Design principles: laughing in the face of change

Perdita Stevens

School of Informatics
University of Edinburgh

Plan

- What are we trying to achieve?
- Review: Design principles you know from Inf2C-SE
- Going further: SOLID design
- Design patterns...

What are we trying to achieve?

All good things: being able to build software faster, cheaper, better...

Overwhelmingly, though:

Resilience in the face of change.

What kind of change?

Especially: change in requirements (functional or non-functional).

Recall: this is also the main motivation for using OO in the first place. Structuring the system in terms of enduring features of the domain makes the structure likely to be stable as requirements change.

Here the focus is mostly on localising change: when requirements change, we want to be able to change the software easily. This encompasses:

- not having to change too much, e.g. minimise knock-on effects (low coupling);
- being able to identify easily what we have to change (understandable design);
- all the changes being in "nearby" code, e.g. in same class (high cohesion).

Review: Design principles you know from Inf2C-SE

Need to make sure you are confident about the meanings of:

- High cohesion
- Low coupling
- Modularisation
- Abstraction
- Encapsulation
- Separation of interface and implementation

Going further: SOLID design

(Acronym coined by Robert Martin: principles developed by many people.)

- Single responsibility principle (SRP)
- Open closed principle (OCP)
- Liskov substitution principle (LSP)
- Interface segregation principle (ISP)
- Dependency inversion principle (DIP)

Single responsibility principle (SRP)

A class should ~~do~~ be one thing, and ~~do~~ be one thing well.

What's a "thing" here? A responsibility...

...Martin says, a reason to change. Essentially about coherence: idea is that a class may offer several operations, but they should be sufficiently coherent that in practice, the kind of requirements change that makes you change one is likely to make you change all.

Open closed principle (OCP)

Bertrand Meyer:

"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"

i.e. you can modify the behaviour of a class/interface without (necessarily) modifying its code. E.g., you can create a subclass.

(For the keen: look at the different historical flavours of this principle, e.g., starting at the Wikipedia page for the principle.)

Liskov substitution principle (LSP)

Barbara Liskov.

When you inherit, feel free to add or reimplement functionality, but don't break it. Clients that think they're getting an object of the base class must still work if they're given an object of the subclass.

LSP more precisely

Let T be a type and let S be a subtype of T.

There are many formulations of LSP, not always easy to relate to the original papers! The aim is:

if we know $\phi(x)$ holds for all objects x of type T, then also, $\phi(x)$ holds for all objects x of type S

(Think of $\phi(x)$ as "my program, which is written terms of type T, behaves correctly when given x as argument" for example, and don't worry too much about the difference between "provable" and "true".)

Original formulation:

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

Interface segregation principle (ISP)

This is essentially the SRP but for interfaces.

A client should not have to rely on parts of an interface it does not need: there should be an interface that includes (almost) only the things the client actually needs.

Dependency inversion principle (DIP)

Robert Martin:

"A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions."

From principles to patterns

Any competent designer has to understand the principles that underlie good design.

However, you do not have to solve every problem from such first principles.

Design patterns capture good, but non-obvious, solutions to common problems that are tricky in the context of OO.

(Functional programming could have patterns too: they might capture things that are obvious in OO but tricky in FP.)

+ Solve the problem

- Add complexity

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 they use both classes and (e.g.) have a variable of each type, and a boolean flag to tell them which they're using. To apply an operation, they

1. check the flag to see 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;
3. if not, just sent the message to the simple Graphics object.

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.

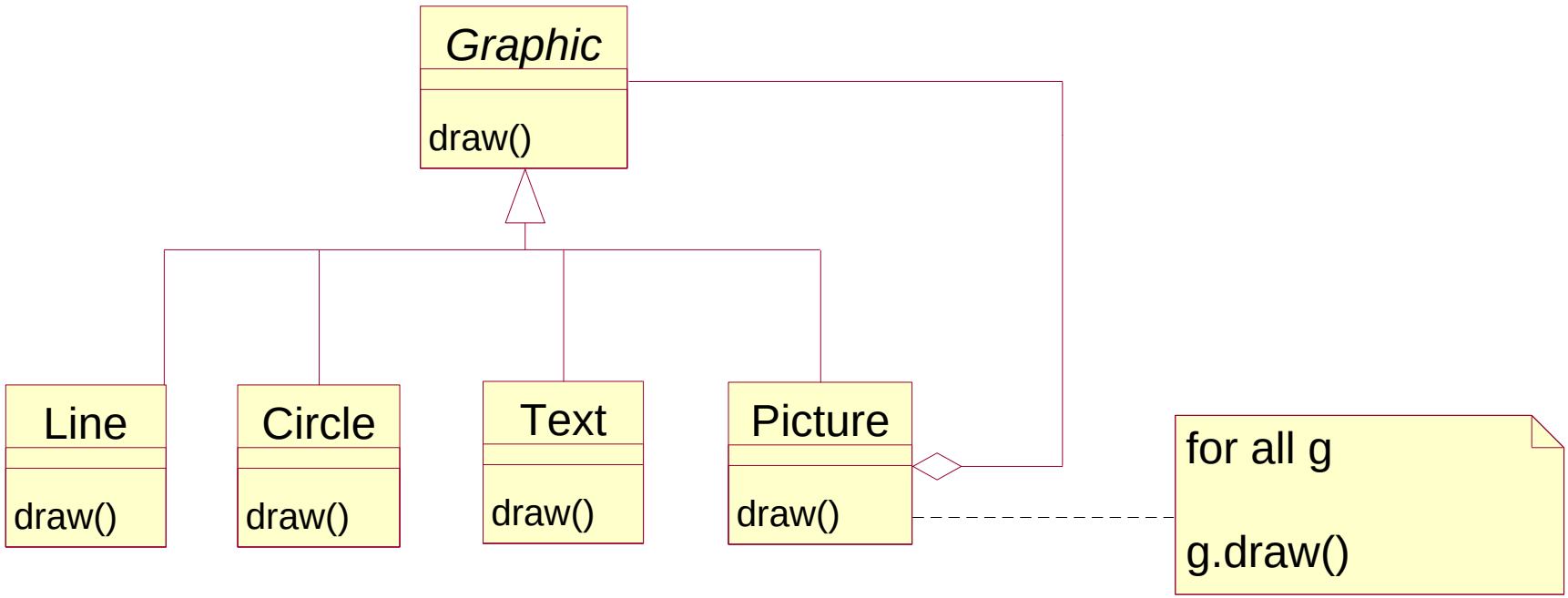
Generalising the situation

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.

To make serious use of them you'll need (access to) a copy of the GoF book; for exam purposes, slides and required reading references will suffice.

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

The specific patterns you need to know are the ones mentioned in lectures or exercise sheets: highlighted on a list in the next set of slides.

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].