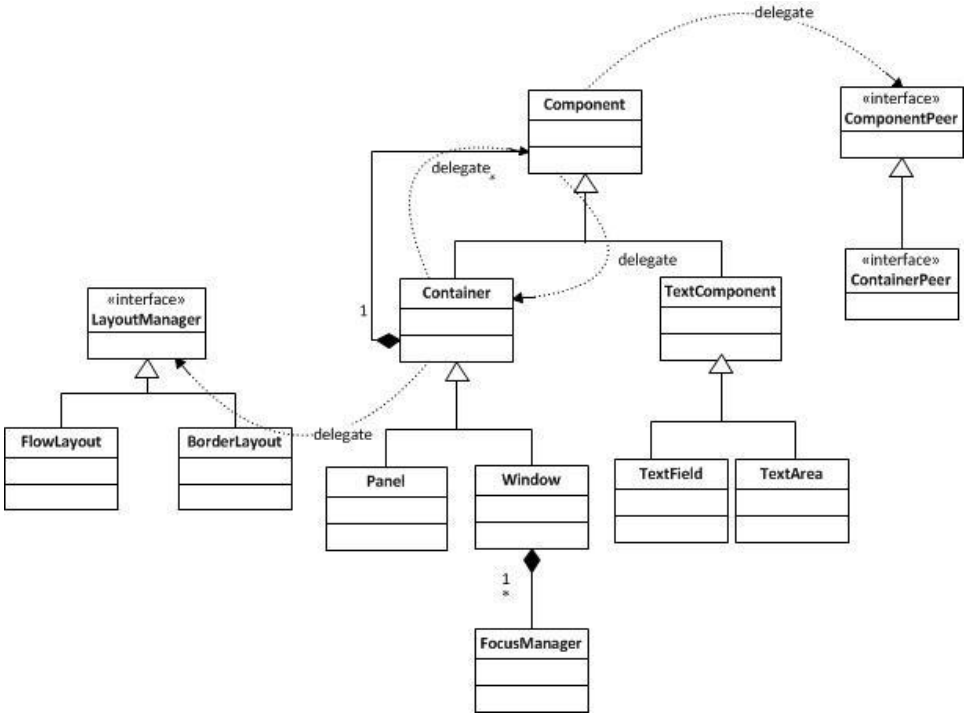
8 Composite Pattern [Gamma et al]

Composite pattern intends a recursive structure in the form of inheritance tree. The pattern classifies each element in the part whole hierarchy as a composite or leaf. Composites are basically container objects that have children whereas a leaf is bottom element that don't have any children. GUI libraries like [28] provide an application example of composite library. Detecting instances of design patterns like composite pattern brings with a lot of complexity reported by [27] and [29]. Indeed sometime pattern definition is abused but yet it is considered to be a variant because it follows the basic structure and intent defined by [1]. Some detection tools ignore the existence of patterns and some try to automate the variant definition using an algorithm such as [30]. Usually tools follow the structural part of a pattern to determine its existence. Developers mold structural part with addition of object oriented mechanisms to provide extra features. Following are the variants of the composite pattern:

8.1 Compound Implementation of Composite pattern in component in Java AWT Library [27]

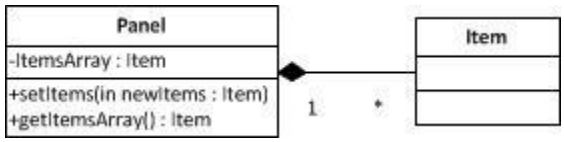
This variant is defined in Java AWT library which has two level of hierarchy i.e. the composites and leaves are abstract classes which have further concrete implementation. The calls are delegated across different objects through aggregation.



8.1 UML Diagram of compound implementation in Java AWT Library [27]

8.2 I-N Relationship using arrays [29]

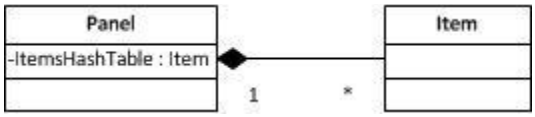
In this variant Leaves / composites are represented as multiples belonging to the same parent using typical array objects. This is a traditional way to add / remove items into a hierarchy.



8.2 Composite implementation using typical array objects of leaves, composite [29]

8.3 I-N Relationship using Hash Tables [29]

This variant makes use of HashTable to contain the referenced attribute i.e. items. HashTable is a container library that provides access to many useful operations that take an extra overhead if explicitly implemented which result in the public visibility of the Items.



8.3 Composite implementation using typical HashTable [29]

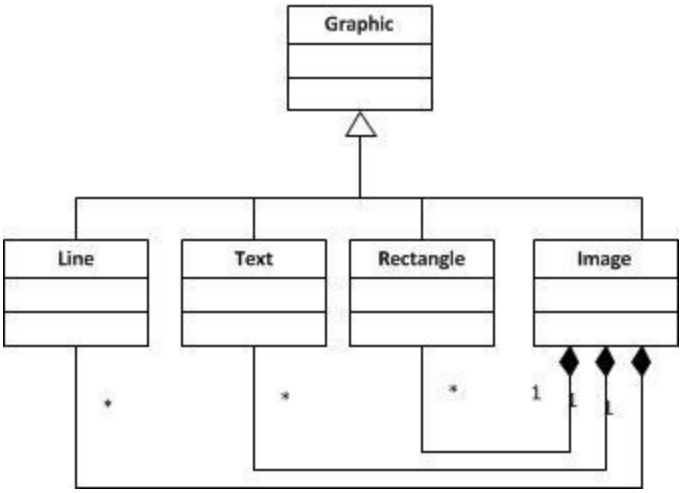
8.4 I-N Relationship using C# Generics An alternative of Vector class in Java [29]

This variant defines the thread safe implementation by using C# generics List<> class. This is a container which can be used to add, remove objects e.g. composites or leaves at run time.

8.4 Composite Implementation using C# Generic List [29]

8.5 Composite composed of other leaves [31]

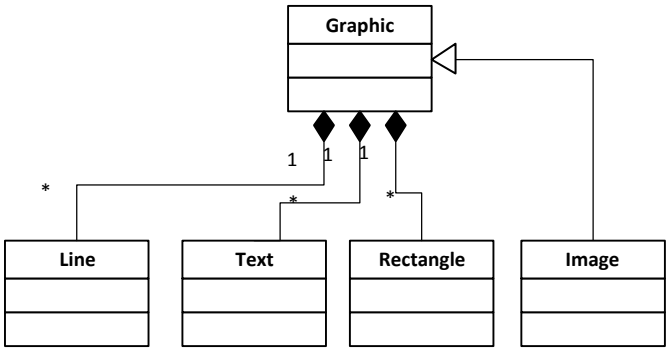
This variant removes the drawback defined by GoF [1] in a problem where a composite consist of component leaves. The example quoted by [31] here is “Image consists of other images or shapes”.



8.5 UML Diagram of composite composed of leaves. [31]

8.6 Memorization of all elements at Component level [32]

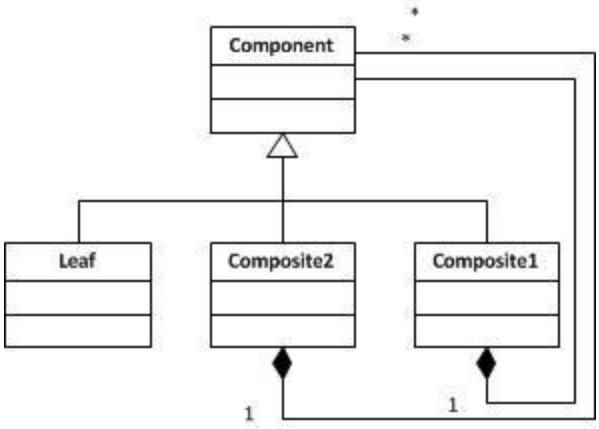
This variant is an alternative discussed above. The author [32] pointed out some spoiled patterns and provides those spoiled fragments of code a complete solution that is a refinement of the problem. In this variant all the composites are remembered at the common protocol (Component Class) level as show below



8.6 UML Diagram of alternative solution of composite pattern memorized at Component Class level [32]

8.7 Reference Participant [32]

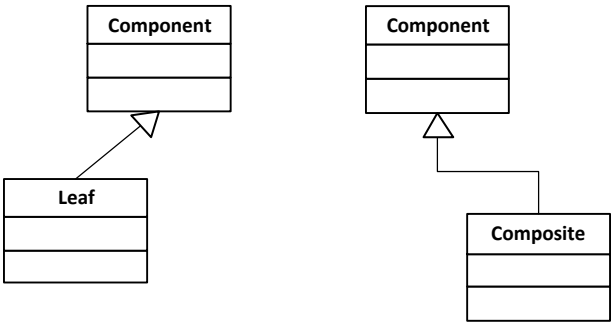
This variant is isomorphic of the composite pattern defined by [1]. Despite having one composite the implementation appends an additional composite to hold the opportunity to contain additional leaves or composites. The author [32] describes this variant into two fragments i.e. {Component, Composite1 and Leaf} and {Component, Composite2 and Leaf} that are identical in structural. This variant is implemented in the model quoted in [32].



8.7 UML Diagram of reference participant implementation of composite pattern [32]

8.8 Sole Leaf and Sole composite [32]

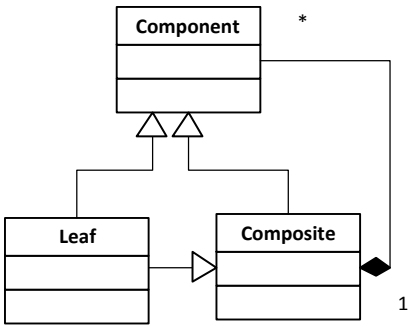
Below are two different implementations of composite design pattern that have one composite and one leaf element. Losing the any participant element (composite or leaf) does not break the intent of pattern. Losing addition of elements into the hierarchical structure maintains the pattern definition despite of losing participant occurrences.



8.8 UML Diagram of one composite and one leaf in composite design pattern [32]

8.9 Supplementary Relationship [32]

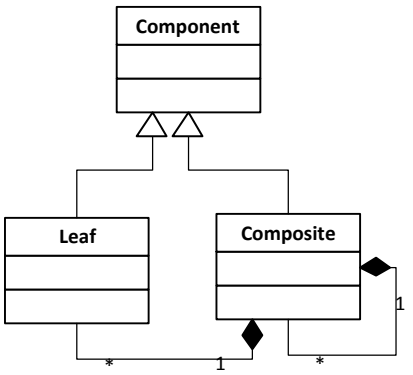
This variant describes an additional responsibility accepted by the leaf element. Indeed such an implementation respects the structural characteristics of the composite pattern. In this variation the leaf implements the composite element to hold charge of behavior. The author [32] discriminates such variant to be fit for detection and thus prepare a repository of such obligatory relationships that abuse the pattern intent. In this variant the leaf element is not a fully specialized terminal because it is depending on another element belonging to the same hierarchy.



8.9 UML Diagram of supplementary relationship variant of composite pattern [32]

8.10 Reflexive connection between leaf and composite [33]

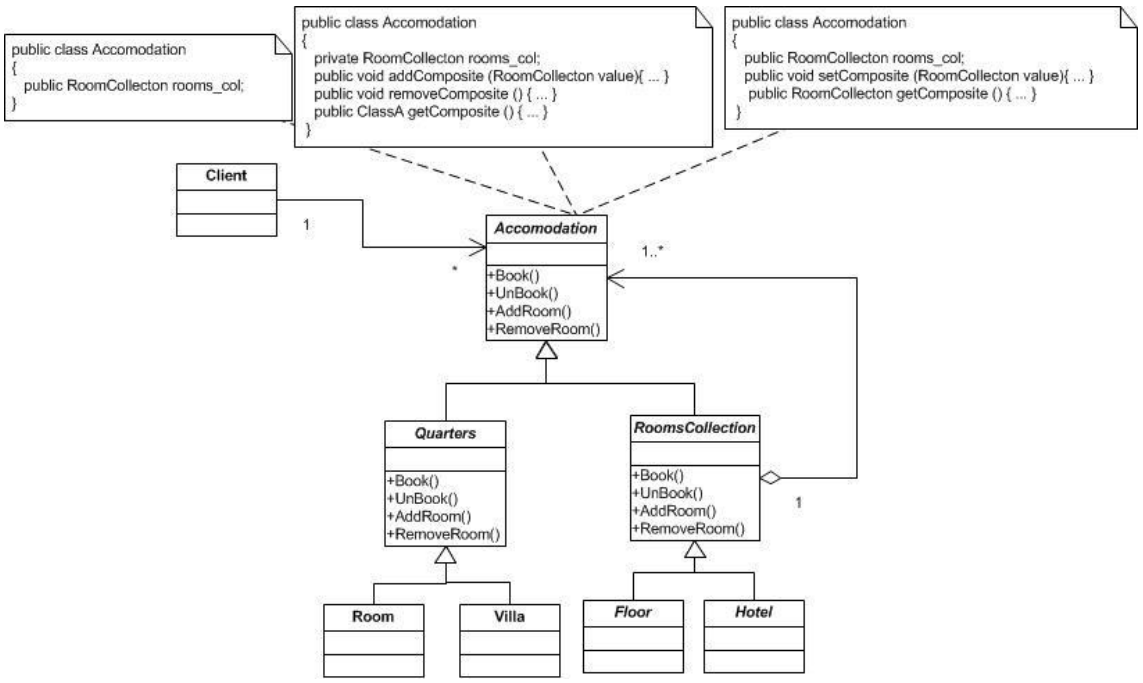
This variant defines the composition developed around the Composite class. The author [33] terms such implementation as spoiled pattern because it breaks the "decoupling and extensibility" feature of composite design pattern [33]. This variant involves Composite participant is composed of its self-reference and a terminal element.



8.10 UML Diagram of reflexive connection variant of composite pattern [33]

8.11 Association Implementations [82]

Below are some other ways to implement association. Such type of implementations focus on the Composite reference to be initialized in multiple ways



8.11 UML Diagram of association implementations of composite pattern [82]