Need to Know: Patterns for Coupling Contexts and Components

James Noble
Computer Science,
Victoria University of Wellington, New Zealand.
kjx@mcs.vuw.ac.nz

Abstract

Many object-oriented design patterns — including State, Strategy, and Observer — split objects up into a context and a component: the context is fixed, while the variable component typically provides some services to the context. Control primarily flows from context to component, and accompanying data must flow the same way. This paper contains four patterns that describe how to design the interfaces between contexts and components. These patterns should help programmers to design the interfaces introduced by patterns that split contexts and components.

Introduction

Taking things apart, that's easy.
Putting them back together, that's the trick.
Attributed to David Holmes

Many object-oriented design patterns from *Design Patterns* [4] or *Patterns of Software Architecture* [3] increase the flexibility, longevity, complexity, and arguably the ineffable *quality* [1] of program designs by introducing (or *"finding"*) new objects. For example, the State, Strategy and Observer patterns introduce eponoymous State, Strategy, and Observer objects: although these are three behavioral (sic) patterns [4] the creational Prototype pattern and structural Bridge patterns share the same dynamic.

A key part of all these patterns this that they split an existing object, introducing a new object (technically a new participant or role [9]) to encapsulate variation, and refactor some of the responsibilities of the existing object into the new object. We call this new object the *component*, and the the remainder of the existing object of which it may have been a part we call the *context* [2]. Figure 1 shows how these patterns split objects into contexts and components.

But, as David Holmes, Humpty Dumpty, and the second law of thermodynamics all attest, it's easier to separate things out than put them back together again. An important part of using all these patterns successfully is designing the interface between the new component and the existing context. This interface design is particularly important for longevity and reuse: especially as many programs and frameworks need to be able to reuse both contexts and

Pattern	**Intent**	**Context**	**Component**
Strategy	vary algorithms	Context	Strategy
State	alter behaviour	Context	State
Observer	vary dependent aspects	Subject	Observer
Prototype	vary class of created object	Client	Prototype
Bridge	decouple abstractions and implementations	Abstraction	Implementor

Figure 1: Contexts and Components

components. Although many patterns motiavate the split on the basis that contexts can reused by changing components, it's often equally important that components can be used in many different contexts — for example, we hope one Subject can be used with many kinds of Observers, but in practice it's also important that one type of Observer can be used with many kinds of Subjects.

This paper presents four basic patterns for designing the interfaces inside these patterns, that is, for coupling contexts and components. Coupling — determining how much components need to know about their contexts and vice versa — is also the main force between these patterns.

Figure 2 summarises the problems dealt with by this collection of patterns, and the solutions they provide.

Form

The patterns are written in modified electric Portland form. Each begins with a question (in italics) describing a problem, followed by a bullet list of forces and discussion of the problems the pattern addresses. A boldface "**Therefore:**" introduces the solution (also italicised) followed by the consequences of using the pattern (the positive benefits first, then the negative liabilities, separated by a boldface **However:**), an example of its use, and some known uses and related patterns.

Pattern	**Problem**	**Solution**
Context passes parameters	How can the context receive the information it needs from the context?	Pass parameters along with the requests from context to componnet.
Component knows parameters	How can the component receive constant information from the context?	Initialise a unique component with the constant information.
Context passes itself	How can the component retrieve large amounts of information from the context?	Pass the context as a parameter to methods that need the information.
Component knows Context	How can the component and context interact without restriction?	Initialise a unique component object with a pointer to the context.

Figure 2: Summary of the Patterns

1 Context passes parameters

How can a component receive the information it needs from the context?

- You have split an object into a context and a component.
- The context delegates operations to the component.
- The context needs to pass information to the component with these operations.
- Most of this information changes for each delegated operation.

Therefore: *Pass parameters with the requests from context to componnet.*

Design the component's interface so that each delegated operation receives the information that it needs as its parameters. Ensure that the context passes the correct information as it delegates operations to the component. When the component's method is starts running, the necessary information should be right at hand.

Example

Consider a system for physiolometric simulation of the consumption of alcoholic drinks. The context (the simulation framework) can construct a new component (a `drink` object) and then send messages to that object, passing parameters to describe the amount of liquid drunk.

```
Drink absolut = new Vodka();
absolut.drink(100);
absolut.drink(150);
```

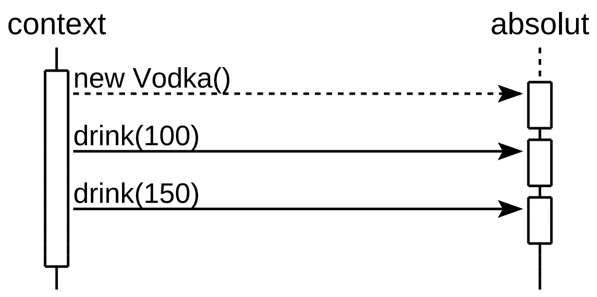

Figure 3: Context passes parameters

Consequences

This pattern makes the component easy to design, because the information its methods need will be present as their arguments. Since components are not storing information about their contexts, a single context can easily be shared amongst several contexts (this is the underlying dynamic of the Flyweight pattern). The data flow is simple, directly from context to component along with (and in the same direction as) the control flow, making this design easy to undestand and debug. Because the component doesn't require callbacks to retrieve information out of the component, this makes a concurrent design easier: the context object can operate in a parallel thread without a direct risk of interfeering with any threads executing in the component.

However: This makes components easy to design by making protocols hard to design: the protocol needs to deliver the right information to the right methods at the right time. Such complex protocols complicate contexts, which must supply this information to the components, whether they need it or not. If there are more than a few parameters to pass, the protocol between context and component can become large and unweildy. To design a minimal protocol, you have to know in advance which parameters will be needed when: and (as Kent Beck would say) I hope you're better at fortune-telling than I am.

Known Uses

Observers in Smalltalk and Java are typically passed parameters describing details of the event they are observing upon. Strategies and Bridge implementations are passed parameters to the operations that are being delegated — in a strategy to format a printout, for example, each element to be printed will be passed as a parameter to the strategy object in turn.

Related Patterns

If the values of the parameters are the same across the calls, consider COMPONENT KNOWS PARAMETERS (2) (or CURRIED OBJECT [7] in extreme cases). If a large amount of dynamic information needs to be transferred, consider using an ARGUMENTS OBJECT [7] to package that information into a single parameter. For a large amount of more static information, consider CONTEXT PASSES ITSELF (3).

2 Component knows parameters

How can the component receive constant information from the context?

- You have split an obejct into component and surrounding context.
- The context delegates operations to the component.
- The context needs to pass information to the component with these operations.
- The component needs a large amount information about its context.
- This information is the same for most operations delegated to the component.

Therefore: *Initialise a unique component with the constant information*

Design the component so that it stores the information it needs to complete its work. You will need to ensure that the component has some way to receive this information, typically by initialising this when it is created using extra parameters to its constructor methods: the component (or other object) will need to pass this information when the component is created. You will also need somewhere to store this information, typically instance variables (member fields) in the component object.

Then, this information will be available when the component needs to carry out its reponsibilities.

Example

Imagine modelling a drink dispenser, rather than a glass where different amounts can be drunk at a time. We can initialise the dispenser with the amount to be dispensed for each shot, and no longer need to pass an amount swallowed each time a drink is drunk.

```
Dispenser finlandia = new Vodka(100);
finlandia.drink();
finlandia.drink();
```

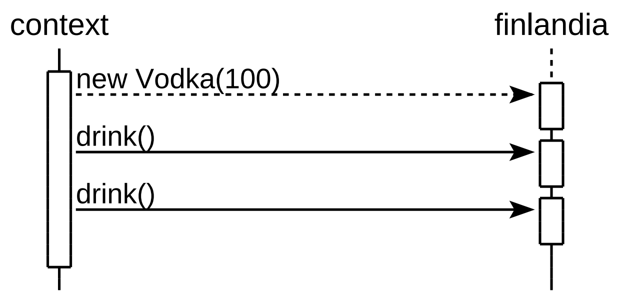

Figure 4: Context knows parameters

Consequences

Once a component knows its parameters, you don't have to pass them every time: this makes the protocol between context and component simpler, and the context easier to write. The component is no more difficult to write either, because the paramters are just as available in an instance variable as they would be in a method parameter.

However:

You have to design the initialisation protocol carefully, so that it passes all the values correctly. You have to store parameter values in the component (but then, you probably had to store them somewhere, right?). You can only share components between different contexts if they have the same values for the shared parameters, or else some context could get a component with the wrong parameter values. You have to know the values of the parameters when you create the component, or you have to provide an subsidiary interface to allow the context to change the values.

Known Uses

Many components accept configuration parameters — for example, a stragegy used to lay out a dialog box will have parameters to control the spacing and margins in the layout; a character flyweight will have a parameter to record the character code and font information. These are often known by the strategy or flyweight rather than being initialised in advance. Observers (such as views or widgets) also often have configuration parameters describing how they should display themselves, such as fonts and colours once again, and so on.

Related Patterns

If the values of the parameters differ across the calls, consider passing them rather than making the component know about them. If large amounts of information is involved, consider passing or storing the context itself, rather than each parameter individually.

3 Context passes itself

How can the component retrieve large amounts of information from the context?

- You have split an object into context and component.
- The context delegates operations to the component.
- The component needs a large amount of information from the context, complicating the protocol between the two objects.
- It's not always obvious what information the component will need, or when it will need it.

Therefore: *Pass the context as a parameter to methods that need the information.*

Rather than trying to guess which information the component will need, make the context pass *itself* to the component. In this way, the component can retrieve whatever information it needs from the context, just when it needs it: that is, when it has been delegated the responsibilty to perform some operation on behalf of the component.

Example

In a wine (or vodka) tasting session, multiple drinkers each drink from the same bottle (usually using different glasses that are then discarded, however). How each drink is rated depends on a number of features of each drinker, such as their thirst, inebriation, and in some cases dipsomania. We can model this by making each drinker a context, and passing the drinker to the (shared) drink.

```
Drinker james = new Drinker();
Drinker ali = new Drinker();

Drink smirnoff = new Drink();
```

```
// ...
smirnoff.drink(james,100);
smirnoff.drink(ali,150);
```

Being passed its context (drinker), the drink object can make whatever inquiries are appropriate as part of the drink method (see Figure 5).

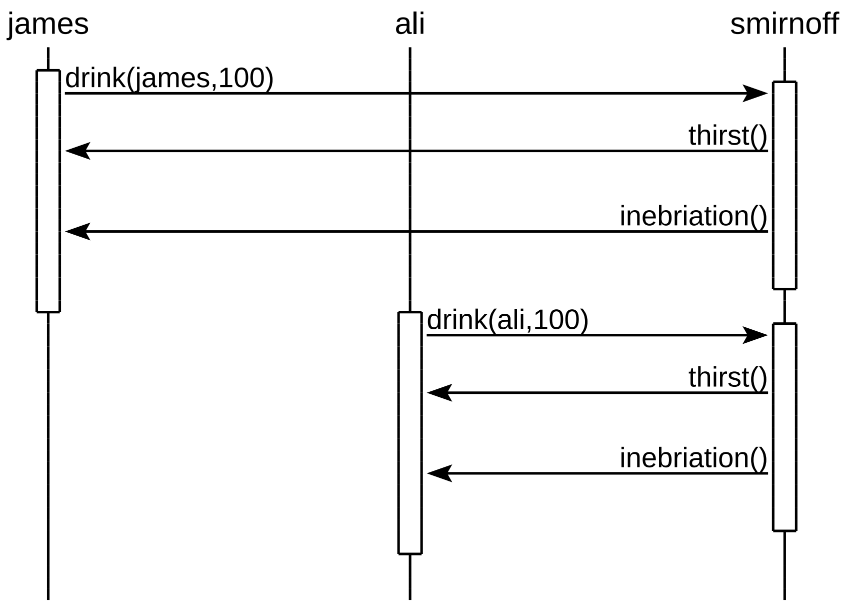

Figure 5: Context passes itself

Consequences

This pattern is quite easy for the context: it simply must pass itself everytime it delegates a message to the component; the protocol must be extended to include this argument, but this is not generally a major complication. This pattern also has the advantage that the component can access any information it may need from the context, without the context having to make any special arrangments to pass that information to the component. The component doesn't have to store any information between calls, so that it doesn't have to allocate memory to store information, and, if there is no context-specific information in a component, it can be shared dynamically between several different contexts.

However: This makes protocols easy to design, but components and contexts harder to design: in particular, the context's interface must make available any information the component may need. The component now needs to know the interface of the context object, or at least a subset of that interface that allows it to collect the information it needs: this information is not provided on a platter, directly as parameters to the methods that need it. The component can retrieve essentially any information from the context, making the program harder to understand and debug. The two-way messages sends (from context to component and back again) don't make things any easier, either, nor does the fact that the control flow (from component to context during the callback. can be in the opposite direction to the data flow (from context to component)

Known Uses

Observers often also use this pattern (as well as other patterns we have described here). In Smalltalk, for example, the object that raised a change notification is always passed to its observer: this allows one observer to distiguish between multiple subjects. Sharable State objects are also often passed their context, so that they can change the state of the correct context object.

Related Patterns

If the same object is always passed as the context to a given conponent, consider COMPONENT KNOWS CONTEXT (4) as an alternative. If the component doesn't need much information from the context, consider just passing those parameters rather than the whole context.

4 Component knows context

Also Known As: Backpointer

How can the component and context interact without restriction?

- You have split an object into a context and a component.
- The context delegates operations to the component.
- The context needs to pass information to the component with these operations.
- You're considering Passing Context as a Parameter, but a large number of delegated operations need to access the context.
- The component needs to contact the context asychronously.
- The component needs to initiate interactions with its context.

Therefore:

Initialise a unique component object with a pointer to its context.

Make an instance variable in the component that can store its context, and extend the component's creation protocol to accept the context as ta parameter that you can use to initialise the variable. Of course, every context that uses this component will have to pass this parameter correctly, but only whenever it creates the component, not every time it delegates operations.

The component's reference to its context is often called a *backpointer* because it points "backwards" up the composition hierarchy.

Example

Consider modelling a waiter who will ask drinkers if they want a refill, and refill them if necessary. The waiter needs to be able to query drinkers whenever necessary, rather than waiting until the drinker asks for a refill (although that should also be possible).

```
Drink largeVodka = new Vodka();
Drinker james  = new Drinker();
Waiter bill = new Waiter();

bill.waitUpon(james);
bill.order(largeVodka);
```

The waiter object knowning about its context (the drinker) allows it to interrogate its context whenever it is appropriate (see Figure 6).

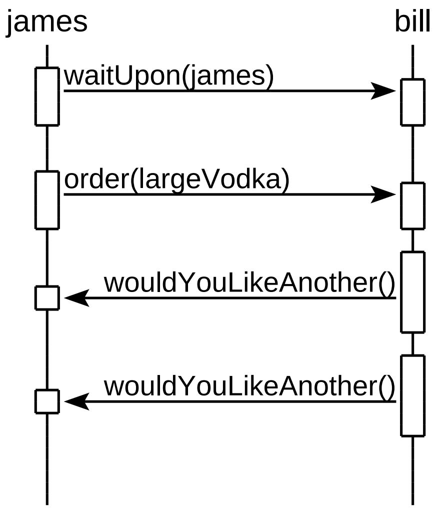

Figure 6: Component knows context

Consequences

If a component knows about its context, the context is always available to the component, which can get whatever it needs, whenever it needs it. This pattern is quite easy for the context, as it simply has to pass itself when initialising the component; after that, the component can look after itself. The protocols between context and component can be very simple, sending only the extra information needed to describe the instantaneous delegated operations, as the other information can be fetched from the context as needed.

However: As when passing contexts as parameters, the context's interface must be designed carefully to make available any information the component may need. The component can now send callbacks to the context at any time. This can make the program more complex and harder to debug that the other approaches we have discussed, and can cause serious problems in concurrent programs (such as deadlock) if the context is protected by a separate lock etc [6].

Finally, because the component object needs to know it's context, is is much harder to share components between contexts: each context will need its own unique component, increasing the number of objects and memory requirements of the system.

Known Uses

The usual suspects: Observers typically know their subjects. The VisualWorks framework [8] had separate view classes for those views that knew their model and those that did not. Complex strategies (such as complex layout managers) that need lots of information about their contexts also often use this pattern: these are then not sharable.

Related Patterns

If the component doesn't need much information from the context, consider just storing those parameters (COMPONENT KNOWS PARAMETERS (2)) rather than the whole context.

5 Patterns and the Topology of Design Space

In describing these patterns, we had an interesting choice: whether to describe the patterns themselves, as we have do, or to describe the underlying options (or forces) on which the patterns are based: to pass parameters versus passing the whole component object; and to pass parameters or contexts dynamically rather than storing them statically in the component object (see Figure 7). Each of these options could certainly have been written up as a pattern itself.

	Parameters	Context
Dynamic	Pass Parameters	Pass Context
Static	Know Parameters	Know Context

Figure 7: The 2^2 Design Space covered by these patterns

In this paper, we chose to focus on the more concrete solution generated by the combination of options, rather than each option individually. This let each pattern be more concrete: describing a particular point in the design space, rather than a partial solution that had to be completed by applying another pattern. The complemantary disadvantage is that each pattern is more concrete than it needs to be: the choice between storing and passing arguments, say, is a very general one that must be faced in many more situations than simply desinging context-component protocols. On the other hand, the more general a technique or pattern, the less useful it is in any given situation [5], and it is becasue we consider the design of these internal protocols to be important in the practical application of many other patterns that we have made our descriptions concrete.

There is another, more pragmatic, reason we have for writing the concrete descriptions: simply that there are so few options that it is practical to treat each concrete point in the design space individually: with a binary planar space (i.e. two axes of choice and only two options on each axis) there are just four discrete points in the space (see table 7): enumerating each point on the each axis would still mean describing four patterns. With a further axis, or three or more options on even one of the axes, the number of concrete spatial points would far exceed the number of axial points, so describing each axial point wouild be the only option.

Acknowledgements

Many thanks to Ali Arsanjani who gamely shepherded this paper through KoalaPLoP 2001. Cheers!

References

[1] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[2] Robert Biddle and Ewan Tempero. Understanding the impact of language features on reusability. In Murali Sitaraman, editor, *Proceedings of the Fourth International Conference on Software Reus e*, pages 52–61. IEEE Computer Society, April 1996.

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

[4] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[5] Michael Jackson. *Software Requirements & Specifications.* ACM, 1995.

[6] Doug Lea. *Concurrent Programming in Java.* Addison-Wesley, second edition, December 1998.

[7] James Noble. Arguments and results. *Object Oriented Systems*, 2000. Accepted May 1998.

[8] ParcPlace Systems. *VisualWorks Smalltalk User's Guide*, 2.0 edition, 1994.

[9] Dirk Riehle. Composite design patterns. In *ECOOP Proceedings*, 1997.