# Composite

Effective representation of whole/part hierarchies

Many programs use trees to represent whole/part hierarchies. Leaf nodes represent atomic parts. Interior nodes represent compositions (or groups) of lower-level parts. A composite may be a child of a higher-level composite.

Examples: Drawing editor (demo), XML tree (XML.png), Org chart (draw on board)

Each type of node is represented by a separate class. Interior nodes and leaf nodes have some fundamental differences (interior nodes can have children, leaf nodes can't; leaf nodes support operations that composites don't).

Directly modeling these differences leads to a design where interior nodes and leaf nodes have different interfaces. However, such a design requires that algorithms that traverse a tree and process its nodes must contain special case code that distinguishes between different types of nodes.

EXAMPLE: composite1

Client algorithms can be greatly simplified if all nodes implement a common interface, thus removing the need for special-case branching.

EXAMPLE: composite2

Examples: Drawing editor (class diagram), Org chart (class diagram)

Generic Composite class diagram

Some operations don't make sense for leaf nodes. The options are:
1) Don't support some operations on leaf nodes (violates interface contract)
    a. composite2
2) Move operations that leaves can't support out of the common interface down to a composite superclass
    a. composite3

Advantages:
1) Simplifies algorithms that process the tree
2) Very flexible - any type of node can be a child of any type of node
3) Easy to add new kinds of nodes (e.g., adding PurpleNode would not affect algorithms that strictly use the common node interface)

Disadvantages:
1) Specific node interfaces are more descriptive

2) With a generic, one-size-fits-all interface you lose the compile-time type safety that comes with custom node interfaces. You must be careful to not create parent/child relationships that are illegal (yet allowed by the compiler).