DESIGN PATTERNS

DR. S. PRABAHARAN

ASSOCIATE PROFESSOR

DEPT. OF CSE

JYOTHISHMATHI INSTITUTE OF TECHNOLOGY AND SCIENCE

A Case Study:

Designing a Document Editor:

Design Problems:

Seven problems in Lexis's design:

Document Structure:

- ✓ The choice of internal representation for the document affects nearly every aspect of Lexis's design. All editing , formatting, displaying, and textual analysis will require traversing the representation.

Formatting:

- ✓ How does Lexi actually arrange text and graphics into lines and columns?
- ✓ What objects are responsible for carrying out different formatting policies?
- ✓ How do these policies interact with the document's internal representation?

Embellishing the user interface:

- ✓ Lexis user interface include scroll bar, borders and drop shadows that embellish the WYSIWYG document interface. Such embellishments are likely to change as Lexis user interface evolves.

Design Problems

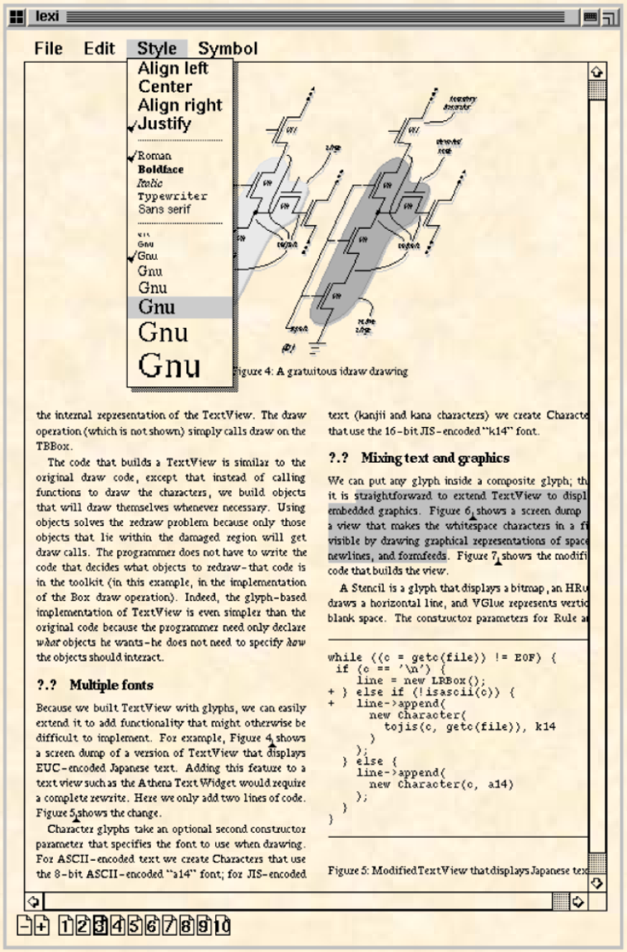
Supporting multiple look-and-feel standards:
✓Lexi should adapt easily to different look-and-feel standards such as Motif and Presentation Manager (PM) without major modification.

Supporting multiple window systems:
✓Different look-and-fell standards are usually implemented on different window system. Lexi's design should be independent of the window system as possible.

User Operations:
✓User control Lexi through various interfaces, including buttons and pull-down menus. The functionality beyond these interfaces is scattered throughout the objects in the application.

Spelling checking and hyphenation.:
✓ How does Lexi support analytical operations checking for misspelled words and determining hyphenation points? How can we minimize the number of classes we have to modify to add a new analytical operation?

Application: Document Editor (Lexi)



7 Design Problems

1. Document structure
2. Formatting
3. Embellishment
4. Multiple look & feels
5. Multiple window systems
6. User operations
7. Spelling checking & hyphenation

Document Structure

Goals:

- present document’s visual aspects
- drawing, hit detection, alignment
- support physical structure (e.g., lines, columns)

Constraints/forces:

- treat text & graphics uniformly
- no distinction between one & many

Document Structure

- The internal representation for a document
- The internal representation should support
  - maintaining the document's physical structure
  - generating and presenting the document visually
  - mapping positions on the display to elements in the internal representations

Document Structure (cont.)

- Some constraints
  - we should treat text and graphics uniformly
  - our implementation shouldn’t have to distinguish between single elements and groups of elements in the internal representation
- Recursive Composition
  - a common way to represent hierarchically structured information

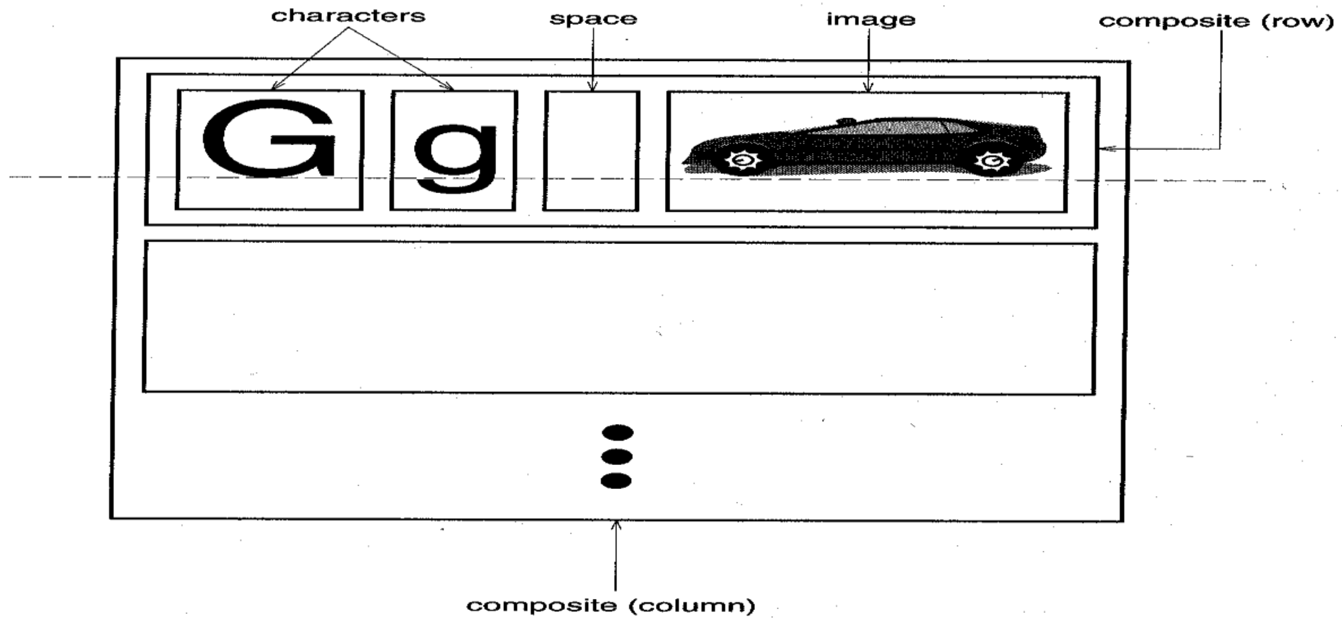


Figure 2.2: Recursive composition of text and graphics

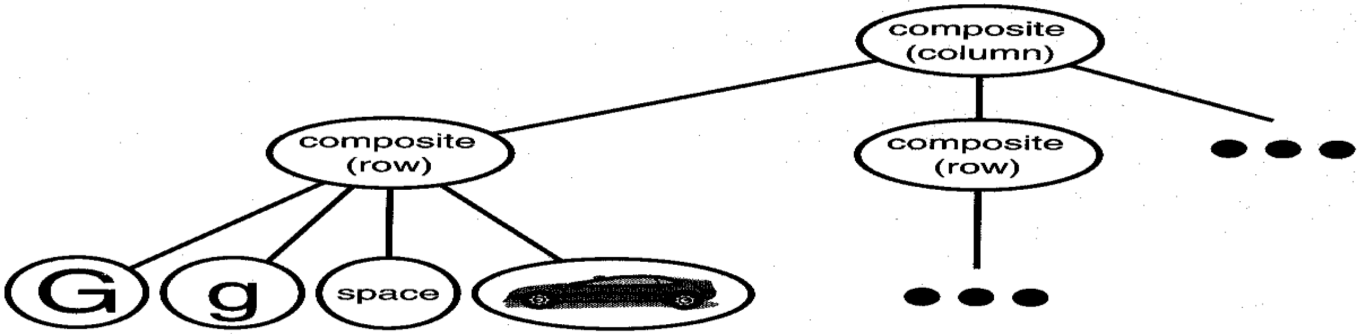


Figure 2.3: Object structure for recursive composition of text and graphics

Document Structure (cont.)

- Glyphs
  - an abstract class for all objects that can appear in a document structure
  - three basic responsibilities, they know
    - how to draw themselves, what space they occupy, and their children and parent
- Composite Pattern
  - captures the essence of recursive composition in object-oriented terms

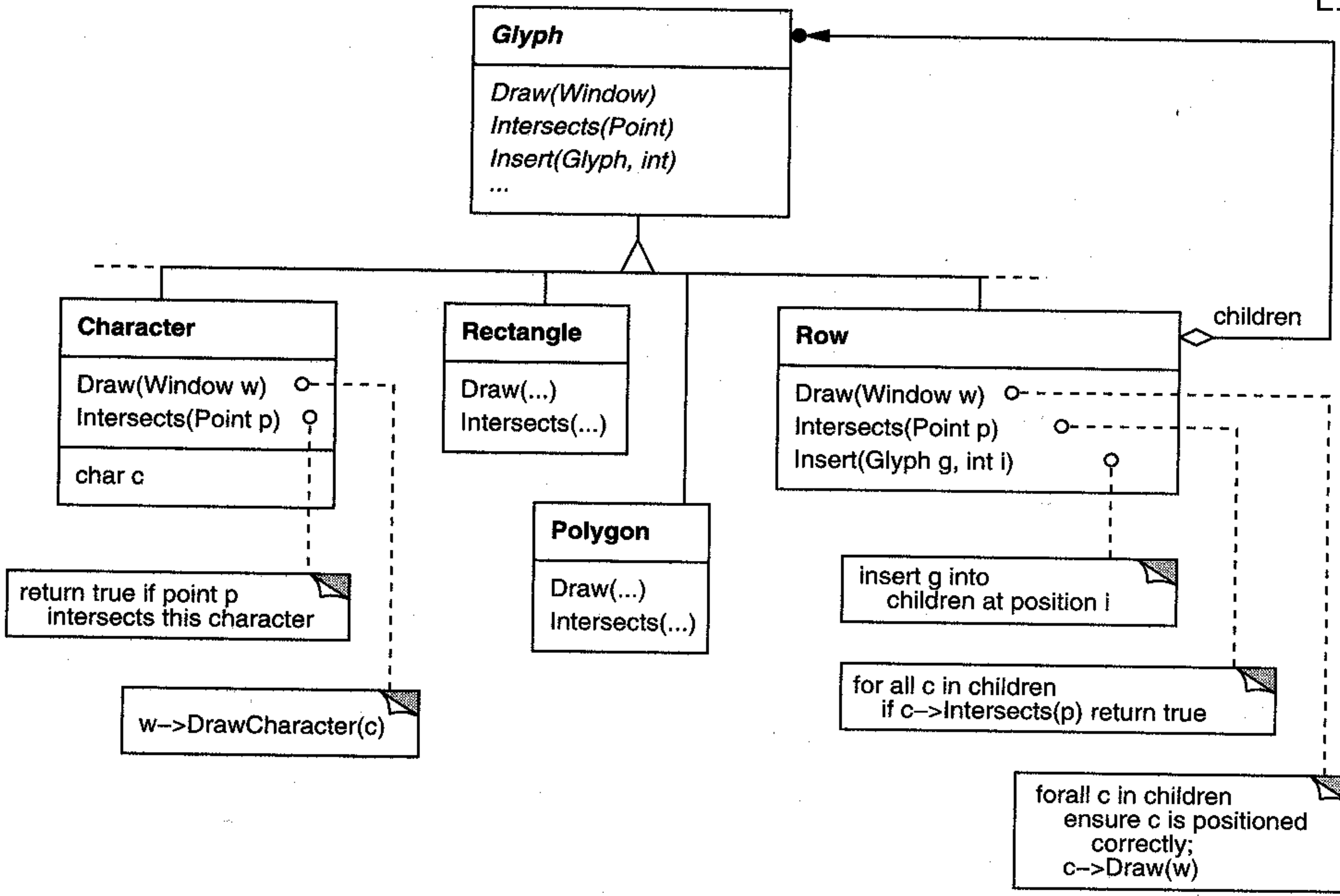


Figure 2.4: Partial Glyph class hierarchy

Responsibility	Operations
appearance	`virtual void Draw(Window*)` `virtual void Bounds(Rect&)`
hit detection	`virtual bool Intersects(const Point&)`

structure	`virtual void Insert(Glyph*, int)`
	`virtual void Remove(Glyph*)`
	`virtual Glyph* Child(int)`
	`virtual Glyph* Parent()`

Table 2.1: Basic glyph interface

Formatting

- A structure that corresponds to a properly formatted document
- Representation and formatting are distinct
  - the ability to capture the document’s physical structure doesn’t tell us how to arrive at a particular structure
- here, we’ll restrict “formatting” to mean breaking a collection of glyphs in to lines

Formatting (cont.)

- Encapsulating the formatting algorithm
  - keep formatting algorithms completely independent of the document structure
  - make it is easy to change the formatting algorithm
  - We'll define a separate class hierarchy for objects that encapsulate formatting algorithms

Formatting (cont.)

- Compositor and Composition
  - We'll define a ***Compositor*** class for objects that can encapsulate a formatting algorithm
  - The glyphs Compositor formats are the children of a special Glyph subclass called ***Composition***
  - When the composition needs formatting, it calls its compositor's *Compose* operation
  - Each Compositor subclass can implement a different line breaking algorithm

Responsibility	Operations
what to format	`void SetComposition(Composition*)`
when to format	`virtual void Compose()`

Table 2.2: Basic compositor interface

Formatting (cont.)

- Compositor and Composition (cont.)
  - The Compositor-Composition class split ensures a strong *separation* between code that supports the document's physical structure and the code for different formatting algorithms
- Strategy pattern
  - intent: encapsulating an algorithm in an object
  - Compositors are strategies. A composition is the context for a compositor strategy

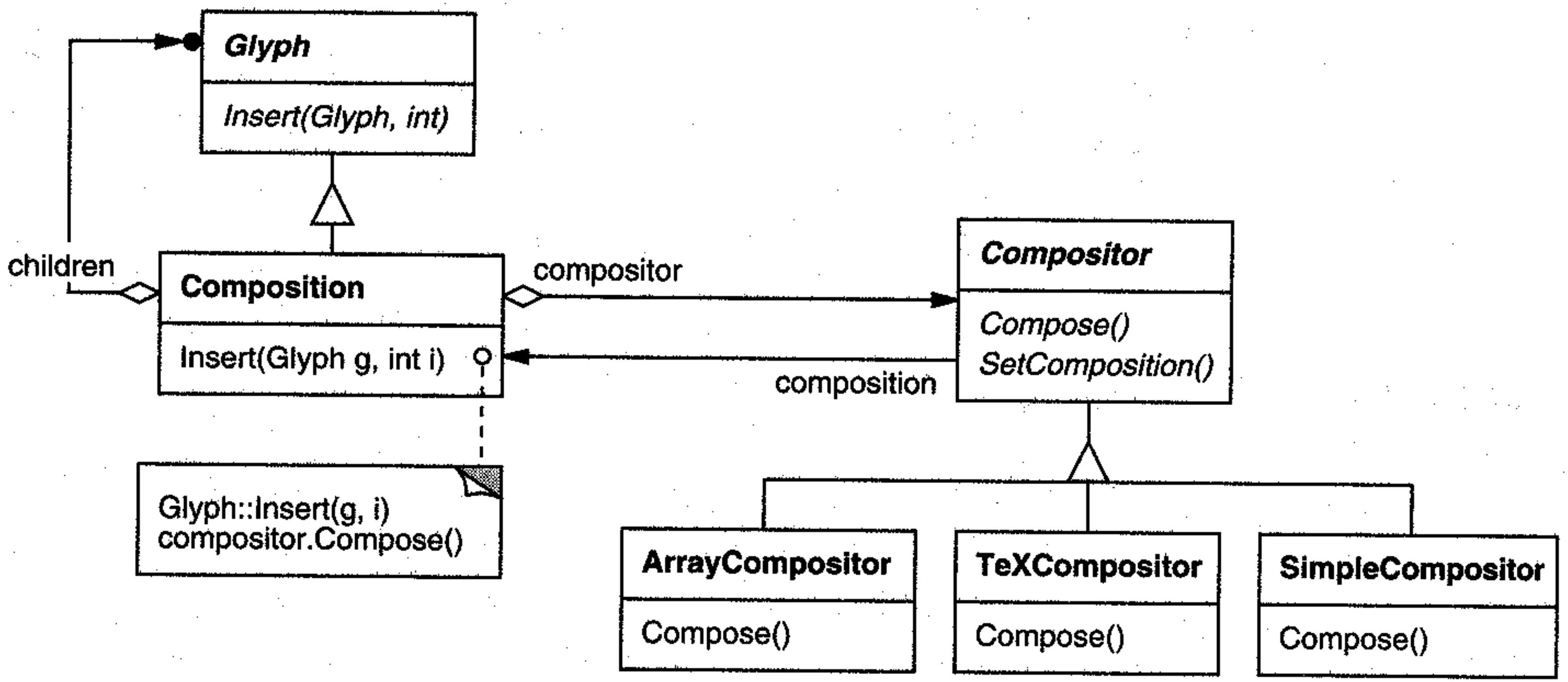


Figure 2.5: Composition and Compositor class relationships

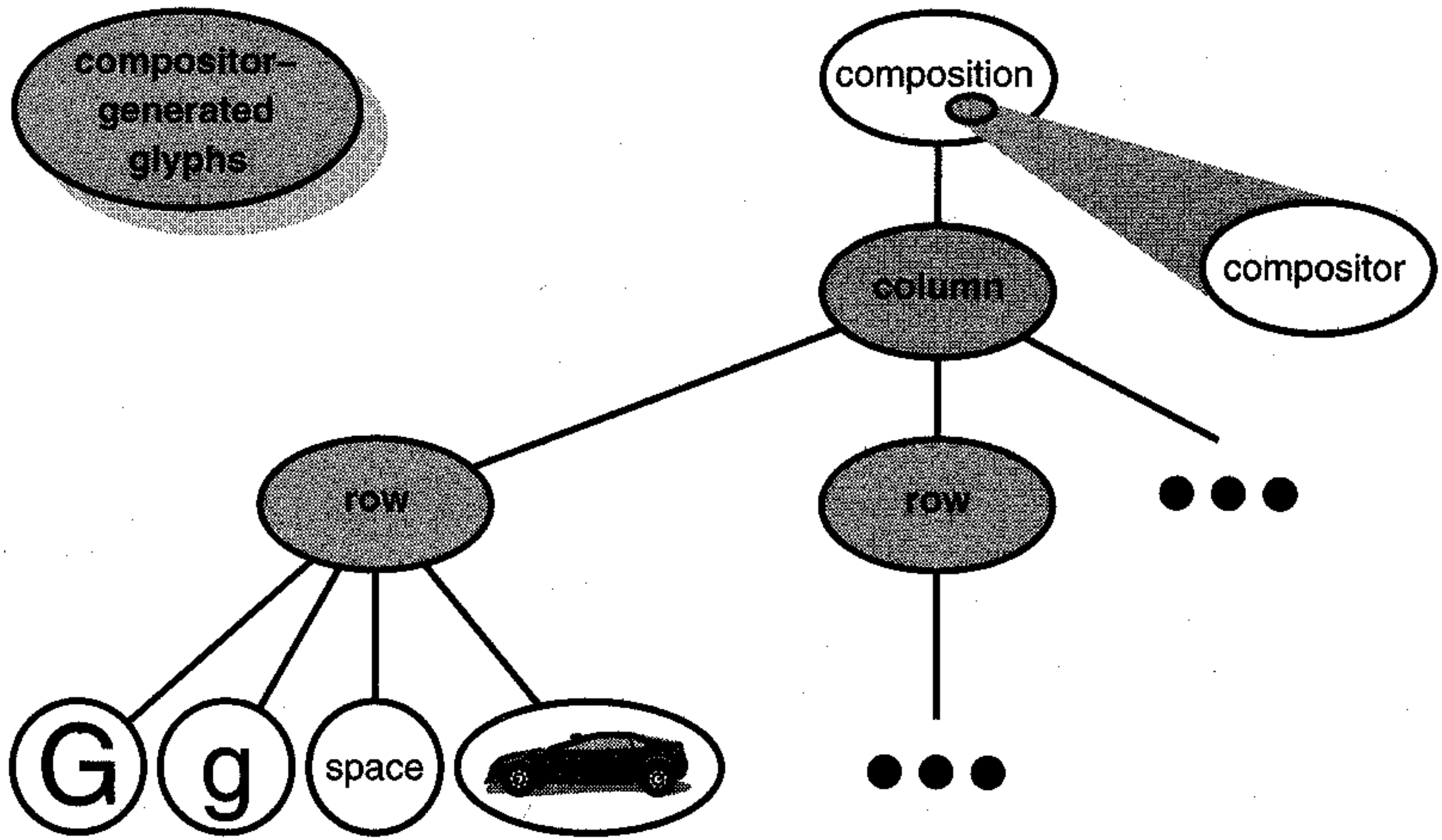


Figure 2.6: Object structure reflecting compositor-directed linebreaking

Embellishing the User Interface

- Considering adds a ***border*** around the text editing area and ***scrollbars*** that let the user view the different parts of the page here
- Transparent Enclosure
  - *inheritance-based* approach will result in some problems
    - Composition, ScollableComposition, BorderedScrollableComposition, …
  - *object composition* offers a potentially more workable and flexible extension mechanism

Embellishing the User Interface (cont.)

- Transparent enclosure (cont.)
  - object composition (cont.)
    - Border and Scroller should be a subclass of Glyph
  - two notions
    - single-child (single-component) composition
    - compatible interfaces

Embellishing the User Interface (cont.)

- Monoglyph
  - We can apply the concept of transparent enclosure to all glyphs that embellish other glyphs
  - the class, Monoglyph
- Decorator Pattern
  - captures class and object relationships that support embellishment by transparent enclosure

```
void MonoGlyph::Draw(Window* w) {
        _component-> Draw(w);
}
void Border:: Draw(Window * w) {
        MonoGlyph::Draw(w);
        DrawBorder(w);
}
```

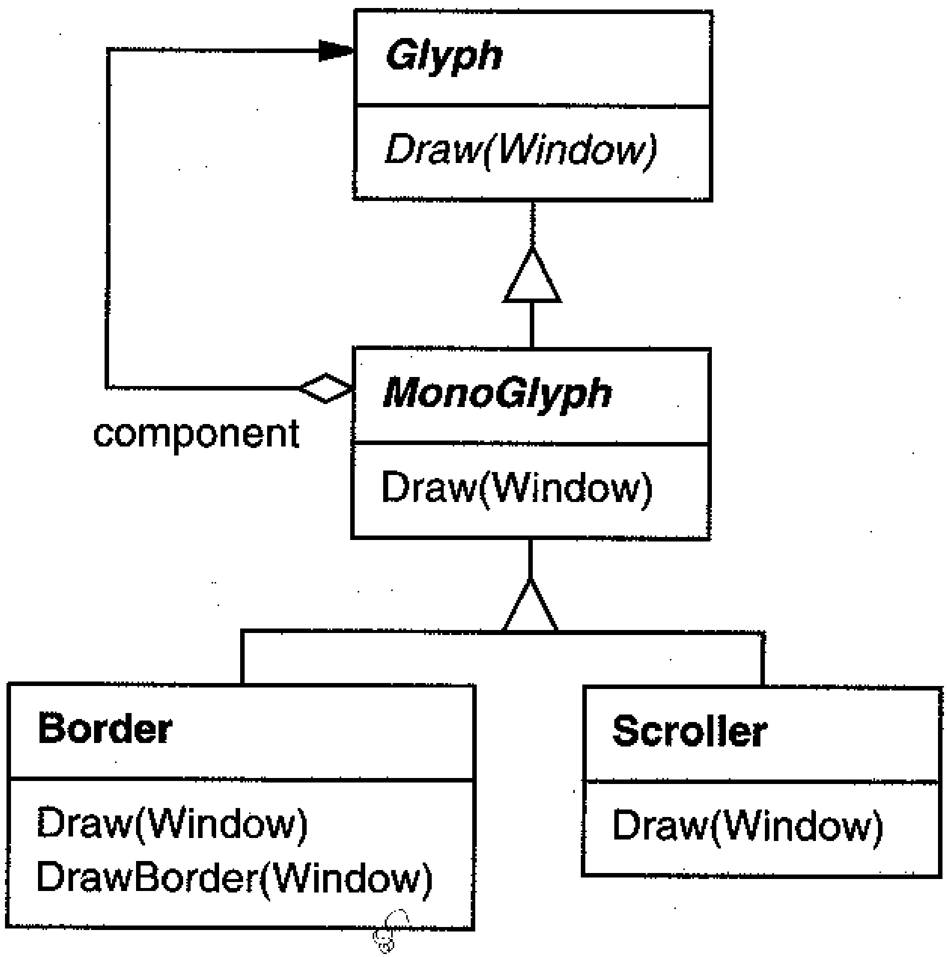


Figure 2.7: MonoGlyph class relationships

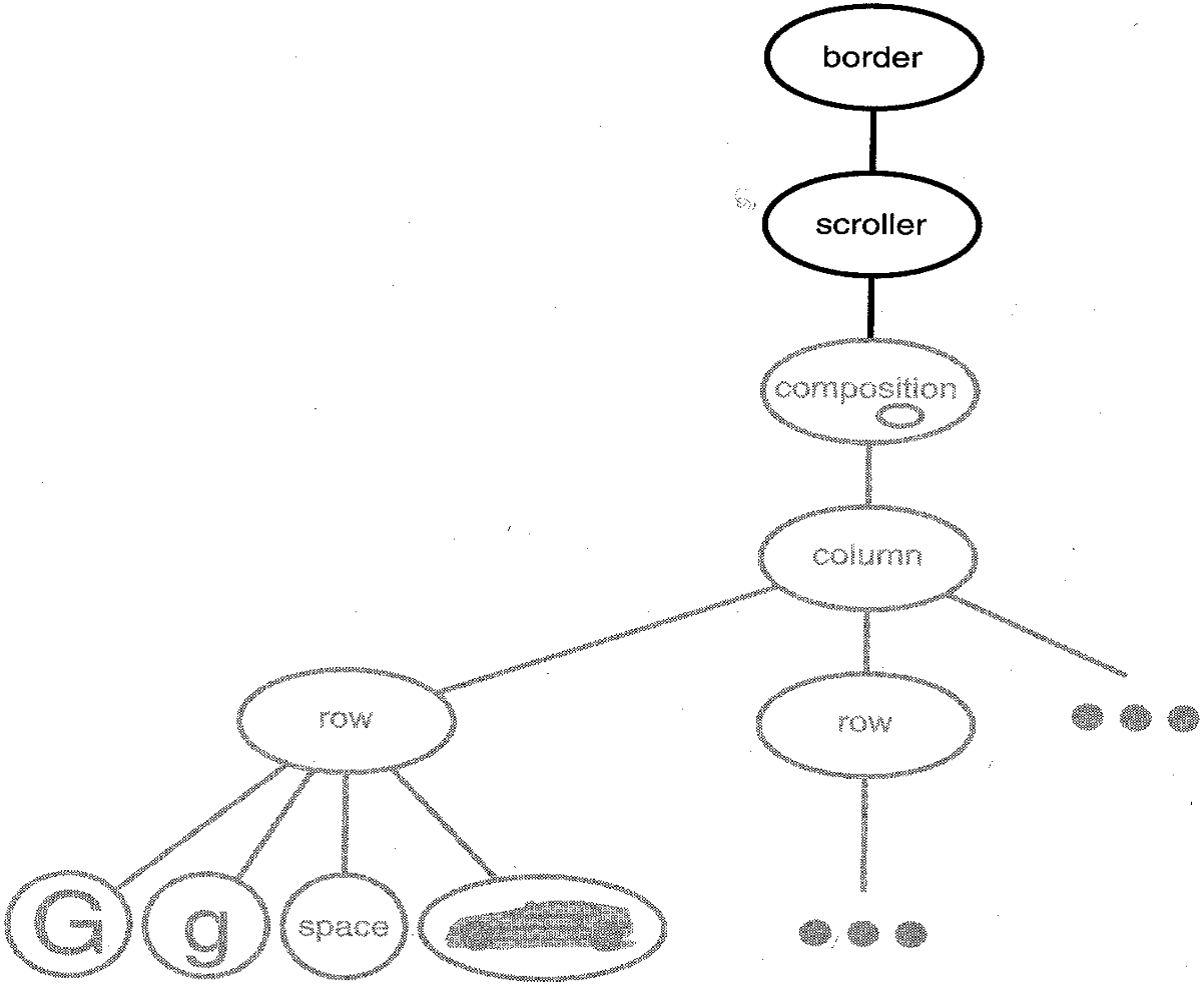


Figure 2.8: Embellished object structure

Supporting Multiple Look-and-Feel Standards

- Design to support the look-and-feel changing at run-time
- Abstracting Object Creation
  - widgets
  - two sets of widget glyph classes for this purpose
    - a set of abstract glyph subclasses for each category of widget glyph (e.g., ScrollBar)
    - a set of concrete subclasses for each abstract subclass that implement different look-and-feel standards (e.g., MotifScrollBar and PMScrollBar)

Supporting Multiple Look-and-Feel Standards (cont.)

- Abstracting Object Creation (cont.)
  - Lexi needs a way to determine the look-and-feel standard being targeted
  - We must avoid making explicit constructor calls
  - We must also be able to replace an entire widget set easily
  - We can achieve both by *abstracting the process of object creation*

Supporting Multiple Look-and-Feel Standards (cont.)

- Factories and Product Classes
  - ***Factories*** create ***product*** objects
  - The example
- Abstract Factory Pattern
  - capture how to create families of related product objects without instantiating classes *directly*

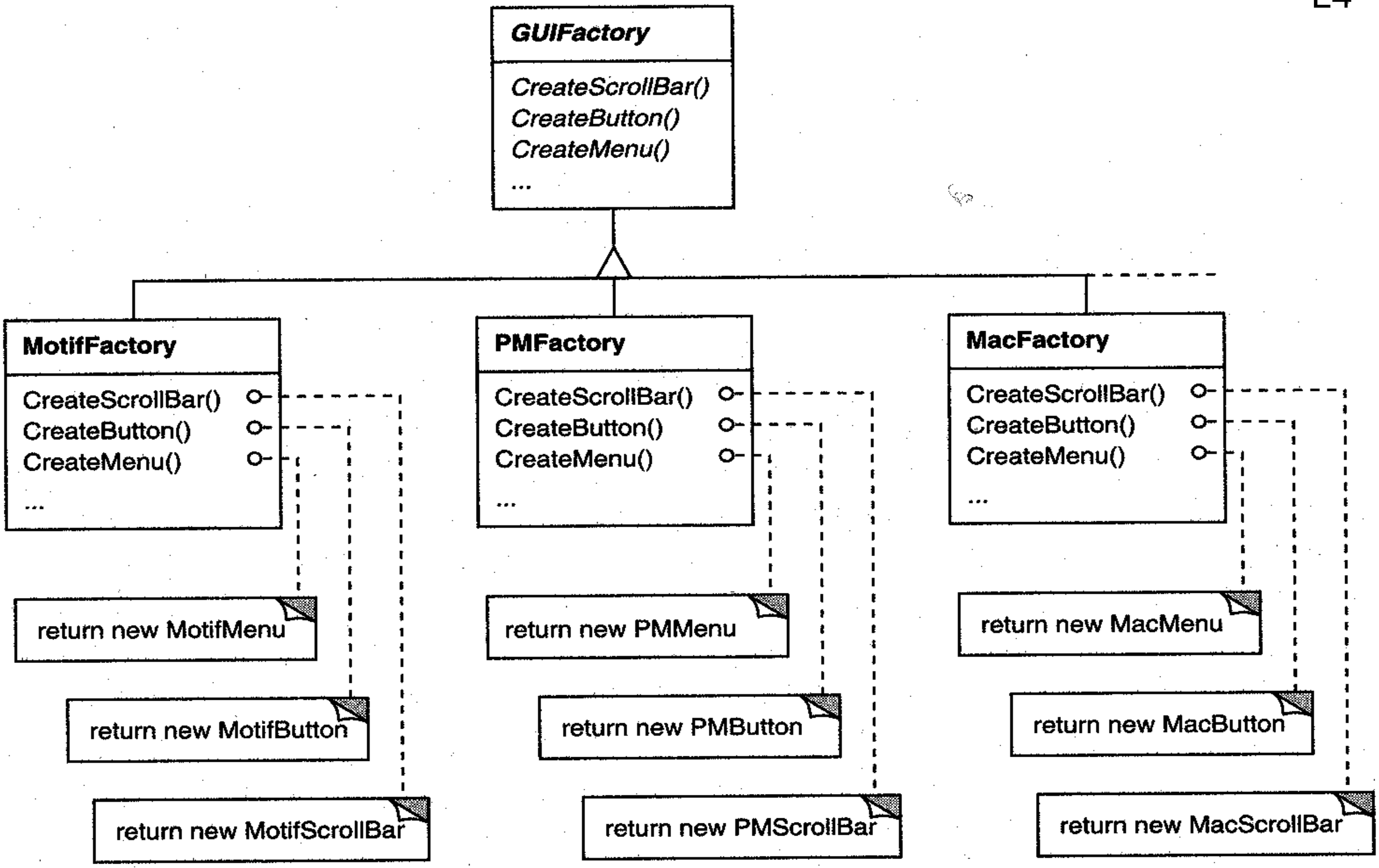


Figure 2.9: GUIFactory class hierarchy

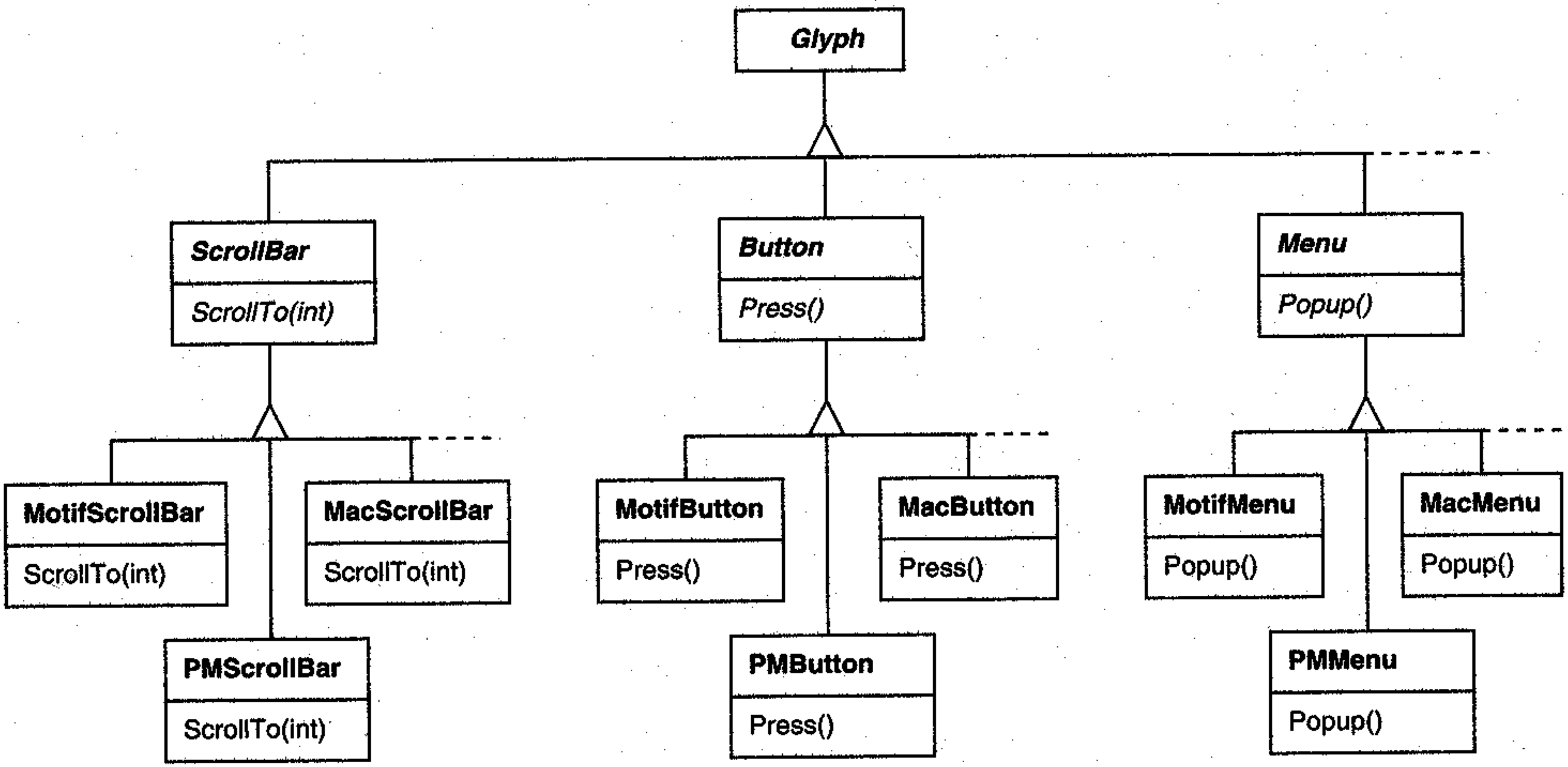


Figure 2.10: Abstract product classes and concrete subclasses

Supporting Multiple Window Systems

- We’d like Lexi to run on many existing window systems having different programming interfaces
- Can we use an Abstract Factory?
  - As the different programming interfaces on these existing window systems, the Abstract Factory pattern doesn‘t work
  - We need a uniform set of windowing abstractions that lets us take different window system impelementations and slide any one of them under a common interface

Supporting Multiple Window Systems (cont.)

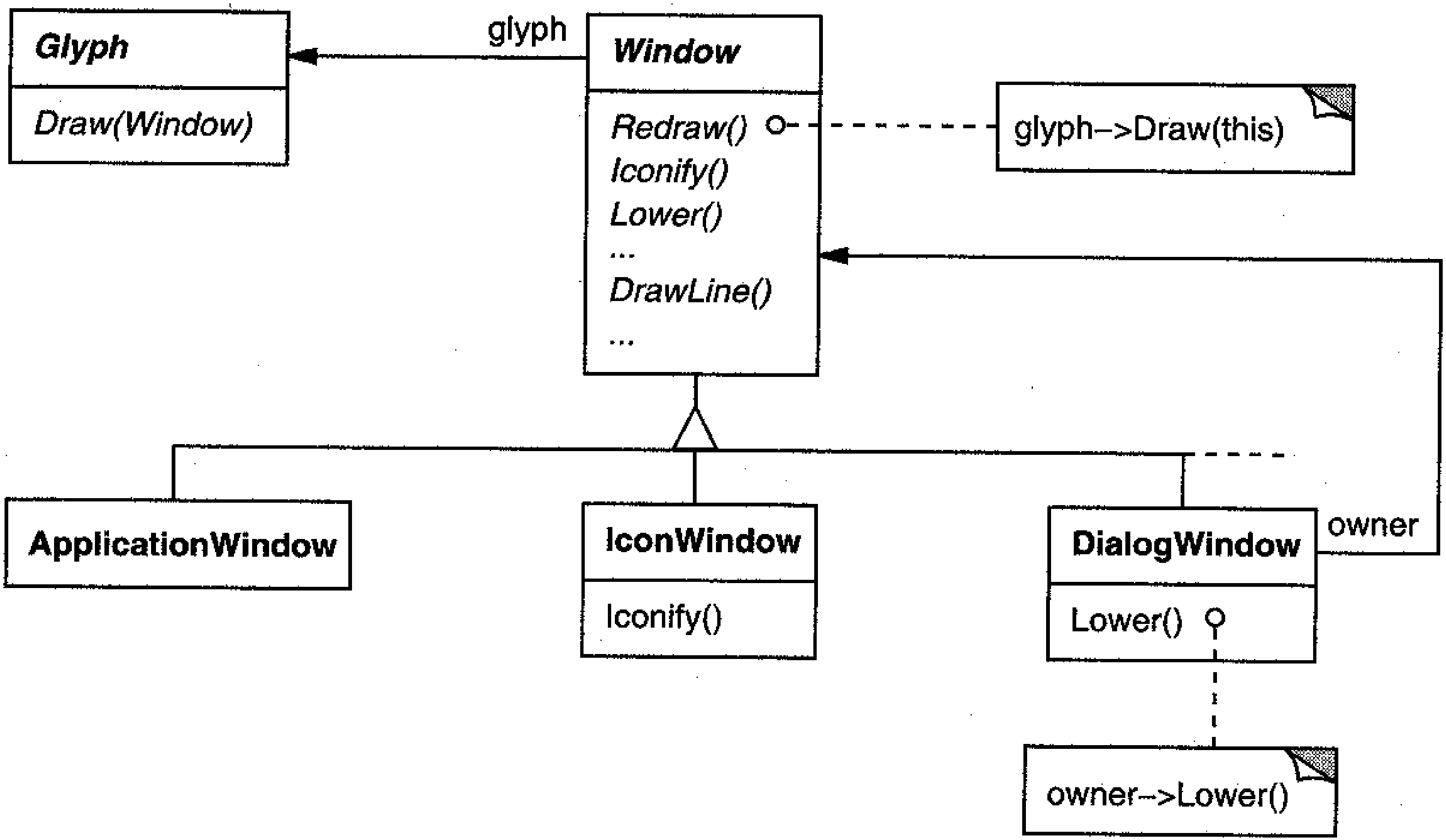
- Encapsulating Implementation Dependencies
  - The Window class interface encapsulates the things windows tend to do across window systems
  - The Window class is an abstract class
  - Where does the implementation live?
- Window and WindowImp
- Bridge Pattern
  - to allow separate class hierarchies to work together even as they evolve independently

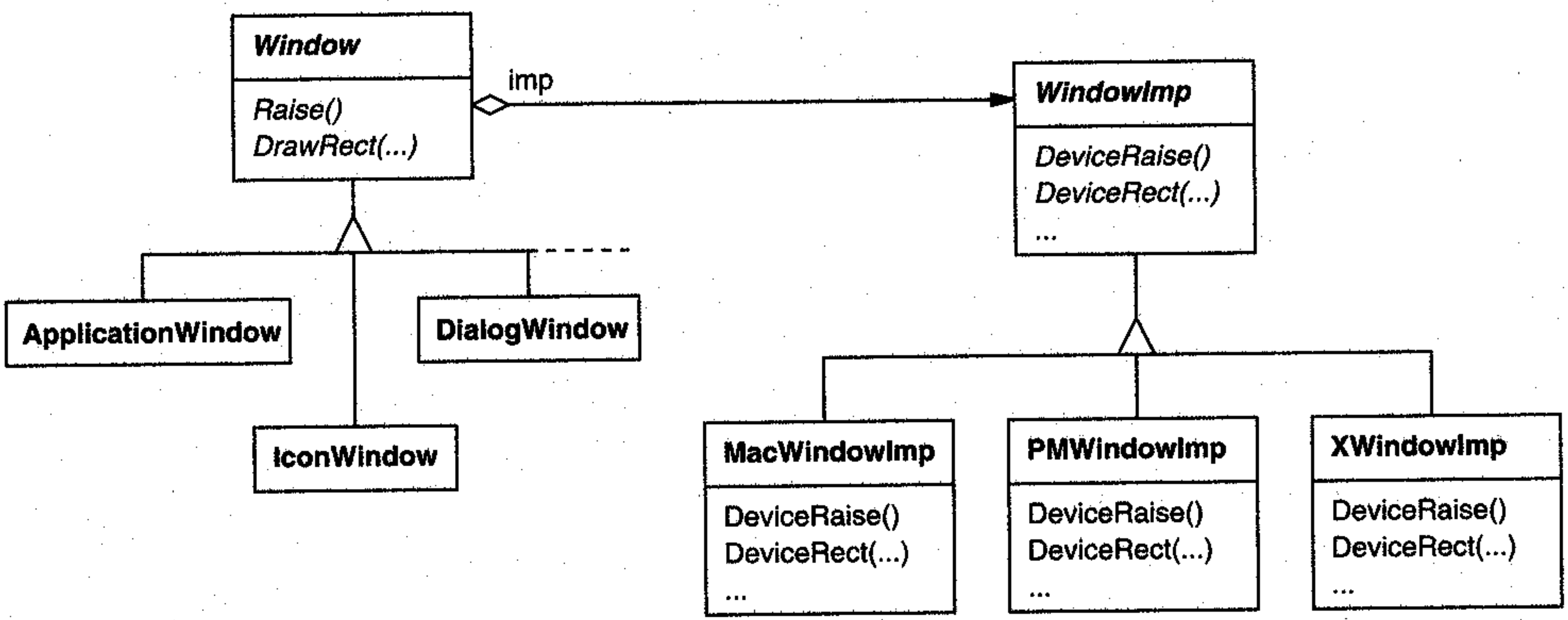
Responsibility	Operations

window management	`virtual void Redraw()`
	`virtual void Raise()`
	`virtual void Lower()`
	`virtual void Iconify()`
	`virtual void Deiconify()`
	`...`

graphics	`virtual void DrawLine(...)`
	`virtual void DrawRect(...)`
	`virtual void DrawPolygon(...)`
	`virtual void DrawText(...)`
	`...`

Table 2.3: Window class interface


Glyph
Draw(Window)
glyph
Window
Redraw()
Iconify()
Lower()
...
DrawLine()
...
glyph->Draw(this)
ApplicationWindow
IconWindow
Iconify()
DialogWindow
Lower()
owner
owner->Lower()


Window
Raise()
DrawRect(...)
imp
WindowImp
DeviceRaise()
DeviceRect(...)
...
ApplicationWindow
DialogWindow
IconWindow
MacWindowImp
DeviceRaise()
DeviceRect(...)
...
PMWindowImp
DeviceRaise()
DeviceRect(...)
...
XWindowImp
DeviceRaise()
DeviceRect(...)
...

User Operations

- Requirements
  - Lexi provides different user interfaces for the operations it supported
  - These operations are implemented in many different classes
  - Lexi supports undo and redo
- The challenge is to come up with a simple and extensible mechanism that satisfies all of these needs

User Operations (cont.)

- Encapsulating a Request
  - We could parameterize MenuItem with a *function* to call, but that’s not a complete solution
    - it doesn’t address the undo/redo problem
    - it’s hard to associate state with a function
    - functions are hard to extent, and it’s hard to reuse part of them
  - We should parameterize MenuItems with an ***object***, not a function

User Operations (cont.)

- Command Class and Subclasses
  - The Command abstract class consists of a single abstract operation called “Execute”
  - MenuItem can store a Command object that encapsulates a request
  - When a user choose a particular menu item, the MenuItem simply calls Execute on its Command object to carry out the request

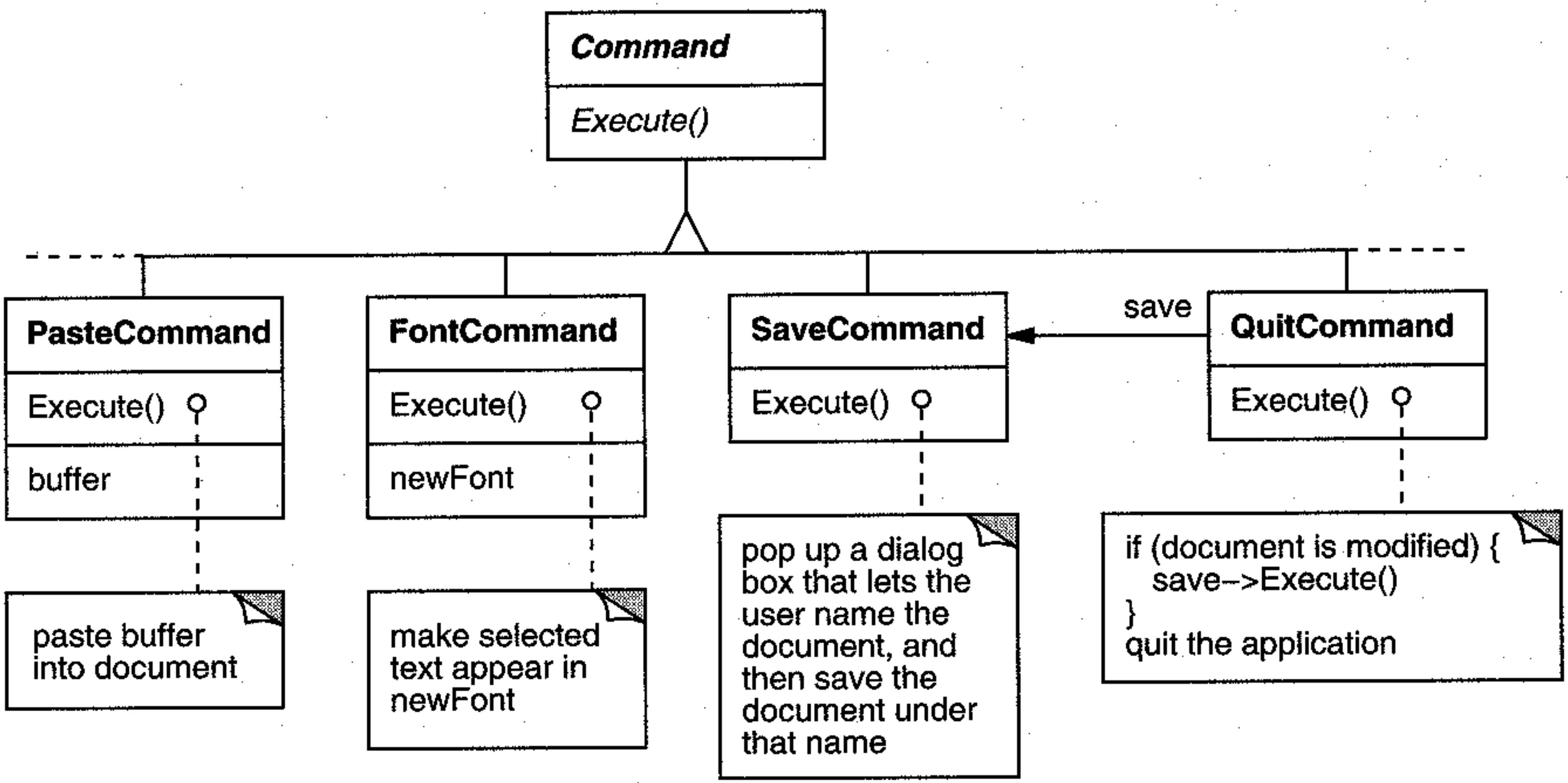


Figure 2.11: Partial Command class hierarchy

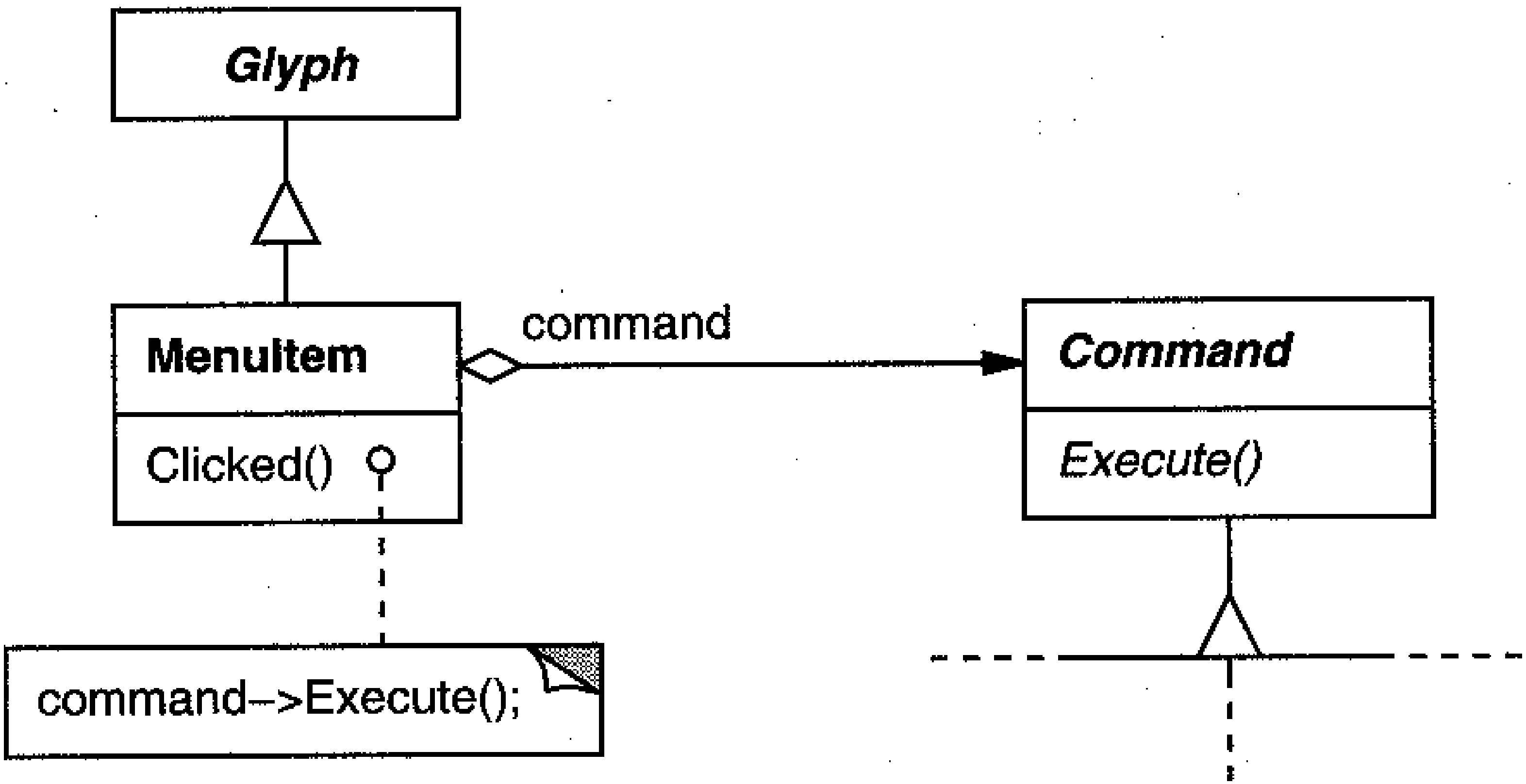
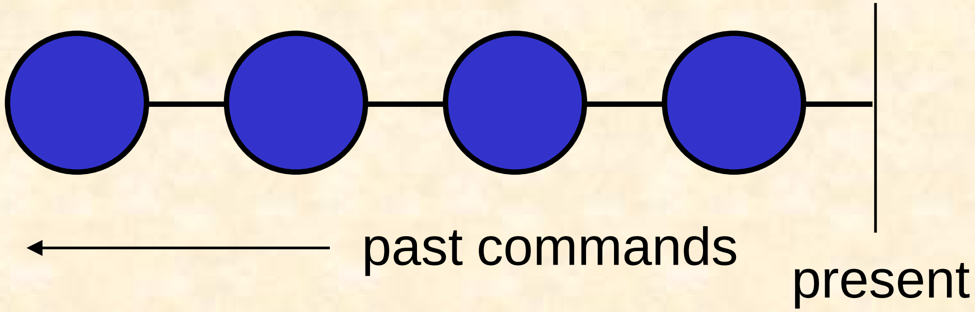


Figure 2.12: MenuItem-Command relationship

User Operations (cont.)

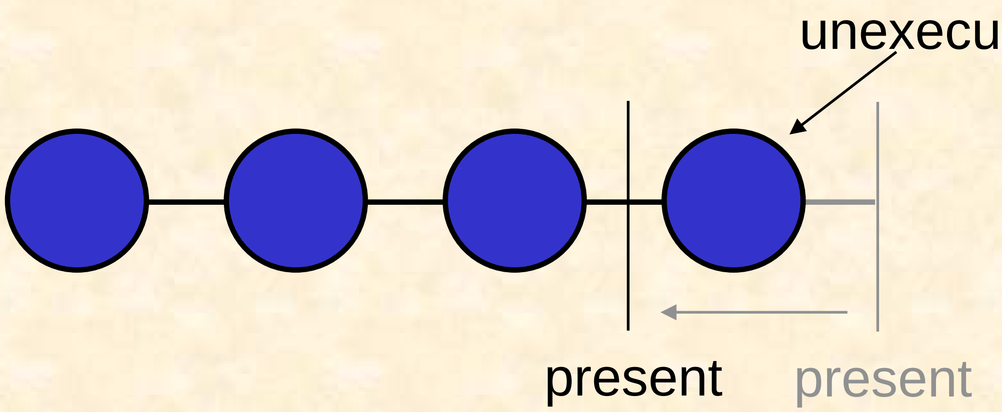
- Undoability
  - To undo and redo commands, we add an *Unexecute* operation to Command's interface
  - A concrete Command would store the state of the Command for Unexecute
  - Reversible operation returns a Boolean value to determine if a command is undoable
- Command History
  - a list of commands that have been executed

Implementing a Command History



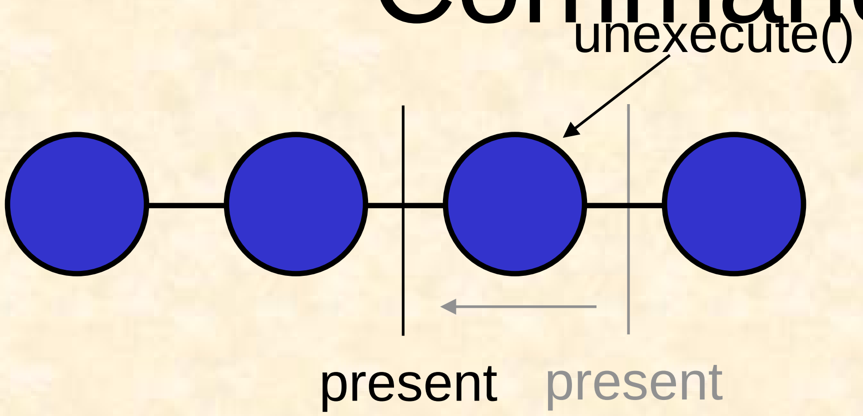
- The command history can be seen as a list of past commands commands
- As new commands are executed they are added to the front of the history

Undoing the Last Command



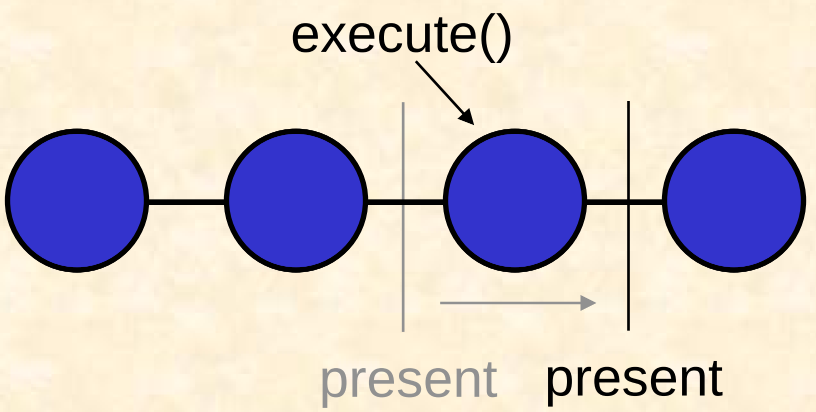
- To undo a command, unexecute() is called on the command on the front of the list
- The "present" position is moved past the last command

Undoing the Previous Command



- To undo the previous command, unexecute() is called on the next command in the history
- The present pointer is moved to point before that command

Redoing the Next Command



- To redo the command that was just undone, execute() is called on that command
- The present pointer is moved up past that command

The Command Pattern

- Encapsulate a request as an object
- The Command Patterns lets you
  - parameterize clients with different requests
  - queue or log requests
  - support undoable operations
- Also Known As: Action, Transaction
- Covered on pg. 233 in the book

Spelling Checking & Hyphenation

Goals:

- analyze text for spelling errors
- introduce potential hyphenation sites

Constraints/forces:

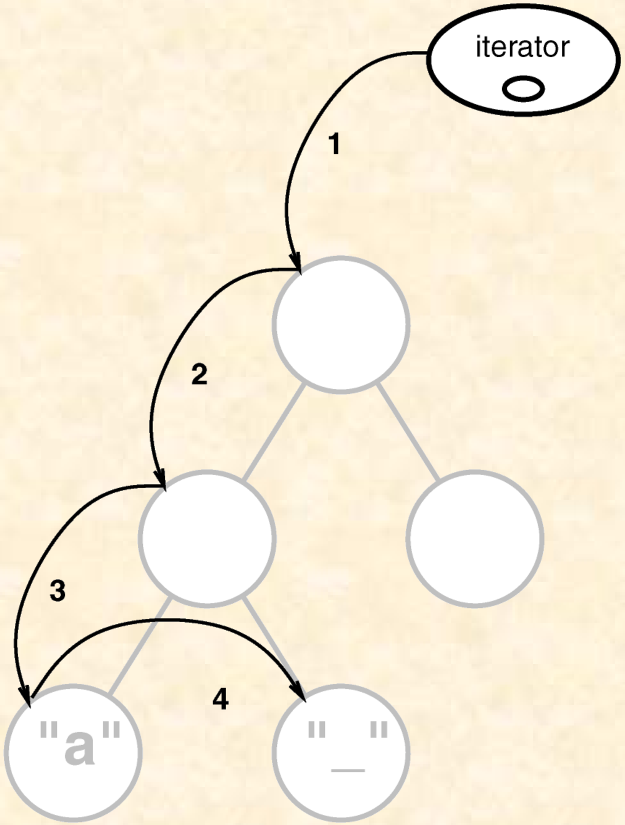
- support multiple algorithms
- don’t tightly couple algorithms with document structure

Spelling Checking & Hyphenation (cont'd)

Solution: Encapsulate Traversal

Iterator

- encapsulates a traversal algorithm without exposing representation details to callers
- uses Glyph's child enumeration operation
- This is an example of a "preorder iterator"



Spelling Checking & Hyphenation (cont'd)

TERATOR object behavioral

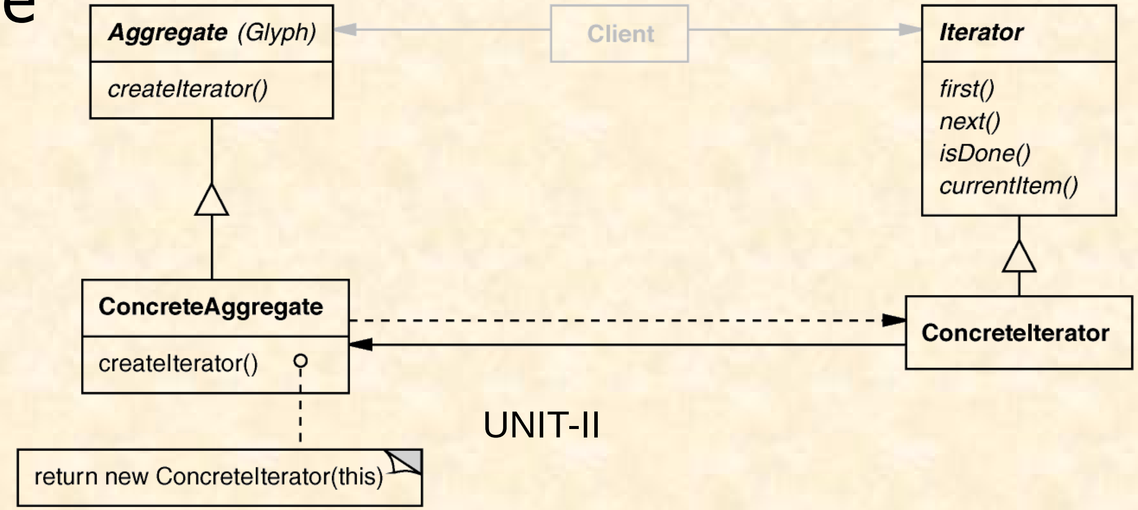
Intent

access elements of a container without exposing its representation

Applicability

- require multiple traversal algorithms over a container
- require a uniform traversal interface over different containers
- when container classes & traversal algorithm must vary independently

Structure



Spelling Checking & Hyphenation (cont'd)

TERATOR (cont'd) object behavioral

Iterators are used heavily in the C++ Standard Template Library (STL)

```
int main (int argc, char *argv[]) {
   vector<string> args;
   for (int i = 0; i < argc; i++)
        args.push_back (string (argv[i]));
   for (vector<string>::iterator i (args.begin ());
         i != args.end ();
         i++)
     cout << *i;
   cout << endl;
   return 0;
}
```

The same iterator pattern can be applied to any STL container!

```
for (Glyph::iterator  i = glyphs.begin ();
     i != glyphs.end ();
     i++)
   ...
```

Spelling Checking & Hyphenation (cont'd)

TERATOR (cont'd) object behavioral

Consequences

+ flexibility: aggregate & traversal are independent
+ multiple iterators & multiple traversal algorithms
– additional communication overhead between iterator & aggregate

Implementation

- internal versus external iterators
- violating the object structure's encapsulation
- robust iterators
- synchronization overhead in multi-threaded programs
- batching in distributed & concurrent programs

Known Uses

- C++ STL iterators
- JDK Enumeration, Iterator
- Unidraw Iterator

Spelling Checking & Hyphenation (cont'd)

Visitor

- defines action(s) at each step of traversal
- avoids wiring action(s) into Glyphs
- iterator calls glyph's **accept(Visitor)** at each node
- **accept()** calls back on visitor (a form of "static polymorphism" based on method overloading by type)

```
void Character::accept (Visitor &v) { v.visit (*this); }

class Visitor {
public:
    virtual void visit (Character &);
    virtual void visit (Rectangle &);
    virtual void visit (Row &);
    // etc. for all relevant Glyph subclasses
};
```

Spelling Checking & Hyphenation (cont'd)
SpellingCheckerVisitor

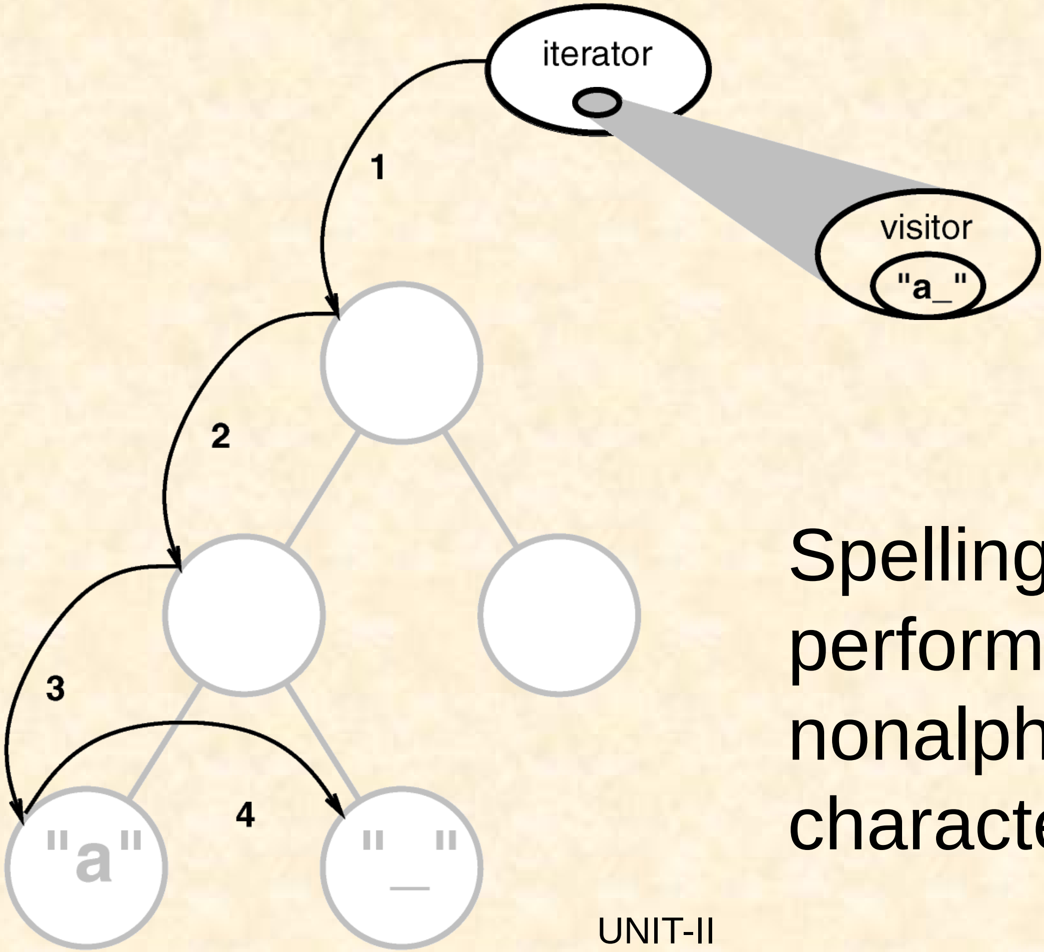
- gets character code from each character glyph

  Can define **getCharCode()** operation just on **Character()** class
- checks words accumulated from character glyphs
- combine with **PreorderIterator**

```
class SpellCheckerVisitor : public Visitor {
public:
    virtual void visit (Character &);
    virtual void visit (Rectangle &);
    virtual void visit (Row &);
    // etc. for all relevant Glyph subclasses
Private:
  std::string accumulator_;
};
```

Spelling Checking & Hyphenation (cont'd)

Accumulating Words

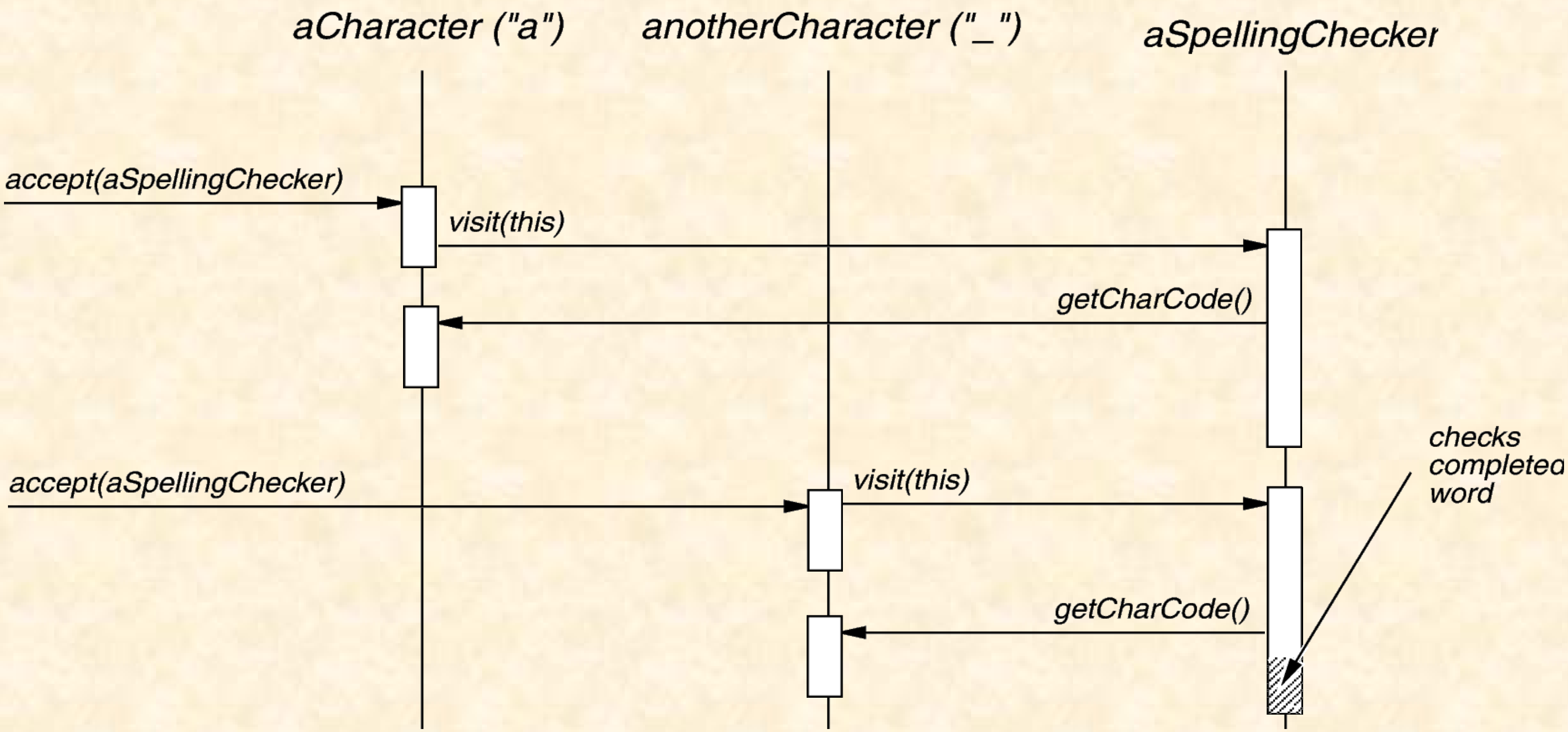


Spelling check performed when a nonalphabetic character it reached

Spelling Checking & Hyphenation (cont'd)

Interaction Diagram

- The iterator controls the order in which accept() is called on each glyph in the composition
- accept() then "visits" the glyph to perform the desired action
- The Visitor can be sub-classed to implement various desired actions



Spelling Checking & Hyphenation (cont'd)

HyphenationVisitor

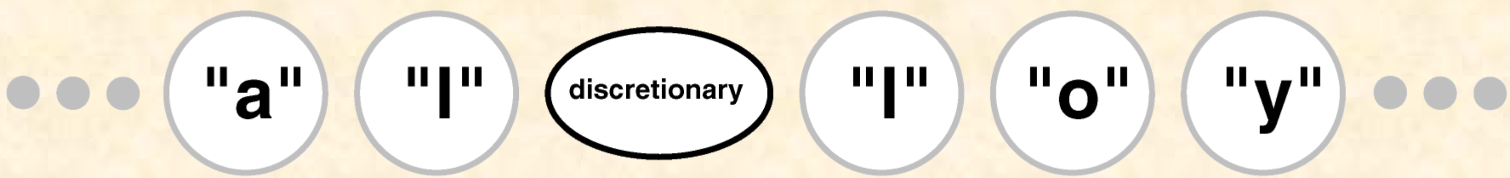
- gets character code from each character glyph
- examines words accumulated from character glyphs
- at potential hyphenation point, inserts a...

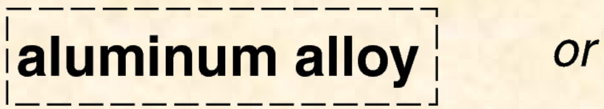
```
class HyphenationVisitor : public Visitor {
public:
    void visit (Character &);
    void visit (Rectangle &);
    void visit (Row &);
    // etc. for all relevant Glyph subclasses
};
```

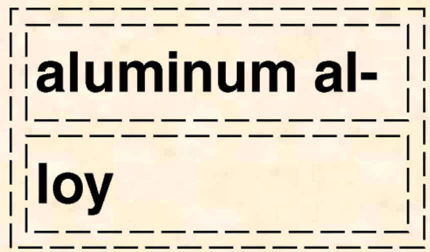
Spelling Checking & Hyphenation (cont'd)

Discretionary Glyph

- looks like a hyphen when at end of a line
- has no appearance otherwise
- Compositor considers its presence when determining linebreaks







Spelling Checking & Hyphenation (cont'd)

VISITOR object behavioral

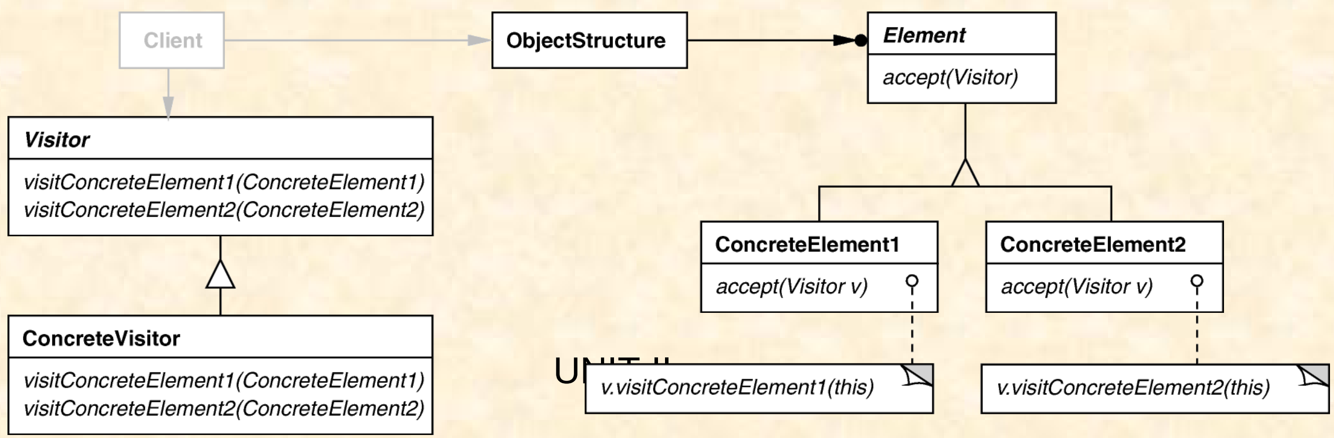
Intent

centralize operations on an object structure so that they can vary independently but still behave polymorphically

Applicability

- when classes define many unrelated operations
- class relationships of objects in the structure rarely change, but the operations on them change often
- algorithms keep state that's updated during traversal

Structure



VISITOR (cont'd) object behavioral

```
SpellCheckerVisitor spell_check_visitor;

for (Glyph::iterator i = glyphs.begin ();
     i != glyphs.end ();
     i++) {
   (*i)->accept (spell_check_visitor);
}

HyphenationVisitor hyphenation_visitor;

for (Glyph::iterator i = glyphs.begin ();
     i != glyphs.end ();
     i++) {
   (*i)->accept (hyphenation_visitor);
}
```

Spelling Checking & Hyphenation (cont'd)

VISITOR (cont'd) object behavioral

Consequences

- + flexibility: visitor & object structure are independent
- + localized functionality
- – circular dependency between Visitor & Element interfaces
- – Visitor brittle to new ConcreteElement classes

Implementation

- – double dispatch
- – general interface to elements of object structure

Known Uses

- – ProgramNodeEnumerator in Smalltalk-80 compiler
- – IRIS Inventor scene rendering
- – TAO IDL compiler to handle different backends

Part III: Wrap-Up

Concluding Remarks

- *design* reuse
- uniform design vocabulary
- understanding, restructuring, & team communication
- provides the basis for automation
- a “new” way to think about design

Pattern References

Books

Timeless Way of Building, Alexander, ISBN 0-19-502402-8

A Pattern Language, Alexander, 0-19-501-919-9

Design Patterns, Gamma, et al., 0-201-63361-2 CD version 0-201-63498-8

Pattern-Oriented Software Architecture, *Vol. 1*, Buschmann, et al., 0-471-95869-7

Pattern-Oriented Software Architecture, Vol. 2, Schmidt, et al., 0-471-60695-2

Pattern-Oriented Software Architecture, Vol. 3, Jain & Kircher, 0-470-84525-2

Pattern-Oriented Software Architecture, Vol. 4, Buschmann, et al., 0-470-05902-8

Pattern-Oriented Software Architecture, Vol. 5, Buschmann, et al., 0-471-48648-5