Design Patterns

What are Design Patterns?

Design patterns describe common (and successful) ways of building software.

What are Design Patterns?

> "A pattern describes a problem that occurs often, along with a tried solution to the problem"
>
> - Christopher Alexander, 1977

- Idea: Problems can be similar, therefore, solutions can also be similar.
  - Not individual classes or libraries
    - Such as lists, hash tables
  - Not full designs
  - Often rely on Object-Oriented languages

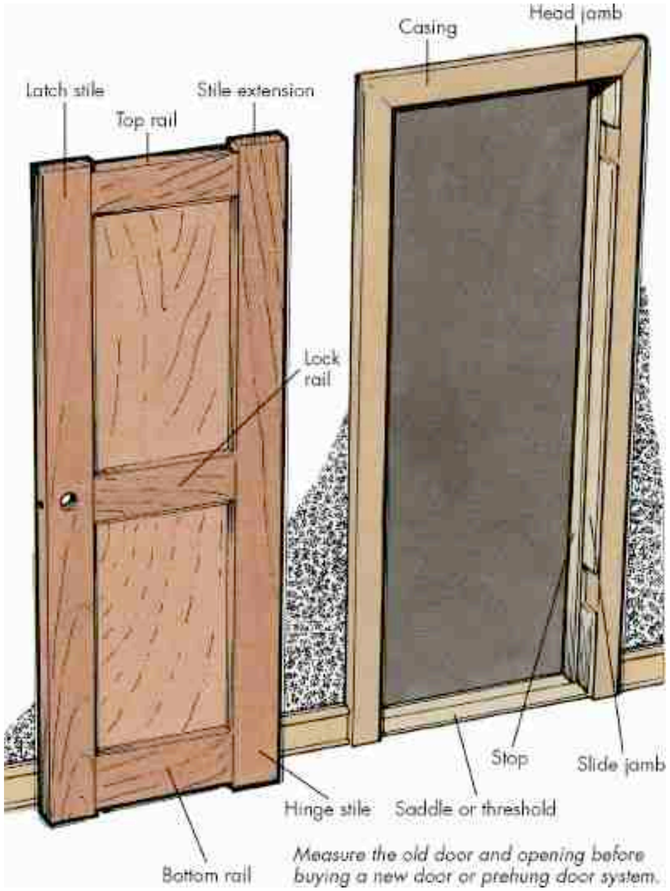
Real-World Example: the “door” pattern

- System Requirements:
  - Portal between rooms
  - Must be able to open and close
- Solution:
  - Build a door

Real-World Example: the “door” pattern

- Doors have multiple components
  - “main” part
  - “hinge” part
  - “rail” part
  - “handle” part

The **design pattern** specifies how these components interact to solve a problem



Measure the old door and opening before buying a new door or prehung door system.

Real-World Example: the "door" pattern

- The "door" design pattern is easy to reuse

- The implementation is different every time, the design is reusable.

©EVOX IMAGES

Advantages

- Teaching and learning
  - It is much easier to learn the code architecture from descriptions of design patterns than from reading code
- Teamwork
  - Members of a team have a way to name and discuss the elements of their design

What design patterns are not

- Not an architecture style:
  - Does not tell you how to structure the entire application
- Not a data structure
  - e.g. a hash table
- Not an algorithm
  - e.g. quicksort

References and resources

- GoF Design Patterns book:
  - http://c2.com/cgi/wiki?DesignPatternsBook
  - https://catalog.swem.wm.edu/Record/3301458
- Head First Design Patterns:
  - http://shop.oreilly.com/product/9780596007126.do
  - https://catalog.swem.wm.edu/Record/3302095
- Design Patterns Quick Reference Cards:
  - http://www.mcdonaldland.info/files/designpatterns/designpatternscard.pdf
- Check SWEM: “Software design patterns”

Software Design Patterns from CS 301

1. Iterator
2. Observer
3. Strategy
4. Composite
5. Decorator
6. Template
7. Singleton

also in Horstmann's book

- Adapter
- Command
- Factory
- Proxy
- Visitor

Software Example: A Text Editor

- Describe a text editor using patterns
  - A running example

- Introduces several important patterns

Note: This example is from the book "Design Patterns: Elements of Reusable Object-Oriented Software", Gamma, et al. : GoF book

Text Editor Requirements

- A WYSIWYG editor
- Text and graphics can be freely mixed
- Graphical user interface
    - Toolbars, scrollbars, etc.
- Traversal operations: spell-checking
- Simple enough for one lecture!

The Game

- I describe a design problem for the editor
- I ask “What is your design?”
  - This is audience participation time
- I give you the wise and insightful pattern

Problem: Document Structure

A document is represented by its physical structure:

- Primitive glyphs: characters, rectangles, circles, pictures, . . .
- Lines: sequence of glyphs
- Columns: A sequence of lines
- Pages: A sequence of columns
- Documents: A sequence of pages

- Treat text and graphics uniformly
  - Embed text within graphics and vice versa

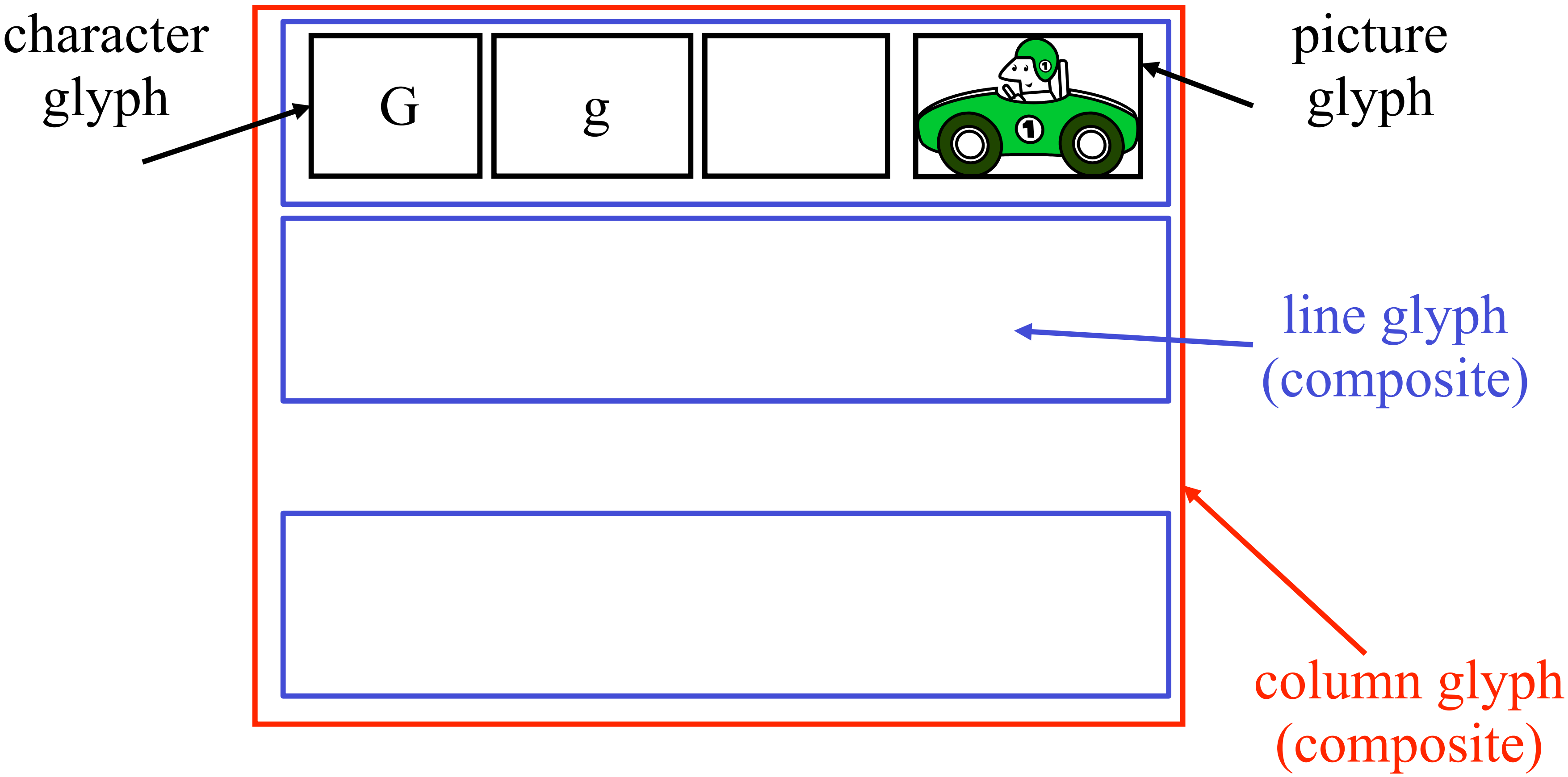
- No distinction between a single element or a group of elements
  - Arbitrarily complex documents

What is your design?

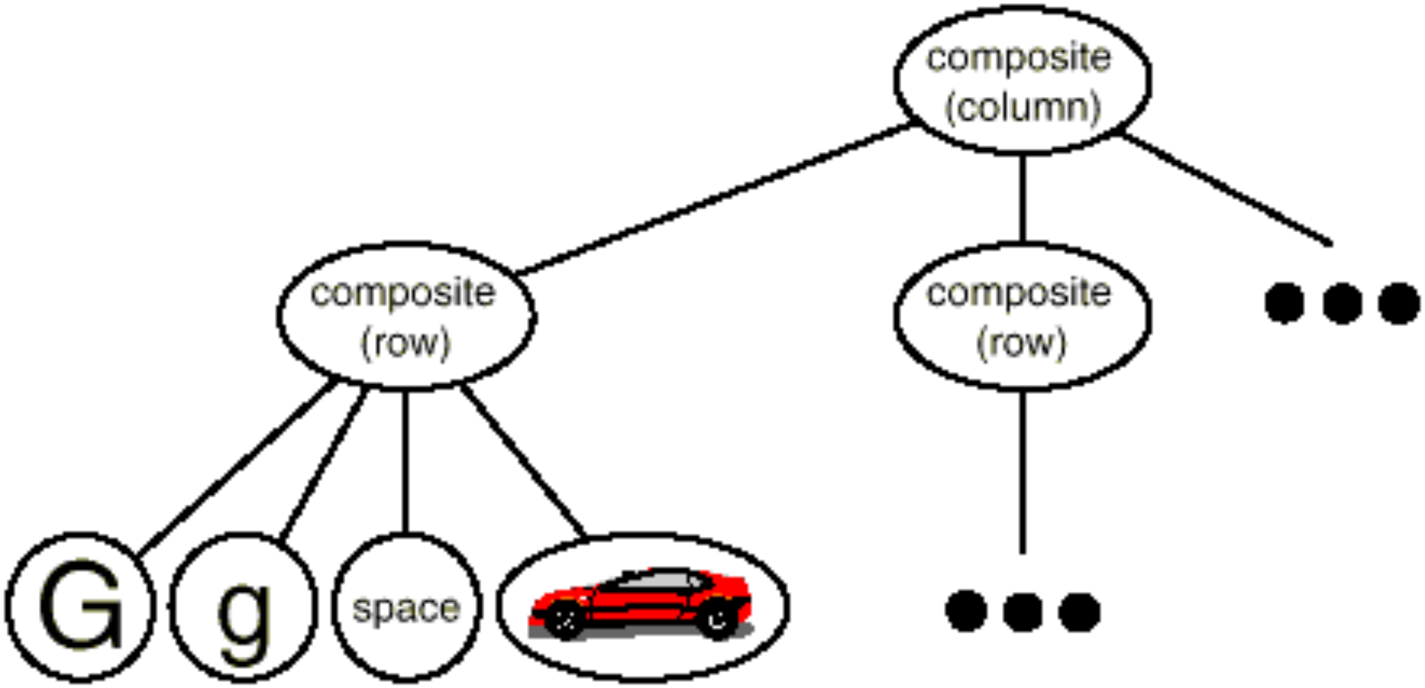
A Design

- Classes for Character, Circle, Line, Column, Page, …
  - Not so good
  - A lot of code duplication
- One (abstract) class of Glyph
  - Each element realized by a subclass of Glyph
  - All elements present the same interface
    - How to draw
    - Mouse hit detection
    - …
  - Makes extending the class easy
  - Treats all elements uniformly

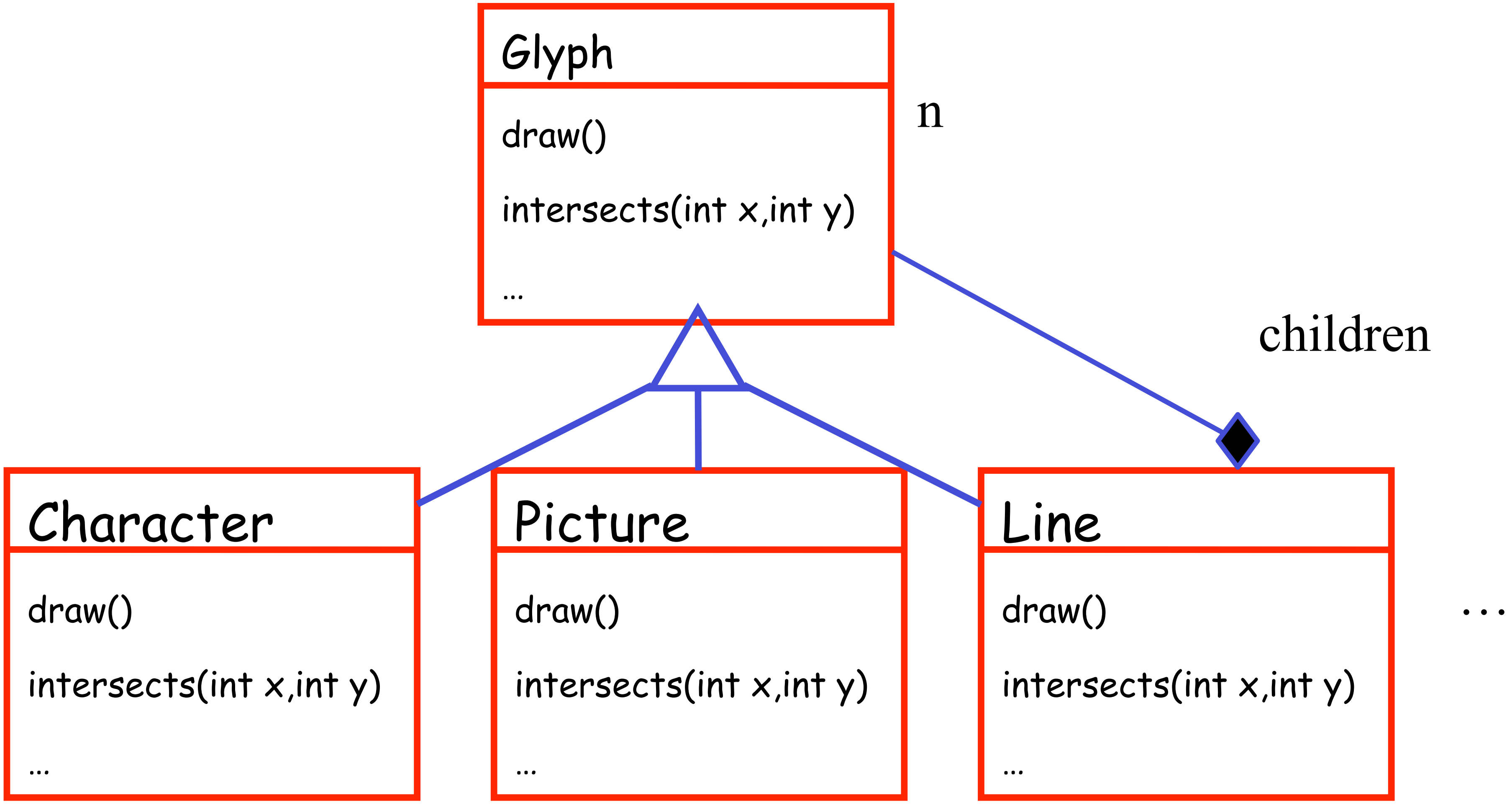
Example of Hierarchical Composition



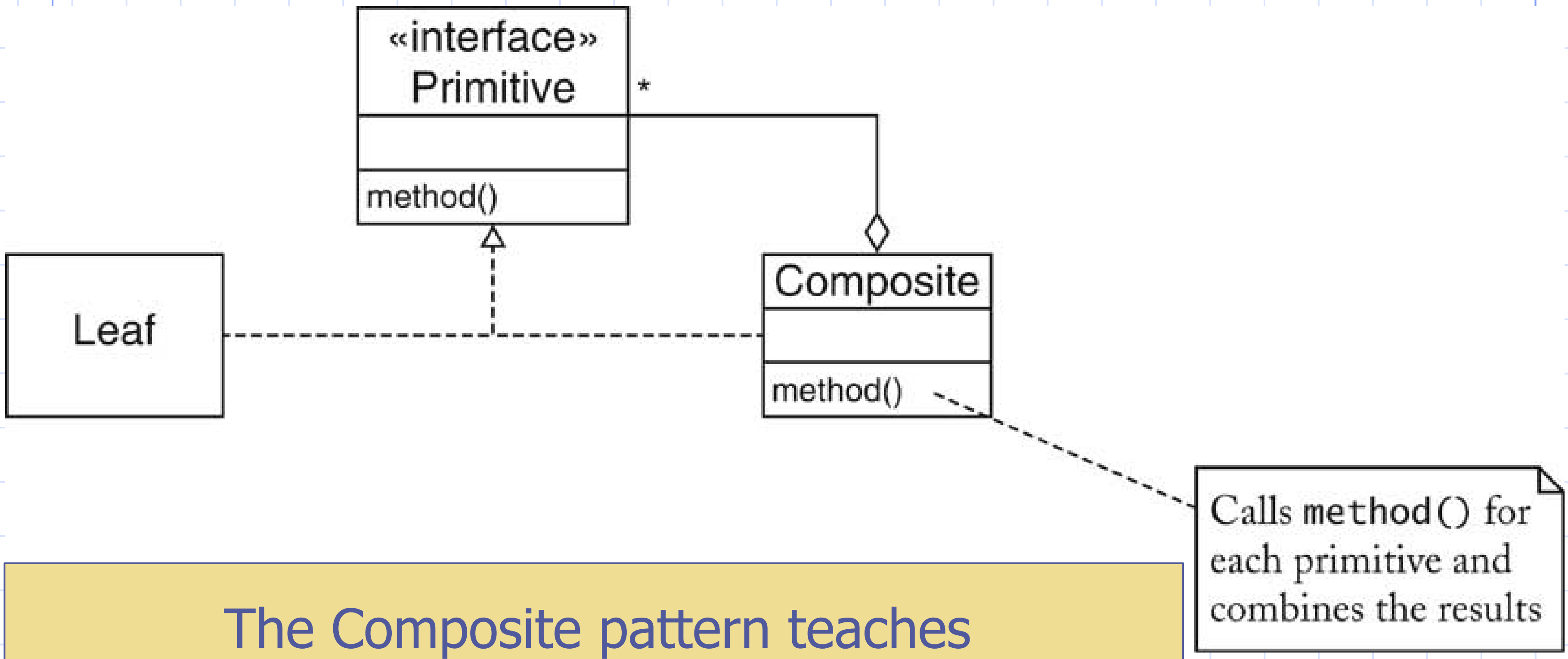
Logical Object Structure



Diagram


Glyph
draw()
intersects(int x,int y)
…
n
children
Character
draw()
intersects(int x,int y)
…
Picture
draw()
intersects(int x,int y)
…
Line
draw()
intersects(int x,int y)
…
…

Composite Pattern



The Composite pattern teaches how to combine several objects into an object that has the same behavior as its parts.

Composites

- This is the composite pattern
  - Composes objects into tree structure
  - Lets clients treat individual objects and composition of objects uniformly
  - Easier to add new kinds of components

Problem: Supporting Look-and-Feel Standards

- Different look-and-feel standards
  - Appearance of rectangles, characters, etc.
- We want the editor to support them all
  - What do we write in code like

Character ltr = new ?

What is your design?

Possible Designs

- Terrible

```
Character ltr = new MacChar();
```

- Little better

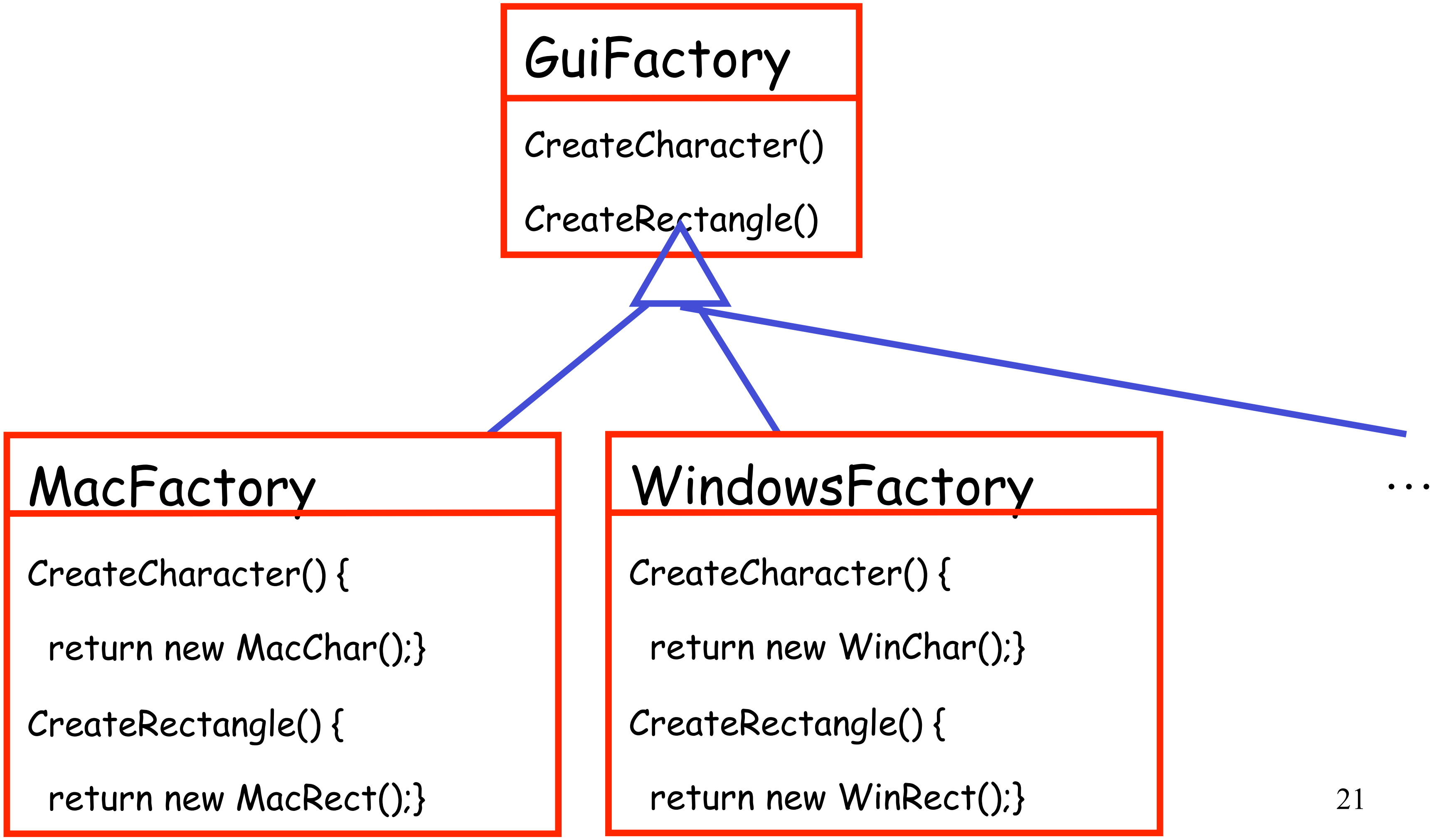
```
Character ltr;
if (style == MAC)
    scr = new MacChar();
else if (style == WINDOWS)
    scr = new WinChar();
else if (style == …)
                   ….
```

Abstract Object Creation

- Encapsulate what varies in a class
- Here object creation varies
  - Want to create different character, rectangle, etc
  - Depending on current look-and-feel

- Define a GUIFactory class
  - One method to create each look-and-feel dependent object
  - One GUIFactory object for each look-and-feel
  - Created itself using conditionals

Diagram



GuiFactory

CreateCharacter()

CreateRectangle()

MacFactory

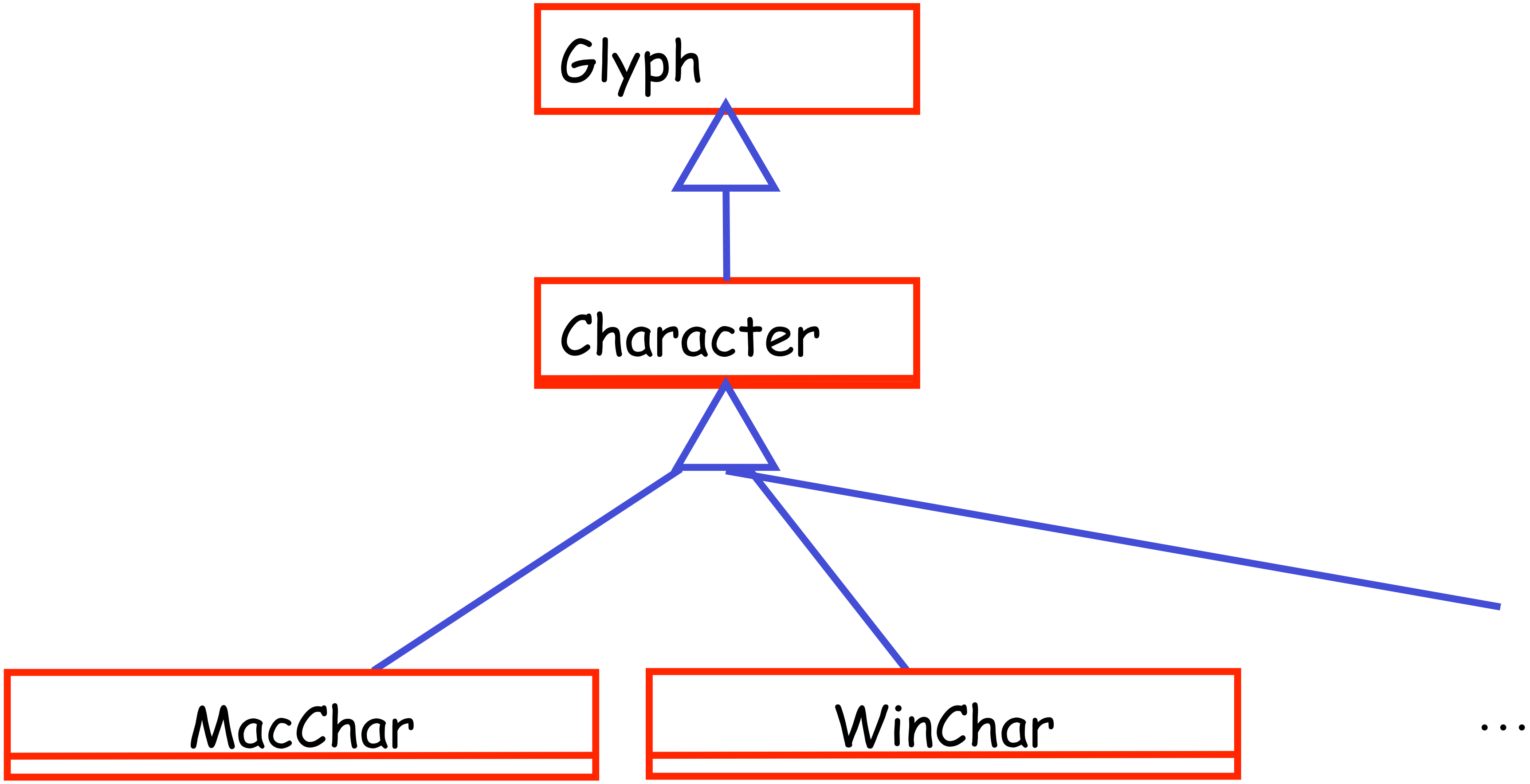
```
CreateCharacter() {
  return new MacChar();}
CreateRectangle() {
  return new MacRect();}
```

WindowsFactory

```
CreateCharacter() {
  return new WinChar();}
CreateRectangle() {
  return new WinRect();}
```

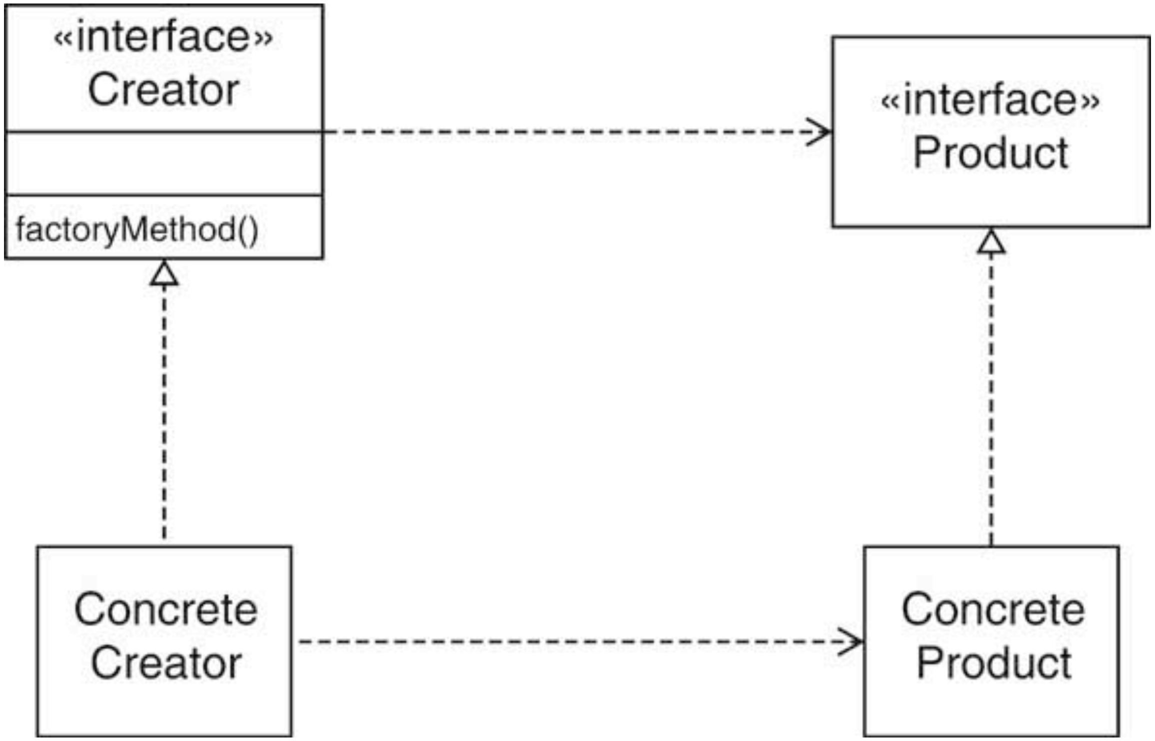
…

Diagram 2: Abstract Products



Factory Pattern

- A class which
  - Abstracts the creation of a family of objects
  - Different instances provide alternative implementations of that family



- Note
- The “current” factory is still a global variable
- The factory can be changed even at runtime

Problem: Spell Checking

- Considerations
  - Spell-checking requires traversing the document
    - Need to see every glyph, in order
    - Information we need is scattered all over the document
  - There may be other analyses we want to perform
    - E.g., grammar analysis

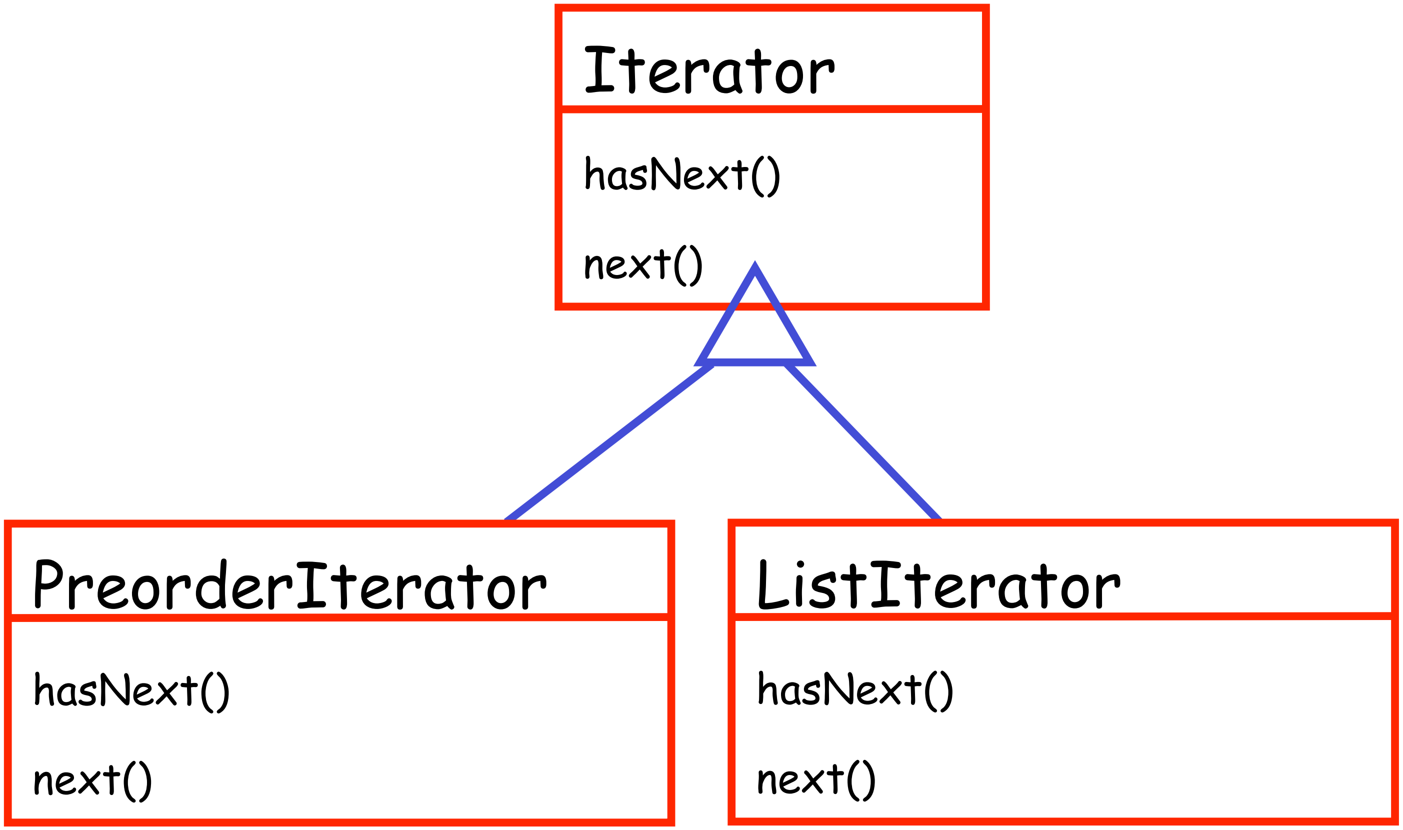
What is your design?

One Possibility

- Iterators
  - Hide the structure of a container from clients
  - A method for
    - pointing to the first element
    - advancing to the next element and getting the current element
    - testing for termination

```
Iterator i = composition.getIterator();
while (i.hasNext()) {
    Glyph g = i.next();
    do something with Glyph  g;
```

Diagram


Iterator
hasNext()
next()
PreorderIterator
hasNext()
next()
ListIterator
hasNext()
next()

Notes

- Iterators work well if we don't need to know the type of the elements being iterated over
  - E.g., send kill message to all processes in a queue
- Not a good fit for spell-checking
  - Ugly
  - Change body whenever the class hierarchy of Glyph changes

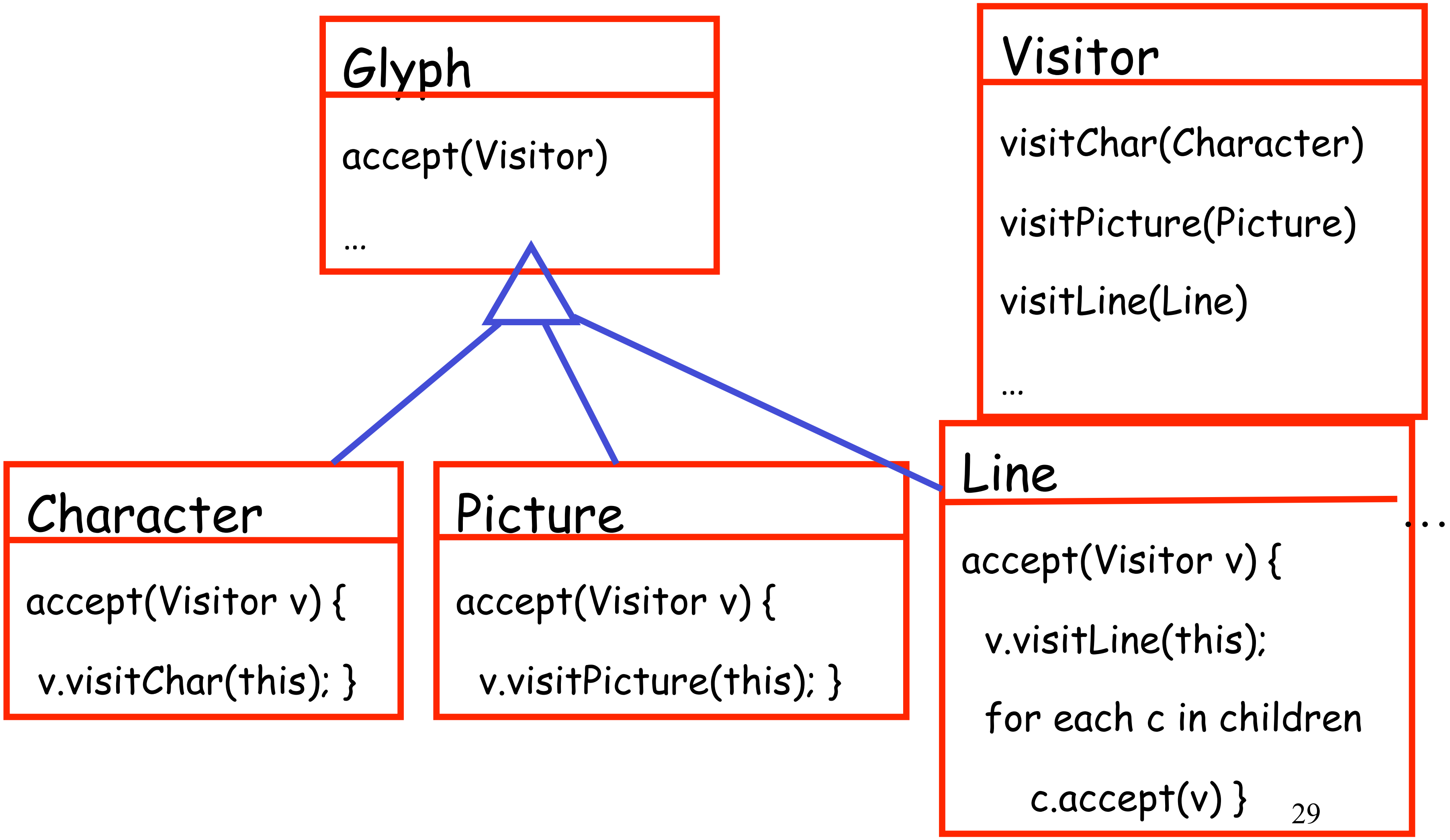
```
Iterator i = composition.getIterator();
while (i.hasNext()) {
    Glyph g = i.next();
    if (g instanceof Character) {
        // analyze the character
    } else if (g instanceof Line) {
        // prepare to analyze children
    of
        // row
    } else if (g instanceof Picture) {
        // do nothing
    } else if (...) ...
}
```

Visitors

- The visitor pattern is more general
  - Iterators provide traversal of containers
  - Visitors allow
    - Traversal
    - <u>And type-specific actions</u>

- The idea
  - Separate traversal from the action
  - Have a "do it" method for each element type
    - Can be overridden in a particular traversal

Diagram



Glyph

accept(Visitor)

…

Visitor

visitChar(Character)

visitPicture(Picture)

visitLine(Line)

…

Character

```
accept(Visitor v) {
 v.visitChar(this); }
```

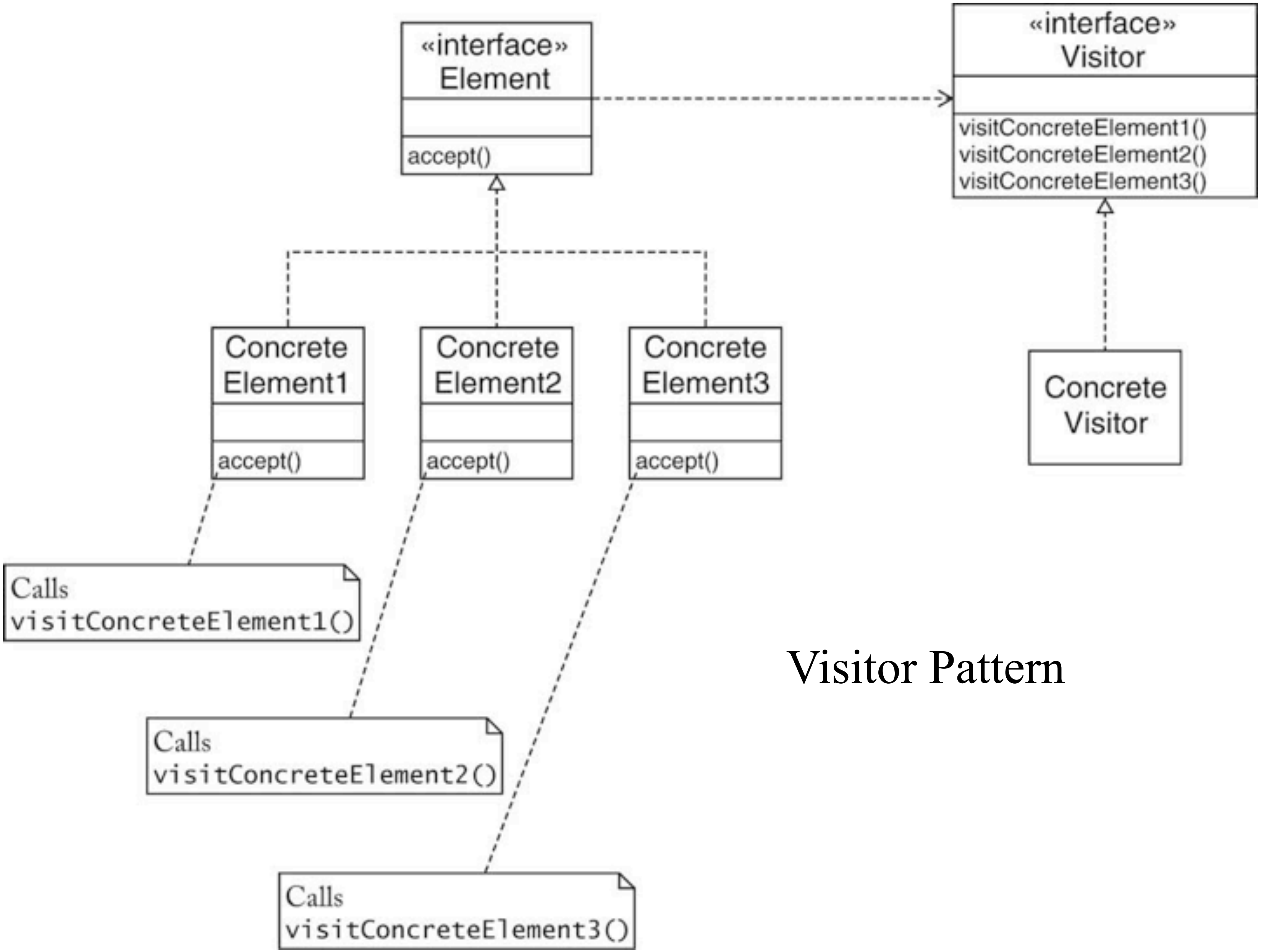
Picture

```
accept(Visitor v) {
  v.visitPicture(this); }
```

Line

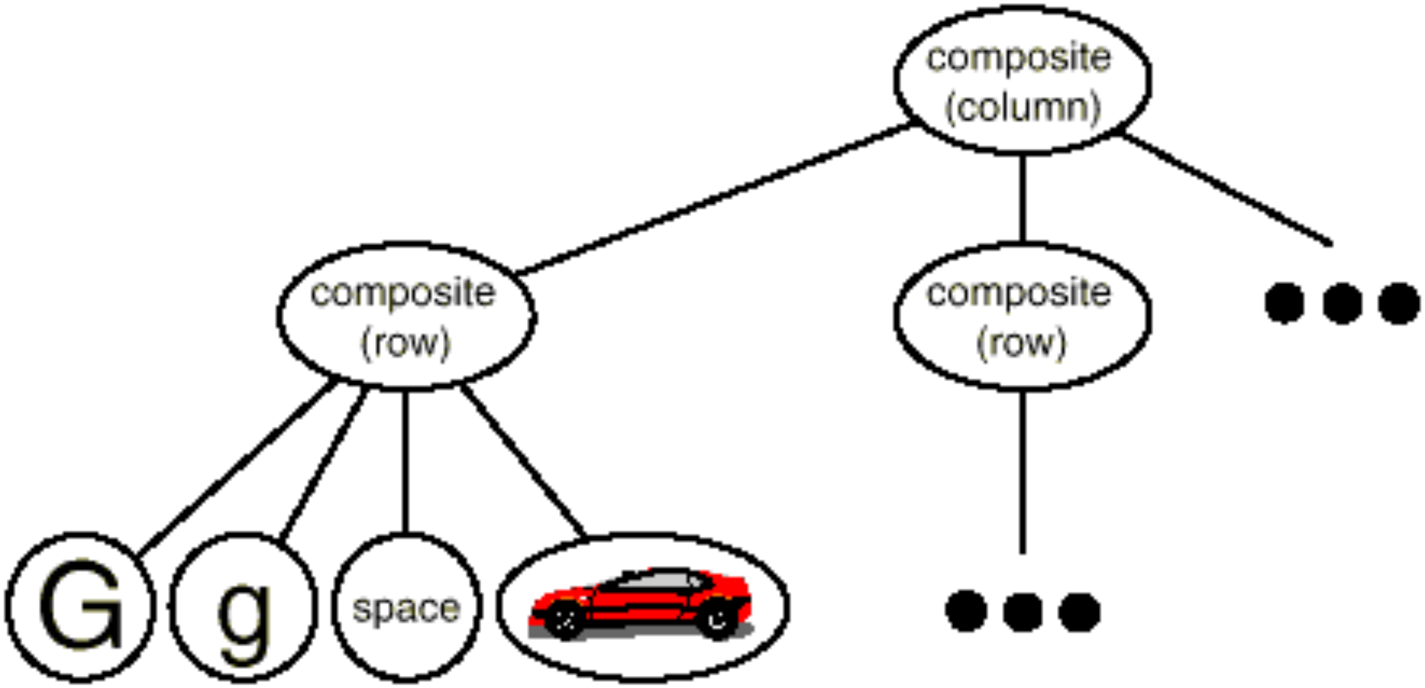
```
accept(Visitor v) {
  v.visitLine(this);
  for each c in children
      c.accept(v) }
```

…



Visitor Pattern

Logical Object Structure



Java Code

```
abstract class Glyph {
    abstract void accept(Visitor vis);
    …
}
class Character extends Glyph {
    …
    void accept(Visitor vis) {
        vis.visitChar(this);
    }
}
class Line extends Glyph {
    …
    void accept(Visitor vis) {
```

```
abstract class Visitor {
    abstract void visitChar (Character c);
    abstract void visitLine(Line l);
    abstract void visitPicture(Picture p);
    …
}
class SpellChecker extends Visitor {
    void visitChar (Character c) {
        // analyze character}
    void visitLine(Line l) {
        // process children }
    void visitPicture(Picture p) {
        // do nothing }
    …
}
```

Java Code

```
SpellChecker checker = new
    SpellChecker();
Iterator i = composition.getIterator();
while (i.hasNext()) {
    Glyph g = i.next();
    g.accept(checker);
}
```

```
abstract class Visitor {
    abstract void visitChar (Character c);
    abstract void visitLine(Line l);
    abstract void visitPicture(Picture p);
    …
}
class SpellChecker extends Visitor {
    void visitChar (Character c) {
        // analyze character}
    void visitLine(Line l) {
        // process children }
    void visitPicture(Picture p) {
        // do nothing }
```

Problem: Formatting

- A particular physical structure for a document
  - Decisions about layout
  - Must deal with e.g., line breaking
- Design issues
  - Layout is complicated
  - No best algorithm
    - Many alternatives, simple to complex

What is your design?

A First Shot:

- Add a format method to each Glyph class
- Not so good
- Problems
  - Can't modify the algorithm without modifying Glyph
  - Can't easily add new formatting algorithms

The Core Issue

- Formatting is complex
  - We don't want that complexity to pollute Glyph
  - We may want to change the formatting method

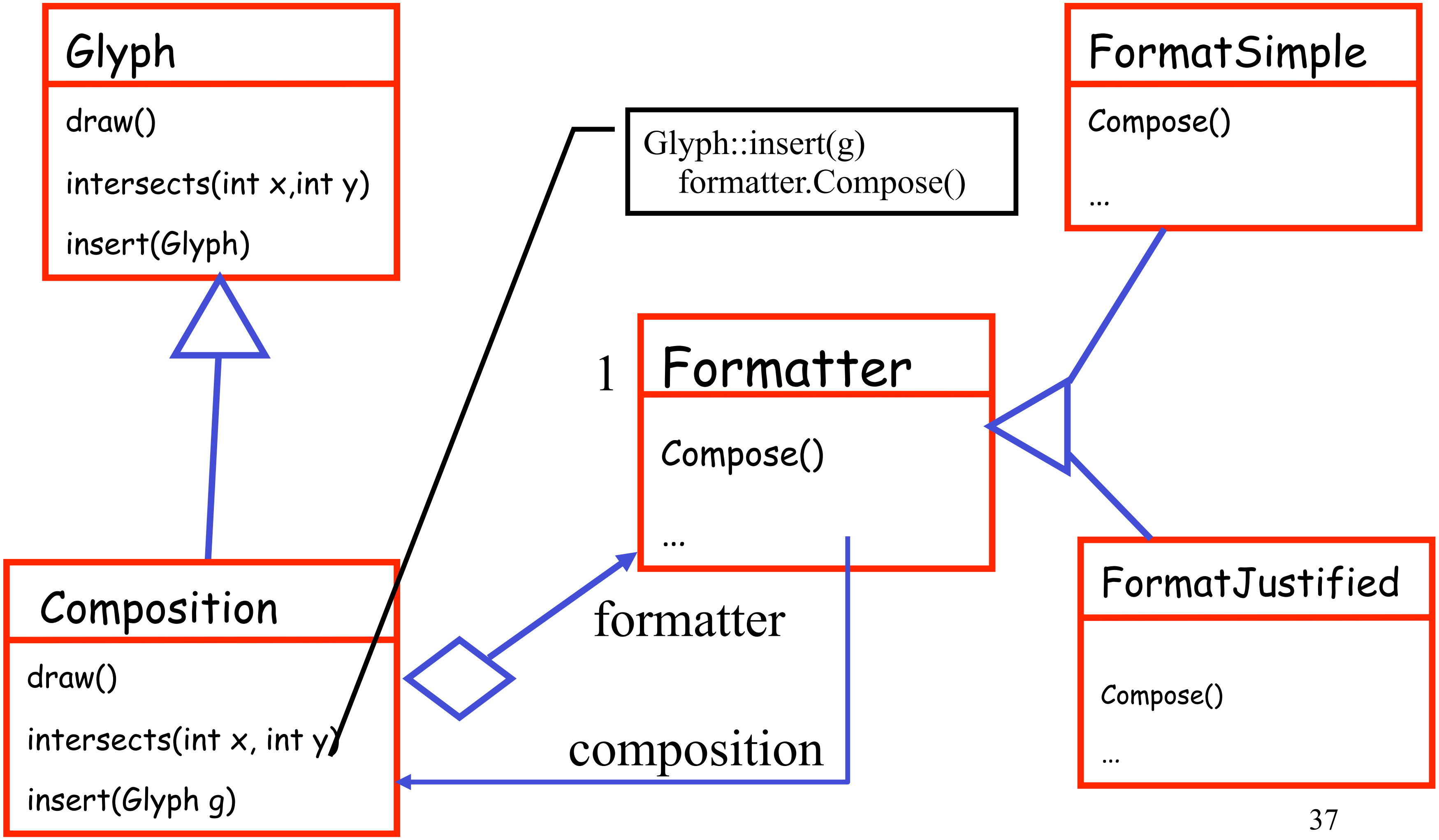
- Encapsulate formatting behind an interface
  - Each formatting algorithm an instance
  - Glyph only deals with the interface

Formatting Examples

We've settled on a way to represent the document's physical structure. Next, we need to figure out how to construct a particular physical structure, one that corresponds to a properly formatted document.

We've settled on a way to represent the document's physical structure. Next, we need to figure out how to construct a particular physical structure, one that corresponds to a properly formatted document.

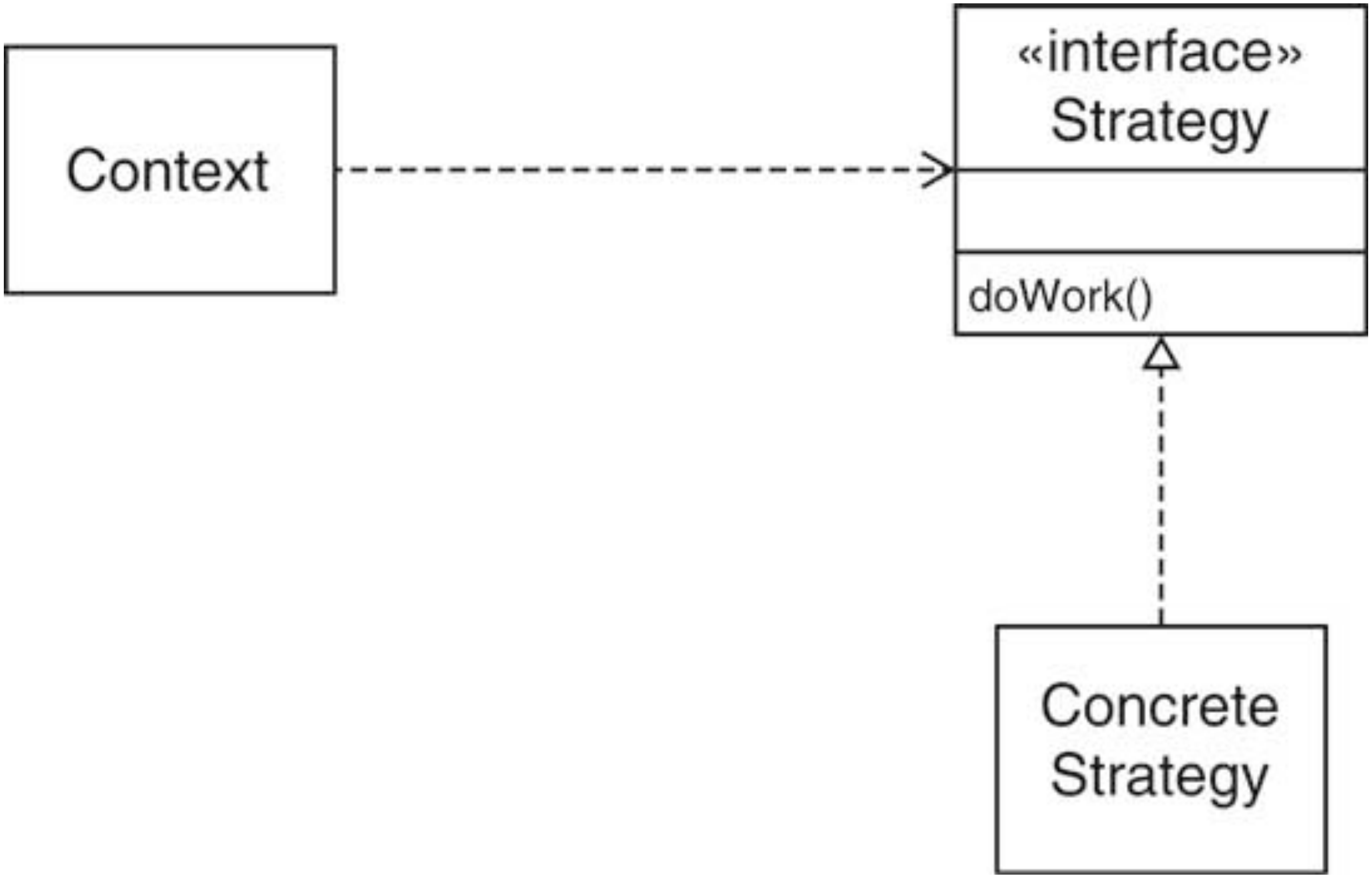
Diagram


Glyph
draw()
intersects(int x,int y)
insert(Glyph)
Glyph::insert(g)
formatter.Compose()
FormatSimple
Compose()
...
1
Formatter
Compose()
...
Composition
draw()
intersects(int x, int y)
insert(Glyph g)
formatter
composition
FormatJustified
Compose()
...

Not clear?

- This is were the pattern idea helps communication!
- Let's understand the pattern first:

Strategy Pattern



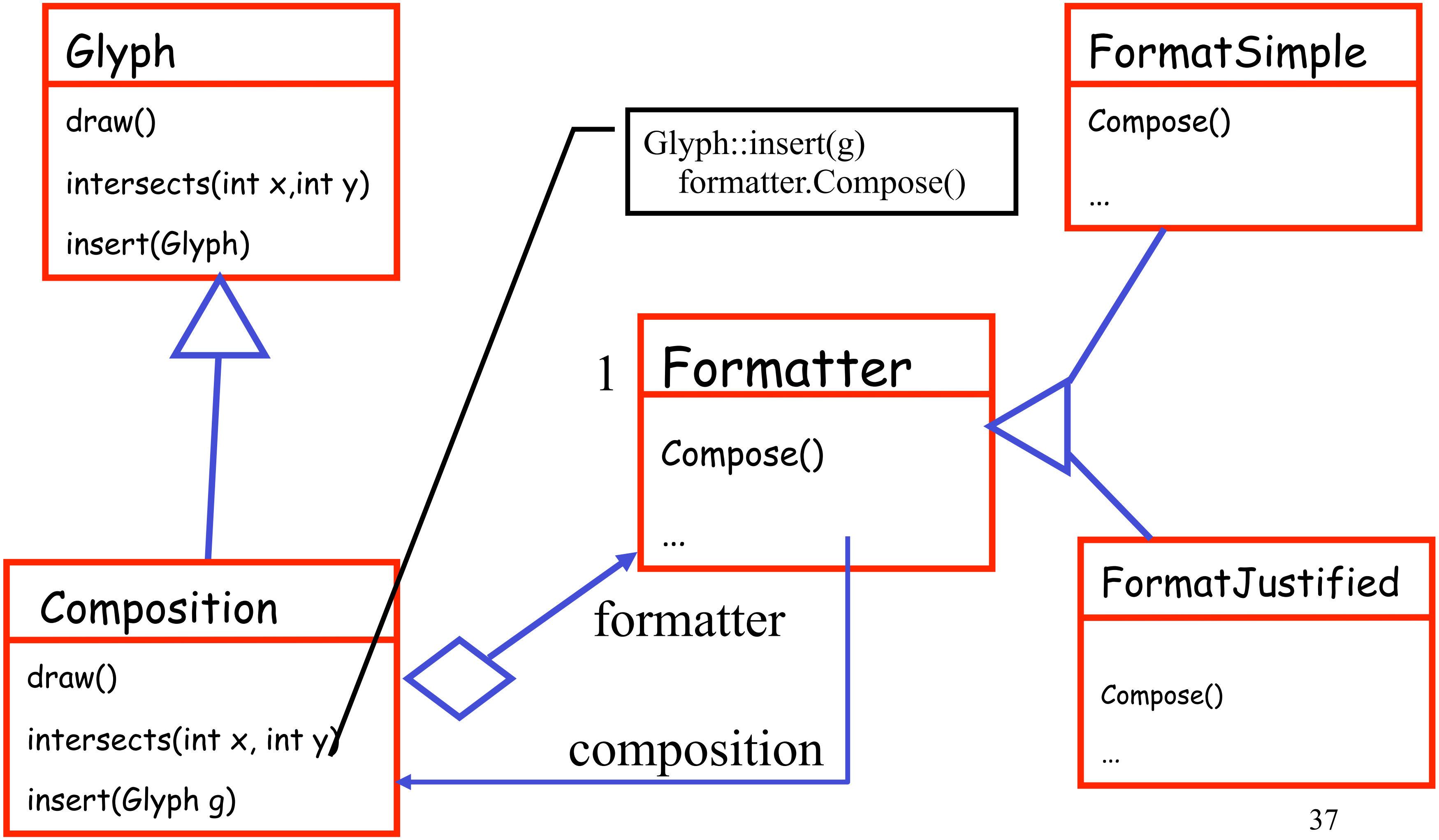
Strategies

- This is the strategy pattern
  - Isolates variations in algorithms we might use
  - Formatter is the strategy, Composition is context
- The GoF book says the Strategy design pattern should: “Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.”
- General principle

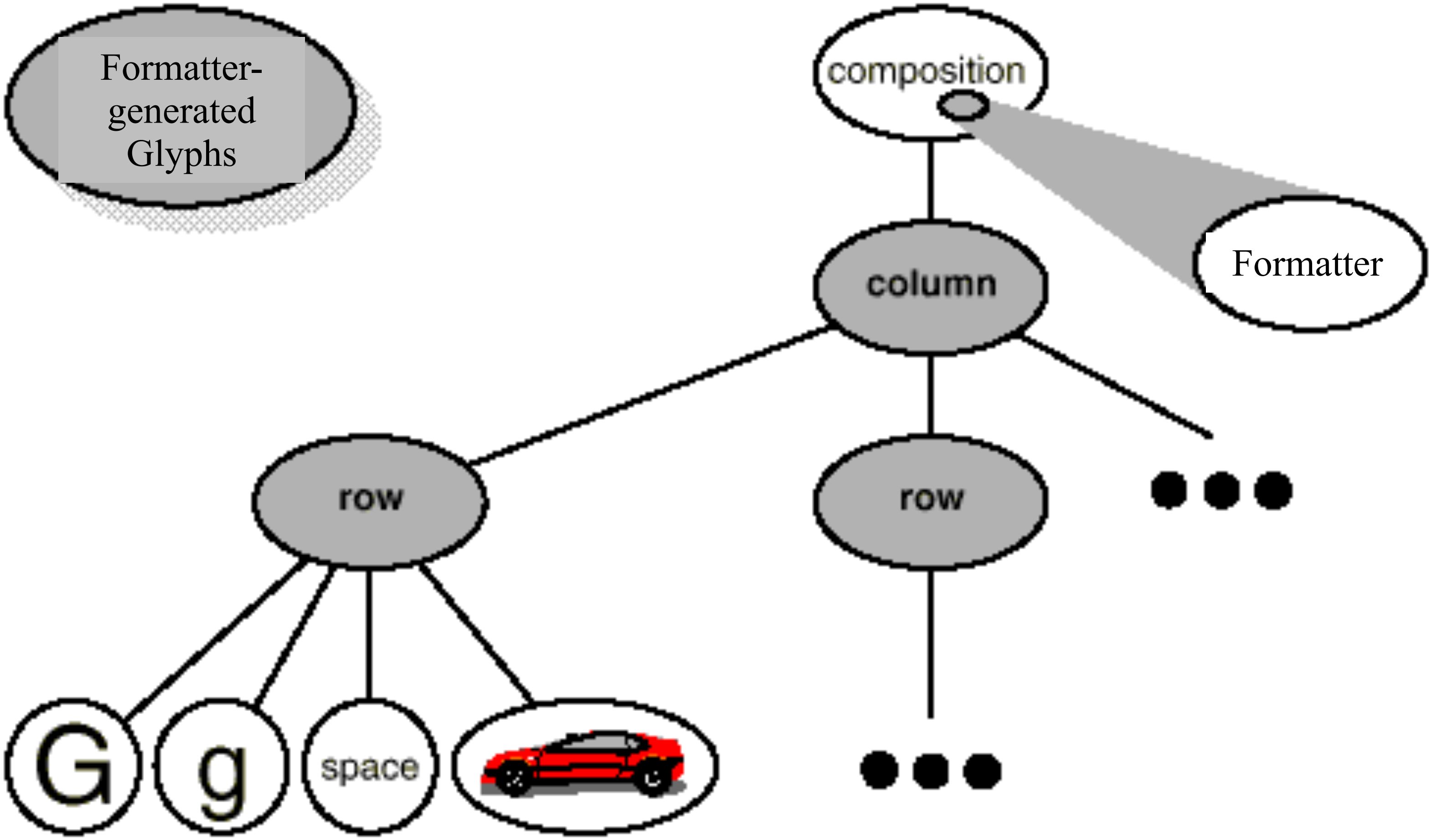
encapsulate variation

- In OO languages, this means defining abstract classes for things that are likely to change

Diagram: Strategy = Formatter



Formatter


Formatter-generated Glyphs
composition
Formatter
column
row
row
G
g
space

Design Patterns Philosophy

- Program to an interface and not to an implementation
- Encapsulate variation
- Favor object composition over inheritance

Acknowledgements

- Many slides courtesy of Rupak Majumdar
- Some from Cay Horstmann