



COMPUTING INTEGRITY

INCORPORATED

60 Belvedere Avenue
Point Richmond, CA 94801-4023
510.233.5400 Sales
510-233.5444 Support
510.233.5446 Facsimile



The “Gang of Four” Decorator Pattern: What Is Its Appropriate Use? 22 February 2010 Thomas Mercer-Hursh, Ph.D.

In discussions on PSDN¹ reference has been made to the use of the Decorator pattern as a mechanism in the Progress Professional Services Model-Set-Entity (M-S-E) pattern for creating subtypes of Business Entities. Decorator is proposed as an alternative to the use of Generalization and Delegation. There are really two separate issues in this discussion. One issue is whether the pattern proposed in M-S-E is to be preferred generally for creating subtypes, or, if not generally, under what circumstances should it be preferred? The other issue relates to whether such usage was intended by the Gang of Four² when they described the Decorator pattern. Because this latter issue seems to have been one which has incited strong conflicting opinions, the present whitepaper has been created to review the text of the Gang of Four description of Decorator with the goal of illuminating the intended use. This is a separate issue from the virtues of the pattern used in Model-Set-Entity, which will be discussed separately³.

This discussion will simply review the text of the Decorator pattern as expressed in the book, extracting quotes as appropriate, and discussing the possible apparent meanings. Section headers correspond to the sections of the pattern presentation in the book. The comparison with respect to handling subtypes will be to Generalization and Delegation⁴. Generalization refers to decomposing responsibilities into common responsibilities, typically in the form of a superclass, and type specific behavior, typically in the form of multiple subclasses of that superclass. Delegation refers to identifying cohesive units of responsibility and separating that responsibility into its own class such that the Delegate assumes the responsibility for some area in support of the primary class with which it is associated. The two can be used in combination, i.e., a Delegate may itself be a superclass with a number of subclasses, representing subtypes of the general area of responsibility of the Delegate. Generalization and Delegation are design-time decisions and thus static. I.e., any given object must be of one specific subtype and must be associated with any required Delegates.

¹ In particular, see <http://communities.progress.com/pcom/thread/21340?tstart=0>

² So called from the book Gamma, Helm, Johnson & Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

³ I.e., by separating this into two discussions, the intent is to separate “What did GoF intend?” from “Is the pattern in M-S-E a good way to handle subtypes?” since the two questions can be addressed independently and the answer to one does not predetermine the answer to the other.

⁴ See Mercer-Hursh, Thomas: *Object-Oriented Vocabulary: An Introduction* for the definitions of words used in this document.

Review of Pattern Sections

Intent

“Attach additional responsibilities to an object dynamically.” “Dynamically” seems to imply something different from the static division into types and subtypes which is the typical result of decomposition analysis using Generalization and Delegation. Of course, Decorator is different in that it is applied at run-time rather than being defined as a part of the static design. Indeed, this dynamic quality is considered an advantage by proponents of Decorator since it means that assignments of functionality can be very flexible. However, greater flexibility in this respect comes at the price of less structure.

“Decorators provide a flexible alternative to subclassing for extending functionality.” This sentence lies at the heart of the differing interpretations. On the surface it seems to place Decorator as an equal and equivalent alternative to using Generalization for defining subtypes. However, one must note that the Intent section is characterized by very terse and simple statements which therefore might not convey the full sense of what was intended.

In particular, the authors might have said the same thing here based on quite a different perspective. For example, in analyzing the a set of objects in the problem space, one might observe that one could identify at least three different types of functionality:

1. **Static Subtypes:** Functionality which was statically divisible into and associated with some number of fixed subtypes;
2. **Separate Static Responsibilities:** Functionality which can be statically separated into a coherent hierarchy independent of the other variations in the main set; and
3. **Dynamic Variations:** Functionality which was dynamically associated with a subtype, i.e., it did not always apply to every object in the subclass or it might apply to a particular object only for a limited time and unpredictable combinations of functionality might be needed by individual objects.

One viewpoint is that the first type is typically addressed with Generalization; the second with Delegation; and the third with Decorator. Thus, one might make the statement in the GoF book based on the notion that the third type of functionality was not easily handled with Generalization and so Decorator was a “flexible alternative” for functionality in that category. One can’t determine which was meant from this one sentence alone, so we need to look at the overall context provided in the pattern for clarification.

Also Known As

This section provides no specific insight except to provide an alternate name, “Wrapper”.

Motivation

“Sometimes we want to add responsibilities to individual objects, not to an entire class.” This seems to reinforce the dynamic character noted in the Intent and seems to clearly separate Decorator from the kinds of subtype distinctions which one makes with Generalization since those are static and all objects in the inheritance hierarchy will always belong to a particular subtype.

“A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component.” This seems to introduce a further wrinkle on the functionality types in that the same functionality is applicable to multiple component types. This is clearly something which would not fit in a normal Generalization hierarchy.

“One way to add responsibilities is with inheritance. ...” This paragraph describes how a border may be unsuitable for Generalization because it is static and inflexible and a client cannot select which behavior to use or not use. This seems to me to be the type of functionality covered in the third type above.

The remainder of this section is simply a description of layering of Decorators to achieve a desired effect focusing on scroll bars and borders on text areas in which there is an assumption that one may want one, both, or neither on any particular instance. Diagrams show the inheritance structure by which Decorator is implemented.

This situation above in which there are two unrelated sets of characteristics is often referred to as the “Crosstab” problem. I.e., like a table showing one set of options along the top and another set of options along the side, all or most of the cells represent possible combinations of functional requirements. This situation is not limited to only two options, but may involve a large number of different functionality options. Let’s identify **Crosstab** as a fourth type of functionality to add to the three described above. A crosstab which is not decomposed into separate responsibilities presents a difficulty for Generalization because one must first branch the hierarchy on one option and then branch it on the other. This means that the responsibilities unique to each branch on the second option need to be duplicated in both sides of the tree, violating good design principles. Moreover, the decision of which option to use for the first branch is often arbitrary and thus it is easy for changing requirements to alter the desired order of branching, resulting in a need to rework significant parts of the tree. If the crosstab is expanded to more than two dimensions, it leads to a combinatorially complex tree. Such complexity is exactly what GoF is trying to simplify.

Another observation that we might make about this example is that the functionality discussed is simply present or not present. Moreover, there is no special new functionality associated with the option not being present, it simply doesn’t exist. I.e., adding scroll bars adds new functionality to the base functionality of the text area, but the state of the text area having no scroll bars, the text area has only the base functionality of the text area, not some alternate functionality. This is quite different from a functionality in which there are two or more states, each of which has its own functionality. This is not a different category of functionality since it might apply in each of the four types identified, but it is a significant characteristic of functionality which we might want to notice later on, so let’s call this **Unary** or **One-sided** functionality.

Applicability

“Use Decorator:

- to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition maybe hidden or otherwise unavailable for subclassing.”

This seems to be a very important section since it is the one that is telling us when to use Decorator.

The first point again emphasizes the dynamic aspect and the focus on individual objects. This seems a very different purpose than Generalization which is about static decomposition that applies to all objects in the set. The text doesn't, of course, say that it is wrong to use Decorator for static typing.

The second point further reinforces this dynamic or transient quality, but focuses on removing the functionality rather than adding it.

The third point seems to directly address the interpretation of Decorator as an alternative to Generalization. Here, there is an explicit statement that it is to be used when Generalization is “impractical” and specifically points to the crosstab problem which exists when there are a multiple, independent aspects of functionality which can be combined onto a single object. It also mentions the case where the class definition is unavailable for Generalization in some way. Thus, in this section, it seems that Decorator is not offered as a general alternative for Generalization, but is suggested in cases where Generalization is not going to work well or is not possible.

One of the cases where Generalization does not work well is where the functionality is dynamic or transient since Generalization is about static classification. A class can have state dependent behavior without Decorators, but the behavior is always a part of the class definition whether the state is active or not. If there are too many states or options, the class can become bloated. This is the Dynamic Variation functionality described above.

The other notable place where Generalization does not work well is where there are multiple alternatives that can be freely combined, i.e., the Crosstab functionality described above. While it is possible to construct hierarchies for this type of functionality, such hierarchies are undesirable because it is necessary to duplicate functionality across branches and they are often fragile in response to changing requirements.

One might also note that one-sided functionality does not fit very well with Generalization since one wants to instantiate terminal subclasses in the tree, but with one-sided functionality, there is only unique functionality on one side of the branch. This implies the need for a bunch of subclasses which contain nothing to represent the state when the functionality is not present, which is clearly undesirable.

Thus, for Dynamic Variation functionality and for options involving one-sided functionality, it can be recognized that Generalization is often not a good solution and Decorator offers itself as a pattern which addresses this type of problem well. But, what about Crosstab functionality?

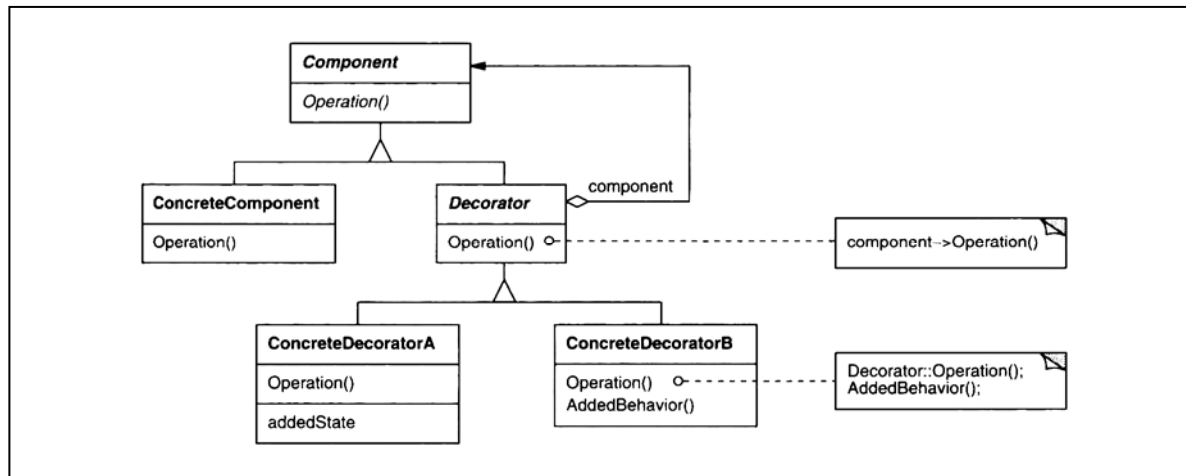
One key question would seem to be whether Crosstab functionality was common in static decompositions. This would seem to depend on the nature of the decomposition which one performs prior to considering whether to use Decorator or some other approach. If the decomposition fails to identify separate, coherent spheres of responsibility which can be properly removed to their own class hierarchy via Delegation, then one is certainly likely to encounter complex objects with complex sets of crosstab functionality possibilities. However, if one does identify and separate those Delegates, then the complexity of the options remaining in the primary object is reduced correspondingly and it seems likely the crosstab character can be eliminated.

There is likely to be a matter of preference here. One person may prefer static decomposition because the various subtypes and possible combinations are clearly delimited by the model

structure. Another person may prefer dynamic combinations because of the flexibility, especially if there is a perception that requirements will change frequently in ways that a static structure either could not support or where the static structure would be disrupted by the change. However, this disruption is less likely if separable areas of responsibility have been moved to delegates, so in both cases, decomposition to eliminate any crosstab characteristics results in a more stable and flexible structure.

Structure

This section illustrates the structure by which the Decorator pattern is implemented and does not apply to the manner of its use. The diagram is provided here for reference.



Participants

This section provides names and description for the classes involved in the Decorator interaction as shown above.

Collaborations

This section notes that the Decorator forwards requests to its Component object and it may invoke its own behavior before or after such invocations. While not explicitly stated here, the Decorator will have its own knowledge and behavior independent of its Component.

Consequences

This section lists two benefits and two liabilities of the Decorator pattern.

“More flexibility than static inheritance. The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance.” This benefit reinforces the idea of Decorator being used in place of Inheritance. As with the statement in the Intent section the question is whether this is meant as a general purpose alternative or an alternative to use in special cases. The balance of the discussion in this paragraph refers specifically to cases in which there are multiple subtypes which intersect, i.e., the crosstab problem. It also references adding the same property twice, e.g., a double border. Thus, the focus, as in the Applicability section, is on contexts in which Generalization is impractical or complex. There does not appear to be any

suggestion that it should be used routinely for cases in which Generalization is simple and straightforward⁵.

“*Avoids feature-laden classes high up in the hierarchy.* Decorator offers a pay-as-you-go approach to adding responsibilities. ... Functionality can be composed from simple pieces. As a result, an application needn't pay for features it doesn't use.” This point complements the first point, i.e., faced with entities with complex functionality one option is to create a class which contains all possible functionality as a part of the standard base class, but this is considered undesirable. Decorator provides one approach by which needed functionality can be added as needed to any specific instance For type 3 functionality and one-sided functionality Decorator would appear to be preferable. Generalization and Delegation provide other ways to manage type 4 crosstab functionality. Selection among these options depends on the context, as will be discussed further in the conclusion.

“*A decorator and its component aren't identical.*” If one has a need to rely on object identity, Decorator can present a problem since the Component and the Decorator are not the same object, so dynamically wrapping the Component with a new Decorator will dynamically change the identity of the package. This is mostly an issue where one is making dynamic changes in functionality. If the Decorators are applied as a part of the building process and remain in place during the lifecycle of the Component, this is less likely to be an issue.

“*Lots of little objects.* A design that uses Decorator often results in systems composed of lots of little objects that all look alike.” Of course, one might also have a lot of classes in any design where there is a rich functionality which is used in multiple combinations. However, with Decorator, each unit of functionality is a wrapper for the Component and thus all wrappers are quite similar to one another. If the same functionality is decomposed using Generalization and Delegation, there may be a similar number of total components, but they will be structured into inheritance and collaboration hierarchies rather than simply being a collection of wrappers. With Generalization and Delegation, it will also be clear what combinations are required and allowed, which is not immediately apparent with Decorator and thus must be enforced with logic in the factory. Of course, the factory needs to build the correct Delegates as well, but those dependencies are inherent in the model, while the allowable and required combinations of Decorator must be managed with external logic.

Implementation

“*Interface conformance.* A decorator object's interface must conform to the interface of the component it decorates. ConcreteDecorator classes must therefore inherit from a common class.” This principle is illustrated in the diagram under Structure in which the Component Class exists to provide a common ancestor for the ConcreteComponent and the Decorator. This class defines the interface presented by the Decorator, except for the additional Decorator-specific interface.

“*Omitting the abstract Decorator class.*” This point merely notes that the abstract Decorator class can be omitted when there is only a single Decorator.

“*Changing the skin of an object versus changing its guts.* We can think of a decorator as a skin over an object that changes its behavior. An alternative is to change the object's guts. The Strategy (315) pattern is a good example of a pattern for changing the guts.” This is an interesting implementation

⁵ It should be noted that Delegation may be required to reduce the complexity of the primary object so that Generalization is simple and straightforward.

point in a couple of different ways. The discussion talks about using the Strategy pattern to create a linked set of alternate functionality descending from within a Component as an alternative to wrapping the Component with Decorators. By selecting from alternate versions and combining them in a chain as needed, the same kind of flexible combination of functionality can be achieved as is achieved with Decorator. These two approaches are contrasted as follows:

- “Since the Decorator pattern only changes a component from the outside, the component doesn't have to know anything about its decorators; that is, the decorators are transparent to the component”
- “With strategies, the component itself knows about possible extensions. So it has to reference and maintain the corresponding strategies”
- “The Strategy-based approach might require modifying the component to accommodate new extensions.”
- “On the other hand, a strategy can have its own specialized interface, whereas a decorator's interface must conform to the component's.”

One point of interest here is that the authors are again clearly talking about a context in which there is a need for flexible, dynamic addition of functionality to the base entity. Nowhere in this Implementation section do they reference an example of using Decorator or Strategy for routine subtyping.

In particular, it is interesting to note the Strategy is structurally similar to Delegation except that in Delegation there is a fixed, specific relationship between the primary entity and the Delegate while in Strategy there is a general purpose “hook” for category of functionality.

One can argue that Decorator is preferable to Strategy because the Primary entity is not coupled to its Decorators, but is coupled to some extent to the Strategy components because it needs to know about them. However, by the same token, the Decorator knows about its Component and the Strategy component does not need to know about the containing entity. One might make a similar observation about Delegates except that the degree to which the entity knows about the Delegate is the degree to which it needs to use the Delegate, i.e., the knowledge is necessary and appropriate.

Sample Code

This section provides some specific C++ code illustrating the same kinds of Window and decorations like borders and scrollbars and shadows which are referred to elsewhere in the discussion.

Known Uses

“Many object-oriented user interface toolkits use decorators to add graphical embellishments to widgets. ... But the Decorator pattern is by no means limited to graphical user interfaces, as the following example (based on the ET++ streaming classes [WGM88]) illustrates.” The example deals with the need to potentially compress an output stream by one of a number of algorithms and/or to fold the stream into 7-bit ASCII. This is a classic crosstab context in which the stream may be folded or unfolded and may be uncompressed or compressed according to one of several algorithms. Thus, the examples continue to be of cases of type 3 or type 4 functionality where Generalization does not work well. In the streams example, the 7-bit ASCII aspect is an example of one-sided functionality. If it were one of several types of processing, e.g., conversion to UTF-8 or UTF-16 or coding to avoid control characters, then the problem could also be addressed by a combination of a compression Delegate and an encoding Delegate.

Related Patterns

“Adapter (139): A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.

Composite (163): A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities-it isn't intended for object aggregation.

Strategy (315): A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.”

Conclusion

It is characteristic of human writing that even what seems like the most clear and unambiguous text can often be read in more than one way. So it is with this pattern. Different orientations, different experience bases, different values can all make an individual stress one aspect over another or shift interpretation to or from a particular point of view. Therefore, what follows is necessarily a personal opinion and other reasonable people might stress different aspects and come to a different conclusion.

There are multiple places in this pattern where the authors clearly indicate that Decorator is an alternative to Generalization, starting with what seems a very simple and direct statement in the Intent section at the beginning. The question which motivated the current discussion is whether these statements were meant to indicate simple, literal equivalence or whether the context of the discussion indicates that Decorator is an alternative which should be considered in special circumstances or contexts where static decomposition is not a good choice. The simple, direct statement in the Intent section seems to argue for the former, but the much more qualified statement in Applicability suggests the latter.

There seem to be three primary aspects to the discussion which are key to this question.

First, it is clear that Decorator is about adding functionality to individual Objects, i.e., specific instances. This seems to separate it from traditional OOA/D decomposition in which one is attempting to identify cohesive, coherent areas of responsibility to collect and arrange into classes. I.e., if an object of a particular class always has a particular knowledge or behavior or if all objects of a particular class are always one of some limited number of types, then one would expect this sort of responsibility to be a part of the class definition and structure. If, however, any particular functionality applies to some instances and not others and/or that functionality is One-sided, then there is more of an inclination to provide the functionality with a run-time solution.

Second, there is frequent reference in this pattern to the dynamic character of the pattern – functionality is added and withdrawn dynamically; functionality depends on run-time context; context can change during the life of the object. Indeed, Decorator's special virtue is exactly this ability to respond to changing conditions in a very flexible and dynamic way. This flexibility contrasts with the formality of structure in Generalization and Delegation. One might, therefore, think that the intent was to segregate functionality into that which can be analyzed into static structures and that which was inherently poorly suited to such static structures because of its dynamic character. Thus, “alternative” should not be read as “replacement for” but rather as “compliment to”.

Third, the reference under *Applicability*, reinforced by language elsewhere, seems to suggest that Decorator is intended to be useful in cases where *Generalization* is not. This reinforces the sense of Decorator complimenting *Generalization* by providing a mechanism that works in cases where *Generalization* is awkward or clumsy, notably when the functionality in question is dynamic rather than inherent.

Thus, on balance, it seems that there is significant indication that the authors intended Decorator to compliment *Generalization*, not to replace it. *Static Subtype* functionality seems best handled by *Generalization*. *Static Separate Responsibility* functionality is best handled by *Delegation*. Both are the result of simple, traditional decomposition of a problem space into sets of like responsibility. But, *Dynamic Variation* functionality is not going to be easily handled by either *Generalization* or *Delegation* because it is dynamic, not static. Thus, *Dynamic Variation* functionality is most appropriately handled by Decorator or possibly *Strategy*. *Crosstab Variation* functionality needs to be carefully examined. If the complexity exists because one has failed to recognize an opportunity to isolate a particular area into a *Delegate*, then creating the *Delegate* would be a preferred solution because it will result in simpler classes and a clearer structure. But, if the crosstab can not be statically decomposed in a meaningful way, Decorator certainly provides one solution. Likewise, one-sided functionality might be an indicator for Decorator, especially if there are multiple instances for a single class.