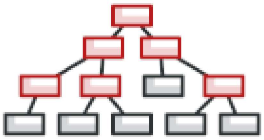

Composite in C++

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

Composite became a pretty popular solution for the most problems that require building a tree structure. Composite's great feature is the ability to run methods recursively over the whole tree structure and sum up the results.

Learn more about Composite

Complexity:

Popularity:

Usage examples: The Composite pattern is pretty common in C++ code. It's often used to represent hierarchies of user interface components or the code that works with graphs.

Identification: If you have an object tree, and each object of a tree is a part of the same class hierarchy, this is most likely a composite. If methods of these classes delegate the work to child objects of the tree and do it via the base class/interface of the hierarchy, this is definitely a composite.

Conceptual Example

This example illustrates the structure of the **Composite** design pattern. It focuses on answering these questions:

- What classes does it consist of?
- What roles do these classes play?
- In what way the elements of the pattern are related?

main.cc: Conceptual example

```cpp
#include <algorithm>
#include <iostream>
#include <list>
#include <string>
/**
 * The base Component class declares common operations for both simple and
 * complex objects of a composition.
 */
class Component {
  /**
   * @var Component
   */
 protected:
  Component *parent_;
  /**
   * Optionally, the base Component can declare an interface for setting and
   * accessing a parent of the component in a tree structure. It can also
   * provide some default implementation for these methods.
   */
 public:
  virtual ~Component() {}
  void SetParent(Component *parent) {
    this->parent_ = parent;
  }
  Component *GetParent() const {
    return this->parent_;
  }
  /**
   * In some cases, it would be beneficial to define the child-management
   * operations right in the base Component class. This way, you won't need to
   * expose any concrete component classes to the client code, even during the
   * object tree assembly. The downside is that these methods will be empty for
   * the leaf-level components.
   */
  virtual void Add(Component *component) {}
  virtual void Remove(Component *component) {}
  /**
   * You can provide a method that lets the client code figure out whether a
   * component can bear children.
   */
  virtual bool IsComposite() const {
    return false;
  }
  /**
   * The base Component may implement some default behavior or leave it to
   * concrete classes (by declaring the method containing the behavior as
   * "abstract").
   */
  virtual std::string Operation() const = 0;
};
/**
```

```
 * The Leaf class represents the end objects of a composition. A leaf can't have
 * any children.
 *
 * Usually, it's the Leaf objects that do the actual work, whereas Composite
 * objects only delegate to their sub-components.
 */
class Leaf : public Component {
 public:
  std::string Operation() const override {
    return "Leaf";
  }
};
/**
 * The Composite class represents the complex components that may have children.
 * Usually, the Composite objects delegate the actual work to their children and
 * then "sum-up" the result.
 */
class Composite : public Component {
  /**
   * @var \SplObjectStorage
   */
 protected:
  std::list<Component *> children_;

 public:
  /**
   * A composite object can add or remove other components (both simple or
   * complex) to or from its child list.
   */
  void Add(Component *component) override {
    this->children_.push_back(component);
    component->SetParent(this);
  }
  /**
   * Have in mind that this method removes the pointer to the list but doesn't
   * frees the
   *     memory, you should do it manually or better use smart pointers.
   */
  void Remove(Component *component) override {
    children_.remove(component);
    component->SetParent(nullptr);
  }
  bool IsComposite() const override {
    return true;
  }
  /**
   * The Composite executes its primary logic in a particular way. It traverses
   * recursively through all its children, collecting and summing their results.
   * Since the composite's children pass these calls to their children and so
   * forth, the whole object tree is traversed as a result.
   */
  std::string Operation() const override {
```

```cpp
    std::string result;
    for (const Component *c : children_) {
      if (c == children_.back()) {
        result += c->Operation();
      } else {
        result += c->Operation() + "+";
      }
    }
    return "Branch(" + result + ")";
  }
};
/**
 * The client code works with all of the components via the base interface.
 */
void ClientCode(Component *component) {
  // ...
  std::cout << "RESULT: " << component->Operation();
  // ...
}

/**
 * Thanks to the fact that the child-management operations are declared in the
 * base Component class, the client code can work with any component, simple or
 * complex, without depending on their concrete classes.
 */
void ClientCode2(Component *component1, Component *component2) {
  // ...
  if (component1->IsComposite()) {
    component1->Add(component2);
  }
  std::cout << "RESULT: " << component1->Operation();
  // ...
}

/**
 * This way the client code can support the simple leaf components...
 */

int main() {
  Component *simple = new Leaf;
  std::cout << "Client: I've got a simple component:\n";
  ClientCode(simple);
  std::cout << "\n\n";
  /**
   * ...as well as the complex composites.
   */

  Component *tree = new Composite;
  Component *branch1 = new Composite;

  Component *leaf_1 = new Leaf;
  Component *leaf_2 = new Leaf;
```

```
  Component *leaf_3 = new Leaf;
  branch1->Add(leaf_1);
  branch1->Add(leaf_2);
  Component *branch2 = new Composite;
  branch2->Add(leaf_3);
  tree->Add(branch1);
  tree->Add(branch2);
  std::cout << "Client: Now I've got a composite tree:\n";
  ClientCode(tree);
  std::cout << "\n\n";

  std::cout << "Client: I don't need to check the components classes even when managing the t
  ClientCode2(tree, simple);
  std::cout << "\n";

  delete simple;
  delete tree;
  delete branch1;
  delete branch2;
  delete leaf_1;
  delete leaf_2;
  delete leaf_3;

  return 0;
}
```

Output.txt: Execution result

```
Client: I've got a simple component:
RESULT: Leaf

Client: Now I've got a composite tree:
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf))

Client: I don't need to check the components classes even when managing the tree:
RESULT: Branch(Branch(Leaf+Leaf)+Branch(Leaf)+Leaf)
```