Design Patterns

Paul Jackson

School of Informatics
University of Edinburgh

Design Patterns

"Reuse of good ideas"

A pattern is a named, well understood good solution to a common problem in context.

- Experienced designers recognise variants on recurring problems and understand how to solve them.
- They communicate their understanding by recording it in design patterns
- Such patterns then help novices avoid having to find solutions from first principles.

Patterns help novices to learn by example to behave more like experts.

Patterns: background and use

Idea comes from architecture (Christopher Alexander): e.g. **Window Place:** observe that people need comfortable places to sit, and like being near windows, so make a comfortable seating place at a window.

Similarly, software design patterns address many commonly arising technical problems in software design, particularly OO design

Patterns also used in: reengineering; project management; configuration management; etc.

Pattern catalogues: for easy reference, and to let designers talk shorthand.

A very simple recurring problem

We often want to be able to model tree-like structures of objects, where an object may be

- a thing without interesting structure, a leaf of the tree, or
- itself composed of other objects
  - which in turn might be leaves or might be composed of other objects...

And we want to implement operations on these tree structures that are uniform in behaviour, whether the trees are single objects or are composed of multiple objects.

Composite is a design pattern which describes a well-understood way of doing this.

Example situation I

A graphics library provides primitive graphics elements like lines, text strings, circles, etc.

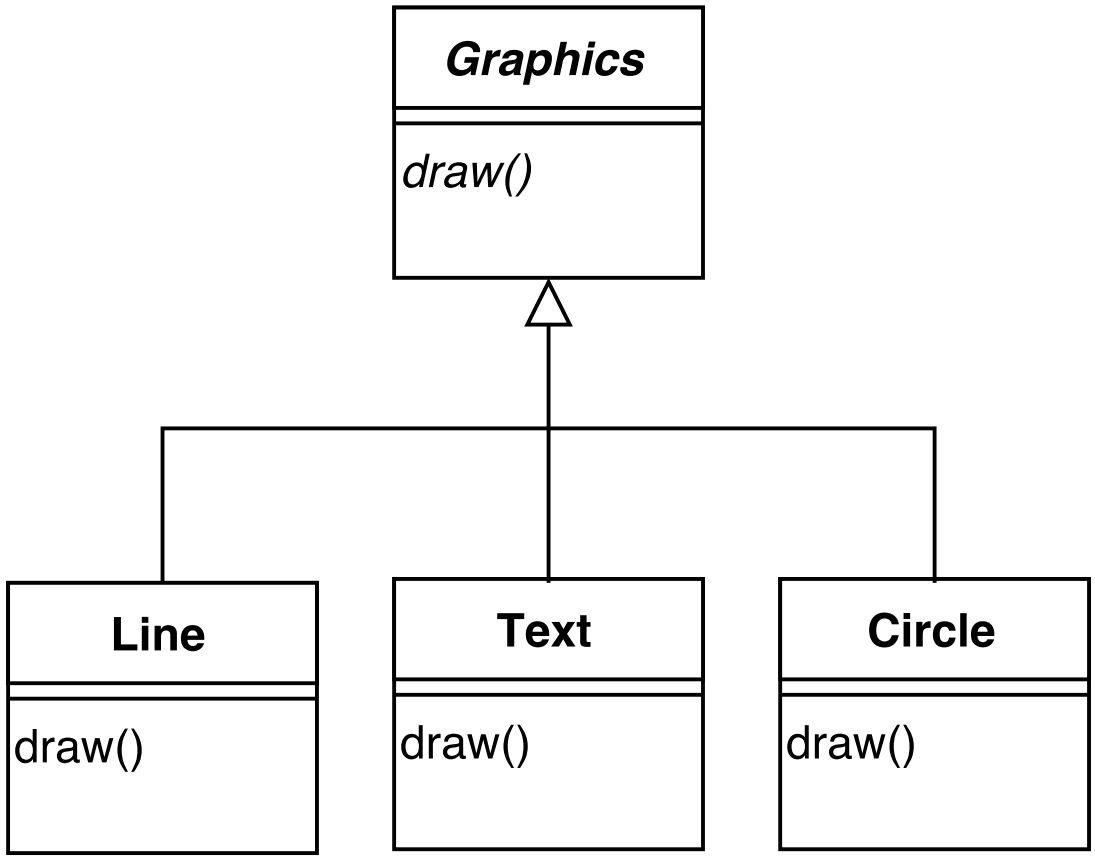
A user of the library wants support for operations on elements that are uniform across different kinds.

- E.g. *move*, *draw*, *change colour*

Makes sense to have a

- classes Line, Text, etc, for each element kind, and
- a Graphics interface or abstract base class that describes the common features of graphics elements

Example situation II



Example situation III

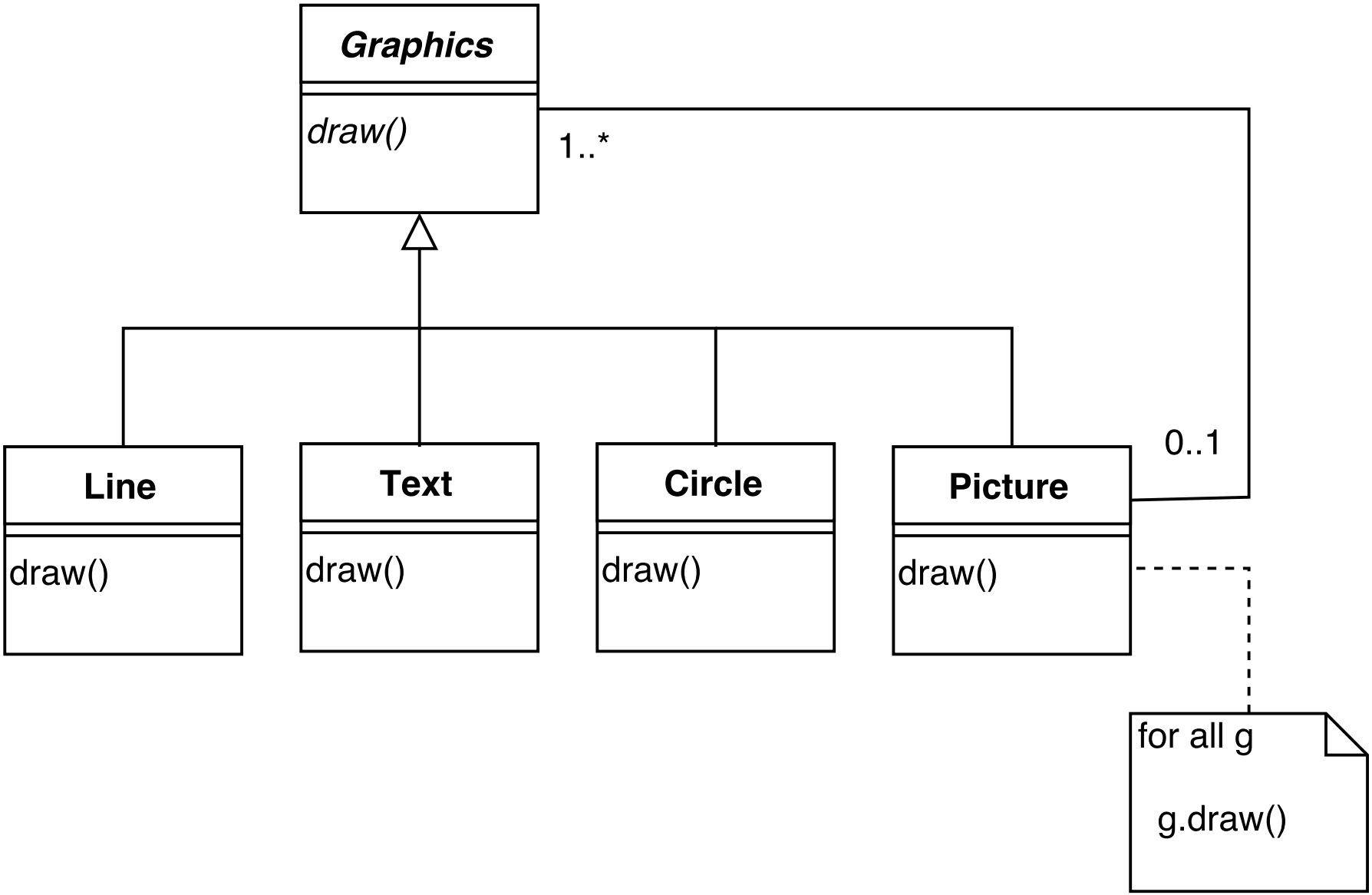
Further, the user wants to group elements together to form pictures, which can then be treated as a whole.

- E.g. user expects to be able to move a composite picture just as they move primitive elements.

And user wants to group pictures and elements together into larger pictures.

Using e.g.List<Graphics> for type of pictures is not enough.

Composite pattern



Benefits of Composite

- Can automatically have trees of any depth: don't need to do anything special to let containers (Pictures) contain other containers
- Is easy to add new kinds of Graphics subclasses, including different kinds of pictures, because user programs don't have to be altered

A drawback of Composite

Code for each operation is spread around the Graphics subclasses.

- May cause maintenance issues

One solution is to code the operation using a single method that walks Graphics object trees and has explicit conditional statements branching on the particular subclass each Graphics object belongs to.

(Use instanceof operator in Java to test class membership)

Another solution is to use the *Visitor* pattern.

- Each operation coded using a single class
- One method provided for the case of each Graphics subclass

Variations on Composite

- Might want to write some new method that walks over a whole Graphics tree. E.g. a *tree-map* method.
- To support it, need methods in Graphics like numChildren() and getChild(int i)
- Graphics then provides default implementations of these methods for the leaf subclasses.

Elements of a pattern

A pattern catalogue entry normally includes roughly:

- Name (e.g. Publisher-Subscriber)
- Aliases (e.g. Observer, Dependants)
- Context (in what circumstances can the problem arise?)
- Problem (why won't a naive approach work?)
- Solution (normally a mixture of text and models)
- Consequences (good and bad things about what happens if you use the pattern.)

Cautions on pattern use

Patterns are very useful *if you have the problem they're trying to solve*.

But they add complexity, and often e.g. performance penalties too. Exercise discretion.

You'll find the criticism that the GoF patterns in particular are "just" getting round the deficiencies of OOPLs. This is true, but misses the point.

(GoF = "Gang of Four", authors of the first major Design Patterns book)

Exercise: write a pattern language for Haskell!

Patterns: Reading

Required: Wikipedia entries (or equivalent) on Observer and Template Method design patterns. What I want you to do is to know and understand those patterns to the extent that you could use them, describe them in UML class and sequence diagrams, and explain what they achieve and how.

Suggested: Read more on design patterns, e.g.

- Stevens: Ch18.2
- Sommerville: Look up *design patterns* in index
- `http://en.wikipedia.org/wiki/Design_Patterns`