CS 247

Software Engineering Principles

Chapter 15

Iterator and Composite Patterns

Victoria Sakhnini

Table of Contents

Well-Managed Collections

There are many ways to stuff objects into a collection. You can use an array; you can use a linked list; you can use a hash map; you can use a balanced tree; you can use a custom container someone wrote in 1987 for a piece of hardware that no longer exists. Each storage shape has its own access pattern: arrays are indexed by integers, linked lists are walked by following next pointers, hash maps are queried by key, and trees are traversed by recursion. The internal representation of a collection determines, in detail, how you ask it for one element after another.

This is fine when a single class owns a single collection and never lets anyone else see how it stores its data. The class can use whatever representation it likes, and outside code never has to know. But the moment a *client* of the class wants to iterate over the collection, print its elements, sum its values, filter it, or search for a particular item, the client has to know how the items are stored. The choice of internal representation has leaked out of the class and become part of its public contract.

The leak in practice

Concretely, suppose you have a `PancakeHouseMenu` that stores its items in an `std::vector<MenuItem>`, and a `DinerMenu` that stores its items in a fixed-size `MenuItem[]` array because someone in 1995 thought heap allocation was a sin. Now the chain hires a single `Waitress` class to print both menus. The Waitress wants to write something like "for each menu item in the menu, print its name and price." But she can't, the two menus use different storage, so the loop she'd write for one menu doesn't work for the other:

```
// Inside Waitress::printMenu(), without the Iterator pattern:

// Print the PancakeHouseMenu (a std::vector)
const auto& breakfastItems = pancakeHouseMenu.getMenuItems();
for (size_t i = 0; i < breakfastItems.size(); ++i) {
    const MenuItem& m = breakfastItems[i];
    std::cout << m.getName() << ", $" << m.getPrice() << '\n';
}

// Print the DinerMenu (a fixed-size C array)
const MenuItem* lunchItems = dinerMenu.getMenuItems();
for (size_t i = 0; i < dinerMenu.size(); ++i) {
    const MenuItem& m = lunchItems[i];
    std::cout << m.getName() << ", $" << m.getPrice() << '\n';
}
```

Two loops, structurally identical, both walking from index zero to size minus one, both calling the same accessor methods on the same kind of item. Yet they must be written separately because the underlying

storage type differs between menus. A third menu, say a `CafeMenu` that uses a `std::unordered_map<std::string, MenuItem>`, would need a third loop, completely different in shape from the first two, because hash maps cannot be walked by integer index at all.

The Waitress class is doing work she shouldn't have to do. She is being asked to know how each menu stores its items. Every time a new menu type is added, the Waitress must be updated. Every time a menu changes its internal storage, the Waitress breaks. The Waitress's code grows in proportion to the number of menus, when intuitively it should grow not at all; what changes is the data, not the act of printing.

The shape of the right answer

The right answer to this leak is captured in a single phrase: the menu should give the Waitress a way to walk through its items without telling her how they are stored. If the menu hands her a small, uniform object that knows how to produce the next item, whatever "next item" happens to mean for that particular storage, the Waitress can use the same loop regardless of which menu she is processing. The menu encapsulates its own traversal in this small helper object, and the Waitress is freed from caring about anything except the items themselves.

That small helper object is what the Gang of Four called an Iterator. The pattern's deceptively small name covers what is, in practice, one of the most useful and most copied design ideas in computing. Every modern language ships with iterators built into its standard library. Every for-each loop in every modern language compiles down to a call to a method called something like "begin" and another called something like "next." Once you have the abstraction, you cannot imagine writing collections without it.

> 💡 **Tip:**
>
> **Two patterns, one chapter.** This chapter covers two related Gang-of-Four patterns. The Iterator Pattern lets clients walk a collection without exposing its representation. The Composite Pattern lets clients treat individual objects and groups of objects uniformly. They go together because the same example, the Objectville Diner's menu system, motivates both, and because in real code they are almost always used in combination: composites are walked by iterators.

The Iterator Pattern

With the motivation in hand, here is the canonical definition, in the wording of the original Design Patterns text:

> *Iterator: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.*

That sentence contains three terms of art that are worth pulling apart before we go further: aggregate, sequential, and underlying representation. Each one carries a specific meaning that the pattern relies on, and each one is the source of a common misunderstanding.

"Aggregate object"

"Aggregate" is the Gang of Four's term for any object whose primary purpose is to hold a collection of other objects. An `std::vector`, a `std::list`, an `std::map`, and an `std::unordered_set` are all aggregates. So is a `PancakeHouseMenu` that holds menu items, a `Library` that holds books, a `Department` that holds employees, an XML document that holds nodes, and a directory in a filesystem that holds files. The aggregate may use any internal storage; it doesn't matter what, and the pattern is concerned with what the aggregate offers to outside code, not how it manages itself internally.

It is worth being explicit about what the pattern is *not* about. A class with one or two scalar fields is not an aggregate. A `Point` with `x` and `y` coordinates is not an aggregate even though it "contains" two numbers; the numbers are constituent parts of a single value, not elements of a collection. The Iterator Pattern only applies when there is a meaningful notion of "one item after another."

"Sequentially"

Iterators produce one element at a time, in some defined order, until they run out. They do not, at least not in the textbook pattern, produce all elements at once, or jump around, or skip backwards, or run two pointers in parallel. "Sequentially" means the client gets element zero, then element one, then element two, and so on, until the iterator signals that there are no more elements.

The order in which elements appear is a property of the iterator and the aggregate together. For a `std::vector`, the order is the order of insertion. For an `std::map`, it is the sorted order of keys. For an `std::unordered_map`, it is whatever the hash function happens to produce. For a tree iterator, it might be in-order, pre-order, or post-order. The Iterator Pattern itself does not specify any particular order; it only states that an order exists and that iteration follows it deterministically.

More advanced versions of the pattern relax the strict sequential constraint. Bidirectional iterators can step backwards as well as forwards. Random-access iterators can jump to arbitrary positions. Multiple iterators can coexist on the same collection. The C++ standard library is built on a richer iterator hierarchy (input, output, forward, bidirectional, random-access) that we will meet in Section 6. The plain

Iterator Pattern, as named in the Gang of Four book, is the simplest case: one direction, one element at a time.

"Without exposing the underlying representation"

This is the constraint that earns the pattern its keep. The client must not be able to tell, from the iterator's interface alone, what kind of storage backs the collection. The iterator's job is to be a uniform window onto the elements; the storage behind that window, array, list, tree, or hash table, is the aggregate's secret. The aggregate decides what to use, can change its mind later, and the client neither knows nor cares.

This is more than aesthetic decoupling. It is what makes generic algorithms possible. Once every collection exposes an iterator with the same interface, you can write a single `sum` function that adds the elements of *any* collection, vector, list, tree, or custom type without modification. Standard libraries from the STL onwards have made this their headline feature: algorithms that work over iterators are immediately applicable to every container that produces them. The Iterator Pattern is the small, local design decision that, applied consistently across a codebase, unlocks library-scale composability.

> **? Question:**
> Suppose a colleague proposes the following "iterator" for a class: a single method `getAllItems()` that returns `std::vector<MenuItem>` containing all the items. The client can then loop over the returned vector. Does this fit the Iterator Pattern? Why or why not?
>
> **Answer.** It does not fit the pattern for two reasons. First, returning a `std::vector` leaks information about the collection's representation, or at least, leaks the fact that the items can be stored in a vector all at once. A collection that holds ten million items, or that is infinite (a stream), cannot reasonably return them all in a single vector. Second, the pattern is about traversing the original collection, not about copying a snapshot. The point is that one element is produced on demand, so memory usage stays small, lazy generation is possible, and the client can stop early without forcing the collection to materialize everything. A `getAllItems` method is a perfectly fine *convenience method*, but it is not an iterator, and it does not give you the pattern's benefits.

Anatomy: Aggregate, Iterator, and the Two Interfaces

The Iterator Pattern has four participants. They are slightly more elaborate than the two-participant Template Method shape from the previous chapter, but the extra structure is small and pays for itself immediately:

- **Aggregate (interface).** Declares one method, typically called `createIterator()`, that returns an Iterator. The aggregate's job, beyond holding its items, is to know how to produce a fresh iterator over them.
- **ConcreteAggregate.** Implements the Aggregate interface for one specific storage shape. There may be many concrete aggregates: `PancakeHouseMenu` backed by a vector, `DinerMenu` backed by an array, and so on. Each one knows how to produce an iterator appropriate to its own storage.
- **Iterator (interface).** Declares the methods clients use to walk the collection: typically `hasNext()` ("is there another element?") and `next()` ("give it to me and advance"). Some versions of the pattern add `remove()` or `reset()`. The C++ STL uses a different but equivalent interface based on dereference and increment operators, which we will see in Section 6.
- **ConcreteIterator.** Implements the Iterator interface for one specific aggregate. It holds whatever state is necessary to know which element comes next, an integer index for an array, a node pointer for a linked list, or a stack of pending nodes for a tree. The state is *encapsulated*: clients never see it.

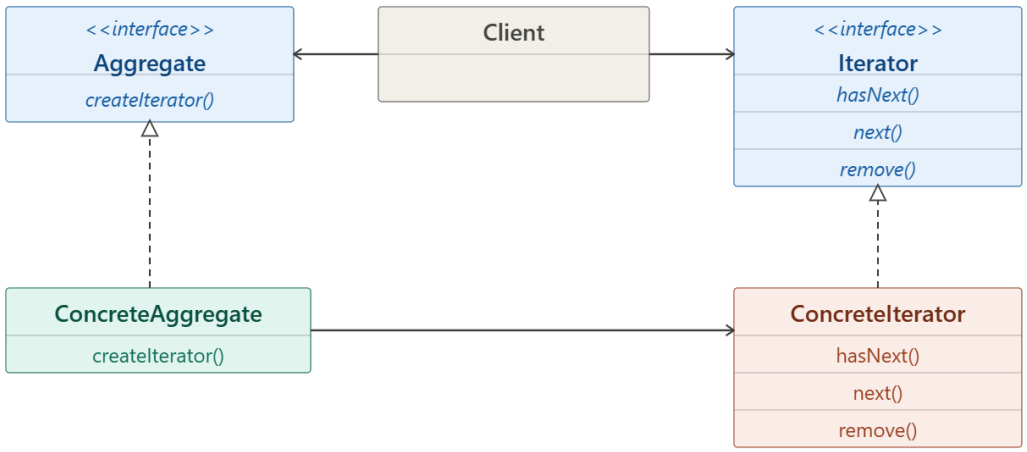


Figure 3.1: The Iterator Pattern.

Why two interfaces?

A natural question to ask is why the pattern needs two interfaces (Aggregate and Iterator) rather than one. Couldn't the Aggregate just have `hasNext()` and `next()` methods on it directly, and skip the indirection?

The two-interface design exists because traversal state differs from the collection itself. If `hasNext` and `next` lived on the collection, the collection itself would have to remember "the current position", which would mean exactly one traversal at a time could be in progress. Two clients walking the same collection in parallel (for instance, a `zip` operation that walks two sequences together) would corrupt each other's progress. A recursive walk that needed to remember several positions at once (the parent's iteration, the child's iteration) would be impossible.

Separating the iterator into its own object makes traversal state independent of the collection. Each call to `createIterator` produces a fresh iterator with its own state; many can coexist; each can be discarded when its work is done; and the collection itself is unchanged by any of this. This is the key reason the pattern is built the way it is.

Reading the iterator interface

The classic Iterator interface has two methods:

```
template<typename T>
class Iterator {
public:
    virtual ~Iterator() = default;
    virtual bool hasNext() const = 0;  // is there another element?
    virtual const T& next()       = 0;  // return it and advance
};
```

`hasNext` is the predicate: it tells the client whether the iteration is finished. It is `const` because asking the question should not consume the answer. `next` does two things in one call: returns the current element and advances the internal cursor, and that combination of read and side-effect is why `next` is *not* `const`. The pattern uses them together: a typical client loop looks like

```
while (it.hasNext()) {
    const MenuItem& item = it.next();
    std::cout << item.getName() << '\n';
}
```

and that loop is the same for every aggregate that produces this kind of iterator, regardless of storage. The client has zero awareness of arrays, vectors, lists, trees, or hash tables, only of the iterator's two methods.

A Worked Example: The Objectville Diner

The motivating story for this pattern, drawn from *Head First Design Patterns*, is the merger of two restaurants. The Objectville Diner has a Lunch menu of sandwiches and soups; the Objectville Pancake House has a Breakfast menu of pancakes and waffles. The two restaurants are joining forces, sharing a single Waitress to read both menus to customers. They are written by different programmers, however, and the two menus use different internal storage: a `std::vector` on one side, a fixed-size C-style array on the other. The Waitress, who knows nothing about C++ storage, would like to print both menus with the same code.

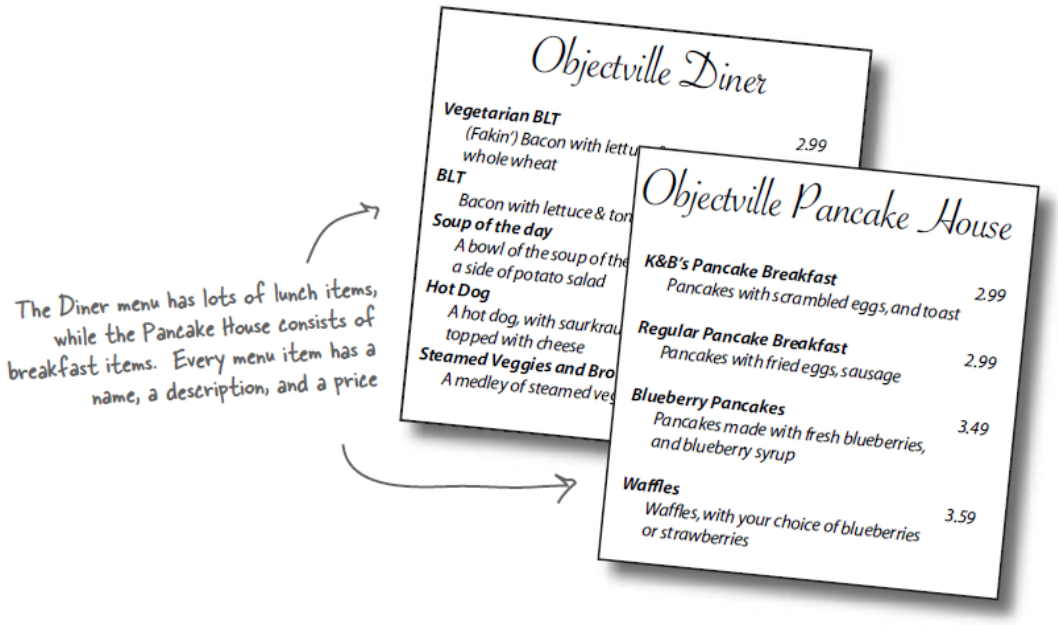


Figure 4.1: The Objectville Diner and Pancake House merger.

It helps to see the pattern with the relationships labelled in everyday language. The figure shows the same four boxes plus the small comments that explain what each piece is doing:

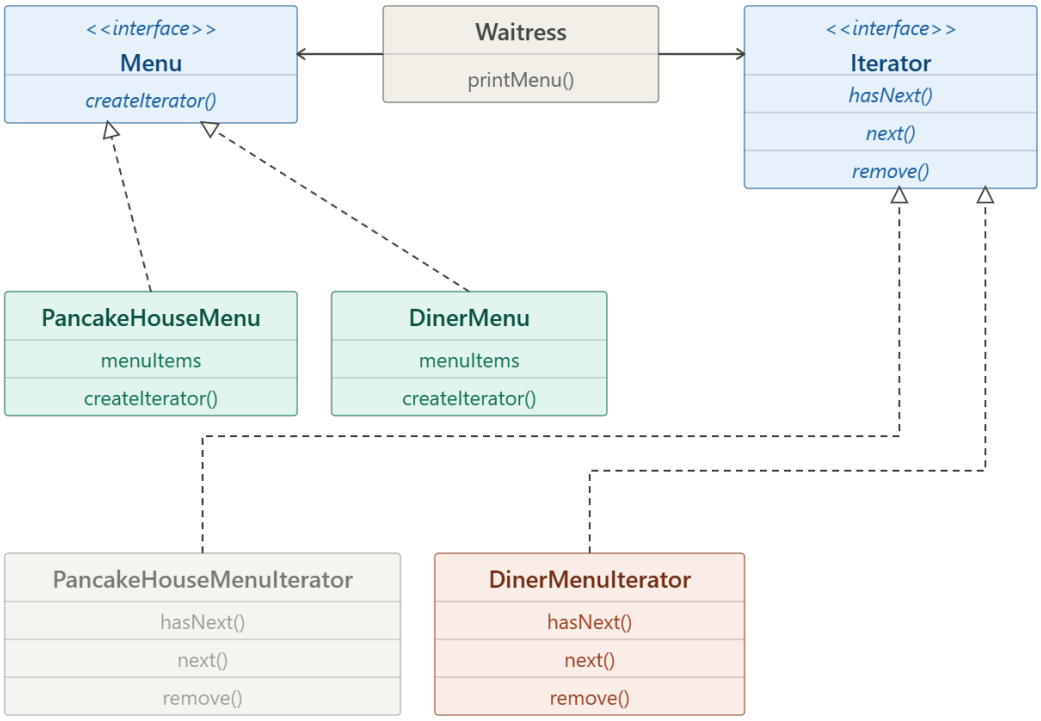


Figure 4.2:

Three things in Figure 3.2 are worth pinning down. First, the dashed arrows are UML's notation for "implements": `PancakeHouseMenu` implements the `Menu` interface, and `DinerMenuIterator` implements the `Iterator` interface. Second, the solid arrows from `Waitress` point to the abstract interfaces, not to the concrete classes; that is what the diagram is showing when it says the client "depends on" only the interfaces. Third, each concrete aggregate creates its own concrete iterator (the curly brace in the diagram represents this); the two concretes *know about each other*, because the menu has to know how to make the right kind of iterator for its storage, but neither one is exposed to the client.

The MenuItem class

First, the smallest piece. Every menu item, regardless of which menu it belongs to, has a name, a description, and a price. This is a simple value class with no behaviour beyond accessors:

```
// MenuItem class
```

```
class MenuItem {
public:
    MenuItem(std::string name, double price, std::string description)
        : name_(std::move(name)),
          price_(price),
          description_(std::move(description)) {}

    std::string_view getName() const noexcept { return name_; }
    double getPrice() const noexcept { return price_; }
    std::string_view getDescription() const noexcept {
           return description_;
    }

private:
    std::string name_;
    double      price_;
    std::string description_;
};
```

Two modern-C++ touches are worth pointing out. The constructor takes its string arguments *by value* and then `std::move`s them into the member fields. This is the modern idiomatic way to write a constructor that needs to store a copy: it works equally well whether the caller passes an lvalue (one move) or an rvalue (one move from the caller's argument). The accessors return `std::string_view`, a non-owning view into the stored string, which is cheaper than returning by value and safer than returning a `const std::string&` because the caller is signalled that they should not store the result long-term.

The Menu interface

Now the Iterator Pattern is applied. The `Menu` interface is what the Waitress sees. It declares only what every menu needs to offer: a way to start traversal and a way to end traversal, and nothing about how the menu stores its items:

```
// Menu interface, returns standard C++ iterators
class Menu {
public:
    using const_iterator = std::vector<MenuItem>::const_iterator;

    virtual ~Menu() = default;
    virtual const_iterator begin() const = 0;
    virtual const_iterator end()   const = 0;
protected:
    Menu() = default;
};
```

This is the textbook Iterator Pattern, adapted to modern C++ conventions. The pure Gang of Four version of the interface would have a single method, `createIterator()`, that returns a custom iterator object. The C++ version instead exposes two methods, `begin()` and `end()`, that return standard library iterators, because that is what every modern C++ container does, and it is what the range-based `for` loop expects to find. The `using const_iterator = ...` line introduces the iterator type as part of the Menu interface; any concrete menu must return that exact iterator type from `begin` and `end`.

There is a real C++ subtlety in this declaration. By tying the iterator type to `std::vector<MenuItem>::const_iterator`, we are committing every concrete Menu to ultimately storing items in a vector, or in something that returns the same iterator type. That commitment is what makes `Menu` usable as a polymorphic interface with virtual functions: the iterator type cannot itself be virtual, so it must be fixed at the interface level. In a strictly idealized version of the pattern, we would either use type erasure (a `std::function`-style iterator) or templates (giving up dynamic dispatch). The vector-iterator compromise is what production C++ code routinely picks; it is good enough in practice and far simpler than the alternatives.

The concrete menus

Now, the two concrete menus. Each one stores its items in a vector internally and forwards begin() and end() to the vector's begin and end. The Waitress has no way of knowing this; she only sees the Menu interface, but for the sake of this chapter, we can do the simple thing and pick the same internal storage for both:

```
// DinerMenu class
class DinerMenu final: public Menu {
public:
    void addItem(MenuItem item) {
        menuItems_.emplace_back(std::move(item));
    }
    const_iterator begin() const override { return menuItems_.begin(); }
    const_iterator end()   const override { return menuItems_.end();   }
private:
    std::vector<MenuItem> menuItems_;
};

// PancakeHouseMenu class
class PancakeHouseMenu final: public Menu {
public:
    void addItem(MenuItem item) {
        menuItems_.emplace_back(std::move(item));
    }
    const_iterator begin() const override { return menuItems_.begin(); }
    const_iterator end()   const override { return menuItems_.end();   }
```

```
private:
    std::vector<MenuItem> menuItems_;
};
```

Notice how little real work each concrete class does. The override bodies are each one line and exist only to forward to the internal vector. This is what "encapsulating traversal" looks like in practice: the concrete menu is responsible for *producing* a working iterator, and that responsibility consists almost entirely of saying "my underlying container's iterator is the right one." If a future `LegacyMenu` used a C array, the override would have to be more elaborate; it would need to produce iterators into the array, but the Menu interface would not change, and the Waitress would not notice.

The Waitress, freed from caring

Now the Waitress. She is handed both menus by interface, stores them by const reference, and prints them with the same loop:

```
// Waitress class, uses range-based for loops
class Waitress {
public:
    Waitress(const Menu& pancakeHouseMenu, const Menu& dinerMenu)
        : pancakeHouseMenu_(pancakeHouseMenu),
          dinerMenu_(dinerMenu) {}

    void printMenu() const {
        std::cout << "MENU\n----\nBREAKFAST\n";
        printMenuSection(pancakeHouseMenu_);
        std::cout << "\nLUNCH\n";
        printMenuSection(dinerMenu_);
    }

private:
    void printMenuSection(const Menu& menu) const {
        for (const auto& item : menu) {
            std::cout << item.getName() << ", $" << item.getPrice()
                      << " -- " << item.getDescription() << '\n';
        }
    }
    const Menu& pancakeHouseMenu_;
    const Menu& dinerMenu_;
};
```

Look at the loop in `printMenuSection`: `for (const auto& item : menu)`. A single loop, working over a `const Menu&`, with no mention of vectors, arrays, indices, or sizes. The range-based `for` loop is

C++'s syntactic sugar for the Iterator Pattern: the compiler rewrites it to a call to `menu.begin()`, then a loop that increments the iterator and compares it to `menu.end()` until they meet. That rewriting is hidden from you, but underneath it is exactly the iterator interface we just defined.

The Waitress class is now what it should be: a printer. It prints menus. It does not know about vectors. If you replaced one menu with a tree-backed implementation and the other with a HashMap-backed implementation, the Waitress would not change, would not need to be recompiled (in the interface sense; implementation files would need to be recompiled because their includes shifted), and would not break. The leak has been plugged.

The driver code

Finally, a main function that exercises everything:

```
int main() {
    // Create menus
    PancakeHouseMenu pancakeHouseMenu;
    pancakeHouseMenu.addItem({"K&B's Pancake Breakfast", 2.99,
        "Pancakes with scrambled eggs, and toast"});
    pancakeHouseMenu.addItem({"Regular Pancake Breakfast", 2.99,
        "Pancakes with fried eggs, sausage"});
    pancakeHouseMenu.addItem({"Blueberry Pancakes", 3.49,
        "Pancakes made with fresh blueberries"});
    pancakeHouseMenu.addItem({"Waffles", 3.59,
        "Waffles, with your choice of blueberries or strawberries"});

    DinerMenu dinerMenu;
    dinerMenu.addItem({"Vegetarian BLT", 2.99,
        "(Fakin') Bacon with lettuce & tomato on whole wheat"});
    dinerMenu.addItem({"BLT", 2.99,
        "Bacon with lettuce & tomato on whole wheat"});
    dinerMenu.addItem({"Soup of the day", 3.29,
        "Soup of the day, with a side of potato salad"});
    dinerMenu.addItem({"Hotdog", 3.05,
        "A hot dog, with sauerkraut, relish, onions, topped with
cheese"});
    dinerMenu.addItem({"Steamed Veggies and Brown Rice", 3.99,
        "Steamed vegetables over brown rice"});
    dinerMenu.addItem({"Pasta", 3.89,
        "Spaghetti with Marinara Sauce, and a slice of sourdough bread"});

    // Print menu
    Waitress waitress(pancakeHouseMenu, dinerMenu);
    waitress.printMenu();
}
```

Three things to notice in `main`. First, the `MenuItems` are constructed in-place using brace-initializer syntax, `{"K&B's Pancake Breakfast", 2.99, "..."}`, which is cleaner than spelling out `MenuItem{"K&B's...", 2.99, "..."}` every time. Second, the `addItem` call takes a `MenuItem` by value, and the brace-init constructs a temporary which is then moved into the vector, no copies, despite the value-semantics surface. Third, the Waitress is constructed with concrete menu objects but stores them by base-class const reference; the polymorphic dispatch for `begin`/`end` is what makes `printMenuSection` work uniformly across the two menus.

> **Tricky:**
> Why does `Waitress` store its menus as `const Menu&` rather than by value? Because a `Menu` is abstract, it has pure virtual functions, and you cannot create a *value* of an abstract type. The Waitress must hold its menus via pointers or references to use the polymorphic interface. References are cleaner than pointers when null is not a meaningful value ("a Waitress without a menu" doesn't make sense), and `const` is correct because the Waitress only reads from her menus.

The Single Responsibility Principle and Cohesion

Before we move on to the Composite Pattern, there is a design principle in the Iterator chapter of the Gang of Four discussion that deserves its own section, because it justifies why the pattern is built the way it is and because it has applications well beyond iterators.

A class should have only one reason to change.

This is the *Single Responsibility Principle*, often abbreviated SRP. The wording, "only one reason to change", is more precise than the colloquial "a class should do one thing," because "one thing" is impossible to pin down. Every class can be described as doing one big thing or many little things, depending on how you squint. "Reason to change," on the other hand, is concrete: it asks you to look at the forces that would cause this class's code to be edited, and to count them.

What does "reason to change" mean

A class with one reason to change is a class such that, when you sit down to edit its code, you are responding to exactly one kind of pressure from the outside world. The `PancakeHouseMenu` class, after the Iterator refactor, has one reason to change: the set of breakfast items on offer. If pancakes become trendier than waffles, or a new dish is added, that is when `PancakeHouseMenu` gets edited, and at no other time. The `Waitress` class also has one reason: the way menu items are displayed to customers. If

management decides to add a print-as-PDF option or change the formatting, that is when the Waitress is edited.

Before the refactor, back when the Waitress had to know about vectors on one side and arrays on the other, the Waitress had at least three reasons to change. She would be edited when display formatting changed, when the Pancake House's storage changed, *and* when the Diner's storage changed. The three pressures came from three different sources, lived in three different teams (so to speak), and yet they all routed through the same file. Every edit risked breaking the others.

Cohesion: the positive form

The positive form of the Single Responsibility Principle is the term *cohesion*. A class has **high cohesion** when its members, fields, methods, and behaviours are all about the same thing, all serve the same purpose, and all change together for the same reason. A class has **low cohesion** when its members span two or three unrelated responsibilities and have to be edited independently of one another.

The pre-Iterator `PancakeHouseMenu`, if it had had a `printItems` method that walked its own storage, would have had low cohesion: storage is one responsibility, formatting is another, and bundling them in one class means they share a file but have no reason to share a reason for change. Pulling the iteration out into a separate iterator object and printing it out into a separate Waitress class gives each remaining class high cohesion: the menu is just storage, the iterator is just traversal, and the waitress is just printing.

SRP and the Iterator Pattern

The Iterator Pattern is one of the cleanest examples of the Single Responsibility Principle applied at the design level. Before the pattern, the menu class had two responsibilities: holding items and being walked. After the pattern, the menu still holds items, but "being walked" is delegated to an iterator object that exists only for that purpose. Each class has one reason to change: the menu changes when its items change, the iterator changes when its storage's traversal logic changes (which is approximately never, because the standard library handles it).

This is why the pattern feels almost forced when you first see it. "Why do we need a whole separate class just to walk a list? Couldn't the list walk itself?" Yes, technically, and then the list would have two reasons to change, and when its storage evolved, you'd be editing the same file as when its semantics changed, and the changes would conflict in code review, and a year later, nobody would remember why *storage* code and *traversal* code lived together. The iterator object isn't there to make the design pretty; it's there to keep the two responsibilities separately editable.

C++ Iterators and the STL

The C++ Standard Template Library, usually abbreviated STL, is built around iterators. Almost everything in `<algorithm>` takes a pair of iterators rather than a container; almost everything in `<ranges>` composes operations over iterator sequences; and the range-based `for` loop is sugar over begin/end. The STL is, in effect, an industrial-strength application of the Iterator Pattern, refined and elaborated over thirty years of practice. To use modern C++ fluently, you need to know the basic shape of its iterator machinery.

The STL iterator interface

Where the textbook Iterator Pattern has methods named `hasNext` and `next`, STL iterators use the operator-overloading machinery that C++ inherited from pointers. The reason is partly historical (STL iterators were designed to generalize pointers into arrays) and partly practical (the resulting interface lets generic algorithms work on raw arrays and STL containers with the same code). The STL iterator's public interface is:

Operation	Meaning	Textbook equivalent
`*it`	Dereference: return the element currently pointed to.	Used inside `next()` to fetch the current element.
`++it`	Advance to the next element.	The advancing half of `next()`.
`it != end`	Check whether the iterator is at the end position.	`hasNext()`, but inverted in flavour.
`it == end`	Check whether two iterators refer to the same position.	No direct equivalent, STL idea.
`it = other`	Copy an iterator to a new variable; both can now walk independently.	No direct equivalent, pointer-like semantics.

Two things make the STL interface more powerful than the textbook one. First, the iterator has a *position* rather than a "have I run out?" flag. The position is compared against a second iterator, the "end" iterator, which represents "one past the last element." The loop ends when `it == end`, not when some `hasNext` returns false. Second, iterators are cheap to copy and independent of one another once copied. You can keep one iterator pointing to an interesting position while you walk another past it. The textbook pattern can't do this without explicit support.

The iterator category hierarchy

STL iterators come in five strengths, each more capable than the last, and STL algorithms advertise the minimum strength they need. Knowing the categories tells you which algorithms work with which containers:

- **Input iterator.** Can read elements once, in order. The classic example is reading from a stream: once you've read an element, you can't go back. Used in algorithms that pass over the data once.
- **Output iterator.** Can write elements once, in order. The classic example is writing to a stream or a back-inserter into a container.
- **Forward iterator.** Can read elements multiple times, in order. `std::forward_list` gives you these. Algorithms that need a second pass over the data require at least this level.
- **Bidirectional iterator.** Can step both forwards and backwards. `std::list`, `std::map`, and `std::set` all give you these. Algorithms like `std::reverse` need this category.
- **Random-access iterator.** Can jump to any position in constant time. `std::vector`, `std::array`, and `std::deque` provide these, as do raw pointers to arrays. Algorithms like `std::sort` need this level; they cannot work on a `std::list`, and the standard library has a separate `std::list::sort` member function to fill that gap.

Why the STL design is worth imitating

When you design your own collection class, the conventional advice is to make it look like an STL container: provide `begin` and `end`, expose nested `iterator` and `const_iterator` types, and give the iterators the standard operators (`*`, `++`, `==`, `!=`, and so on). This convention buys you, for free:

- **Range-based for loops.** Your class can be used in `for (const auto& x : my_collection)`.
- **Compatibility with <algorithm>.** Functions like `std::find`, `std::count_if`, `std::transform`, and `std::accumulate` work on your class without any extra code.
- **Compatibility with <ranges> and views.** The pipe-style `my_collection | std::views::filter(...) | std::views::transform(...)` syntax works.
- **Familiarity for callers.** Any C++ programmer can use your class without learning a new API; they already know how vectors work.

This is the Iterator Pattern paying compound interest. The pattern was originally about decoupling clients from a single collection's storage; in the STL, it has become the basis for an entire interoperable ecosystem of containers, algorithms, and views. Whenever you reach for `std::vector`, you are using the pattern. Whenever you write a `for` loop over a container, you are using the pattern. It is so deeply ingrained that it has become invisible, but the underlying design idea is exactly the one the Gang of Four wrote down in 1994.

Motivation: From Menus to Menus-Within-Menus

So far, the Iterator Pattern has solved a clean, flat problem: two menus, each a list of items, walked uniformly through an iterator. That works as long as a menu really is just a list of items.

Now the requirement changes. The merged restaurants want a dessert submenu attached to the Diner menu, so dessert items appear under the lunch items in a tree-shaped hierarchy. A few months later, they added a separate Cafe menu for dinner. A few months after that, they want a separate kids' sub-menu under the Pancake House menu. The pattern starts to creak.

Why the Iterator Pattern alone is not enough

Iterators walk *flat* sequences. They produce one element at a time, in a defined order, and then stop. What does the iterator do when an element is itself a menu containing more elements? The textbook iterator doesn't have a concept of "recurse into this element"; it just hands the element to the client. The client would then have to test whether the element is a menu item or a sub-menu, and, if it is a sub-menu, get a new iterator for it and walk that. The Waitress's printing code would suddenly need to know about the type structure, which menu items can have children, which cannot, and the leak in the Iterator Pattern carefully plugged would open back up.

There is an obvious half-solution: make every iterator a tree-walking iterator that automatically descends into sub-menus. The Waitress sees only leaves, never composites; sub-menus are invisible to her. But she now has lost the ability to print the structural information, "BREAKFAST," "LUNCH," "DESSERT", because the iterator collapses the hierarchy. To print the headers, she needs to see the composite nodes. And the moment she needs to see composites, she needs a way to distinguish them from leaves, and we are back to type-checking.

The shape of the right answer

The right answer is to stop treating composites and leaves as different kinds of things in the client's eyes. From the Waitress's point of view, a menu and a menu item should both be "something I can ask to print itself." If it is a menu item, it prints its own line. If it is a menu, it prints its header and then asks each of its children to print themselves. The Waitress doesn't need to know which case she has; she just calls print on whatever she is holding.

To make this work, menus and menu items must share a common interface; they must be the same type from the Waitress's point of view. They differ internally: a menu has children, and a menu item does not. But that internal difference is hidden behind a uniform external interface. Both types support the operations the client cares about; the client does not test the type.

This is the Composite Pattern. It is what lets you take a tree structure of mixed objects and walk it as if it were a flat collection. Together with the Iterator Pattern, it is the foundation of every file-system

browser, every XML or JSON parser, every GUI framework's widget hierarchy, and every game's scene graph.

The Composite Pattern

The definition, in the original Gang of Four phrasing:

> *Composite: Composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.*

Two ideas in that definition, both worth pulling apart.

"Part-whole hierarchies"

A *part-whole hierarchy* is any structure where a thing can be either a single object or a collection of things of the same kind. A file is a single object; a directory is a collection of files *and directories*. A shape is a single drawing element; a group is a collection of shapes *and groups*. An expression is a literal value; a sub-expression is a collection of operators *and sub-expressions*. The pattern is named for this recursive nesting: the part and the whole are the same kind of thing, and the whole is made up of parts that are themselves potentially wholes.

The recursive shape is what distinguishes a Composite from a flat aggregate. A `std::vector<MenuItem>` is a flat aggregate: it holds menu items, period. A Composite-style menu holds *menu components*, where a menu component is either a menu item (a leaf) or another menu (which itself holds more menu components). The vector can be walked by a single iterator over its items. The Composite can only be walked by recursion: at every node, you find out whether it is a leaf or a composite, and if it is a composite, you walk into its children.

"Treat individual objects and compositions uniformly"

This is the constraint that earns the pattern its name. The client must be able to write code that calls operations on a node without knowing or caring whether the node is a leaf or a composite. `node.print()` should work; whether it prints one line (leaf) or many (composite) is the node's business, not the client's. `node.getName()` should work; whether it returns the item's name or the menu's name is internal. The client treats all nodes as instances of a common base type, and the type system never forces them to be downcast.

This is what makes the pattern more than just "a tree." Building a tree of nodes where leaves and internal nodes have different types is easy and obvious; you just write two classes and join them with pointers. The Composite Pattern is the discipline of giving the two classes the same interface so that

client code is type-blind. The recursion is in the data structure; the polymorphism is in the interface; together, they let the client write loops as if every node were the same.

There is a tension here that we will revisit in Section 11. Some operations only make sense on leaves ("get my price"); others only make sense on composites ("add a child"); and the uniform interface forces both kinds of operation onto both kinds of node. The pattern's transparency comes at the cost of giving leaves methods they cannot honour. How you handle that tension, exceptions, no-ops, and refusing to compile is the chief design choice in any Composite implementation.

> **? Question:**
> Is a `std::vector<std::vector<int>>`, a vector of vectors of ints, a Composite? Why or why not?
>
> **Answer.** Not quite. It is a *tree of depth two*, but it does not satisfy the "treat uniformly" half of the pattern: the inner type (`std::vector<int>`) is a different type from the outer (`std::vector<std::vector<int>>`), and the leaves (`int`) are a third type. Client code that walks the structure has to know at each level whether it is looking at an outer vector, an inner vector, or an int, and to test that knowledge with type information. The Composite Pattern would require a single `Component` base class that both the inner and outer collections inherit from, and the ints would be wrapped in leaf objects that also inherit from `Component`. At that point, the client could walk arbitrarily deep nesting without any type checks, which is the property the pattern gives you. A vector of vectors does not give you that property; it just happens to be tree-shaped.

Anatomy: Component, Leaf, and Composite

The Composite Pattern has three participants. The structural shape is small, three boxes in a diamond, but each piece has a specific role to play:

- **Component (abstract base class).** Declares the common interface that all objects in the composition must support. This includes both the leaf-side operations (the actual content operations: print itself, return its name, return its price) and the composite-side operations (the structural operations: add a child, remove a child, get the i-th child). All clients see Components; nothing else.
- **Leaf.** Represents a node that has no children. Implements the Component interface for its content operations; for the composite-side operations, it either throws an exception, returns nothing, or simply does not implement them, depending on which variant of the pattern you choose.

- **Composite.** Represents a node that has children. Stores a collection of Components, possibly leaves, possibly other composites; the type system can't tell. Implements the Component interface by, in most cases, forwarding to its children: "print yourself" becomes "print my header, then ask each child to print itself."

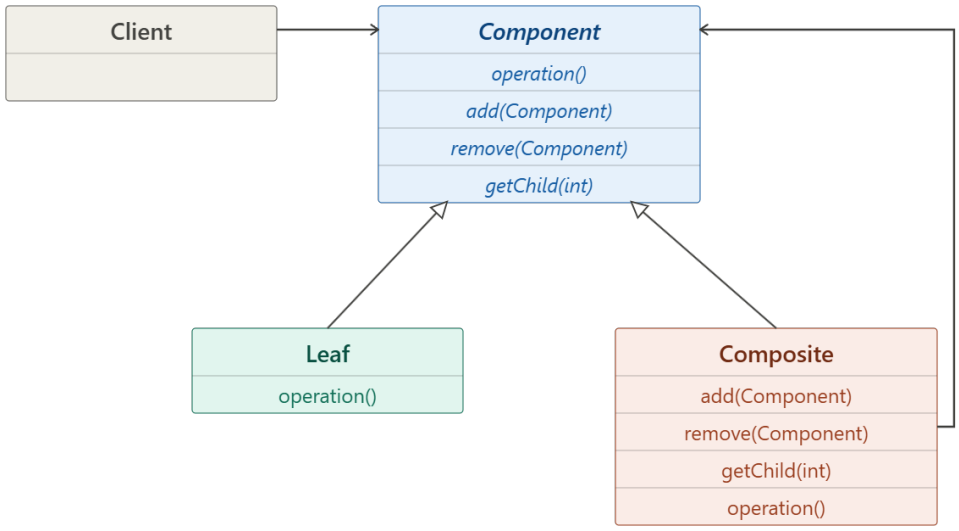


Figure 9.1: The Composite Pattern.

Three details in Figure 9.1 are worth dwelling on. First, the `Composite` has a reflexive arrow back to `Component`, that is, the diagram's way of saying "a Composite holds a collection of Components." The recursion is right there in the arrow: because `Composite` *is-a* `Component`, a Composite can hold Composites, and so on to arbitrary depth.

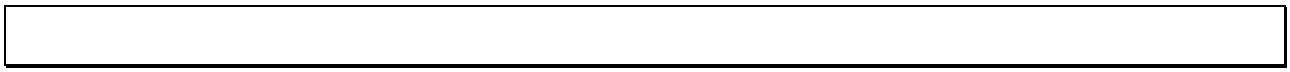
Second, both `Leaf` and `Composite` inherit from `Component`, and *both* inherit `add`, `remove`, and `getChild`. These operations only really make sense on composites, but the textbook pattern declares them at the Component level, so that leaves and composites have a single, uniform type. A leaf that inherits `add` but cannot meaningfully implement it has three escape options: throw an exception, do nothing, or hide the method, which we will compare in Section 11.

Third, the `Client` depends only on `Component`, not on `Leaf` or `Composite`. The client never says, "This is a leaf, so I'll do X," or "This is a composite, so I'll do Y." That is what "treat uniformly" means in practice: every client-side operation is `component.something()`, and the polymorphism inside the type system picks the right behaviour. If the client ever writes `dynamic_cast<Composite*>(...)`, that is a sign that the pattern has been compromised.

The bidirectional commitment

The trade-off encoded in this structure is worth naming explicitly. Putting child-management methods on the base Component interface is a deliberate design choice. It buys uniformity; the client can walk the tree without type-tests. It costs safety; the type system no longer prevents the client from calling "add a child" on a leaf. Asking a leaf to add a child is a runtime error rather than a compile-time error.

This is the central design tension in any Composite implementation. The textbook pattern prioritizes transparency over safety. Some derivative variants of the pattern flip the choice, keeping the structural operations off the base class and only on the composites, at the cost of forcing the client to test types when traversing. Real systems pick one side, document the choice, and live with the consequences. We will spend Section 11 on this.



A Worked Example: The Menu Composite

Now we apply the pattern to the menu problem. We need a tree structure where:

- Individual menu items (Vegetarian BLT, Apple Pie, Pancakes) are leaves.
- Menus (Pancake House Menu, Diner Menu, Dessert Menu, Cafe Menu, top-level All-Menus) are composites holding mixtures of items and sub-menus.
- The Waitress walks the whole structure with one call: `allMenus.print()`.

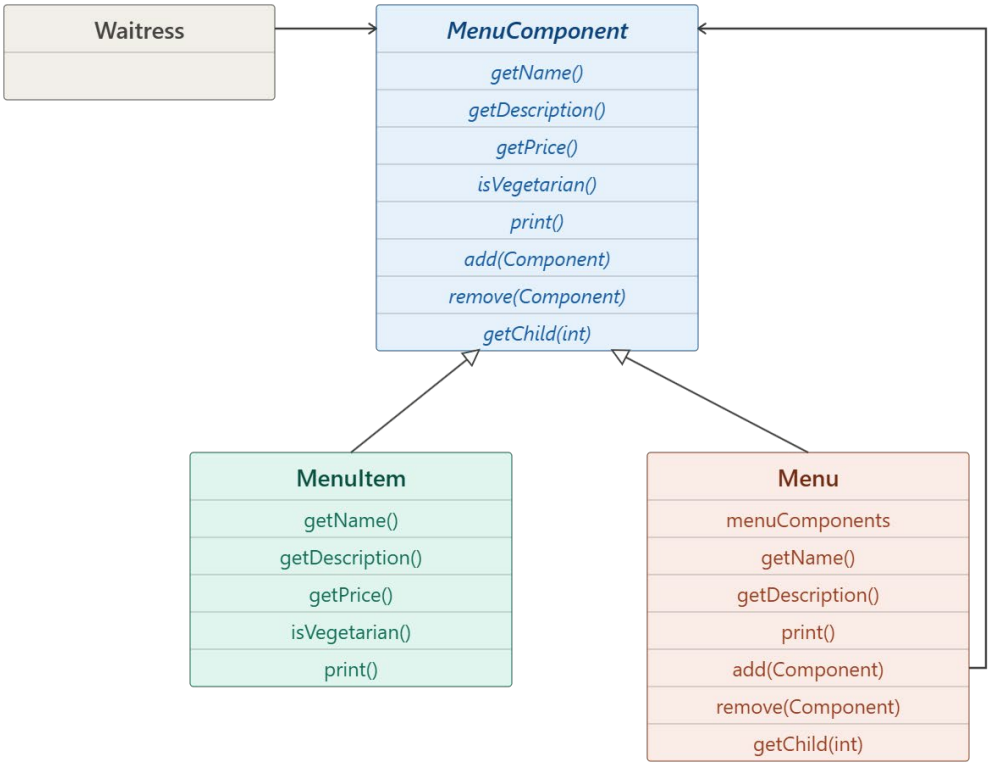


Figure 10.1: The menu hierarchy as a Composite.

The Component: MenuComponent

First, the base class. `MenuComponent` declares every operation any concrete component might need: leaf operations (`getName`, `getDescription`, `getPrice`, `isVegetarian`), structural operations (`add`, `remove`, `getChild`), and the universal operation (`print`). Operations that only make sense on one of the two sides throw `std::runtime_error` by default, the textbook "transparency" choice:

```
// MenuComponent base class (Composite pattern)
class MenuComponent {
public:
    virtual ~MenuComponent() = default;

    // Operations that may not be supported by all components
    virtual void add(std::unique_ptr<MenuComponent> menuComponent) {
        throw std::runtime_error("Operation not supported");
    }
    virtual void remove(const MenuComponent* menuComponent) {
        throw std::runtime_error("Operation not supported");
    }
    virtual MenuComponent* getChild(size_t index) {
        throw std::runtime_error("Operation not supported");
    }

    virtual std::string_view getName() const {
        throw std::runtime_error("Operation not supported");
    }
    virtual std::string_view getDescription() const {
        throw std::runtime_error("Operation not supported");
    }
    virtual double getPrice() const {
        throw std::runtime_error("Operation not supported");
    }
    virtual bool isVegetarian() const {
        throw std::runtime_error("Operation not supported");
    }

    // The one method every component must implement
    virtual void print() const = 0;
protected:
    MenuComponent() = default;
};
```

The choice to default every method to "throw" is deliberate and important. A leaf class will inherit add, `remove`, and `getChild` without override, and *if a client ever tries to call those on a leaf*, they will get an exception at runtime telling them they cannot. The leaf does not have to write out empty no-op overrides, and the exception message is clear. The same applies in the other direction: a composite that doesn't have a price inherits `getPrice` and throws an exception when asked.

The single pure virtual method is `print`. Every component, leaf or composite, must implement printing, that is, the operation the Waitress universally needs, and the type system enforces it. Anything else is optional from the base class's perspective; concrete classes override what they care about and inherit the throw-on-call default for the rest.

The leaf: MenuItem

Now the leaf. A MenuItem has a name, a description, a vegetarian flag, and a price. It implements the leaf-side operations and overrides print() to print its own one-line entry. It does not implement add, remove, or getChild, calling those on a MenuItem will trip the inherited throws:

```
// MenuItem class (Leaf node)

class MenuItem final : public MenuComponent {

public:
    MenuItem(std::string name, std::string description,
             bool vegetarian, double price)
        : name_(std::move(name)),
          description_(std::move(description)),
          vegetarian_(vegetarian),
          price_(price) {}

    std::string_view getName()        const override { return name_; }
    std::string_view getDescription() const override { return
description_; }
    double           getPrice()       const override { return price_; }
    bool             isVegetarian()   const override { return vegetarian_;
}

    void print() const override {
        std::cout << "  " << getName();
        if (isVegetarian()) {
            std::cout << " (v)";
        }
        std::cout << ", $" << getPrice() << '\n';
        std::cout << "     -- " << getDescription() << '\n';
    }

private:
    std::string name_;
    std::string description_;
    bool        vegetarian_;
    double      price_;
};
```

Two small modern-C++ patterns are visible here. The class is marked `final`; nothing inherits from it, which lets the compiler de-virtualize some calls and signals to readers that the leaf hierarchy is closed. The constructor takes its strings by value and moves them in, using the same idiom we saw with `MenuItem` in the Iterator example. The `print` implementation embeds the formatting of a single menu

item; it knows nothing about other items, its parent menu, or indentation across the tree. That tight focus is exactly what the pattern is supposed to enable.

The composite: Menu

Now the composite, which is where most of the work lives. A Menu has a name and description (like a leaf), plus a collection of child components. It implements add, remove, getChild for child management, and overrides print to print its header and then recursively ask each child to print itself:

```
// Menu class (Composite node)
class Menu: public MenuComponent {
public:
    Menu(std::string name, std::string description)
        : name_(std::move(name)),
          description_(std::move(description)) {}

    void add(std::unique_ptr<MenuComponent> menuComponent) override {
        menuComponents_.push_back(std::move(menuComponent));
    }
    void remove(const MenuComponent* menuComponent) override {
        auto it = std::find_if(menuComponents_.begin(),
                               menuComponents_.end(),
                              [menuComponent](const auto& component) {
                               return component.get() == menuComponent;
                               });
        if (it != menuComponents_.end()) {
            menuComponents_.erase(it);
        }
    }
    MenuComponent* getChild(size_t index) override {
        if (index >= menuComponents_.size()) {
            throw std::out_of_range("Index out of range");
        }
        return menuComponents_[index].get();
    }
    std::string_view getName() const override { return name_; }
    std::string_view getDescription() const override {
                 return description_;
    }
    void print() const override {
        std::cout << '\n' << getName() << ", " << getDescription() <<
        '\n';
        std::cout << "---------------------\n";
        for (const auto& component : menuComponents_) {
            component->print();   // recursive call
        }
    }
```

```
private:
    std::vector<std::unique_ptr<MenuComponent>> menuComponents_;
    std::string name_;
    std::string description_;
};
```

Three things to look at here, because they are the heart of the pattern. First, the child storage is `std::vector<std::unique_ptr<MenuComponent>>`. The `unique_ptr` is what gives the Menu ownership of its children: when the Menu is destroyed, its vector is destroyed, which destroys each `unique_ptr`, which destroys each child. A whole tree is freed by destroying the root. No manual `deletes, no leaks, no double frees`. This is the modern C++ way to write tree structures.

Second, add takes its argument as `std::unique_ptr<MenuComponent>` *by value*, forcing the caller to `std::move` the pointer in. This is how the function signature documents transfer of ownership: "You used to own this child; now I do." Trying to add a `unique_ptr` you still wanted to keep would not compile, which catches a class of bugs at build time.

Third, and most importantly, look at `print`. It does two things: it prints its own header line, and then it loops over its children and calls `component->print()` on each. That call is the recursive step of the pattern. The compiler doesn't know whether each child is a leaf or another composite; the virtual dispatch picks the right `print` at runtime. If the child is a `MenuItem`, it prints one line. If the child is a `Menu`, it prints a header and recurses into *its* children. A single loop in a single class walks an arbitrarily deep, arbitrarily wide tree.

Specialized menu classes and the Waitress

For convenience and to match the example's naming, we can derive thin subclasses for the specific menus, each just calling the Menu constructor with appropriate defaults:

```
// Specialized menu classes
class DinerMenu final: public Menu {
public:
    DinerMenu() : Menu("DINER MENU", "Lunch") {}
};

class PancakeHouseMenu final: public Menu {
public:
    PancakeHouseMenu() : Menu("PANCAKE HOUSE MENU", "Breakfast") {}
};

// Waitress class (Client)
```

```
class Waitress {
public:
    explicit Waitress(const MenuComponent& allMenus)
        : allMenus_(allMenus) {}

    void printMenu() const {
        allMenus_.print();
    }
private:
    const MenuComponent& allMenus_;
};
```

Look at the size of the new Waitress class. Three lines of meaningful code: store a reference, call print on it. That is *all* she does. The Waitress doesn't know about menus or items. She doesn't know about trees. She doesn't know about depth or ordering or formatting. She holds one `MenuComponent`, the top-level all-menus composite, and asks it to print itself. The entire tree of menus, sub-menus, and menu items prints itself in the correct order, with the correct headers, with the correct indentation, because every node knows how to print itself and how to ask its children to do the same.

The runtime tree

Visualizing the data structure at runtime helps fix the picture. The top-level All-Menus composite holds three child menus (Pancake House, Diner, Cafe). The Diner menu holds menu items plus a Dessert sub-menu. The Dessert sub-menu holds dessert items. Every node is a MenuComponent; leaf nodes (small circles) are menu items; non-leaf nodes are menus:

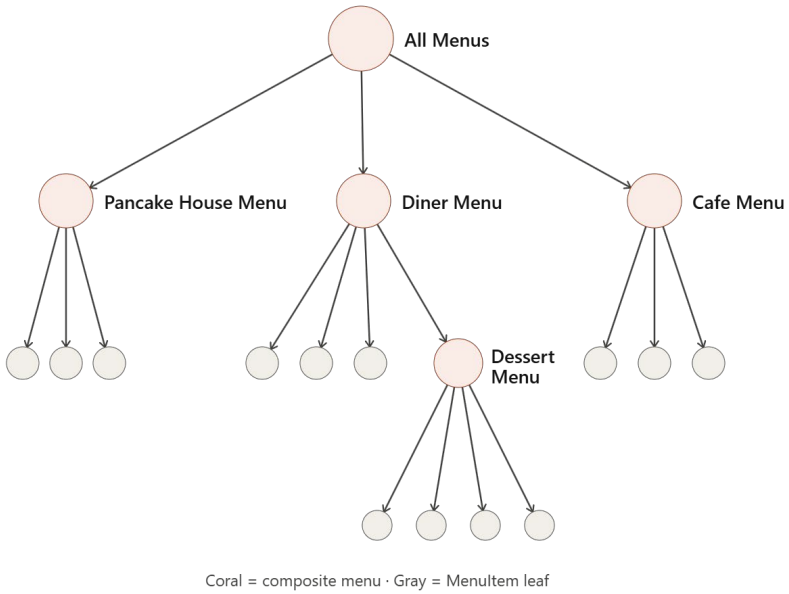


Figure 10.2: The menu composite at runtime.

The driver code

Finally, the main function. It constructs the tree bottom-up, transferring ownership of each child to its parent via std::move, and ends with a single call to printMenu:

```
// Helper function to create menu items
auto createMenuItem(std::string name, std::string description,
                    bool vegetarian, double price) {
    return std::make_unique<MenuItem>(
        std::move(name), std::move(description), vegetarian, price);
}

int main() {
    // Create menu hierarchy using smart pointers
    auto pancakeHouseMenu = std::make_unique<PancakeHouseMenu>();
    auto dinerMenu        = std::make_unique<DinerMenu>();
    auto cafeMenu         = std::make_unique<Menu>("CAFE MENU", "Dinner");
    auto dessertMenu      = std::make_unique<Menu>("DESSERT MENU",
"Dessert of course!");
    auto allMenus         = std::make_unique<Menu>("ALL MENUS", "All menus
combined");

 // Keep raw pointers for adding items (ownership transferred to allMenus)
    auto* pancakePtr = pancakeHouseMenu.get();
    auto* dinerPtr   = dinerMenu.get();
    auto* cafePtr    = cafeMenu.get();
    auto* dessertPtr = dessertMenu.get();

 // Build menu hierarchy
    allMenus->add(std::move(pancakeHouseMenu));
    allMenus->add(std::move(dinerMenu));
    allMenus->add(std::move(cafeMenu));

 // Populate pancake menu
    pancakePtr->add(createMenuItem("K&B's Pancake Breakfast",
        "Pancakes with scrambled eggs, and toast", true, 2.99));
 // ... more items ...

 // Attach dessert sub-menu to diner menu
    dinerPtr->add(std::move(dessertMenu));
    dessertPtr->add(createMenuItem("Apple Pie",
        "Apple pie with a flaky crust, topped with vanilla icecream",
        true, 1.59));
 // ... more dessert items ...

 // Print menu using Waitress
    Waitress waitress(*allMenus);
    waitress.printMenu();
     return 0;
}
```

There is one C++-specific subtlety worth understanding in this main. After we `std::move` `pancakeHouseMenu` into `allMenus->add(...)`, the original `pancakeHouseMenu` unique-pointer is empty. But we still want to add menu items to that pancake menu! The solution is to grab a raw pointer *before* the move (`auto* pancakePtr = pancakeHouseMenu.get()`), and use it afterwards. The raw pointer is not an owning pointer, so it doesn't conflict with the unique pointer's ownership, but it does let us access the object that `allMenus` now owns. This is the standard idiom for "transfer ownership but keep a working handle."

And look at what we get for our trouble. The complete menu, four menus, one sub-menu nested inside another, fifteen-ish individual menu items, headers, indentation, vegetarian markers, prints in correct order from a single call. The Waitress's `printMenu` function is two lines, including the brace. The tree's structure lives in the data, not the code; the code is uniform recursion. This is what the Composite Pattern feels like when it is working.

The Transparency-Safety Trade-Off

The textbook Composite Pattern violates the Single Responsibility Principle of Section 5. The `Menu` class manages a hierarchy *and* performs leaf operations like `getName` and `print`. That is two responsibilities, two reasons to change, and yet the pattern bakes them into one class. This is not an accident, it is a deliberate trade, and understanding the trade is essential to applying the pattern well.

Why the pattern violates SRP

Look at the `MenuComponent` interface from Section 10. It has two distinct groups of methods:

- **Child-management methods.** `add`, `remove`, `getChild`. These are about structural responsibility and keeping track of a tree of children.
- **Content methods.** `getName`, `getDescription`, `getPrice`, `isVegetarian`, `print`. These are about the leaf responsibility and are menu items with menu-item properties.

A clean SRP-respecting design would split these into two interfaces. A `Container` interface would have only the child-management methods. An `Item` interface would have only the content methods. Leaves would implement `Item`; composites would implement both Item and Composite. The client code would have to know at each step whether it was looking at a Container or an Item and would have to test the types.

The Composite Pattern, as the Gang of Four described it, deliberately does not do this. Instead, it puts *both* groups of methods on a single Component interface and accepts that some methods will be no-ops or errors on some implementations. The client benefits enormously; it can write code that doesn't test types, but the design pays a price in clarity. This trade is called *transparency* vs *safety*, and there are two ways to make it.

The transparent variant

Transparency means the client cannot tell from the type whether a component is a leaf or a composite. Both expose the same interface. The benefit is uniformity: the client treats all components identically, recurses without case analysis, and never downcasts.

This is the variant we built in Section 10. `MenuComponent` declares `add`, `remove`, and `getChild`; both `MenuItem` and `Menu` inherit them. The leaf's inherited `add` throws `std::runtime_error("Operation not supported")` because adding a child to a menu item doesn't make sense. If a client calls `item.add(...)` by accident, the program throws an error at runtime, and the bug is discovered.

The cost is real. The interface advertises operations that don't work on all implementations. A junior developer reading `MenuComponent`'s declaration sees nine methods and is given no signal about which ones are safe to call on which instances. The mistake "call `getChild` on a menu item" is something the type system would catch in a saner design; here, it is left to runtime exceptions.

The safe variant

Safety means the type system prevents the wrong method from being called on the wrong kind of node. The structural methods live only on the composite class; the leaf class does not have them; client code that wants to call add must hold a `Composite*`, not a `Component*`.

In code, this looks like:

```
// Safe variant, structural methods only on Composite
class MenuComponent {
public:
    virtual ~MenuComponent() = default;
    virtual std::string_view getName()        const = 0;
    virtual std::string_view getDescription() const = 0;
    virtual void             print()          const = 0;
};

class MenuItem final: public MenuComponent {
    // ... leaf-only data and methods, no add/remove/getChild ...
};

class Menu: public MenuComponent {
public:
    void add(std::unique_ptr<MenuComponent> child);
    void remove(const MenuComponent* child);
    MenuComponent* getChild(size_t index);
    // ... composite-specific data ...
};
```

The benefit is clarity. The interface tells you the truth: `MenuComponent` supports just the three operations every node has; `Menu` extends it with the structural operations. The type system catches the "add a child to a menu item" mistake at compile time. There is no runtime exception path to test.

The cost is also real, in the other direction. The client now has to know, when it has a `MenuComponent*`, that the structural operations are not available. If the client wants to walk the tree and add a child to every Menu node along the way, it must distinguish menus from items, typically with `dynamic_cast<Menu*>(component)`, which is exactly the type-testing the transparent variant was designed to avoid. The pattern's uniformity is gone; the client recovers it only by reintroducing the test that the transparent variant got rid of.

Which to pick?

Aspect	Transparent (textbook)	Safe (refined)
Interface size	Large: one interface holds everything.	Smaller: two interfaces split responsibilities.
Type safety	Runtime exceptions on misuse.	Compile-time errors on misuse.
Client code	Uniform, no type tests.	Tests required to use structural ops.
SRP	Violated two responsibilities per class.	Respected, one per class.
Best for	Clients that primarily traverse and read, mostly leaf operations.	Clients that frequently mutate the structure.

The right answer depends on what the Composite's clients do most often. If the dominant usage is *traversal and querying*, printing menus, computing total price, finding all vegetarian items, walking the tree to apply some operation, and picking transparency. The structural operations are called rarely (at construction time), and the cost of an occasional runtime error is small. If the dominant usage is *mutation*, adding and removing children, reorganizing the tree, and picking safety. The structural operations are central to the workload, and you want the type system to keep them honest.

The *Head First* version of the pattern, the one in our Section 10, picks transparency, on the implicit grounds that menu hierarchies are constructed once and read many times. Production C++ code in libraries like Qt and various tree-based data structures tends to pick the safe variant, because their clients more often add and remove than walk.

> ⚠ **Warning:**
> **Don't try to have both.** Some implementations try to hide the trade-off by exposing the transparent interface *and* providing a separate "safe" sub-interface. The result is usually worse than either pure choice: the surface area doubles, clients still have to think about which methods are safe, and the documentation becomes a maze. Pick one variant for the codebase and stick with it.

Iterator and Composite Together

The two patterns of this chapter compose. In production code, they appear together more often than not, because the structures that motivate Composite, trees, hierarchies, and nested groupings are exactly the structures that benefit from iteration. The two patterns line up cleanly: Composite gives you the data structure, and Iterator gives you a way to walk it without exposing the structure.

Tree iteration as a single pattern

Suppose we want to do something other than print our menu hierarchy, say, iterate over *only the menu items*, computing the total price of every vegetarian item across the whole tree. The recursive printing built into `Menu::print` is wedded to one operation (printing). To do a different operation, we'd have to write a different recursive method on every Menu and MenuItem in the hierarchy.

The fix is to give `MenuComponent` a `createIterator()` method, the Iterator Pattern from earlier, that returns an iterator that walks the entire subtree rooted at this component. For a `MenuItem` leaf, the iterator yields the single item and stops. For a `Menu` composite, the iterator yields the menu itself (if appropriate), then iterates over each child's iterator, and then stops. The recursion lives inside the iterator's `next` method; clients see only a flat sequence.

```
// A tree-walking iterator that flattens a Composite

class CompositeIterator {
public:
    explicit CompositeIterator(MenuComponent* root) {
        if (root) stack_.push(root);
    }
    bool hasNext() const { return !stack_.empty(); }
    MenuComponent* next() {
        if (stack_.empty()) return nullptr;
        MenuComponent* current = stack_.top(); stack_.pop();

   // If composite, push children in reverse so leftmost comes off first.
```

```
        if (auto* menu = dynamic_cast<Menu*>(current)) {
           for (int i = static_cast<int>(menu->size()) - 1; i >= 0; --i) {
                stack_.push(menu->getChild(i));
            }
        }
        return current;
    }
private:
    std::stack<MenuComponent*> stack_;
};
```

This is a single iterator that performs a depth-first walk of the entire tree, regardless of how deeply nested it is or how the children are arranged at any node. The client can now write:

```
double totalVegPrice(MenuComponent& root) {
    double total = 0.0;
    CompositeIterator it(&root);
    while (it.hasNext()) {
        MenuComponent* c = it.next();
        try {
            if (c->isVegetarian()) total += c->getPrice();
        } catch (const std::runtime_error&) {
            // Skip composites that don't have isVegetarian/getPrice
        }
    }
    return total;
}
```

The `try/catch` handling around the leaf-only operations is a direct consequence of the transparent variant of Composite from Section 11; the iterator visits both menus and items, but only items support `isVegetarian` and `getPrice`, so calls on menus throw. In a safe-variant codebase, the same code would use `dynamic_cast<MenuItem*>` instead. Either way, the Composite Pattern provides the structure, and the Iterator Pattern provides the flat traversal.

STL ranges and tree views

Modern C++20 generalizes this. The `<ranges>` library lets you compose iteration with filters, transforms, and slicing on top of any iterator. Once a `MenuComponent` hierarchy exposes an iterator, you can write:

```
// Pseudocode with std::ranges over a CompositeIterator
```

```
auto total = std::ranges::all_view(allMenus)
           | std::views::filter([](auto* c) { return isVegetarianItem(c);
})
           | std::views::transform([](auto* c) { return c->getPrice(); })
           | std::ranges::fold_left(0.0, std::plus{});
```

This is the long-term payoff of the two patterns. Once the tree is walkable by an iterator, every algorithm written in terms of iterators applies to it, including ones that didn't exist when the tree was designed. New summarising operations, new filters, new transformations, all of them work without modifying the tree code. The pattern composition is the foundation; the algorithms are the upper floors.

Real-World Uses

Both patterns are pervasive in production code. They almost always appear together because real-world hierarchies, files, GUI widgets, XML/JSON, and AST nodes are exactly the structures that benefit from uniform tree traversal.

File systems

- **Files and directories.** The classic example. A `File` is a leaf; a `Directory` is a composite holding files and subdirectories. Both implement a common `FilesystemNode` interface that supports `getName`, `getSize` (a directory's size is the total size of its children), `delete` (a directory recursively deletes its children), and so on. Every file-explorer GUI is walking a Composite with an Iterator.
- **Disk usage tools.** `du -sh` recursively sums file sizes, a textbook Composite traversal. Implementing it correctly without the pattern would require explicit recursion at every site that wanted the total size.

GUI widget hierarchies

- **Every windowed application.** A `Button` is a leaf; a `Panel` or `Window` is a composite holding child widgets. Both inherit from a common `Widget` interface that supports `draw`, `layout`, and `handleEvent`. When the window's draw method is called, it draws itself and recursively calls each child's draw method. When a mouse event fires, the topmost widget receives it and may dispatch to children. Composite is what makes the GUI tree work; Iterator is how the rendering pipeline walks it.
- **CSS box model.** Web browsers represent rendered HTML as a tree of boxes; each box is either an inline leaf, a block composite, or some intermediate. Layout, repaint, and hit-testing all walk the tree uniformly through a Composite-pattern interface.

Parsers and ASTs

- **Abstract Syntax Trees.** Every compiler and interpreter represents source code as a tree of expression nodes. Literals are leaves; binary operators are composites with two children; function calls are composites with a variable number of children; whole programs are composites of statements. Common visitor or interpreter passes walk the AST through iterators; type checkers, optimizers, and code generators are all Composite traversals.
- **XML and JSON parsers.** A document is a tree of elements, each of which may contain text leaves or sub-elements. XPath, XSLT, and JSONPath queries are all Iterator-Pattern walks over Composite-Pattern structures.

Game engines

- **Scene graphs.** 3D games arrange their world as a tree of `SceneNode` objects; some are concrete meshes (leaves), some are transform nodes holding other scene nodes (composites). Rendering walks the graph, accumulating transforms and submitting draw calls; physics walks the graph for collision detection; visibility culling walks it again. One pattern, many traversals.
- **Inventory systems.** Containers within containers, bags inside chests inside dungeons, are a Composite. Listing what the player owns is an Iterator over that Composite.

Office and creative software

- **Drawing applications.** In Inkscape, Illustrator, or Figma, a `Shape` is a leaf, and a `Group` is a composite. Selecting a group and rotating it rotates each child (which may itself be a group, recursively). Exporting walks the tree; saving to SVG/PDF walks the tree.
- **Word processors.** A document is a tree of sections, paragraphs, runs, and inline elements. Cursor navigation, search-and-replace, formatting, and rendering are all Composite traversals.

The library-level reusability

There is a meta-observation about all of the above examples. In each case, the tree of objects existed before anyone reached for a pattern; the question was always, "How do we write algorithms over this tree without coupling the algorithms to the tree?" The Iterator and Composite Patterns are how. They are not just "useful when you have a tree", they are the cost of admission for writing reusable code over trees. Without them, every algorithm is local to its tree; with them, algorithms become library-grade.

Summary

Two patterns, deeply related, both ubiquitous in real code.

An iterator in seven points

- **Decouple traversal from storage.** An aggregate holds elements in whatever shape suits it; an iterator walks them through a uniform interface.
- **Two interfaces, two responsibilities.** Aggregate (`createIterator` or `begin`/`end`) and Iterator (`hasNext`/`next`, or `*`/`++`/`!=`) are kept separate so that traversal state and storage are independently editable.
- **Uniform client code.** A single loop works over any aggregate that produces a compatible iterator. Generic algorithms are implemented as library functions rather than one-off code.
- **Single Responsibility Principle made concrete.** "Hold elements" and "walk elements" are two reasons to change; pulling them apart is exactly what the pattern does.
- **Multiple parallel iterations.** Each call to createIterator (or begin) produces a fresh iterator with its own state; many can coexist.
- **STL iterators are the production form.** Modern C++ uses `begin`/`end`, operator overloading, and iterator categories (input, forward, bidirectional, random-access) instead of a single Iterator interface.
- **Range-based for loops compile to iterators.** `for (auto& x : container)` is the pattern, made invisible by syntax.

Composite in seven points

- **Part-whole hierarchies.** When individual objects and groups of objects need to be treated the same way, give them a common interface.
- **Three participants.** Component (the common interface), Leaf (individual object), Composite (group that holds other Components).
- **Recursion in the structure, polymorphism in the interface.** The Composite holds Components, which may themselves be Composites; client code calls a method on a Component, and the correct behaviour is dispatched automatically.
- **Uniform client code.** The Waitress's printMenu does not test whether a node is a leaf or composite; she calls `print()` and lets the tree do the rest.
- **Transparency vs safety.** The textbook pattern violates Single Responsibility by putting structural and content operations into a single interface. The trade buys uniformity; the alternative buys type safety. Pick deliberately.
- **Pairs naturally with Iterator.** Trees are walked by iterators; once you have a Composite, a tree-flattening Iterator gives you arbitrary algorithms over the structure.
- **Ownership management is a real C++ concern.** Modern implementations use `std::unique_ptr` so destroying the root recursively frees the whole tree.

The two patterns in one sentence

An iterator hides how a collection stores its items; a composite hides whether a node is a single item or a subcollection.

Together, they let the client code walk arbitrarily complex hierarchical data with a single loop and a single method call. Once you have written code that uses both, once you have built a tree of components and reduced its traversal to a single line, you start seeing them everywhere in other people's code, and the experience changes how you read libraries forever.

Practice Problems

Problem 1

Implement an `ArrayIterator` class that walks a `MenuItem` C array of fixed size. It should expose `hasNext()` and `next()` methods. Then write a `LegacyMenu` class backed by a C array that implements the abstract `Menu` interface using your iterator. Show that the Waitress code from Section 4 works on it unchanged.

Problem 2

Read the following client code. What is wrong with it from the perspective of the Iterator Pattern?

```
void printMenu(PancakeHouseMenu& menu) {
    auto& items = menu.getMenuItems();   // returns std::vector<MenuItem>&
    for (size_t i = 0; i < items.size(); ++i) {
        std::cout << items[i].getName() << '\n';
    }
}
```

Rewrite the function in a way that fits the pattern, then explain why your version is more robust to changes in PancakeHouseMenu's storage.

Problem 3

Pick a class from a recent project of yours that you suspect might be doing too much. List, by hand, every reason that the class's source file might be edited over the next year. (Include things like: a new feature, a bug fix, an upstream API change, a performance optimization, a refactoring of an internal data structure.) If you list more than two distinct categories of reasons, propose a split into two or three classes that would each have a tighter set.

Problem 4

For each of the following STL containers, name the iterator category their iterators belong to (input, forward, bidirectional, random-access):

- `std::vector<int>`
- `std::list<int>`
- `std::forward_list<int>`
- `std::map<std::string, int>`
- `std::unordered_set<int>`

For each, name one `<algorithm>` function from the standard library that would NOT compile on it because the iterator category is insufficient.

Problem 5

Design a `FileSystemNode` hierarchy with a common base class and two derived classes `File` (leaf) and `Directory` (composite). Every node has a name and a size; a directory's size is the sum of its children's sizes. Pick the transparent variant of the pattern. Write the three classes and a `printTree` function that prints the hierarchy with indentation. Test it on a small example.

Problem 6

Take your file-system code from Problem 5 and refactor it to the safe variant of the Composite Pattern (structural operations only on `Directory`). Show how `printTree` must change to accommodate the type distinction. Which version do you prefer, and why? Under what circumstances would your preference flip?

Problem 7

Using your `FileSystemNode` code from Problem 5, write a `FileSystemIterator` that performs a depth-first walk of an entire directory tree. The iterator should yield every node (files and directories alike), in the order a `find` command would visit them. Use your iterator to write three functions:

- Total bytes in the tree.
- Count of `.cpp` files in the tree.
- Names of all directories deeper than depth 2.

All three should reuse the iterator; none of them should hand-code the recursion.

Problem 8

A colleague writes the following "Composite":

```
class Component {
public:
    virtual void render() const = 0;
};
class Square : public Component {
public:
    void render() const override { /* draw square */ }
};
class Document {
public:
    void render() const {
        for (auto& s : squares_) s->render();
    }
private:
    std::vector<std::unique_ptr<Square>> squares_;
};
```

Why is this *not* the Composite Pattern? What would have to change for it to qualify?

Problem 9

In the STL, modifying a container while you are iterating over it can invalidate the iterator and produce undefined behaviour. Look up the specific invalidation rules for `std::vector`, `std::list`, and `std::map`. For each container: which operations invalidate (a) all iterators, (b) some iterators, (c) no iterators? Why do the rules differ between the three containers? What does this tell you about the Iterator Pattern's hidden contracts?

Problem 10

Design a Composite hierarchy for arithmetic expressions: a `NumberNode` leaf holds a `double`; a `BinaryOpNode` composite holds a left child, a right child, and an operator character ('+', '-', '*', '/'). The common base class `ExprNode` exposes a method `eval()` that returns a `double`.

- Implement the three classes using the transparent variant.
- Construct the expression `(3 + 4) * (10 - 5)` and call `eval` on the root; verify it returns 35.
- Add a `toString()` method to the common interface that returns the expression as a string with parentheses.
- Now: write a function `countNumbers(const ExprNode& root)` that returns the count of `NumberNodes` in the tree. Implement it twice: once with type tests (`dynamic_cast`), once without (by adding a method to the base class). Discuss which is cleaner, and what design rule each version honours or violates.