Software design, examples of simple design patterns.

Julian Bradfield

School of Informatics
University of Edinburgh

What is design?

For example: what are the classes in your system?

What is design?

For example: what are the classes in your system?

Design is the process of deciding how software will meet requirements.

Usually excludes detailed coding level.

What is design?

For example: what are the classes in your system?

Design is the process of deciding how software will meet requirements.

Usually excludes detailed coding level.

What is good design?

A quotation from Donald Schön

Designers put things together and bring new things into being, dealing in the process with many variables and constraints, some initially known and some discovered through designing. Almost always, designers' moves have consequences other than those intended for them. Designers juggle variables, reconcile conflicting values, and maneuver around constraints – a process in which, although some design products may be superior to others, there are no unique right answers.

Donald A. Schön
Educating the Reflective Practitioner
Jossey-Bass, San Francisco, 1987.

Design principles

Many of the principles of good design can be summed up as

- maximize coherence
- minimize coupling

(Why?)

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.

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.)

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 it may itself be 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. Probably there should be an interface or an abstract base class, say Graphics, which describes the common features of graphics elements, with 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.

Naive solution

Create a new class, say Container, which contains collection of Graphics elements.

Rewrite the clients so that instead of blindly sending a draw() message to a Graphics object, they

1. check whether they are dealing with a container;
2. if so, they get its collection of children and send the message to each child in turn.

Drawbacks of naive solution

Every client now has to be aware of the Container class and to do extra work to handle the fact that they might be dealing with a Container.

And can a Container contain other Containers? Not if we implement Container and Graphics as unrelated classes with the Container having a collection of Graphics objects.

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

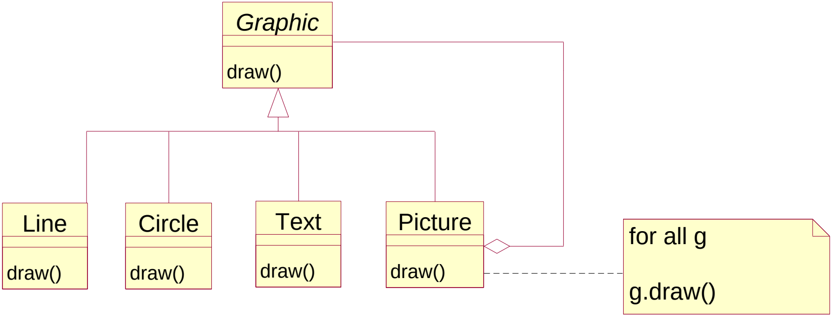
```
data graphicsElement =
        Line
      | Text
      | Circle
      | Picture [graphicsElement]

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 ML does not support abstraction as nicely as OOPLs: you can't (straightforwardly) wrap up the functions that should operate on a graphicsElement along with the datatype itself.

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 you do about this?)