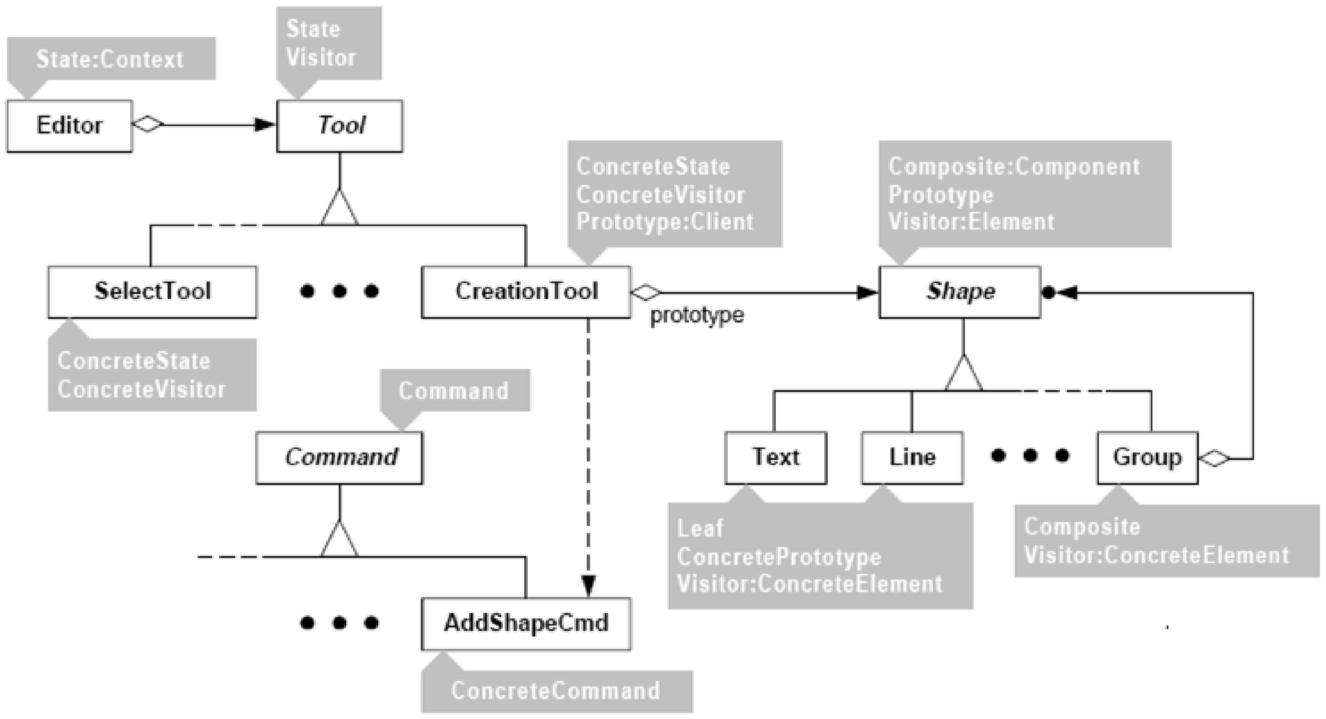
Tooled Composite - Architectural Design Pattern

Ever wanted to create a "direct manipulation" program where you select various tools and manipulate stuff on a workspace? These sorts of applications are quite difficult to build due to the many possible combinations of behavior that is needed. Every tool must know what to do with each type of object. How do we manage this sort of complexity?

- How do you represent shapes?
- How do you represent tools?
- How do tools and shapes interact?
- How do you enhance the editor with new shapes and tools?

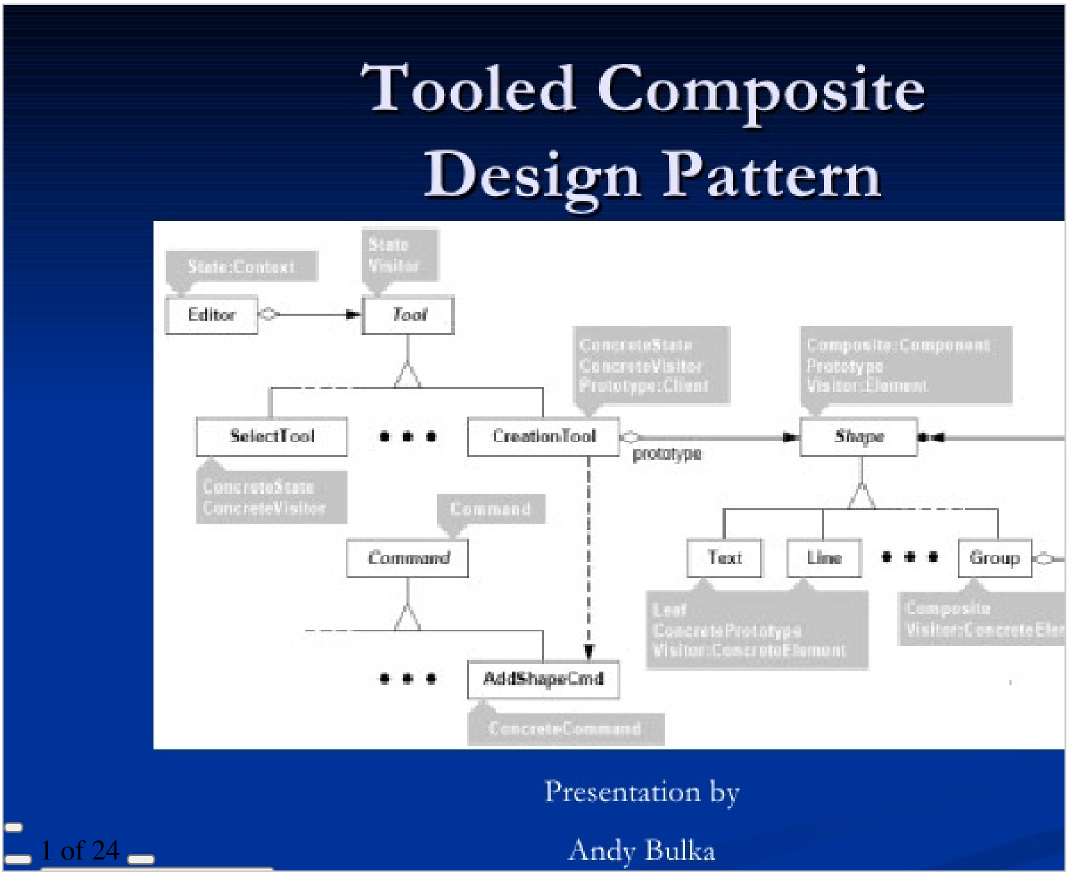
Well the GOF (gang of four) author John Vlissides documented this pattern in this paper

http://www.research.ibm.com/designpatterns/pubs/ph-sep99.pdf



Tooled Composite Pattern

Here is a presentation I gave at the Melbourne Patterns Group about using this pattern in the implementation of the software Rationale where I was Chief Software Architect. It ended up being a success, though we did learn a few things along the way (see below).


Tooled Composite Design Pattern

State:Context
State Visitor
Editor
Tool
ConcreteState ConcreteVisitor Prototype:Client
Composite:Component Prototype Visitor:Element
SelectTool
CreationTool
prototype
Shape
ConcreteState ConcreteVisitor
Command
Command
Text
Line
Group
Leaf ConcretePrototype Visitor:ConcreteElement
Composite Visitor:ConcreteElement
AddShapeCmd
ConcreteCommand

Presentation by

Andy Bulka

1 of 24

Tooled Composite Design Pattern from **tcab22**

What we learned

Whilst implementing this pattern basically saved our bacon, in terms of getting the job done, having the pattern rely on visitor so much did add a layer of complexity that was tiresome to maintain.

Here are some some things we learnt after living with this pattern for a few years:

Classic approach	Possibly a more Practical approach
visitor design pattern	use RTTI (or equivalent e.g. have each shape return a shapeType enum) for better comprehensibility.
3D table of possibilities, with events, shapes, tools on each axis.	table too sparse and complex, so just code for the cases you want.
some blend of visitShape() / visitEdge() etc methods and mouse event methods, within each tool	skip most of the visit methods and do the logic in the mouse handling methods. Generalise the mouse handling into one event (mouseAction) and use if statements to catch the situations of interest. You know what the current shape is by having a pointer to it (set up for you by the tool or something).
use tooled composite for all interactivity	have pockets of interactivity where a component takes over and looks after the gui instead of doing everything the tooled composite way (more explanation below)

With regards to the last point, what we were doing was considering the idea of creating self contained components that had their own little world inside themselves, that was totally independent of the tool/visit system. This way we didn't have to use this architectural pattern for absolutely everything, since that was getting really intricate and tedious.

We considered for example building a box (shape with text in it with various sub editable parts like titles and radio buttons) editor using the regular .NET component system and use their well understood way of handling events, and editing etc. - rather than coding visitors and miniscule tool interactions. So the idea was that as we went into say, "edit mode", we would switch away from the tooled composite pattern and into the world of .NET - just during the edit.

Final Verdict

Make no mistake, we desperately needed the Tooled Composite Pattern approach to hold everything together, and to be the default system in play. The .NET component approach (described above) definitely wasn't suited to a deep and complex drawing tool like Rationale, with zooming, mini maps, panning, layout algorithms etc. In fact we found in experiments / spikes of a fully .NET component approach that .NET events would fire all over the place causing too much screen refreshing.

So in the end a we used Tooled Composite for mostly everything, and dropped into .NET components only when needed eg. the text editor that you get when you double click on a workspace box in order to edit the text inside it was a pure .NET text control. We never got to push this hybrid approach any further - we wanted to write a super fancy text editor component (with hover zones, places for specifying urls and bold headings etc.) ourselves purely in .NET - we just didn't have the time - thus we mostly stayed completely within the classic Tooled Composite Pattern. And I'm not sure what we would have done without it!