# The Composite Pattern

Siyuan Guo (s44guo), Vegard Seim Karstang (vskarsta)
Owen Linton (oclinton), Johan Sjöberg (jewsjobe)

SpaceQuest
Group 18

March 13, 2018

## Motivation

Composite is a design pattern used to create a hierarchy of (1) *individual* and (2) *collections* of objects through a shared interface. This "unity" enables the client to treat both single objects and groups equally, without having to distinguish between the two (Freeman, Freeman, Sierra, & Bates, 2004). Since a single object and a collection of objects can be referred to by their common interface, the client doesn't have to know whether she is managing a single instance or a larger collection. Instead, she can make use of polymorphism without having to apply unwanted conditional statements that would make the code untidy and harder to read (Skrien, 2009).

To illustrate this concept, imagine your computer's file system, which contains both individual files and folders. Each file can be treated as an individual object, whereas a folder is a collection of several different files and/or sub-folders. There are situations in which you want to perform the same operation on a component `f`, whether it's a file or a folder. For example, you should be able to call `f.copy()` without having to be concerned about the specific type of `f`, since this functionality is shared by both files and folders.

## Vocabulary

**1. The Component** is the shared interface that specifies the operation(s) each component must implement. A component can be either a leaf or a composite.
**2. A Leaf** is an individual component that exist on its own. These can be combined into larger composites.
**3. A Composite** is a collection of components. It defines the group's behavior and has a reference to every child. It can include both leaves and other composites.
**4. The Client** gets access to the different components via the interface, taking advantage of polymorphism.

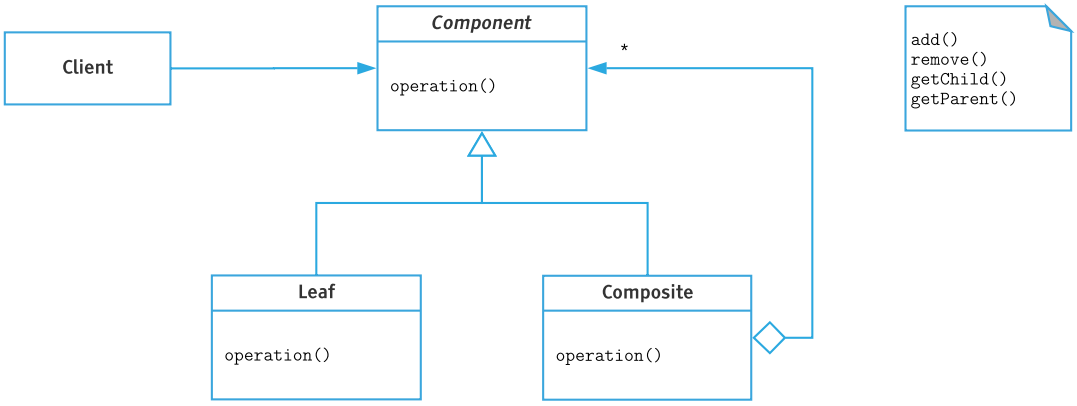


**Figure 1:** *The UML Diagram.*

In the UML diagram above, `operation()` represents the shared functionality. This method can be called on any component. The specific implementation, however, will differ between different components so to achieve the desired class-specific behavior. A call to a leaf will be direct, whereas calling a composite will most likely trigger a chain of operations, including separate calls to its children. Lastly, there may be a range of different leaves and various composites enclosing a variety of components.

## Varying Structures

Traversing the tree structure is an important aspect of the composite pattern. Four methods, relevant for traversal and especially the manipulation of composites are: `add()`, `remove()`, `getChild()` and `getParent()`. However, these methods are not usually relevant for leaves (assuming they cannot be transformed into composites). This poses the dilemma of where the implementation should take place; should these methods be listed in the component interface, or is it better to realize them only in composites? This questions touches upon the balance between transparency and safety. The decision will depend on the specific implementation (Freeman et al., 2004). Furthermore, instead of an interface, the overarching component can be implemented as an abstract class. This approach enables the implementation of default behaviors for the shared methods which can be overridden in each subclass (Gamma, Vlissides, Johnson, & Helm, 1994). Initial questions to be answered:

**1.** *What methods should leaves and composites share?*
**2.** *Should the components implement an interface or inherit form an abstract class with default implementations?*
**3.** *How should the composites store information about their children? What data structure should be used?*

## Benefits and Restrictions

This pattern simplifies manipulation of objects since the client doesn't have to distinguish between leaves and composites. Furthermore, it arranges components

in a tree structure that can be traversed using recursion in a part-whole fashion. It imposes flexibility by allowing the client to add and/or remove new components without much effort (Freeman et al., 2004). Its applicability is often emphasized in graphical settings where larger structures are created by combining smaller components, which can be further decomposed into basic building blocks. The composite design simplifies the manipulation of graphics by only having to consider larger composites, while ignoring its subcomponents. For example, calling `refresh()` on the encompassing view will cause that command to propagate down the hierarchy, enclosing all of its subcomponents. Performance can be further improved by using caching (Gamma et al., 1994). Here are the **benefits** summarized:

**1.** *Simplifies the client's use of single and groups of objects, allowing the same operations to be performed on both.*
**2.** *Applicable when components can be organized in tree structures, such as graphical views.*
**3.** *Allows flexible manipulation of the hierarchy and its components.*
**4.** *Can perform efficient traversal using caching.*

However, one might argue that the combination of traversal operations and other calculation methods violates the *single responsibility principle*, implying that the iteration and calculation should be handled separately. Furthermore, Gamma et al. (1994) argues that the flexibility imposed by this pattern may result in undesirable options. For example, there are cases when it is desirable to restrict a composite's children to certain types (e.g., a bike includes a handlebar and two pedals, but not a steering wheel and a throttle). Lastly, as discussed above, the client must consider the balance between transparency and security. Adding methods shared by composites (but not by leaves) to the interface or abstract class will violate the *interface segregation principle*, stating that no component should depend on methods it doesn't utilize. Here are the **restrictions** summarized:

**1.** *The combination of traversal and separate calculation methods violates the single responsibility principle.*
**2.** *The benefit of flexible addition and removal imposes a higher risk of intricate hierarchies and illegal composites.*
**3.** *Forces the client to make a trade-off between transparency and security.*

## Non-Functional Properties

Based on the discussion above, we can conclude that the composite pattern supports and inhibits the following non-functional properties:

**Benefits:** *Scalability, Efficiency and Evolvability*
**Issues:** *Complexity and Security*

## Presentation Description

The presentation illustrates how a client can use separate components (both leaves and composites) to draw two different objects: a bike and a car.

**Scenario 1:** The client wants to draw a bike and a car. The client creates several components, composites and adds leaves. The client calls `draw()` to draw the bike. The client then removes the component that draws the bike frame and replaces it with a component that can draw a car frame. The client also removes the bike wheels from the composite and adds some fancy wheels. However when adding the leaves and components that draws these wheels the client makes a mistake and creates a loop. The client calls `draw()` again to draw the car, but because of the loop the car is not drawn properly. The client removes the loop and calls `draw()` again to draw the car with the fancy wheels. This scenario shows how the system can easily evolve and scale, but that the structure may quickly become very complex which can cause issues.

**Scenario 2:** In the second scenario, the client updates the wheels of the component. Thus, the old leaf is replaced with a new version. To accomplish the drawing of the second wheel, the client will give the composite a reference to itself; however, this causes an infinite recursion. This scenario shows that the patterns flexibility may lead to complex hierarchy structures with components having children of undesired type.

**Scenario 3:** The last scenario shows how a call (different from drawing) to the main component propagates down the hierarchy. The client wants to make the car bigger so she calls `resize(2)` on the main component. This composites forward the message to its leaves and each leaf sets its size to twice the current size.

## References


Freeman, E., Freeman, E., Sierra, K., & Bates, B. (2004). *Head First Design Pattern*. O'Reilly.

Gamma, E., Vlissides, J., Johnson, R., & Helm, R. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Skrien, D. (2009). *Object-Oriented Design Using Java*. McGraw-Hill.