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. Without patterns, novices have 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, there are many commonly arising technical problems in software design.

Pattern catalogues: for easy reference, and to let designers talk shorthand. Pattern *languages* are a bit more...

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

A very simple recurring problem

We often want to be able to model tree-like structures of objects: 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...

We want other parts of the program to be able to interact with a single class, rather than having to understand about the structure of the tree.

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

Example situation

A graphics application has primitive graphic elements like lines, text strings, circles etc.

A client of the application interacts with these objects in much the same way: for example, it might expect to be able to instruct such objects to draw themselves, move, change colour, etc.

Makes sense to have a

- Graphics interface or an abstract base class which describes the common features of graphics elements, and
- subclasses Text, Line, etc.

Want to be able to group elements together to form pictures, which can then be treated as a whole: for example, users expect to be able to move a composite picture just as they move primitive elements.

Familiar (?) way to do this kind of task in Haskell

```
data graphicsElement =
        Line
      | Text
      | Circle
      | Picture [graphicsElement]

draw Line = -- whatever
draw Text = -- whatever
draw Circle = -- whatever
draw (Picture l) = (let x = map draw l in ())
```

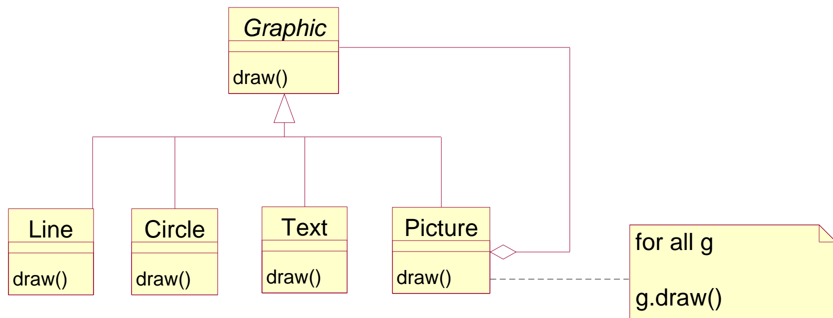
Drawbacks of the Haskell way

Clients must write recursive functions which pattern-match on the structure of the graphicsElement they have, *so all clients do in fact have to be aware of how elements of the datatype are built up.*

But this is just an example of how this mechanism does not support abstraction as well as we'd like: you can't (straightforwardly) wrap up the functions that should operate on a graphicsElement along with the datatype itself.

(Or can you? This example adapted from ML, and I don't speak Haskell...)

Composite pattern: best of both worlds



Benefits of Composite

- can automatically have trees of any depth: don't need to do anything special to let containers (Pictures) contain other containers
- clients can be kept simple: they only have to know about one class, and they don't have to recurse down the tree structure themselves
- it's easy to add new kinds of Graphics subclasses, including different kinds of pictures, because clients don't have to be altered

Drawbacks of Composite

- It's not easy to write clients which don't want to deal with composite pictures: the type system doesn't know the difference.
  (A Picture is no more different from a Line than a Circle is, from the point of view of the type checker.)

What could be done about this?

- Could use run-time checks on subtyping

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.

Exercise: write a pattern language for Haskell!

Patterns: Reading

Required: Wikipedia entries on Observer and Template Method (or equivalent: 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`