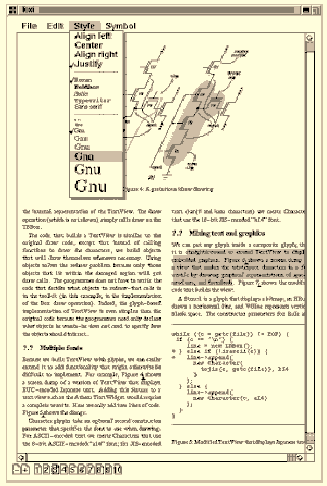
Lexi Case Study

- A WYSIWYG document editor.
- Mix text and graphics freely in various formatting styles.
- The usual
  - Pull-down menus
  - Scroll bars
  - Page icons for jumping around the document.
- Going through the design, we will see many patterns in action.
- History: Ph.D. thesis of Paul Calder (s. Mark Linton) 1993

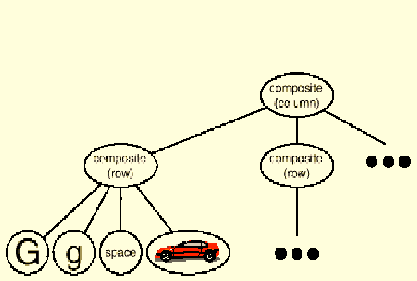


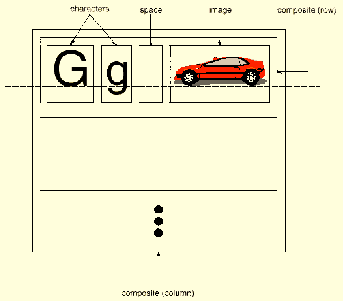
Document Structure

- A hierarchical arrangement of shapes.
- Viewed as lines, columns, figures, tables, …
- UI should allow manipulations as a group
  - E.g. refer to a table as a whole
- Internal representation should support
  - Maintaining the physical structure
  - Generating and presenting the visuals
  - Reverse mapping positions to elements
- Want to treat text and graphics uniformly
- No distinction between single elements or groups.
  - E.g. the 10th element in line 5 could be an atomic character, or a complex figure comprising nested sub-parts.

Recursive Composition

- Building more complex elements out of simpler ones.
- Implications:
  - Each object type needs a corresponding class
  - All must have compatible interfaces (inheritance)
  - Performance issues.



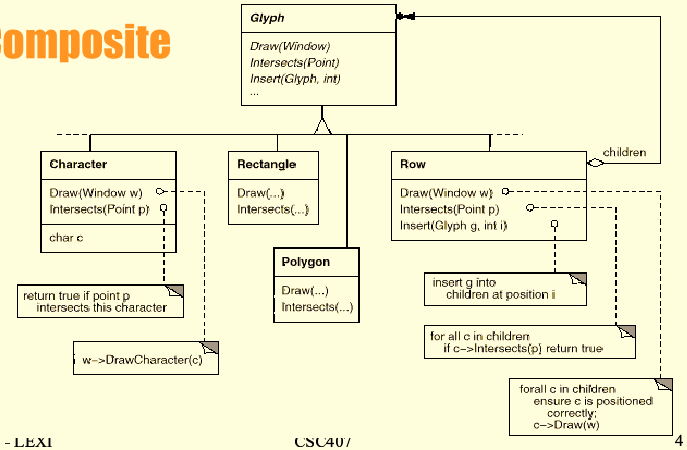


Glyph Class

An Abstract class for all objects that can appear in a document.

- Both primitive and composed.

Composite



Lexi Glyph Interface and responsibilities

```
public abstract class Glyph {
    // appearance
    public abstract void draw(Window w);
    public abstract Rect getBounds();
    // hit detection
    public abstract boolean intersects(Point);
    // structure
    public abstract void insert(Glyph g, int i);
    public abstract void remove(Glyph g);
    public abstract Glyph child(int i);
    public abstract Glyph parent();
}
```

- Glyphs know how to draw themselves
- Glyphs know what space they occupy
- Glyphs know their children and parents

Interviews 3.1 glyph

```
class Glyph : public Resource {
public:
    virtual void request(Requisition&) const;
    virtual void allocate(Canvas*, const Allocation&, Extension&);
    virtual void draw(Canvas*, const Allocation&) const;
    virtual void pick(Canvas*, const Allocation&, int depth, Hit&);

    virtual Glyph* component(GlyphIndex) const;
    virtual void insert(GlyphIndex, Glyph*);
    virtual void remove(GlyphIndex);
    virtual GlyphIndex count() const;
protected:
    Glyph();
}
```

several methods omitted!

Glyph and containers

- The Glyph class, from which all other drawable classes inherit, defines methods to access its parts, as if it were a container.
- However, although many Glyphs are containers many are not.
- Putting methods like `component(GlyphIndex)` into the base class is a canny realization that non-containers can often ignore component management messages.
- This can simplifies composition.

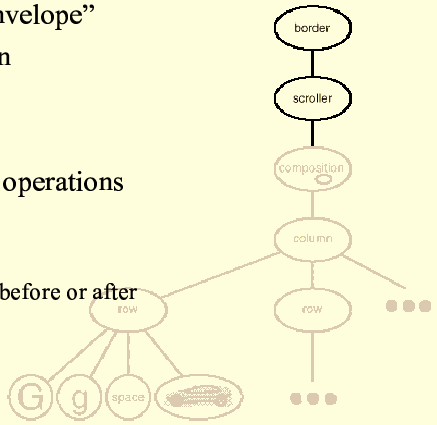
Embellishments

- Wish to add visible borders and scroll-bars around pages.
- Inheritance is one way to do it.
  - leads to class proliferation
    - BorderedComposition, ScrollableComposition, BorderedScrollableComposition
  - inflexible at run-time
- Will have classes
  - Border
  - Scroller
- They will be Glyphs
  - they are visible
  - clients shouldn't care if a page has a border or not
- They will be composed.
  - but in what order?

Transparent Enclosure

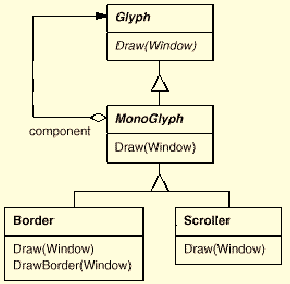
- also known as “letter-envelope”
- single-child composition
- compatible interfaces

- Enclosure will delegate operations to single child, but can
  - add state
  - augment by doing work before or after delegating to the child.



MonoGlyph

- Border calls { MonoGlyph.draw(); drawBorder(); }



Decorator

IV monoglyph.h

```
class MonoGlyph : public Glyph {
public:
    virtual ~MonoGlyph();

    virtual void body(Glyph*);
    virtual Glyph* body() const;
    void bodyclear();
  //remaining methods just like Glyph..
protected:
    MonoGlyph(Glyph* = nil);
private:
    Glyph* body_;
};
```


IV monoglyph.c

```
//apart from body management methods
//Monoglyph methods forward to body.

void MonoGlyph::draw(Canvas* c, const
Allocation& a) const {
    if (body_ != nil) {
        body_->draw(c, a);
    } else {
  Glyph::draw(c, a);
    }
}
```

MonoGlyph

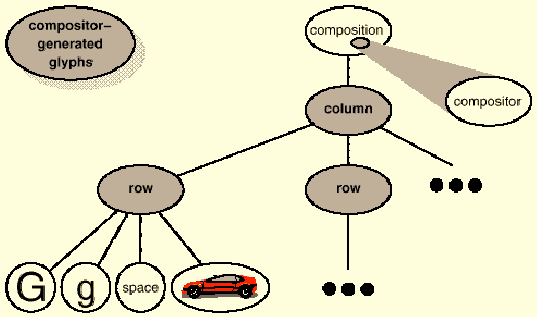
- Very often users need to tweak the behavior of a component by changing the behavior of only one method
- For instance, perhaps we need to draw a border around an existing Glyph.
- Monoglyph is a canny way of forwarding all methods to the "body" and overriding only what is needed.
- This is often called "interposition". We interpose a border between a Glyph and its container.
- For instance, to add a border we would override draw and the geometry negotiation methods to reserve space for the border.
- Purpose is to make it easier to assemble complex composites from relatively simple components.

Formatting

- Breaking up a document into lines.
  - Many different algorithms
    - trade off quality for speed
  - Complex algorithms
- Want to keep the formatting algorithm well-encapsulated.
  - independent of the document structure
    - can add formatting algorithm without modifying Glyphs
    - can add Glyphs without modifying the formatting algorithm.
- Want to make it dynamically changeable.

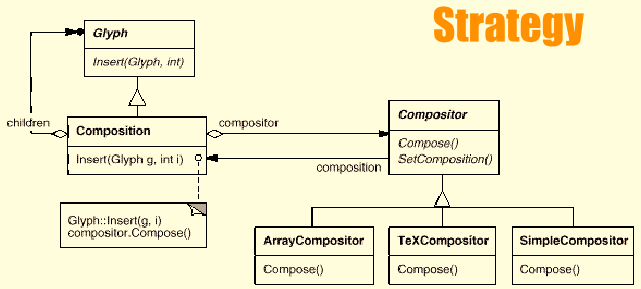
Composition & Compositor

- Initially, an unformatted Composition object contains only the visible child Glyphs.
- After running a Compositor, it will also contain invisible, structural glyphs that define the format.



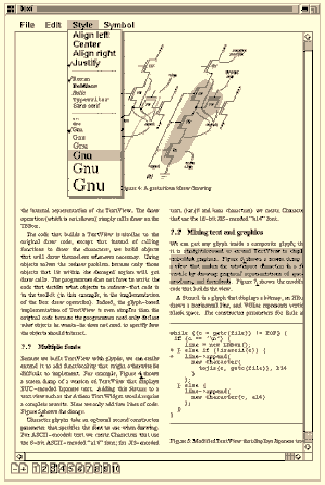
Compositor & Composition

- **Compositor** class will encapsulate a formatting algorithm.
- Glyphs it formats are all children of **Composition**



Supporting Multiple Window Systems

- Want the application to be portable across diverse user interface libraries.
- ***Every*** user interface element will be a Glyph.
- Some will delegate to appropriate platform-specific operations.



Multiple Look-and-Feel Standards

- Goal is to make porting to a different windowing system as easy as possible.
  - one obstacle is the diverse look-and-feel standards
  - want to support run-time switching of l&f.
  - Win, Motif, OpenLook, Mac, …
- Need 2 sets of widget glyph classes
  - abstract
    - ScrollBar, Button, …
  - concrete
    - MotifScrollBar, WinScrollBar, MacScrollBar, MotifButton, …
- Need indirect instantiation.

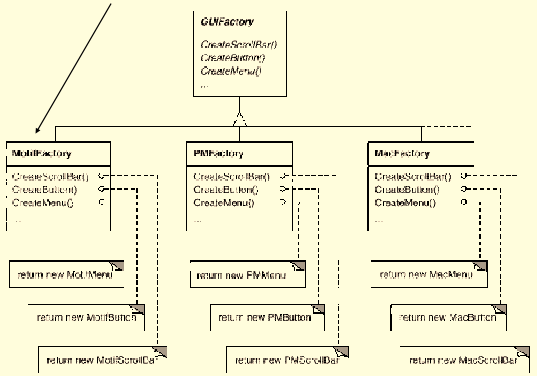
Object Factories

Usual method:

```
ScrollBar sb = new MotifScrollBar();
```

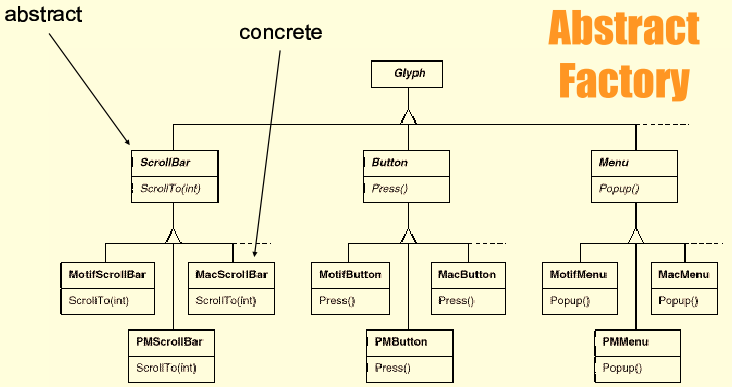
Factory method:

```
ScrollBar sb = guiFactory.createScrollBar();
```



Product Objects

- The output of a factory is a product.



Building the Factory

- If known at compile time (e.g., Lexi v1.0 – only Motif implemented).

```
GUIFactory guiFactory = new MotifFactory();
```

- Set at startup (Lexi v2.0)

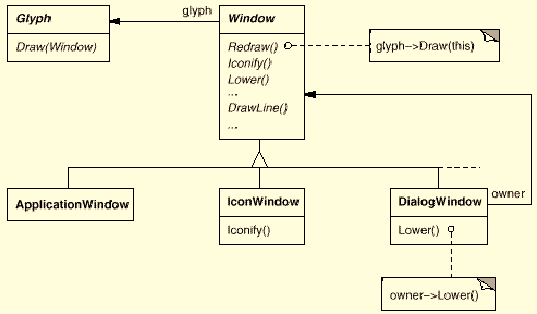
```
String LandF = appProps.getProperty("LandF");
GUIFactory guiFactory;
if( LandF.equals("Motif") )
  guiFactory = new MotifFactory();
...
```

- Changeable by a menu command (Lexi v3.0)
  - re-initialize 'guiFactory'
  - re-build the UI

Singleton

Multiple GUI Libraries

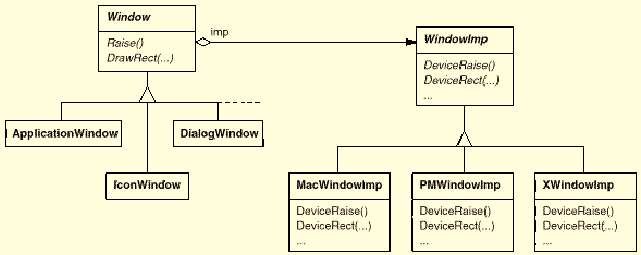
- Can we apply Abstract Factory?
  - Each GUI library will define its own concrete classes.
  - Cannot have them all inherit from a common, abstract base.
  - but, all have common principles
- Start with an abstract Window hierarchy (does not depend on GUI library)



Window Implementations

- Defined interface Lexi deals with, but where does the real windowing library come into it?
- Could define alternate Window classes & subclasses.
  - At build time can substitute the appropriate one
- Could subclass the Window hierarchy.
- Or …

Bridge



Window Implementation Code Sample

```
public class Rectangle extends Glyph {
  public void draw(Window w) { w.drawRect(x0,y0,x1,y1); }
  ...
}

public class Window {
  public void drawRect(Coord x0,y0,x1,y1) {
    imp.drawRect(x0,y0,x1,y1);
  }
  ...
}

public class XWindowImp extends WindowImp {
  public void drawRect(Coord x0,y0,x1,y1) {
    ...
    XDrawRectangle(display, windowId, graphics, x,y,w,h);
  }
}
```

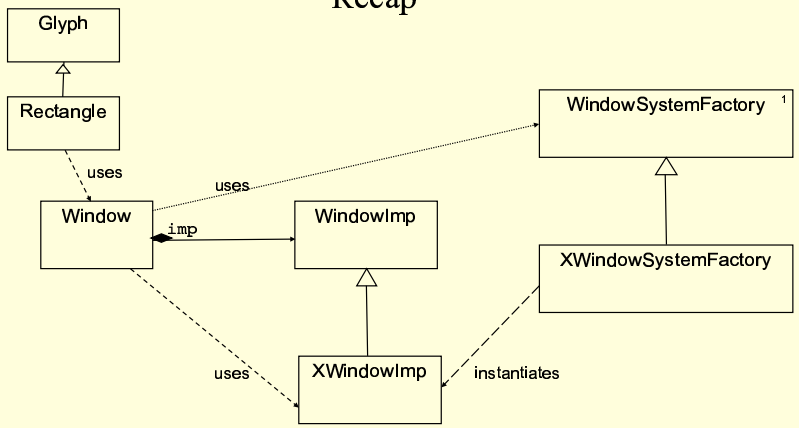
Configuring 'imp'

```
public abstract class WindowSystemFactory {
  public abstract WindowImp createWindowImp();
  public abstract ColorImp  createColorImp();
  ...
}

public class XWindowSystemFactory extends WindowSystemFactory {
  public WIndowImp createWindowImp() {
    return new XWindowImp();
  }
  ...
}

public class Window {
  Window() {
    imp = windowSystemFactory.createWindowImp();
  }
  ...
}
```

well-known object

Abstract Factory

Recap



User Operations

- Operations
  - create new, save, cut, paste, quit, …
- UI mechanisms
  - mousing & typing in the document
  - pull-down menus, pop-up menus, buttons, kbd accelerators, …
- Wish to de-couple operations from UI mechanism
  - re-use same mechanism for many operations
  - re-use same operation by many mechanisms
- Operations have many different classes
  - wish to de-couple knowledge of these classes from the UI
- Wish to support multi-level undo and redo

Commands

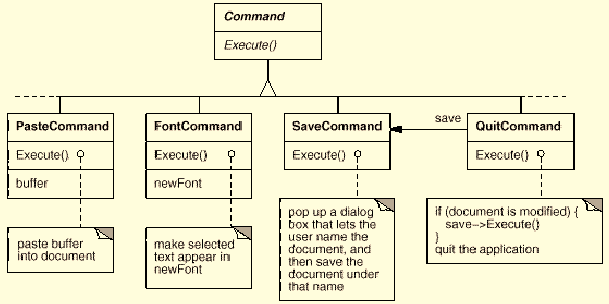
- A button or a pull-down menu is just a Glyph.
  - but have actions command associated with user input
  - e.g., MenuItem extends Glyph, Button extends Glyph, …
- Could…

  PageFwdMenuItem extends MenuItem

  PageFwdButton extends Button
- Could…
  - Have a MenuItem attribute which is a function call.
- Will…
  - Have a MenuItem attribute which is a command object.

Command Hierarchy

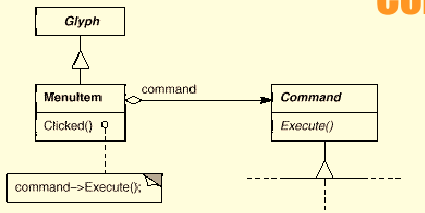
- Command is an abstract class for issuing requests.



Invoking Commands

- When an interactive Glyph is tickled, it calls the Command object with which it has been initialized.

Command



Unidraw Color change command

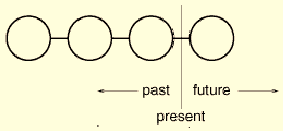
```
class ColorCmd : public Command {
public:
    ColorCmd(ControlInfo*, PSColor* , PSColor*);
    ColorCmd(Editor* = nil, PSColor*, PSColor*);

    virtual void Execute();
    PSColor* GetFgColor();
    PSColor* GetBgColor();

    virtual Command* Copy();
    virtual void Read(istream&);
    virtual void Write(ostream&);
    virtual ClassId GetClassId();
    virtual boolean IsA(ClassId);
protected:
    PSColor* _fg, *_bg;
};
```


Unidraw ColorCmd implementation

```
void ColorCmd::Execute () {
  ColorVar* colorVar = //current colour
  if (colorVar != nil) {
     PSColor* fg = (_fg == nil) ?
         colorVar->GetFgColor() : _fg;
     PSColor* bg = (_bg == nil) ?
         colorVar->GetBgColor() : _bg;
         colorVar->SetColors(fg, bg);
      }
      Command::Execute();
  }
```

Undo/Redo

- Add an `unexecute()` method to `Command`
  - Reverses the effects of a preceding `execute()` operation using whatever undo information `execute()` stored into the `Command` object.
- Add a `isUndoable()` and a `hadnoEffect()` method
- Maintain Command history:



Spell Checking & Hyphenation

- Textual analysis
  - checking for misspellings
  - introducing hyphenation points where needed for good formatting.
- Want to support multiple algorithms.
- Want to make it easy to add new algorithms.
- Want to make it easy to add new types of textual analysis
  - word count
  - grammar
  - legibility
- Wish to de-couple textual analysis from the Glyph classes.

Accessing Scattered Information

- Need to access the text letter-by-letter.
- Our design has text scattered all over the Glyph hierarchy.

- Different Glyphs have different data structures for storing their children (lists, trees, arrays, …).
- Sometimes need alternate access patterns:
  - spell check: forward
  - search back: backwards
  - evaluating equations: inorder tree traversal

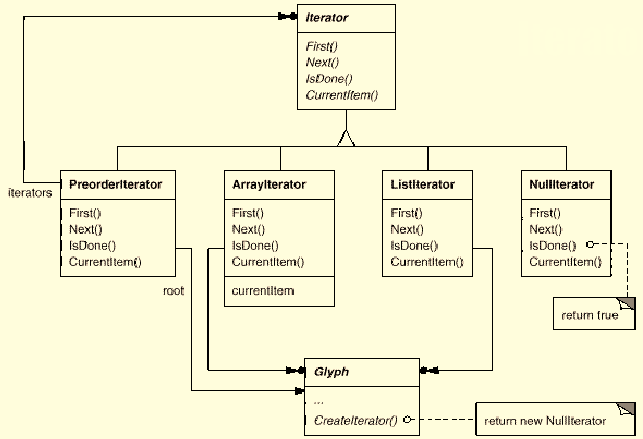
Encapsulating Access & Traversals

- Could replace index-oriented access (as shown before) by more general accessors that aren't biased towards arrays.

```
Glyph g = …
for(g.first(PREORDER); !g.done(); g->next()) {
  Glyph current = g->getCurrent();
  …
}
```

- Problems:
  - can't support new traversals without extending enum and modifying all parent Glyph types.
  - Can't re-use code to traverse other object structures (e.g., Command history).

Iterator Hierarchy



Using Iterators

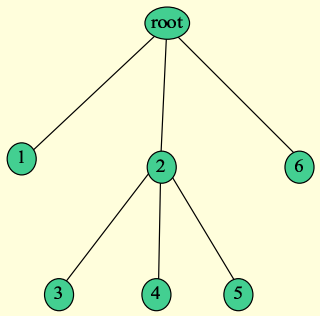
```
Glyph* g;
Iterator<Glyph*>* i = g->CreateIterator();
for (i->First(); !i->IsDone(); i->Next()) {
    Glyph* child = i->CurrentItem();
    // do something with current child
}
```

Note this is different in style than Java Enumeration and Iterator that want to move on to next element and fetch current in one "`nextElement`" method.

Initializing Iterators

```
Iterator<Glyph*>* Row::CreateIterator () {
    return new ListIterator<Glyph*>(_children);
}
```


Pre-order Iterator



Approach

- Will maintain a stack of iterators.
- Each iterator in the stack will correspond to a level in the tree.
- The top iterator on the stack will point to the current node
- We will rely on the ability of leaves to produce "null iterators" (iterators that always return they are done) to make the code more orthogonal.

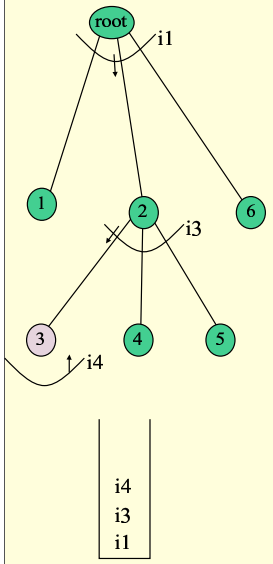
Implementing a Complex Iterator

```
void PreorderIterator::First () {
    Iterator<Glyph*>* i = _root->CreateIterator();
    if (i) {
        i->First();
        _iterators.RemoveAll();
        _iterators.Push(i);
    }
}

Glyph* PreorderIterator::CurrentItem () const {
    return _iterators.Size() > 0 ? _iterators.Top()-
  >CurrentItem() : 0;
}
```


Implementing a Complex Iterator (cont'd)

```
void PreorderIterator::Next () {
   Iterator<Glyph*>* i = _iterators.Top()->CurrentItem()->CreateIterator();
   i->First();
   _iterators.Push(i);
   while ( _iterators.Size() > 0 && _iterators.Top()->IsDone() ) {
      delete _iterators.Pop();
      _iterators.Top()->Next();
   }
}
```

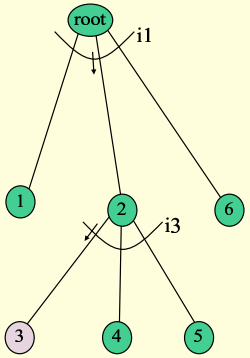

Pre-order Iterator




```
void PreorderIterator::Next () {
   Iterator<Glyph*>* i =
      _iterators.Top()->CurrentItem()->CreateIterator();
   i->First();
   _iterators.Push(i);
   while ( _iterators.Size() > 0 && _iterators.Top()->IsDone() ) {
      delete _iterators.Pop();
      _iterators.Top()->Next();
   }
}
```

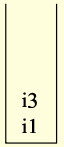
```
Glyph* PreorderIterator::CurrentItem () const {
   return _iterators.Size() > 0 ? _iterators.Top()->CurrentItem() : 0;
}
```

Pre-order Iterator

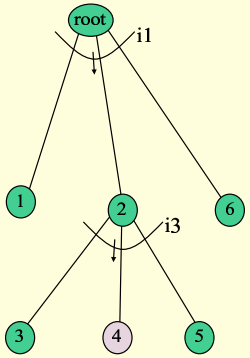



```
void PreorderIterator::Next () {
    Iterator<Glyph*>* i =
        _iterators.Top()->CurrentItem()->CreateIterator();
    i->First();
    _iterators.Push(i);
    while ( _iterators.Size() > 0 && _iterators.Top()->IsDone() ) {
        delete _iterators.Pop();
        _iterators.Top()->Next();
    }
}
```

```
Glyph* PreorderIterator::CurrentItem () const {
    return _iterators.Size() > 0 ? _iterators.Top()->CurrentItem() : 0;
}
```

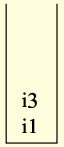


Pre-order Iterator

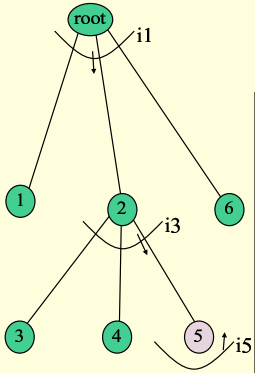



```
void PreorderIterator::Next () {
    Iterator<Glyph*>* i =
        _iterators.Top()->CurrentItem()->CreateIterator();
    i->First();
    _iterators.Push(i);
    while ( _iterators.Size() > 0 && _iterators.Top()->IsDone() ) {
        delete _iterators.Pop();
        _iterators.Top()->Next();
    }
}
```

```
Glyph* PreorderIterator::CurrentItem () const {
    return _iterators.Size() > 0 ? _iterators.Top()->CurrentItem() : 0;
}
```

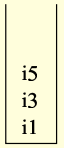


Pre-order Iterator

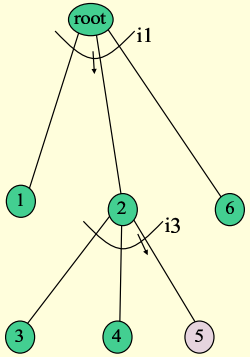



```
void PreorderIterator::Next () {
    Iterator<Glyph*>* i =
        _iterators.Top()->CurrentItem()->CreateIterator();
    i->First();
    _iterators.Push(i);
    while ( _iterators.Size() > 0 && _iterators.Top()->IsDone() ) {
        delete _iterators.Pop();
        _iterators.Top()->Next();
    }
}
```

```
Glyph* PreorderIterator::CurrentItem () const {
    return _iterators.Size() > 0 ? _iterators.Top()->CurrentItem() : 0;
}
```

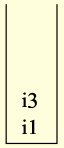


Pre-order Iterator

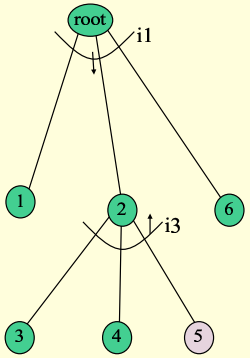



```
void PreorderIterator::Next () {
    Iterator<Glyph*>* i =
        _iterators.Top()->CurrentItem()->CreateIterator();
    i->First();
    _iterators.Push(i);
    while ( _iterators.Size() > 0 && _iterators.Top()->IsDone() ) {
        delete _iterators.Pop();
        _iterators.Top()->Next();
    }
}
```

```
Glyph* PreorderIterator::CurrentItem () const {
    return _iterators.Size() > 0 ? _iterators.Top()->CurrentItem() : 0;
}
```

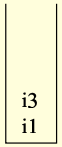


Pre-order Iterator

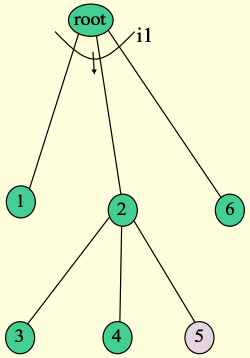



```
void PreorderIterator::Next () {
    Iterator<Glyph*>* i =
        _iterators.Top()->CurrentItem()->CreateIterator();
    i->First();
    _iterators.Push(i);
    while ( _iterators.Size() > 0 && _iterators.Top()->IsDone() ) {
        delete _iterators.Pop();
        _iterators.Top()->Next();
    }
}
```

```
Glyph* PreorderIterator::CurrentItem () const {
    return _iterators.Size() > 0 ? _iterators.Top()->CurrentItem() : 0;
}
```

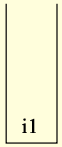


Pre-order Iterator

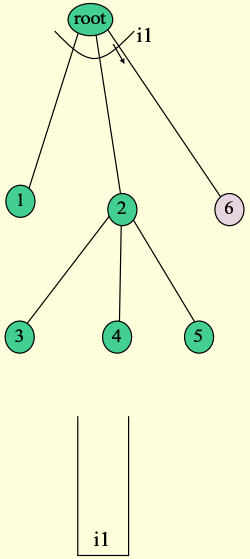



```
void PreorderIterator::Next () {
    Iterator<Glyph*>* i =
        _iterators.Top()->CurrentItem()->CreateIterator();
    i->First();
    _iterators.Push(i);
    while ( _iterators.Size() > 0 && _iterators.Top()->IsDone() ) {
        delete _iterators.Pop();
        _iterators.Top()->Next();
    }
}
```

```
Glyph* PreorderIterator::CurrentItem () const {
    return _iterators.Size() > 0 ? _iterators.Top()->CurrentItem() : 0;
}
```



Pre-order Iterator




```
void PreorderIterator::Next () {
    Iterator<Glyph*>* i =
        _iterators.Top()->CurrentItem()->CreateIterator();
    i->First();
    _iterators.Push(i);
    while ( _iterators.Size() > 0 && _iterators.Top()->IsDone() ) {
        delete _iterators.Pop();
        _iterators.Top()->Next();
    }
}
```

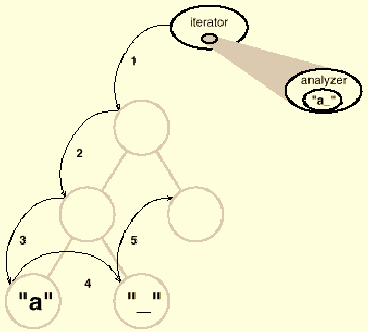
```
Glyph* PreorderIterator::CurrentItem () const {
    return _iterators.Size() > 0 ? _iterators.Top()->CurrentItem() : 0;
}
```


Traversal Actions

- Now that we can traverse, we need to add actions while traversing that have state
  - spelling, hyphenation, …
- Could augment the Iterator classes…
  - …but that would reduce their reusability
- Could augment the Glyph classes…
  - …but will need to change Glyph classes for each new analysis
- Will need to encapsulate the analysis in a separate object that will "visit" nodes in order established by iterator.

Actions in Iterators

- Iterator will carry the analysis object along with it as it iterates.
- The analyzer will accumulate state.
  - e.g., characters (and hence mispelled words) for a spell check



Avoiding Downcasts

- How can the analysis object distinguish different kinds of Glyphs without resorting to switch statements and downcasts?
  - e.g., avoid:

```
public class SpellingChecker extends … {
    public void check(Glyph g) {
        if( g instanceof CharacterGlyph ) {
            CharacterGlyph cg = (CharacterGlyph)g;
            // analyze the character
        } else if( g instanceof RowGlyph ) {
            rowGlyph rg = (RowGlyph)g;
            // prepare to analyze the child glyphs
        } else …
    }
}
```


Accepting Visitors

```
public abstract class Glyph {
    public abstract void accept(Visitor v);
    …
}

public class CharacterGlyph extends Glyph {
    public void accept(Visitor v) {
        v.visitCharacterGlyph(this); //override..
    }
    …
}
```


Visitor & Subclasses

```
public abstract class Visitor {
    public void visitCharacterGlyph(CharacterGlyph cg)
        { /* do nothing */ }
    public abstract void visitRowGlyph(RowGlyph rg);
        { /* do nothing */ }
    …
}
```

Visitor

```
public class SpellingVisitor extends Visitor {
    public void visitCharacterGlyph(CharacterGlyph cg)//override
   {
        …
    }
}
```

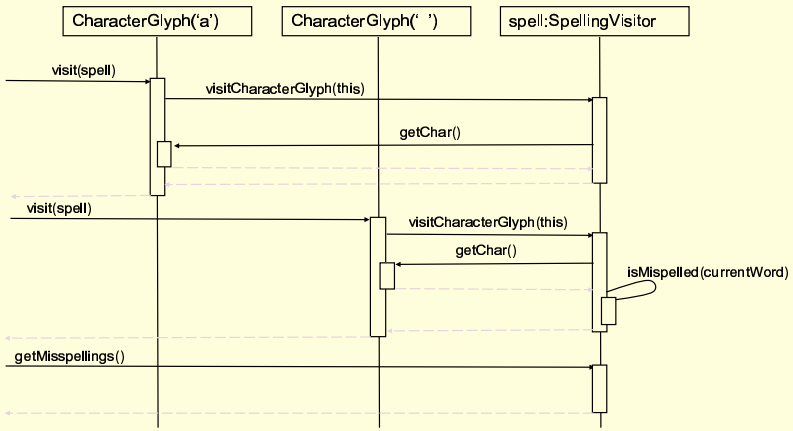
SpellingVisitor

```
public class SpellingVisitor extends Visitor {
    private Vector misspellings = new Vector();
    private String currentWord = "";

    public void visitCharacterGlyph(CharacterGlyph cg) {
        char c = cg->getChar();
        if( isalpha(c) ) {
            currentWord += c;
        } else {
            if( isMispelled(currentWord) ) {
                // add misspelling to list
                misspelling.addElement(currentWord);
            }
            currentWord = "";
        }
    }

    public Vector getMisspellings {
        return misspellings;
    }
    ...
}
```


Using SpellingVisitor

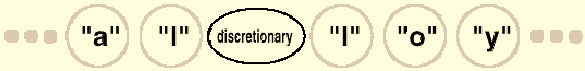
```
PreorderIterator i = new PreorderIterator();
i.setVisitor(new SpellingVisitor());
i.visitAll(rootGlyph);
Vector misspellings =
   ((SpellingVisitor)i.getVisistor()).getMisspellings()
   ;

public class Iterator {
    private Visitor v;
    public void visitAll(Glyph start) {
        for(first(); !isDone(); next()) {
            currentItem().visit(v);
        }
    }
    // . . .
}
```


Visitor Activity Diagram



HyphenationVisitor

- Visit words, and then insert "discretionary hyphen" Glyphs.



aluminum alloy *or* aluminum al–
loy

Summary

- In the design of LEXI, saw the following patterns.
  - Composite
    - represent physical structure
  - Strategy
    - to allow different formatting algorithms
  - Decorator
    - to embellish the UI
  - Abstract Factory
    - for supporting multiple L&F standards
  - Bridge
    - for supporting multiple windowing platforms
  - Command
    - for undoable operations
  - Iterator
    - for traversing object structures
  - Visitor
    - for allowing open-ended analytical capabilities without complicating the document structure