CS 247

Software Engineering Principles

Chapter 13

Adapter and Façade Patterns

Victoria Sakhnini

Table of Contents

Adapters All Around Us 5
Definition: The Adapter Pattern 6
Why "convert" and not "rewrite" 7
Two kinds of adapters 7
Anatomy: Target, Adaptee, Adapter, Client 7
The translation, in one sentence each 8
Object Adapter vs. Class Adapter 8
The Object Adapter 8
The Class Adapter 9
Side-by-side comparison 10
Worked Example: Enumeration to Iterator 10
The mapping 11
The class diagram 12
Implementing the Enumeration Adapter in C++ 13
The Target, Iterator.h 13
The Adaptee, Enumeration.h 14
A concrete Adaptee, ConcreteEnumeration 14
The Adapter, EnumerationAdapter 15
The Client, main 16
A Second Example: Adapting a Legacy Rectangle 17
The setup 17
The adapter 18
Using the adapter 19
Caveats and Modern C++ Refinements 20
Implementing an adapter can be easy or hard 20
Smart-pointer choices 20
std::optional for translated return values 20
Move semantics for efficiency 21
const correctness 21
Adapters are not magic. 21
Adapters in the Wild 22
Network protocols 22
Database drivers 22

Payment gateways 22
Graphics-card drivers 23
Logging frameworks 23
Operating-system file APIs 23
The footprint 23
The Other Pattern in This Module 24
Definition: The Façade Pattern 25
The class diagram 25
What the Façade does (and does not) do 26
The Home Theatre Example 26
The subsystem 27
Watching a movie, the hard way 27
Watching a movie, with a Façade 29
Implementing the Home Theatre Façade in C++ 30
The Façade, header 31
The Façade, watchMovie and endMovie 32
Using the Façade, main 33
Principle of Least Knowledge 34
The rule, in code 35
How the Façade Pattern applies 35
When to push back 36
Adapter vs. Façade vs. Decorator 36
The three-way comparison table 37
Shape vs. intent, the canonical confusion 37
What about Proxy? 38
A worked discrimination 38
Façades and Adapters in the Wild 39
Compiler façades 39
Game-engine initialization 39
Email client façades 39
Payment façades 39
React's component model 39
std::filesystem 39
The footprint 40
Summary 41

The Adapter Pattern 41
The Façade Pattern 41
The shared moral 42
Practice Problems 43

Adapters All Around Us

Imagine you have just flown from Toronto to Paris with your laptop. You arrive at your hotel, find a wall outlet, pull out your charging cable, and discover the prongs do not fit. The outlet exposes one interface, three vertical slots for European plugs, while your laptop expects a different one: two flat prongs and a round ground pin. The outlet works perfectly well, the laptop works perfectly well, but they cannot talk to each other because their interfaces are incompatible.

You do not throw away the laptop, nor do you rewire the hotel

. You buy a *plug adapter*: a small device that plugs into a European wall outlet on one side and exposes a North American socket on the other. It does not generate electricity, change its voltage, or convert AC to DC. All it does is *translate* between two incompatible physical interfaces so that two things that could not previously connect can.

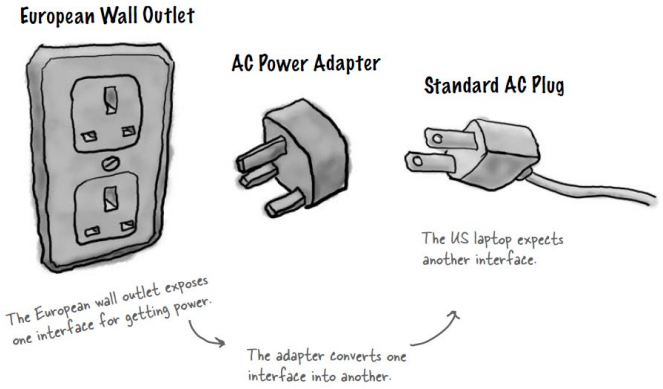

Figure 1.1: A physical adapter.

This is exactly the situation we run into constantly in software. You have an existing system, your code, that calls into some other class to get work done. The other class might be a library you bought, a legacy module written years ago by someone who has since left, a third-party SDK whose source you cannot modify, or the C API of an operating system. Whatever it is, the names, parameters, and return types of its methods are not the ones your code expects.

Concretely: your system calls `processPayment(double amount)`, but the vendor's class only offers `chargeStripe(double amount, std::string currency)`. The two systems are not at war; they simply speak different dialects. You could rewrite either side, but neither option is attractive. Rewriting your own system means a sprawl of changes across every call site. Rewriting the vendor's class is impossible without the source, and unwise even with it; you do not want to maintain a fork forever.

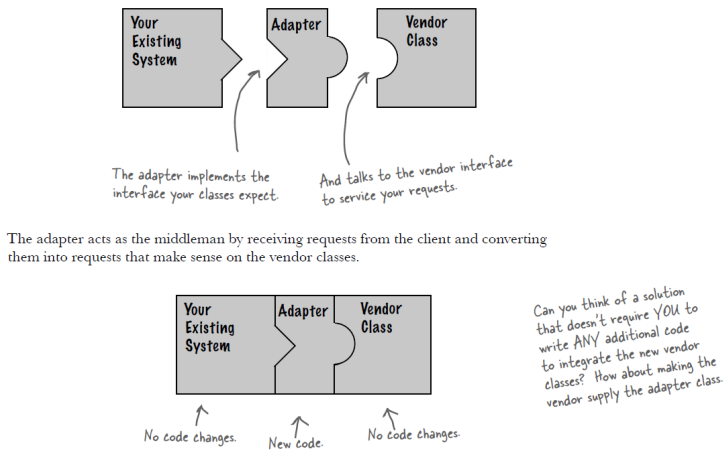

Figure 1.2: The puzzle-piece view.

The annotation in the bottom diagram is worth lingering on: "*No code changes* / *New code* / *No code changes*." This is the Adapter Pattern's value proposition in a single line. The two original parties stay exactly as they were. The only thing you write is the middle piece, the adapter itself. And, as the side note observes, if the vendor is reasonable, they may even ship adapter classes for you, so that the integration cost falls to zero.

Definition: The Adapter Pattern

Following the Gang-of-Four template you have used throughout this course, the Adapter Pattern is defined as follows.

> **Tip:**
>
> **Adapter Pattern (Intent).** Convert a class's interface into another interface that clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces.

Three nouns in that sentence carry the entire definition:

- **Interface,** the set of method names, parameter types, and return types that a class exposes to its callers.
- **Client,** the code that uses the interface, and that you do not want to modify.
- **Adapter,** a new class whose public interface is the one the client expects, and whose private workings forward to the class that actually does the work.

Put together: an *adapter* is a class that lets two classes with mismatched interfaces collaborate, by sitting between them and translating in both directions.

Why "convert" and not "rewrite"

Notice the word the GoF definition uses: *convert*. Not *rewrite*, not *re-implement*, not *port*. The Adapter Pattern is fundamentally about renaming and reshaping, not about reimplementing logic. The class being adapted continues to do all the real work; the adapter just makes the calls reach it in a form it can understand, and the answers come back in a form the client can understand.

This is why an adapter is normally a small class. If your "adapter" has hundreds of lines of business logic in it, something has gone wrong; you are not adapting anymore, you are rewriting, and you should ask whether the legacy class was actually doing what you needed in the first place.

Two kinds of adapters

The original GoF book identifies two distinct realizations of the same intent:

- **Object Adapter,** the adapter *composes* the adaptee. It holds an instance (typically through a pointer) and forwards calls to it. This is the dominant form in modern C++.
- **Class Adapter,** the adapter *inherits* from both the target interface and the adaptee. Each call enters through the inherited target interface and is forwarded to the adaptee parent's implementation. This form requires multiple inheritance.

Anatomy: Target, Adaptee, Adapter, Client

Before we look at the two forms, let us nail down the four participants whose names appear on every Adapter diagram in the GoF book and in every textbook since. These names are the shared vocabulary you will reach for when you discuss an adapter design with a teammate.

Role	What it is	Who writes it
Target	The interface the client expects to call. Often, an abstract base class or a pure virtual interface in C++.	You, or someone before you.
Client	The code that talks to Target and never knows anything else exists.	You, and you do not want to change it.
Adaptee	The existing class has an interface that is wrong for the client. Could be vendor code, legacy code, or anything else you cannot or should not modify.	Someone else, often before you arrived.
Adapter	The new class that IS-A Target (so the client accepts it) and HAS-A or IS-A Adaptee (so it can do the work).	You. This is the only file you add.

Read those four rows together, and the pattern becomes a story: the *client* speaks to the *target*; the *adapter* implements the target but delegates the work to an *adaptee*. Each role has exactly one responsibility and a clear reason to be in the design.

The translation, in one sentence each

- **Client → Adapter:** the client calls a Target method on what it believes is a Target. It is right, the Adapter IS-A Target.
- **Adapter → Adaptee:** the adapter receives the call, reshapes the arguments if needed, and invokes the corresponding Adaptee method (which has a different name and possibly different parameters).
- **Adaptee → Adapter:** the Adaptee returns its native result. The adapter reshapes the return value as needed (different type, units, or error convention) before returning it to the client.
- **Adapter → Client:** the client gets back a value of the type its Target interface promised, and never knows an Adaptee was involved.

Every adapter you write, small or large, simple or fiddly, fits into this four-step dance. The only thing that varies is *how much* reshaping happens in each step.

> ❓ **Question:**
>
> What would it mean, structurally, if the Adapter did not implement the Target interface? Why would the pattern fall apart?
>
> **Answer.** If the adapter did not implement Target, the client could not hold a reference to it via a Target pointer or call Target methods. The whole point is that the client thinks it is talking to a Target. If the adapter exposed a *different* interface, then the client would have to know about that interface, and we would be back at square one, the client coupled to a class whose name and methods it should not need to know.

Object Adapter vs. Class Adapter

There are two ways to make a class "IS-A Target and connected to an Adaptee". The first uses composition; the second uses multiple inheritance. Both achieve the GoF Adapter intent, but the structural diagrams look different, and the trade-offs differ. Let us look at each.

The Object Adapter

In an Object Adapter, the adapter *inherits* from Target (so the client can pass it where a Target is expected) and *holds a member* of type Adaptee (so it can delegate the work). Calls flow as follows: the

client calls a Target method; the adapter receives the call; the adapter calls a method on its stored Adaptee instance and returns the (possibly translated) result.

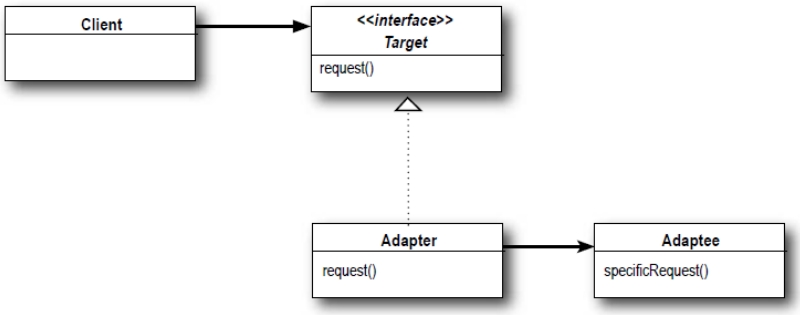

Figure 4.1: The Object Adapter UML. Client depends on the abstract Target interface; Adapter is a concrete Target (dashed inheritance line indicates interface realization); Adapter holds a reference to an Adaptee (solid arrow with diamond, indicating composition); when Target.request() is called on the Adapter, the Adapter calls Adaptee.specificRequest().

Notes:

- **Any subclass of the adaptee works.** Because the adapter holds the adaptee through a *pointer to the adaptee base*, you can pass it any concrete subclass, and the polymorphic dispatch will work. The adapter does not care whether it was given a `Vector`-based enumeration or a `LinkedList`-based one.
- **Many adapters, one client.** Because the client only knows the Target interface, you can drop in multiple adapters that translate from different sources. Want a new backend? Write a new adapter. The client does not change.
- **Adapters added after the fact.** If, six months from now, a new class appears whose interface does not match the target, you can write a new adapter for it without modifying any existing code. The Open–Closed Principle in action.

The Class Adapter

In a Class Adapter, the adapter *inherits* from both Target and Adaptee. It implements the Target methods by calling the inherited Adaptee methods directly; no member field is needed.

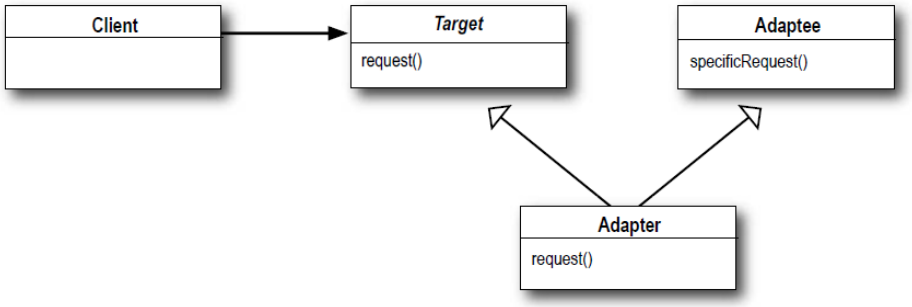

Figure 4.3: The Class Adapter UML. Both arrows from Adapter are open triangles indicating inheritance, the Adapter is-a Target AND is-an Adaptee. There is no composition arrow; the Adapter inherits the Adaptee's state and methods directly.

Side-by-side comparison

Aspect	Object Adapter	Class Adapter
Relationship to Target	Inherits (IS-A)	Inherits (IS-A)
Relationship to Adaptee	Composition (HAS-A) via pointer or reference	Inheritance (IS-A)
Requires multiple inheritance?	No	Yes
Adapts subclasses of Adaptee?	Yes, automatically (polymorphism)	No, must add a new adapter per subclass
Override Adaptee behaviour?	Hard (you'd need a subclass of Adaptee)	Easy, just override in the adapter
Decided at	Runtime, swap adaptees by passing different objects	At compile time, inheritance is fixed
Recommended in modern C++	Yes, strongly preferred	Only when you specifically need to override Adaptee behaviour

The bottom line: "Object Adapters are recommended over class adapters in modern C++ because they use composition (more flexible, supports runtime adaptation) rather than multiple inheritance (more rigid, compile-time only)." For the rest of this chapter, we focus on the object adapter form.

Worked Example: Enumeration to Iterator

Time for the running example. Two interfaces with the same purpose but different names often appear in real codebases when an older API is being phased out in favour of a newer one. The classic case is the `Enumeration` / `Iterator` pair from Java's standard library.

In early Java, the way to walk through a collection was the `Enumeration` interface, which exposed two methods:

```
interface Enumeration {
    boolean hasMoreElements();
    Object  nextElement();
}
```

Later, Java added the `Iterator` interface, which has the same semantics but different method names, plus an additional `remove()` operation for editing the underlying collection mid-walk:

```
interface Iterator {
    boolean hasNext();
    Object  next();
    void    remove();
}
```

The two interfaces do the same job; they were just designed years apart by different people, and the new code has nothing to do with the old names. We often encounter legacy code that exposes the Enumeration interface, yet we want our new code to use only Iterators. It looks like we need to build an adapter.

The mapping

Before writing any code, work out the mapping between the two interfaces. This is the entire "design" of an adapter; once the mapping is clear, the implementation is mechanical.

Iterator method (Target)	Maps to Enumeration method (Adaptee)
hasNext()	hasMoreElements(), exact correspondence, just renamed.
next()	nextElement(), exact correspondence, just renamed.
remove()	(no equivalent), Enumeration cannot be removed. The adapter must decide what to do.

That last row is the interesting one. The Target interface promises a `remove()` operation; the Adaptee does not support it. The adapter cannot invent the capability out of thin air. The standard solutions are:

- **Throw an exception** (the original Java approach: throw `UnsupportedOperationException`). Honest, but means the adapter cannot be a drop-in replacement for an iterator that actually does support removal.
- **Silently ignore the call**. Easier on callers, but silently wrong if the caller depended on the removal happening.
- **Document the limitation** and let the caller deal with it.

In our C++17 version, we will simplify further: the `Iterator` interface drops `remove()` entirely, because the example focuses on the renaming-only translation. In production code, you would have to choose one of the three options above for any operation that the Adaptee cannot perform.

The class diagram

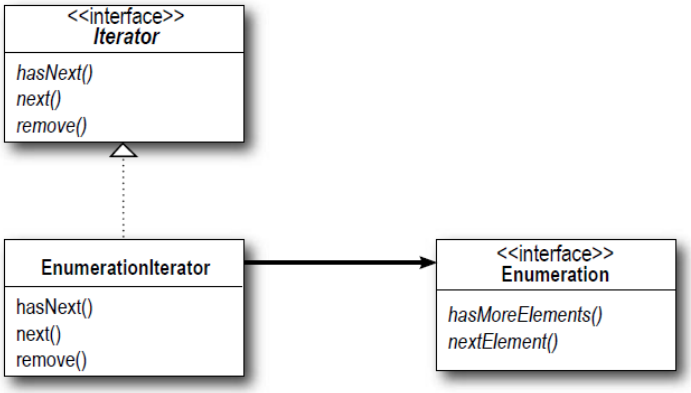

Figure 5.1: The plain UML.

> **Tip:**
>
> The key relationships:
>
> - Client wants: Iterator interface
> - We have: Enumeration interface (incompatible)
> - Adapter: Implements Iterator interface (so client is happy)
> - Adapter contains: Enumeration object (so it can delegate to it)
>
> **Adapter Pattern components:**
>
> - **Target Interface (Iterator):** What the client code expects
> - **Adaptee (Enumeration):** The existing incompatible interface
> - **Adapter (EnumerationAdapter):** Translates between Target and Adaptee
> - **Client (main):** Uses the Target interface, unaware of adaptation

With the roles labelled and the mapping written out, the implementation is now mostly typing. Let us write it.

Implementing the Enumeration Adapter in C++

The implementation in modern C++17 uses three building blocks you have seen earlier in the course: pure-virtual interfaces, smart pointers, and `std::optional`. Together, they make the adapter both safe and idiomatic.

The Target, Iterator.h

First, the Target interface. This is what the client will call. It is a pure virtual base class with two operations: "`hasNext`" and "`next`". Returning an `std::optional<int>` from `next` is a deliberate design choice: rather than throwing or returning a sentinel, the adapter tells the caller "there is no next element" by returning `std::nullopt`.

```
#ifndef ITERATOR_H
#define ITERATOR_H
 #include <optional>
class Iterator {
public:
    [[nodiscard]] virtual bool hasNext() const = 0;
    [[nodiscard]] virtual std::optional<int> next() = 0;
    virtual ~Iterator() = default;
};
#endif // ITERATOR_H
```

Three details worth noting in this five-line interface:

- **[[nodiscard]]** on both methods means the compiler will warn if a caller writes `iterator.next();` and ignores the result. For an `optional` return type, this is exactly what you want; ignoring the optional means losing the value.
- **hasNext() is const,** because asking whether there are more elements should not change the iterator's position. (Whether the underlying adaptee can honour this depends on the adaptee, of course.)
- **The virtual destructor** is mandatory for any base class that will be deleted through a base-class pointer. Skipping it is a classic way to leak resources from a polymorphic hierarchy in C++.

The Adaptee, Enumeration.h

Next, the Adaptee interface. Same shape as Iterator, different names. In a real legacy system, you would not be writing this; it would already exist in the form you cannot change. Here we mock it up so the example is self-contained.

```
#ifndef ENUMERATION_H
#define ENUMERATION_H

#include <optional>

class Enumeration {
public:
    [[nodiscard]] virtual bool hasMoreElements() const = 0;
    [[nodiscard]] virtual std::optional<int> nextElement() = 0;
    virtual ~Enumeration() = default;
};

#endif // ENUMERATION_H
```

A concrete Adaptee, ConcreteEnumeration

To show the adapter actually working, we need at least one concrete Adaptee. `ConcreteEnumeration` wraps a `std::vector<int>` and walks it from front to back. (The lecture notes uses a `std::vector`, but any container would do; that flexibility is the point of having an Enumeration interface in the first place.)

```
#ifndef CONCRETE_ENUMERATION_H
#define CONCRETE_ENUMERATION_H
#include "Enumeration.h"
#include <vector>
#include <optional>

class ConcreteEnumeration : public Enumeration {
private:
    std::vector<int> elements;
    mutable size_t index;
public:
    explicit ConcreteEnumeration(std::vector<int> elems);
    [[nodiscard]] bool hasMoreElements() const override;
    [[nodiscard]] std::optional<int> nextElement() override;
};
#endif // CONCRETE_ENUMERATION_H
```

```
#include "ConcreteEnumeration.h"

ConcreteEnumeration::ConcreteEnumeration(std::vector<int> elems)
    : elements(std::move(elems)), index(0) {}

bool ConcreteEnumeration::hasMoreElements() const {
    return index < elements.size();
}

std::optional<int> ConcreteEnumeration::nextElement() {
    if (index >= elements.size()) {
        return std::nullopt;
    }
    return elements[index++];
}
```

Notice the use of `std::move(elems)` in the constructor's member-initializer list. Without `std::move`, the vector would be copied; with it, the vector is moved, which for a non-trivial vector is dramatically cheaper.

The Adapter, EnumerationAdapter

Now the star of the show. `EnumerationAdapter` inherits from `Iterator` (so it IS-A target) and holds a `std::unique_ptr<Enumeration>` (so it HAS-A adaptee). The `unique_ptr` expresses that the adapter exclusively owns the adaptee; when the adapter is destroyed, so is the wrapped enumeration. If you wanted the adaptee to be shareable, you would use `std::shared_ptr` instead, but for a wrapper that is the sole user of its adaptee, `unique_ptr` is the right choice.

```
#ifndef ENUMERATION_ADAPTER_H
#define ENUMERATION_ADAPTER_H

#include "Iterator.h"
#include "Enumeration.h"
#include <memory>
#include <optional>
// Adapter from Enumeration to Iterator
class EnumerationAdapter : public Iterator {
private:
    std::unique_ptr<Enumeration> enumeration;
public:
    explicit EnumerationAdapter(std::unique_ptr<Enumeration> enm);
    [[nodiscard]] bool hasNext() const override;
    [[nodiscard]] std::optional<int> next() override;
};
#endif // ENUMERATION_ADAPTER_H
```

```
#include "EnumerationAdapter.h"

EnumerationAdapter::EnumerationAdapter(std::unique_ptr<Enumeration> enm)
    : enumeration(std::move(enm)) {}

bool EnumerationAdapter::hasNext() const {
    return enumeration->hasMoreElements();
}

std::optional<int> EnumerationAdapter::next() {
    return enumeration->nextElement();
}
```

This is what we promised: a small class with very little code. The adapter does no real work. All the work is done by the Adaptee; the adapter just routes the calls to the right method names.

The Client, main

Finally, the client. The whole point of the pattern is that the client knows nothing about Enumeration. It works with an `Iterator`, calls `hasNext` and `next`, and is content. The fact that an Enumeration is doing the work beneath the surface is invisible.

```
#include <iostream>
#include <vector>
#include <memory>
#include "EnumerationAdapter.h"
#include "ConcreteEnumeration.h"

int main() {
    // Create a vector of elements
    std::vector<int> elements = {1, 2, 3, 4, 5};
    // Create an enumeration with move semantics
    auto enumeration =
        std::make_unique<ConcreteEnumeration>(std::move(elements));
    // Adapt the enumeration to an iterator using the adapter pattern
    EnumerationAdapter iterator(std::move(enumeration));
    // Iterate through elements using the iterator interface
    while (iterator.hasNext()) {
        if (auto value = iterator.next()) {
            std::cout << *value << ' ';
        }
    }
    std::cout << '\n';

    return 0;
}
```

The flow of ownership is worth tracing carefully:

- Line 9: the vector `elements` is created on the stack.
- Line 12: it is moved into a `ConcreteEnumeration` managed by a `unique_ptr`. The vector is now owned by the enumeration; the local variable is left in a valid-but-moved-from state.
- Line 15: the `unique_ptr<Enumeration>` is moved into the adapter. The adapter now exclusively owns the enumeration.
- Lines 18-22: the adapter is used. When `iterator` goes out of scope at the end of `main`, its destructor runs, which destroys the unique_ptr, which destroys the enumeration, which destroys the vector. No leaks, no manual deletes.

> ⚠ **Warning:**
>
> **If you see double-deletes or use-after-frees in adapter code,** 95% of the time it is because the adapter and someone else both think they own the adaptee. Using `unique_ptr` makes this nearly impossible; the type system refuses to compile a second `unique_ptr` to the same object, and `std::move` is the only way to transfer ownership.

The output of this program is:

```
1 2 3 4 5
```

…which looks like nothing remarkable, and that is the point. The adapter is invisible. The client wrote iterator code; the iterator code is what executed; the legacy Enumeration was quietly fed through it.

A Second Example: Adapting a Legacy Rectangle

The Enumeration → Iterator example showed an adapter where the translation was nearly trivial, just renaming methods. The next example shows an adapter in which the translation involves reshaping parameters: the target interface and the adaptee interface disagree on how rectangles are described.

The setup

A legacy `LegacyRectangle` class draws rectangles given two diagonal corners: `draw(x1, y1, x2, y2)`. A new Shape interface defines a single `drawShape()` method and stores a rectangle as (x, y, width, height), the modern, more common convention.

These are mathematically equivalent; given one, you can compute the other in three additions and three identity copies, but they are syntactically incompatible. The client wants to call `drawShape()` on a `Shape`, not `draw(x1, y1, x2, y2)` on a `LegacyRectangle`.

The adapter

The adapter does three things: it implements the Target interface (`Shape`), it stores both the legacy rectangle and the modern (x, y, w, h) representation, and it performs the coordinate translation in `drawShape()`:

```
class LegacyRectangle {                         // LEGACY CLASS (Adaptee)
public:
    void draw(int x1, int y1, int x2, int y2) const {
        std::cout << "Drawing rectangle from (" << x1 << ", " << y1
                  << ") to (" << x2 << ", " << y2 << ")\n";
    }
};

class Shape {                                   // MODERN INTERFACE (Target)
public:
    virtual void drawShape() const = 0;
    virtual ~Shape() = default;
};

class RectangleAdapter : public Shape {         // ADAPTER
private:
    std::shared_ptr<LegacyRectangle> legacyRectangle;
    int x, y, width, height;
public:
    RectangleAdapter(std::shared_ptr<LegacyRectangle> rectangle,
                     int x, int y, int width, int height)
        : legacyRectangle(std::move(rectangle)),
          x(x), y(y), width(width), height(height) {}

    void drawShape() const override {
        // Translate (x, y, w, h) to (x1, y1, x2, y2)
        legacyRectangle->draw(x, y, x + width, y + height);
    }
};
```

The key line is in `drawShape()`: `legacyRectangle->draw(x, y, x + width, y + height)`. The adapter is doing what an adapter does, translating the call. The modern client gives (x, y, w, h); the legacy class wants (x1, y1, x2, y2); the adapter computes the second corner as `(x + width, y + height)` and forwards.

Using the adapter

The client code looks identical to any other Shape-based code:

```
int main() {
    auto legacyRectangle = std::make_shared<LegacyRectangle>();
    RectangleAdapter adaptedRectangle(legacyRectangle, 10, 20, 30, 40);
    std::cout << "Using modern Shape interface:\n";
    adaptedRectangle.drawShape();

    std::cout << "\nCreating another adapted rectangle:\n";
    RectangleAdapter anotherRectangle(legacyRectangle, 50, 60, 100, 80);
    anotherRectangle.drawShape();
    return 0;
}
```

Output:

```
Using modern Shape interface:
Drawing rectangle from (10, 20) to (40, 60)

Creating another adapted rectangle:
Drawing rectangle from (50, 60) to (150, 140)
```

Two things to notice. First, the adapter holds a `std::shared_ptr<LegacyRectangle>` rather than a `std::unique_ptr`, because both adapters share the same legacy rectangle object. This is the right call when an adapter genuinely is used by multiple adapters; if there were only one adapter per legacy object, `unique_ptr` would be the simpler and stricter choice.

Second, the modern Shape interface is the *only* thing the client mentions, except for constructing the adapter. In a fully-decoupled design, you would not call `RectangleAdapter` directly; you would call a factory that returns a `std::unique_ptr<Shape>`, and the client would never name `RectangleAdapter` or `LegacyRectangle` at all. That is exactly the layering Chapter 12's Factory Method gives you.

Caveats and Modern C++ Refinements

Implementing an adapter can be easy or hard

Implementing an adapter may require little work or a great deal of work, depending on the size and complexity of the target interface. The Enumeration → Iterator adapter we just wrote is on the easy end: two-method interfaces, name-for-name mapping. A more realistic adapter might face:

- **Mismatched return-value conventions**, Adaptee throws on error; Target returns `std::optional`; or vice versa. The adapter must translate, which often means a try/catch wrapper around every call.
- **Mismatched lifecycle rules**: Adaptee uses raw `new`/`delete`; Target uses smart pointers. The adapter must be careful about who owns what.
- **Mismatched threading models**: Adaptee is not thread-safe; Target promises thread safety. The adapter must add synchronization, which can be subtle.
- **Mismatched units or conventions**, Adaptee uses centimetres; Target uses metres. Adaptee uses 0-based indices; Target uses 1-based. These are easy to get wrong silently, and tests are essential.
- **Functionality the Adaptee cannot perform**, Target has a method the Adaptee has no equivalent for (the `remove()` problem from §5.1). The adapter must throw, no-op, or document the limitation.

None of these turns the Adapter Pattern into a different pattern; they change how much code the adapter contains.

Smart-pointer choices

Choice	When to use
std::unique_ptr<Adaptee>	Default. The adapter exclusively owns the adaptee. No other code touches it. Cheap and safe.
std::shared_ptr<Adaptee>	When the same adaptee genuinely needs to be wrapped by multiple adapters, or shared with something outside the adapter.
Adaptee* (raw pointer)	Only when the adaptee is owned by someone else, and the adapter is guaranteed not to outlive it. Avoid unless you have a specific reason.
Adaptee& (reference)	When the adapter is short-lived and an enclosing scope obviously bounds its lifetime.

std::optional for translated return values

`std::optional` is the right tool when the Adaptee's API uses exceptions or sentinel values, but you want the Target interface to use a more explicit "this might not produce a value" return type.

Example: a legacy `lookup(key)` function that returns `nullptr` on miss. Wrapping it in an adapter whose target returns `std::optional<Value>` makes the absence explicit at every call site and eliminates the implicit null-check obligation.

Move semantics for efficiency

Every adapter you write that takes ownership of an Adaptee should take the Adaptee by *rvalue reference or by value, then move it into the member*:

```
EnumerationAdapter(std::unique_ptr<Enumeration> enm)
    : enumeration(std::move(enm)) {}
```

The `std::move` in the member-initializer list is what makes the transfer cheap. Without it, you would be copying the smart-pointer, which, for `unique_ptr`, does not even compile. (Try removing the `std::move` from the lecture's example and watch the compiler explain.)

const correctness

"const methods since façades usually do not modify state." The same applies to adapters whose Target methods are themselves `const`: the corresponding adapter method should be `const` too, and it should forward to the adaptee through a `const` member-function pointer.

If the Adaptee's method is not `const` but the Target's is, you have a problem: a `const` adapter cannot call a non-`const` method on its member. The escape valve is meant to indicate that the adaptee member is `mutable`, but that is a code smell and should make you ask whether the Target's `const` contract is honest.

Adapters are not magic.

> ⚠ **Warning:**
>
> **An adapter cannot grant the adaptee abilities it lacks.** If the Target interface promises a thread-safe, persistent, undoable operation, and the Adaptee is none of those things, the adapter cannot deliver. Sticking an Adapter in front of an unsuitable adaptee is a common form of false promise; the call sites believe one thing, the implementation does another, and the gap is precisely where bugs accumulate. If the Adaptee genuinely cannot do what the Target promises, the right answer is to weaken the Target's promise or replace the Adaptee, not to pretend with an adapter.

Adapters in the Wild

Once you know the shape of the Adapter Pattern, you start seeing it everywhere.

STL container adaptors

The clearest example of the pattern in the C++ standard library is the family of *container adaptors*: `std::stack`, `std::queue`, and `std::priority_queue`. Each of these takes an underlying container, by default a `std::deque` for stack and queue, a `std::vector` for priority_queue, and adapts it to a different interface.

A `std::stack<int>` exposes `push`, `pop`, `top`, `empty`, and `size`. It does not expose iterators, `operator[]`, `front`, or `back`. Under the hood, `push` calls `push_back` on a deque; `pop` calls `pop_back`; `top` calls `back`. The Target is a stack interface; the Adaptee is the deque; the Adapter is `std::stack` itself.

This is also where you see the *class adapter* form taken to its logical conclusion via templates: the adaptee type is a template parameter, so a single `std::stack<T, Container>` template can adapt any container that supports `push_back`/`pop_back`/`back`. You can use `std::stack<int, std::vector<int>>` or `std::stack<int, std::list<int>>`, and the same adapter code works for both.

Network protocols

Applications need to send and receive data, but the wire formats vary: TCP, UDP, WebSocket, HTTP/2, gRPC. Higher-level libraries expose a single "message-send / message-receive" interface and adapt it to the configured protocol. The client code asks the connection adapter to send a message; the adapter packages it in the correct wire format and sends it.

Database drivers

Java's JDBC, Python's DB-API, .NET's ADO.NET, and Go's database/sql are all adapter frameworks. The Target is a generic database interface ("connect, execute, fetch rows, close"); the Adaptees are MySQL, Postgres, Oracle, SQLite, SQL Server. Each database vendor ships a driver that adapts its native protocol to the standard interface, and your application code is written against the standard interface so it can switch databases by changing a connection string.

Payment gateways

This is the lecture's first practice problem and the example we used in §1. PayPal, Stripe, Square, Adyen, and dozens of regional gateways each have their own SDK with their own method names. A typical e-commerce application defines a `PaymentProcessor` interface and ships one adapter per gateway. Switching processors is one configuration change; adding a new processor is one new adapter class.

Graphics-card drivers

"Graphics card drivers are adapting hardware to the standard API." OpenGL and Vulkan expose a standard API. The vendor drivers (NVIDIA, AMD, Intel) implement that API by issuing the right hardware commands for their specific GPU architectures. Your game does not know whether it is talking to an NVIDIA driver or an AMD driver; it knows it is talking to OpenGL.

Logging frameworks

Most logging libraries, such as SLF4J in Java, Serilog in .NET, and the Python `logging` module, have a similar shape: a logging interface (info/warn/error/debug) and a pluggable backend (file, syslog, network, console). Each backend is an adapter from the abstract logging interface to a concrete sink.

Operating-system file APIs

C++17 `std::filesystem` is itself a giant adapter layer: a single C++ interface for path manipulation and file I/O that the implementation translates into Win32 calls on Windows, POSIX calls on Linux/macOS, and so on. Without this layer, every program would have to write its own platform branches.

The footprint

Once internalized, the Adapter Pattern is one of the patterns you will spot most often in mature codebases. Any time a library says "we support multiple backends" or "we have drivers for X, Y, and Z", there is almost certainly an adapter framework at the bottom. The cost, one small class per backend, is small enough that it is essentially the default architecture for any system that needs to integrate with the outside world.

The Other Pattern in This Module

This chapter is about two patterns. We have spent the first half on Adapter; now we turn to its slightly different sibling, *Façade* (pronounced *fuh-SAHD*, French for the front of a building, the polished outer face).

Why bundle the two together? Because they share half their machinery. Both patterns involve altering an interface: you have one set of classes with one interface, but your client code wants a different one. In both patterns, the new class you add is a small wrapper that sits between the client and the existing code and forwards calls to the existing code.

The difference is *why* you alter the interface:

> 💡 **Tip:**
>
> **Adapter vs. Façade, the one-sentence distinction.** "You have seen how the Adapter Pattern converts the interface of a class into one that a client is expecting. ... We are going to look at a pattern now that alters an interface, but for a different reason: to *simplify* the interface. It is aptly named the Façade Pattern because this pattern hides all the complexity of one or more classes behind a clean, well-lit façade."

Put another way:

- **Adapter** changes an interface because the *name* is wrong (Target.next vs Adaptee.nextElement). The number of classes does not change; one wrapper sits in front of one adaptee.
- **Façade** changes an interface because the *size* is wrong (twelve confusing classes vs. one watchMovie call). One wrapper sits in front of a *subsystem* of many classes.

This will be the recurring image: an adapter is a one-to-one wrapper; a façade is a one-to-many wrapper. Hold that distinction in mind as you read the rest of the chapter; we will sharpen it further when we compare both patterns to the Decorator from Chapter 10.

Definition: The Façade Pattern

> **Tip:**
>
> **Façade Pattern (Intent).** Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

Three nouns again, but they are different from the Adapter's:

- **Subsystem,** a collection of classes that, taken together, solve some larger problem, but which are individually too small and too low-level to use comfortably.
- **Façade,** a single class that offers a small, friendly interface and orchestrates the subsystem to fulfill each request.
- **Client,** the caller, which now talks only to the Façade and never has to learn the names or interactions of the subsystem classes.

The Façade is the front of the building. The plumbing, wiring, structural beams, HVAC, and fire suppression are all behind it. You walk in the front door, you state your need ("*watch a movie*"), and inside the building, someone orchestrates the popcorn maker, the lights, the projector, and the sound system to make it happen.

The class diagram

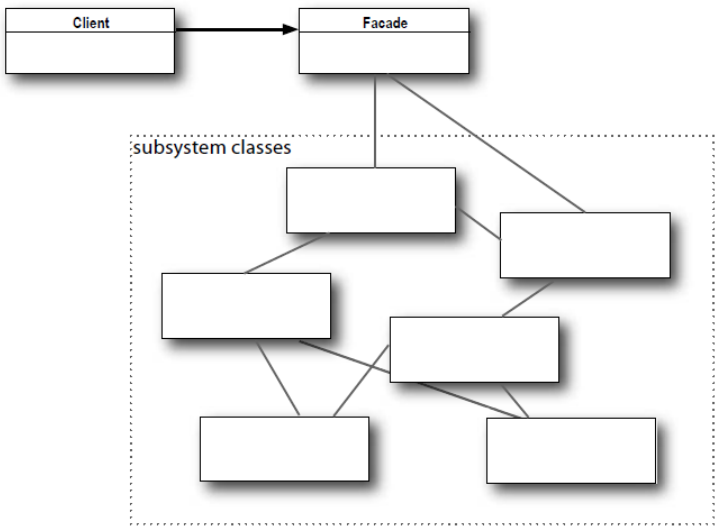

Figure 11.1: The Façade Pattern UML. A single Client talks to a single Façade. The Façade is associated with many subsystem classes (drawn inside the dotted boundary). The subsystem classes are themselves interconnected, they form a web of collaborations that the client would otherwise have to navigate directly.

What the Façade does (and does not) do

Three points are worth nailing down before we look at code:

- **The Façade does not hide the subsystem classes.** They are still accessible to anyone who wants direct access. The Façade is an *optional* convenience layer, not a wall. A power user who needs fine-grained control can still go around the Façade and talk to `Amplifier` directly. This is intentional. Façades simplify the common case without forcing the rare case to suffer.
- **The Façade does not add new functionality.** It does not invent operations that the subsystem cannot perform. It composes existing operations into convenient bundles. If you find your Façade growing real business logic, the work has migrated from the subsystem to the wrong place, push it back down where it belongs.
- **You can have more than one Façade per subsystem.** From lecture notes: You can implement more than one façade for a subsystem." A *BasicTheaterFacade* might offer `watchMovie` and `endMovie`; an *AdvancedTheaterFacade* might offer `watchMovieInRegion`, `schedulePremiere`, and `manageGuestList`. Each serves a different audience.

> ⚠ **Warning:**
>
> **A Façade is not a god class.** There is a temptation, especially among less experienced developers, to write a single Façade class with seventy methods, one for every conceivable use of the subsystem. That defeats the purpose. A Façade is supposed to *simplify;* if it has more methods than the subsystem classes combined, it is not a Façade, it is a god class with extra steps. If you cannot describe your Façade's purpose in a sentence, split it.

The Home Theatre Example

Let us run through an example of the Façade Pattern: a home theatre setup with a popcorn popper, theatre lights, a movie screen, a projector, an amplifier with multiple inputs, a DVD player, a CD player, and a tuner. Eight subsystems, each a separate class, each with its own set of methods. Together, they let you watch a movie. Individually, they require significant coordination.

The subsystem

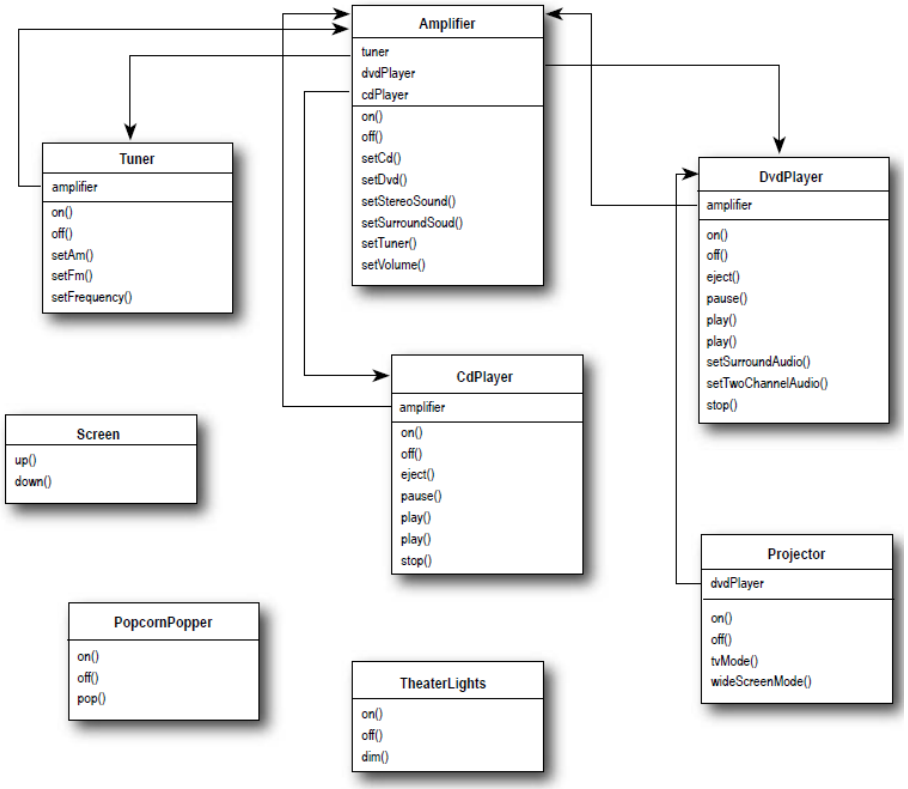

Figure 12.1: The home-theatre subsystem class diagram.

That is a lot of classes, a lot of interactions, and a large set of interfaces to learn and use. Indeed.

Watching a movie, the hard way

Without a Façade, every client who wants to watch a movie has to perform the whole twelve-step ritual itself:

Step	Action
1.	Turn on the popcorn popper.
2.	Start the popper popping.
3.	Dim the lights to 10%
4.	Put the screen down

Step	Action
5.	Turn the projector on
6.	Put the projector in wide-screen mode
7.	Turn the amplifier on
8.	Set the amplifier input to DVD
9.	Set the amplifier to surround sound
10.	Set the amplifier volume to 5
11.	Turn the DVD player on
12.	Start the DVD player playing

In code:

```
popper->on();
popper->pop();
lights->dim(10);
screen->down();
projector->on();
projector->wideScreenMode();
amp->on();
amp->setDvd(dvd);
amp->setSurroundSound();
amp->setVolume(5);
dvd->on();
dvd->play(movie);
```

Twelve lines of code, six different classes touched, and the client has to know exactly which methods to call, in which order, on which objects.

> **Tip:**
>
> When the movie is over, how do you turn everything off? Wouldn't you have to do all of this over again, in reverse? Would it not be as complex to listen to a CD or the radio? If you decide to upgrade your system, you will likely need to follow a slightly different procedure.

Each new mode of use, watching a movie, ending a movie, listening to a CD, listening to the radio, requires its own twelve-step recipe. Each recipe has to be learned, written out at every call site, and

maintained as the subsystem evolves. If `Amplifier` adds a new setHDMI method and deprecates the old setDVD, every `12-step recipe in your codebase has to be found` and updated. This is the kind of pain that drives engineers to design patterns.

Watching a movie, with a Façade

```
homeTheater.watchMovie("Raiders of the Lost Ark");
```

One line. One method call. The client never mentions the popper, projector, amplifier, screen, or DVD player. It says "watch a movie," and the Façade (HomeTheaterFacade) handles the orchestration internally.

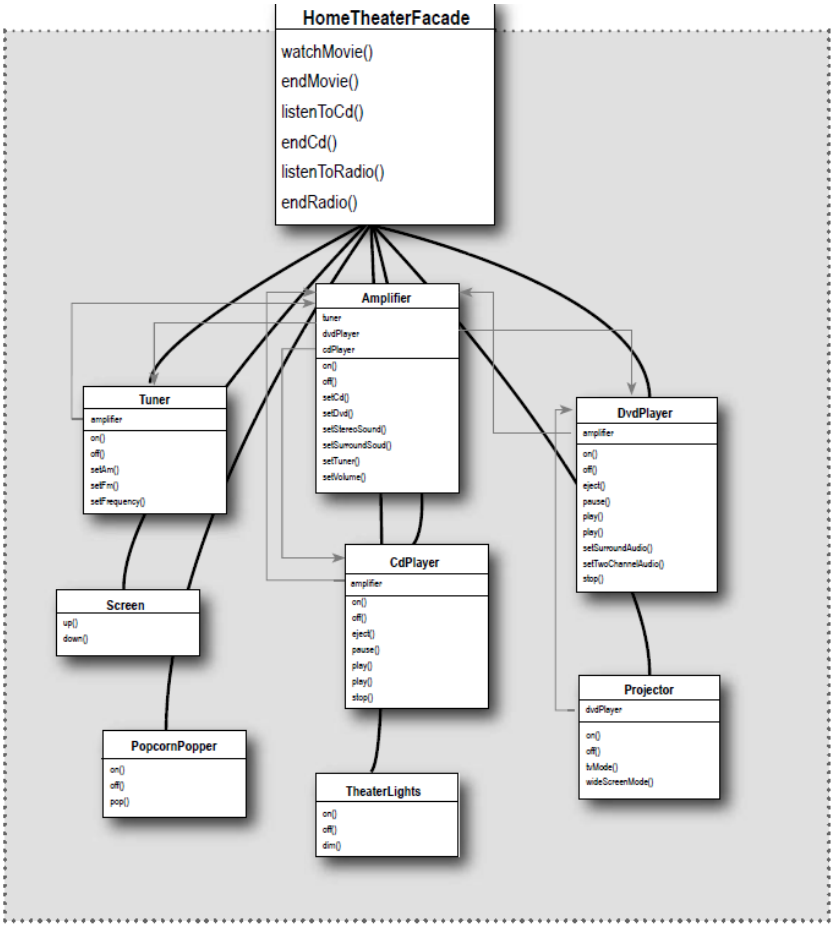

Figure 12.2: The home theatre with a Façade.

Notice that the Façade does not hide the subsystem; those eight classes are still visible to anyone who needs them. The Façade adds a new layer *on top*, not a layer *in front of*. An advanced user who needs to set the amplifier volume to a non-standard level for a particular movie can still go straight to `amp->setVolume(7)`; the Façade makes the common case trivially easy.

Implementing the Home Theatre Façade in C++

Now we build the Façade for real.

The subsystem classes

The eight subsystem classes are declared up front. Each is small, a handful of methods, and each is independent of the Façade. (You could, and should, write and test these classes long before the Façade exists.)

```
#include <iostream>
#include <string>
#include <string_view>
#include <memory>

// Forward declaration
class DvdPlayer;

class Amplifier { ... };

class Tuner { ... };

class DvdPlayer { ... };

class CdPlayer { ... };

class Projector { ... };

class TheaterLights { ... };

class Screen { ... };

class PopcornPopper { ... };
```

None of these classes know the Façade exists. That is important: dependencies flow only one way. The Façade depends on the subsystem; the subsystem does not depend on the Façade. If the Façade were deleted tomorrow, the subsystem would still compile and work. The reverse is, of course, not true.

The Façade, header

```
class HomeTheaterFacade {
private:
    std::shared_ptr<Amplifier> amp;
    std::shared_ptr<Tuner> tuner;
    std::shared_ptr<DvdPlayer> dvd;
    std::shared_ptr<CdPlayer> cd;
    std::shared_ptr<Projector> projector;
    std::shared_ptr<TheaterLights> lights;
    std::shared_ptr<Screen> screen;
    std::shared_ptr<PopcornPopper> popper;

public:
    HomeTheaterFacade(std::shared_ptr<Amplifier> amp,
                      std::shared_ptr<Tuner> tuner,
                      std::shared_ptr<DvdPlayer> dvd,
                      std::shared_ptr<CdPlayer> cd,
                      std::shared_ptr<Projector> projector,
                      std::shared_ptr<Screen> screen,
                      std::shared_ptr<TheaterLights> lights,
                      std::shared_ptr<PopcornPopper> popper)
        : amp(std::move(amp)),
          tuner(std::move(tuner)),
          dvd(std::move(dvd)),
          cd(std::move(cd)),
          projector(std::move(projector)),
          lights(std::move(lights)),
          screen(std::move(screen)),
          popper(std::move(popper)) {}
```

Two design choices deserve comment:

- **Why** `std::shared_ptr` and not `std::unique_ptr`? Because a subsystem component might be referenced by more than one party. Maybe an `AdvancedTheaterFacade` also wants the `Amplifier`; maybe direct-access code needs the `Projector`. With `shared_ptr`, multiple owners share the lifetime cleanly. With `unique_ptr`, only the Façade can hold the object; fine if the Façade is the only thing that uses it, but restrictive otherwise.
- **Why move the parameters?** Each constructor parameter is taken by value as a `shared_ptr`, then `std::moved` into the member. This means the caller can pass either an rvalue (no copy) or an lvalue (one ref-count increment, then a cheap move into the member). It is the most flexible signature for a constructor that takes ownership of pointer-managed resources.

The Façade, watchMovie and endMovie

```
    void watchMovie(std::string_view movie) const {
        std::cout << "Get ready to watch a movie...\n";
        popper->on();
        popper->pop();
        lights->dim(10);
        screen->down();
        projector->on();
        projector->wideScreenMode();
        amp->on();
        amp->setDvd(dvd.get());
        amp->setSurroundSound();
        amp->setVolume(5);
        dvd->on();
        dvd->play(movie);
    }

    void endMovie() const {
        std::cout << "Shutting movie theatre down...\n";
        popper->off();
        lights->on();
        screen->up();
        projector->off();
        amp->off();
        dvd->stop();
        dvd->eject();
        dvd->off();
    }
};
```

Read `watchMovie` carefully. It is exactly the twelve-step ritual from before, popper, lights, screen, projector, amp, dvd, wrapped in a single function. Nothing has been added. Nothing has been moved. The complexity is the same; what has changed is *who* is paying for it. Previously, every client had to copy and paste the twelve steps. Now, twelve steps live in one place, every client calls `watchMovie`, and the maintenance bill drops accordingly.

`endMovie` does the reverse: turn off the popper, raise the screen, turn on the lights, turn off the projector and amp, stop and eject the DVD, turn the DVD off. This is the operation the instructor-notes paragraph in §12.2 was asking about, "how do you turn everything off?" The Façade has the answer in one line at the call site.

Using the Façade, main

```
int main() {
    // Create all subsystem components using smart pointers
    auto amp = std::make_shared<Amplifier>();
    auto tuner = std::make_shared<Tuner>();
    auto dvd = std::make_shared<DvdPlayer>();
    auto cd = std::make_shared<CdPlayer>();
    auto projector = std::make_shared<Projector>();
    auto screen = std::make_shared<Screen>();
    auto lights = std::make_shared<TheaterLights>();
    auto popper = std::make_shared<PopcornPopper>();

    // Create the façade
    HomeTheaterFacade homeTheater(amp, tuner, dvd, cd, projector,
                                  screen, lights, popper);

    // Use the simplified interface
    homeTheater.watchMovie("Raiders of the Lost Ark");
    std::cout << '\n';
    homeTheater.endMovie();

    // Smart pointers automatically clean up - no delete needed!
    return 0;
}
```

The client code is now seven significant lines:

```
auto amp       = std::make_shared<Amplifier>();
auto tuner     = std::make_shared<Tuner>();
auto dvd       = std::make_shared<DvdPlayer>();
// ... five more subsystem components ...

HomeTheaterFacade homeTheater(amp, tuner, dvd, cd,
                              projector, screen, lights, popper);

homeTheater.watchMovie("Raiders of the Lost Ark");
homeTheater.endMovie();
```

Construction is still a bit verbose, eight components have to be built and wired in. Could you simplify that? Yes, in three ways:

- **Move construction inside the Façade.** If the subsystem components are not shared with anyone, the Façade itself can construct them in its default constructor. Then `main` writes `HomeTheaterFacade homeTheater;`. The trade-off is that test code cannot inject mock subsystem objects.
- **Use a Factory.** A `HomeTheaterFactory` knows how to assemble all eight components and hand back a fully-wired Façade. The Façade itself stays "dumb" and relies on its dependencies; the Factory determines which concrete subsystem classes to use. (See Chapter 12.)
- **Use Dependency Injection.** A DI framework, or a manual composition root, builds all components once at program start and hands them to the Façade. The Façade then doesn't care where they came from.

None of these is required. The straightforward construction is fine for small systems; the alternatives kick in when the system grows.

> ❓ **Question:**
>
> Suppose you want to add a `listenToCd` method to the Façade. What changes? What does not?
>
> **Answer.** You add a new method `void listenToCd(std::string_view album) const` to `HomeTheaterFacade` that, internally, turns on the amplifier and CD player, sets the amp input to CD, and starts playback. No other code changes. The subsystem classes (`Amplifier`, `CdPlayer`) were already capable of these operations; the Façade is just exposing them under a new packaging. Adding capabilities to a Façade is local and cheap, which is exactly why it is a useful pattern.

Principle of Least Knowledge

The design principle that the pattern embodies is sometimes called the *Law of Demeter*, more commonly the *Principle of Least Knowledge*:

> 💡 **Tip:**
>
> **Principle of Least Knowledge.** Talk only to your immediate friends. Reduce the interactions between objects to just a few close "friends". When designing a system, for any object, be careful about the number of classes it interacts with and how it interacts with them.

The motivating intuition is social: if you are designing a person rather than an object, you want the person to have a small, stable circle of relationships. A person who has to negotiate with everyone in the

room every time they want to do something simple has too many points of failure. A person who can talk to a few trusted friends and let those friends handle their own circles is more robust.

The rule, in code

In code, the principle suggests: in any method of class X, you should only invoke methods on:

- **The object itself** (this, your own methods).
- **Objects passed to the method as parameters.**
- **Objects you create yourself** inside the method.
- **Objects held as fields of the class.**

That is the friend list. Anyone else is a stranger, and calling their methods is "talking to strangers". The classic warning sign is the *train wreck*, a chain of dotted method calls like:

```
station.getTrain().getEngine().getFuel().getLevel();
```

Here, `station` is a friend (supposedly a parameter), but `station.getTrain()` returns a `Train`, a stranger. Asking the stranger's engine, the engine's fuel, and the fuel's level is a chain of conversations across four objects. If any of those four classes change their interfaces, every caller of this expression breaks.

The fix is to let the friend do the asking. Add a `getFuelLevel()` method to `Station` that walks the chain internally. The client says `station.getFuelLevel()`, talks only to its friend, and is shielded from changes to `Train`, `Engine`, or `Fuel`.

How the Façade Pattern applies

The home-theatre-without-a-Façade code is exactly the kind of train wreck the principle warns against. The client touches six different classes (`Popper`, `Lights`, `Screen`, `Projector`, `Amp`, `Dvd`), each with several methods. Any change to any of these classes ripples through every client.

With a Façade, the client talks to one friend, `HomeTheaterFacade`. The Façade itself communicates with the six subsystem classes, but that conversation is confined to a single place. If `the amplifier` changes, only `HomeTheaterFacade` needs to be updated; all clients are shielded from it.

So the Façade is more than a convenience. It is also a structural application of the Principle of Least Knowledge: it gives clients a single friend, and isolates them from the complications behind that friend's door.

When to push back

> ⚠ **Warning:**
>
> **Like all design principles, this one can be overdone.** If you apply it religiously, you end up wrapping every class in a façade, every façade in a façade, and so on, until your codebase is a stack of pass-through layers. Each layer adds maintenance overhead, runtime indirection, and an extra place for bugs. The principle is a guideline for *how much* your classes know about each other; it is not a mandate to wrap every relationship.

A useful rule of thumb: apply the principle when the train wreck is genuine, many classes are touched, and many places in the codebase are doing the same multi-step ritual. Resist applying it when a one-step chain happens once and is unlikely to recur. Premature abstraction is more expensive than late refactoring.

Adapter vs. Façade vs. Decorator

All three patterns covered in this chapter and Chapter 11 wrap one or more existing objects in a new object. The diagrams look similar: a Wrapper class holds a reference to a Wrapped object, and the client communicates only with the Wrapper. The intent is what distinguishes them:

> 💡 **Tip:**
>
> The key distinction: Adapter solves 'I have X, but I need Y' (interface conversion), Façade solves 'This is too complicated' (interface simplification), and Decorator solves 'I need X to do more' (behaviour extension).

The three-way comparison table

Pattern	What it wraps	Why	Number of wrapped objects
Adapter	An adaptee with the wrong interface	To translate one interface to another	One
Façade	A subsystem of related classes	To simplify a complex interface	Many
Decorator	An object of the same interface	To add new behaviour to that object at runtime	One, and decorators can be stacked

Shape vs. intent, the canonical confusion

Students seeing all three patterns for the first time often get confused because the class diagrams look so similar. Each has a wrapper class with the same superficial structure. The distinguishing signal is at the *interface* of the wrapper:

- **Adapter: same wrapper interface as Target, different from the wrapped object's interface.** The whole point is that wrapper and wrapped have different interfaces; otherwise, no translation is needed.
- **Façade: wrapper has its own simplified interface; wrapped objects are many, each with its own interface.** The wrapper is not pretending to be any of the wrapped classes; it is a new entity sitting on top of them.
- **Decorator: wrapper has the same interface as wrapped.** This is what allows decorators to stack: the output of one decorator is an IS-A target that can be passed to the next decorator.

Pattern	Wrapper IS-A	Wrapper HAS-A
Adapter	Target	Adaptee (different from Target)
Façade	(no specific Target)	Multiple subsystem classes
Decorator	Component	Component (same as itself)

The last column for Decorator is the giveaway. Decorator is the only one of the three in which the wrapper and the wrapped have the same type, which allows chaining. An adapter has different types (translation); Façade has many types (simplification). Decorator has one type used twice (augmentation).

What about Proxy?

You may meet a fourth wrapper-shaped pattern later in your career: *Proxy*. Proxy looks identical to Decorator in the diagram ("wrapper that has the same interface as wrapped"), but its intent is different again: control access, not add behaviour. A logging proxy intercepts all write calls and logs them. A caching proxy intercepts calls and returns saved results when possible. A protection proxy intercepts calls and refuses unauthorized ones. The Decorator-vs-Adapter-vs-Façade table above could be extended to four columns; for the same wrapper shape, intent dictates which pattern it is.

A worked discrimination

> ❓ **Question:**
>
> Here are three short scenarios. Without consulting the table, decide whether each is an Adapter, a Façade, or a Decorator.
>
> **(a)** A new logging library exposes `Logger::log(Severity, std::string)`, but your codebase has years of calls to `oldLog(const char*, int)`. You write a class that exposes the old signature and forwards to the new library.
>
> **(b)** A microservice has eight internal classes (database, cache, queue, auth, telemetry, config, scheduler, mailer). You write one class with five methods, `startUp`, `shutDown`, `processRequest`, `getStatus`, `checkHealth`, that coordinates all eight.
>
> **(c)** You have an `HttpRequest` interface with a `send()` method. You want to add retry logic that calls `send()` up to three times if it fails. You write a class that IS-A `HttpRequest`, HAS-A `HttpRequest`, and on each `send()` call delegates to the wrapped request and retries on failure.
>
> **Answers.** (a) Adapter, different interface (old vs. new), no new behaviour. (b) Façade, many subsystem classes wrapped into one simpler interface. (c) Decorator, same interface, augmented behaviour (retry).

Façades and Adapters in the Wild

Compiler façades

A modern compiler, gcc, clang, MSVC, is a pipeline of distinct phases: lexer, parser, semantic analyzer, optimizer, code generator, linker. Each phase is a substantial subsystem in its own right. The compiler driver, the thing you actually invoke when you type `g++ file.cpp`, is a Façade: it hides the phases behind a *compile this file* interface. Power users can invoke individual phases (`g++ -E` runs only the preprocessor; `g++ -S` stops at assembly), but the common case is a single command.

Game-engine initialization

A typical game engine has dozens of subsystems: rendering, audio, physics, networking, input, scripting, asset loading, scene graph, particle systems, and collision detection: "`GameEngine.initialize()` hides hundreds of subsystem initializations." One call sets up everything. Unity, Unreal, Godot, and every commercial engine you have heard of work this way at the top level.

Email client façades

Email is not one protocol; it is several. SMTP for sending, POP3 and IMAP for receiving, S/MIME and PGP for encryption, and OAuth for authentication. An email client like Thunderbird or Outlook presents a single "`send`" and "`receive`" interface to the user, and behind that interface, a Façade orchestrates the correct protocol depending on the account configuration.

Payment façades

From slide 40: "Single `payment.process()` hiding tokenization, fraud detection, transaction processing." A real payment call goes through several phases: validate the card, tokenize it for storage, run fraud-detection heuristics, contact the gateway, handle the response, and log the result. Stripe, Square, and the like expose this as a single API call; you do not see the seven internal phases unless something goes wrong.

React's component model

Outside C++, React's `Component` class is a Façade over the underlying DOM operations. A React developer calls `setState` and the component re-renders; React internally diffs the virtual DOM, computes the minimum set of real DOM operations, batches them, and applies them. The complexity of efficient DOM manipulation is hidden behind a simple state-update interface.

std::filesystem

Mentioned in §9.7 as an adapter; it is equally an example of a Façade. From a client's perspective, `std::filesystem::copy` is one call. Internally, it has to walk directories, handle symbolic links,

propagate permissions, deal with errors at every step, and translate between Windows and POSIX conventions. The standard library hides all of that behind a single function.

The footprint

As with Adapter, once you know the shape of Façade, you find it everywhere. Any time a library says "easy to get started, powerful when you need it," it is almost certainly a façade for a more elaborate API. The pattern is the structural answer to the user experience question: "How do we serve both the beginner and the expert from one codebase?" The Façade is the beginner's door; the subsystem is the expert's toolshed.

Summary

Two patterns in one chapter, Adapter and Façade, both about altering an interface, but for different reasons.

The Adapter Pattern

Intent: convert the interface of a class into another interface that the clients expect.

Built from four roles:

- **Target**, the interface the client expects.
- **Client**, the code that uses the Target, and that we do not want to change.
- **Adaptee**, the existing class with the wrong interface.
- **Adapter**, a new class that IS-A Target and HAS-A Adaptee (object form) or IS-A Target and IS-A Adaptee (class form).

Two forms exist (Object Adapter uses composition, Class Adapter uses multiple inheritance), and Object Adapter is the strongly preferred form in modern C++17.

The Adapter's value:

- Let you reuse an existing class whose interface is wrong without modifying it.
- Let you swap which concrete class is being adapted at runtime by writing different adapters.
- Localize the translation cost to a single, small class.
- Plays well with smart pointers, `std::optional`, and move semantics for clean modern-C++ implementations.

The Façade Pattern

Intent: provide a unified interface to a set of interfaces in a subsystem. The Façade defines a higher-level interface that makes the subsystem easier to use.

Built from three roles:

- **Subsystem**, a collection of classes that, together, solve some larger problem.
- **Façade**, a single class with a simpler interface, holding references to the subsystem objects and orchestrating them.
- **Client**, talks only to the Façade and is shielded from the subsystem.

The Façade's value:

- Reduce the surface area the client has to learn, twelve method calls become one.
- Localize subsystem-orchestration logic in one place, so changes to the subsystem affect only the Façade.
- Embody the Principle of Least Knowledge: clients talk only to their immediate friend (the Façade) and let it deal with strangers.

- Does not hide the subsystem; power users can still talk directly to the subsystem classes when needed.

The shared moral

Both patterns are structural answers to the question "*the interface I have is not the interface I want.*" Adapter answers, "The name is wrong."; Façade answers, "T*he size is wrong*." In both cases, the fix is the same: a small new class that sits between the client and the existing code. But the reason for the new class determines which pattern you are using.

Memorize the three-way distinction from §15.1:

- Adapter: "I have X, I need Y."
- Façade: "This is too complicated."
- Decorator: "I need X to do more."

…and you have the patterns straight, even if the diagrams look alike.

Practice Problems

Problem 1.

You are building a Payment Gateway Aggregator that needs to support multiple payment providers. Each provider has its own interface:

- `PayPal API: void processPayPalPayment(double amount)`
- `Stripe API: void chargeStripe(double amount, std::string currency)`
- `Legacy Bank API: void makeTransaction(int cents)`

Your system requires a common interface:

```
class PaymentProcessor {
public:
    virtual void pay(double amount) = 0;
    virtual ~PaymentProcessor() = default;
};
```

Task. Implement adapters for each payment provider so they conform to `PaymentProcessor`. For each adapter:

(a) Decide on the smart-pointer strategy. Should the adapter own the underlying provider, share it, or borrow it? Justify in one sentence per choice.

(b) Sketch the constructor signature for each adapter.

(c) Implement `pay(amount)` for each. Watch out: the Stripe API needs a currency string. Where does it come from? The Legacy Bank API takes cents, not dollars. What happens if the `amount` is `19.995`?

(d) Write a `main` that uses three adapters polymorphically through `std::vector<std::unique_ptr<PaymentProcessor>>`, and processes one payment of $12.50 through each.

Problem 2.

You are designing a Video Streaming Service that integrates three subsystems:

- **AuthenticationSystem,** handles user login and subscription checks. Methods: `login(user, pass)`, `isSubscribed(user)`, `logout(user)`.
- **ContentDelivery,** streams video content. Methods: `startStream(title)`, `stopStream()`, `buffer(seconds)`.
- **RecommendationEngine,** suggests videos based on user history. Methods: `recordView(user, title)`, `getSuggestions(user)`.

Currently, clients have to interact with all three subsystems individually. You need to simplify.

Task. Implement a Façade called `StreamingFacade` that provides three high-level methods:

- `watchMovie(user, title)`, handles authentication, checks subscription, starts the stream, and records the view for recommendations.
- `getRecommendations(user)`, fetches personalized suggestions.
- `logout(user)`, stops any active stream and terminates the session.

(a) Sketch the UML class diagram. Use shared_ptr for subsystem ownership.

(b) Implement the three Façade methods. Handle the case where authentication or subscription check fails. What does `watchMovie` do? Throw? Return a status code? Use `std::optional`?

(c) Reflect: why is this a Façade rather than an Adapter? Which sentence from §15 best captures the distinction in this case?

Problem 3.

For each of the following six scenarios, identify whether the design described is best understood as an Adapter, a Façade, a Decorator, or none of the above. Justify in one sentence.

(a) A wrapper class implements the same Stream interface as the wrapped object and adds gzip compression on every `write`.

(b) A wrapper class exposes a `save()` method that, internally, validates the data, writes it to a temp file, fsyncs, renames over the original, and updates an audit log.

(c) A wrapper class exposes the `std::iostream` interface and forwards reads and writes to a network socket whose native API is `send(buf, len)` / `recv(buf, len)`.

(d) A wrapper class exposes the same DB connection interface as the underlying driver, but caches results of recent queries so repeated calls return without hitting the DB.

(e) A class with three methods, `startGame`, `pauseGame`, and `quitGame`, that coordinates the Renderer, AudioEngine, InputHandler, and SaveSystem.

(f) A class that takes a `std::vector<double>` of measurements in Fahrenheit and offers a `getValueInCelsius(i)` method.

Problem 4.

In C++, you have the choice. Recall the Enumeration→Iterator adapter from §6.

(a) Rewrite `EnumerationAdapter` as a *class adapter* using multiple inheritance: have it inherit publicly from `Iterator` and privately from `ConcreteEnumeration`. (Hint: protected/private inheritance is the right choice here; you do not want the Enumeration interface leaking through the adapter.)

(b) Compare the two implementations. How many lines does each take? Which one supports adapting different concrete Enumeration subclasses? Which one makes it easy to override the adaptee's behaviour?

(c) Suppose a new `LinkedListEnumeration` subclass appears. How many code changes are in each form?

Problem 5.

A legacy module has the following interface for a key-value store:

```
class LegacyStore {
public:
    void   put(const char* key, int value);
    int    get(const char* key);    // returns -1 if missing
    void   remove(const char* key);
    int    count();                  // number of entries
};
```

Task. Write an adapter `MapStoreAdapter` that exposes the following interface and wraps a `std::unordered_map<std::string, int>`:

```
class KVStore {
public:
    virtual void put(std::string key, int value)                    = 0;
    virtual std::optional<int> get(const std::string& key) const    = 0;
    virtual bool remove(const std::string& key)                     = 0;
    virtual std::size_t size() const                                = 0;
    virtual ~KVStore() = default;
};
```

(a) Notice the four mismatches between Legacy and Modern: `const char*` vs `std::string`; `int` return with `-1` sentinel vs `std::optional<int>`; `void remove` vs `bool remove` (success indicator); `int count()` vs `size_t size() const`. For each, write one sentence explaining how the adapter resolves it.

(b) Now write the adapter. The adapter must *not* depend on `LegacyStore`; instead, it wraps a `std::unordered_map` directly. (This is a slightly unusual adapter, you are adapting *the standard library type* to the codebase's preferred interface. This is the same shape as how STL container adaptors work internally.)

(c) What does `get("absent")` return for your adapter? What does the original legacy `get` return? Why is your adapter's behaviour safer?

Problem 6.

A build system has five separate phases, each implemented as a class:

- `SourceFetcher`, pulls source from git: `checkout(ref)`, `lastCommit()`.
- `Compiler`, compiles source: `compile(srcDir, outDir)`, `warnings()`.
- `Tester`, runs tests: `runTests(outDir)`, `passed()`, `failed()`.
- `Packager`, packages a build: `makePackage(outDir, version)`.
- `Publisher`, uploads to artifact store: `upload(pkg, target)`.

(a) Design a `BuildPipelineFacade` with two methods: `nightlyBuild(branch)` and `releaseBuild(tag, target)`. Sketch the UML.

(b) What does `nightlyBuild` do that `releaseBuild` does not, and vice versa? List each step.

(c) Suppose a CI engineer wants to run only the tests, not the full pipeline. Does your design force them to use the Façade, or can they go straight to `Tester`? Should it force them? Justify.

(d) Now imagine a second Façade, `DeveloperLoopFacade`, aimed at a developer running iterative tests locally. What methods would it expose? How do the two Façades over the same subsystem differ in purpose?

Problem 7.

Consider the following situation. Your codebase uses a `Timer` interface that emits a tick every 100ms. A new requirement says some clients need a `DebouncedTimer` with the same interface, but ticks fire only if at least 500ms have elapsed since the last "reset" call.

(a) Sketch this as an Adapter. Where does it break down?

(b) Sketch it as a Decorator. Where does it succeed?

(c) State, in one sentence, the difference between "adapting an interface" and "adding behaviour to an interface". Use this problem as an example.

Problem 8.

You are reviewing the following method from a coworker:

```
void processOrder(Order order) {
    auto items = order.getCustomer()
                      .getShoppingCart()
                      .getItems();
    for (const auto& item : items) {
        if (item.getProduct().getCategory().isHazardous()) {
            order.getShipper().setSpecialHandling(true);
        }
    }
}
```

(a) Count the violations of the Principle of Least Knowledge. Which classes is `processOrder` talking to that it should not be?

(b) Rewrite the method so that it only talks to its immediate friends. You may add new methods to `Order`, `Customer`, `ShoppingCart`, or any other class, but each new method should follow the principle itself.

(c) After your refactor, how many classes does `processOrder` name? How does that affect what breaks if (i) `ShoppingCart` changes its API, or (ii) `Product` introduces a new field?

Problem 9.

Some real situations are not cleanly either pattern. Consider:

A new class, UnifiedFileSystem, exposes openFile(path), listDirectory(path), and `deleteFile(path)`. Internally, it handles three different storage backends: local disk, AWS S3, and a network mount, each with completely different APIs. A configuration setting determines which backend is used.

(a) Is this an Adapter or a Façade? Argue both sides in two sentences each.

(b) If the answer changes when the configuration is fixed at compile time versus chosen at runtime, explain why.

(c) Sketch a design that is *both* a Façade over a family of Adapters. Where is each pattern, and what does it contribute?

(d) If you also had to support a new feature that wraps each operation in retry-and-log logic, how would you fit a Decorator into the design? Where does it sit relative to the Adapter and the Façade?

Problem 10.

Take the home-theatre Façade and extend it.

(a) Add a `listenToCd` / `endCd` pair and a `listenToRadio` / `endRadio` pair. Show the new method bodies, what calls does each make on the subsystem?

(b) Implement a fail-safe: if `watchMovie` is called twice without an intervening `endMovie`, the Façade should call `endMovie` internally before starting the second movie. Where does this logic go? Does it belong in the Façade or in one of the subsystem classes? Justify.

(c) Add unit tests for the Façade. To test it in isolation, you need to inject mock subsystem objects. Refactor the subsystem classes to inherit from abstract interfaces (Amplifier becomes IAmplifier with a concrete RealAmplifier), so the Façade can be tested with MockAmplifier, MockDvd, etc. How big is the refactor? Is it worth it for a small system? For a large one?

(d) Reflect: how does Question (c) interact with Chapter 13's Factory Method Pattern? If you had to choose, would you build *HomeTheaterFactory* before the abstract interfaces, after them, or as part of the same refactor? Justify in three sentences.