Designing C++ library interfaces

Introduction

Library encapsulation

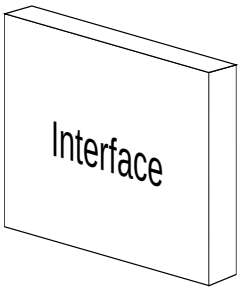
Subtyping, subclassing, and delegating

Using callbacks for wiring

Observations

Encapsulation of an object

Client



Implementation

Specify object interface abstractly

```
// OMG –Object Management Group
// IDL – Interface Definition Language

interface Window {
   void map();
   void unmap();
   boolean is_mapped();
   void raise();
   void lower();
};
```

Compiler-generated header file

```
class WindowType :
   public BaseObjectType {
protected:
   WindowType();
   ~WindowType();
public:
   virtual void map() /* = 0 */;
   virtual void unmap();
   virtual Boolean is_mapped();
   virtual void raise();
   virtual void lower();
};
```

More IDL compiler tricks

```
typedef WindowType* WindowRef;

// Refcount'd pointer class
class Window {
protected:
   WindowRef _p;
public:
   Window() { _p = 0; }
   Window(WindowRef p) { _p = p; }
   ~Window() { _release(_p); }
   operator WindowRef() { return _p; }
   WindowRef operator ->() { return _p; }
   // other stuff
};
```

Abstract interfaces

All virtual functions

No data members

No protected members

Biases subclassing over instancing

Like public for maintenance

Object-oriented programming

Data abstraction

One implementation at any one time, but can be changed without modifying caller

Polymorphism

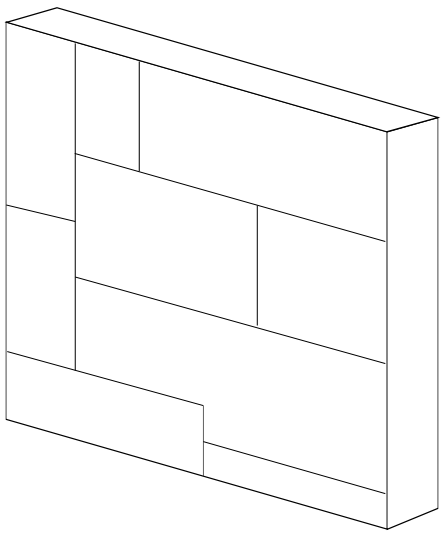
Multiple implementations at once

More flexible – distribution, evolution

Trend is polymorphism for most interfaces

Encapsulation of a library

Client



Implementation

Library encapsulation issues

interface(objects) != (object interfaces)

Hide implementation classes

Create objects indirectly

Implementation classes are not abstract

Protected, non-virtual, data

Domain: User interfaces

Applications should be easy to use

Interfaces are hard to build

Direct manipulation => object-oriented

Views of complex algorithms and data

InterViews = Inter*active* Views

Glyphs – lightweight, shareable objects
Uniform model for UI, text, graphics

Resolution-independence (printing)

Document-quality layout

Drawing editor framework (Unidraw)

C++ class library on top of X

Fresco – standards effort

Standard object model

Distributed objects

MT-aware (supports concurrency)

Resolution independence

Graphical embedding

Creating objects

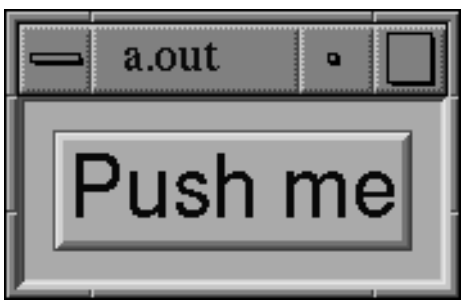
- *Subclassing vs. composition may be an implementation choice*
- *Use kit objects instead of constructors*
- *Eliminate/hide implementation classes*
- *Maintain state for remote objects*

Example 1: Push button

Button class defines interface

Push button has particular look+feel

Is push button a subclass or instance?
(could be either)



WidgetKit – create buttons, etc.

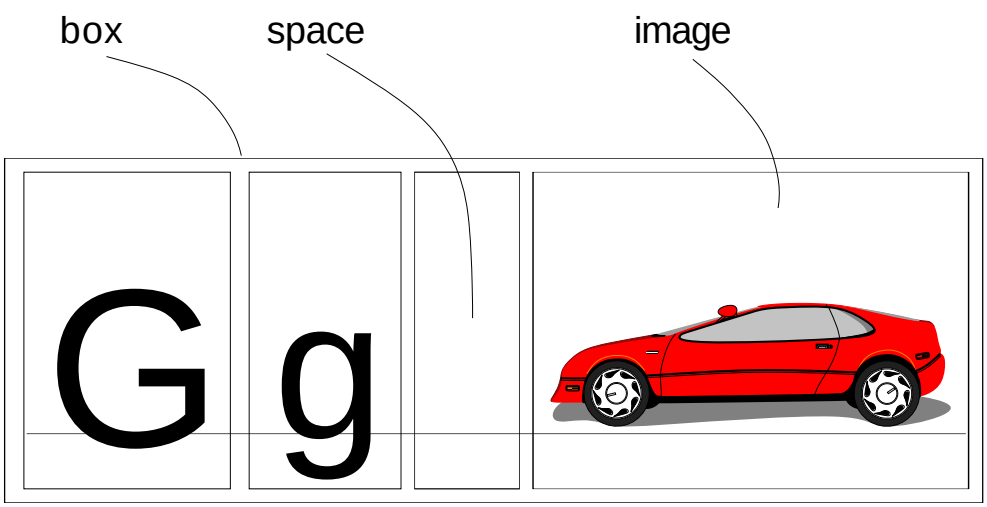
```
interface WidgetKit {
   ...
   Button push_button(
      in CharString s, in Action a
   );
   ...
};
```

b = widgets->push_button(...);

Hides subclass versus instance choice

Implementations offer alternate GUIs

Example 2: Layout objects



Layout example (cont'd)

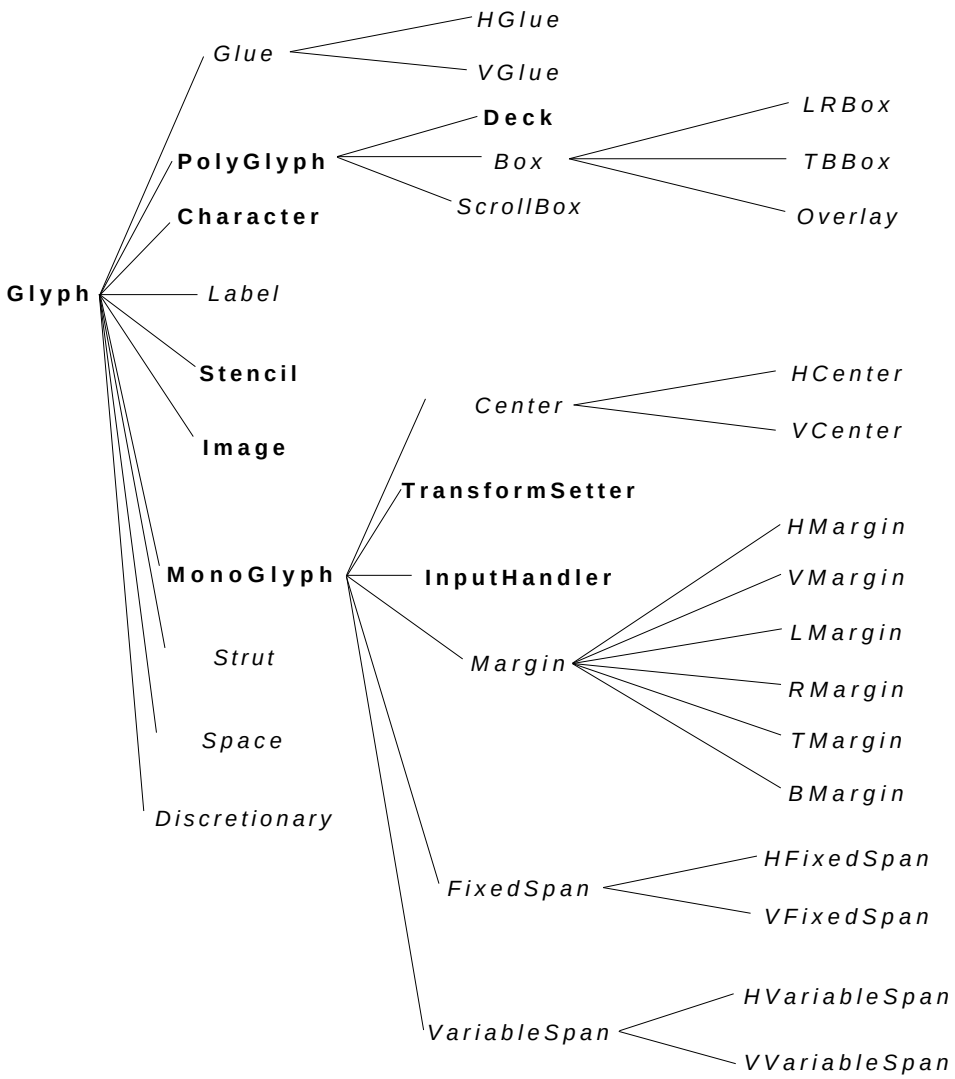
Original:

```
VGlue::VGlue() : Glue() { }
VGlue::VGlue(Coord natural) : Glue(
   Dimension_Y, natural, fil, 0, 0
) { }
VGlue::VGlue(
   Coord natural, Coord stretch, Coord shrink
) : Glue(Dimension_Y, natural, stretch, shrink, 0) { }
```

New Improved:

```
Glyph LayoutKitImpl::vglue() { return vglue(0); }
Glyph LayoutKitImpl::vglue(Coord natural) {
   return vglue(natural, fil, 0);
}
Glyph LayoutKitImpl::vglue(
   Coord natural, Coord stretch, Coord shrink
) {
   return new Glue(
      Dimension_Y, natural, stretch, shrink, 0
   );
}
```

Layout classes



Efficiency observations

Fewer classes means fewer vtbls

Runtime costs are comparable

Can add subclasses later to optimize
e.g., vglue with natural size of zero

Avoids constructor conversions

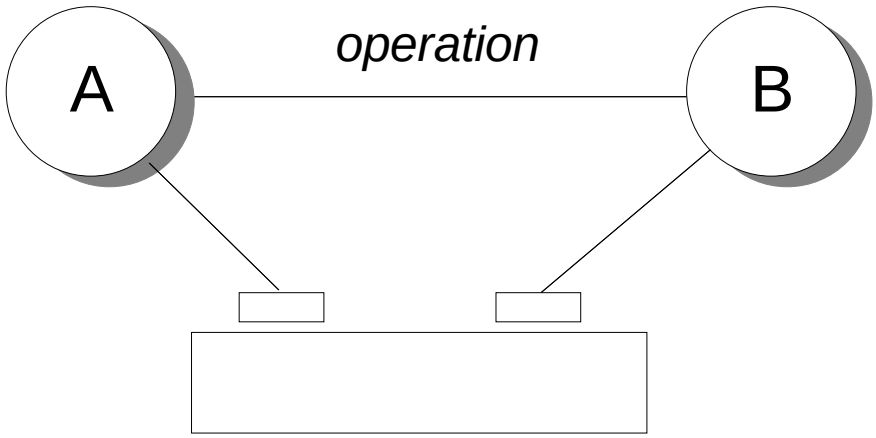
```
class Semaphore {
public:
   Semaphore(int);
   void wait();
   void signal();
};

Semaphore mutex(1);
mutex = 1; // eek!

// Using a kit
SemaphoreRef mutex;
mutex = thread_kit->mutex();
// BaseObject::operator = is private
```

Allow distributed objects

Perform an operation on a remote object



No concept of global functions or data

Kits can maintain convenient state

Inheritance

Subtyping – definitely “is a”

Subclassing – implementation sharing

Delegating – implementation forwarding

Inheritance is good

- *Reuse*
- *Fewer lines of code (factoring)*
- *Heterogeneous data structure with homogeneous operations*
- *E.g., objects that draw*

Inheritance is bad

- *Scatters functionality*
- *Creates dependencies, surprises*
- *Hard to change*

Composition is often better

```
interface Button : Viewer {
   // inherits Glyph body
   attribute TelltaleObj telltale;
   attribute Action click_action;
      ...
};
```

Separate appearance and action

Button interprets input events

Simpler, more flexible than subclassing

Requires relatively cheap objects

Subtyping

Interface inheritance

Subtype requires base operations

Think in terms of instances

Examples
- *Viewer is a Glyph*
- *Button is a Viewer*

Multiple inheritance is rare

Subclassing

Implementation inheritance

Not visible in library interface
Easier to change

Virtual bases right but expensive

Avoid multiple inheritance because of current implementations

Multiple inheritance example

```
class BaseType
class AType : public virtual BaseType
class BType : public virtual AType

class Aimpl : public virtual A
class Bimpl : public virtual B, public virtual Aimpl
class Cimpl : public virtual Bimpl

sizeof(AType) = 4
sizeof(BType) = 24
sizeof(Aimpl) = 24
sizeof(Bimpl) = 68
sizeof(Cimpl) = 92
```

Delegating

Subobject implements some operations

May require forwarding code and calls

Can’t get back to original object
Ok for implementations

Example: Bimpl contains Aimpl
Bimpl::f() calls a.f()
sizeof(Bimpl) = 8
Smaller code, data size

Bigger example from Fresco

StyleObj Glyph::style()

Viewer : Glyph (is_a)

Viewer 1:1 with StyleObj (has_a)

ViewerImpl

- *Implements Viewer interface*
- *Delegates to glyph implementation*
- *Delegates to style implementation*
- *Viewer::style() returns subobject*

Using callbacks

Single operation (e.g., menu item action)

No reason to augment behavior (super::execute)

Ok to have extra object (reference counting)

Example: Actions

```
interface Action {
   void execute();
   boolean is_reversible();
   void unexecute();
};
```

Instantiating actions

```
template<class T>
class ActionCallback {
public:
   ActionCallback(T*, void(T::*)());
   virtual ~ActionCallback();

   virtual void execute();
   virtual Boolean is_reversible();
   virtual void unexecute();
private:
   T* obj_;
   void (T::*func_)();
};
```

C++ observations

Powerful

- *Inheritance, virtual functions*
- *Data abstraction*

Dangerous

- *Inheritance, conversions, default asg*

Efficient, portable

- *Better than C, Modula-3, CLOS, Smalltalk, Ada, Pascal, ...*

Inefficient

- *Programming environment*
- *Static objects*

Conclusions

Separate interface and implementation
Library as a whole

Sharing interfaces
Subtype, instance

Sharing implementations
Subclass, delegate

Use objects to create objects

Kits organize library functionality