Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 11: Composite Design Pattern

Contents

11.1 Introduction

This lecture note will explain the structure of the composite design pattern. The composite design pattern effectively builds a structure of related objects in the form of a tree. These objects can be traversed by making using of the run-time polymorphic properties of object oriented programming. To successfully implement the composite design pattern design issues are considered and the clearing of the entire structure from memory after it goes out of scope. In order to address these issues the C++ Standard Template Library is introduced.

11.2 Programming Preliminaries

11.2.1 C++ Standard Template Library

The Standard Template Library (STL) for C++ is a generic library comprising of containers, algorithms, iterators and functors. The library relies heavily on compile-time polymorphism and therefore template programming constructs for the implementation of data structures and algorithms for multiple object types [4, 1].

Containers are data structures that contain other classes. The classes contained by the containers must at least be copy constructible and assignable. In order to apply algorithms on the containers and to a lesser extent iterators, the classes contained must also be less-than comparable. Stated in other words, the class needs to define a copy constructor, assignment operator and a less than operator. Muldner [3] refer to objects that are canonically constructible, meaning that they have a constructor, copy constructor, assignment operator and destructor defined in them. A good principle to follow when using the STL is to ensure that all objects are canonically constructible and that they are further less-than comparable (have a less-than operator defined).

Containers are classified as either sequence or associative in nature. Every element in a sequence container has either a predecessor and/or a successor with the exception of the first and last element. Examples of sequence containers are: `vector`s, `deque`s and `list`s. Associative containers, as their name refers to, exhibit a relationship between the elements of the container. This relationship may be in terms of another element or as a result of a "key" defined to access the object. Examples of associative containers are: `set`s and `multiset`s, and `map`s and `multimap`s. Other containers exist that make use of the above containers as base but provide a different interface.

In order to traverse containers, the STL provides iterators for the containers. These iterators are built on the principles that will be seen during the discussion of the iterator design pattern. The STL further provides algorithms, such as searching and sorting that along with the iterators can be applied to the containers. Functors or function pointers are pointers to functions and provide the functionality that a function may be sent as a parameter to another function.

In the section that follows two sequence containers in the STL will briefly be discussed. Choosing between the two is dependent on the application for which they are being used. A vector is similar to an array, but will dynamically change its size when elements are inserted or deleted. Elements are inserted at the back of the vector and removed from

Members Category	signature	Vector	List
Construction	constructor	y	y
	destructor	y	y
	operator=	y	y
Accessor	size	y	y
	empty	y	y
	front	y	y
	back	y	y
	operator[]	y	n
	at	y	n
Mutator	push_front	n	y
	pop_front	n	y
	push_back	y	y
	pop_back	y	y
	clear	y	y
Iterator	begin		
	end		

Table 1: Summary of vector and list members

the back as well. A list is optmised for insertion, deletion and moving elements around in the containers. Elements in a list are inserted or deleted either at the front or the back.

11.2.1.1 Summary of Vector and List container members

A summary of the members of the STL vector and STL list implementations are given in table 1. The members are categorised in the following categories: Construction/Destruction, Accessor, Mutator and Iterator. For each of these categories an indication of whether a specific member is implemented for the container or not is given.

Other members also exist for each of the STL containers. Further details regarding the containers are beyond the scope of this lecture note and can be found by searching the internet. There are also good examples regarding the use of the containers available on the internet.

11.2.1.2 Application to previous design patterns

Lists and vectors can easily be used by the caretaker of the Memento pattern in order to store the state of more than one object. They can also be effective be applied to the Abstract Factory example given in Chapter 7 so that all memory is cleared and the example presented in the lecture notes does not exhibit a memory leak.

11.2.2 Anonymous objects

In order to understand the concepts better, some C++ background regarding anonymous objects needs to be explained. An anonymous object is an object for which the code creating the object has not got a handle to the object. Consider the following example:

Listing 1: Example of anonymous object creation

```
1  class A {
2    // all the class stuff
3  };
4
5  class B {
6    public:
7      B(A* in) { a = in; };
8    private:
9      A* a;
10 };
11
12 // Some client code
13  B b(new A());
```

The object created and sent as a parameter to the constructor of class B is anonymous. The client has no way of accessing this object and therefor when it goes out of scope the object is not cleared from memory resulting in a memory leak.

The problem comes down to who takes ownership of the object. If the client keeps ownership, then the client must be able to acquire a handle to the object in order to be able to delete it. This can be achieved by defining a variable for the object `a` as follows and replacing line 13 in listing 1 with the following code:

```
A* a = new A();
B b(a);
```

Another possibility for the client to gain ownership is if class B defines a getter operation that returns a pointer to the object. For both these scenarios, it is now the obligation of the main program to delete the object from the heap before the program goes out of scope. As object `b` is on the stack, it will go out of scope when the main program terminates. The following code will ensure that object `a` is delete from the heap.

```
delete a;
```

Assuming that the design decision made for listing 1 is that ownership of the object is given to class `B`. This means that when object `b` goes out of scope the object of class `A` it points to must all be destroyed. Class `B` must therefore implement a destructor that provides this functionality as the default destructor does not remove objects from the heap memory by including the following code after line 7 of listing 1.

```
virtual ~B() { delete a; };
```

The destructor has been defined as `virtual` to ensure that if another class inherits from `B`, the destructor is still called when that class goes out of scope.

11.3 Composite Design Pattern

11.3.1 Identification

Name	Classification	Strategy
Composite	Structural	Delegation (Object)
Intent		
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. ([2]:163)		

11.3.2 Structure

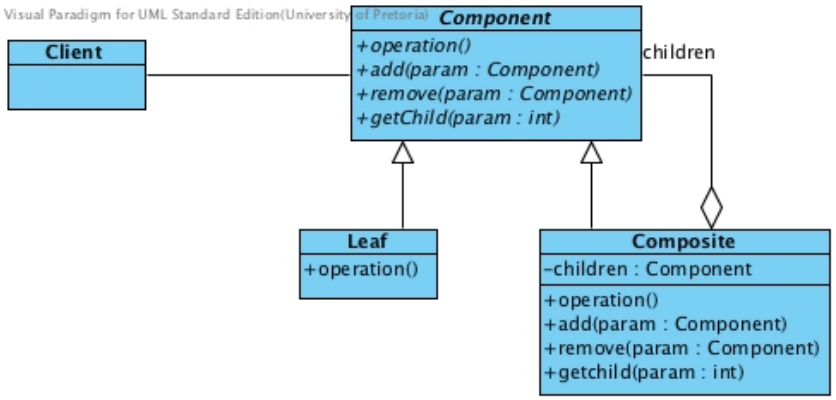

Figure 1: The structure of the Composite Pattern

11.3.3 Problem

Used in hierarchies where some objects are composite of other. Makes use of a store for the children defined by Composite

11.3.4 Participants

Component

- provides the interface with which the client interacts.

Leaf

- do not have children, define the primitive objects of the composition.

Composite

- contains children that are either composites or leaves.

Client

- manipulates the objects that comprise the composite.

11.4 Composite Pattern Explained

The composite pattern inherently builds a tree (refer to Figure 2) with the intermediate nodes being instances of composite and the leaf nodes being instances of the leaf participants.

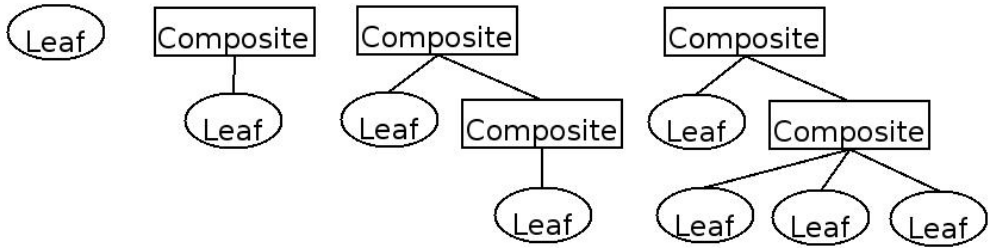

Figure 2: Examples of tree structures

11.4.1 Implementation Issues

During the design of the system a decision must be made regarding the destruction of the composite. This decision will influence how the client may construct objects and pass them to the composite. There are two options with regards to destruction of the composite, the first requires the client to take responsibility. The second destruction design leaves the responsibility of the destruction of objects up to the composite. Both these implementations have their advantages and drawbacks. If the client takes responsibility for the destruction of objects then no anonymous objects may be created. On the other hand, having the composite handle the destruction of objects will allow the client to pass anonymous objects to the composite. In this case the client will need to guarantee that it will not use any objects for which it still has handles after it has destructed the composite.

11.4.2 Related Patterns

Chain of Responsibility
Creates a structure that defines a component-parent link.

Decorator
Used in conjunction with components to add state to the components. When a decorator and a composite are combined, they usually share the same parent class.

Flyweight
Allows sharing of objects, particularly the leaf nodes

Iterator and Visitor
Used to traverse the composite structure.

11.5 Example

11.5.1 Tree

In this example a tree will be built in which each node of the tree is represented by an integer. The UML class diagram is given in Figure 3. The abstract `Tree` class defines operations that can be applied to the tree. In this case a Tree node can be added to the tree or the node can be printed.

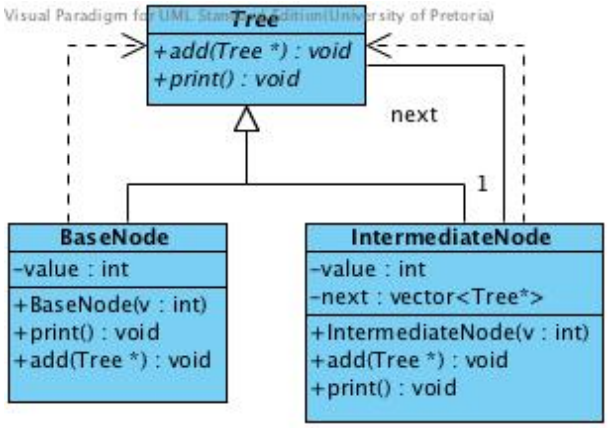

Figure 3: Tree example

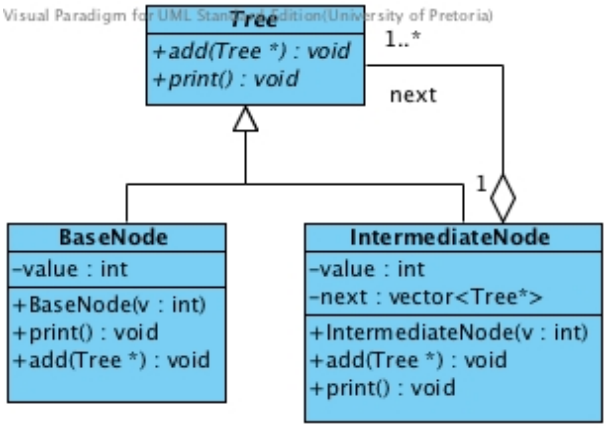

Figure 4: Tree example updated

When the association relationship between the composite participant `IntermediateNode` and the component participant `Tree` is drawn in Visual Paradigm, it is drawn as both a dependency and an association. The dependency is as a result of a `Tree*` being sent as a parameter to `add`, which also accounts for the dependency between `BaseNode` and `Tree`. The association is as a result of the vector `next` of `Tree*`. According to the structure of the pattern given in Figure 1 this association should be aggregation and it is not necessary to include the dependencies. The class diagram given in Figure 4 is more in line with the intended structure of the pattern and needs to be manually updated in Visual Paradigm.

Both the examples in Figures 3 and 4 require the client to take responsibility for the deletion of the objects, unfortunately this can lead to memory leaks. In the example in Figure 5, the deletion of the objects in the hierarchy is the responsibility of the composite.

The implementation of the destructor of the `Tree` class is empty as is that of `BaseNode`. The destructor of `IntermediateNode` must iterate through the `next` vector and delete each element is points to. The implementation of the `IntermediateNode` destructor, along with the implementations of all the other operations of the `IntermediateNode` class shown in Figure 5, is given in the listing below. This code illustrates how elements are inserted into the vector container as well as how to iterate through the container.

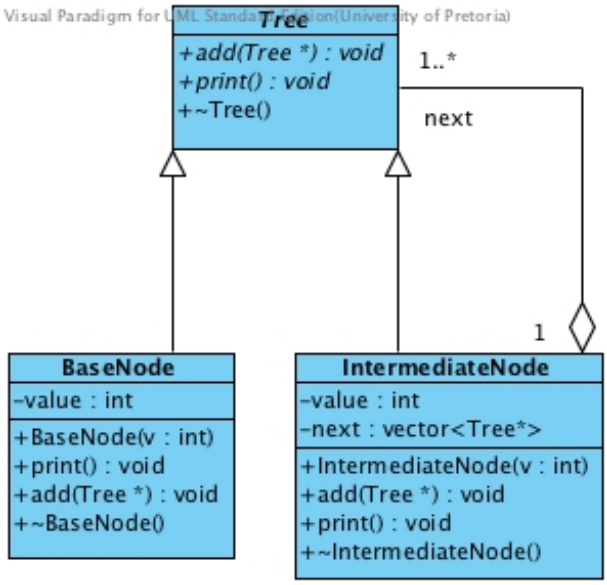

Figure 5: Tree example with the composite taking ownership of destruction

```
class IntermediateNode : public Tree
{
  public:
        IntermediateNode(int v);
        virtual void add(Tree*);
        virtual void print();
        virtual ~IntermediateNode();
  private:
        int value;
        vector<Tree*> next;
};

IntermediateNode::IntermediateNode(int v) : value(v)
{
}

void IntermediateNode::add(Tree* t)
{
        next.push_back(t);
}

void IntermediateNode::print()
{
        cout << "-" << value << "[";
        vector<Tree*>:: iterator it;

        for (it = next.begin(); it != next.end(); ++it)
        {
                (*it)->print();
```

```
        }
        cout << "]";
}


IntermediateNode::~IntermediateNode()
{
        vector<Tree*>:: iterator it;

        for (it = next.begin(); it != next.end(); ++it)
        {
                delete *it;
        }
}
```

11.5.2 Graphics

Consider the Shape Hierarchy that was introduced in the Abstract Factory chapter. Figure 6 serves as a refresher for this hierarchy.

From the hierarchy it is clear that the three (3) concrete triangle classes, the part of the hierarchy from Parallelogram down, and Ellipse and its subclass Circle will form leaf nodes of the Composite design pattern. The Shape class will fulfill the role of the Component participant and a new class that represents the composite participant must be designed. For this design, the area and perimeter of the composite will be determined by calculating the sum of the area and perimeter of its constituents respectively. Figure 7 provides the class structure for the composite shape hierarchy.

11.6 Exercises

1. Refactor the Tree example given in Figure 5 so that `int value` is defined in one place only.

2. How would you modify the composite participant to build a binary tree? A binary tree is a tree where the composite participant only has two children. These children may either be an instance of composite or a leaf participant.

References

[1] cplusplus.com. STL Containers, 2011. URL `http://www.cplusplus.com/stl/`. [Online; accessed 31 August 2011].

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[3] Tomasz Müldner. *C++ Programming with Design Patterns Revealed*. Addison Wesley, 2002.

[4] Wikipedia. Standard template library, 2011. URL `http://en.wikipedia.org/wiki/Standard_Template_Library`. [Online; accessed 31 August 2011].

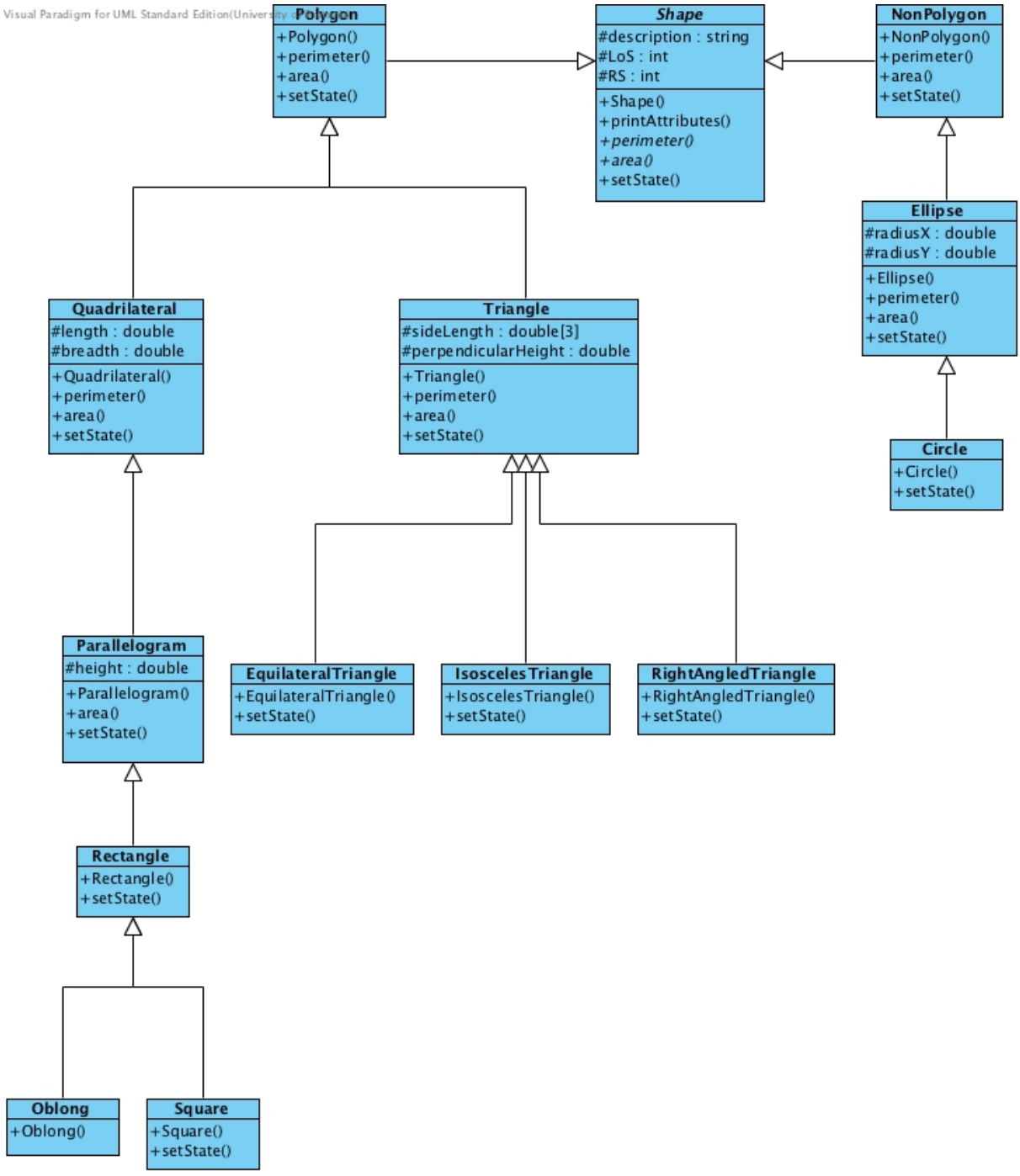

Figure 6: Shape hierarchy

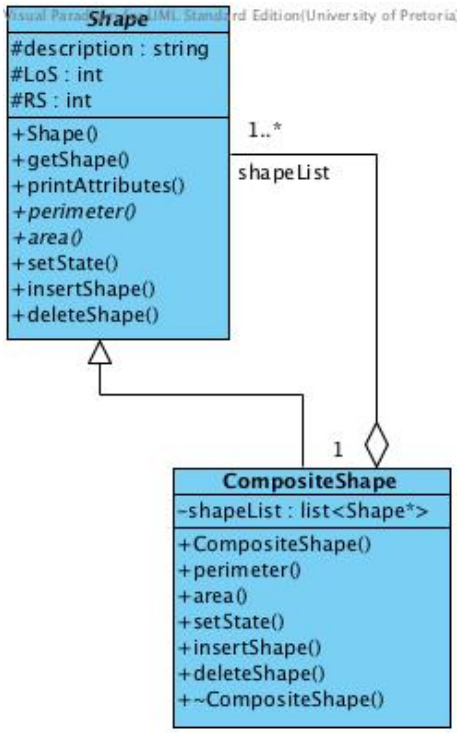

Figure 7: Composite Shape hierarchy