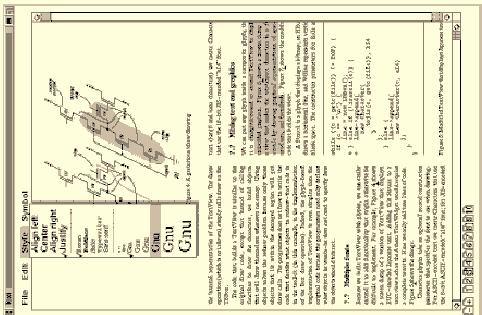


## Lexi Case Study

- 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



07,08 - LEXI CSC407

1

## Document Structure

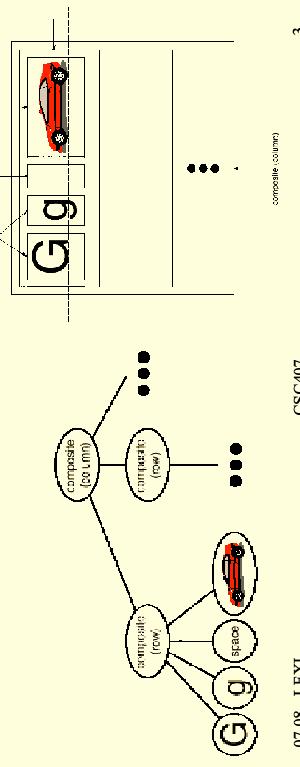
- A hierarchical arrangement of shapes.
- Viewed as lines, columns, figures, tables, ...
- UI should allow manipulations as a group
  - E.g. refer to a table as a whole
- Internal representation should support
  - Maintaining the physical structure
  - Generating and presenting the visuals
  - Reverse 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.

2

07,08 - LEXI CSC407

## 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.



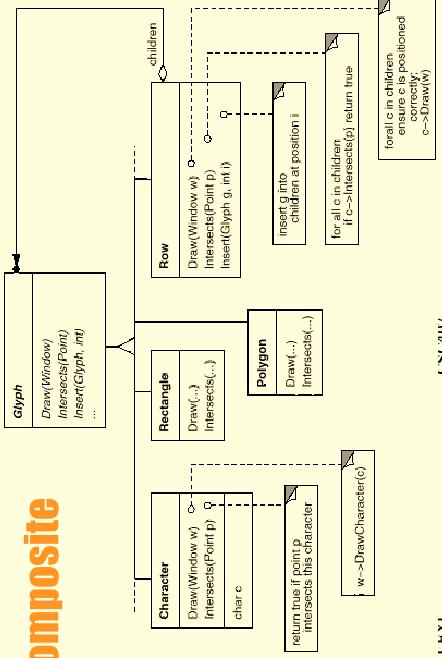
07,08 - LEXI CSC407

3

## Glyph Class

- An Abstract class for all objects that can appear in a document.
- Both primitive and composed.

## Composite

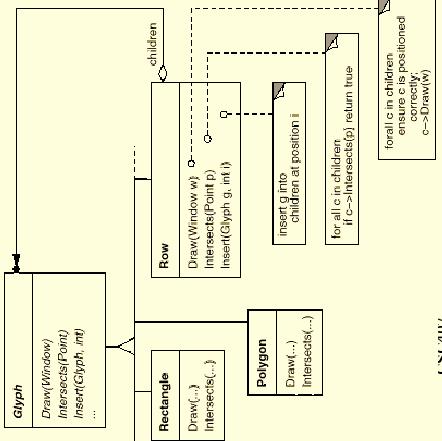


CSC407

4

- Internal representation 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.

- An Abstract class for all objects that can appear in a document.
- Both primitive and composed.



CSC407

2

## 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

07,08 - LEXI

CSC407 5

## 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();  
}
```

07,08 - LEXI CSC407 6

## 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 savvy realization that non-containers can often ignore component management messages.
- This can simplifies composition.

07,08 - LEXI CSC407 7

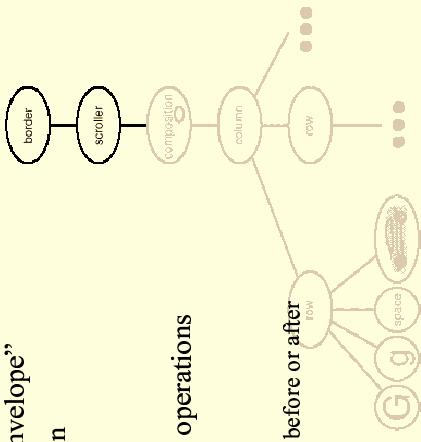
## Embellishments

- Wish to add visible borders and scroll-bars around pages.
- Inheritance is one way to do it.
  - leads to class proliferation
    - BorderedComposition, ScrollableComposition
    - 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?

07,08 - LEXI CSC407 8

## 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.



07,08 - LEXI CSC407 9

## IV monoglyph.h

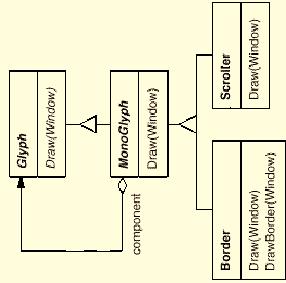
```
class MonoGlyph : public Glyph {
public:
    virtual ~MonoGlyph();
    virtual void body(Glyph* );
    virtual Glyph* body() const;
    void bodyclear();
    //remaining methods just like Glyph..
protected:
    MonoGlyph(Glyph* = nil);
    Glyph* body_;
private:
};
```

07,08 - LEXI CSC407 11

## MonoGlyph

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

## Decorator



10 CSC407

## 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);
    }
}
```

12 CSC407

## 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.

07,08 - LEXI 13 CSC407

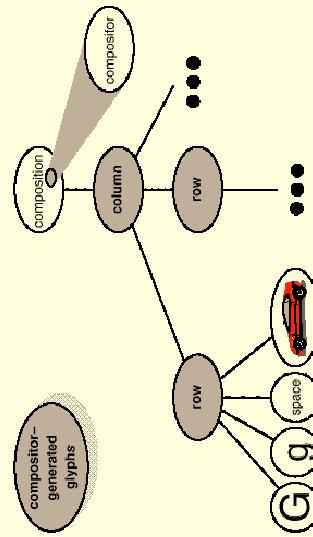
## Formatting

- 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.

14 CSC407

## Composition & Compositor

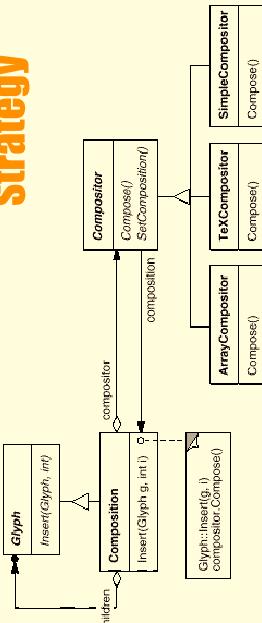
- Initially, an unformatted Composition object contains only the visible child Glyphs.
- After running a Compositor, it will also contain invisible, structural glyphs that define the format.



07,08 - LEXI 15 CSC407

## Strategy

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



16 CSC407

## 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.

File Edit Style Symbol Align Left Align Right Justify... Paragraph Font Size 10pt 12pt 14pt 16pt 18pt 20pt 22pt 24pt 26pt 28pt 30pt 32pt 34pt 36pt 38pt 40pt 42pt 44pt 46pt 48pt 50pt 52pt 54pt 56pt 58pt 60pt 62pt 64pt 66pt 68pt 70pt 72pt 74pt 76pt 78pt 80pt 82pt 84pt 86pt 88pt 90pt 92pt 94pt 96pt 98pt 100pt

**Gm**

**Gm**

Diagram illustrating the relationship between Gm and Gm'.

The diagram shows two shaded regions labeled "Gm" and "Gm'". The "Gm" region is bounded by the real axis and the imaginary axis, representing the right half-plane. The "Gm'" region is a larger shaded area that includes the "Gm" region and extends further into the left half-plane, representing the complex plane excluding the right half-plane.

Below the diagram, the text states: "So instead of working with the right half-plane, we can work with the left half-plane, which is easier to work with. In fact, it's easier to work with the left half-plane than the right half-plane because the right half-plane has poles, while the left half-plane does not have poles."

**2.2. Right half-plane poles**

The last point is that we can't work with the right half-plane because it has poles. Let's see what we mean by poles. First, let's recall that poles are points where the denominator of a rational function goes to zero. For example, in the function  $\frac{1}{s+1}$ , the pole is at  $s = -1$ . This means that if we substitute  $s = -1$  into the function, the denominator will be zero, and the function will be undefined. This is why we say that  $s = -1$  is a pole of the function.

Now, let's consider the right half-plane. The right half-plane is the set of all complex numbers  $s$  such that  $\operatorname{Re}(s) > 0$ . This means that the real part of  $s$  must be positive. If we substitute a complex number with a negative real part into the function, the denominator will not be zero, and the function will be defined. This is why we say that the right half-plane does not have poles.

So, instead of working with the right half-plane, we can work with the left half-plane, which is easier to work with. In fact, it's easier to work with the left half-plane than the right half-plane because the right half-plane has poles, while the left half-plane does not have poles.

**2.3. Poles**

Let's start by defining what we mean by a pole. We can define a pole as a point where the denominator of a rational function goes to zero. For example, in the function  $\frac{1}{s+1}$ , the pole is at  $s = -1$ . This means that if we substitute  $s = -1$  into the function, the denominator will be zero, and the function will be undefined. This is why we say that  $s = -1$  is a pole of the function.

Now, let's consider the right half-plane. The right half-plane is the set of all complex numbers  $s$  such that  $\operatorname{Re}(s) > 0$ . This means that the real part of  $s$  must be positive. If we substitute a complex number with a negative real part into the function, the denominator will not be zero, and the function will be defined. This is why we say that the right half-plane does not have poles.

So, instead of working with the right half-plane, we can work with the left half-plane, which is easier to work with. In fact, it's easier to work with the left half-plane than the right half-plane because the right half-plane has poles, while the left half-plane does not have poles.

**2.4. Stability**

Now that we know what poles are, we can talk about stability. We can define stability as the ability of a system to return to its equilibrium state after being disturbed. For example, consider a simple pendulum. If we pull the pendulum to one side and release it, it will swing back and forth until it comes to rest at its original position. This is a stable system because it returns to its equilibrium state after being disturbed.

Now, let's consider the right half-plane. The right half-plane is the set of all complex numbers  $s$  such that  $\operatorname{Re}(s) > 0$ . This means that the real part of  $s$  must be positive. If we substitute a complex number with a negative real part into the function, the denominator will not be zero, and the function will be defined. This is why we say that the right half-plane does not have poles.

So, instead of working with the right half-plane, we can work with the left half-plane, which is easier to work with. In fact, it's easier to work with the left half-plane than the right half-plane because the right half-plane has poles, while the left half-plane does not have poles.

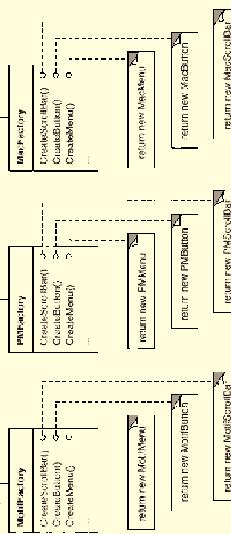
**2.5. Summary**

In this section, we learned about the right half-plane and the left half-plane. We saw that the right half-plane has poles, while the left half-plane does not have poles. This is why we can work with the left half-plane instead of the right half-plane. We also learned about poles and stability. We saw that a system is stable if it returns to its equilibrium state after being disturbed. We also saw that the right half-plane is the set of all complex numbers  $s$  such that  $\operatorname{Re}(s) > 0$ .

Z. 08 - LEXI CSC407 17

Object Factories

```
    }  
    ScrollBar sb = new MotifScrollBar();  
    sb.createMethod();
```



708 - ILEXI

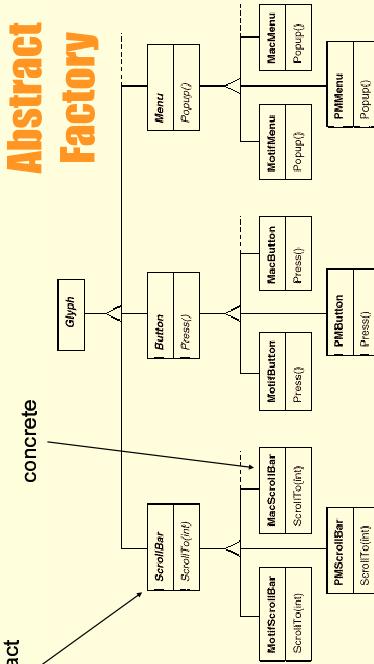
## Multiple Look-and-Feel Standards

- 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&t.
    - Win, Motif, OpenLook, Mac, ...

CSC407 18

Product Objects

- The output of a factory is a product.



19

## 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 I&F
    - Win, Motif, OpenLook, Mac, ...

CSC407 18

**Abstract  
Factory**

- Need indirect instantiation.

20 CSC407

## 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

07,08 - LEXI

21

## Singleton

CSC407

22

## 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)
 

```

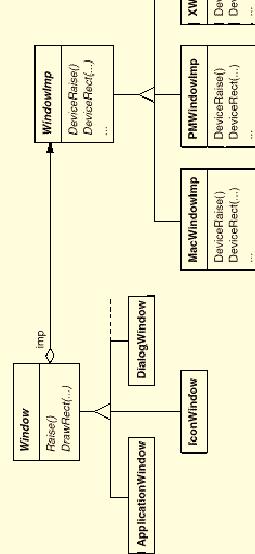
classDiagram
    class Window {
        <> drawWindow()
        <> iconify()
        <> lower()
        ...
    }
    class ApplicationWindow {
        <> drawWindow()
        <> iconify()
        <> lower()
        ...
    }
    class IconWindow {
        <> iconify()
        <> lower()
        ...
    }
    class DialogWindow {
        <> drawLine()
        <> drawRect()
        ...
    }
    class OwnerWindow {
        <> owner->lower()
        <> owner->raise()
        ...
    }
    ApplicationWindow --> Window : <> drawWindow()
    IconWindow --> Window : <> iconify()
    DialogWindow --> Window : <> drawLine()
    OwnerWindow --> Window : <> lower()
  
```

CSC407

22

## 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 ...



## 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) {
        imp.drawRect(x0,y0,x1,y1);
    }
    ...
}

public class XdrawRectangle(display, windowId, graphics, x,y,w,h);
}

```

07,08 - LEXI

23

24

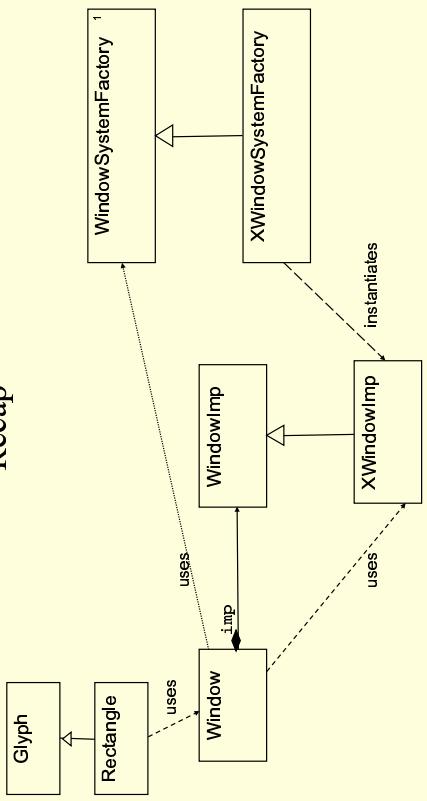
## Configuring 'imp'

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

07,08 - LEXI

25

## Recap



CSC407

26

## Abstract Factory

## User Operations

- Operations
  - create new, save, cut, paste, quit, ...
- UI mechanisms
  - mousing & typing in the document
  - pull-down menus, pop-up menus, buttons, lkd 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.

07,08 - LEXI

27

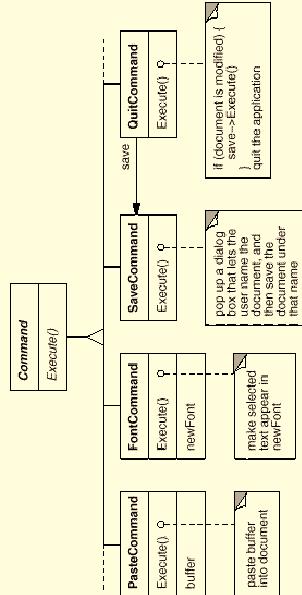
CSC407

28

- Command is an abstract class for issuing requests.

## Command Hierarchy

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



07,08 - LEXI CSC407 29

## Unidraw Color change command

```

class ColorCmd : public Command {
public:
    ColorCmd(ControlInfo*, PSColor*, 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);

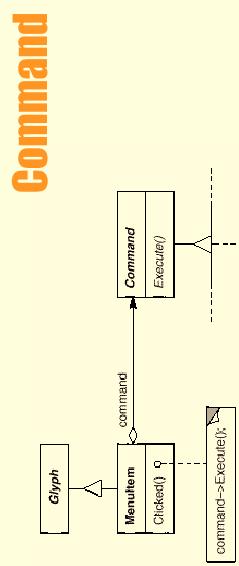
protected:
    PSColor* _fg, * _bg;
};

07,08 - LEXI CSC407 31

```

## Invoking Commands

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



07,08 - LEXI CSC407 30

## Unidraw ColorCmd implementation

```

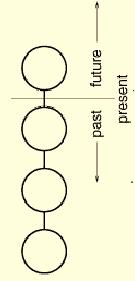
void ColorCmd::Execute () {
    ColorVar* colorVar = //current colour
    if (colorVar != nil) {
        PSColor* fg = (_fg == nil) ?
            colorVar->GetFgColor() : _fg;
        PSColor* bg = (_bg == nil) ?
            colorVar->GetBgColor() : _bg;
        colorVar->SetColors(fg, bg);
    }
  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

07,08 - LEXI CSC407 32

```

## 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 `undoEffect()` method
- Maintain Command history:



07,08 - LEXI CSC407 33

## 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.

07,08 - LEXI CSC407 34

## 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

07,08 - LEXI CSC407 35

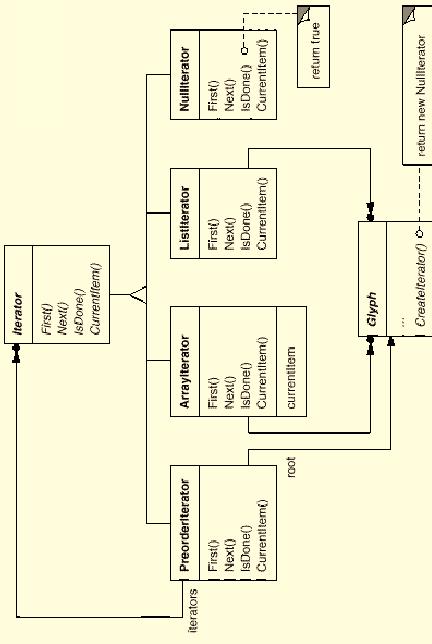
## Encapsulating Access & Traversals

- 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).

07,08 - LEXI CSC407 36

07,08 - LEXI CSC407 36

## Iterator Hierarchy



07,08 - LEXI

CSC407

37

## 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.

38

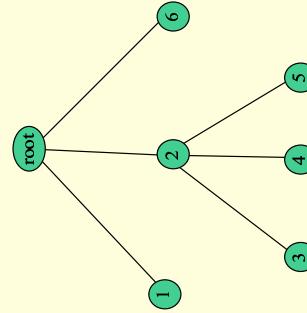
CSC407

## Initializing Iterators

```

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

## Pre-order Iterator



07,08 - LEXI

CSC407

39

CSC407

40

## 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.

```
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;
}
```

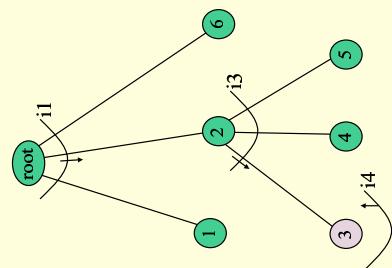
07,08 - LEXI CSC407 41

## Implementing a Complex Iterator (cont'd)

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

07,08 - LEXI CSC407 42

## Pre-order Iterator



i4  
i3  
i1

07,08 - LEXI CSC407 43

```
void PreorderIterator::Next () {
    Iterator<Glyph*>* i =
    _iterators.Top()->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;
}
```

07,08 - LEXI CSC407 44

### Pre-order Iterator

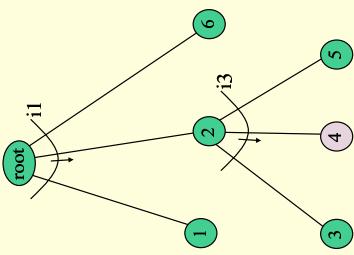
```
void PreorderIterator::Next () {
    Iterator<Glyph*> i =
        _iterators.Top()>CurrentItem()>CreateIterator();
    i->First();
    _iterators.Push(i);
    while (_iterators.Size() > 0 && _iterators.Top()>IsDone() ) {
        delete _iterators.Pop();
        _iterators.Top()>Next();
    }
}
```

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

07,08 - LEXI

CSC407

45



### 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;
}
```

07,08 - LEXI

CSC407

46

### Pre-order Iterator

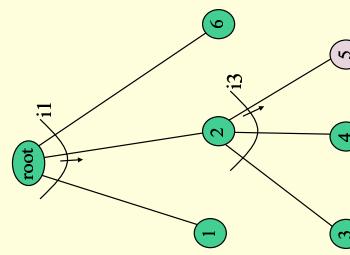
```
void PreorderIterator::Next () {
    Iterator<Glyph*> i =
        _iterators.Top()>CurrentItem()>CreateIterator();
    i->First();
    _iterators.Push(i);
    while (_iterators.Size() > 0 && _iterators.Top()>IsDone() ) {
        delete _iterators.Pop();
        _iterators.Top()>Next();
    }
}
```

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

07,08 - LEXI

CSC407

47



i3  
i1

### Pre-order Iterator

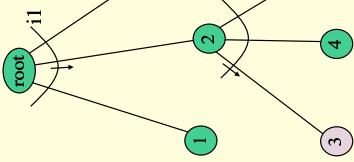
```
void PreorderIterator::Next () {
    Iterator<Glyph*> i =
        _iterators.Top()>CurrentItem()>CreateIterator();
    i->First();
    _iterators.Push(i);
    while (_iterators.Size() > 0 && _iterators.Top()>IsDone() ) {
        delete _iterators.Pop();
        _iterators.Top()>Next();
    }
}
```

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

07,08 - LEXI

CSC407

48



## Pre-order Iterator



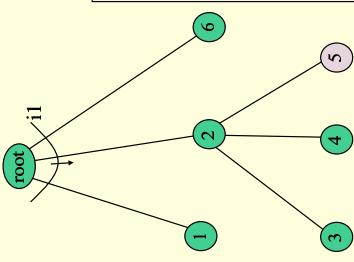
## Pre-order Iterator

```
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;
}
```

07,08 - LEXI CSC407 49

## Pre-order Iterator



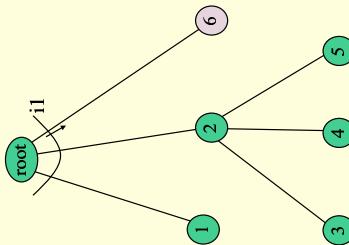
## Pre-order Iterator

```
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;
}
```

07,08 - LEXI CSC407 50

## Pre-order Iterator



## Traversals 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.

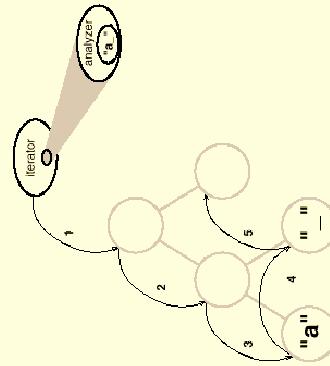
07,08 - LEXI CSC407 51

51

52

## 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 misspelled words) for a spell check



07,08 - LEXI

53

CSC407

54

## 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 ...  
    }  
}
```

CSC407

54

07,08 - LEXI

## 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...  
    }  
    ...  
}
```

07,08 - LEXI

55

## Visitor & Subclasses

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

CSC407

56

## Visitor

## 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( !isMisspelled(currentWord) ) {
                // add misspelling to list
                misspelling.addElement(currentWord);
            }
            currentWord = "";
        }
    }
    public Vector getMisspellings() {
        return misspellings;
    }
}
```

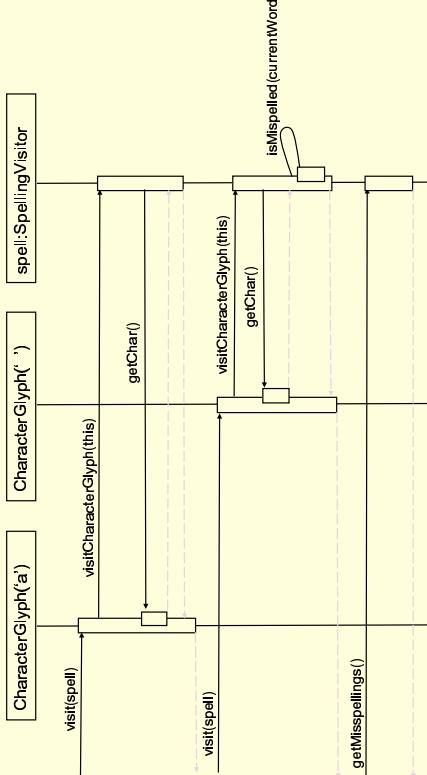
07,08 - LEXI CSC407 57

## Using SpellingVisitor

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

public class Iterator {
    private Visitor v;
    public void visitAll(Glyph start) {
        for(first(); !isDone(); next()) {
            currentItem().visit(v);
        }
    }
    // . . .
}
07,08 - LEXI CSC407 58
```

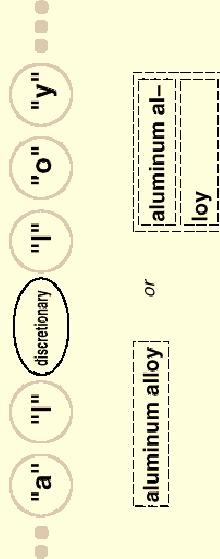
## Visitor Activity Diagram



07,08 - LEXI CSC407 59

## HyphenationVisitor

- Visit words, and then insert “discretionary hyphen” Glyphs.



```
aluminum al-
loy
or
[aluminum alloy]
07,08 - LEXI CSC407 60
```

## 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