C++:
What to Use, What to Avoid

Mark Linton
Silicon Graphics
linton@sgi.com

1. Background
2. Classes and libraries
3. Inheritance and multiple inheritance
4. Distributed objects
5. Concluding remarks

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

Platform-independence

Drawing editor framework

C++ class library on top of X

Drawing editor

Document editor

./intro.doc

File Edit Font Size Color Alignment Macro Item Special

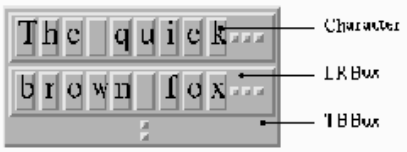
Glyphs:
Flyweight Objects for User Interfaces

6 Introduction

Most user interface toolkits are object-oriented because program objects are a natural way to represent the objects that a user manipulates. However, current toolkits have shied away from a whole-hearted commitment to the object-oriented model. These toolkits provide objects such as buttons and menus that let programmers build interfaces to application commands, but they do not provide objects for building interfaces to application data. Without this support, programmers must often define many new components from scratch; the savings afforded by the toolkit is offset by the cost of handcrafting the new components.

6.1 Overview

To offer the full benefits of an object-oriented model, a toolkit must encourage programmers to use objects for even the smallest components in the interface. Current toolkits let programmers build interfaces that contain hundreds of objects. But to reflect the fine-grained structure in application data, programmers may need hundreds of thousands of objects. For example, text is logically composed of characters, so a text view should be built from objects that represent individual characters. A moderate-sized document, such as a technical paper or a chapter in a book, will need at least 50,000 character objects.



TextView object structure

6.2 Other toolkits

Most user interface toolkits are object-oriented because program objects are a natural way to represent the objects that a user manipulates. However, current toolkits have shied away from a wholehearted commitment to the object-oriented model. These toolkits provide objects such as buttons and menus that let programmers build interfaces to application *commands*, but they do not provide objects for building interfaces to application *data*. Without this support, programmers must often define many new components from scratch; the savings afforded by the toolkit is offset by the cost of handcrafting the new components.

This approach can dramatically simplify an implementation, but it is practical only if objects are simple and cheap. The combination of faster hardware and more efficient object-oriented languages has now made it possible to use objects at this finer level of granularity without sacrificing interactive response.

We have designed a set of "flyweight" components, called glyphs, that are simple and efficient. The Glyph base class defines a protocol for drawing; subclasses define specific appearances such as graphic primitives (lines and

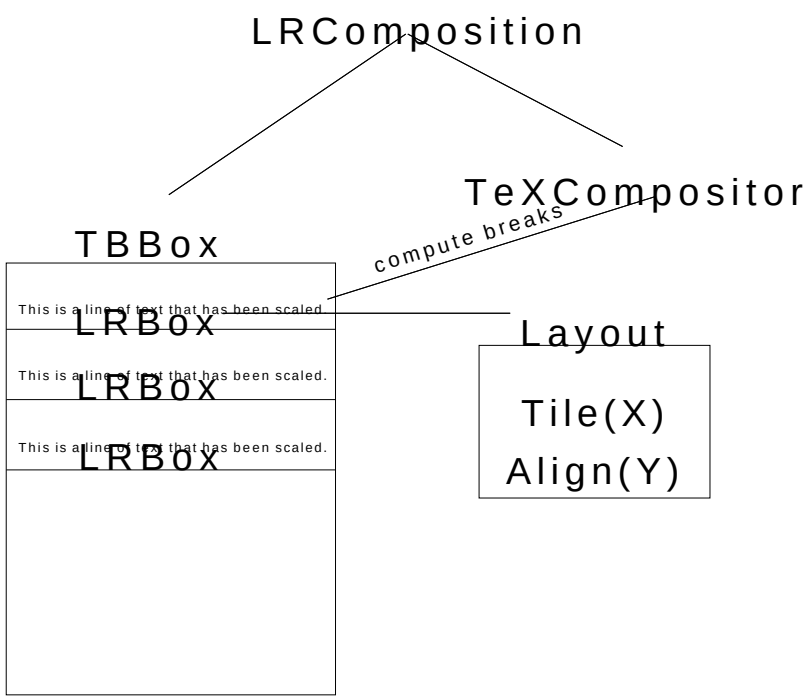
```
TextView::TextView(FILE* file) {
  TBBox* page = new TBBox();
  LRBox* line = new LRBox();
  int c;
  while ((c = getc(file)) != EOF) {
    if (c == '\n') {
      page->append(line);
      line = new LRBox;
    } else {
      line->append(new Character(c));
    }
  }
  body(page);
}
```

A TextView built from glyphs

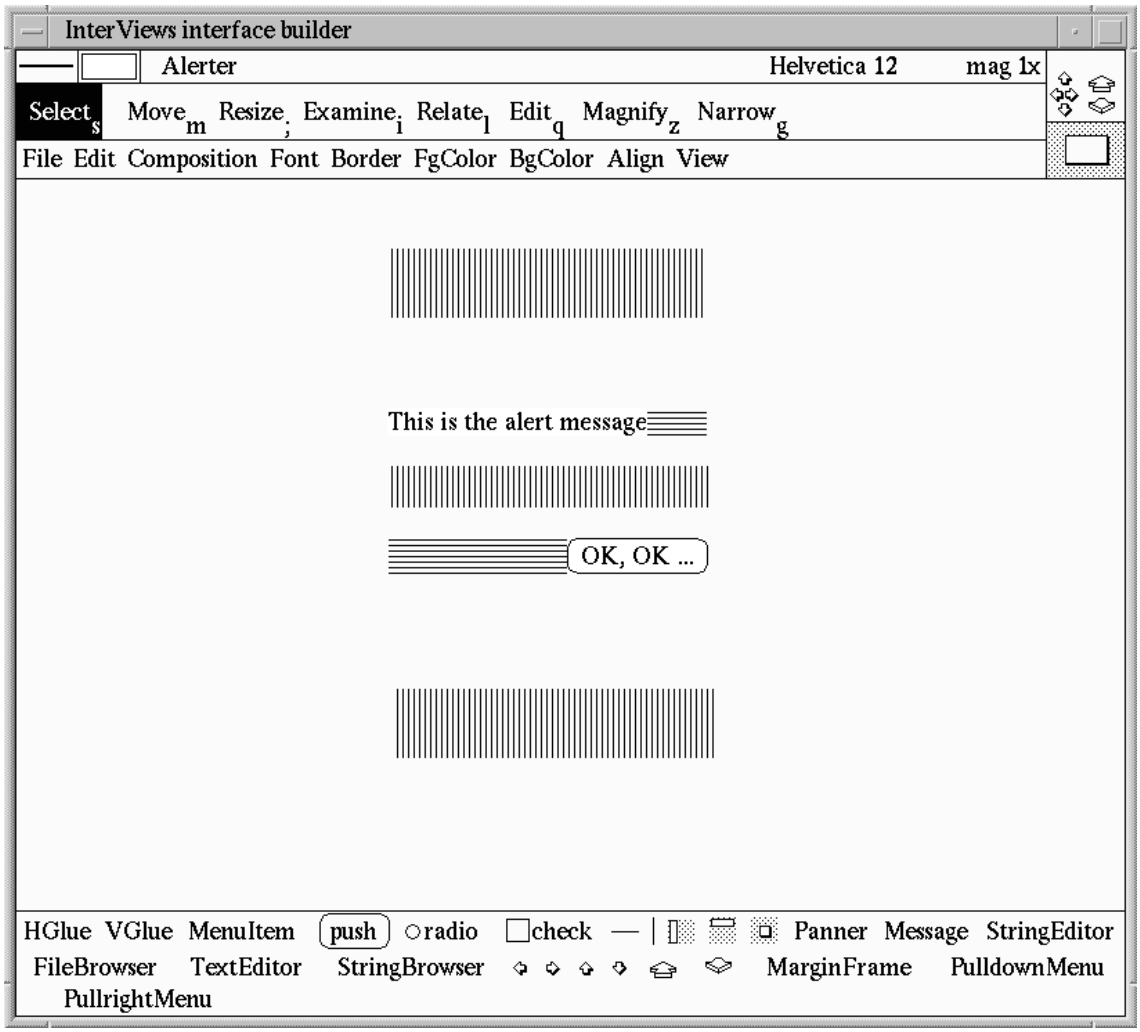
Request	**preference**	10
Allocates	size	20
Draw	appearance	30

Glyph protocol

Objects in a column



Interface builder



Current status

- 200+ available classes
- 100,000+ lines of C++
- Publicly available distribution
- Runs on all Unix/X workstations
- In use at thousands of sites world-wide
- Evolving toward standard

History

Research project at Stanford in late ’80s

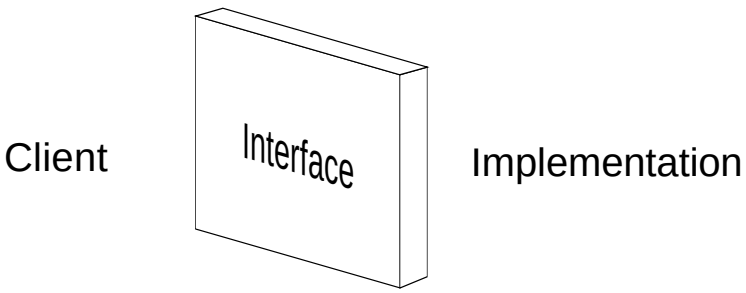
10 releases over 5 years

3-4 developers

95% of implementation is different
from the first release

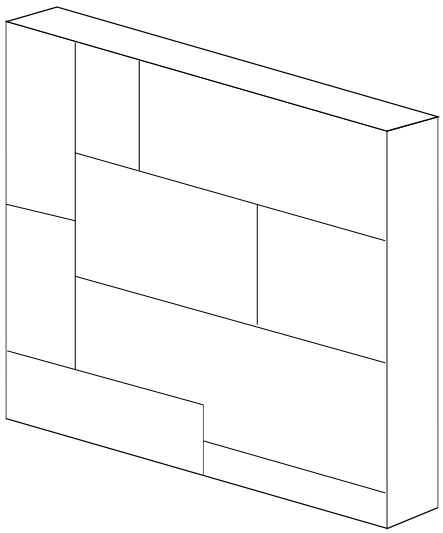
Typical release characteristics:
30% of implementation is different
99% source upward compatibility
10% classes become obsolete

Encapsulation of an object



Encapsulation of a library

Client



Implementation

Library encapsulation is hard

(Encapsulated objects)
!=
Encapsulated (objects)

Creating objects

- Subclassing vs. instancing may be an implementation choice
- Use kits instead of constructors
- Eliminate/hide classes
- Simplify constructors

Example: Layout objects

Original:

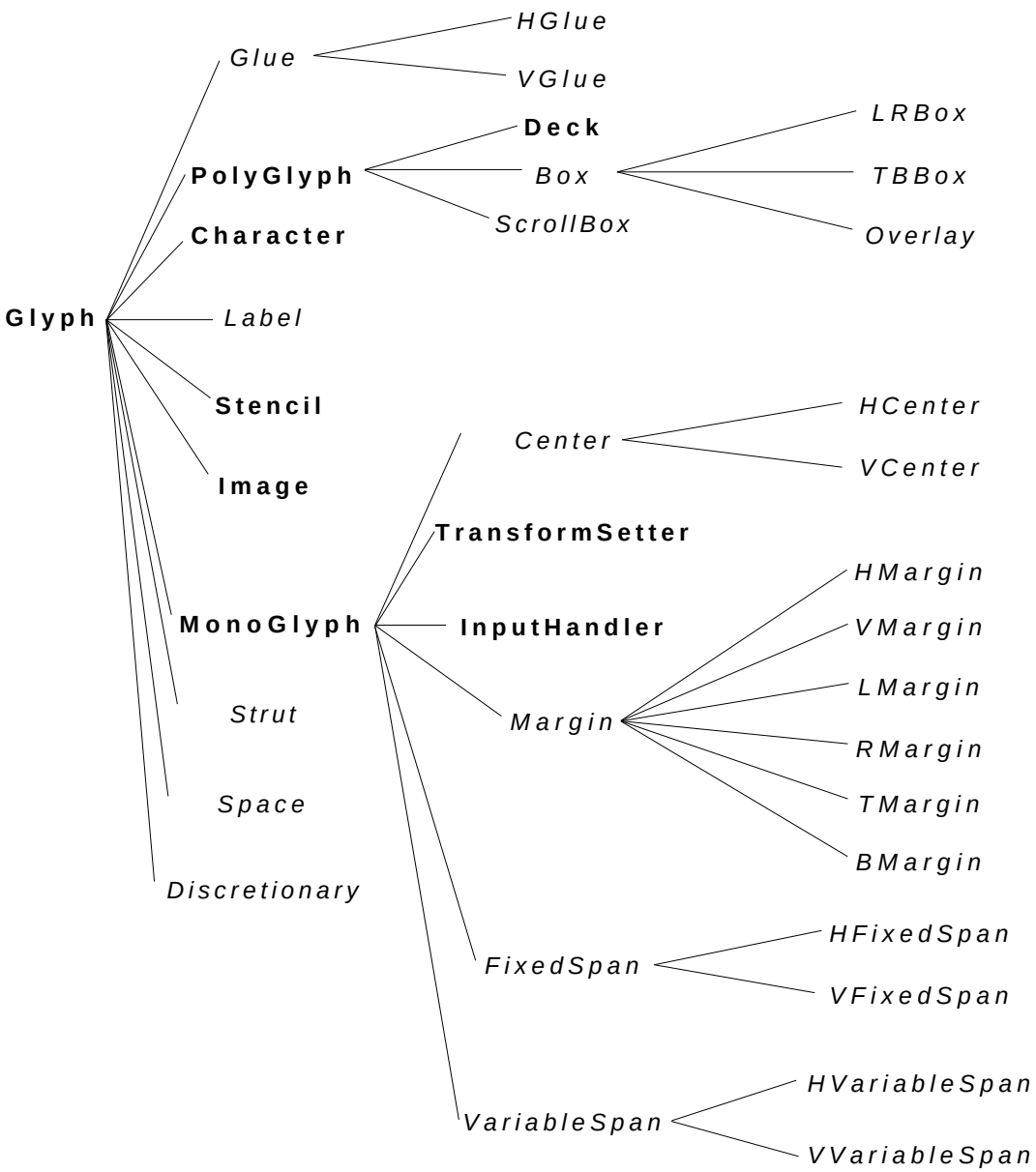
```
VGlue::VGlue() : Glue() { }
VGlue::VGlue(Coord natural) : Glue(
  Dimension_Y, natural, fil, 0, 0
) { }
VGlue::VGlue(
  Coord natural, Coord stretch, Coord shrink

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

New Improved:

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

Layout classes



Member access

- Data is private
  - Access functions
  - Compatibility – easier to change implementation than usage
- Protected biases subclasses over clients
- Protected is like public for maintenance
- Access functions can be inline
- Separate read/write access protection

Simple naming convention

Upper case for types

Lower case for everything else

Leading/trailing “_” for member data

Example:

```
class TypeName {
public:
   float x() const;
protected:
   void x(float);
private:
   float x_;
};
```

Inline functions

- Good for time-critical functions
- Bad for most functions
- Can cause extra recompilation
- Restrict binary compatibility
- Less than 5% of InterViews functions are inline

Const member functions?

```
boolean Transformer::identity() const {
    if (changed_) {
        Transformer* t = (Transformer*)this;
        t->identity_ = compute_identity();
    }
    return identity_;
}
```

Class interface should not depend
on implementation

Either classes do not contain const
Or implementation may need to cast **this**

Use of const is a coding convention

Ambiguous conversion operators

```
class MyNumber {
public:
   MyNumber(long);
   MyNumber(double);
   operator long() const;
   operator double() const;
   MyNumber operator +(MyNumber);
};

MyNumber n, m;
n = 1 + m;      // ambiguous!
```

Order conversions without circularity

Better yet: don’t use conversion ops

Beware constructor conversions

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

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

Solution is to protect assignment

```
class Semaphore {
public:
    ...
  Semaphore& operator =(
    const Semaphore&
  );
private:
  void operator =(long);
};
```

Or use a kit

```
Semaphore* semaphore(long count);
```

Summary: Hide library implementation

Use kits to abstract object creation

Allow subclass/instance tradeoffs

Eliminate/hide classes that do not add protocol

cheap => immutable => no protocol

Simplifies and organizes functionality

Inheritance and Multiple Inheritance

When is inheritance appropriate?

When is inheritance undesirable?

Is multiple inheritance necessary?

Reasons to use inheritance

- Redefining virtuals (polymorphism)
  - Heterogeneous data structure
  - Homogeneous protocol
  - Example: objects that draw
    - Button, ScrollBar, Polygon, ...
- Sharing implementation (reuse)
  - Polymorphism is external “reuse”

Reasons NOT to use inheritance

Scatters functionality

Creates header file dependencies

Might cause unexpected results

Hard to undo in next release

Composition is usually better

```
class Button : public Glyph {
public:
   Button(Telltale*, Action*);
      ...
};
```

Separate appearance and action

Button interprets input events

Simpler, more flexible than subclassing

Justify inheritance with instances

- Design protocols between instances
- Look for common protocols
- Make sure commonality will continue
- Mold into classes later

Example: Scrolling protocol

Adjustable is receiver

Adjuster is sender

Want to be able to reuse senders

Want to be able to write new receivers

Need polymorphism

Counter-example: Aggregate glyphs

Aggregates have a list of children

Could derive glyph from list

But glyphs are not used as lists

Might not want list protocol changes

Has-a relationship, not *is-a*

Multiple inheritance

- Think about instances
- Justify inheritance along both paths
- Justify not splitting into two objects
- MI is useful but rare in InterViews
- Example: ScrollBox is derived from PolyGlyph and Adjustable

When to use virtual base classes

Non-virtual semantics usually wrong
- class Resource defines refcount
- class A : public Resource
- class B : public Resource
- class C : public A, public B
- C gets two refcounts!
- A and B must define Resource
  - as virtual base

Can use non-virtual as hint to avoid MI

Example:
- No reason for MI among glyphs

Callbacks

```
class Action : public Resource {
public:
   virtual void execute() = 0;
};

#define ActionCallback(T)   . . .

ActionCallback(T)::ActionCallback(T)(
   T* obj, void (T::*func)()
) {
   obj_ = obj;
   func_ = func;
}

void ActionCallback(T)::execute() {
   (obj_->*func_)();
}
```

When to use callbacks

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

No reason to augment behavior
(never call base::execute)

Ok to have extra object
(distinct reference count)

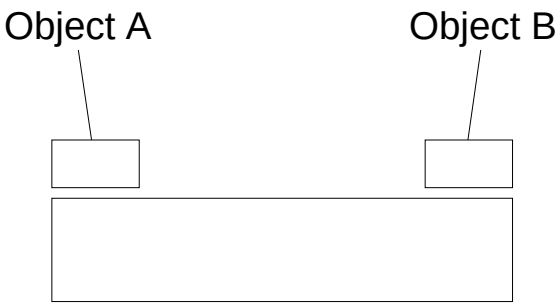
Distributed objects

Interface Definition Language (IDL)

References to remote objects

Short-circuit for local objects

Bindings for multiple languages



OMG’s IDL

- Specify interfaces, not implementation
- Attributes for access functions
- No data, protected, private
- No overloading, operators
- No references, const
- No inlines

Example

```
typedef long GlyphIndex;

interface Glyph : Resource {
   void request(in Requisition r);
   void draw(in Canvas c, in Allocation a);
      ...
   GlyphIndex count();
   Glyph component(GlyphIndex i);
};
```

Translation to C++:

```
class Glyph : public Resource {
public:
   virtual void request(Requisition*) = 0;
      ...
};
```

Some implications of distribution

- Hard to isolate what is distributable
- Better specification mechanism
- Support more multiple languages
- Define implementations as subclasses
- Hard to inherit implementations
- No references, const, operators, ...
- Cannot access parameter representation

C++: What to use

- Classes, access functions
- Inheritance for polymorphism
- Virtual for most member functions
- Conventions for const and references
  - Abstract const
  - Ref param when callee
    - will not store pointer
- Friends for restricting access
- Indirect representation for flexibility

C++: What to (try to) avoid

Public data

Protected members

Multiple, protected, or private inheritance

Inline functions

Operators, especially conversions

Default assignment

Abstract -> abstract cast

C++: What is missing

Experience with templates

Class name scopes

```
#include <InterViews/enter-scope.h>
#define Glyph ivGlyph
. . .
```

Garbage collection

Multiple dispatch

```
draw(virtual Glyph*, virtual Canvas*);
```

Concluding remarks

C++ offers the power

Programmer provides the brains

Think–design–code–repeat

Build on experience and code of others

Generalize solutions, not problems