Design Patterns

Perdita Stevens, University of Edinburgh

July 2010

Agenda

- What are design patterns and how can they be useful?
- Close look at examples of Gamma book patterns
- Limitations and pitfalls of using patterns
- *Beyond design: architectural patterns, risk management patterns etc.*
- *Pattern-writing and organisation-specific patterns*

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.

But in fact, this may not be the most important thing they do – we'll come back to this.

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: patterns written down in a standard format 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.)

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. Clearly 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.

So far so simple. But we also 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.

The collection of available primitive elements, the kinds of grouping available, and even what's primitive, may change relatively frequently as the program evolves.

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.

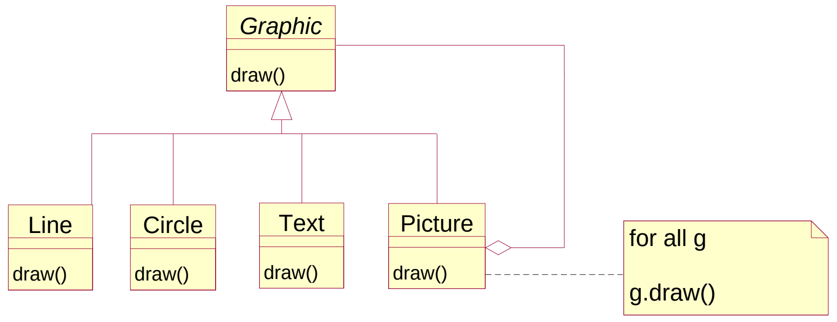
Generalising the situation

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... The collection of kinds of leaves or ways of composing them may be relatively unstable.

To avoid spreading dependencies on the precise collection throughout the program, we'd like other parts of the program to be able to interact with a single class, insulated from the structure of the tree.

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

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
- 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?)

Pattern collections

The two best known:

- Gang of Four, GoF or Gamma book: *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, Vlissides
- POSA book: *A System of Patterns: pattern oriented software architecture* by Buschmann et al.

Lots of others. I like

- *Pattern Hatching* by Vlissides: not a pattern catalogue, more an explanation of how to use the GoF book
- *Analysis Patterns: reusable object models* by Martin Fowler.

Lots of pattern-writing events e.g. PLoP, EuroPLoP.

Learning and using patterns

We'll talk through some but I rejected the idea of trying to put full information on slides.

Ideally you will need (access to) a copy of the GoF book...

Wikipedia articles on each pattern are also useful, and good for code examples, but can be down-bogging.

Creational patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral patterns

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

Some patterns jargon

(some a bit mystical for me, but you need to have heard the terms)

- forces
- QWAN
- generativity
- Rule of Three

Forces

Interesting design problems often involve potentially conflicting priorities, or forces.

E.g. you want your system to be

- easy to extend – which might push you towards including an abstract superclass;
- but also small and simple, which might push you away from that solution.

Some pattern writers, but not the GoF, make forces explicit in their pattern writing.

Forces come from the problem: no design pattern is always the right answer.

QWAN

Alexander in *The timeless way of building*

“Quality without a name”

wholeness; vitality; integrity

Generativity

Alexander again. Patterns are, or can be, generative in two senses:

- a pattern tells you how to build something
- patterns used well together can have good emergent properties

Do patterns generate architectures? It has been claimed, but I think not, at least not unless we allow "pattern" to include things that have never been written down. Most architectures will involve at most a few GoF patterns.

The Rule of Three

Patterns are supposed to have three independent successful uses before they "count"

- to avoid codifying "neat ideas"

- to get the description at the right level of abstraction

GoF book includes a "Known uses" section for this.

Ways in which patterns are useful

- To teach solutions (not actually the most important)
- As high-level vocabulary
- To practise general design skills
- As an authority to appeal to
- If a team or organisation writes its own patterns: to make "how we do things" explicit.

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.

Challenge: write a pattern language for [your favourite non-OO language].