CS 247

Software Engineering Principles

Chapter 11

The Decorator Pattern

Victoria Sakhnini

Contents

The Problem: Starbuzz Coffee

Imagine you have been hired by *Starbuzz Coffee*, a fictional chain that has grown faster than its software. The original menu was simple: four coffees (`HouseBlend`, `DarkRoast`, `Decaf`, `Espresso`), each priced individually. The original engineers, with their object-oriented hats on, designed it like this:

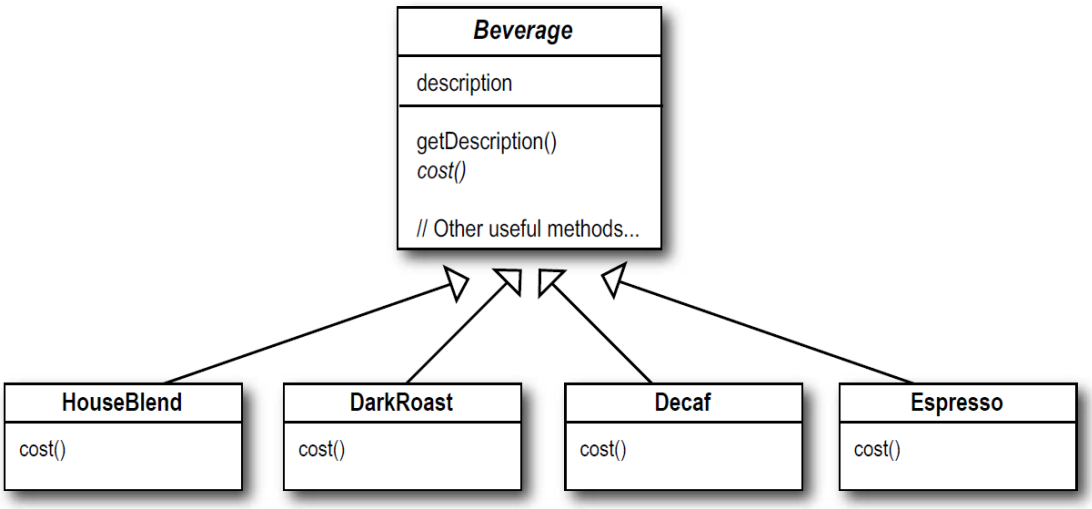

Figure 1.1: The original Starbuzz Beverage hierarchy. Beverage is an abstract class; each concrete coffee subclass implements cost().

Read the diagram carefully. `Beverage` is an abstract base class with a `description` field, a concrete `getDescription()` method that returns the description, and an *abstract* `cost()` method that every subclass must implement. Each of the four coffees inherits from `Beverage` and supplies its own `cost()`. This is textbook OO modelling, and as long as the menu stays that small, it works.

But customers want condiments

Then a product manager points out the obvious: real customers don't just order a *Dark Roast*; they order "a Dark Roast with two pumps of mocha, soy milk, and whipped cream". Starbuzz needs to handle that. The customer is charged a small extra amount for each condiment, the receipt has to list every modification, and the menu of available add-ons (steamed milk, soy, mocha, whip, caramel, cinnamon, ...) will itself grow over time.

Your first instinct, looking at the diagram above, might be "easy, just add condiment fields and subclasses". And that instinct is going to lead us, in the next section, into a very instructive disaster.

> **? Question**
>
> Before reading on, stop and try to design this yourself. How would you add condiments to the existing Starbuzz hierarchy? You can change Beverage, change the subclasses, add new subclasses, add data members, add methods, whatever you like. Try to come up with at least two designs, and ask yourself: which one will age poorly as the menu grows to 20 coffees and 15 condiments?

Why Inheritance Alone Isn't Enough

The most natural first attempt, and the one many people genuinely try, even outside the classroom, is to express "DarkRoast with Mocha" as a *subclass*. After all, "DarkRoast with Mocha" *is a* Beverage, so inheritance fits the IS-A relationship. The result is something like this:

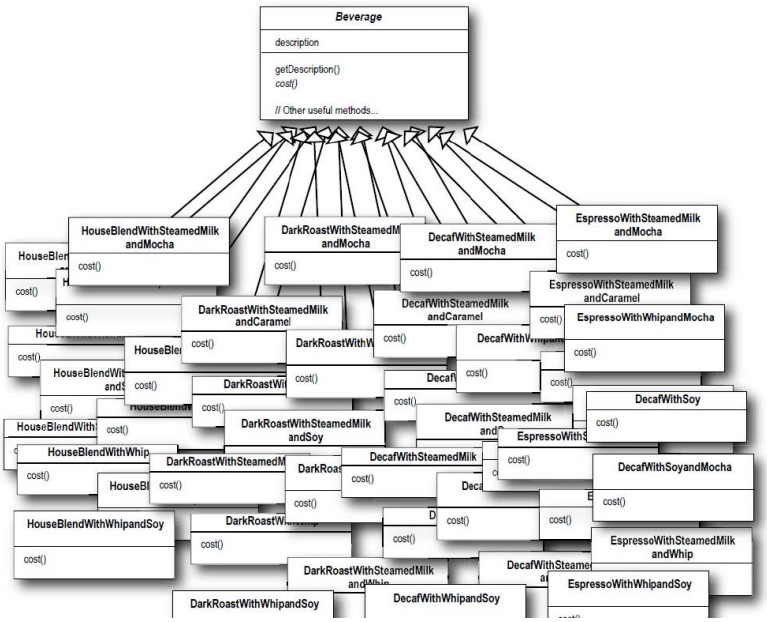

Figure 2.1: One subclass per (coffee × condiment combination). Four coffees, four condiments, and we already have a class hierarchy that is impossible to fit on a slide.

That diagram is not stylized. Every box is a real class that you would have to write, name, document, and test. With four coffees and four condiments, you already have 16 leaf classes just for single-condiment beverages, and a customer who orders Mocha *and* Whip needs another 16, and one who orders Mocha *and* Whip *and* Soy needs another 16, and so on. The full hierarchy explodes combinatorially in the number of condiments.

Counting the disaster

Let us count. With c coffee types and k condiments, and allowing each condiment to be present or absent independently, the number of distinct beverages is:

```
#beverages = c × 2^k

c = 4, k = 4   →   4 × 16  =   64 classes
c = 4, k = 8   →   4 × 256 = 1024 classes
c = 4, k = 12  →   4 × 4096 = 16 384 classes
```

The 2^*k* term is the killer, every new condiment *doubles* the number of classes you must maintain. And we haven't even mentioned "two pumps of mocha" versus "one pump of mocha", which would multiply this further. The combinatorial explosion is so severe that no organization, no matter how disciplined, can keep up with it.

What actually goes wrong

Class explosion isn't just about file counts. Every one of the symptoms below follows from it:

- **Code duplication.** The cost calculation for "add Mocha" lives inside every `*WithMocha` class. Change Mocha's price and you must edit dozens of files.
- **Forced ordering of decoration.** Is `DarkRoastWithMochaAndSoy` the same as `DarkRoastWithSoyAndMocha`? You have to decide and impose a naming convention; otherwise customers can order beverages that don't exist by name.
- **Inflexible to runtime composition.** If a customer asks to *add* Whip to their existing order, you can't simply tack it on, the existing object's type is fixed at construction. You would need to *destroy* the original and *create* a different subclass.
- **Closed to new condiments.** Adding caramel doubles the entire hierarchy. Every existing concrete class needs a `*WithCaramel` sibling, and every multi-condiment class needs to grow a caramel variant.
- **Hard to add new *coffees*.** A new coffee, say `ColdBrew`, needs a full set of `ColdBrewWithMocha`, `ColdBrewWithMochaAndWhip`, and so on. Every coffee × condiment combination must be re-spawned.

Every one of these symptoms is a property of the same root cause: we used inheritance to express something that is naturally *composition*. A Dark Roast with Mocha is not really "a special kind of Dark Roast". It is "a Dark Roast with an extra thing attached". The *attachment* is the part that inheritance can't model gracefully, and that is exactly the gap the Decorator Pattern fills.

> ⚠ **Warning**
>
> **Inheritance is structurally rigid.** Class definitions are fixed at compile time. You cannot dynamically add or remove a base class while the program runs. If your problem domain involves *orthogonal features* that compose freely, like coffee × condiment, or window × scrollbar × border × shadow, inheritance always fights you. Reach for composition instead.

The Open–Closed Principle

Before we name the pattern, the lecture deck stops to introduce a design principle, and it is worth pausing to absorb it, because it explains *why* the pattern looks the way it does.

> ⚡ **Tricky**
>
> **Design Principle: Open–Closed.** Classes should be **open for extension** but **closed for modification**.

This sounds like a contradiction the first time you read it. "Open for extension" means you can add new behaviour. "Closed for modification" means you cannot edit the existing source. How do you add behaviour to code without editing it?

The answer is: through *hooks* the original code already provides. Polymorphism is one such hook; a base class declares a virtual method, and a subclass adds new behaviour by overriding it, without the base needing any change. Composition is another; a class accepts an object via a constructor parameter, and you can supply a new flavour of that object without modifying the host class. Both of these are forms of extension that leave the existing source untouched.

The pay-off of Open–Closed is concrete:

- **Resilience to change.** Code that has been deployed for months and tested by thousands of users does not need to be re-tested every time a new feature is added.
- **Reduced regression risk.** If you don't edit the existing class, you cannot introduce a regression in the existing class.
- **Third-party extensibility.** Other teams (or other companies, when you ship a library) can add new behaviour without forking your code.
- **Smaller diffs.** Code reviews are easier, version control history is cleaner, and merge conflicts are rarer.

But don't take it too far

Open–Closed is a principle, not a rule, and naïve readings of it lead to worse code, not better. Three common over-applications:

- **Designing for hypothetical extensions.** Every *possible* extension point you add costs in complexity. If no one will ever extend a class in a given direction, leaving it closed, then it is fine.
- **Avoiding all modifications, forever.** Sometimes the right fix is to edit the existing class. Open–Closed says "prefer extension when extension is genuinely available"; it does not say "never refactor".
- **Confusing "open for extension" with "open for everything".** A class can be open for one kind of extension (e.g., new condiments) while remaining closed for others (e.g., the internal pricing algorithm).

The Starbuzz problem is *exactly* the situation where Open–Closed earns its keep: we know the menu of condiments will keep growing, and we want to avoid modifying the existing Beverage, DarkRoast, or Espresso code whenever a new condiment is invented.

> **Tip**
>
> **How to spot a place where OCP helps.** Look for axes of variation in your domain that are known to keep growing: file formats, payment methods, pricing rules, and condiments. If those things are going to expand, design the interface so new ones can be added without editing the old ones. Conversely, if an axis is finite and stable ("days of the week"), don't bend the design over backwards to make it extensible.

Definition: The Decorator Pattern

Here is the GoF definition, word for word:

> **Tricky**
>
> **Decorator Pattern.** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Three phrases are doing the heavy lifting:

- **"Additional responsibilities."** A decorator does not *replace* the wrapped object's behaviour; it *augments* it. The original cost calculation still runs; the decorator adds something to it. The original description is still produced; the decorator appends to it.
- **"Dynamically."** Decoration happens at runtime, not at compile time. The decision to add Mocha or Whip can come from a customer interaction, a configuration file, or any other runtime signal, not from the type system.
- **"Alternative to subclassing."** Where you might be tempted to write a subclass to add behaviour, write a decorator instead. The wrapped object is composed into the wrapper; the type of the wrapper itself is fixed, but what it wraps is not.

Two intent statements

The lecture deck offers two complementary phrasings worth memorizing:

- "Gives your objects new responsibilities without making code changes to the underlying classes."
- "Gives extension at runtime rather than at compile time."

Together with the GoF definition, these three sentences tell you everything you need to know about when to reach for the pattern: when you need to *add* responsibilities, *after* the class hierarchy is already designed, *at runtime*, and you *cannot* or *should not* modify the existing code.

The core trick

The trick at the heart of the Decorator Pattern is short enough to fit in two sentences:

> **Tricky**
> A *decorator* is a class that **implements the same interface as its target** and **holds an instance of that interface**. Each method on the decorator forwards to the wrapped object's method, optionally adding behaviour before or after.

If you take only one sentence away from this chapter, take that one. Everything else, the UML diagrams, the C++ code, the trade-offs, is bookkeeping around those two sentences.

Anatomy of the Pattern

The canonical UML class diagram for the Decorator Pattern looks like this:

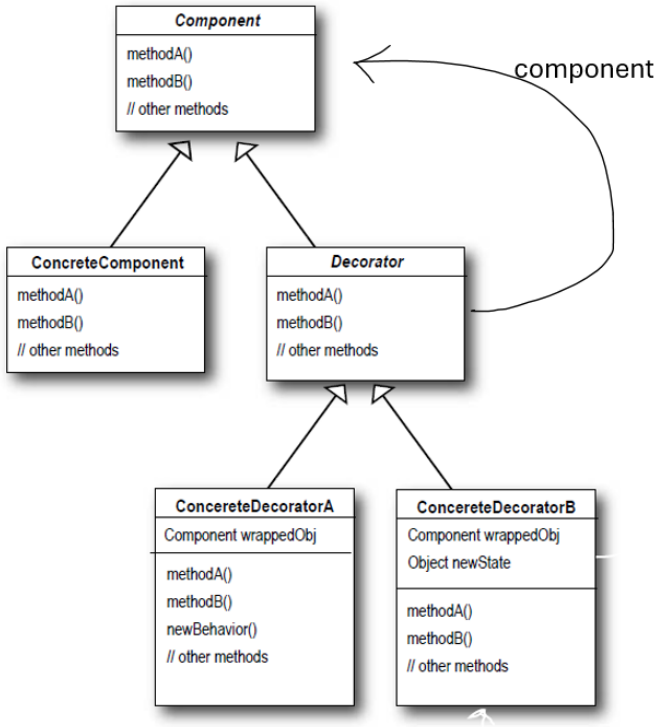

Figure 5.1: The Decorator Pattern. Both ConcreteComponent and Decorator implement Component; ConcreteDecorator inherits from Decorator and holds a Component reference.

There are four roles, two hierarchies, and one critical association. Take them in order.

Component

Top of the diagram. **Component** is the abstract interface (or abstract base class) shared by both the things that can be decorated *and* the decorators themselves. It declares the methods that clients will call, `methodA`, `methodB`, and so on. In our Starbuzz example, Component will be `Beverage` with its `cost()` and `getDescription()` methods.

Note that Component is *not* itself abstract in every implementation; it can be a concrete class, an interface, or an abstract class with some defaulted methods. What matters is that every participant in the decoration hierarchy shares this common type, so clients can treat them uniformly.

ConcreteComponent

Bottom-left of the diagram. **ConcreteComponent** is one of the "basic" objects, the kind you can use on its own, with no decoration. In Starbuzz these will be `HouseBlend`, `DarkRoast`, `Decaf`, `Espresso`, each implementing `cost()` with its own price and returning its own description.

In a Java I/O example, this is `FileInputStream`. In a GUI library, it is the plain `Window`. In a logging library, it is the bare `ConsoleLogger`. These are the things at the centre of the onion; everything else wraps them.

Decorator

Bottom-middle-right of the diagram (the abstract one). **Decorator** is the abstract base class for all decorators. It inherits from Component (so it can be used wherever a Component is expected), and it holds a `Component wrappedObj`, the thing it is wrapping.

Note the curved arrow in the diagram going from Decorator back up to Component. That is the *"has-a"* relationship, the decorator *contains* a Component reference. This double role of Decorator ("is a Component AND has a Component") is the structural heart of the pattern.

ConcreteDecorator

Bottom of the diagram. **ConcreteDecorator** classes are the actual flavours of decoration. Each one adds a specific behaviour. In Starbuzz, they will be Mocha, Soy, Milk, and `Whip`. Each implements the Component methods by calling the wrapped object's version *and* adding its own contribution. Mocha returns `wrapped->cost() + 0.20` and `wrapped->getDescription() + ", Mocha"`.

Some ConcreteDecorators also add *new* methods that the underlying Component didn't have (note `newBehavior()` on the diagram). This is allowed but rare. The cleanest decorators only forward and augment; they don't expand the interface.

The wrapping intuition

There is a visual intuition for what decoration looks like at runtime that the diagram doesn't convey. Imagine the ConcreteComponent as an *inner* object. Each decorator is a layer wrapped around it, like the layers of an onion or the paper nesting around a gift. From the outside, the wrapped object looks like a Component. When you call a method on it, the outermost layer responds, possibly delegating to the next layer, which delegates to the next, until the innermost ConcreteComponent runs.

That delegation chain is where the magic happens. Each layer is independent. The order of the layers determines the order of the contributions. The same set of decorators can be combined in many orders, producing different results. "Dark Roast with Mocha then Whip" might be structurally identical to "Dark Roast with Whip then Mocha" for cost purposes, but the description string differs.

Designing Starbuzz with Decorators

Time to map the canonical pattern onto our concrete problem. Here is the Starbuzz design after applying the Decorator Pattern:

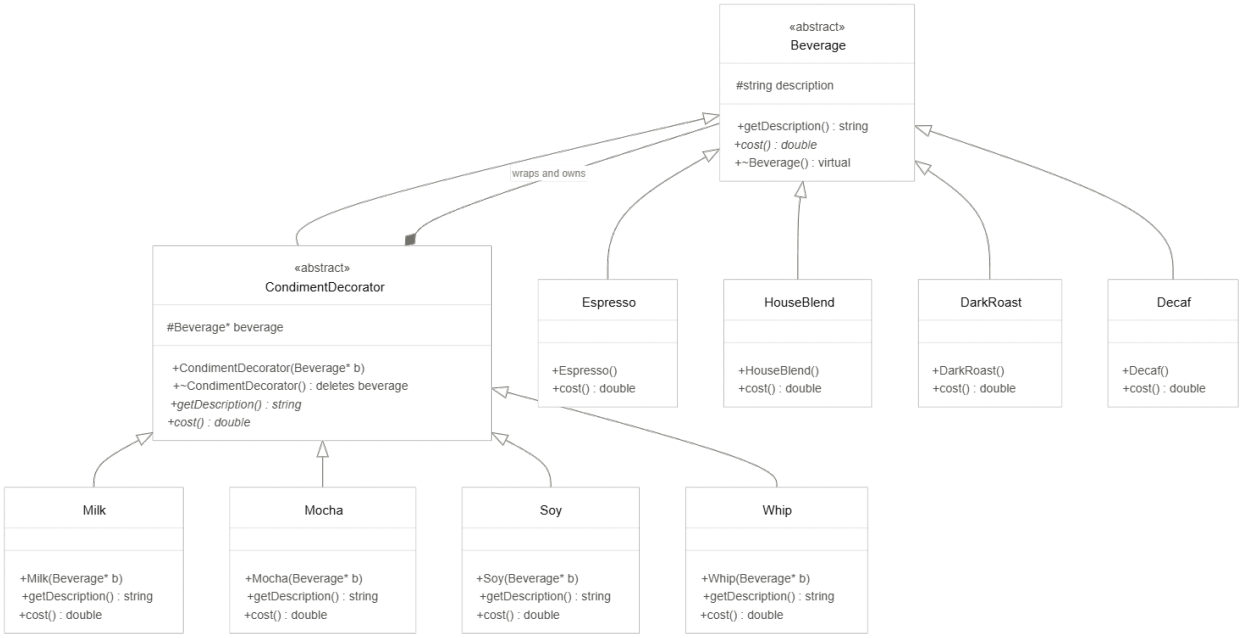

Figure 6.1: The Starbuzz design with Decorator. Beverage is the Component. HouseBlend/DarkRoast/Decaf/Espresso are ConcreteComponents. CondimentDecorator is the abstract Decorator. Milk/Mocha/Soy/Whip are ConcreteDecorators that each wrap a Beverage.

Compare this to the canonical UML in Figure 5.1 and verify the mapping:

Canonical role	Starbuzz class
Component	Beverage (abstract base)
ConcreteComponent	HouseBlend, DarkRoast, Decaf, Espresso
Decorator	CondimentDecorator (abstract)
ConcreteDecorator	Milk, Mocha, Soy, Whip

Notice what is **not** in the diagram: there is no `DarkRoastWithMocha`, no `EspressoWithSoyAndWhip`, no `HouseBlendWithMochaAndMochaAndWhip`. The combinatorial explosion has vanished. With 4 coffees and 4 condiments, we have exactly 4 + 4 + 1 (the abstract decorator) = 9 leaf classes. Add a fifth

condiment, caramel, and we add *one* class. Add a sixth coffee, cold brew, and we add *one* class. Linear growth instead of exponential.

What a runtime object looks like

Let us trace one example. A customer orders "Dark Roast with Mocha and Whip". At runtime, the object graph for this order is a chain of wrappers, each pointing to the next:

```
+----------+        +----------+        +-----------+
|   Whip   |  --->  |  Mocha   |  --->  | DarkRoast |
| beverage |        | beverage |        |           |
+----------+        +----------+        +-----------+

The outer object (Whip) has-a Mocha.
The Mocha has-a DarkRoast.
The DarkRoast is the ConcreteComponent.
```

When a client calls `cost()` on the outermost Whip object, the call chain looks like:

```
Whip::cost()        // adds whip price, then delegates to its beverage
  └─> Mocha::cost() // adds mocha price, then delegates to its beverage
        └─> DarkRoast::cost()   // returns the base price (no further delegation)
        └─< 0.99
  └─< 0.99 + 0.20 = 1.19
└─< 1.19 + 0.30 = 1.49      // returned to the client

Total: $1.49
```

Each call in the chain adds its contribution and forwards. The innermost call hits the ConcreteComponent, which returns the base value. Then the recursion unwinds, each layer adding its contribution on the way back. The same trick produces the description "Dark Roast Coffee, Mocha, Whip", each layer appends its name to whatever the inner layer produced.

Crucially, the order of the wrapping determines the order of the contributions. "Whip(Mocha(DarkRoast))" and "Mocha(Whip(DarkRoast))" cost the same dollar amount (addition is commutative) but produce different description strings, and could produce wildly different results for non-commutative operations like "Uppercase" and "AddBorder" (which we will see in the practice problems).

Implementing Starbuzz in C++

Eight C++ source files. We will build them in the order a designer typically picks them up: the Component first, then the ConcreteComponents, then the abstract Decorator, then the ConcreteDecorators, then `main.cpp`. We use std::unique_ptr throughout for safe, ownership-clear resource management.

The Component: Beverage

File `beverage.h`:

```
#ifndef BEVERAGE_H
#define BEVERAGE_H

#include <string>

class Beverage {
protected:
    std::string description = "Unknown Beverage";
public:
    virtual ~Beverage() = default;
    virtual std::string getDescription() const {
        return description;
    }
    virtual double cost() const = 0;
};

#endif // BEVERAGE_H
```

Three details to absorb:

- **Protected description with a default.** `std::string description = "Unknown Beverage"`, initialized at the declaration. Subclasses inherit this field and can overwrite it in their constructors. The default value makes it easy to detect a coffee subclass that forgot to set its description.
- **Virtual destructor.** Because Beverage will be used polymorphically through pointers, the destructor must be virtual so that `delete beveragePtr` correctly invokes the most-derived destructor. We default it to the compiler-generated implementation, since there is nothing to clean up in the interface itself.
- **getDescription is virtual but not pure.** The base provides a default implementation (return the description field), and subclasses can either inherit it as-is (the coffees do) or override it (the decorators do). cost(), in contrast, is *pure* virtual (`= 0`), every subclass must implement it.

> **Tricky**
>
> **Why protected, not private?** The `description` field is `protected` because the coffee subclasses need to write to it inside their constructors. We could have provided a `setDescription` method instead and kept the field private, which is arguably cleaner; the lecture chose the more concise protected approach.

ConcreteComponent: DarkRoast

The first coffee. The pattern is identical for the others, Espresso, HouseBlend, Decaf, so we will only show one in full and skip the rest. Header `darkroast.h`:

```
#ifndef DARKROAST_H
#define DARKROAST_H

#include "beverage.h"

class DarkRoast: public Beverage {
public:
    DarkRoast();
    double cost() const override;
};

#endif // DARKROAST_H
```

And the implementation `darkroast.cpp`:

```
#include "darkroast.h"

DarkRoast::DarkRoast() {
    description = "Dark Roast Coffee";
}

double DarkRoast::cost() const {
    return 0.99;
}
```

Eleven lines of code total. The constructor sets the inherited description; `cost()` returns a fixed price. No condiment logic, no "and Mocha" branches. The coffee is the coffee. The condiments will be applied externally through wrapping, which is the whole point.

The other three coffees follow the same template:

```
// espresso.cpp
#include "espresso.h"

Espresso::Espresso() {
    description = "Espresso";
}
double Espresso::cost() const {
    return 1.99;
}
```

Adding a new coffee in the future means writing a new pair of files. No edits to any existing class.

The abstract Decorator: CondimentDecorator

This is where the structural cleverness of the pattern lives. The abstract decorator inherits from `Beverage` (so a decorated beverage IS-A Beverage from the type system's point of view) *and* holds a `Beverage*` (so it can wrap another Beverage).

Header `condimentdecorator.h`:

```
#ifndef CONDIMENTDECORATOR_H
#define CONDIMENTDECORATOR_H

#include "beverage.h"
#include <memory>
class CondimentDecorator: public Beverage {
protected:
    std::unique_ptr<Beverage> beverage;  // the component being decorated
public:
    explicit CondimentDecorator(std::unique_ptr<Beverage> b):
         beverage(std::move(b)) {}
    ~CondimentDecorator() override = default;   // unique_ptr cleans up
    std::string getDescription() const override = 0;
    double cost() const override = 0;
};

#endif // CONDIMENTDECORATOR_H
```

This file has every important moving part of the Decorator Pattern packed into 17 lines. Walk through them:

- **Inheritance from Beverage.** `class CondimentDecorator : public Beverage`, this is what makes "Mocha around Espresso" still a Beverage. Without this, clients couldn't treat decorated and undecorated beverages uniformly.
- **Protected Beverage pointer.** `Beverage* beverage`, the thing being wrapped. Protected so concrete decorators can read it in their cost() and getDescription() overrides.
- **Constructor takes the wrapped object.** `CondimentDecorator(std::unique_ptr<Beverage> b) : beverage(std::move(b)) {}`, initialiser-list assignment. The decorator does not allocate the inner beverage; the caller does, and passes ownership in.
- **Destructor is defaulted.** `~CondimentDecorator() override = default;` The `unique_ptr` member owns the wrapped beverage and destroys it automatically when the decorator is destroyed — no manual `delete` needed. When the outermost decorator goes out of scope, the whole chain unwinds automatically.
- **Pure virtual overrides.** Both `getDescription` and `cost` are re-declared `= 0` so that concrete decorators must supply their own. Note `getDescription`, we override it even though Beverage gave a default, because every condiment will want to append its name to the inner description.

> ⚠ **Warning**
>
> **Ownership is enforced by the type system.** Because the wrapped beverage is held as a `std::unique_ptr`, the compiler prevents accidental sharing. You cannot wrap the same beverage twice — doing so requires `std::move`, which nulls out the source, making any second use a detectable bug rather than silent memory corruption. There are no manual `delete` calls, and the chain is exception-safe: if a constructor throws, every already-constructed layer is properly destroyed.

ConcreteDecorator: Mocha

With the abstract decorator in place, the concrete decorators are tiny. Header `mocha.h`:

```
#ifndef MOCHA_H
#define MOCHA_H

#include "condimentdecorator.h"

class Mocha : public CondimentDecorator {
public:
```

```
    Mocha(std::unique_ptr<Beverage> b);
    std::string getDescription() const override;
    double cost() const override;
};

#endif // MOCHA_H
```

And the implementation `mocha.cpp`:

```
#include "mocha.h"

Mocha::Mocha(std::unique_ptr<Beverage> b): CondimentDecorator(std::move(b))
{}

std::string Mocha::getDescription() const {
    return beverage->getDescription() + ", Mocha";
}

double Mocha::cost() const {
    return 0.20 + beverage->cost();
}
```

Eleven lines, again. The constructor delegates to the parent. The two methods do exactly what the GoF definition promised: they call the wrapped object's method *and add Mocha's contribution*. The description appends ", Mocha" to whatever the inner beverage was called. The cost adds 20 cents to whatever the inner beverage cost.

The same template applies to Soy, Whip, and Milk:

```
// soy.cpp
Soy::Soy(std::unique_ptr<Beverage> b): CondimentDecorator(std::move(b)) {}
std::string Soy::getDescription() const {
        return beverage->getDescription() + ", Soy";
}
double Soy::cost() const { return 0.15 + beverage->cost(); }

// whip.cpp
Whip::Whip(std::unique_ptr<Beverage> b): CondimentDecorator(std::move(b))
{}
std::string Whip::getDescription() const {
        return beverage->getDescription() + ", Whip"; }
double Whip::cost() const { return 0.10 + beverage->cost(); }

// milk.cpp  not included in main
```

```
Milk::Milk(std::unique_ptr<Beverage> b): CondimentDecorator(std::move(b))
{}
std::string Milk::getDescription() const {
        return beverage->getDescription() + ", Steamed Milk";
}
double Milk::cost() const { return 0.10 + beverage->cost(); }
```

Adding a new condiment in the future, say Caramel, requires exactly one new `caramel.h` / `caramel.cpp` pair, with the same shape. No edits to any existing file. This is the Open–Closed Principle in action.

The driver

Finally, `main.cpp` wires up three example orders and prints their descriptions and costs:

```
#include <iostream>
#include <iomanip>      // for setprecision
#include <memory>
#include <string>

#include "beverage.h"
#include "espresso.h"
#include "darkroast.h"
#include "houseblend.h"
#include "mocha.h"
#include "soy.h"
#include "whip.h"

int main() {
    std::cout << std::fixed << std::setprecision(2);

    // Order 1: plain Espresso
    auto beverage = std::make_unique<Espresso>();
    std::cout << beverage->getDescription()
              << " $" << beverage->cost() << std::endl;

    // Order 2: Dark Roast with two Mochas and Whip
    auto beverage2 = std::make_unique<DarkRoast>();
    beverage2 = std::make_unique<Mocha>(std::move(beverage2));
    beverage2 = std::make_unique<Whip>(std::move(beverage2));
    std::cout << beverage2->getDescription()
              << " $" << beverage2->cost() << std::endl;

    // Order 3: House Blend with Soy, Mocha, and Whip
    auto beverage3 = std::make_unique<HouseBlend>();
    beverage3 = std::make_unique<Soy>(std::move(beverage3));
    beverage3 = std::make_unique<Mocha>(std::move(beverage3));
```

```
    beverage3 = std::make_unique<Whip>(std::move(beverage3));
    std::cout << beverage3->getDescription()
              << " $" << beverage3->cost() << std::endl;
    std::cout << "done!!!!" << std::endl;

    // unique_ptrs clean up automatically; no delete needed.
    return 0;
}
```

There are a few things worth pulling apart in this driver.

- **Output formatting.** `std::cout << std::fixed << std::setprecision(2)`, sets cout to print floats with exactly two decimal places. The `std::fixed` stops it from switching to scientific notation, and `setprecision(2)` caps the precision at the hundredths column, which is what we want for currency.
- **Reassignment using move.** `beverage2 = std::make_unique<Mocha>(std::move(beverage2))`, read this carefully. `std::move(beverage2)` transfers ownership of the current DarkRoast into the new Mocha constructor, then the assignment overwrites `beverage2` with the new Mocha. Result: `beverage2` now owns a Mocha that owns the original DarkRoast.
- **Stacking decorators.** The `beverage2 = std::make_unique<Mocha>(std::move(beverage2))` line is then *repeated*, wrapping a Mocha around the Mocha that already wraps DarkRoast. This is a double mocha. Then Whip wraps the whole thing.
- **Automatic cleanup for three orders.** When each `unique_ptr` goes out of scope at the end of `main`, it walks down the chain destructor-by-destructor. The Whip's destructor destroys the Mocha, which destroys the inner Mocha, which destroys the DarkRoast. The entire stack unwinds automatically with no manual `delete` calls required.

Compile and run

From a command line:

```
$ g++ -std=c++17 main.cpp \
       condimentdecorator.cpp \
       espresso.cpp darkroast.cpp houseblend.cpp \
       mocha.cpp soy.cpp whip.cpp -o shop
$ ./shop
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
done!!!!
```

Read the output one line at a time and verify the math:

- **Espresso $1.99.** Plain espresso. No wrapping, no decoration; just `Espresso::cost()` returning 1.99.
- **Dark Roast Coffee, Mocha, Mocha, Whip $1.49.** DarkRoast (0.99) + two Mochas (0.20 each) + Whip (0.10) = 0.99 + 0.20 + 0.20 + 0.10 = 1.49. The description string was built up by recursive appends, so it reads from innermost to outermost: "Dark Roast Coffee" → "Dark Roast Coffee, Mocha" → "Dark Roast Coffee, Mocha, Mocha" → "Dark Roast Coffee, Mocha, Mocha, Whip".
- **House Blend Coffee, Soy, Mocha, Whip $1.34.** HouseBlend (0.89) + Soy (0.15) + Mocha (0.20) + Whip (0.10) = 1.34.

This is the entire Decorator Pattern in 100 lines of C++. The pattern's elegance is not in any one file, every file is short, but in the way the pieces *combine*. Four coffees, four condiments, runtime composition, linear class growth, and no edits to anything when we want to add a fifth condiment.

Caveats and Modern C++ Refinements

The Decorator looks tidy in the diagram and concise in code, but real-world use has rough edges. The lecture deck flags several in its notes section, and you should know about them before shipping decorators into production C++. We will cover ownership (the big one), order-dependence, the proliferation of small objects, and a few language-specific concerns.

Ownership: enforced by std::unique_ptr

The implementation in Section 7 uses `std::unique_ptr<Beverage>` throughout, expressing the ownership convention directly in the type system. The concrete benefits are:

- **Single ownership is enforced by the compiler.** Wrapping the same beverage twice requires `std::move`, which nulls out the source. A second use of the moved-from pointer is detectable rather than silent memory corruption.
- **No manual delete calls.** The chain unwinds automatically when the outer `unique_ptr` goes out of scope. The destructor on `CondimentDecorator` is defaulted; the `unique_ptr` member does all the work.
- **Exception safety is automatic.** If a constructor throws partway through building the chain, every successfully constructed layer is properly destroyed. No leak is possible.

The only visible cost at call sites is the `std::move` noise during construction; a small price for the safety guarantees gained.

Order dependence

Decorator chains are sometimes order-sensitive. Let us think about three cases:

- **Commutative cost.** For Starbuzz, `Mocha`, and Whip, each adds a fixed amount to the cost. Order doesn't matter for the price.

- **Description ordering.** Wrapping `Whip(Mocha(DarkRoast))` produces "Dark Roast Coffee, Mocha, Whip", while `Mocha(Whip(DarkRoast))` produces "Dark Roast Coffee, Whip, Mocha". The strings differ. Whether this matters depends on whether you sort or compare descriptions.
- **Non-commutative operations.** In the text-formatting practice problem at the end of this chapter, `Uppercase(Border(t))` and `Border(Uppercase(t))` produce different results; one uppercases the asterisk border, the other does not. Order matters intensely.

The lesson: decorators are not a commutative algebra. Treat the chain order as part of the design. If two compositions are supposed to be equivalent, document that they are; do not assume it.

Lots of small objects

A chain of `Whip(Mocha(Mocha(DarkRoast)))` has four objects on the heap and four virtual calls for every `cost()` or `getDescription()`. For a coffee shop, this is fine; for a graphics pipeline processing millions of pixels per second through a decorator chain, the cost of indirection can be high.

Mitigations, in order of cost-effectiveness:

- **Don't optimize prematurely.** Most Decorator chains are short and called rarely. Measure before worrying.
- **Cache results when callers ask.** If a client computes the description of a chain once per second, cache it.
- **Collapse the chain at construction time.** Pre-compute the total cost once when the chain is finalized, store it, and have every layer's cost; just return the cached value. (Now you can't decorate further without rebuilding, but that may be fine.)
- **Use the Curiously Recurring Template Pattern (CRTP).** Templates can compose decoration at compile time, eliminating the virtual dispatch entirely. This is what makes some C++ libraries (Eigen, expression templates) so fast. The slide deck calls this out in passing; see the speaker's note on slide 19. It is a significant change in style; we will not pursue it here.

Interface bloat is the limit

Every method in the Component interface must be implemented *by every decorator*. If `Beverage` gains a third method `temperature()`, every decorator class must override it (typically just by forwarding to `beverage->temperature()`). A handful of methods is fine; an interface with 30 methods makes decorators tedious.

The Decorator Pattern is sensitive to interface size for this reason. If the component interface is large and the decorations only touch one or two methods, consider whether each decorator should add a *filter* instead, that is, hold a reference to a smaller interface and let the wrapper itself implement the rest by direct delegation through a template or a helper. But these are advanced refinements; the straight Decorator handles the typical case well.

Comparing two decorated objects is tricky

Two beverages that produce the same description and cost may differ structurally. `Whip(Mocha(DarkRoast))` and `Whip(Mocha(DarkRoast))`, constructed separately, are two different objects with two different sets of pointers, even though they cost the same and describe the same. Equality testing on decorator chains usually means comparing the externalized state (description, cost) rather than the internal structure.

This is rarely a problem in practice; most clients of the Decorator Pattern do not test object identity, but it is worth knowing if you ever find yourself implementing `operator==` for a decorated type.

Decorators in the Wild

Once you know the shape, "object that implements interface X and holds an X", you start spotting Decorator everywhere. The deck calls out three rich domains; here they are with a few additions.

Java's java.io

The standard example, and the one the deck embeds. Java's input/output classes are organized as a textbook Decorator hierarchy. The abstract Component is `InputStream`. ConcreteComponents are `FileInputStream`, `ByteArrayInputStream`, `StringBufferInputStream`, each producing bytes from a different source. The abstract Decorator is `FilterInputStream`, which holds an `InputStream` and forwards reads to it.

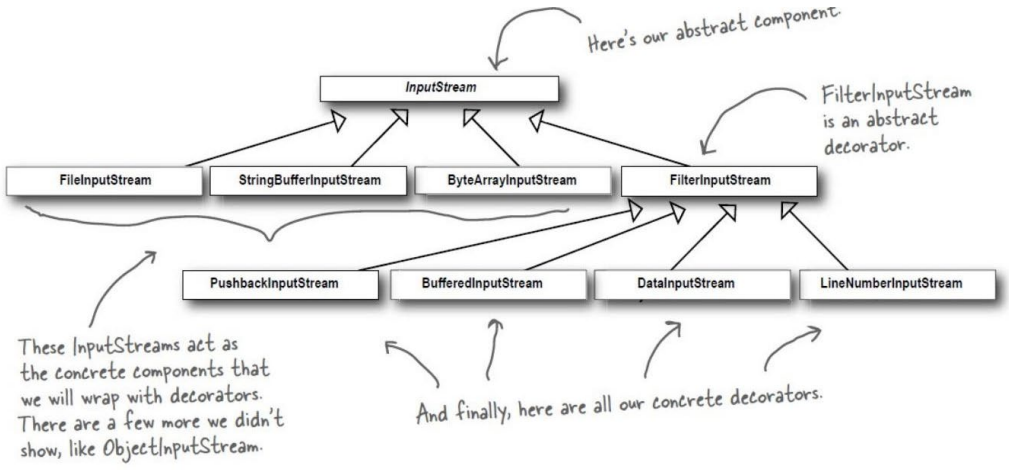

Figure 9.1: Java's java.io hierarchy. InputStream is the abstract component. FileInputStream/ByteArrayInputStream/StringBufferInputStream are concrete components. FilterInputStream is the abstract decorator; PushbackInputStream, BufferedInputStream, DataInputStream, LineNumberInputStream are concrete decorators that wrap any other InputStream and add a behaviour (buffering, line counting, data parsing, push-back-by-one).

Each concrete decorator adds a single capability:

- `BufferedInputStream`, wraps any stream and adds a memory buffer, so reads are faster.
- `DataInputStream`, wraps any stream and adds methods to read typed values (read an int, read a double, read a UTF-8 string).
- `LineNumberInputStream`, wraps any stream and tracks how many newlines have been consumed.
- `PushbackInputStream`, wraps any stream and lets you "un-read" one byte (useful for tokenisers).

And, because they all implement `InputStream`, they can be stacked freely: a `DataInputStream` wrapping a `BufferedInputStream` wrapping a `FileInputStream` gives you typed reads, buffered, from a file. That is exactly the same call pattern as our Mocha-wraps-DarkRoast example. Java's `java.io.OutputStream` hierarchy, `Reader`, and `Writer` follow the same shape.

Once you see this, reading Java I/O documentation goes from "why are there so many classes?" to "oh, of course, these are pluggable decorators around a few simple sources".

Unix pipelines

The pipe operator in a shell is essentially the Decorator Pattern at the operating-system level. Each command in

```
cat file.txt | grep "pattern" | sort | uniq
```

reads a stream of bytes from its predecessor and writes a transformed stream of bytes to its successor. Every command has the same interface: "read from stdin, write to stdout". Every command can be inserted anywhere in the chain. The chain composes freely; the order matters; the components are independent.

If you think of the file (`file.txt`) as the ConcreteComponent and `grep`, `sort`, `uniq` as ConcreteDecorators, the analogy is exact, except that here "the interface" is the Unix file descriptor convention rather than a C++ abstract class. The pattern shows up in countless tools: streaming media pipelines, log processors, ETL frameworks, and Unix's own design philosophy.

GUI widgets

Most graphical user interface toolkits use decorator-style wrapping for visual augmentation. Take a plain `Window`, a rectangular area that draws itself. You can wrap it in a `ScrollableWindow` that adds scroll bars when the content overflows, or a `BorderedWindow` that draws a frame around it, or a `ShadowedWindow` that adds a drop shadow. Each wrapper implements the same `Window` interface (draw, resize, get-bounds) and forwards calls to the wrapped instance while adding its own contribution.

The Java AWT and Swing libraries do this; so do the older Apple Cocoa frameworks; so does the Visitor / Decorator hybrid in many game-engine scene graphs. When you see a class hierarchy where every "add-this-visual-effect" is a separate class, and they all implement the same drawable interface, that is the Decorator Pattern.

Logging frameworks

Logging libraries are the other classic case. A bare `Logger` writes a message to a sink. Wrap it in:

- `TimestampLogger`, prepends the current time to every message.
- `LevelLogger`, prepends INFO / WARN / ERROR / DEBUG.
- `ThreadIdLogger`, adds the thread id.
- `FilterLogger`, drops messages below a threshold.
- `RateLimitLogger`, drops messages if too many are coming in.

Each layer is independent; each can be added or removed at runtime; the chain can be composed freely. Java's `log4j` and the .NET `Microsoft.Extensions.Logging` pipelines are textbook examples. So is the Python `logging` module's handler chains.

Where you might NOT use Decorator

Worth mentioning briefly. Decorator is the wrong tool when:

- The augmentations are *known and finite* at compile time; a switch statement or a strategy might be simpler.
- Order does not matter, and the augmentations are commutative; a single flag-field approach might be simpler.
- There is *only one* possible augmentation; there is no need for a pattern.
- The augmentations need to know about each other. Decorator deliberately keeps them independent; if they aren't independent, the pattern is fighting you.

Decorator vs. Other Patterns

Decorator is structurally similar to several other patterns, and students routinely confuse them. The table below lays out the differences for the patterns you have met so far in CS 247, plus two ahead of you in the course.

Pattern	Shape	Key intent
Decorator	Object wraps another object of the same type	Add responsibility to an object at runtime

Pattern	Shape	Key intent
Observer	Subject holds a list of observers	Notify many objects when one object changes
Strategy	Object holds an algorithm object	Swap an algorithm at runtime
Composite	Object holds many children of the same type	Treat one and many uniformly (tree structures)
Adapter	Object wraps another object of a different type	Convert one interface to another
Proxy	Object wraps another object of the same type	Control access (caching, lazy load, security)

Three of these, Decorator, Adapter, and Proxy, all have the shape "object wraps another object". Distinguishing them is a frequent exam question:

- **Decorator: same interface, augmented behaviour.** The wrapper IS-A Component and HAS-A Component. The point is to *add* responsibilities.
- **Adapter: different interface.** The wrapper exposes one interface and wraps an object exposing a *different* interface. The point is to *translate*.
- **Proxy: same interface, controlled access.** The wrapper looks identical to its target but adds a guard (caching, access control, lazy initialization). The point is to *control*, not to augment.

The visual heuristic: Decorator *adds*; Adapter *converts*; Proxy *guards*. The diagrams are similar; the intent is different.

Decorator vs. Composite

Composite (which you will meet later in the course) has a strikingly similar diagram to Decorator. Both involve a hierarchy with a base class and a class that holds references to the base. The difference is what they're modelling:

- **Composite: trees of things.** A Composite node *contains many* Component children; it models part-whole hierarchies (a folder contains files; a paragraph contains spans). Recursion is a *data structure*.
- **Decorator: stacked behaviours.** A Decorator wraps *exactly one* Component; it models behavioural augmentation. The recursion is *behavioural*.

In practice, both patterns can appear in the same hierarchy. A GUI tree of widgets uses Composite for the parent-child structure and Decorator for the scrollbar/border/shadow effects.

Summary

The Decorator Pattern dynamically attaches additional responsibilities to an object. It provides a flexible alternative to subclassing for extending functionality.

It is built from four roles:

- **Component**, abstract base class or interface for both decorated objects and decorators.
- **ConcreteComponent**, a basic, undecorated object that implements Component.
- **Decorator**, an abstract class that inherits from Component *and* holds a Component reference.
- **ConcreteDecorator**, a concrete subclass of Decorator that adds a specific augmentation.

The core trick: *the decorator implements the same interface as its target and holds an instance of that interface*. Each method on the decorator forwards to the wrapped object's method, optionally adding behaviour before or after.

The pattern's value:

- Eliminates the combinatorial class explosion caused by inheriting one subclass per feature combination.
- Allows runtime composition of responsibilities, features can be added or omitted per-instance.
- Honours the Open–Closed Principle: new decorators can be added without modifying existing classes.
- Plays well with other patterns: I/O streams, GUI widgets, logging, ETL pipelines, and Unix pipes are all decorator-shaped.

Things to remember when you implement it in C++:

- Use a virtual destructor on the Component so destruction through a base pointer cleans up correctly.
- Decide an ownership policy *explicitly*: this implementation uses `std::unique_ptr`, which enforces "decorator owns wrapped object" in the type system rather than in prose.
- Watch for order-dependent compositions, decoration is not always commutative.
- Be cautious with Component interfaces that have many methods; every decorator must implement them all.
- Don't reach for Decorator when a flag, a single subclass, or a simple strategy would do.

And remember the family resemblance: Decorator *adds*; Adapter *converts*; Proxy *guards*; Composite is a *tree*. The diagrams are alike; the intent is what tells them apart.

Practice Problems

Problem 1.

For each of the following systems, identify the Component, ConcreteComponent(s), and at least two plausible ConcreteDecorators:

(a) A PDF document viewer that can optionally annotate, watermark, encrypt, or sign the displayed pages.

(b) A pizza delivery app where you can add bubble-wrap insulation, ice-pack, gift-message-card, or signature-required to any order.

(c) A vehicle insurance policy where the base policy is liability-only, and customers can add collision coverage, comprehensive coverage, roadside assistance, or rental reimbursement.

(d) A messaging library where outgoing messages can be optionally compressed, encrypted, signed, or rate-limited.

(e) A drawing program with shapes (circle, square, triangle) that can optionally be filled, outlined, rotated, or shadowed.

Problem 2.

Suppose we extend Starbuzz with a new condiment `Caramel` that costs $0.25 and appends ", Caramel" to the description. Trace the call sequence and the final result for the following order:

```
Beverage* b = new HouseBlend();           // 0.89
b = new Soy(b);                           // +0.15
b = new Caramel(b);                       // +0.25
b = new Whip(b);                          // +0.10
std::cout << b->getDescription()
          << " $" << b->cost() << '\n';
```

(a) Draw the runtime object diagram (the chain of wrappers, innermost to outermost).

(b) List the sequence of `cost()` calls in order, what calls what, and what value each one returns.

(c) Do the same for `getDescription()`.

(d) If the customer then changes their mind and removes the Whip, what code would you write to do this without leaking memory? (Note: this is harder than it looks with raw pointers, explain why.)

Problem 3.

Consider three possible ownership models for a decorator chain, and analyse the trade-offs:

(a) Decorator owns wrapped object via raw pointer (the lecture's model). List the rules that a client must follow to use this correctly. What goes wrong in each of these scenarios?

- Client wraps the same Beverage twice with two different decorators.
- Client deletes the inner Beverage manually, then deletes the outer decorator.
- Client puts the outer decorator into a `std::vector` of Beverage pointers and lets the vector's destructor run.

(b) Decorator holds wrapped object via `std::unique_ptr`. Re-examine the three scenarios above, which ones become compile errors, which become runtime errors, and which become correct behaviour?

(c) Decorator holds wrapped object via `std::shared_ptr`. What new use cases does this enable? What new pitfalls does it open? (Hint: think about cycles.)

(d) Justify which of the three models you would choose for a production Starbuzz system, and why.

Problem 4.

Consider four decorators for a `Text` type that wraps a `std::string`:

- `Uppercase`, converts every letter to uppercase.
- `Reverse`, reverses the entire string.
- `Border`, surrounds the string with three asterisks on each side ("*** Hello ***").
- `Indent`, prepends four spaces.

Starting from the input string "Hello World", state the output for each of the following compositions:

(a) `Uppercase(Reverse("Hello World"))`

(b) `Reverse(Uppercase("Hello World"))`

(c) `Border(Uppercase("Hello World"))`

(d) `Uppercase(Border("Hello World"))`

(e) `Indent(Border(Uppercase("Hello World")))`

(f) `Border(Indent(Uppercase("Hello World")))`

(g) Identify a pair of decorators in this list that *are* commutative (order doesn't change the result for this input).

Problem 5.

Starting from the Starbuzz code in Section 7, add a new condiment `Caramel` that costs $0.25 and appends ", Caramel" to the description.

(a) Write the header file `caramel.h`.

(b) Write the implementation file `caramel.cpp`.

(c) List the files in the existing Starbuzz code (other than `main.cpp`, which is unaffected by the Decorator Pattern) that you had to modify. The list should be empty.

(d) Now add an order in `main.cpp` that builds a Caramel-Mocha-DarkRoast and prints its description and cost. Verify the cost arithmetic by hand.

(e) Compare this with what you would have to do under the inheritance-only design from Section 2. Estimate how many files you would need to create or modify if Caramel were to be combined with the existing six condiments (Mocha, Soy, Whip, Steamed Milk, etc.) across the four coffees.

Problem 6.

For each scenario, decide whether the Decorator pattern is appropriate. If yes, sketch the Component and at least two decorators. If no, identify a better pattern or design and explain why.

(a) An online shop's checkout flow has exactly four steps that always run in the same order: validate cart, calculate tax, charge card, and send receipt. The order never changes; steps are never skipped.

(b) A graphics editor lets users apply visual filters to an image: blur, sharpen, sepia, and brighten. Filters can be combined in any order; the user can preview the chain and reorder or remove individual filters at runtime.

(c) A sort routine needs to compare values using one of three comparison policies (ascending numeric, descending numeric, lexicographic). The policy is chosen once when the sort begins and remains unchanged.

(d) A web framework lets developers add middleware around an HTTP handler: authentication, logging, gzip compression, and rate limiting. Each middleware processes the request, optionally calls the next layer, and processes the response.

(e) A `Stack<T>` data structure offers push, pop, top, size, and empty. Users want a variant that tracks the maximum element seen so far (`getMax()`).

Problem 7.

Create a text processing system where you can apply multiple formatting operations to a string. Implement a `Text` interface with a `render()` method that returns a `std::string`. Create decorators for:

- **Bold formatting**, wraps text in `<b>...</b>` tags.
- **Italic formatting**, wraps text in `<i>...</i>` tags.
- **Uppercase transformation**, converts every letter to uppercase.
- **Asterisk border**, wraps text with `"*** "` and `" ***"`.

Allow these decorators to be combined in any order.

(a) Sketch the UML class diagram for your design. Identify the Component, ConcreteComponent, abstract Decorator, and ConcreteDecorators.

(b) Implement the full system in C++. Use `std::unique_ptr` for ownership.

(c) Write a small main that produces and prints the following four compositions of the string "Hello":

- `Bold(Italic(Hello))`
- `Border(Uppercase(Hello))`
- `Uppercase(Border(Hello))`
- `Italic(Border(Uppercase(Bold(Hello))))`

(d) Identify two compositions in (c) that produce strings of equal length, but that differ visually. Identify two compositions that produce identical strings.

Problem 8.

Design a pizza ordering system using the Decorator Pattern. Start with base pizzas (Margherita, Pepperoni) and allow customers to add toppings (olives, mushrooms, extra cheese, peppers). Each topping adds to the cost and to the description. Additionally, implement special *preparation* decorators like "Gluten-Free Crust" or "Stuffed Crust" that modify both the description and price.

Challenge: Ensure that certain decorators (the crust modifiers, there can be at most one) can only be applied once, while toppings can be added multiple times.

(a) Sketch a class diagram. Distinguish the topping decorators from the crust decorators in some way (different parent classes? a marker interface? something else?).

(b) Implement the topping decorators normally. They follow the standard pattern.

(c) Implement the crust decorators with the at-most-one constraint enforced. Discuss at least two possible enforcement strategies:

- (i) Runtime check, when a crust decorator is constructed, walk the wrapped chain and throw if it already contains another crust decorator.
- (ii) Type-level enforcement, make a separate abstract class `CrustPizza` that is *not* itself a Pizza-decorator-of-arbitrary-Pizza, but a once-only outermost wrapper that takes a `Pizza` and exposes `Pizza`'s interface; so the chain order is enforced ("crust last").

(d) Write a main that constructs the following orders and prints the description and cost of each:

- Margherita with extra cheese and mushrooms.
- Pepperoni with olives, olives (yes, double olives), and Gluten-Free Crust.
- Margherita with Stuffed Crust and Gluten-Free Crust, and verify that the at-most-one constraint kicks in.

(e) Reflect: did you have to choose between expressiveness and constraint enforcement? Which strategy from (c) would you ship, and what trade-off did you accept?

Problem 9.

Write a C++ program that demonstrates the Decorator Pattern by applying a sequence of text transformations specified on the command line. Your executable must be named `tform` and run like this:

```
./tform "Hello World" upper underscore reverse
```

The first argument is the input string (assume it is quoted if it contains spaces). The remaining arguments are zero or more transformation names. Your program must apply transformations in the order given and print the final result as a single line.

Design requirements (from the deck):

- Define an abstract base class (e.g. `Text`) with a virtual destructor and a single virtual method `render() const` that returns a `std::string`.
- Implement `PlainText` that stores the original string and returns it unchanged.
- Implement an abstract decorator class `TextDecorator` that inherits from `Text` and wraps another `Text`.
- Implement each transformation as its own concrete decorator class that overrides `render()`.
- Support these three transformations: `upper` (uppercase letters), `reverse` (reverse the whole string), `underscore` (replace spaces with _). Transformations may repeat and must compose correctly.

(a) Sketch the UML class diagram.

(b) Implement the full program. Use `std::unique_ptr` throughout. Process command-line arguments with a simple registry/factory: a `std::map<std::string, std::function<std::unique_ptr<Text>(std::unique_ptr<Text>)>>` that maps transformation names to factory lambdas. Iterate `argv[2]` through `argv[argc-1]` and wrap accordingly.

(c) Verify your implementation produces these outputs:

```
./tform "Hello World"                    => Hello World
./tform "Hello World" upper               => HELLO WORLD
./tform "Hello World" upper reverse       => DLROW OLLEH
./tform "Hello World" upper underscore    => HELLO_WORLD
./tform "Hello World" upper underscore reverse  => DLROW_OLLEH
./tform "Hello World" reverse upper       => DLROW OLLEH
./tform "Hello World" upper upper         => HELLO WORLD   (idempotent)
./tform "Hello World" reverse reverse     => Hello World   (involution)
./tform "ab cd"        underscore underscore => ab_cd      (idempotent if no spaces
left)
```

(d) Adding `lower`, a transformation that lowercases all letters, to your program should require touching exactly two places: the factory registration map, and a new `LowerDecorator` class. List those two changes explicitly.

(e) Stress-test composition: state, with justification, the output of:

```
./tform "" upper reverse                      (empty input)
./tform "Hello"                               (no transformations)
./tform "a"                                   (single char, no transformations)
./tform "a" upper reverse upper reverse       (idempotent + involution combined)
```

(f) Reflect on what your program would have to look like *without* the Decorator Pattern, perhaps using one massive `if/else` chain or a switch statement that handles every combination of transformations. What advantage does the decorator approach offer when you go from three transformations to thirty?