🎯 Step 1: Adapting the Legacy Synchronous Processor to `IPaymentGateway`

1.1. Statement of the Problem

> **Why do we need an adapter?**
> Our **new system** has standardized on `IPaymentGateway` (an interface), which demands a method `Pay(PaymentDetails payment)`.
> Our **legacy code** is a class `LegacyPaymentProcessor` with a method `ProcessPayment(string cardNumber, double amount)`.
> We **cannot** modify `LegacyPaymentProcessor` (it's "rock-solid" and off-limits).
> We **must** bridge between the old and new without touching the old.

I ask myself: *What exactly is an "adapter pattern"?*

- It's a **design pattern** that allows incompatible interfaces to work together.
- Formally, it **wraps** (or "adapts") one interface in another.
- In C#: we'll create a **new class** that **implements** `IPaymentGateway`, and **internally calls** the legacy method.

1.1.1. Fundamental Terms

- **Interface (** `IPaymentGateway` **)**:
  - A class-like construct that declares **methods** (and possibly properties) but **no implementations**.
  - Here: `void Pay(PaymentDetails payment);`
  - **Why interfaces?** They allow **polymorphism**—various implementations can be swapped without code changes.
- **Class (** `LegacyPaymentProcessor` **)**:
  - A blueprint for objects, containing **fields**, **methods**, and possibly **state**.
  - Here: it has a method `ProcessPayment(string, double)`.
  - We presume it **logs** or **executes** a payment.
- **Adapter**:
  - A middle-man class that **implements** the target interface (`IPaymentGateway`) and **delegates** calls to the legacy class.

1.2. Designing the Adapter Class

1.2.1. Naming the Adapter

- I ask: *What's a good name?*
- It should reflect that it "adapts" `LegacyPaymentProcessor` to `IPaymentGateway`.
- Common convention: `XxxAdapter`, so I choose `PaymentProcessorAdapter`.
- **Why include "PaymentProcessor"?**
  - Clarity: it tells future readers *what* it adapts.
  - Avoids generic "Adapter" names.

1.2.2. Defining the Class Skeleton

```
public class PaymentProcessorAdapter : IPaymentGateway
{
    // ...
}
```

- `public`: so any consumer can see it.
- **Extends** (`:`) the interface `IPaymentGateway`.
- **Why implement an interface?**
  - Required: the adapter must conform to the new system's expectations.

1.2.3. Holding a Reference to the Legacy Processor

I need a way for the adapter to call into the old code. That means:

1. A private field:

   ```
   private readonly LegacyPaymentProcessor _legacyProcessor;
   ```

   - `private`: encapsulation, prevents outside tampering.
   - `readonly`: ensures the reference can only be set once (in the constructor). That guards against accidental reassignment.
   - **Naming**: by convention underscore + camelCase (`_legacyProcessor`).
2. A constructor that accepts a `LegacyPaymentProcessor`:

   ```
   public PaymentProcessorAdapter(LegacyPaymentProcessor legacyProcessor)
   {
       _legacyProcessor = legacyProcessor;
   }
   ```

- **Why inject via constructor?**
  - **Dependency Injection (DI)**: allows the caller to supply the specific legacy instance (testable, flexible).
  - Avoids the adapter having to `new` it itself (which would hard-code dependencies).

1.2.4. Implementing the `Pay` Method

The interface says:

```
void Pay(PaymentDetails payment);
```

- **Goal**: inside `Pay`, invoke `legacyProcessor.ProcessPayment` with the right arguments.

1.2.4.1. Mapping Between Types

- `PaymentDetails` has two properties:
  - `string CardNumber { get; set; }`
  - `double Amount { get; set; }`
- **Legacy method** signature:
  - `void ProcessPayment(string cardNumber, double amount)`

I need to extract `payment.CardNumber` and `payment.Amount`, and pass them along.

1.2.4.2. Writing the Method

```
public void Pay(PaymentDetails payment)
{
    // Step 1: Validate inputs? (Optional but wise)
    if (payment == null)
        throw new ArgumentNullException(nameof(payment));
    if (string.IsNullOrWhiteSpace(payment.CardNumber))
        throw new ArgumentException("Card number must be provided",
nameof(payment.CardNumber));
    if (payment.Amount <= 0)
        throw new ArgumentException("Amount must be greater than zero",
nameof(payment.Amount));

    // Step 2: Delegate to legacy
    _legacyProcessor.ProcessPayment(payment.CardNumber, payment.Amount);
}
```

- **Why validate?**

    - Defensive programming: protects the legacy code from bad data.
    - Even if legacy code handles nulls, I want explicit, clear errors.
- **Parameter null check**: `ArgumentNullException` is idiomatic in .NET.
- **Empty/whitespace check**: `string.IsNullOrWhiteSpace` handles `null`, `""`, `" "`.
- **Amount > 0**: We assume zero or negative payments make no sense.

1.2.5. Final Adapter Code

Putting it all together:

```
public class PaymentProcessorAdapter : IPaymentGateway
{
    private readonly LegacyPaymentProcessor _legacyProcessor;

    public PaymentProcessorAdapter(LegacyPaymentProcessor legacyProcessor)
    {
        if (legacyProcessor == null)
            throw new ArgumentNullException(nameof(legacyProcessor));

        _legacyProcessor = legacyProcessor;
    }

    public void Pay(PaymentDetails payment)
    {
        if (payment == null)
            throw new ArgumentNullException(nameof(payment));
        if (string.IsNullOrWhiteSpace(payment.CardNumber))
            throw new ArgumentException("Card number must be provided",
nameof(payment.CardNumber));
        if (payment.Amount <= 0)
            throw new ArgumentException("Amount must be greater than zero",
nameof(payment.Amount));

        _legacyProcessor.ProcessPayment(payment.CardNumber, payment.Amount);
    }
}
```

- **I added** a constructor null-check for the injected processor itself—another layer of safety.
- **Every line** now has reasoning and explicit error handling.
- **I can't modify** `LegacyPaymentProcessor`, so this adapter is the **only bridge**.

1.2.6. Usage Example

```
// Somewhere in composition root or startup:
var legacy = new LegacyPaymentProcessor();
IPaymentGateway gateway = new PaymentProcessorAdapter(legacy);

// Later, when processing:
var details = new PaymentDetails
{
    CardNumber = "1234-5678-9012-3456",
    Amount = 250.00
};
gateway.Pay(details);
```

- **Why use `IPaymentGateway` reference?**
  - Caller only knows about the interface, not the concrete adapter.

🎯 Step 2: Evolving to Asynchronous Support

2.1. The New Requirement

> The legacy processor is synchronous.
> The new system now expects **asynchronous** operations (`Task` -based, `async/await`).

I pause: *Why asynchronous?*

- To **avoid blocking** threads—especially important in high-throughput or UI contexts.
- To integrate with modern .NET patterns (`async` / `await` , `Task` s).

2.2. Challenges

1. **Legacy method** is `void ProcessPayment(...)` .
2. **Async interface** wants `Task PayAsync(PaymentDetails payment)` (or `Task Pay(PaymentDetails payment)` returning a Task).
3. **We cannot** modify `LegacyPaymentProcessor` .

Thus we must **"fake"** async: wrap the sync call in a Task, offloading to a thread-pool thread.

- **Trade-off**: thread-pool thread still blocks while calling legacy code—but at least the caller's thread is free.
- **Caveat**: if legacy is CPU-bound or blocking IO, this merely shifts the blocking.

2.3. Defining the Async Interface

We declare:

```
public interface IPaymentGateway
{
    Task PayAsync(PaymentDetails payment);
}
```

- `Task`: represents an asynchronous operation with no return value.
- Consumers will call `await gateway.PayAsync(details);`

2.4. Updating the Adapter

2.4.1. Class Signature

```
public class PaymentProcessorAdapter : IPaymentGateway
{
    // same private field and constructor
    // ...
}
```

No change here—still implements `IPaymentGateway`.

2.4.2. Implementing `PayAsync`

I ask: *How to wrap synchronous code in a Task?* Two main options:

1. `Task.Run`:
    - Queues work to the thread-pool.
    - Returns a `Task` that completes when delegate finishes.
2. `Task.Factory.StartNew`:
    - More configurable, but often misused (doesn't flow `async` context by default).

Best practice: use `Task.Run` in a library when wrapping CPU-bound work, because it's simple and context-aware.

2.4.2.1. Write the Method

```
public Task PayAsync(PaymentDetails payment)
{
    if (payment == null)
        throw new ArgumentNullException(nameof(payment));
    if (string.IsNullOrWhiteSpace(payment.CardNumber))
```

```
        throw new ArgumentException("Card number must be provided",
nameof(payment.CardNumber));
    if (payment.Amount <= 0)
        throw new ArgumentException("Amount must be greater than zero",
nameof(payment.Amount));

    // Here we “offload” the synchronous call:
    return Task.Run(() =>
        _legacyProcessor.ProcessPayment(payment.CardNumber, payment.Amount)
    );
}
```

- **Why validate again?** Input validation remains crucial in every public method.
- **Why not mark `async`?**
  - If I wrote `public async Task PayAsync(...) { await Task.Run(...); }`, that would introduce an extra state machine—slightly less efficient.
  - Returning the `Task` directly is more concise.

2.4.2.2. Potential Enhancements

- **Cancellation**: If the new interface supported `CancellationToken`, we could honor it:

  ```
  public Task PayAsync(PaymentDetails payment, CancellationToken ct)
  {
      // Validate...
      return Task.Run(() =>
      {
          ct.ThrowIfCancellationRequested();
          _legacyProcessor.ProcessPayment(payment.CardNumber,
  payment.Amount);
      }, ct);
  }
  ```

  - **Why?** Allows callers to cancel long-running payments.
  - **Trade-off**: if legacy code doesn’t poll for cancellation, cancellation only takes effect *before* delegation.
- **Progress Reporting**: Not usually relevant for quick payments.

2.5. The Complete Async Adapter

```
public interface IPaymentGateway
{
    Task PayAsync(PaymentDetails payment);
```

```
}

public class PaymentProcessorAdapter : IPaymentGateway
{
    private readonly LegacyPaymentProcessor _legacyProcessor;

    public PaymentProcessorAdapter(LegacyPaymentProcessor legacyProcessor)
    {
        if (legacyProcessor == null)
            throw new ArgumentNullException(nameof(legacyProcessor));
        _legacyProcessor = legacyProcessor;
    }

    public Task PayAsync(PaymentDetails payment)
    {
        if (payment == null)
            throw new ArgumentNullException(nameof(payment));
        if (string.IsNullOrWhiteSpace(payment.CardNumber))
            throw new ArgumentException("Card number must be provided",
nameof(payment.CardNumber));
        if (payment.Amount <= 0)
            throw new ArgumentException("Amount must be greater than zero",
nameof(payment.Amount));

        return Task.Run(() =>
            _legacyProcessor.ProcessPayment(payment.CardNumber,
payment.Amount)
        );
    }
}
```

- **Every line** is accompanied by validation.
- **Asynchronous bridging** is done via `Task.Run`.

🌐 Reflections, Caveats, and Beyond

1. **Threading Model**
   - By offloading to thread-pool threads, we may **starve** the pool under heavy load.
   - If payment processing is I/O-bound (e.g., network calls), better to put I/O in truly async libraries.
2. **Error Handling**
   - Exceptions thrown by `ProcessPayment` will surface as **faulted** Tasks.

- Callers must `await` (or observe) the Task to catch them.

3. **Performance**
   - `Task.Run` has overhead. For extremely high-throughput scenarios, consider rewriting or batching.
4. **Testability**
   - We can **mock** `LegacyPaymentProcessor` by providing a fake in tests.
   - We can also **simulate** delays using `Task.Delay`.
5. **Alternatives**
   - **Decorator** pattern: if we wanted to wrap additional behavior (logging, retry), we could further decorate this adapter.
   - **Facade**: if we needed to simplify multiple legacy calls, a facade might be appropriate.

🚀 Final Takeaway

1. I **can't** change the legacy code, so an **Adapter** is the perfect structural solution.
2. I **wrapped** the old `ProcessPayment(string, double)` into the new `Pay(PaymentDetails)` method.
3. I **evolved** the adapter to `async` by using `Task.Run`, preserving the legacy implementation under the hood.
4. I surfaced **every assumption**, validated **every input**, and linked **every step** back to first principles of clean design and defensive programming.