CIB

Component Interface Binder for C++

Satya Das

Jargon

- **Library –** A DLL or executable.
- **Client –** A DLL or executable that is built using library.
- **SDK –** A set of headers and binary components that are published to work with library.
- **Component –** Both library and client are components.

C++ is great but ...

- Not suitable for:
  - forward and backward compatible SDK.
  - compiler independent SDK.

Features (and problems) of C++

- Encapsulation
- Runtime polymorphism
- Function overloading

Encapsulation

- Tightly integrated with object layout.
- Even change in private members can cause incompatibility.

```
class A {
  ...
private:
  int x;
  float y;
};
```

```
class A {
  ...
private:
  int x;
  double y;
};
```

- Workaround: use bridge pattern.

Runtime Polymorphism

- Virtual table can become incompatible.

```
class Base {
  int x;
public:
  virtual void f1() = 0;
  virtual void f2() = 0;

};

class Derived : public Base {
public:
  virtual void g1() = 0;

  virtual void g2() = 0;
};
```

```
class Base {
  int x;
public:
  virtual void f1() = 0;
  virtual void f2() = 0;
  virtual void f3() = 0;
};

class Derived : public Base {
public:
  virtual void g1() = 0;
  virtual void g3() = 0;
  virtual void g2() = 0;
};
```

- Workaround:
  - Only add new virtual methods at the end.
  - Don’t worry about incompatibility.

Function Overloading

- Implemented using name mangling.
- Different compilers use different algorithms.
- **Hits us even when overloading is not used.**
- Workaround:
  - Mandatory use of particular compiler.
  - Release source code and not only SDK.

Workaround Limitations

- Too restrictive.
- Doesn't help in forward or backward compatible SDK.

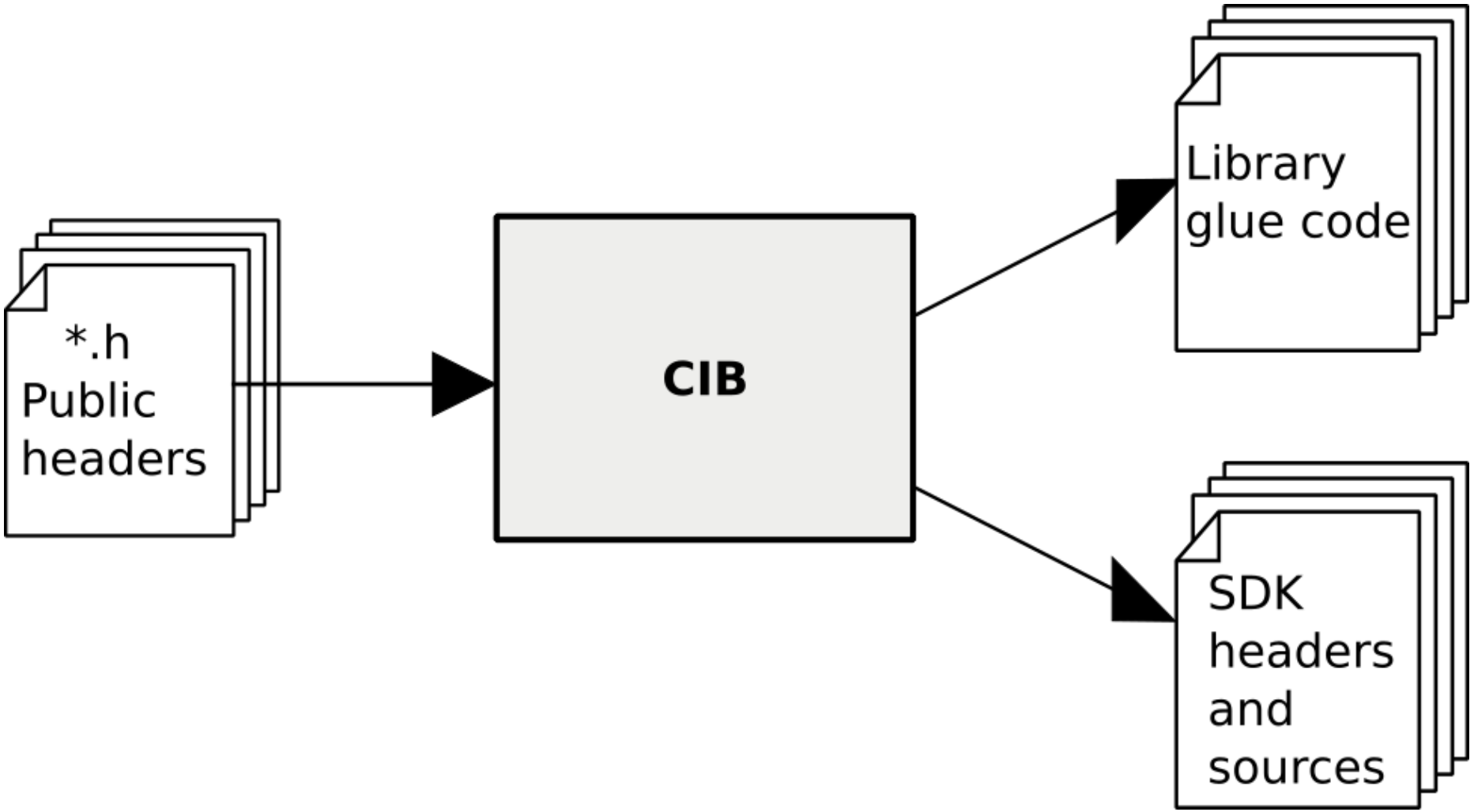
CIB

- A tool that makes it easy to publish C++ SDK.
- Published SDK is:
  - Forward and backward compatible.
  - Compiler independent.
- **CIB** stands for **C**omponent **I**nterface **B**inder.

DEMO #1: Forward compatibility

- Usually change in vtable enforces recompilation.
- **But recompilation not needed if CIB is used.**

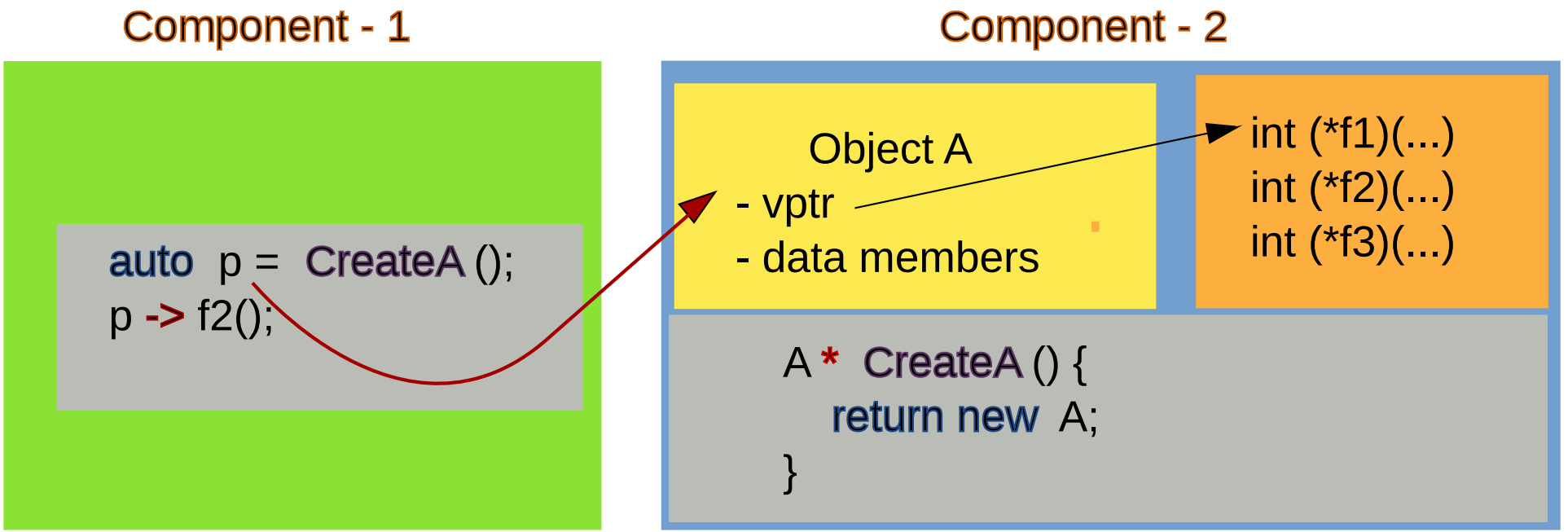
CIB Overview



Big problem of C++ SDK

- Sharing of:
  - Object layout
  - Virtual table
  - Typeid / RTTI

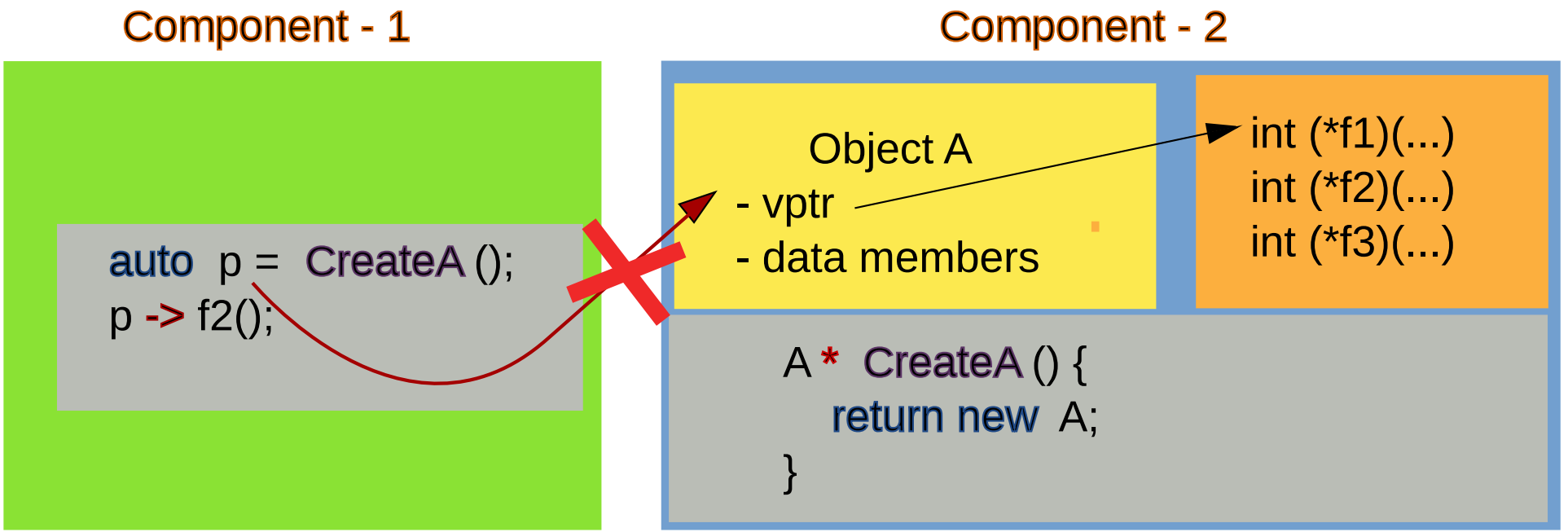
  across components.



CIB's solution to big problem

- Stop sharing:
  - Object layout
  - Virtual table
  - Typeid / RTTI

  across components.



Solution Overview

- Bridge pattern across DLL/SO boundary.
- Delegation using C style functions.

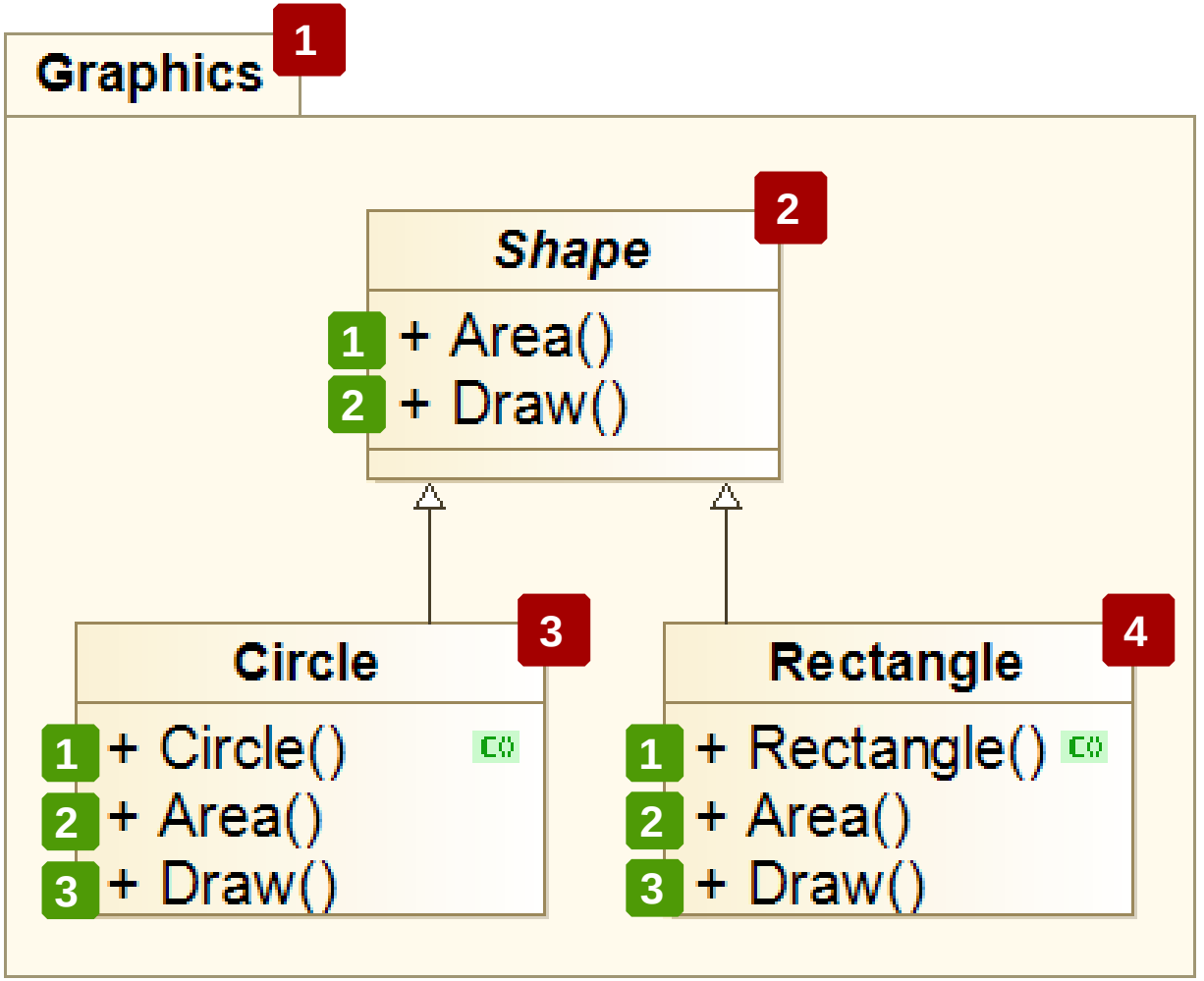
```
// CIB generated SDK header for client
struct HANDLE;
class Circle : public Shape {
public:
  Circle(int r, int Ox, int Oy)
    : handle(Circle_New(r, Ox, Oy))
  {}
  float Area() const {
    return Circle_Area(handle);
  }
private:
  HANDLE* handle;
};
```

DLL/SO Boundary

```
// Library code
class Circle : public Shape {
public:
  Circle(int r, int Ox, int Oy)
    : mRadius(r), mOx(Ox), mOy(Oy)
  {}
  float Area() const {
    return 3.1416*mRadius*mRadius;
  }
private:
  int mRadius, mOx, mOy;
};
```

CIB Architecture

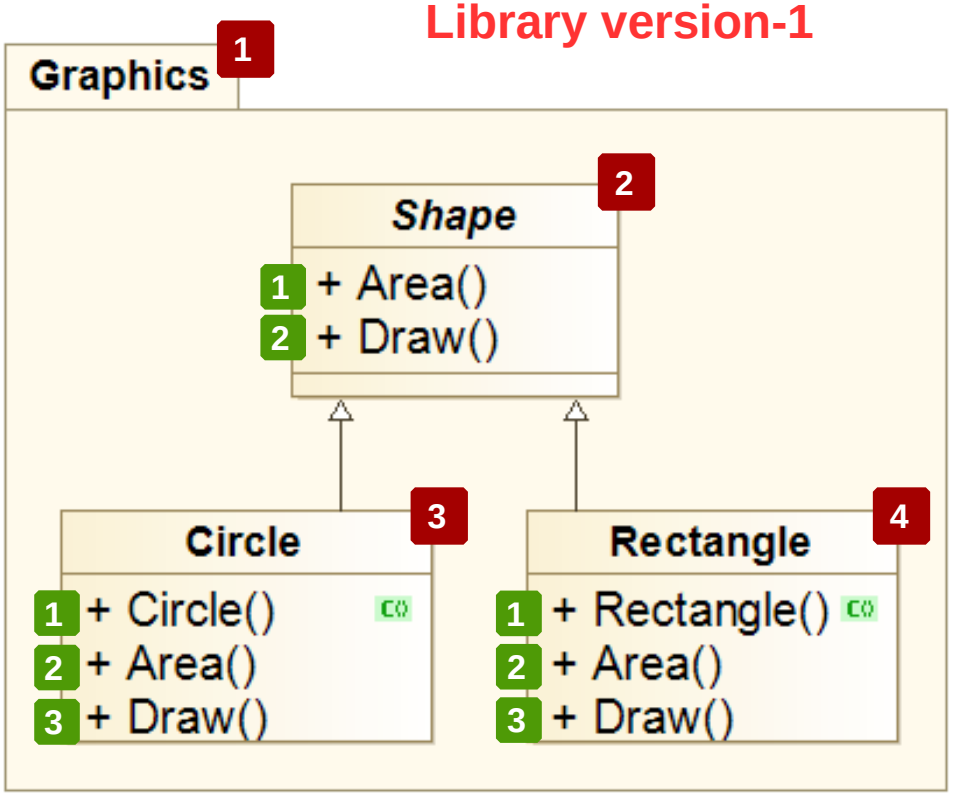
- All entities are numbered.



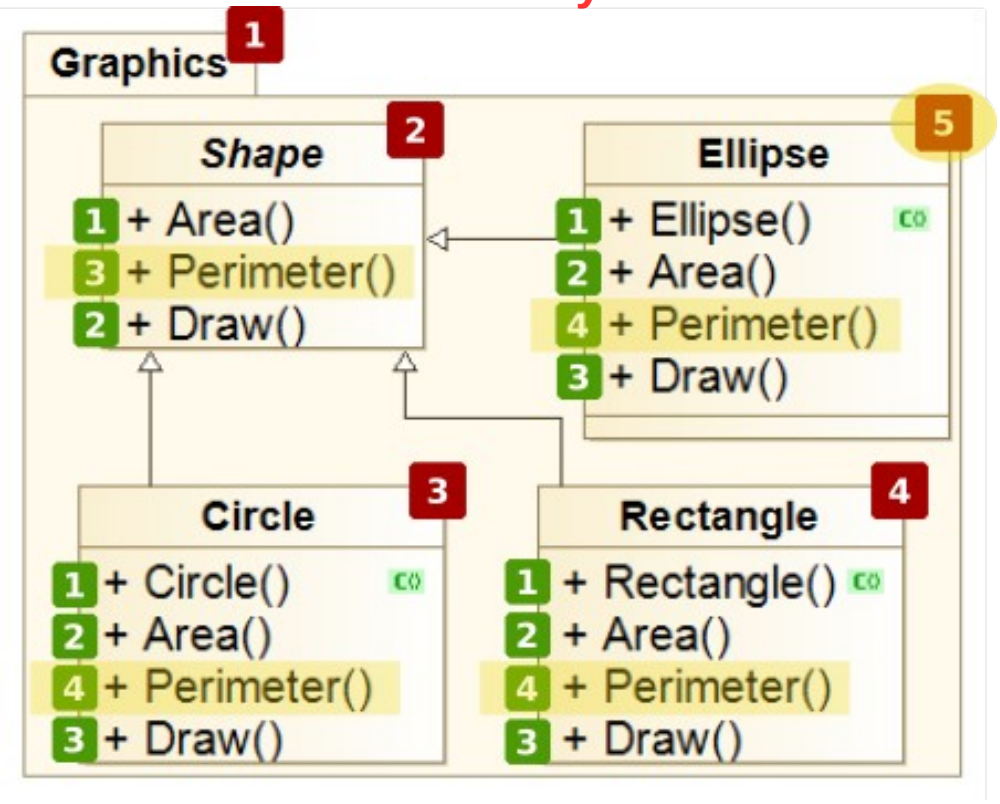
CIB Architecture

- Numbers remain same across releases.

Library version-1

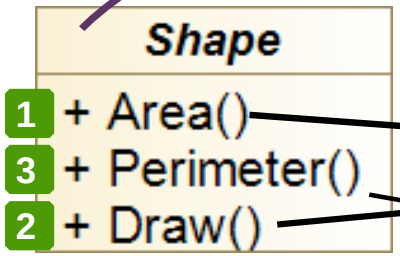


Library version-2



CIB Architecture

- For every class a method table is generated.



Library Glue Code

```
float Shape_Area(Shape* shape) {
    return shape->Area();
}
float Shape_Perimeter(Shape* shape) { ... }
void Shape_Draw(Shape* shape) { ... }

using MethodEntry = void(*)();
static const MethodEntry shape_MethodTable[] = {
    (MethodEntry) nullptr,
    (MethodEntry) Shape_Area,
    (MethodEntry) Shape_Draw,
    (MethodEntry) Shape_Perimeter
};

using MethodTable = const MethodEntry*;
MethodTable Shape_GetMethodTable() {
    return shape_MethodTable;
}
```

CIB Architecture

- Library glue code has function to return method table for given class ID.

```
extern "C" DLLEXPORT
MethodTable GraphicsLib_GetMethodTable(int classId) {
    switch(classId) {
        case 2: return Shape_GetMethodTable();
        case 3: return Circle_GetMethodTable();
        case 4: return Rectangle_GetMethodTable();
        case 5: return Ellipse_GetMethodTable();
    }
    return nullptr;
}
```

CIB Architecture

- On client side, class methods delegate calls to functions of method table.

```
struct HANDLE;
class Circle : public Shape {
public:
    Circle(int r, int Ox, int Oy)
        : handle(Circle_New(r, Ox, Oy)) {
    }
    float Area() const {
        return Circle_Area(handle);
    }
    ...
private:
    HANDLE* handle;
};
```

```
MethodTable GetMethodTable() {
    static mtbl = GraphicsLib_GetMethodTable(3);
    return mtbl;
}
HANDLE* Circle_New(int r, int Ox, int Oy) {
    return GetMethodTable()[1](r, Ox, Oy);
}
float Circle_Area(const HANDLE* circ) {
    return GetMethodTable()[2](circ);
}
...
```

CIB Architecture

- Similar but different arrangements are made to let Library call interface implemented by client.

CIB Architecture Principle

- Only PODs and Method Tables are shared between components.

DEMO #2 Library calling client

- Client implementation of interface is used by library.

DEMO #3 Loosely coupled inheritance

- Making non-breaking inheritance change doesn’t enforce compilation of client.

Recap of CIB Benefits

- Clients don’t need recompilation when **non-breaking changes** are done:
  - Change in data member.
  - Change in virtual table.
  - Change in inheritance.
- Client and Library can chose to support backward compatibility.
- Client and Library can be built using different compilers.
- CIB can inform when breaking changes are done.

Thanks

That’s it!