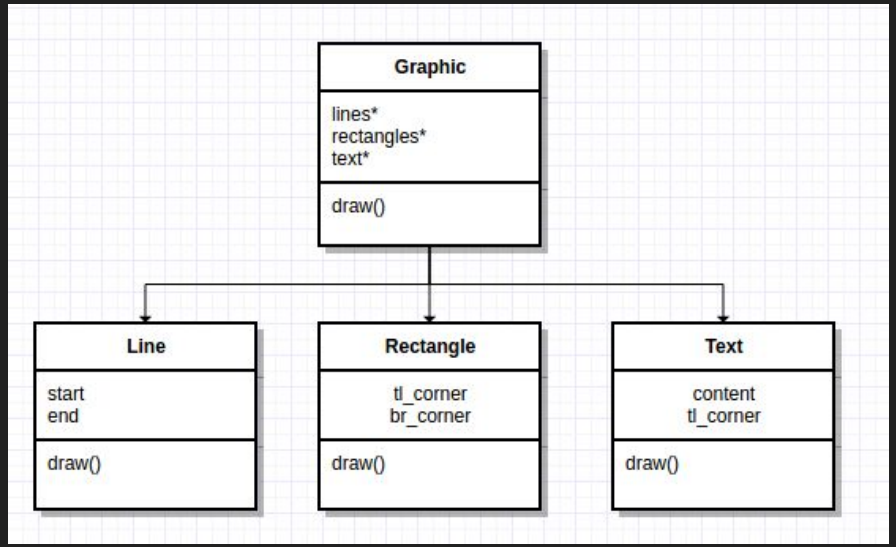
Composite Pattern

Structural, Object focused

Motivation

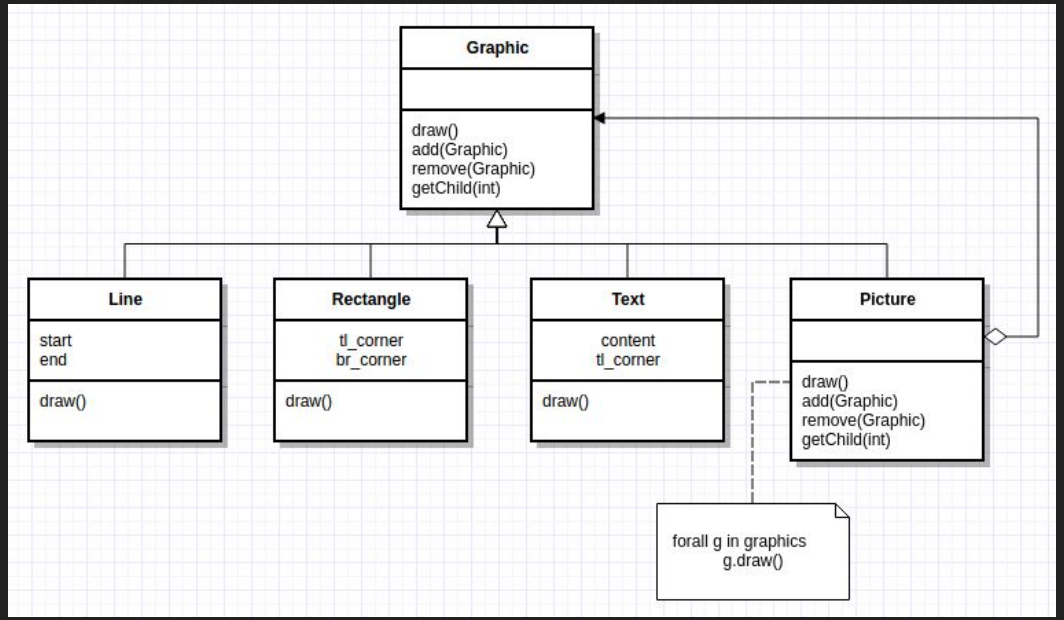
Often we want to represent both simple and complex hierarchies uniformly. For example, graphics programs allow users to build complex diagrams out of simple primitive components. Imagine the following implementation:

- **Primitives** (line, rectangle, text) each have their own class
- More complex graphics are composed of those primitives or other graphics (compositions)
- **Primitives** and **compositions** are different classes



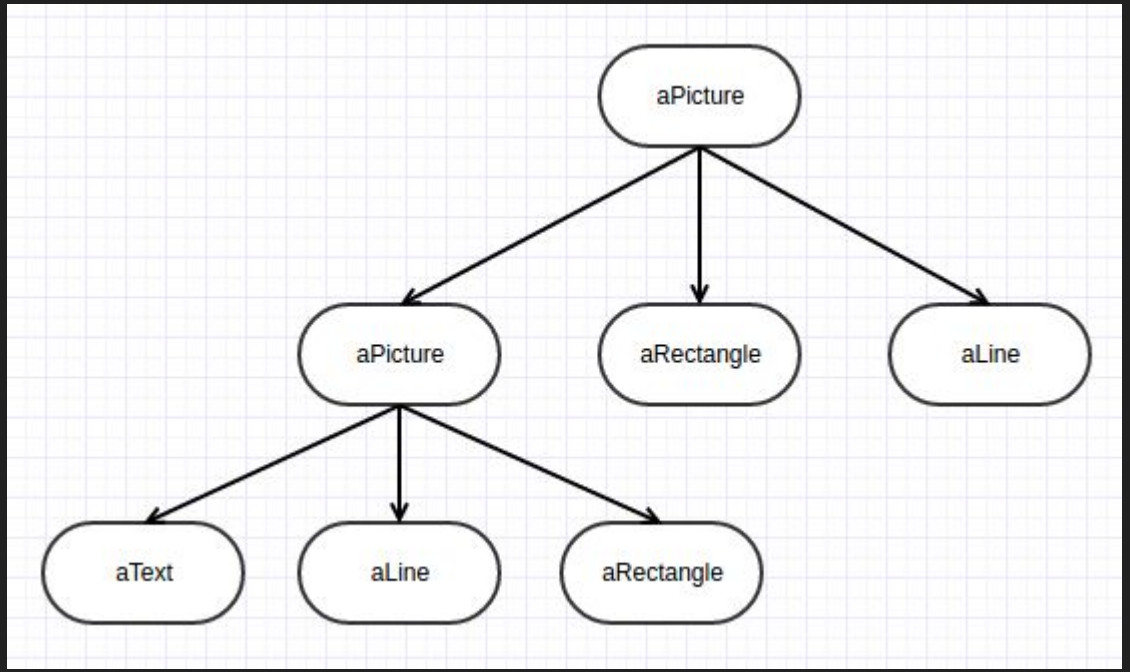
Motivation

However, clients typically treat primitives **and** compositions the same. Notice the “draw()” function in each class



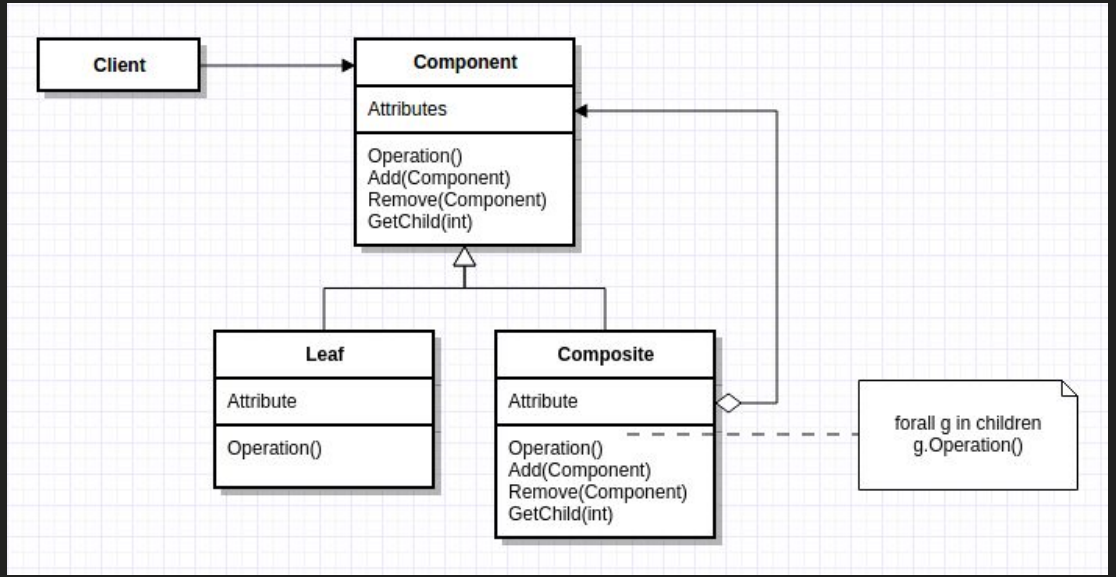
Motivation

Creates a hierarchy of objects using recursively composed Graphic objects



Structure

- **Component**
  - Declares the *interface*
  - default behavior
- **Leaf**
  - behavior of **primitives**
- **Composite**
  - Stores children
  - Implements child related operations
- **Client**
  - Manipulates composition through interface



Consequences

- Pros:
  - Defines class hierarchies consisting of **primitive** and **composite** objects
    - Primitive objects can be composed into more complex objects (composites)
  - Makes client interaction simple. Clients treat composites and primitive the same, using the same *interface*
  - Adding additional components (primitives/composites) is straightforward
- Cons:
  - Design is overly general
  - Harder to restrict components of a composite

Example from “Design Patterns: Elements of Reusable Object-Oriented Software”

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Designing A Document Editor

We are designing a “What-You-See-Is-What-You-Get” (or “WYSIWYG”) document editor called Lexi. The document editor can

- mix text and graphics freely
- Utilize a variety of formatting styles
- etc.

Surrounding the document are

- pull-down menus
- scroll bars,
- a collection of page icons for jumping to a particular page in the document
- etc.

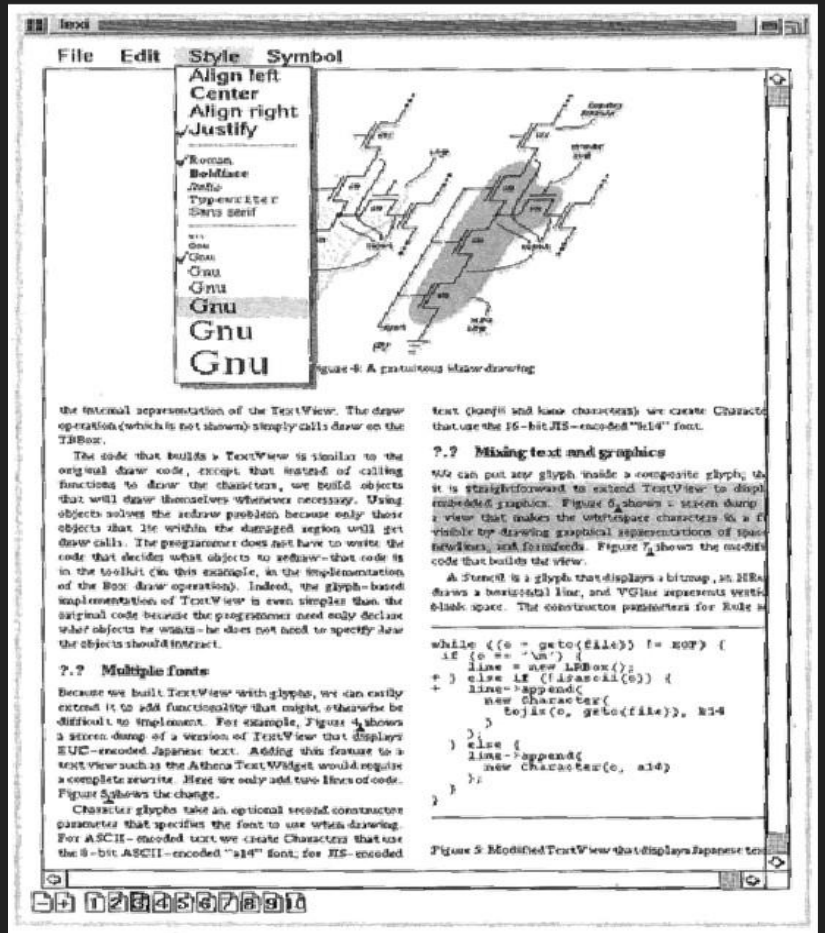


Figure 2.1: Lexi's user interface

Document Structure

- The choice of internal representation for the document affects almost every aspect of Lexi’s design
- All editing, formatting, displaying and textual analysis will require traversing the representation
- The way we organize the information will impact the design of the rest of the application

Document Structure

- A document is ultimately just an arrangement of basic graphical elements such as character’s, lines, polygons and other shapes
- Authors often view the document in terms of its physical structure - lines, columns, figures, tables, etc.
- Each substructure has substructures of their own
    - columns with lines, lines with figures, tables with columns and rows, etc.

Document Structure

The internal representation should support the following:

1. Maintaining the document's physical structure, that is, the arrangement of text and graphics into lines, columns, tables, etc.
2. Generating and presenting the document visually
3. Mapping positions on the display to elements in the internal representation. This lets Lexi determine what the user is referring to when he/she points to something in the visual representation

Document Structure

It should also have the following constraints:

1. Text and graphics should be treated uniformly
2. Our implementation shouldn’t have to distinguish between single elements and groups of elements in the internal representation
3. We will still need to analyze text for things like spelling errors and potential hyphenation points

Recursive Composition

- A common way to represent hierarchically structured information is through a technique called **recursive composition**.
- Increasingly complex elements are built out of simpler ones through a common base class using **inheritance**
- Sets of characters and graphics can be tiled into rows, and groups of rows in columns, and groups of columns in pages, etc.

Recursive Composition

- The physical structure can be represented by devoting an object to each important element
- These elements are not always visible elements (characters, graphics, etc.) but also invisible structural elements (rows, columns, pages, etc.)
- Invisible structural elements will be made up of collections of visible elements, or collections of other invisible elements in a tree like hierarchical structure

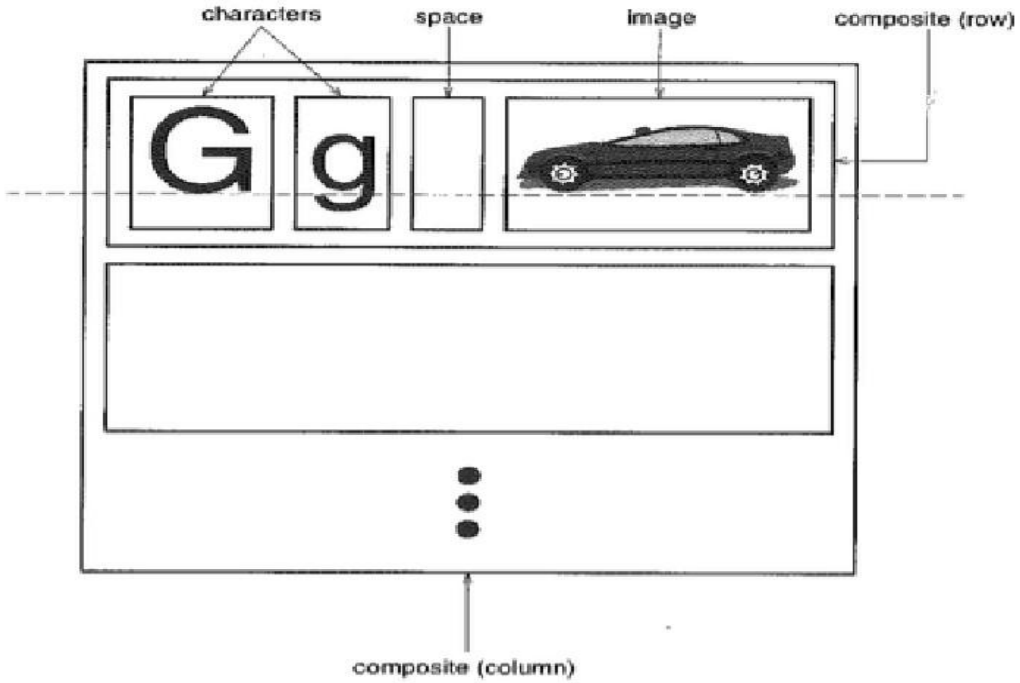


Figure 2.2: Recursive composition of text and graphics

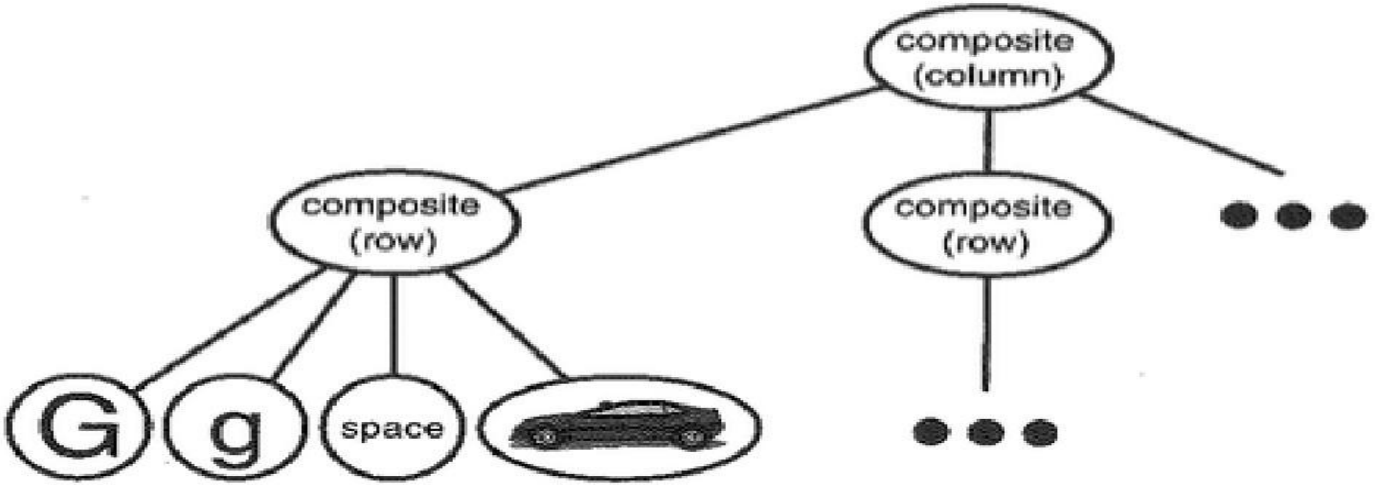
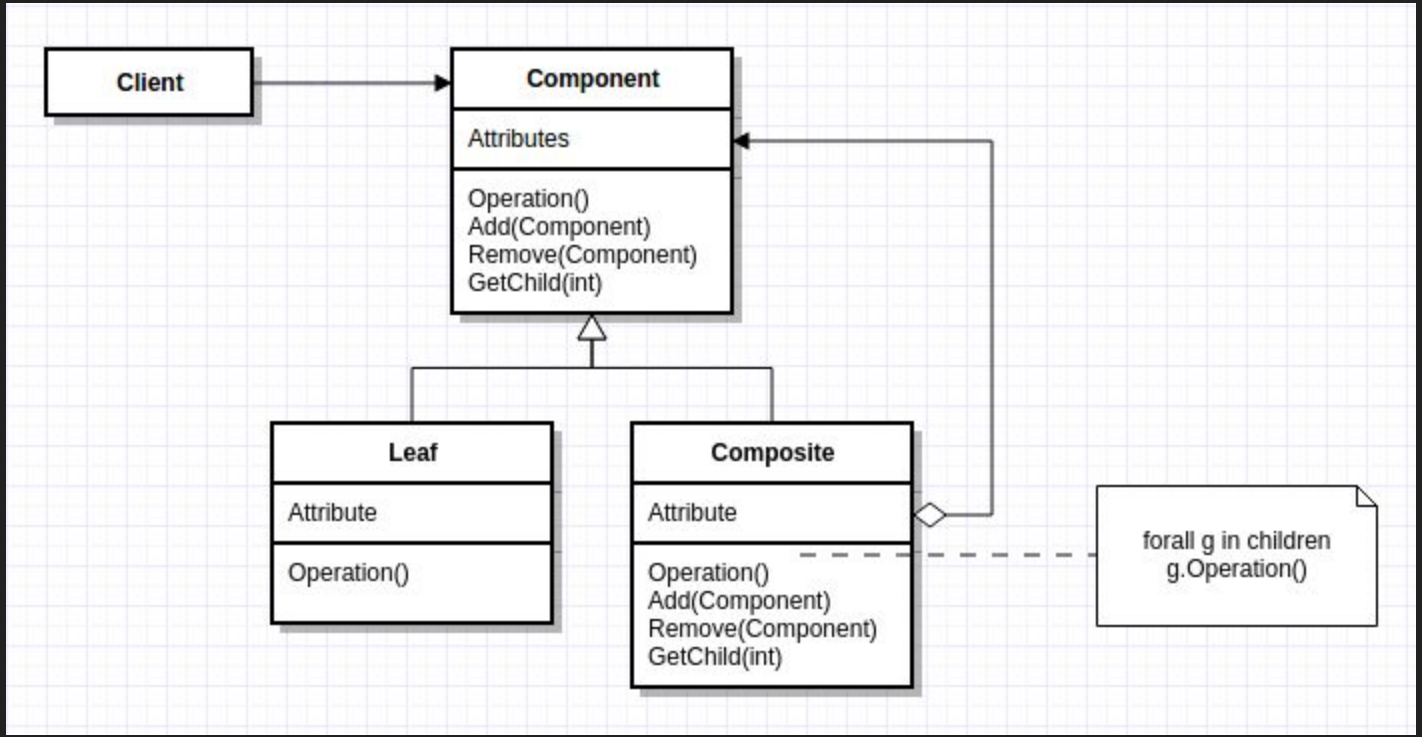


Figure 2.3: Object structure for recursive composition of text and graphics


Client
Component
Attributes
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Attribute
Operation()
Composite
Attribute
Operation()
Add(Component)
Remove(Component)
GetChild(int)
forall g in children
g.Operation()

Glyphs

- We start with an abstract base class for all objects that can appear in a document structure called **`Glyph`**
- Its subclasses define both
  - base graphical elements (characters, graphics; **Leaf or primitives**), and
  - Structural elements (rows, columns, pages; **Composites**)
- What common interface(s) make sense for all of these objects to share in common?

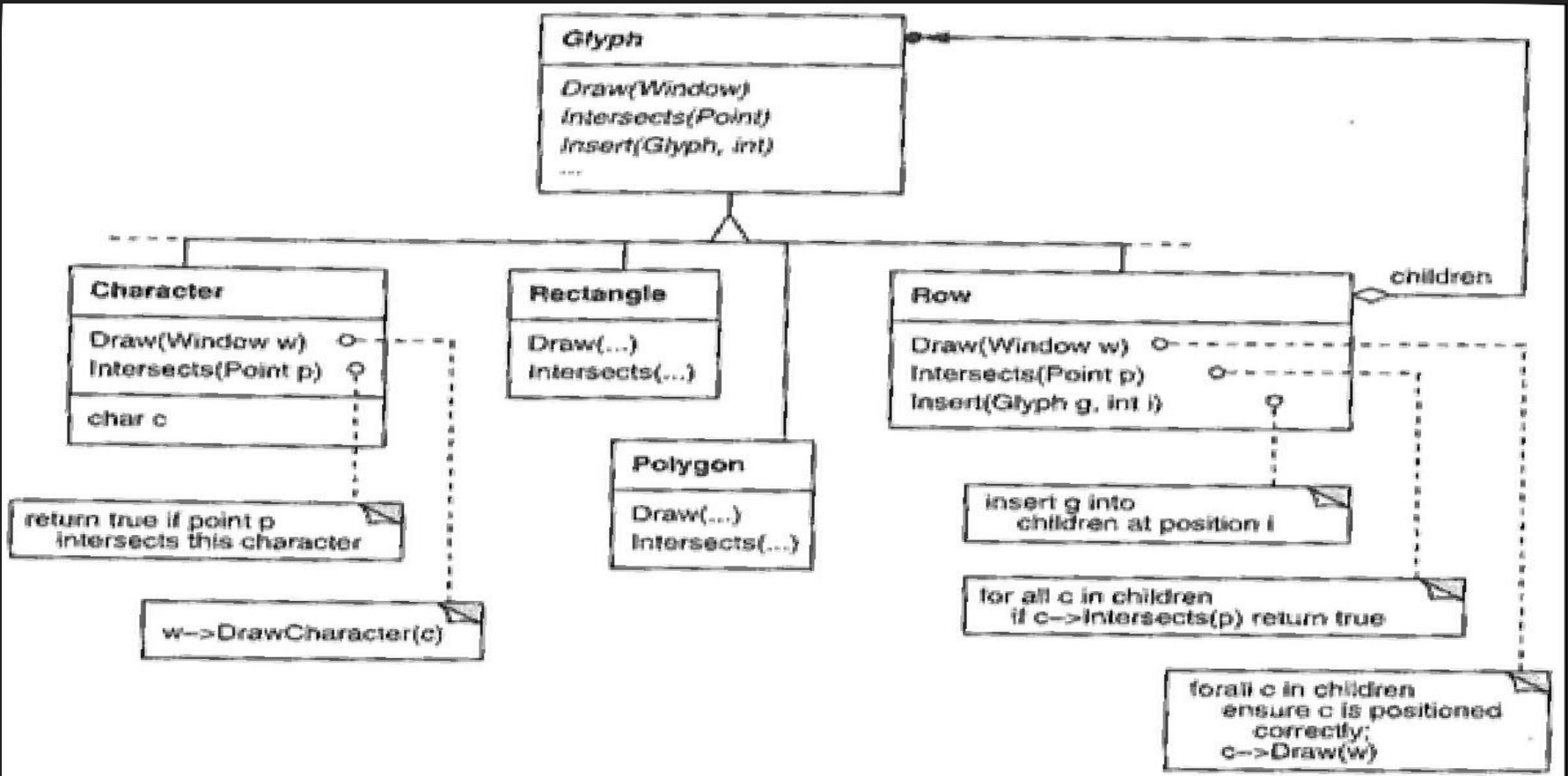


Figure 2.4: Partial Glyph class hierarchy

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

Glyphs

- Glyphs have three basic responsibilities
  1. They know how to draw themselves
  2. They know what space they occupy
  3. They know their children and parent

Glyphs

- Each Glyph subclass will redefine the `draw` operation to render themselves onto a Window (more on Window later)
- A Rectangle subclass of Glyph might redefine `draw` like this

```
void Rectangle::draw(Window* w) {
    drawRect(w->x + this->ul.x, w->y + this->ul.y,
        w->x + this->lr.x, w->y + this->lr.y);
        // The window is an offset point for the rectangle
}
```

Glyphs

- A parent Glyph often needs to know how much space a child glyph occupies, for example, to arrange it and other glyphs in a line so that none overlap
- The `Bounds` operation returns the rectangular area that the glyph occupies
- This is used to return the width and height, since we don't know which the parent may need in order to properly account for a specific offset

```
void Rectangle::bounds(Rectangle& r) {
    r->ul = this->ul;
    r->lr = this->lr;
    // Since we are already drawing a rectangle, return
    // the internal attributes as the bounds
}
```

Glyphs

- The `Intersects` operation returns whether or not a specified point intersects a specific glyph, and Lexi uses it to determine what object in the structure you are clicking.
- The Rectangle class redefines this operation to compute the intersection of the rectangle and the given point, and return if their is or is not an intersection

```
void Rectangle::intersects(Point& p) {
    if(p->x > this->loc.x + this->ul.x &&
    p->x < this->loc.x + this->lr.x && ...) {
        return true;
    } else { return false; }
}
```

Glyphs

- Because glyphs can have children, we need a common interface to add, remove and access those children. For example, a Row's children are the glyphs it arranges into a row
- The `Insert` operation inserts a glyph at a position specified by an integer index, the `Remove` operation removes a specified glyph if it is indeed a child
- The `Child` operation returns the child (if any) at the given index, and glyphs with children should use `Child` internally instead of accessing the child data structure directly so we don't have to refactor the class when internal data structures change (such as going from an array to linked list)
- `Parent` provides the same type of interface to the parent