Composite Design Pattern

Purpose and Motivation

The Composite design pattern is a partitioning design pattern and describes a group of objects that are treated the same way as a single instance of the same type of object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. It allows you to have a tree structure and ask each node in the tree structure to perform a task. The key concept is that you can manipulate a single instance of the object just as you would manipulate a group of them. The operations you can perform on all the composite objects often have the least common denominator relationship.

Intended Use Cases

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly, which greatly simplifies clients and makes them easier to implement, change, test, and reuse.

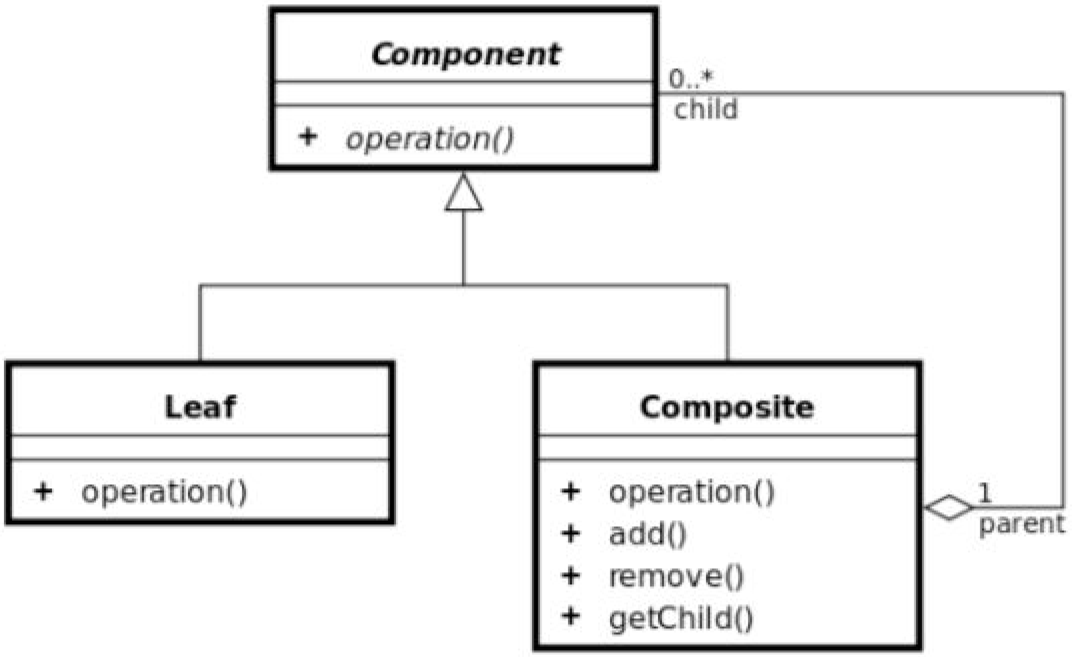
Vocabulary

Client: The Client manipulates the objects in the composition through the component interface. It is only aware of Leafs and Composites through the Component Interface.
Component: Component declares the interface for objects in the composition and for accessing and managing its child components. It also implements default behaviour for the interface common to all classes as appropriate.
Leaf: Leaf defines behaviour for primitive objects in the composition. It represents leaf objects in the composition.
Composite: A Component that can contain other components. It stores child components and implements child-related operations in the component interface.

Structure and Runtime Behaviour

The Figure below shows the class structure for the Composite design pattern. Both Leaf and Composite inherit from Component. The difference is that Leaf, which represents a leaf node in the tree structure, cannot contain other Component. The Composite class can have an array of Component objects. Thus, Composite has more methods related to array operations, such as

add(), remove(), and getChild(), which are not supported in Leaf. Usually, the operation method of a Composite object will operate all its children in its array or list separately.
The critical point in the Composite design pattern is that it can have a mixed of Leaf and Composite. The children of a Composite can also be mixed. In a Client perspective, the client declares a Component object, without caring whether it is a Leaf or a Composite. When the client uses it and calls the operation method, the object uses the correct method accordingly.



Known Consequences

Positive Consequences

- Simplification of execution: Clients can treat composite structures and individual objects uniformly. Clients normally don’t know (and shouldn’t care) whether they’re dealing with a leaf or a composite component. This simplifies client code, because it avoids having to write a case statement in the functions over the classes that define the composition.
- Easy to add new kinds of components: Newly defined Composite or Leaf subclasses work automatically with existing structures and client code. Clients don’t have to be changed for new Component classes.

Negative Consequences

- General Design: The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

NFPs

Improved NFPS

- Adaptability: the composite design pattern forces containers to work with child nodes through a common interface which allows for recursive operations on the entire hierarchy. This means that as new compositions and leaves are added to the hierarchy they can adapt to the existing code.

Inhibited NFPS

- Low Complexity: All classes follow a similar interface so the client does not have to worry about class-specific code. The client can just treat primitives and composites as homogeneous classes.
- Low Coupling: The composite design pattern reduces coupling by utilizing the same interface for each of the components. Whether it is a leaf or a composite, the class does not need to know any information about the other classes. Thus each class can easily be changed in the future without affecting any of the other component classes.

Credits

Students:

Apex (Brian Chan, Umesh Dhurvas, Joel Parikh, Harsh Patel)

Bug404 (Minghua Fan, Jinshan Gu, Kaisong Huang, Linguan Yang)

Instructor and TAs:

Mei Nagappan, Arman Naeimian, Ivens Portugal