Composite

It often happens that we 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.

Here's an example to explain:

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.

For any non-trivial application, users will 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 naïve thing to do is to have a new class, say Container, which contains a collection of Graphics Elements, and to rewrite the clients so that instead of blindly sending a draw() message to a Graphics object, they first check whether they are dealing with a container or not. If they are dealing with a container they get its collection of children and send the message to each child in turn.

This way of doing it has several obvious disadvantages though. 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.

Fortunately as you have ML experience, you're already familiar with a design idiom that incorporates the essence of a good solution to this problem: the use of recursive datatypes with recursive functions on them. In ML it would be quite natural to write:

(Simplifying, e.g. to leave out the arguments that specify *which* line we have):

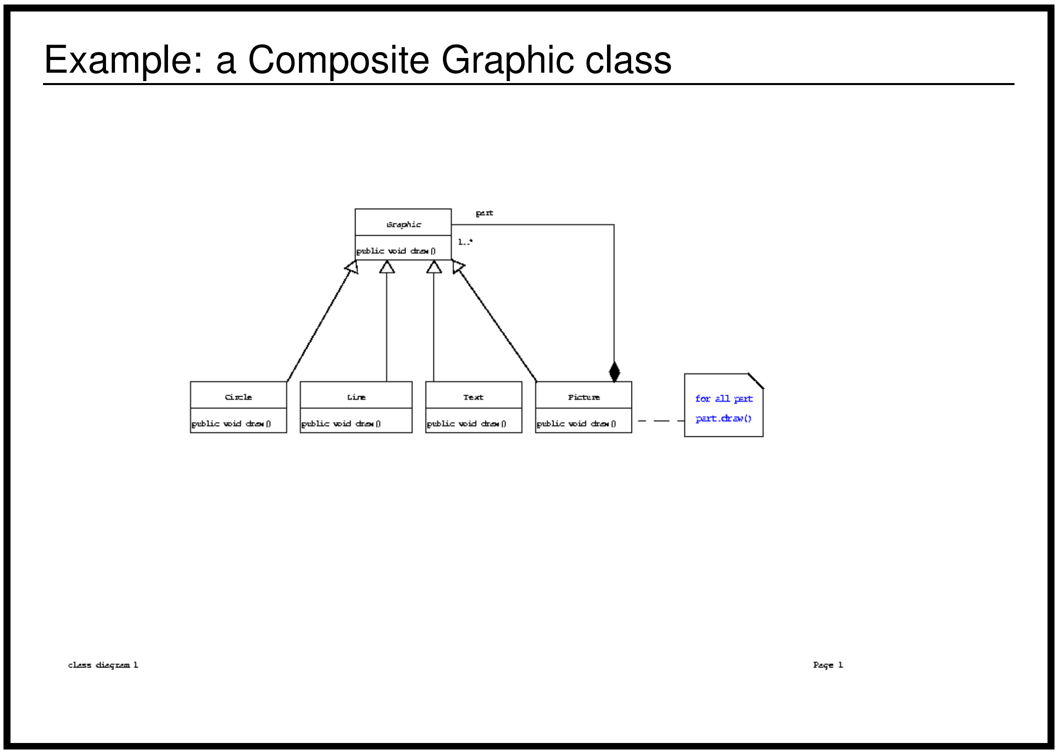
Familiar way to do this kind of task in ML

```
datatype graphicsElement = Line
                         | Text
                         | Circle
                         | Picture of graphicsElement list

fun draw Line = (* whatever *)
  | draw Text = (* whatever *)
  | draw Circle = (* whatever *)
  | draw Picture l = (map draw l; ())
```

Slide 1

This has lots of disadvantages, of course: clients of the graphicsElement datatype will interact with it by themselves writing 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 object oriented languages do: you can't (straightforwardly) wrap up the functions that should operate on a graphicsElement along with the datatype itself. Given the basic idea, that a graphicsElement might be a Line, a Text, a Circle, or a Picture *which itself consists of graphicsElements* we can combine the advantages of the two approaches and build a class structure like this:



Slide 2

Here too, a Graphic object could be either a basic Text, Circle or Line object, or it could be a Picture object which contains some Graphics objects. Because these are classes, we can wrap up the behaviour too. The diagram shows that Line, Circle and Text each implement the operation draw() in some unspecified way, and that Picture implements draw() by sending the draw() method to all of its children.

Now a client can interact with an object knowing only that it has class Graphic – without knowing, more specifically, whether it's a Line or a Picture or something else – and can send it the draw() method anyway. The Graphic object will Do The Right Thing, whatever that happens to be.

Some good consequences are:

- we can automatically have trees of any depth: we 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 Graphic subclasses, including different kinds of pictures, because clients don't have to be altered

A sometimes bad consequence is that 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?)

Once you understand this way of relating classes for these purposes, there are many situations in which it can be used. This is an example of a standard way of doing things in an object oriented design that expert designers know. The idea of a *design pattern* is to teach such expertise efficiently. Each pattern is small enough to be taken in easily, and specific enough for the learner to recognise application areas. The full description of this pattern in Gamma et al consists of these sections:

COMPOSITE Object Stuctural
Intent
Motivation
Applicability
Structure
Participants
Collaborations
Consequences
Implementation
Sample Code
Known Uses
Related Patterns

Different pattern structures are used in other books and the details of what goes where aren't important in themselves. What's important is that once you understand the structures used in a particular book, it's quick and easy to find the information you're looking for.

Chapter 18 of *Using UML* gives a slightly fuller description of design patterns. For now you just need to read Section 18.2, pages 219–223; we'll cover the rest of the chapter later in the course so you might prefer to read the whole chapter now.