3F6 - Software Engineering and Design

Handout 6

Design Patterns (I)

With Markup

Ed Rosten

Contents

Copies of these notes plus additional materials relating to this course can be found at:
`http://mi.eng.cam.ac.uk/~er258/teaching`

Design Patterns

Software systems can be very large and very complex. However, we often find the same architectural structures occurring repeatedly (with subtle variations), created in response to commonly recurring problems. These solutions can be identified and recorded as *design patterns*.

This course will look at a few of the most common design patterns with two aims:

1. To explain how to use these specific patterns in software designs and in communicating about software that uses them.
2. To introduce the language of design patterns and illustrate the more general benefits from thinking about software construction in this way.

A more comprehensive set can be found in

Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma et al, Addison-Wesley, 1998

which describes 23 design patterns in detail.

The Structure of Patterns

Each pattern is described using a standard format:

Problem: outline some specific functionality that we would like our software to provide.

Bad solutions: explore some ways of providing this functionality and discuss their limitations.

Good Solution: present a preferred solution based on a design pattern.

Code Example: an example of what the design solution looks like when coded in C++.

Design Pattern: discuss the design pattern which is the general principle underlying the good solution and its applicability to other situations. This will be accompanied by UML showing the generic design pattern.

Disadvantages: discuss the shortcomings the design pattern and why you might not want to use it for certain cases.

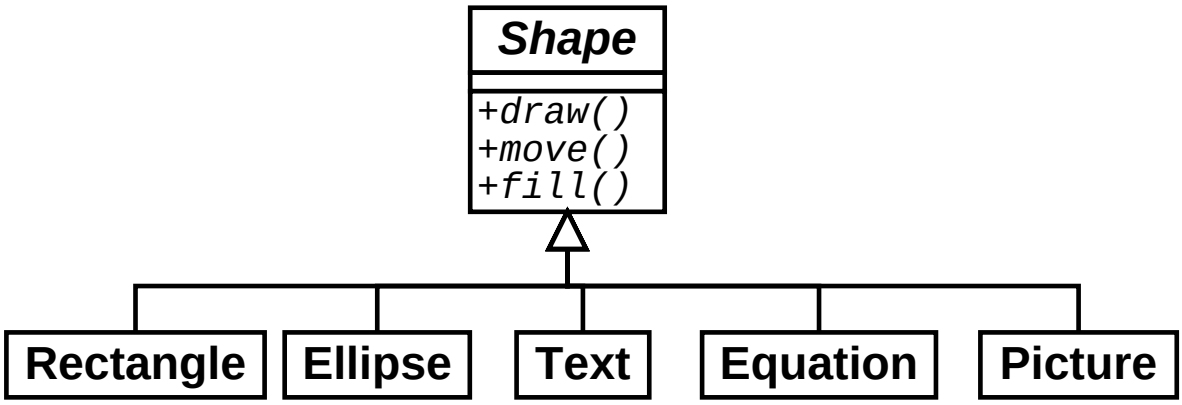
To illustrate each of these aspects, the Drawing Editor example will be used.

This is the approach used to motivate and explain class derivation and polymorphism (virtual functions). They seem obvious now because they are built in to languages like C++ but earlier generations of C programmers might have regarded them as design patterns.

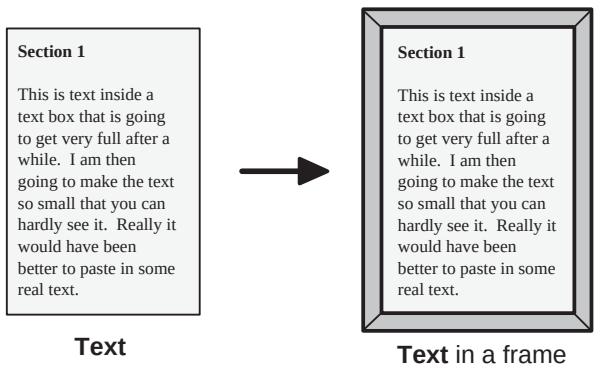
Decorator Pattern

Problem

Suppose our drawing editor allows us to include many sorts of shapes including rectangles, ellipses, text, equations, pictures etc.

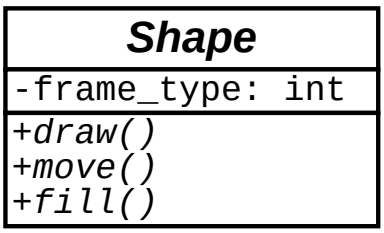


Now we want to introduce a facility into the editor to allow frames to be added to arbitrary objects. For example we might want to put a picture frame around an image, or we might want to frame an equation or some text in a simple box.



Solution 1

Since we want to be able to add frames to objects of all types, we could add an attribute into the `Shape` class to specify the type of frame the object has (if any).

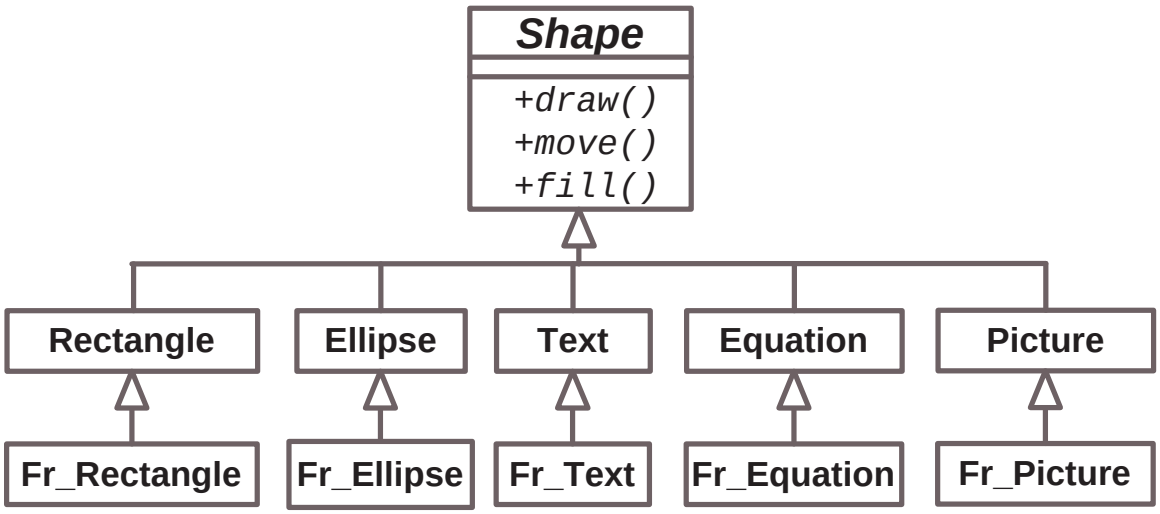


Pros:	-	simple and adequate for case where we only want to add one special attribute to shapes
Cons:	-	wastes storage since all objects contain all attribute data
	-	the code itself will become clumsy since, for example, the `draw` method will need to have a case switch for each of the possible frame types

```
void Shape::draw() {
   switch(frame_type) {
     case NONE:
           break;
     case SIMPLE_FRAME:
           draw_simple_frame();
           break;
     ...
   }
}
void Text::draw() {
   Shape::draw();
   // render text
}
```

Solution 2

An alternative would be to derive new classes such as Fr_Rectangle, Fr_Picture, Fr_Equation etc. to provide framed versions of each shape class:

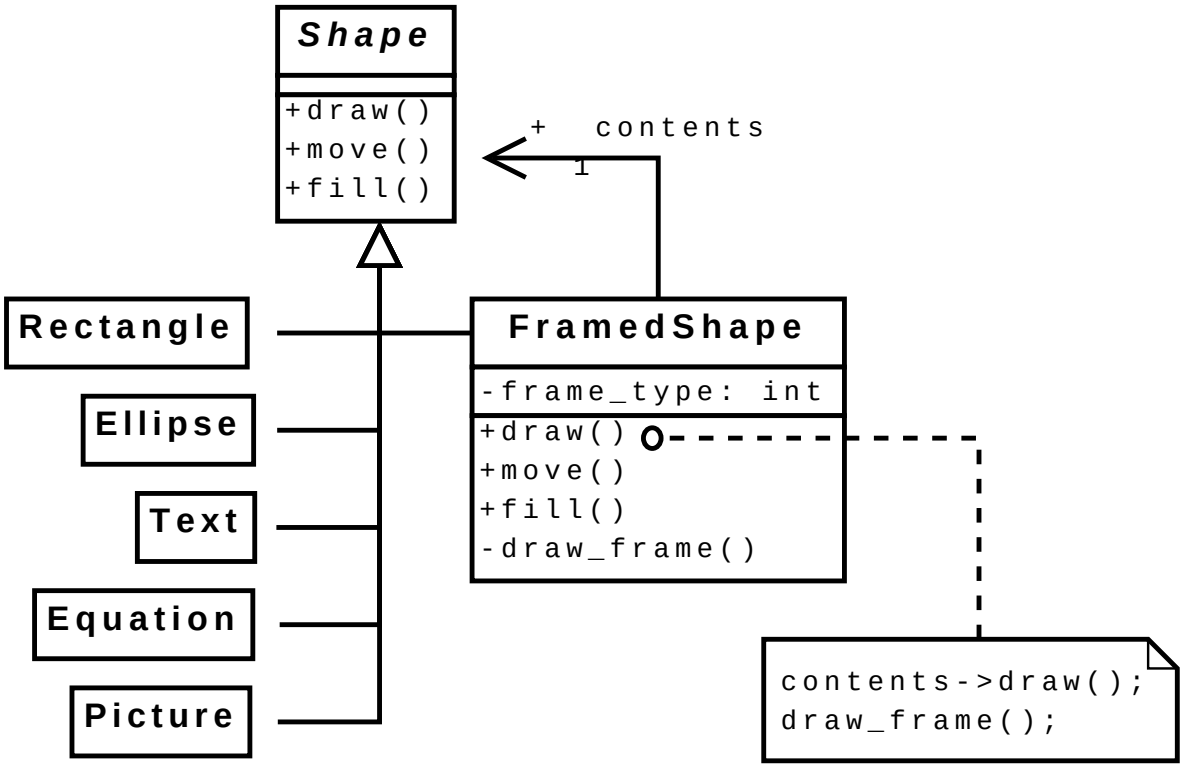


Pros:	-	framing can be restricted to particular shapes
	-	efficient use of storage since frame data is only allocated when actually needed
Cons:	-	huge proliferation in classes
	-	hard to turn decorations on and off at run-time

Note that the framed versions will inherit exactly the same interface as their parents. This is important since it is essential that any client using any shape object sees an identical interface.

Good Solution

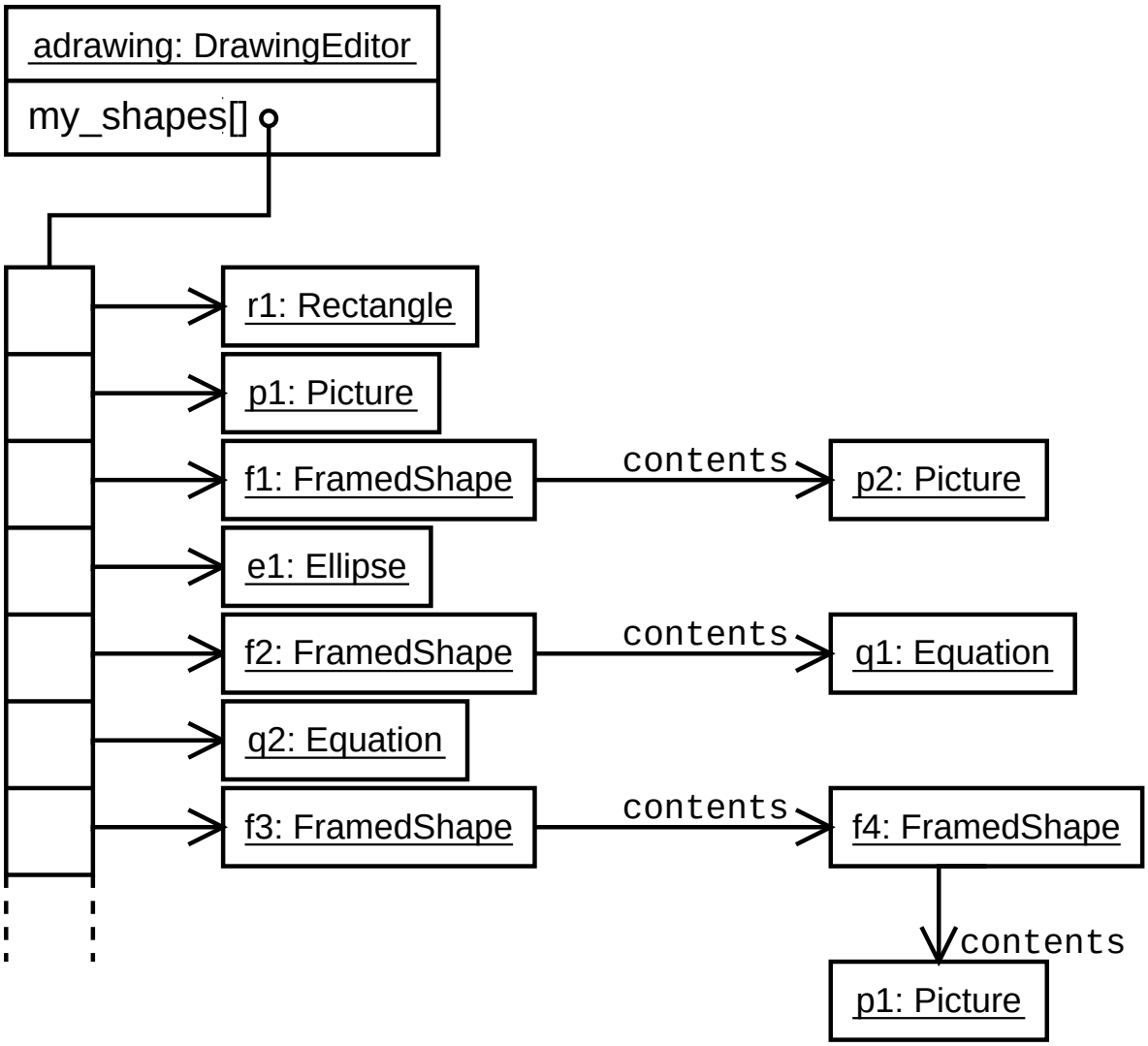
A much better way to solve this problem is to add a single new subclass of `Shape` called `FramedShape`. Each FramedShape will have a pointer to a `Shape` object which is the shape contained in the frame.



The addition of this extra class allows us to frame any kind of shape, simply by creating a `FramedShape` object and making its `contents` point to the `Shape` object that we want to frame.

We can even create a frame around a `FramedShape` (see example in the object diagram below!).

This software architecture will give rise to run-time structures similar to that shown in the following object diagram:



Note that the picture `p1` is embedded in two frames.

Code Example

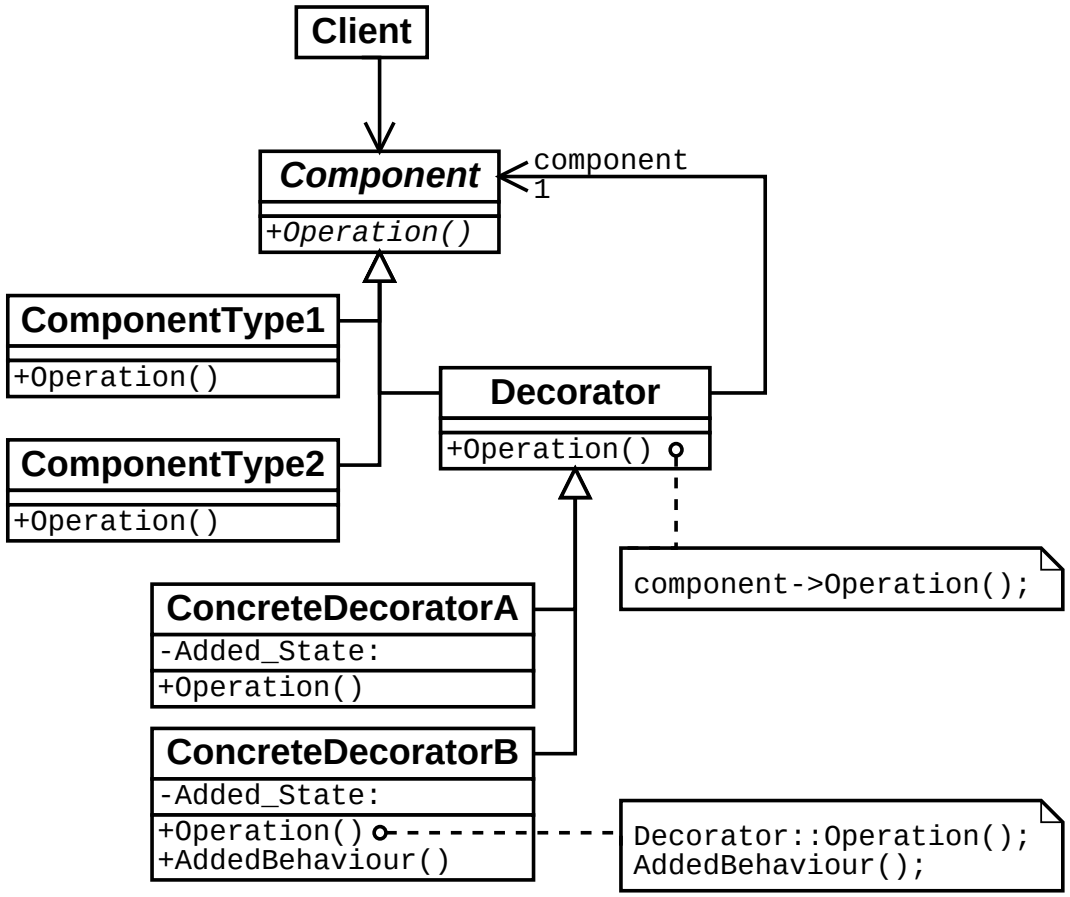
```
class Shape {
 public:
  virtual void draw()=0;
  virtual void move(int dx, int dy)=0;
  virtual void fill(Colour c)=0;
};
class FramedShape : public Shape {
 public:
  virtual void draw();
  virtual void move(int dx, int dy);
  virtual void fill(Colour c);
 private:
  void draw_frame();
  Shape *contents;
  int frame_type;
};

void FramedShape::draw(){
  contents->draw();  // draw the contents of the frame
  draw_frame();  // draw the frame
}
```

The Decorator Design Pattern

Provides a way of adding optional functionality (ie *decoration*) to all classes in a hierarchy without changing the code for either the base class or any of the subclasses.

Using this pattern, multiple decorations can be applied to an object (e.g. we can add a picture frame and scrollbars (in either order) to a picture in the drawing editor). If there are several different kinds of decoration that we want to be able to use,

we can derive a number of classes from the `Decorator` class to handle these separate kinds of added functionality.



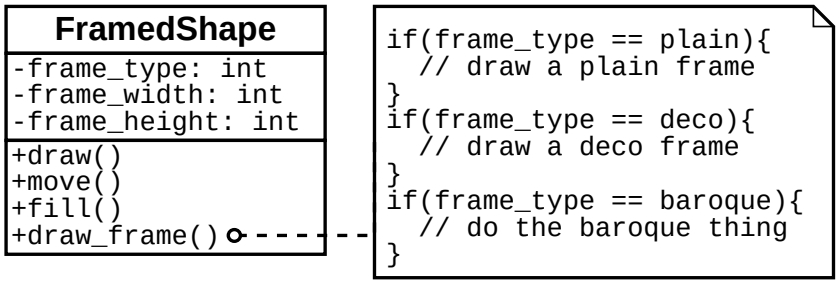
Disadvantages

If there are not too many kinds of added functionality and they appear fairly commonly, it may be more convenient to use solution 1 (above). The decorator pattern can make it hard to resolve the identity of the objects we are dealing with since the decorator is a distinct object from the component it decorates. In a running system, this can result in long chains of small objects that point to each other, making the software hard to debug.

State Pattern

Problem

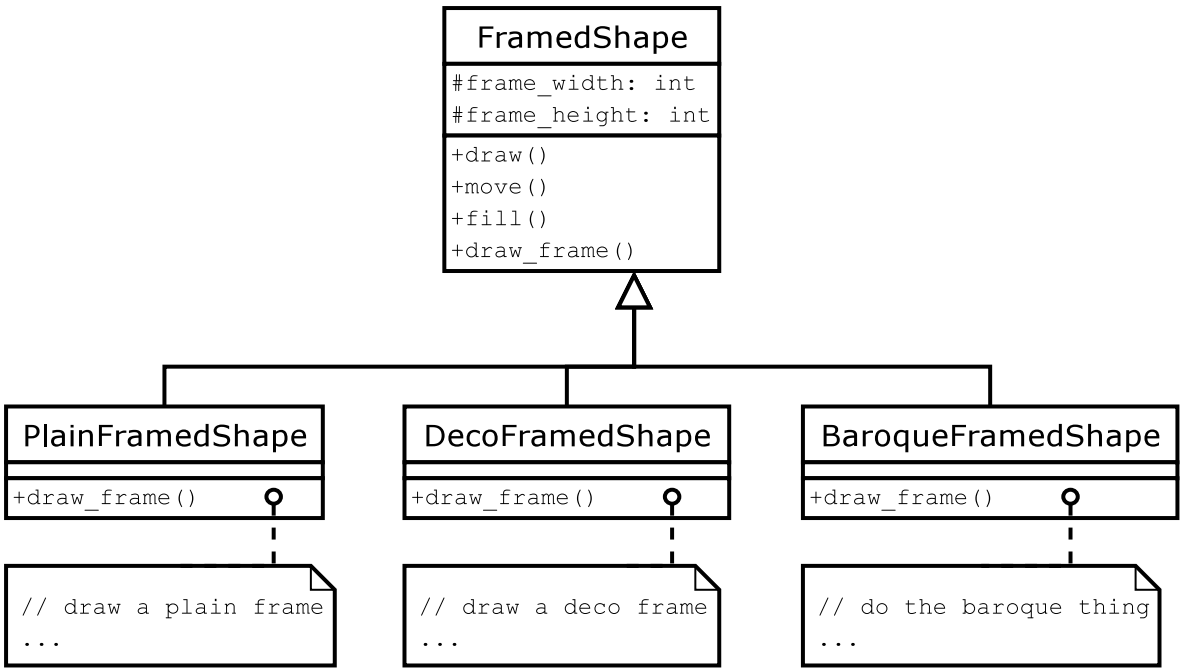
Suppose that the number of frame types available for framing shapes in the drawing editor becomes very large. The `draw_frame()` function in `FramedShape` will become very complicated and difficult to maintain (cf Solution 1 for the Decorator Pattern).



Solution 1

Introduce a class hierarchy to derive the different kinds of frame required from `FramedShape`. The attribute information that is common to all types of frame (i.e. `frame_width` and `frame_height`) continues to be stored in the base class. In this diagram it is shown as protected (the "#" symbol) which means that it is available to the subclasses of `FramedShape` but private with respect to the outside world.

Pros:	-	enables new frames to be introduced without disturbing existing code
	-	easy to maintain code
Cons:	-	very difficult to change frames at runtime



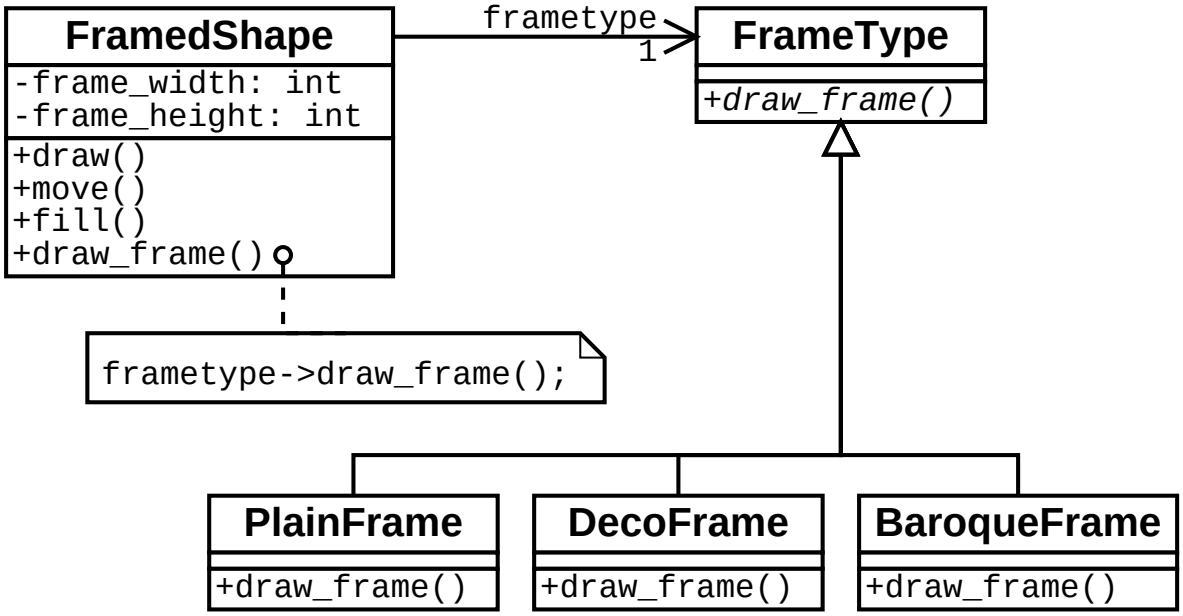
Consider trying to convert a deco frame into a baroque frame:

1. Construct a `BaroqueFramedShape` object.
2. Copy all the state information (i.e. `frame_width` and `frame_height`)fr the `DecoFramedShape` object into the `BaroqueFramedShape` object.
3. Find every pointer in the entire system that points to the `DecoFramedShape` object and change it to point to the `BaroqueFramed` object instead.
4. Delete the `DecoFramedShape` object.

Step 3 above is typically very difficult to achieve.

Good Solution

A better way to achieve this is to create a stand-alone class hierarchy derived from `FrameType` to represent frame types and to alter `FramedShape` so that it has-a `FrameType`. In this way, we can simply destroy one kind of `FrameType` object and replace it with another when we want to change frame types without having to replace the `FramedShape` object.



Note that the width and height must be passed from FramedShape to FrameType via the call

```
frametype->draw_frame(frame_width,framed_height)
```

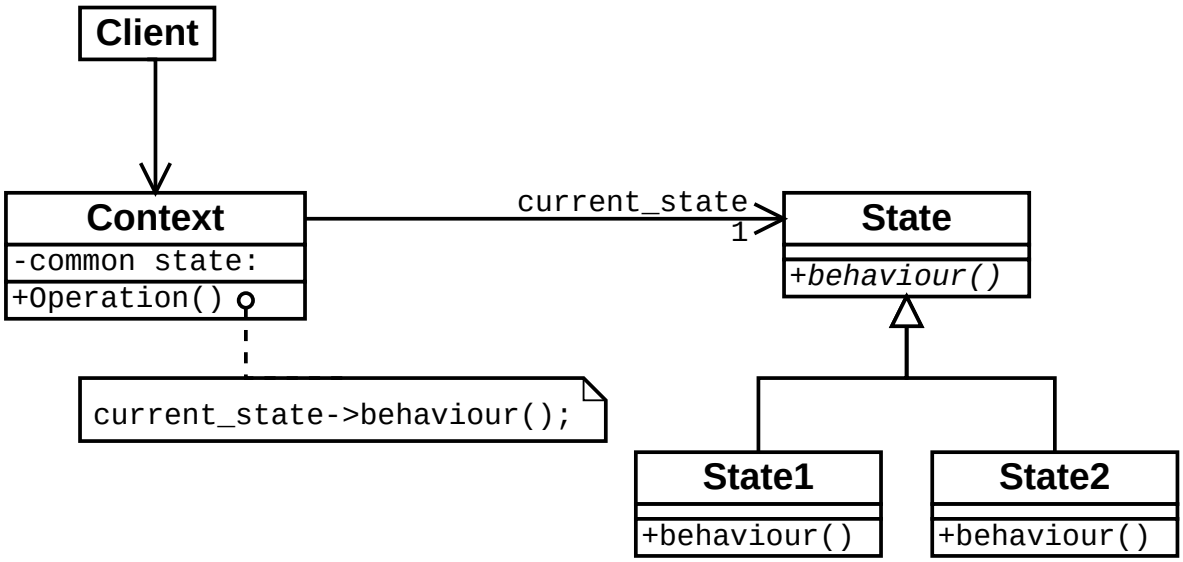
this is not shown in the above. Also, these variables become private again rather than protected

Code Example

```
class FrameType {
 public:
  virtual void draw_frame(int width, int height)=0;
};

class PlainFrame : public FrameType {
 public:
  virtual void draw_frame(int width, int height);
}

class FramedShape : public Shape {
 public:
  // all the other stuff
  void draw_frame();
 private:
  FrameType* frametype;
  int frame_width;
  int frame_height;
};

void FramedShape::draw_frame() {

  frametype->draw_frame(frame_width, frame_height);

}
```

The State Design Pattern

The state pattern is useful for dynamically changing some aspect of the behaviour of an object without destroying the object itself and replacing it with a new one.



Disadvantages

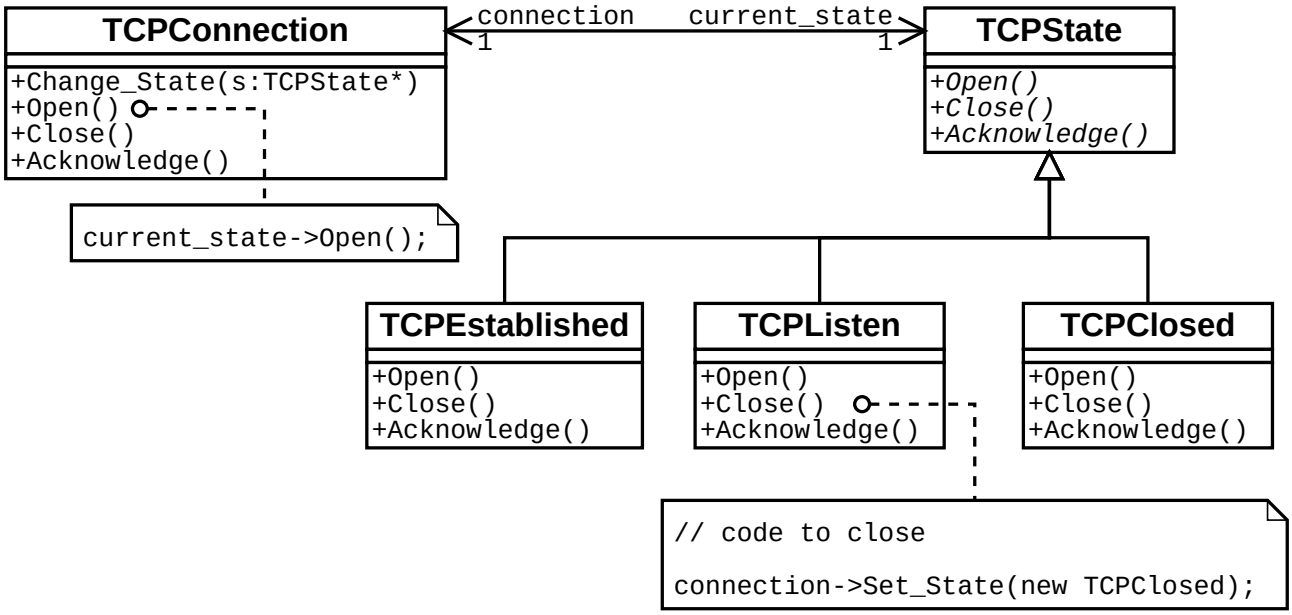
By extracting state dependent behaviour into a separate class, it becomes impossible to override this behaviour by deriving from the `Context` class. This means that the behaviour becomes fixed for users of the `Context` class.

Another example of using the State Pattern

In the drawing editor example, the state is switched explicitly by the user.

The state pattern is also useful for implementing state machines where the behaviour changes at each state transition.

For example, software to manage (e.g. to manage TCP connections):



Note that in this example the state objects determine the next state themselves and explicitly invoke transitions to the next state.

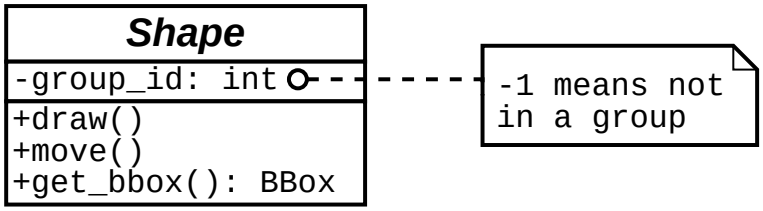
Composite Pattern

Problem

We want our drawing editor to support grouping and ungrouping operations so that a number of shapes can be collected together and treated as a single entity.

Solution 1

We could add a group member field into `Shape` to indicate which group each shape belongs to (using the number -1 to indicate that the object is not in any group).



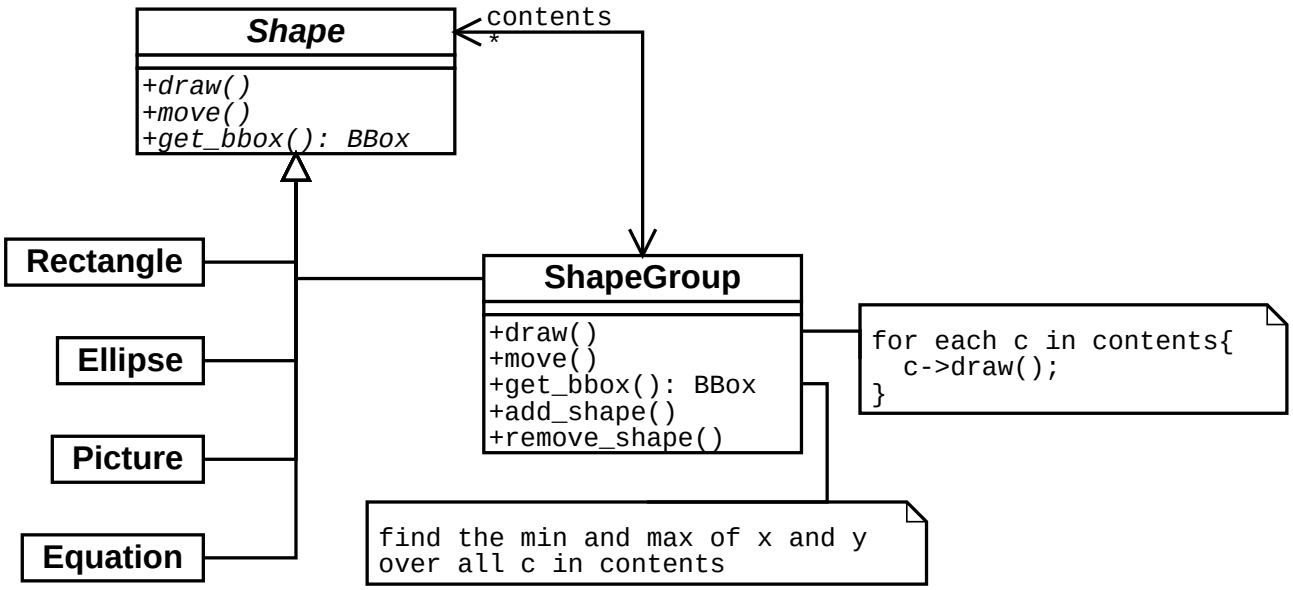
Pros:	-	simple
Cons:	-	cannot support nested groups

Good Solution

A better approach is to introduce a new class `ShapeGroup` to manage a group of shapes. This new class is a subclass of `Shape` and so it preserves the standard `Shape` class interface.

The `ShapeGroup` class provides a means by which several shapes can be grouped together into a single entity which behaves in the same way as a single shape.

Most of the `ShapeGroup` class methods are implemented simply by calling the same function for each of its constituent shapes. Computing the bounding box is a only little more complicated and can be done in a similar manner.

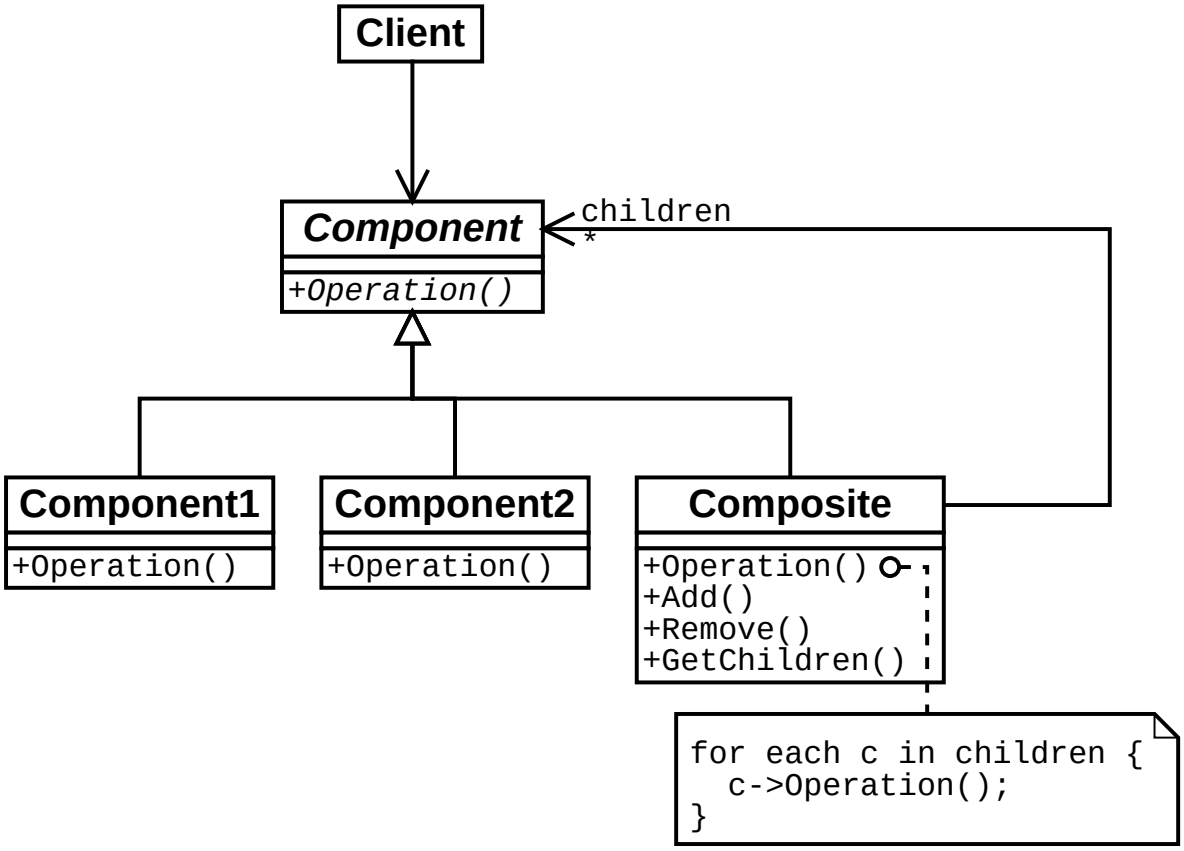


Code Example

```
class ShapeGroup : public Shape {
 public:
  virtual void draw();
  virtual void move(int dx, int dy);

  void add_shape(Shape *s);
  void remove_shape (Shape* s);

 private:
  list<Shape*> contents;
};

void ShapeGroup::draw(){
  for(list<Shape*>::iterator iter = contents.begin();
      iter != contents.end(); iter++) {
    (*iter)->draw();
  }
}

void ShapeGroup::add_shape(Shape *s){


  contents.push_back(s);


}
```

The Composite Design Pattern

Provides a means of grouping together several objects of type T such that the grouped object is also of type T.



Disadvantages

The composite pattern is very powerful, but can sometimes be too general. For example, it is difficult to restrict the objects which can be included in the composite group.

Since the `Composite` class usually has to be extended to provide access to the individual group members (as shown), client code must be able to distinguish between composite objects and non-composite objects.