Composite Design Pattern

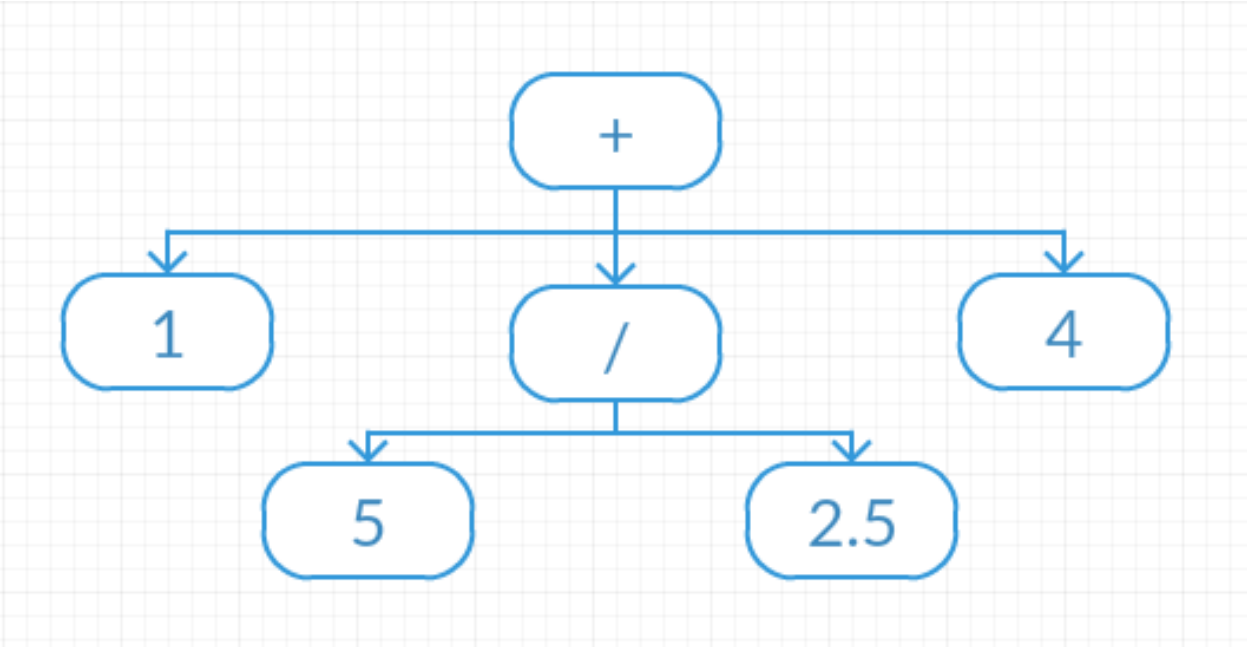
Name: Composite

Intended Use: The composite pattern is a structural design pattern. Specifically the composite design pattern is used to create recursive hierarchical tree structures where clients don't need to distinguish between the nodes and leaves of the tree. The key to the composite pattern is the abstract class that represents both the nodes and leaves of the tree, and thus enables them to be treated identically.

Examples: Any recursive tree structure easily lends itself to the composite pattern. Examples include arithmetic expressions, parse trees, and graphical hierarchies to name a few.

Arithmetic Expressions: Primitives such as integers, floats, reals etc are the leaves while aggregation operations such as addition, subtraction, negation represent the nodes.

For example the arithmetic expression (1 + (5 / 2.5) + 4) would have the following structure:



Parse Trees: Primitive expressions such as types and values represent the leaves and compositions of expressions such as if statements, loops, and function declarations represent the nodes.

Graphical Hierarchies: Primitives like lines and basic shapes represent the leaves, compositions of these primitives like pictures and transformations represent the nodes of the hierarchy.

Vocabulary:

Component: The abstract class that defines the interface for objects in the hierarchy. It is responsible for implementing default behaviour common to all classes as well as defining an interface for accessing and managing children components.

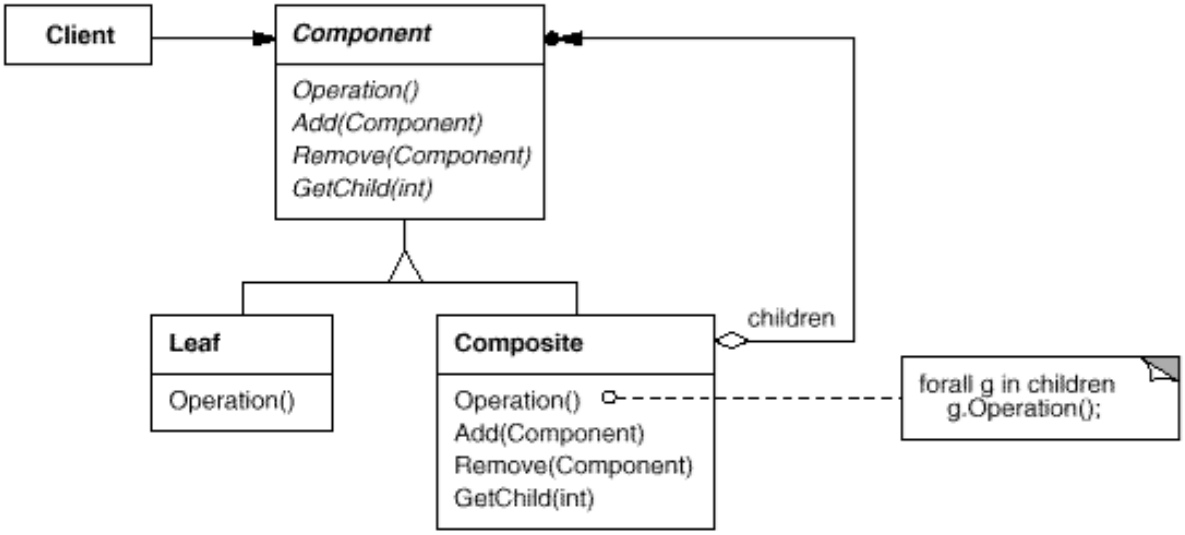
Primitive / Leaf: Represent the leaf nodes of the tree and hence have no children. These are responsible for defining the behaviour of primitive objects in the composition.

Composite / Node: Represent the internal nodes of the trees that have children. Are responsible for implementing child-related functions from the Component class as well as defining behaviour for compositions of other components.

Client: Anything that will interact with the hierarchy through the component interface such as algorithms and functions expecting compositions as input.

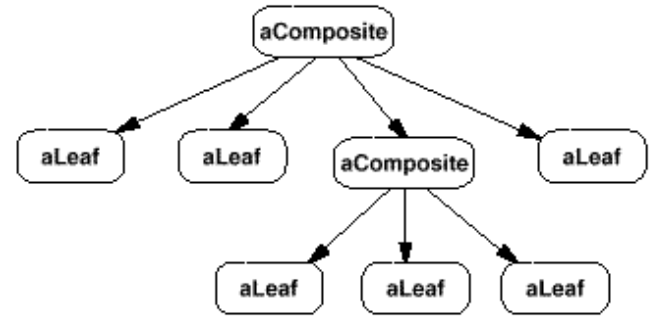
Structure:

The class diagram of the component pattern follows:



Composite Design Pattern Structure (Gamma, Helm, Johnson, & Vissides, 2016)

An example of the runtime structure of the composite pattern looks like:



Runtime Composition Hierarchy (Gamma et al., 2016)

Consequences:

Because primitives and compositions of objects follow the same abstract interface any client code that expects a primitive may also take a composition. This makes client code simple. They don't need to write specific code for compositions or primitives separately as both appear to the client to be the same.

The abstract component interface also makes it easy to add additional composite and primitive classes. As long as these new classes follow the existing interface they will work with any existing client code without the client code needing to change.

A downside to the composite pattern is that because all classes in the hierarchy must follow the abstract interface it can lead to overly general classes. For example you cannot restrict the children of a certain composition class at compile time and instead must rely on runtime checks to achieve this behaviour.

NFPs:

Adaptability and Evolvability: As mentioned above the composite pattern makes it easy to add new composition and primitive classes to the component structure thus making it easy to adapt and evolve existing code to handle new capabilities.

Complexity: The composite pattern allows for simple client code that doesn't require class specific code since all classes in the structure follow the same interface.

High Cohesion: Each class in the composition accomplishes a single purpose. In the case of primitives this would be the primitive behaviour they define, and in the case of compositions this would be the aggregation of child components. As such there is high cohesion within each class in the hierarchy.

Low Coupling: Since each class in the composition follows the same interface the classes don't know the internal implementations of other classes. This results in very low coupling between classes making it easy to change the implementation of single class without affecting the implementation of other classes in the composition structure.

Related Patterns:

Decorator: The component pattern can used in conjunction with the decorator pattern. When used together the abstract class for both the composite pattern and decorator pattern is combined into a single parent class for the combo-pattern.

Flyweight: The flyweight pattern can be useful to help reduce memory concerns by allowing multiple compositions to share the same primitives.

Iterator: The iterator pattern can be a useful pattern for traversing the component hierarchy once it is created.

Visitor: The visitor pattern can be used to localize behaviour to components that would otherwise be spread across composition and leaf classes.

References:

Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (2016). Design patterns: Elements of Reusable Object-Oriented Software. Boston, MA: Addison-Wesley.