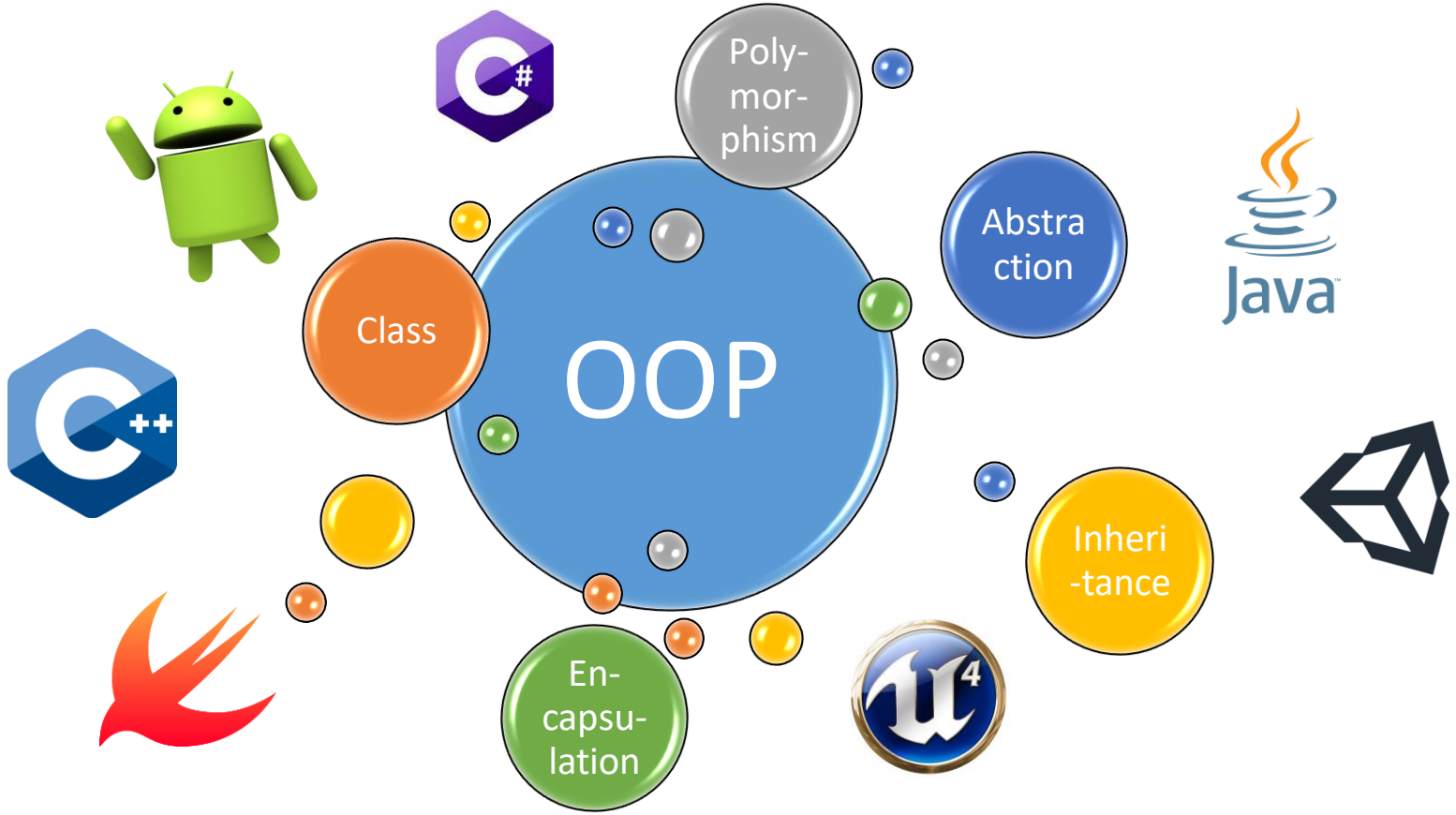


Introduction to Object-Oriented Programming

Kuan-Ting Lai

2020/2/29

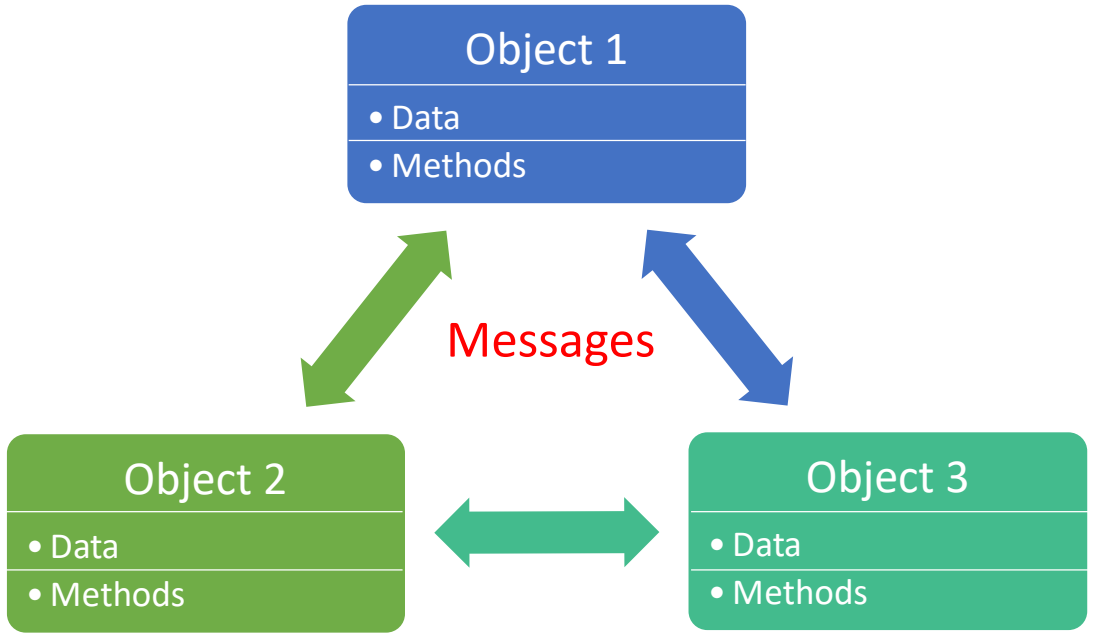
What is Object-Oriented Programming (OOP)?

- A program paradigm based on the concept of Objects, which contain data and functions (Wikipedia)

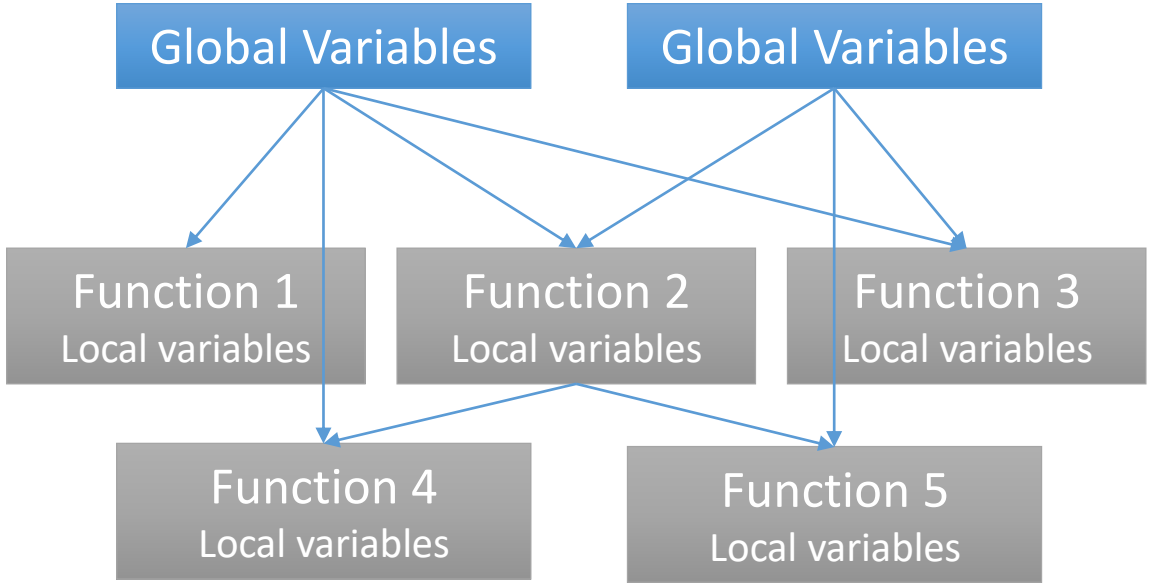
- Common OOP terminologies:
  - Object -> Class
  - Data -> fields or attributes
  - Functions -> method or behavior

OOP vs. Functional Programming

Object-Oriented Programming

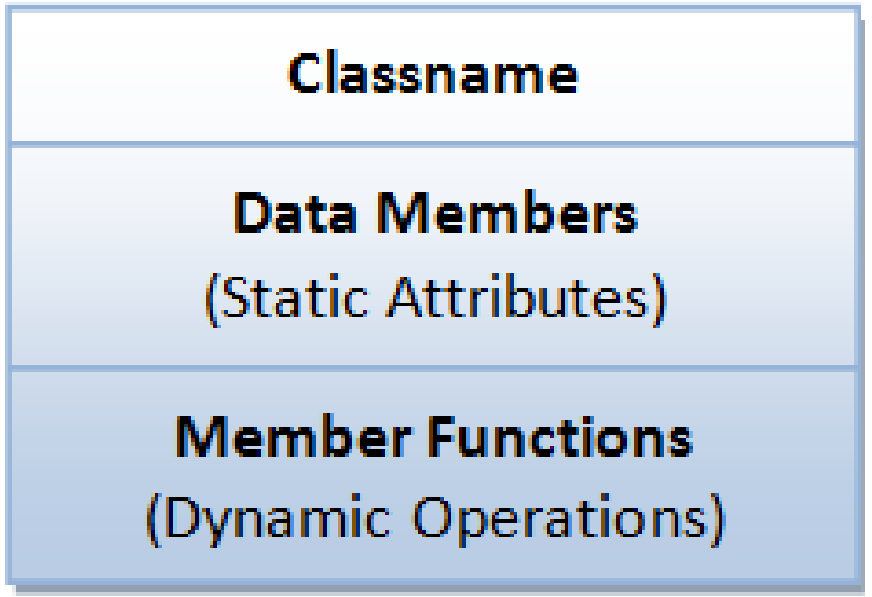


Functional (Procedure) Programming



Object (Class)

- Class Name
  - Identifier for the class
- Data
  - Or variables, attributes, fields. Save attributes of the class
- Functions
  - Or methods. Manipulate the data.



https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp3_OOP.html

4 Principles of OOP

Abstraction

Encapsulation

Inheritance

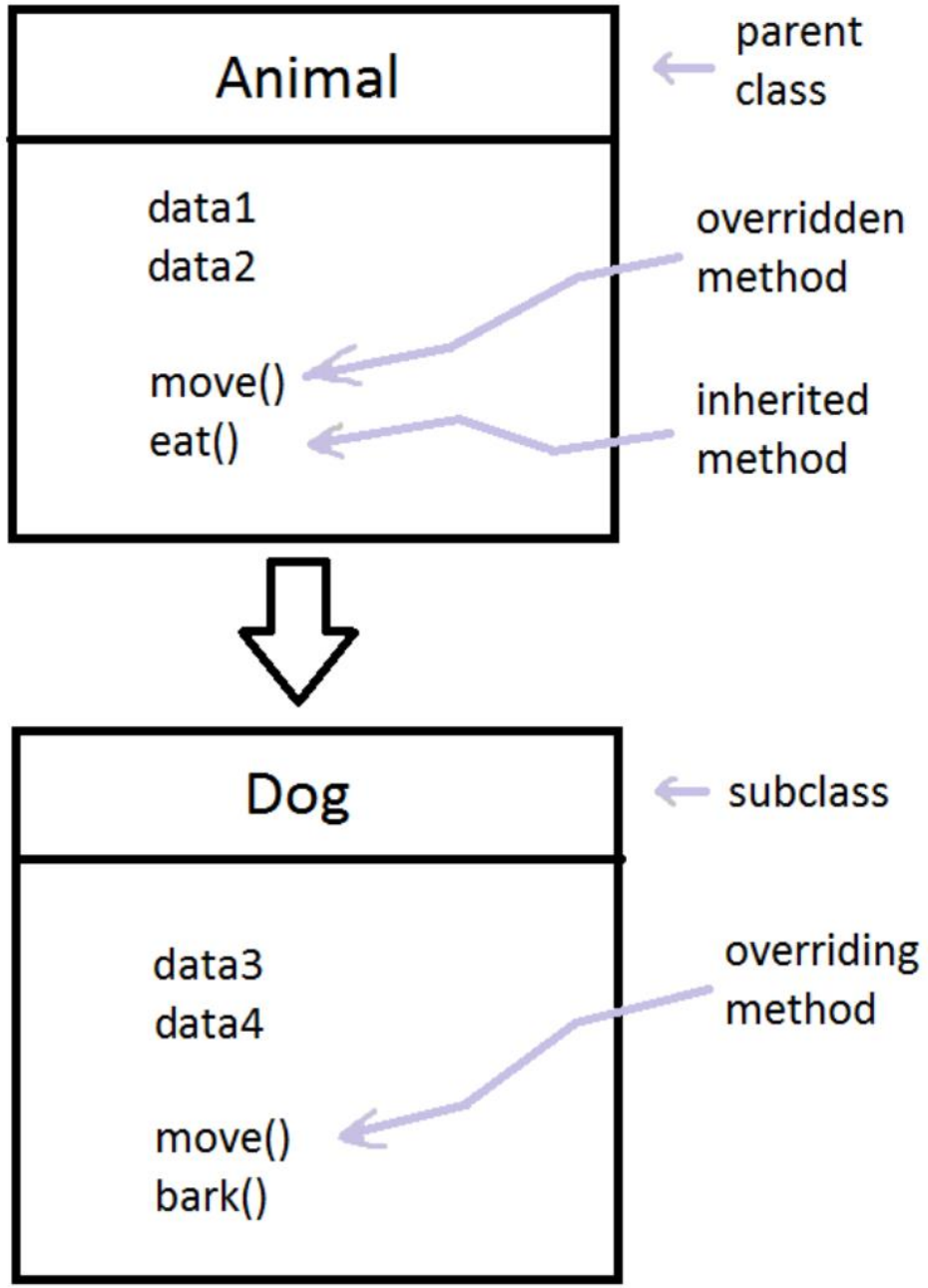
Polymorphism

Data Abstraction & Encapsulation

- Encapsulation
  - Wrap data & functions into a Class
- Abstraction
  - Define functions but hide the details of implementation

Inheritance

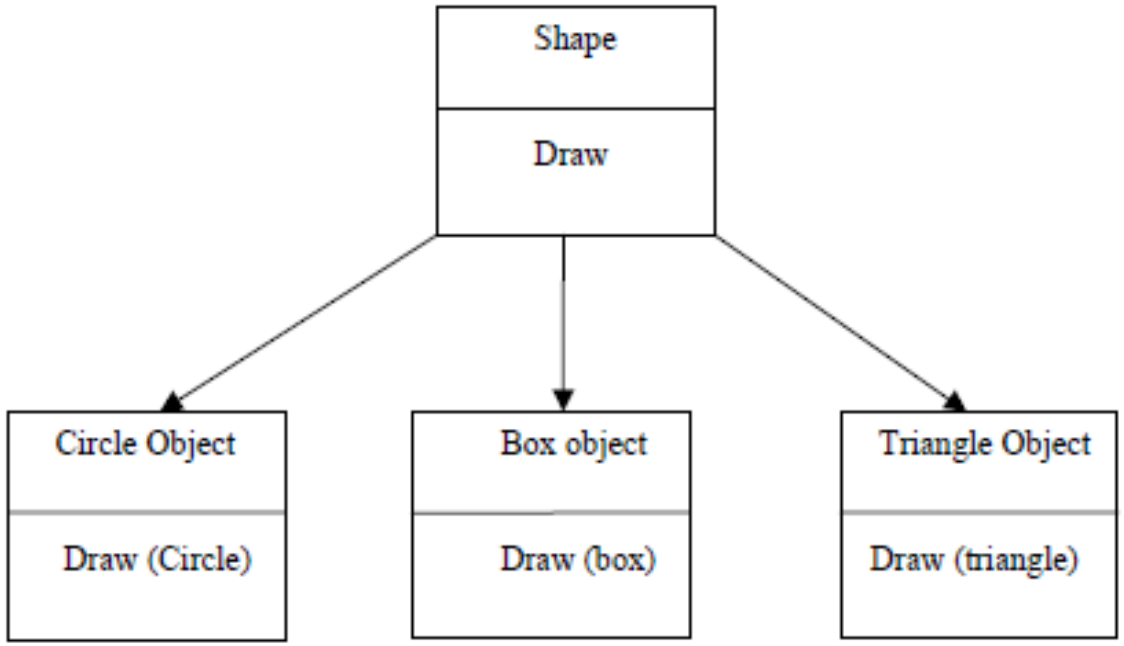
- Derived class can inherit data and functions from parent class



https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)

Polymorphism

- Override the behavior of a parent's function
  - Dynamic polymorphism: Overriding
  - Static polymorphism: Overloading



Pooja Chawla, OOP with C++

Dynamic Binding

- Mechanism for function call of polymorphism and inheritance
- The function is decided at run-time, which depends on the types of pointers

Example: C++ Class

- Define a Box class with length, width, height

```
class Box {
public:
    double length;  // Length of a box
    double width;   // Width of a box
    double height;  // Height of a box
};
```

Create Instances of Class

- Declare Box classes and create objects

```
Box Box1;          // Declare Box1 of type Box
Box Box2;          // Declare Box2 of type Box
```

Print Volume

- Volume = length * width * height
- Output
  - Volume of Box1: 210
  - Volume of Box2: 1560

```
#include <iostream>
using namespace std;

class Box {
public:
      double length;   // Length of a box
      double width;    // Width of a box
      double height;   // Height of a box
};

int main()
{
      Box Box1;        // Declare Box1 of type Box
      Box Box2;        // Declare Box2 of type Box
      double volume = 0.0; // Store the volume of a box

       // box 1 specification
      Box1.height = 5.0;
      Box1.length = 6.0;
      Box1.width = 7.0;

      // box 2 specification
      Box2.height = 10.0;
      Box2.length = 12.0;
      Box2.width = 13.0;

      // volume of box 1
      volume = Box1.height * Box1.length * Box1.width;
      cout << "Volume of Box1 : " << volume << endl;

      // volume of box 2
      volume = Box2.height * Box2.length * Box2.width;
      cout << "Volume of Box2 : " << volume << endl;
      return 0;
}
```

C++ Inheritance (2-1)

- Declare a base class “Shape”
  - Data: width, height
  - Functions: setWidth, setHeight
- Declare a derived class “Rectangle”
- Inheritance Syntax
  - `class derived_class : public base_class`

shapes.h

```
#ifndef _SHAPES_H_
#define _SHAPES_H_

// Base class
class Shape {
public:
    void setWidth(int w) {
        width = w;
    }
    void setHeight(int h) {
        height = h;
    }
protected:
    int width;
    int height;
};

// Derived class
class Rectangle : public Shape {
public:
    int area() {
        return (width * height);
    }
};
#endif
```

private:

C++ Inheritance (2-2)

- Create a instance of Rectangle
  - `Rectangle Rect;`
- Call the functions of base class to set width & height
  - Rect.setWidth(5);
  - Rect.setHeight(7);
- Call the function of derived class
  - Rect.getArea()

main.cpp

```
#include <iostream>
using namespace std;
#include “shapes.h”

int main() {
    Rectangle Rect;

    Rect.setWidth(5);
    Rect.setHeight(7);

    // Print the area of the object.
    cout << "Total area: " << Rect.area();
    cout << endl;

    return 0;
}
```

C++ Abstraction

- Force derived classes to implement a specific function
- A virtual function “= 0” is a pure virtual function
- In C++, pure virtual function is also called interface

```
// Base class
class Shape {
public:
    void setWidth(int w) {
        width = w;
    }
    void setHeight(int h) {
        height = h;
    }
    virtual int area() = 0;

protected:
    int width;
    int height;
};
```

C++ Polymorphism (2-1)

- Overriding area()

```
class Rectangle : public Shape {
public:
    Rectangle(int a = 0, int b = 0) :Shape(a, b) { }

    int area() {
        cout << "Rectangle class area :" << endl;
        return (width * height);
    }
};

class Triangle : public Shape {
public:
    Triangle(int a = 0, int b = 0) :Shape(a, b) { }

    int area() {
        cout << "Triangle class area :" << endl;
        return (width * height / 2);
    }
};
```

C++ Polymorphism (2-2)

- Create a base class pointer
  - `Shape *shape;`
- Create instances of derived classes
  - `Rectangle rec(10, 7);`
  - `Triangle  tri(10, 5);`
- Get reference of instance
  - `shape = &rec;`
- Print result
  - `shape->area();`

```
#include <iostream>
using namespace std;
#include “shapes.h”

int main() {
    Shape *shape;
    Rectangle rec(10, 7);
    Triangle  tri(10, 5);

    // store the address of Rectangle
    shape = &rec;

    // call rectangle area.
    shape->area();

    // store the address of Triangle
    shape = &tri;

    // call triangle area.
    shape->area();

    return 0;
}
```

Constructor & Destructor

- Constructor
  - Called when an object is created
  - Initialize data
- Destructor
  - Called when an object is deleted
  - Can be used to clean data

```
// Base class
class Shape {
public:
    Shape(int a=0, int b=0) {
        width = a; height = b;
    }
    ~Shape(){}

    void setWidth(int w) {
        width = w;
    }
    void setHeight(int h) {
        height = h;
    }
    virtual int area() = 0;

protected:
    int width;
    int height;
};
```

Function Overloading & Operator Overloading

- Function overloading
  - Define functions with the same name, but having different types and/or number of arguments
- Operator overloading
  - Redefine or overload most of the built-in operators available in C++

Function Overloading

- Define a function that can print different types of data

```
#include <iostream>
using namespace std;

class printData {
public:
    void print(int i) {
        cout << "Printing int: " << i << endl;
    }
    void print(double  f) {
        cout << "Printing float: " << f << endl;
    }
    void print(char* c) {
        cout << "Printing character: " << c << endl;
    }
};

int main(void) {
    printData pd;

    pd.print(5);// Print integer
    pd.print(500.263);// Print float
    pd.print("Hello C++");// Print characters

    return 0;
}
```

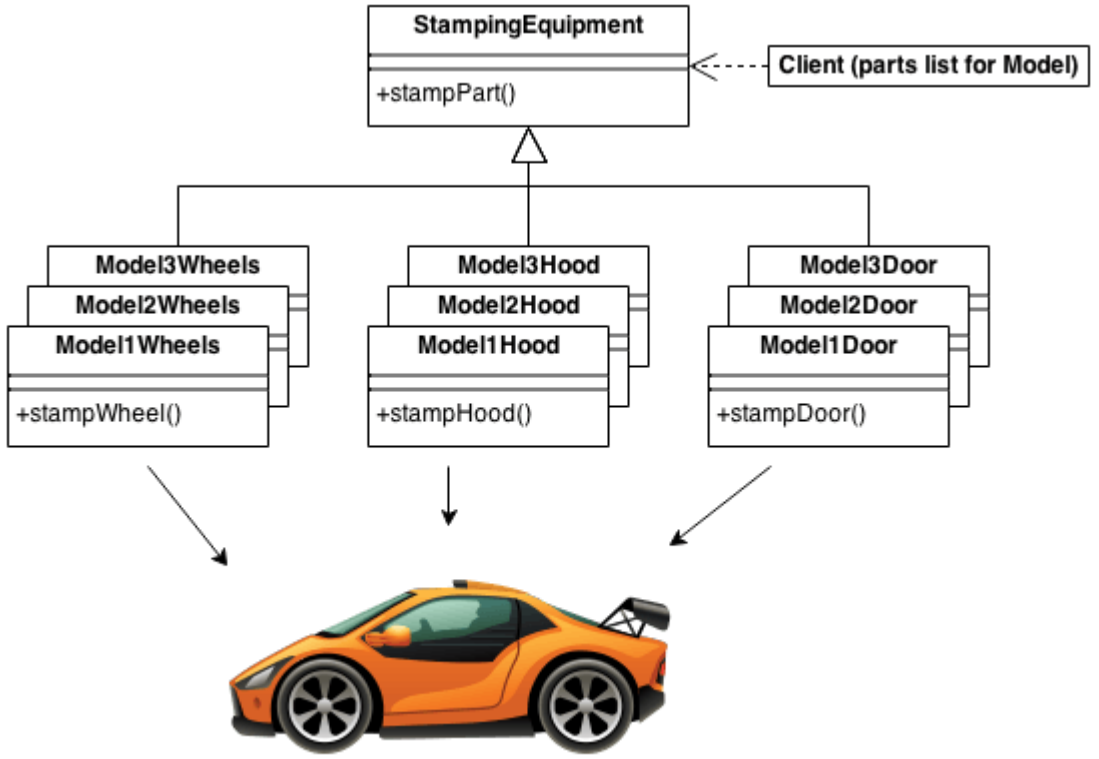
Operator Overloading

- Can overload default operators (=,+, -, *, /, %,…)
- Example: Implement “+” for `class Box`

```
class Box {
public:
    // Overload + operator to add two Box objects.
    Box operator+(const Box& b) {
        Box box;
        box.length = this->length + b.length;
        box.width = this->width + b.width;
        box.height = this->height + b.height;
        return box;
    }
    double length;
    double width;
    double height;
};

int main() {
    Box Box1; Box Box2;

    Box1.length = 6.0; Box1.width = 7.0; Box1.height = 5.0;
    Box2.length = 12.0; Box2.width = 13.0; Box2.height = 10.0;
    cout << "Volume of Box1 : " << Box1.getVolume() << endl;
    cout << "Volume of Box2 : " << Box2.getVolume() << endl;
    // Add two object as follows:
    Box3 = Box1 + Box2;
    cout << "Volume of Box3 : " << Box3.getVolume() << endl;
    return 0;
}
```

Design Patterns

- **Design pattern** is a general reusable solution to a commonly occurring problem in software **design**
- Three main categories:
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns



https://sourcemaking.com/design_patterns

Origin of Design Patterns

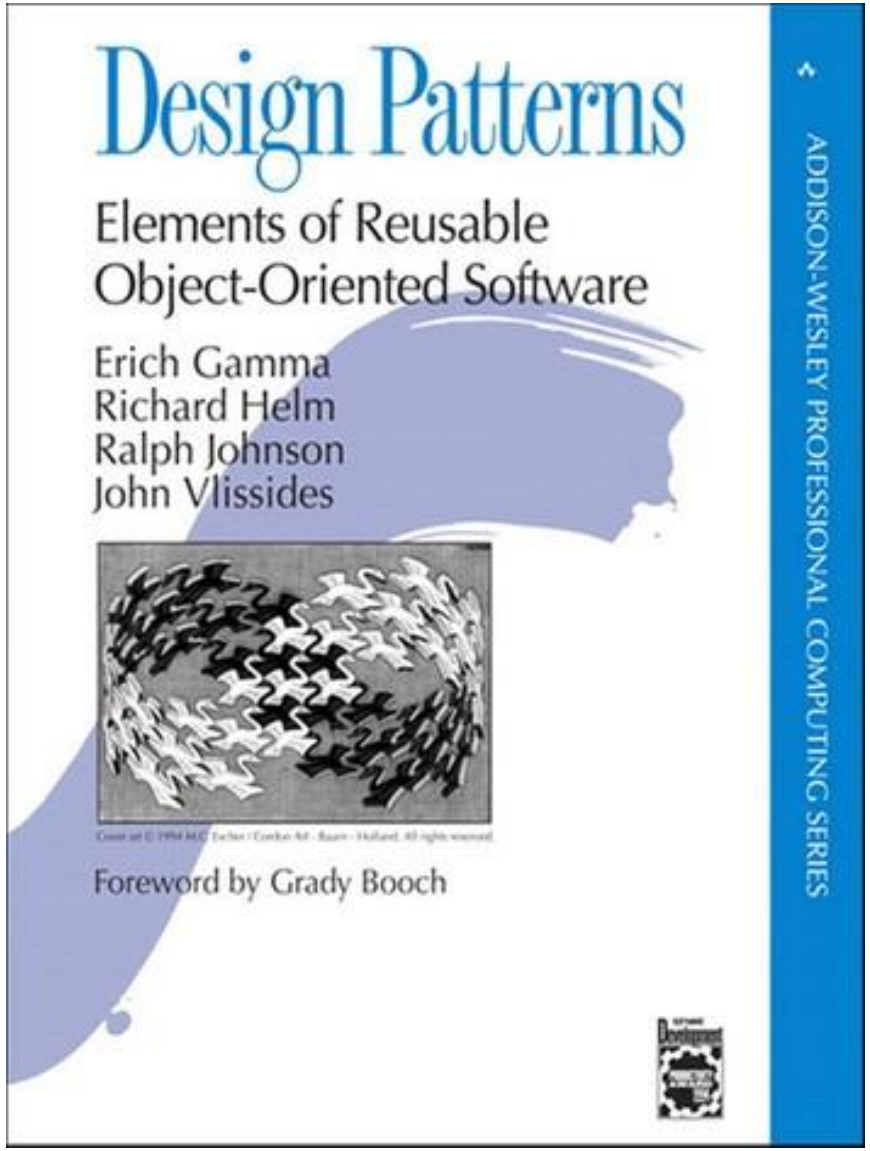
Erich Gamma

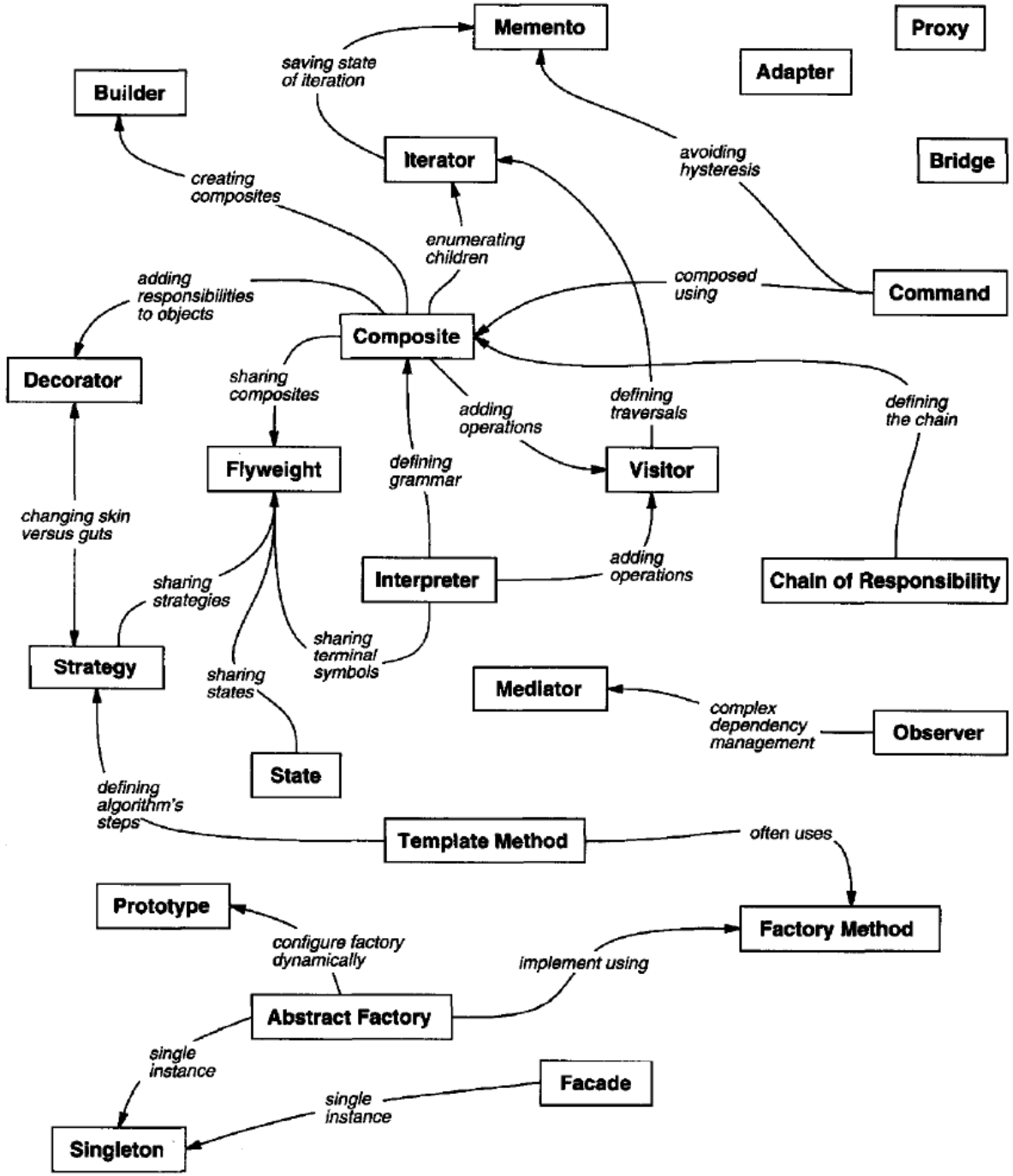
Richard Helm

Ralph Johnson

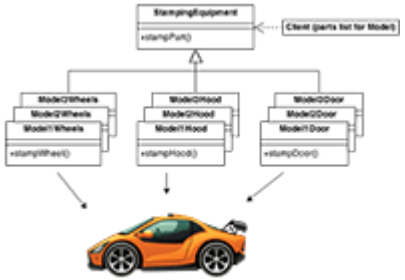
John Vlissides

October 21, 1994



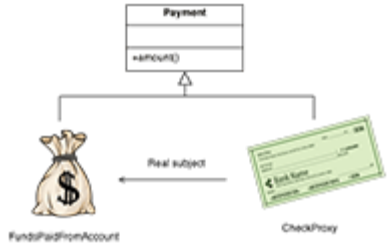

Memento
Proxy
saving state of iteration
Adapter
Builder
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
sharing strategies
Interpreter
adding operations
Chain of Responsibility
sharing terminal symbols
Strategy
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Facade
single instance
Singleton

Design Patterns



- Creational design patterns
  - Initialize objects or create new classes

- Structural design patterns
  - Use inheritance to compose interfaces and compose objects to obtain new functionality

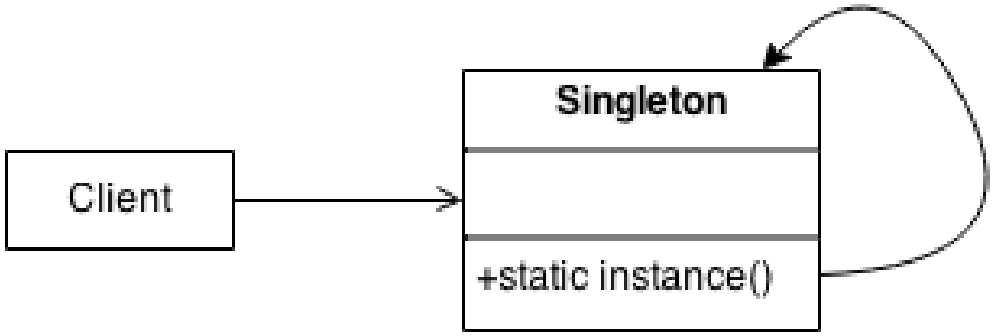


- Behavioral design patterns
  - Communication between objects

https://sourcemaking.com/design_patterns

Singleton Pattern

- Ensure a class has only one instance, and provide a global point of access to it.
- Encapsulated "just-in-time initialization" or "initialization on first use".

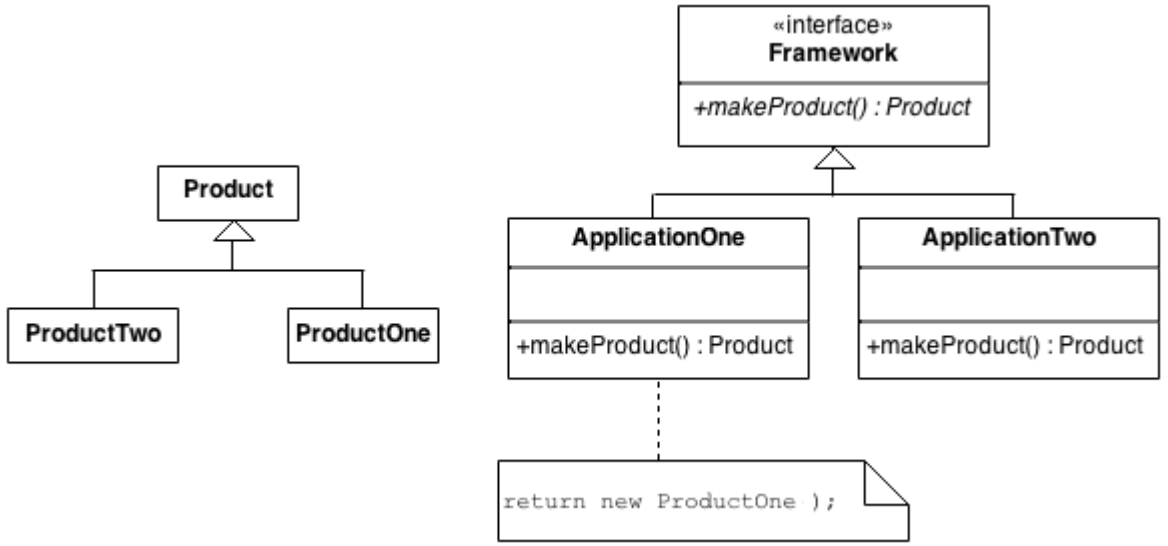


C++ Singleton Example

```
class Singleton
{
public:
    static Singleton* getInstance();

private:
    static Singleton* instance; // Here will be the instance stored.

    Singleton(); // Private constructor to prevent instancing.
};

/* Null, because instance will be initialized on demand. */
Singleton* Singleton::instance = 0;

Singleton* Singleton::getInstance()
{
    if (instance == 0)
        instance = new Singleton();

    return instance;
}
```

Factory Pattern

- Define an interface for creating an object, but let subclasses decide which class to instantiate
- Factory Method lets a class defer instantiation to subclasses
- Defining a "virtual" constructor
- The new operator considered harmful



C++ Factory Pattern

```
enum VehicleType {VT_TwoWheeler, VT_ThreeWheeler};
// Library classes
class Vehicle {
public:
        virtual void printVehicle() = 0;
        static Vehicle* Create(VehicleType type);
};
class TwoWheeler : public Vehicle {
public:
        void printVehicle() {cout << "I am two wheeler" << endl;}
};
class ThreeWheeler : public Vehicle {
public:
        void printVehicle() { cout << "I am three wheeler" << endl;}
};
// Factory method to create objects of different types.
Vehicle* Vehicle::Create(VehicleType type) {
    if (type == VT_TwoWheeler)
        return new TwoWheeler();
    else if (type == VT_ThreeWheeler)
        return new ThreeWheeler();
    else return NULL;
}
```

https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/

References

1. Pooja Chawla, OOP with C++
2. https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm
3. https://en.wikipedia.org/wiki/Object-oriented_programming
4. https://sourcemaking.com/design_patterns