Decorator Pattern

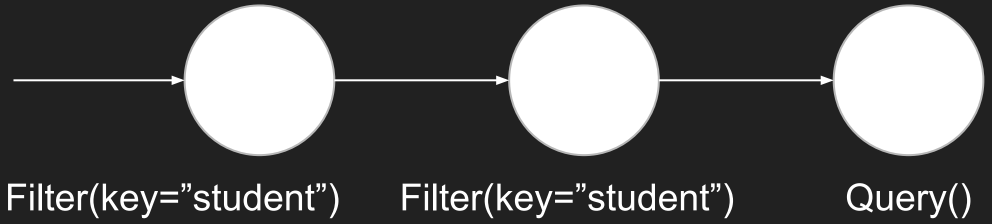
Structural, Object focused

Decorator Pattern

- The **Decorator pattern** captures class and object relationships that support "embellishment" through **transparent enclosure**
- In the Decorator pattern, embellishment refers to anything that adds *responsibilities* to an object through the modification of that objects interface (this can including adding additional "data" through an interface call)
- Transparent enclosure is a system that uses another objects interface to allow the decorator to masquerade as the object itself while "embellishing" through a modification to one (or more) interface calls (parameters or returns)

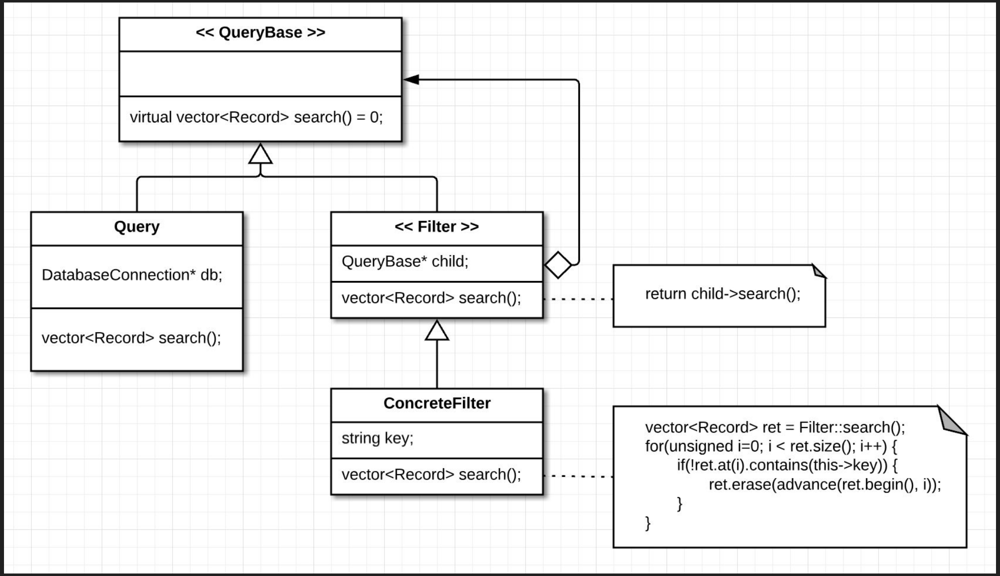
Motivation

- Sometimes you want to add minor “embellishments” or modifications to a subset of objects in your system or allow for different modifications to a single object
- Often, you want to be able to compose a number of these “embellishments” together on top of each other to create varying levels of difference
- Example: a database query is at its most basic level an iteration on tuples that returns the results, we often want to add different numbers/types of filters in order to get useful results. These filters “embellish” the return of the data



Query

- **Query/QueryBase**
  - Defines an object which can interact with the database and pull a table of records
  - This should be a base class, when looking to decorate a single class a base class interface should be created
- **Filter**
  - Interface for all different types of filters
  - Has a reference to a Query object (or the Query base class)
- **Concrete Filter**
  - Specific filter which modifies one or more interface functions
  - Removes elements in between the call to its childs search() and its own return

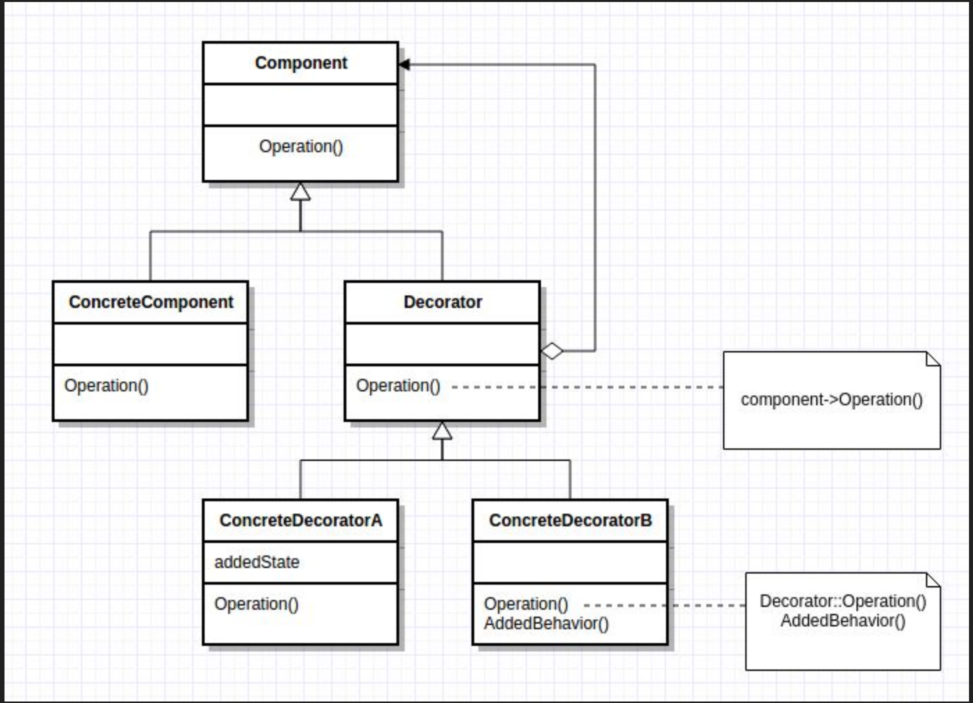


```
return child->search();
```

```
vector<Record> ret = Filter::search();
for(unsigned i=0; i < ret.size(); i++) {
      if(!ret.at(i).contains(this->key)) {
            ret.erase(advance(ret.begin(), i));
      }
}
```

Structure

- **Component**
  - Defines interface for objects that can have responsibilities added dynamically
- **ConcreteComponent**
  - Defines object to which responsibilities will be attached
- **Decorator**
  - Maintains reference to a Component object
  - Defines interface conforming to Component's interface
- **ConcreteDecorator**
  - Adds responsibilities to the component



Consequences

- Pros:
  - More flexibility than static inheritance
  - Avoids feature-laden classes high up in the hierarchy
  - Decorators and its component are **not** identical
- Cons:
  - Creates lots of little objects
  - Many of these objects look quite similar

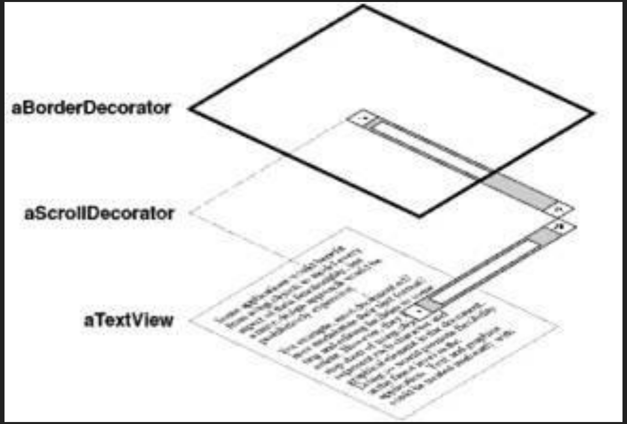
Example from “Design Patterns: Elements of Reusable Object-Oriented Software”

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Embellishing the User Interface

We will consider two embellishments in Lexi's user interface

1. Adding a border to demarcate the page of text
2. Adding scroll bars

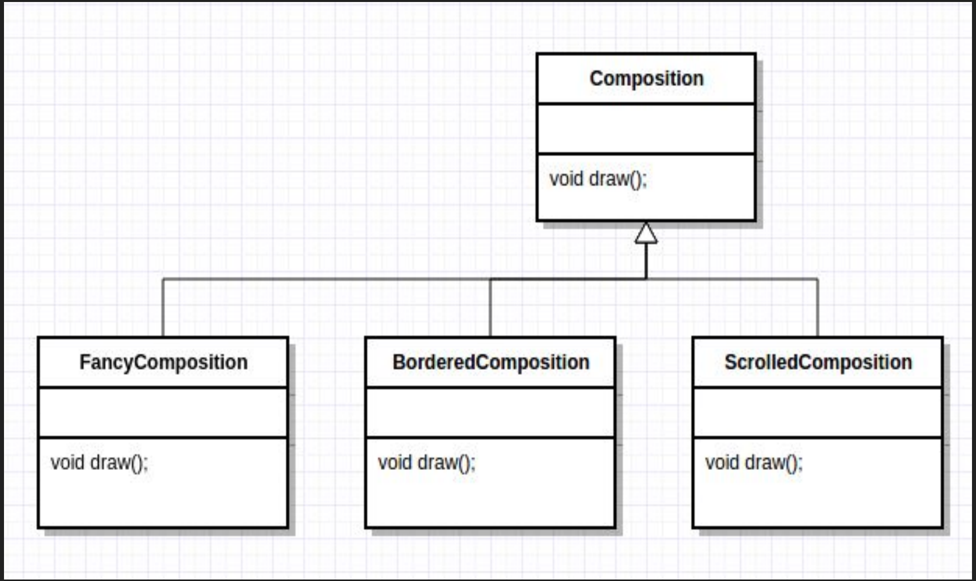


Requirements

- Easy to dynamically add and remove the embellishments
  - **Don't** use inheritance to add them to the user interface
- Flexibility is achieved by other user interfaces being oblivious to embellishments existence

Cons of using inheritance

- Adding a border to `Composition` by subclassing: `BorderedComposition`
- Adding `ScrolledComposition`
- Adding `FancyComposition`



Object Composition offers more flexibility

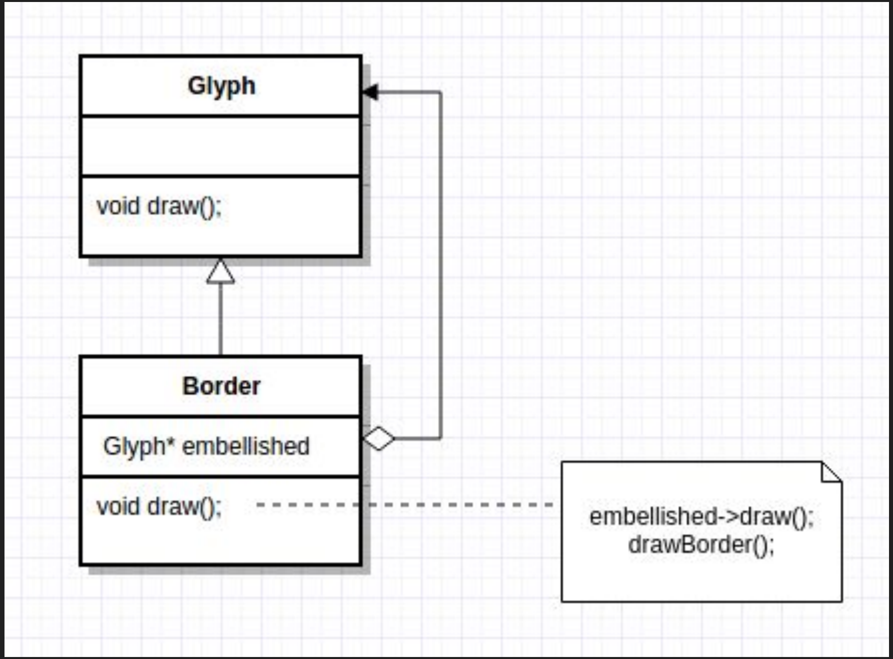
Two candidates for composing the Glyph and Embellishment objects

- Glyph contains the Embellishment (Border)
  - We have to update its subclasses to make them aware
- Embellishment (Border) contains the Glyph
  - The border-drawing code can be contained without affecting Glyph classes

How would we do this?

Creating a `Border` class

- Borders have appearances, so they should inherit from `Glyph`
- Additionally, drawn objects (`Glyphs`) should be treated uniformly
- `Border` should extend draw to both draw the `Glyph`, and the `Border`
- The `Border` interface now matches the `Glyph` interface

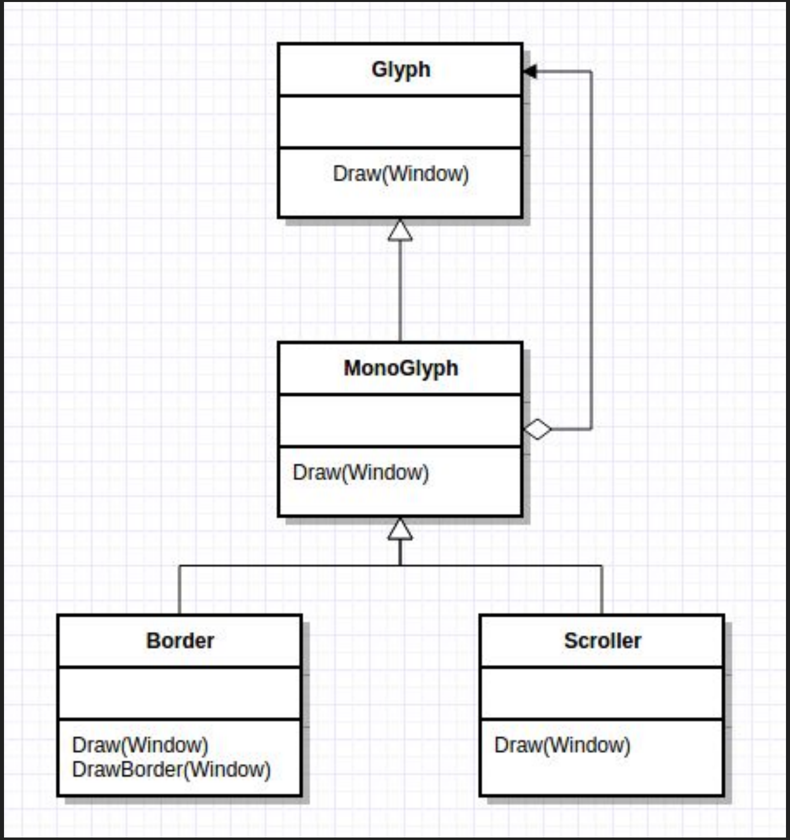


Transparent Enclosure

- The Border class utilizes **Transparent Enclosure**, combining two notions:
  - Single-child (or single-component) composition
  - Compatible interfaces
- The client can’t tell if the component they are dealing with is **enclosed** (embellished) or not.
- The enclosure augments the component’s behavior

Monoglyph

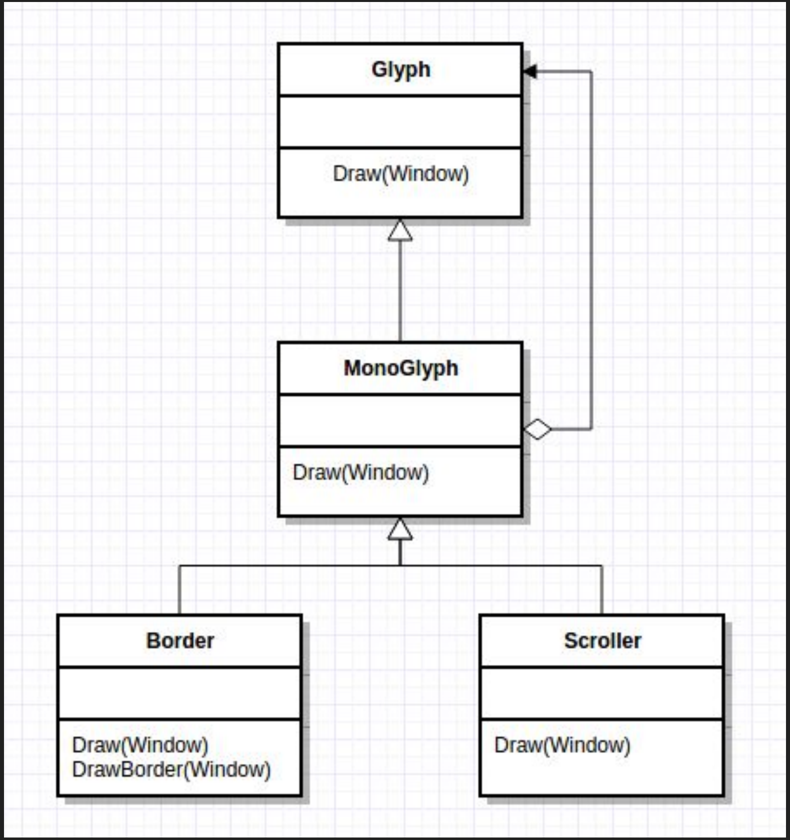
- Let’s apply the concept of **transparent enclosure** to all glyphs that embellish other `Glyphs`
- We’ll define a subclass of `Glyph` called `MonoGlyph` to serve as an **abstract class** for “embellishment glyphs” (like `Border`)



Implementation

- MonoGlyph stores a reference to a Glyph (component) and forwards all requests to it
- For example, MonoGlyph would implement the Draw operation like this:

```
void MonoGlyph::Draw(Window* w) {
        this->component->Draw(w);
}
```

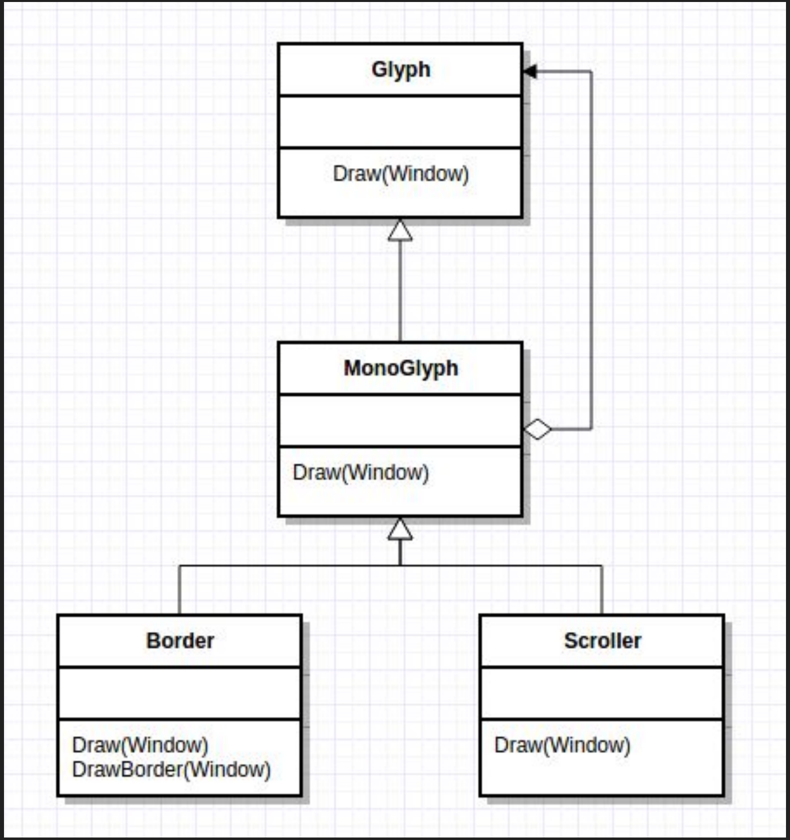


Implementation

- `MonoGlyph` subclasses reimplement forward operations

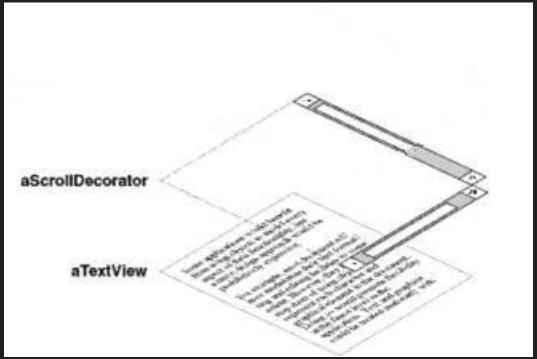
```
void Border::Draw(Window* w) {
    MonoGlyph::Draw(w);
    DrawBorder(w);
}
```

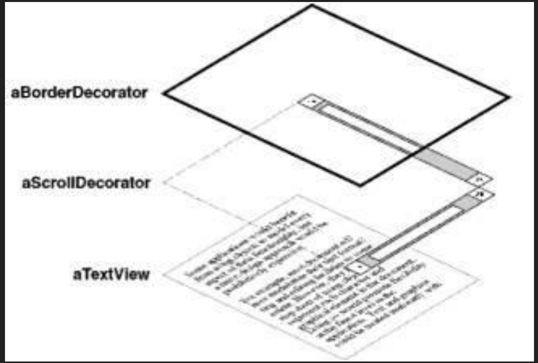
1. `Border::Draw` invokes the parent class operation `MonoGlyph::Draw`
2. `MonoGlyph::Draw` calls `Draw` on the Glyph (component)
3. Then, `Border::Draw` calls `DrawBorder(w)` to draw the border

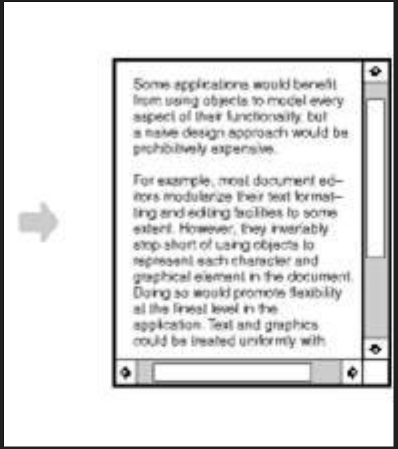


Decorating Decorators

- We can compose the existing `Composition` instance into a `Scroller` instance
  - This adds the scrolling interface
- This scrolling decorated `Composition`, the `Scroller` instance, can then be composed into a `Border` instance

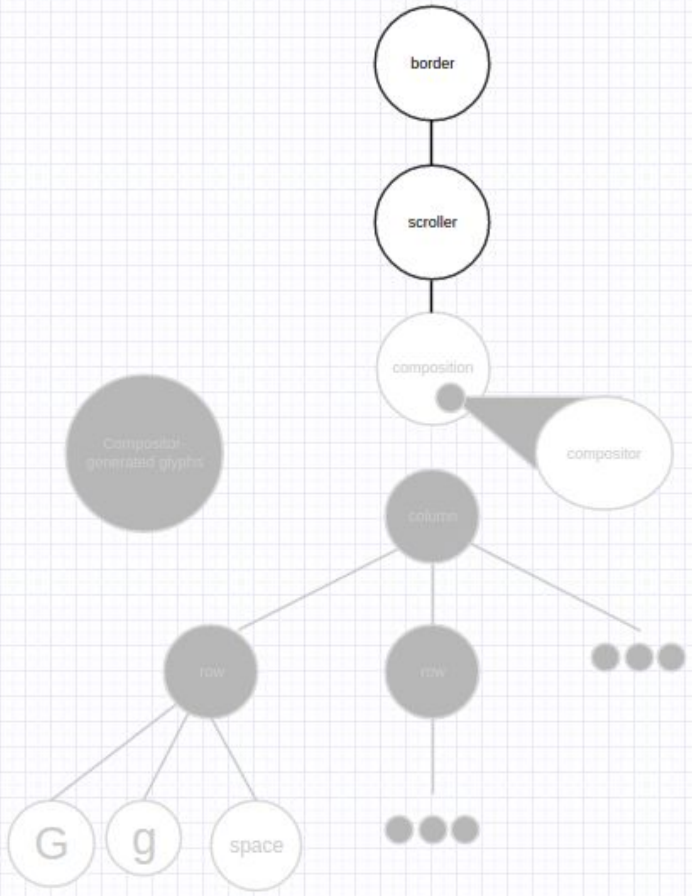






Some applications would benefit
from using objects to model every
aspect of their functionality, but
a naive design approach would be
prohibitively expensive.

For example, most document ed-
itors modularize their text format-
ting and editing facilities to some
extent. However, they invariably
stop short of using objects to
represent each character and
graphical element in the document.
Doing so would promote flexibility
at the finest level in the
application. Text and graphics
could be treated uniformly with


border
scroller
composition
compositor
Compositor-generated glyphs
column
row
row
G
g
space