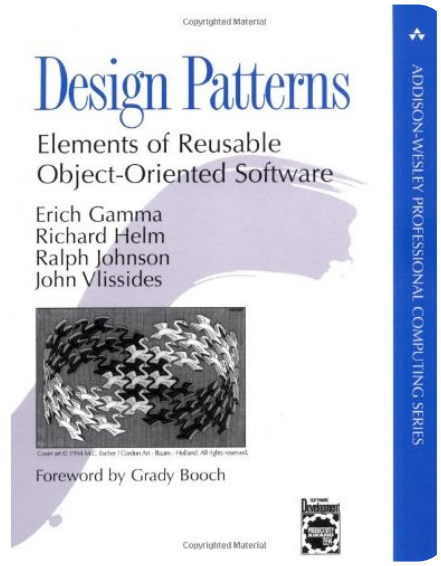
Design Patterns

Elena Punskaya, elena.punskaya@eng.cam.ac.uk

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:
  - To explain how to use these specific patterns in software designs and in communicating about software that uses them
  - 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 – AKA the "Gang of Four" (GoF) book
- which describes 23 design patterns in detail



Why Patterns?

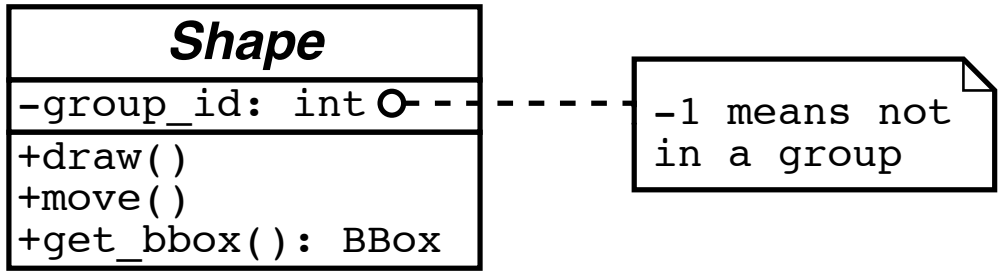
- While software projects are very diverse, conceptually, there are many things that are commonly desired
- Can we have a notification when something specific happens?
- Yes, we can! – **Observer**
- Can we undo the last operation?
- Yes, we can! – **Memento** and **Command**
- Can we access all elements of a collection in a sequential order?
- Yes, we can! – **Iterator**
- Can we build an effective system that allows us to display and manipulate data?
- Indeed! – **Model View Controller** (MVC)
- All modern programming languages implement a lot of these patterns in their API, e.g. Collections-Iterators

Structure of Patterns

- Each pattern could be described using a standard format.
- **Motivation**: outline some specific functionality that we would like our software to provide.
- **Solution options**: explore some ways of providing this functionality and discuss their limitations.
- **Optimal solution**: present a preferred solution based on a design pattern.
- **Code example**: an example of what the design solution looks like using any programming language.
- **Design pattern**: discuss the general principle underlying a good solution and its applicability to other situations. Show the generic design pattern using UML.
- **Disadvantages**: discuss the shortcomings of the design pattern and why you might not want to use it for certain cases.
- We are just familiarising ourselves so will use light version of this approach!

Composite Pattern - Problem

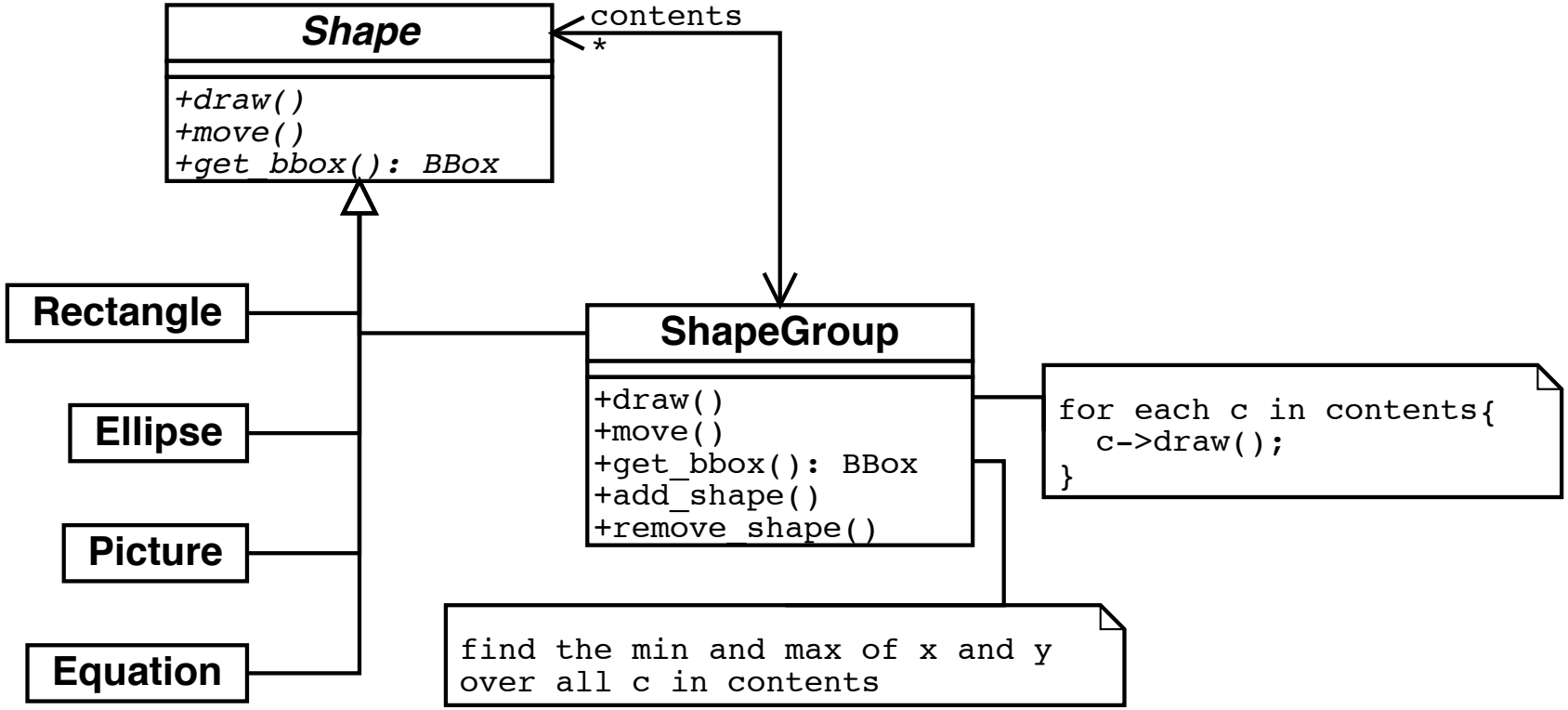
- **Composite** design pattern is used when we want to **operate on individual items** and groups of those **in a common way**
- Problem
  - We want our drawing editor to support grouping and ungrouping operations so that a number of shapes can be collected together and treated as a single entity.
- 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
- Other options? 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

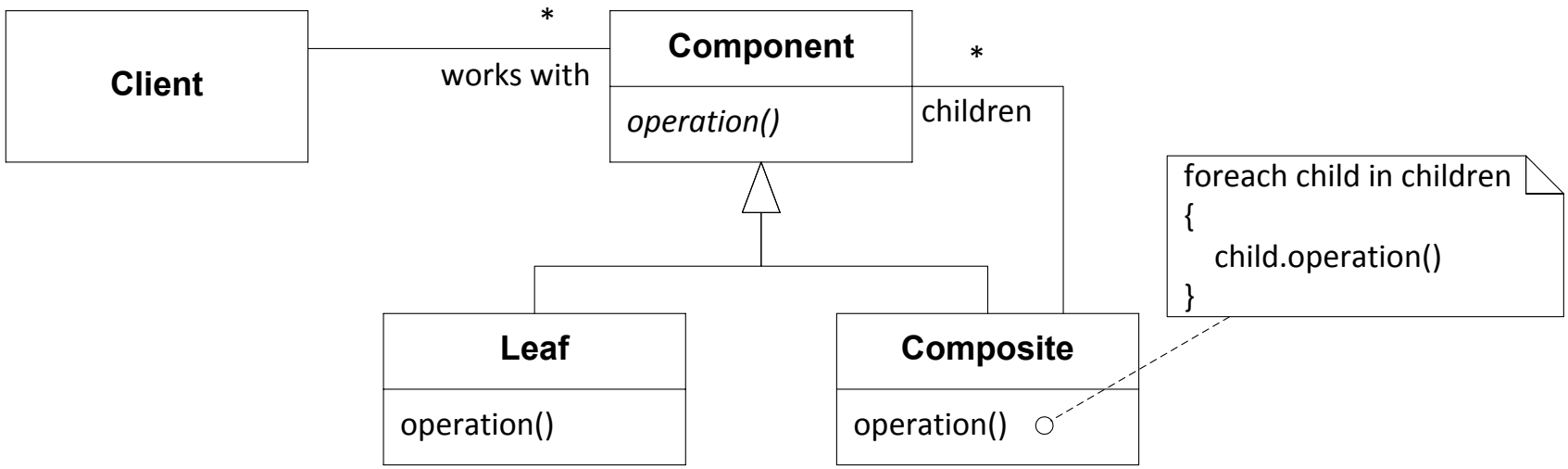
Composite Pattern - Example of Optimal Solution

- 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 only a little bit more complicated and can be done in a similar manner.



Composite Pattern - General UML

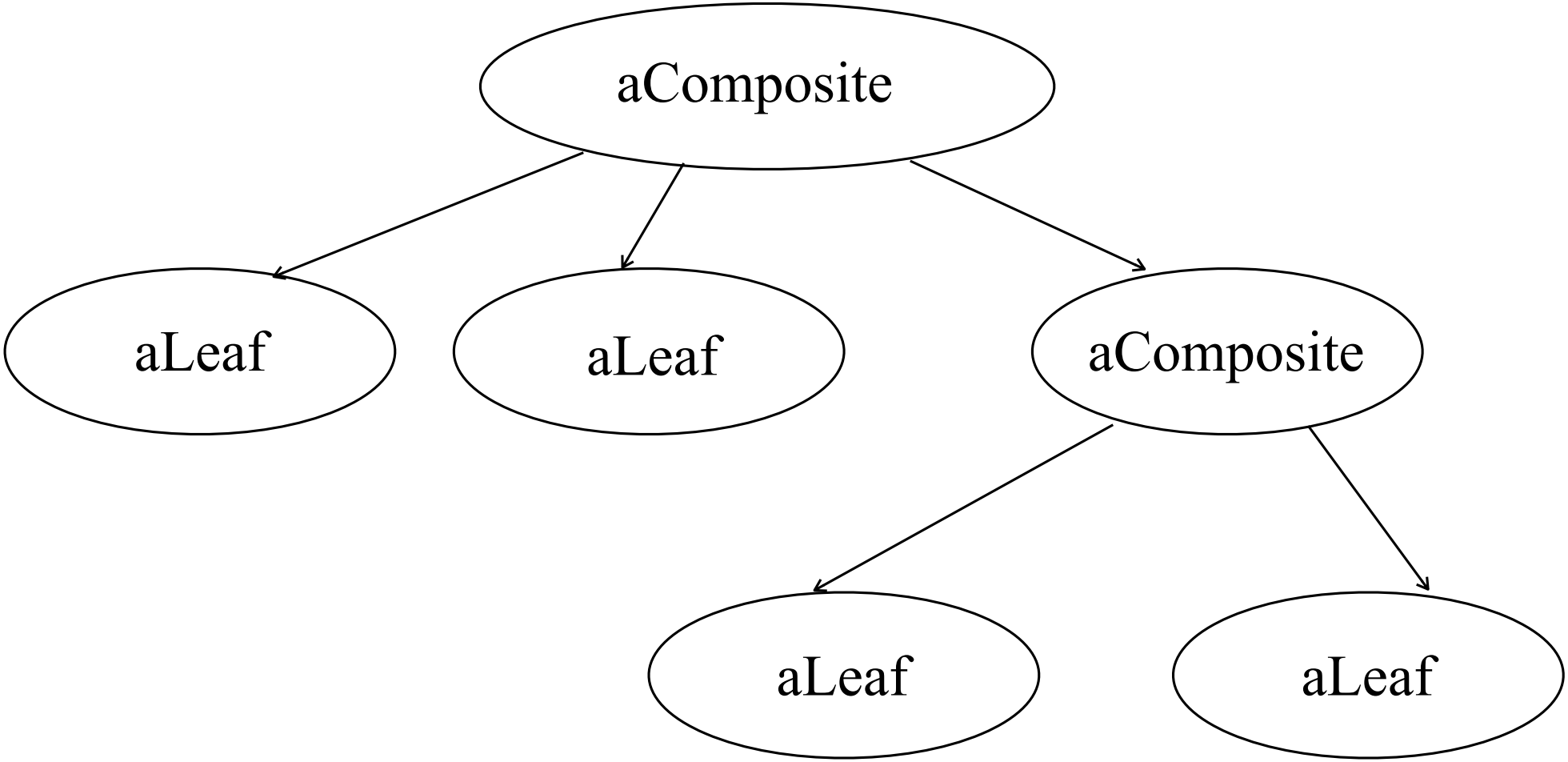
- **Composition** of objects: each component can be a leaf or a composite of other components that in turn can each be either a leaf or a composite
- *Client class does not refer to the Leaf and Composite classes directly, instead it refers to the common interface to treat both Leaf and Component in the same way*



- 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 (add/remove), client code must be able to distinguish between composite objects and non-composite objects.

Composite Pattern - Object Diagram

- Object Diagram (sketch)



10 year retrospective

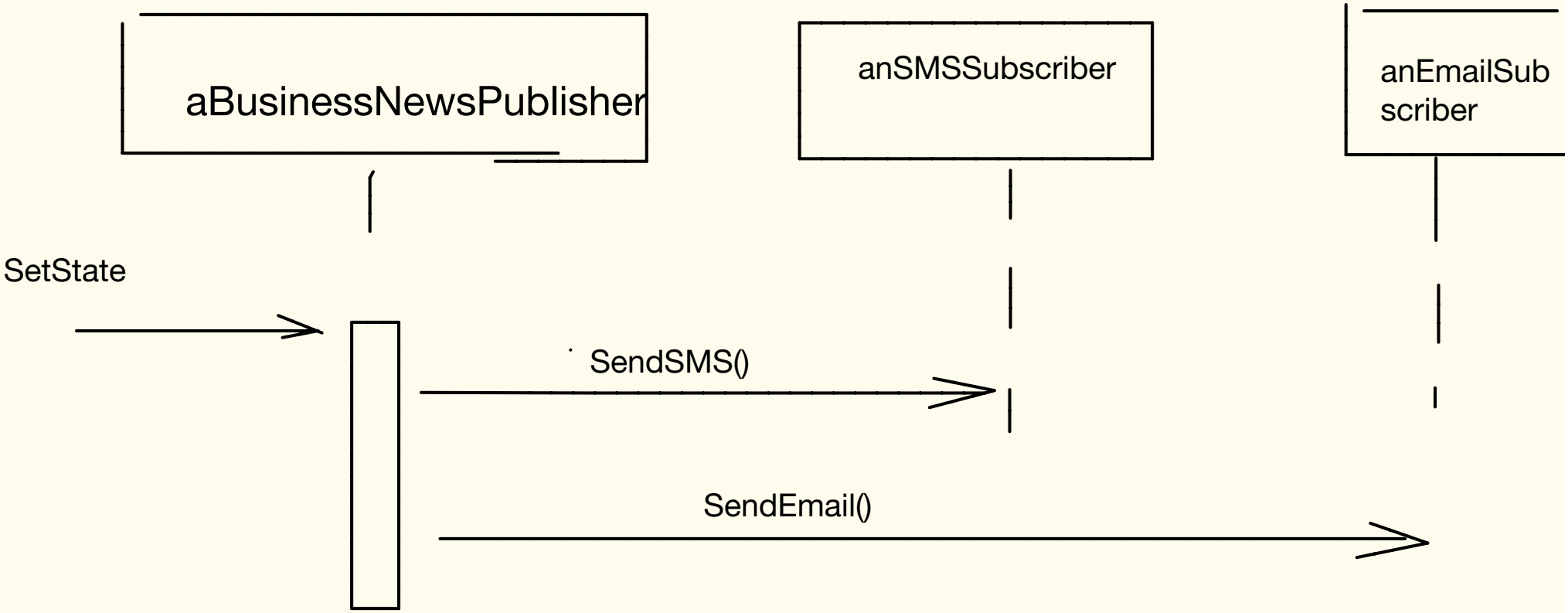
- Erich Gamma, a co-author of the “original” (published in 1994) book on Design Patterns – one of the “Gang of Four”
- Interviewed in 2004 to reflect on 10 years of Design Patterns
  - Source: http://www.artima.com/lejava/articles/gammadp.html

> *Erich Gamma: I think patterns as a whole can help people learn object-oriented thinking: how you can leverage polymorphism, design for composition, delegation, balance responsibilities, and provide pluggable behaviour. Patterns go beyond applying objects to some graphical shape example, with a shape class hierarchy and some polymorphic draw method.* ***You really learn about polymorphism when you've understood the patterns.*** *So patterns are good for learning OO and design in general.*

> *Erich Gamma: One comment I saw in a news group just after patterns started to become more popular was someone claiming that in a particular program they tried to use all 23 GoF patterns. They said they had failed, because they were only able to use 20. They hoped the client would call them again to come back again so maybe they could squeeze in the other 3.*
>
> ***Trying to use all the patterns is a bad thing****, because you will end up with synthetic designs—****speculative designs that have flexibility that no one needs****. These days software is too complex. We can't afford to speculate what else it should do. We need to really focus on what it needs. That's why I like* ***refactoring*** *to patterns. People should learn that when they have a particular kind of problem or code smell, as people call it these days, they can go to their patterns toolbox to find a solution.*

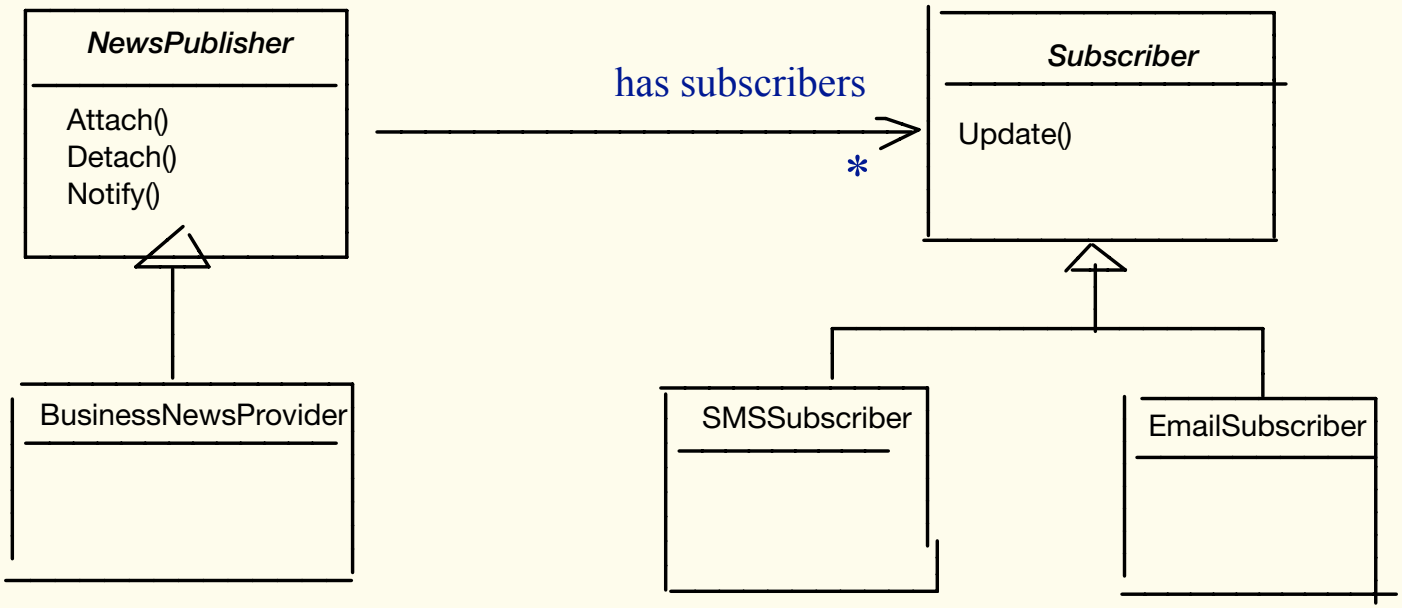
News Publisher Example - Simple Solution

We have a News Publishėr which needs to inform immediately its Subscribers about the event when it occurs.



Observer Pattern - News Publisher Example

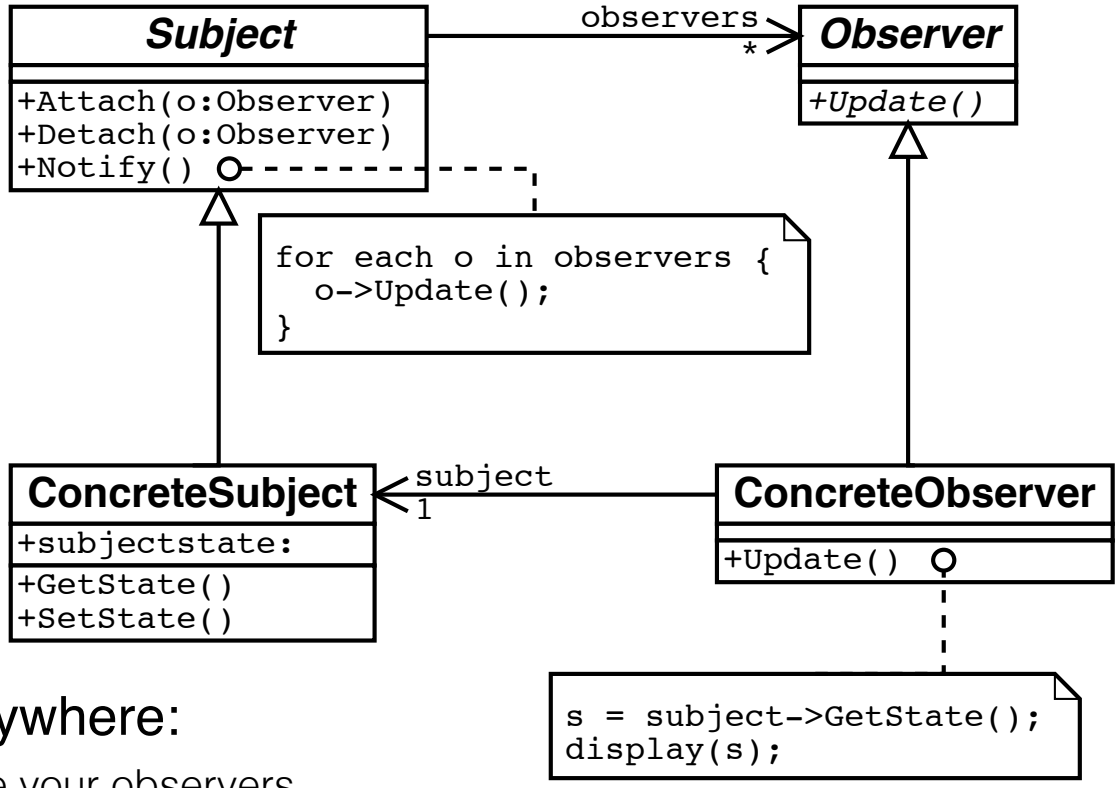
We need to extend the system to introduce more and more different types of Subscribers. It pollutes NewsPublisher class which does not need to know anything about Subscribers or new communication technologies/extensions.



- NewsProvider does not update the state of the dependent objects directly. Instead, Subscriber interface is called by NewsPublisher to update the state

Observer - Class Diagram

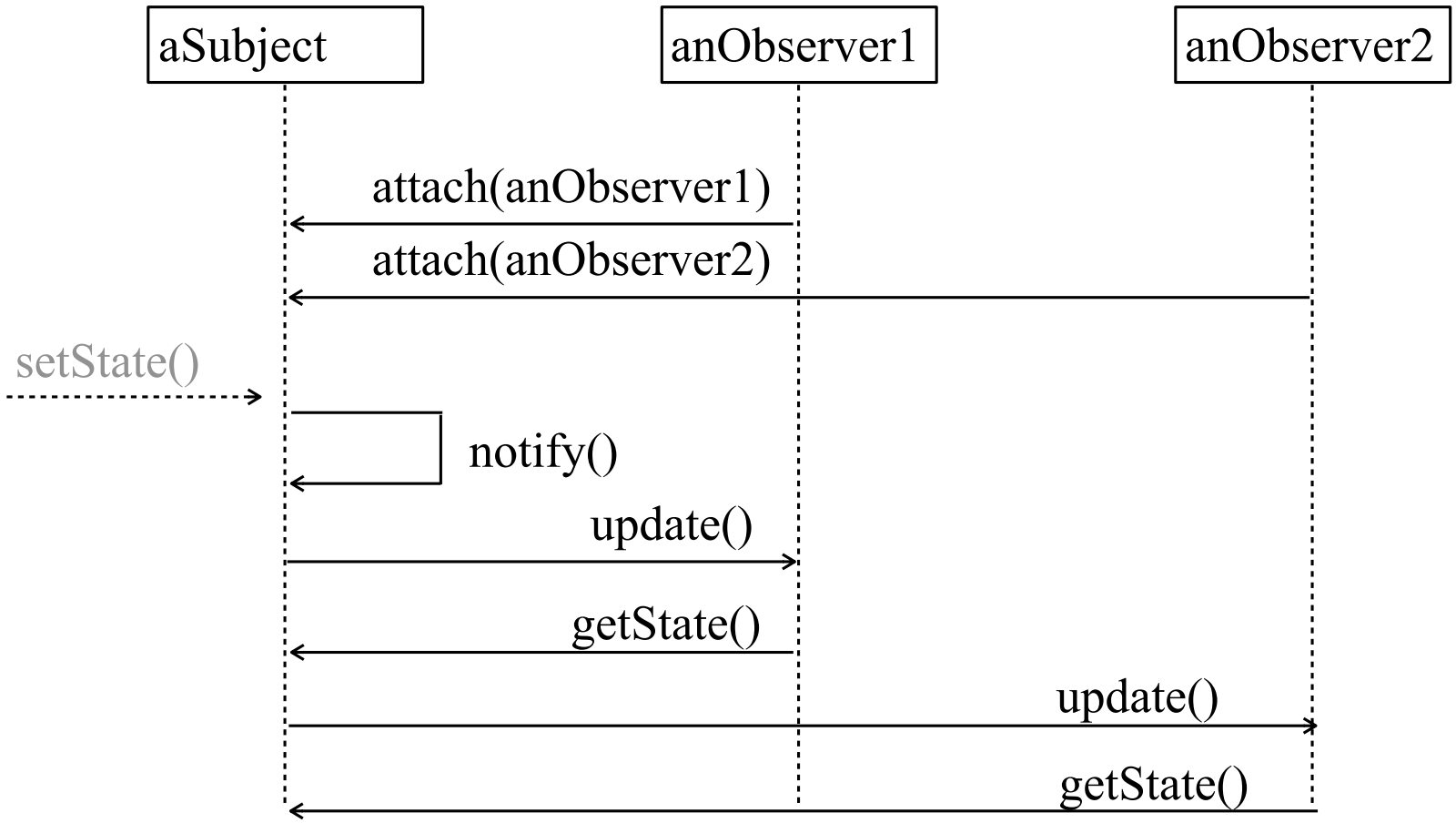
- Allows **multiple objects** to maintain a consistent view on the **state of the object** of interest



- Applies virtually everywhere:
  - your Twitter followers are your observers
  - when you type a search term on Google website, it is observing each keystroke as you type and tries to provide a match dynamically
  - a camera on the phone can notify your app when a snapshot is available
  - a multi-window (multi-view) application can maintain a synchronised view

Observer - Sequence Diagram

- Sequence Diagram (sketch)

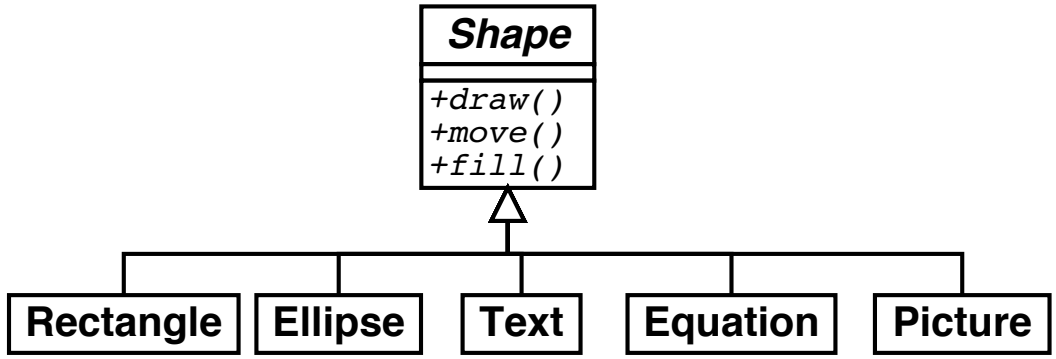


Observer

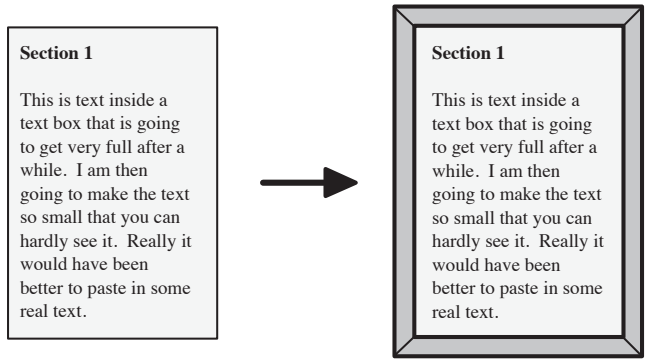
- Disadvantages
  - This pattern could lead to a large amount of computational overhead if not safe-guarded, in particular, if a rate of notifications is high and the reaction to those updates is a heavy-load operation.
- For example, consider an augmented reality mobile app
  - it requests the camera for a real-time snapshots, when the snapshot is ready, the app can analyse it – a heavy operation, involving image processing, Internet access and update of the user interface
  - however, while we analysing the snapshot 1, a snapshot 2 can be available already
  - need to make sure we ignore “snapshot ready” notifications while analysing

Decorator

- **Decorator** Pattern allows to add functionality without changing the original class
- Problem
  - Suppose our drawing editor allows us to include many sorts of shapes including rectangles, ellipses, text, equations, pictures etc.



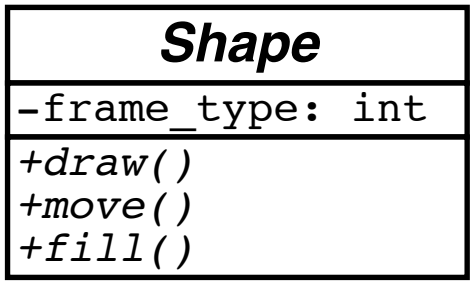
  - Now we want to introduce a facility into the editor to allow frames to be added to arbitrary objects. For example we might want to put a picture frame around an image, or we might want to frame an equation or some text in a simple box



Text **Text** in a frame

Decorator

- 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)

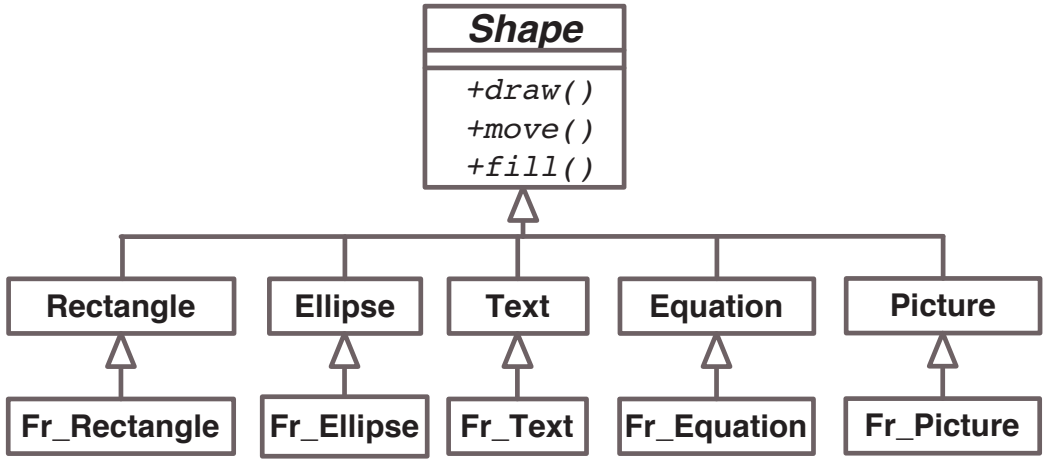


- Pros: simple and adequate for case where we only want to add one special attribute to shapes
- Cons: the code can become clumsy since, for example, the draw method would need a case switch for each of the possible frame types

```
switch(frame_type) {
    case NONE: break;
    case SIMPLE_FRAME:
        draw_simple_frame();
        break;
    ...
}
```

Decorator

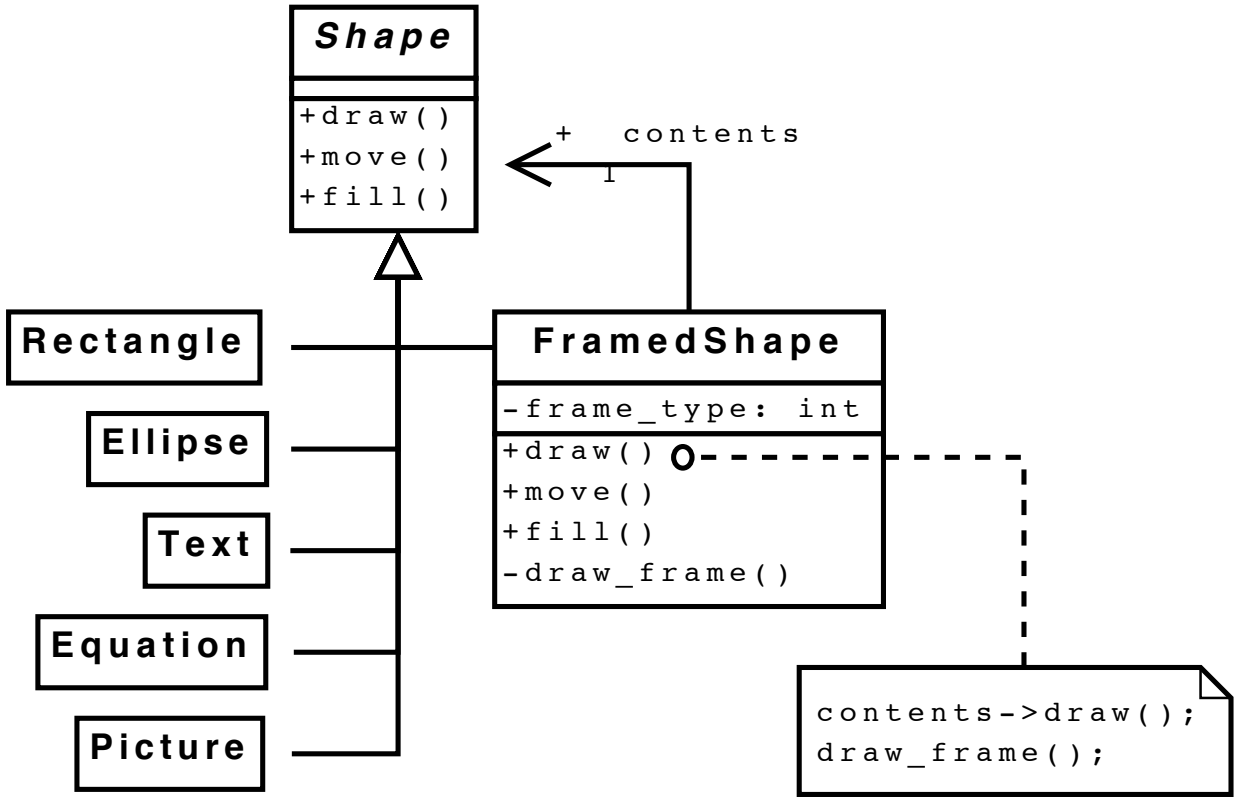
- Solution 2
  - An alternative would be to derive new classes such as Fr Rectangle, Fr Picture, Fr Equation etc. to provide framed versions of each shape class



- Pros: framing can be restricted to particular shapes, efficient use of storages since frame data is only allocated when actually needed
- Cons: huge proliferation in classes, hard to turn decorations on and off at runtime
- Note that the framed versions will inherit exactly the same interface as their parents, as it is essential that any client using any shape object sees an identical interface

Decorator

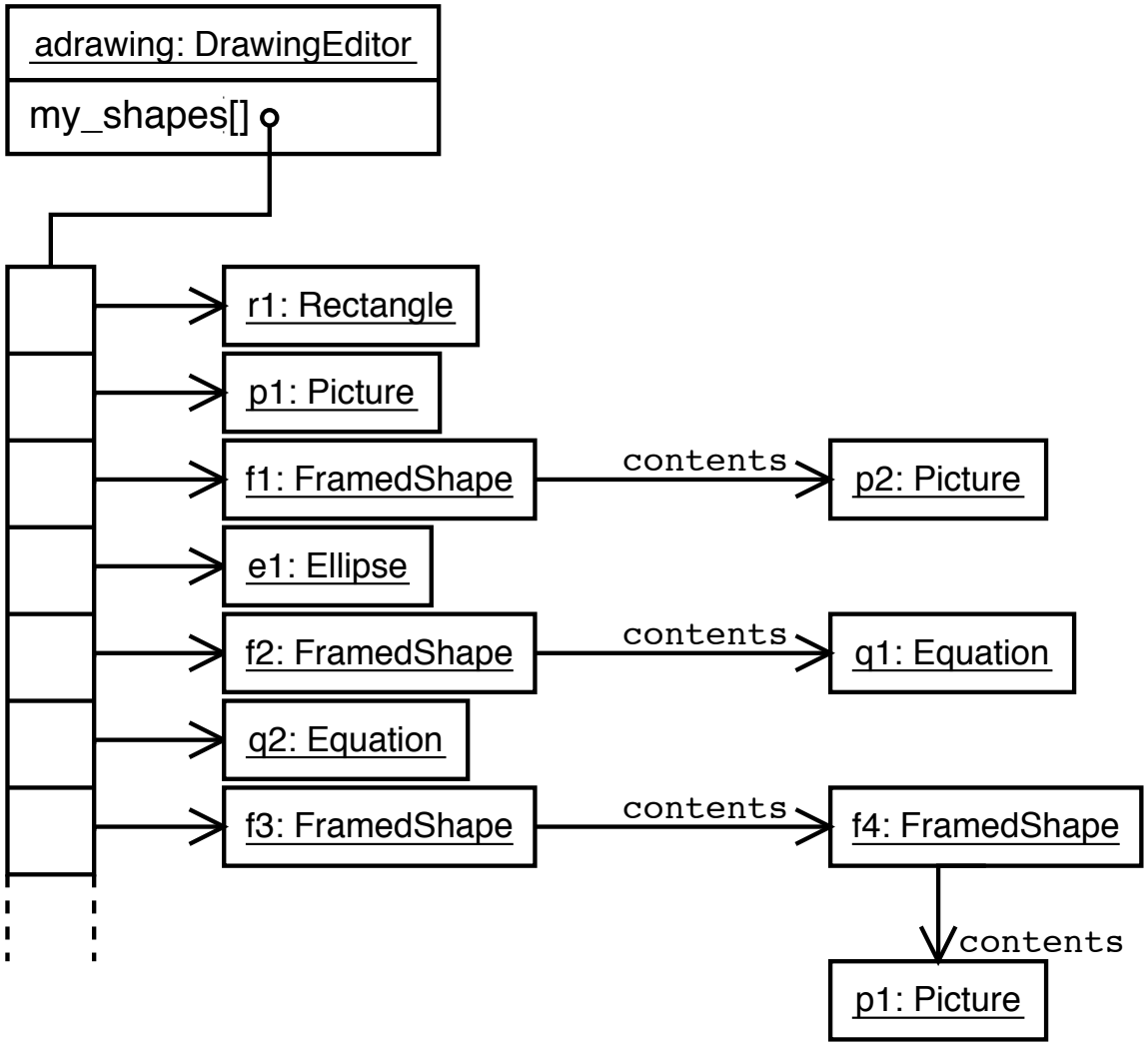
- Optimal solution
  - A much better way to solve this problem is to add a single new subclass of **Shape** called **FramedShape**. Each FramedShape will have a pointer to a **Shape** object which is the shape contained in the frame



  - 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**!

Decorator

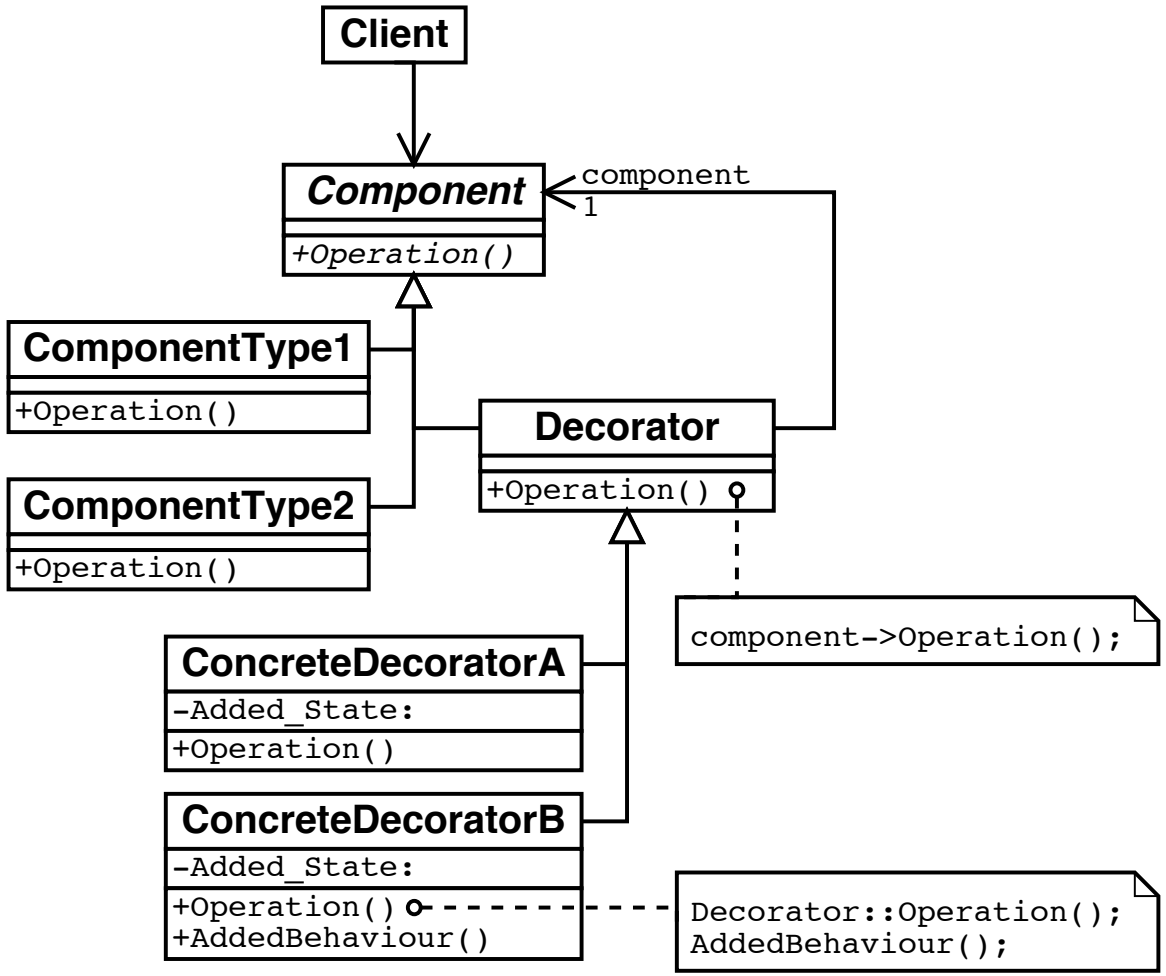
- Example of the object structure at runtime
  - Note, Picture p1 has a double frame



Decorator

- **Decorator** Pattern provides a way of adding optional functionality (“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



Decorator

- 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
- 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
- Consider

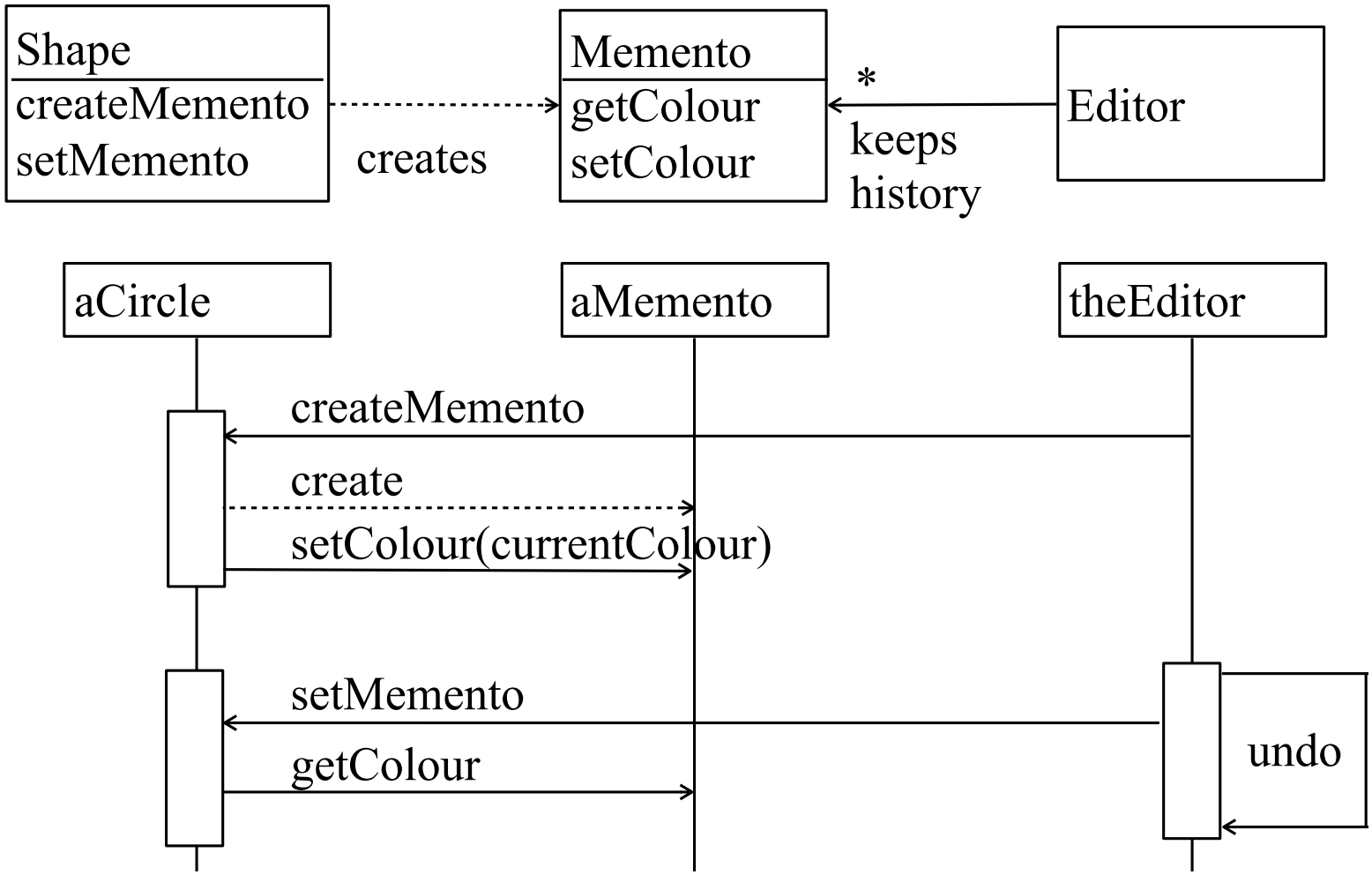
- Not every feature should become a decorator class!

Memento

- **Memento** Pattern allows us to track the state of the object externally, without knowing all details of that state
- Problem
  - A drawing editor is not very useful if it does not support Undo/Redo functionality
- Solution 1
  - each action makes a copy of the object before applying changes and saves it
  - when Undo is called, the editor substitutes the current reference to the object with a reference to the previously saved copy of the object
  - e.g. before calling ChangeColour(Red) the editor makes a copy of the Shape as ShapeBeforeTheColourChange, then to Undo, it will “forget” the reference to the Shape and instead change it to the ShapeBeforeTheColourChange
  - What about Redo?
- Pros: the history is maintained without knowing what changed inside the object
- Cons: expensive – each action makes a full copy of the whole object

Memento

- Optimal solution
  - encapsulate the change into an object of the dedicated class – Memento
  - make the Editor to request a Memento from the Shape before applying any changes
  - make the Shape responsible for creating and restoring “Mementos”



Memento

- Memento Design pattern allows to capture and externalise object's internal state without breaking encapsulation

  - Using this pattern, the Caretaker decides when to request a Memento from the Originator (*knows when the why and when the Originator needs to save itself)*
  - the Originator knows what data should be saved for restoring its State later and can create a Memento and return it to the Caretaker (*Originator knows how to save itself)*
  - the Caretaker can keep an ordered history of Mementos
  - (*Memento is a locked black box that is taken care of by Caretaker but can be opened by Originator only*)

- Disadvantages
  - Originator's state could include a lot of data (probably, the overall design needs a review)
  - When managing a multiple number of Originators, need to maintain which Memento is for which Originator
  - Both can lead to performance degrading, consider:
    - Adobe Photoshop can support up to 1000 history states (Undo's) but limits it to 20(!) states by default
    - Before Photoshop v.5.0 (1998) only a single level of Undo was available

Design Patterns Summary

- When thinking of ways of implementing functionality, it helps to check whether other designers/developers have already come across a similar problem? If yes, then maybe there is already a “recommended” way of solving it
- Do not re-invent the wheel – use established patterns
- Helps to keep the implementation design (code) extensible (“design for change”) and easy to understand
- There are many sources to get familiar with patterns and concrete implementations in specific languages/application types:

- However, using patterns != good design
  - it is NOT about “we implemented 15 patterns in our app, it must be good” or “this is a simple problem, but I am sure I can make it more complicated and then use design patterns to solve it – everyone will see then how clever I am”