Composite

12

The Composite Pattern, which I will now introduce, belongs to the Structural Patterns. It describes how objects are put together to form larger meaningful units. A composite structure is great for displaying in a JTree. So, in this chapter I will also describe how to define your own data models for a JTree.

12.1 Principle of Composite

You have a structure that consists of several units. These units can optionally contain smaller units. You want to be able to call certain methods on all of these units without having to worry about whether or not the unit contains other units. What is meant by this?

If you keep a budget book, write down the individual items of your income and expenses. The income includes your salary from your main job. Maybe you also have a side job, then this is also written under the heading of income. To the expenses belongs certainly your rent or rates for an own real estate. In addition, you also have items that can be subdivided. Under the heading of food, there are various sub-categories: Lunch, Eating out with friends, etc. In these subcategories you have the individual items like "Pizzeria 16.00 €" or "Canteen 4.00 €". You can ask at the lowest level: "How much did the visit to the pizzeria cost?" But you can also add up by asking, "What did I pay in total for food?" And finally, at the top level, you can ask: "What did I take in and spend in total?"

Or think of a company that consists of different departments that are divided into several hierarchical levels; if you ask an employee there about the personnel costs, he will name his own salary. If you ask a department manager for his personnel costs, he will first determine the personnel costs of his subordinate employees, add his own salary and give

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_12

you the total sum back. If you ask the owner of the company, he will return the total personnel costs of all departments in addition to his own.

Very practical: The file system of your computer returns the size of a folder as well as of a single file.

In short, the Composite Pattern helps you represent tree structures. Let's clarify some terms! Each item in a tree structure is a node. A node that has no child nodes is a leaf; for example, this is the single item "Pizzeria 16.00 €". A node that has subnodes is called a composite; this is, for example, the item "food". The node at the top of the tree (the company owner) is the root.

There are two different approaches to implement the composite pattern.

12.2 Implementation 1: Security

Leafs and composites must have different behavior; consequently, they must be defined in different classes. However, composites can store both other composites and leafs as nodes; therefore, composites and leafs must have a common interface.

In the sample project Budget_1 you will find a first draft. The class Node is the common interface for leaves and composites. It defines a data field that describes the revenue or spending item; this field is needed for both leaves and composites. And finally, the `print()` method is prescribed.

```
public abstract class Node {
    protected final String description;
    // … abridged
    public abstract void print(int indentation);
}
```

The class Leaf extends the class Node. It additionally defines a field in which it is stored whether the position is required or whether it is luxury. For revenue, this flag certainly doesn't matter. However, you will certainly think about whether an expense was necessary or not. It also defines a data field that stores the amount of the item. The print method takes an integer value that describes the number of indentations; tabs are inserted according to this number before the value of the item is printed. Items that are required are preceded by an exclamation mark; outputs that are not strictly required are not marked separately.

```
public class Leaf extends Node {
    private final boolean required;
    private double amount = 0.0;
    // … abridged
    @Override
    public void print(int indentation) {
        for (var i = 0; i < indentation; i++)
```

```
            System.out.print("\t");
        System.out.println(this);
    }

    @Override
    public String toString() {
        var prefix = required ? "(!) " : "( ) ";
        var tempAmount =
       NumberFormat.getCurrencyInstance().format(amount);
        return prefix + description + ": " + tempAmount;
    }
}
```

The Composite class defines a list in which the child nodes are stored and the required access methods. This includes, for example, a method that returns the number of child nodes and a method that returns the child at a specified position. The print method is overridden in such a way that, according to the indentation, first the value of the toString method is output and then recursively all child nodes.

```
public class Composite extends Node {
    private final List<node> kinder = new ArrayList<>();
    // … abridged
    public node getKind(int index) {
        return children.get(index);
    }

    public int getNumberofChildNodes() {
        return kinder.size();
    }

    @Override
    public void print(int indentation) {
        for (var i = 0; i < indentation; i++)
            System.out.print("\t");
        System.out.println(this);
        children.forEach((node) -> {
            node.print(indentation + 1);
        });
    }

    @Override
    public String toString() {
        return description;
    }
}
```

The client creates variables for the root and various expense categories, for example, income and expenses. Among the expenses, there is the category books, where two books are set. I am only printing the listing in abbreviated form.

```
final var root = new Composite(“Budget book“);
final var january = new Composite(“January“);
final var income = new Composite(“Income“);
final var expenses = new Composite(“Expenses“);
final var books = new Composite(“Books“);
january.add(income);
january.add(expenses);
root.add(january);
revenue.add(new Leaf(“Main job“, 1900.00, true));
revenue.add(new Leaf(“Side job“, 200.00, true));
expenses.add(books);
expenses.add(new Leaf(“rent“, -600.00, true));
books.add(new Leaf(“Design Patterns“, -29.9, true));
books.add(new Leaf(“trashy novel“, -9.99, false));
root.print(0);
System.out.println(“Only the expenses: “);
outputs.print(0);
```

The client can now restrict itself to calling `root.print(0)`. Then the income and expenses for January are clearly output to the console:

```
Budget book
   January
        Income
             (!) Main job: 1.900,00 €
             (!) Side job: 200,00 €
        Expenses
             Insurances
                  (!) Car: -50,00 €
                  (!) ADB: -100,00 €
             Books
                  (!) Design Patterns: -29,90 €
                  ( ) Trashy novel: -9,99 €
             (!) Rent: -600,00 €
```

It is possible to have only the January expenses printed; to do this, call the print command on the expenses composite: `expenses.print(0)`. This solution works perfectly. However, it proves to be too inflexible in certain places – and you will see and correct this in the following section.

12.3 Implementation 2: Transparency

The client code violates the principle that one should program against interfaces and not against implementations: `final composite root = new composite("Budget book")`. This was unavoidable because the interface node declares only those methods that are actually needed by both leaves and composites. This "lowest common denominator" does not include `add()`. And this is where a huge problem starts to mature – the program from the previous approach could only be made to work with a little "trick". The print method calls itself recursively on all objects. However, if you wanted to access the list with an external iterator or have a single item returned, you would have to work with numerous comparisons (`instanceof composites`) and downcasts. However, downcasts are the programmer's crowbar; if possible, they should be avoided.

To demonstrate the problem, I want to print the list to the console again, but this time using an external iterator. I delete the print method in the sample project Budget_2. Now the client itself has to take care of the iteration. So, the most important change in this project is in the test routine area. First, create some income and expenses as in the previous example. Then, within the test class, call a newly defined print method. In this method, first the required tabs are printed and then the object itself. It then checks whether the item to be printed is a composite. If so, the node object is cast to a composite object. And finally, ask the composite for the number of its child nodes and output each child node to the console.

```
public static void main(String[] args) {
    final var root = new Composite("Budget book");
    final var january = new Composite("January");
    final var february = new Composite("February");
    // … abridged
    print(root, 0);
}

private static void print(node node, int indentation) {
    for (var i = 0; i < indentation; i++)
        System.out.print("\t");
    System.out.println(node);

    if (node instanceof Composite composite) {
    var numberChildren = composite.getNumberChildNodes();
        for (var j = 0; j < numberChildren; j++) {
            var childNode = composite.getChild(j);
            print(childNode, indentation + 1);
        }
    }
}
```

Do you think the example is a little contrived? Not at all! You're about to see an implementation of the TreeModel interface – where it would be difficult to implement with an internal driver.

Do you notice anything about the instanceof command? Here I used a preview feature newly added in Java 14: Pattern Matching for instanceof, found under the Java Enhancement Proposal JEP 305. You can only use it if you enable preview features in Java 14 or 15 (JEP 375). This is done by the additional compiler option –enable-preview, which you have to add in the compiling project settings of NetBeans or just in the command line call of the Java compiler javac. In Java 16 (JEP 394) this feature is final. But what does this pattern matching actually do?

We get the reference to the node, which can be either a leaf or a composite. If we want to act differently depending on the subclass, we check with instanceof that it belongs to the class Composite and then cast the reference "down" to that class. Previously, in Java, this was two steps: first the check, then the casting. Now it works in a single step. You can see in the code that after the previously used `if (node instanceof Composite)`, there is now another `composite.` This automatically introduces this variable and casts the node to the class composite. The additional line `Composite composite = (Composite) node;` is thus omitted completely.

Such checks are used very frequently in Java. Pattern matching simplifies them in both programming effort and readability.

The Budget_3 sample project shows how you establish generality. You define the following two methods in the class Node:

```
public abstract class Node {
    protected final String description;

    public Node(String description) {
        this.description = description;
    }

    public node getChild(int index) {
        throw new RuntimeException("No child nodes");
    }

    public int getNumberChildNodes() {
        return 0;
    }
}
```

The class Composite overrides these methods as before. What is new in this realization is that a leaf can now also specify how many children it has – namely none. If a client nevertheless tries to call a child at a specific index, a RuntimeException flies up in its face.

So the client has to make sure that the node to be printed actually has child nodes. The print method in the test class is formulated more generally.

```
private static void print(node node, int indentation) {
    for (var i = 0; i < indentation; i++)
        System.out.print("\t");
    System.out.println(node);
   for (var j = 0; j < node.getNumberChildNodes(); j++) {
        var childNode = node.getChild(j);
        print(childNode, indentation + 1);
    }
}
```

A leaf object returns 0 as the number of children. The condition of the for loop is therefore not fulfilled, a recursive call of the print method does not take place.

12.4 Consideration of the Two Approaches

The first approach relies on a narrow interface. A leaf can only do what it absolutely must be able to do. Therefore, you will find management methods for the list of child nodes only for composites. Security is the primary concern with this approach. The client has to compare and cast, but it has the guarantee that the methods it calls are executed in a way that makes sense. The AWT, for example, is based on this principle. There is the abstract class `Component`. In it, methods are defined that use both leaves and composites: Register listeners, change visibility, etc. The various controls (`Button`, `Label`, `CheckBox`, etc.) are derived from `Component`. In addition, the `Container` class inherits from `Component`. It defines methods to manage components as child nodes. Only those classes inherit from Container that must be able to accommodate other components, i.e. Frame and Panel.

The alternative is to use a broad interface: leaves and composites can in principle do anything. Swing is built on this approach. There is the class `JComponent`, which inherits from Container. All controls inherit from `JComponent`, including `JLabel`, `JButton`, etc. So they all have the method `add()`, to which you can pass a Component object. Subsequently, it is possible to store a `JPanel` in a `JLabel`, which you can check with the following code, e.g. in the Java Shell of NetBeans.

```
javax.swing.JPanel pnlTest = new javax.swing.JPanel();
javax.swing.JLabel lblTest = new javax.swing.JLabel();
lblTest.add(pnlTest);
int count = lblTest.getComponentCount();
System.out.println("Number of children in JLabel: " + count);
System.out.println("Parent of JPanel: " + pnlTest.getParent());
```

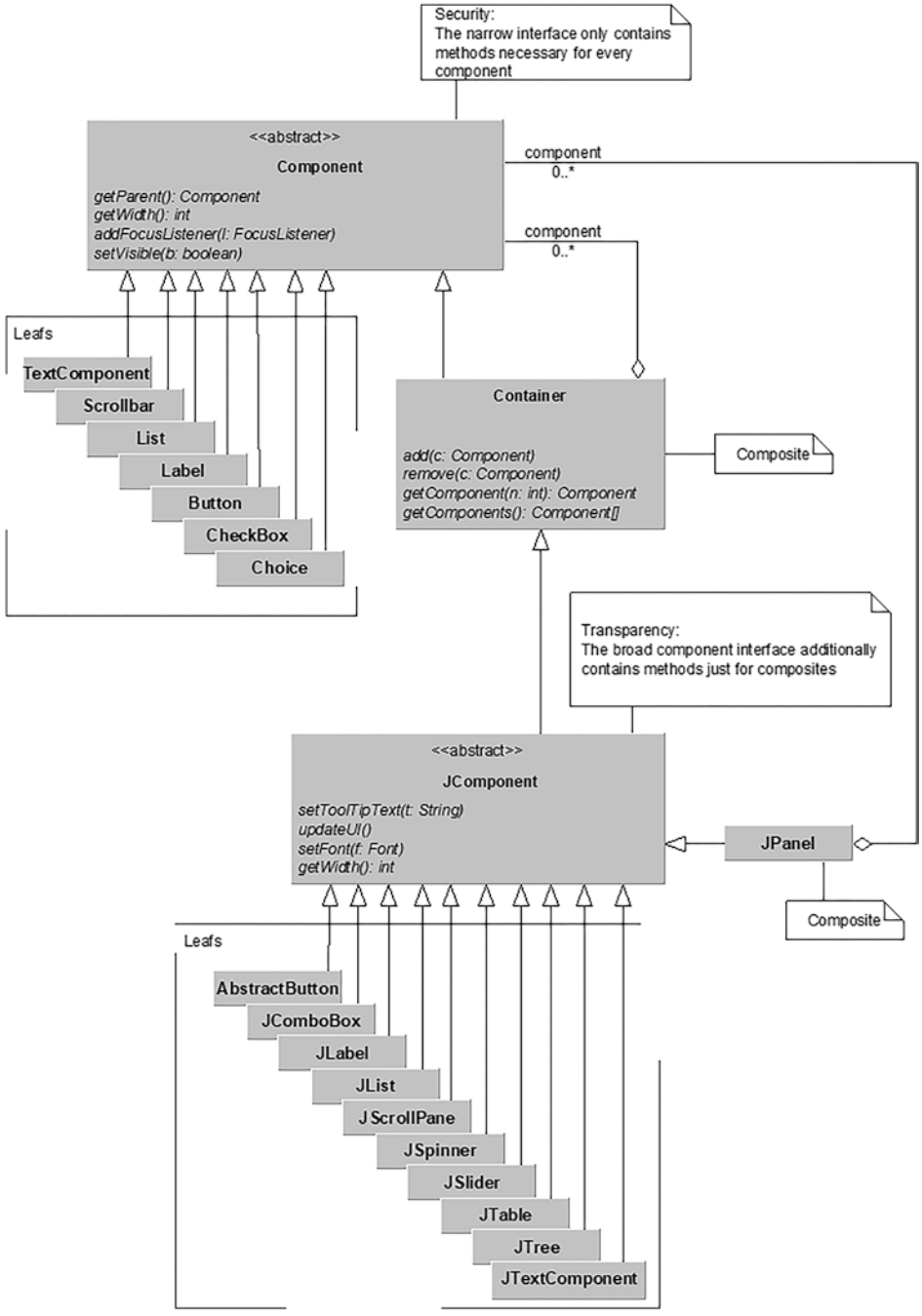

Fig. 12.1 Class diagram AWT and Swing components, borrowed from www.PhilippHauer.de

On the console – shortened – is output:

```
Number of children in JLabel: 1
Parent of the JPanel: javax.swing.JLabel...
```

The interaction of AWT and Swing is shown in Fig. 12.1. I borrowed the diagram from Philipp Hauer. Philipp deals with design patterns, among other things, on his site www.PhilippHauer.de. It is definitely worth visiting his site.

An interface that is too broad is problematic. Take project Budget_3 as a basis. There, you define methods that are specific to a composite in the interface in order to obtain generality. In the test class, you query the number of child nodes and have a child node returned. I'm sure that's not a problem. But it gets tricky when you need to include the `addChild()` method in the interface. How does the Leaf class implement this method? What does the default implementation look like? One solution might be to have the leaf simply do nothing `add(){}`. But an empty implementation is not without its problems – the client will certainly want to know that its job cannot be executed. So the client can't avoid making a case distinction up front. Although this approach can be problematic, it is favored in practice, and the GoF also advocates it.

12.5 Going One Step Further

Let's expand the project now.

12.5.1 Creating a Cache

It would be a useful thing if the categories could sum up the individual child items, and this is realized with the example project Budget_4. To do this, you create a cache in the composites that stores the sum of the child nodes. In the interface, you declare an abstract method `calculateCache()`, which must be overridden by leaves in the same way as by composites. Also, both leaves and composites should be able to return the internally stored value.

```
public abstract class Node {
    // … abridged
    abstract double getValue();
    public abstract void calculateCache();
}
```

The definition in the Leaf class is quite simple – a leaf does not need to create a cache, so the method can be overridden empty. However, a leaf must be able to name the amount stored:

```
public class Leaf extends Node {
    // … abridged
    private double amount = 0.0;

    @Override
    public void calculateCache() {
    }

    @Override
    double getValue() {
        return amount;
    }
}
```

The composite class returns the cache when the `getValue()` method is called. The `calculateCache()` method is defined so that the cache is first cleared. Then, all children are recursively instructed to calculate their cache. Finally, the stored amount is queried from all children and added to their own cache.

```
public class Composite extends Node {
    // … abridged
    private double cache = 0.0;

    @Override
    public void calculateCache() {
        cache = 0;
        for (var node : children) {
            node.calculateCache();
            cache += node.getValue();
        }
    }

    @Override
    double getValue() {
        return cache;
    }

    @Override
    public String toString() {
        var tempCache =
```

```
            NumberFormat.getCurrencyInstance().format(cache);
        return description + " (Sum: " + tempCache + ")";
    }
}
```

The test of the program differs only minimally from the previous examples. The tree is constructed as before. Then the root node receives the instruction to calculate its cache.

```
final var root = new Composite("Budget book");
// … abridged
books.add(new sheet("Design Patterns", -29.9, true));
books.add(new sheet("Trashy novel", -9.99, false));
ROOT.calculateCache();
print(ROOT, 0);
```

When you start the program, you get the subtotals of the categories.

```
Budget book (Sum: € 1,310.11)
    January (Sum: € 1,310.11)
        Income (Sum: € 2,100.00)
            (+) Main job: 1,900.00 €
            (+) Side job: 200.00 €
        Expenses (Sum: -789.89 €)
            Insurances (Sum: -150.00 €)
                (+) Car: -50.00 €
                (+) ADB: -100.00 €
            Books (Sum: -€39.89)
                (+) Design Patterns: -€29.90
                (-) Trashy novel: -€9.99
            (+) Rent: -600.00 €
    February (Sum: €0.00)
```

Since the project is very manageable, it would certainly have been reasonable for the categories to recalculate the cache each time `getValue()` is called. However, a cache makes sense if the call of `getValue()` in the leaf causes high costs.

12.5.2 Referencing the Parent Components

In the following step, the structure is created as a doubly linked list – the parents know their children, but the children also know their parents. You probably know this from

Swing – you call the method `getParent()` on a component; the parent node is returned. In the Budget_5 example project, you maintain new income or expenses; the parent category is notified and can update its cache. In the Node class, add the Composite `parent` field with the appropriate access methods.

```
public abstract class Node {
    // … abridged

    private composite parent = null;


    protected void setParent(Composite parent) {
        this. Parent = parent;
    }

    protected Composite getParent() {
        return this.parent;
    }
}
```

The `add()` method of the Composite class is extended. A composite passes itself as parent to each newly added child node. If the new child to be added already has a parent, an exception is thrown – because the situation should not occur.

```
public class Composite extends Node {
    // … abridged

    public void add(node child) {
        children.add(child);
        var parent = kind.getParent();
        if (parent == null)
            kind.setParent(this);
        else
            throw new RuntimeException(child + " already has a parent:
            " + parent);
    }
}
```

The introduction of a reference to the parent node does not yet have an effect in this project version. The parent node is used in the sample project Budget_6 to recalculate the cache. If you insert a new leaf or change the value of a revenue or expense item, all affected composites pass this information on to the parent until the message reaches the root. The root node then recursively calls the `calculateCache()` method on all child nodes, as in the previous project.

You want only the composites to recalculate their caches that are affected. You introduce a flag in the `Composite` class that indicates whether the cache needs to be recalculated. When a new node is added to a composite, the cache is no longer valid. So `add()` causes its own cache and the parent node's cache to be recomputed. The same is true when a node is removed. If the `calculateCache()` method just recalculated the cache, it is certainly correct and the flag may be set to `true`. The newly added `setCacheIsValid()` method is interesting. If `true` is passed to it, the flag is corrected. If `false` is passed to it, the parent node is additionally instructed to mark the cache as invalid. If there is no parent – this only applies to the root node – the child nodes are instructed to recalculate their cache. However, these only recalculate the cache if the flag `isValid` is set to `false`.

```
public class Composite extends Node {
    // … abridged
    private boolean cacheIsValid = false;

    public void add(node child) {
        children.add(child);
        var parent = child.getParent();
        if (parent == null)
            child.setParent(this);
        else
            throw new RuntimeException(child + " already has
            a parent: " + parent);
        this.setCacheIsValid(false);
    }

    public void remove(node child) {
        children.remove(child);
        child.setParent(null);
        this.setCacheIsValid(false);
    }

    void setCacheIsValid(boolean isValid) {
        this.cacheIsValid = isValid;
        if (!isValid) {
            var parent = this.getParent();
            if (parent == null)
                this.calculateCache();
            else
                if (this != parent)
                    parent.setCacheIsValid(isValid);
        }
    }
```

```
    @Override
    public void calculateCache() {
        if (!cacheIsValid) {
            cache = 0;
            for (var node : children) {
                node.calculateCache();
                cache += node.getValue();
            }
            this.setCacheIsValid(true);
        }
    }
}
```

I take the same test routine as in the previous project. I can do it without `root.calculateCache()` now, because the structure calculates the intermediate values "automatically" when new positions are entered. To test the new functions in more detail, I add a retirement provision to the insurances, which costs 1000.00 €. I have the budget book output again and find that the figures are correct. I notice that the retirement provision is not monthly with 1000.00 €, but annually. So, I delete the insurance and have the budget book displayed again – the figures are correct again. I insert the retirement provision again, but this time at the same level as the months, i.e., directly below the root. And again, the correct values are output. The structure of the pattern and the calculation method allow a very easy to use automatic update.

```
final var root = new Composite(“Budget book“);
// … abridged
print(root, 0);
var retirement = new Leaf(“Retirement provisions“, -1000.00, true);
insurances.add(retirement);
print(root, 0);
insurance.remove(retirement);
print(root, 0);
root.add(retirement);
print(root, 0);
```

Did you think it was a bit awkward how I moved the retirement provisions? Me too – let's correct that in the next section.

12.5.3 Moving Nodes

The sample project Budget_7 deals with the question of how to move a node to another node. Since both leaves and composites must be moved, the method `changeParent()`

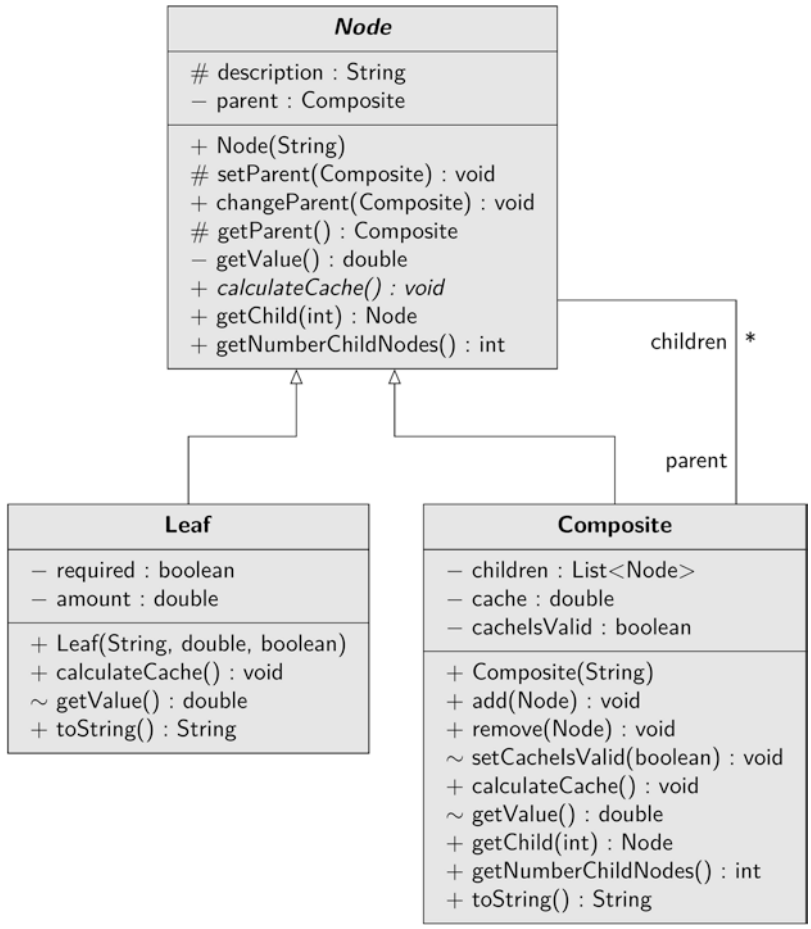

Fig. 12.2 UML diagram of the composite pattern. (Example project Budget_7)

is defined in the class `Node`, to which you pass a reference to the new parent node. The method returns the current parent and deletes itself there as the child node. With the new parent, the node enters itself as a child node. Since both `add()` and `remove()` cause the caches of the composites to be recalculated, the entire tree structure is up-to-date and correctly calculated again.

```
public void changeParent(Composite newParent) {
    var parent = this.getParent();
    parent.remove(this);
    newParent.add(this);
}
```

12.6 Composite – The UML Diagram

From the example project Budget_7 you will find the UML diagram in Fig. 12.2.

12.7 Summary

Go through the chapter again in key words:

- A composite represents hierarchical structures.
- The structures are formed from nodes (leaves and composites).
- The node that has no parent node is called the root.
- Nodes can be nested to any depth.
- Leaves and composites behave identically for the client.
- Leaves and composites inherit from a common interface.
- The width of the interface brings either security or transparency.
- An interface that is too broad will result in designs that are too general.
- Nodes can optionally hold a reference to their parent node.
- Composites can cache values of their child objects.

12.8 Description of Purpose

The Gang of Four describes the purpose of the "Composite" pattern as follows:

> Compose objects into tree structures to represent part-whole hierarchies. The composition pattern allows clients to treat both individual objects and compositions of objects consistently.