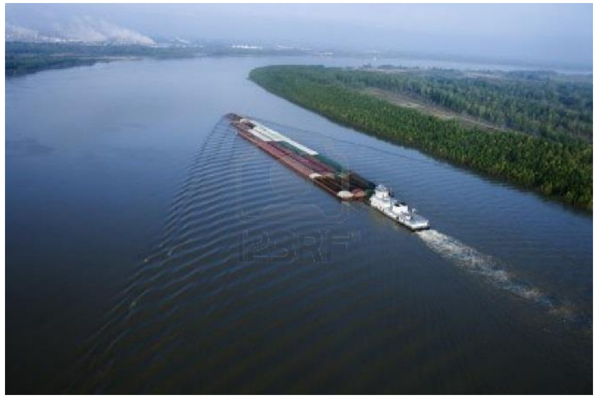
```
struct D : B
{
    void f1();         // doesn't override
    void f2(int);      // doesn't override
    void f3(char);     // doesn't override
    void f4(int);      // overrides
};
```

Types of Inheritance

- Interface inheritance
  - Derived class depends on an interface that is defined by the base class
  - Code relies on interface for invoking functionalities
  - Our graphics system critically depends on the `draw_lines()` interface
- Implementation inheritance
  - Simplify derived classes by moving common functionalities to base class
  - Our `Shape` class manages colors, line styles, and list of points
- All of this ensures independence of graphics system (Window) from kinds of Shapes
  - We can add new shapes without recompiling
  - In fact `Window` doesn't know anything about concrete shapes (`Circles`, `Rectangles`, etc.)



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP