Created by DeepSeek R1 Distilled 32B 8bit Running Locally

Prompt used had some minor errors hence some of the output artifacts (bad start, some repeats). The output was produced in 16 minutes on a Mac Studio. No edits. Output in Markdown format, easily exportable to PDF.

This is what you can do with a basic prompt! It took about 3 minutes to write the prompt. If you incorporate staged dialogs the results are truly remarkable.

DeepSeek's 70B 8 bit model is even better, but you'll need lots of RAM to run it locally.

6. Adapter Pattern

- **Code Example**: Convert the interface of a class into another interface that clients expect.

```
public interface ITarget
{
    void Request();
}

public class Adaptee
{
    public void SpecificRequest() => Console.WriteLine("Adaptee's specific request");
}

public class Adapter : ITarget
{
    private readonly Adaptee _adaptee;

    public Adapter(Adaptee adaptee) => _adaptee = adaptee;

    public void Request() => _adaptee.SpecificRequest();
}

var adaptee = new Adaptee();
ITarget target = new Adapter(adaptee);
target.Request();
```

- **Use Case**: Useful for integrating legacy systems or third-party libraries with incompatible interfaces.

7. Proxy Pattern

- **Code Example**: Provide a surrogate or placeholder for another object.

```
public interface IImage
{
    void Display();
}

public class RealImage : IImage
{
    private string _filename;

    public RealImage(string filename)
    {
        _filename = filename;
        LoadFromDisk();
    }

    private void LoadFromDisk() => Console.WriteLine($"Loading {_filename} from disk");

    public void Display() => Console.WriteLine($"Displaying {_filename}");
}

public class ProxyImage : IImage
{
    private RealImage _realImage;
    private string _filename;

    public ProxyImage(string filename) => _filename = filename;

    public void Display()
    {
        if (_realImage == null)
```

```
            _realImage = new RealImage(_filename);
        _realImage.Display();
    }
}

IImage image = new ProxyImage("test.jpg");
image.Display(); // Loads and displays on first call
```

- **Use Case**: Useful for lazy loading, access control, or adding additional layers of security.

8. Chain of Responsibility Pattern

- **Code Example**: Passes requests along a chain of handlers until one handles it.

```
public interface IHandler
{
    void SetNext(IHandler handler);
    void HandleRequest(string request);
}

public abstract class AbstractHandler : IHandler
{
    protected IHandler next;

    public void SetNext(IHandler handler) => next = handler;

    public abstract void HandleRequest(string request);
}

public class ConcreteHandlerA : AbstractHandler
{
    public override void HandleRequest(string request)
    {
        if (request == "A")
            Console.WriteLine("ConcreteHandlerA handles the request");
        else if (next != null)
            next.HandleRequest(request);
    }
}

public class ConcreteHandlerB : AbstractHandler
{
    public override void HandleRequest(string request)
    {
        if (request == "B")
            Console.WriteLine("ConcreteHandlerB handles the request");
        else if (next != null)
            next.HandleRequest(request);
    }
}

var handlerA = new ConcreteHandlerA();
var handlerB = new ConcreteHandlerB();
handlerA.SetNext(handlerB);
handlerA.HandleRequest("A"); // Handler A handles it
handlerA.HandleRequest("B"); // Handler B handles it
```

- **Use Case**: Useful for workflow management or logging systems where each handler decides whether to process the request.

9. Command Pattern

- **Code Example**: Encapsulates a request as an object, allowing for logging, queuing, and undo operations.

```
public interface ICommand
{
    void Execute();
    void Undo();
}

public class ConcreteCommand : ICommand
{
    private readonly Receiver _receiver;

    public ConcreteCommand(Receiver receiver) => _receiver = receiver;
```

```
    public void Execute() => _receiver.Action();
    public void Undo() => _receiver.UndoAction();
}

public class Receiver
{
    public void Action() => Console.WriteLine("Receiver action executed");
    public void UndoAction() => Console.WriteLine("Undo receiver action");
}

public class Invoker
{
    private readonly ICommand _command;

    public Invoker(ICommand command) => _command = command;

    public void ExecuteCommand() => _command.Execute();
    public void UndoCommand() => _command.Undo();
}

var receiver = new Receiver();
ICommand command = new ConcreteCommand(receiver);
Invoker invoker = new(command);
invoker.ExecuteCommand();
invoker.UndoCommand();
```

- **Use Case**: Useful for implementing undo/redo functionality, task queues, or command-line interfaces.

10. Iterator Pattern

- **Code Example**: Provides a way to access elements of an aggregate object sequentially without exposing its underlying structure.

```
public interface IIterator
{
    bool HasNext();
    object Next();
}

public class ConcreteIterator : IIterator
{
    private readonly List<object> _items;
    private int _index;

    public ConcreteIterator(List<object> items)
    {
        _items = items;
        _index = 0;
    }

    public bool HasNext() => _index < _items.Count;

    public object Next()
    {
        if (HasNext())
            return _items[_index++];
        throw new InvalidOperationException("No more elements");
    }
}

public class Aggregate
{
    private readonly List<object> _items = new();

    public void Add(object item) => _items.Add(item);
    public IIterator CreateIterator() => new ConcreteIterator(_items);
}

var aggregate = new Aggregate();
aggregate.Add(1);
aggregate.Add(2);
IIterator iterator = aggregate.CreateIterator();

while (iterator.HasNext())
    Console.WriteLine(iterator.Next());
```

- **Use Case**: Useful for iterating over collections in a standardized way, especially in GUI or data processing applications.

11. Template Method Pattern

- **Code Example**: Defines the skeleton of an algorithm, deferring some steps to subclasses.

```
public abstract class AbstractClass
{
    public void TemplateMethod()
    {
        Step1();
        Step2();
        Step3();
    }

    protected abstract void Step1();
    protected abstract void Step2();
    protected virtual void Step3() => Console.WriteLine("Default Step 3");
}

public class ConcreteClassA : AbstractClass
{
    protected override void Step1() => Console.WriteLine("Concrete A Step 1");
    protected override void Step2() => Console.WriteLine("Concrete A Step 2");
}

public class ConcreteClassB : AbstractClass
{
    protected override void Step1() => Console.WriteLine("Concrete B Step 1");
    protected override void Step2() => Console.WriteLine("Concrete B Step 2");
    protected override void Step3() => Console.WriteLine("Concrete B Step 3");
}

var a = new ConcreteClassA();
a.TemplateMethod();

var b = new ConcreteClassB();
b.TemplateMethod();
```

- **Use Case**: Useful for frameworks where common algorithms need to be customized by subclasses.

12. Composite Pattern

- **Code Example**: Composes objects into tree structures to represent part-whole hierarchies.

```
public interface IComponent
{
    void Add(IComponent component);
    void Remove(IComponent component);
    void Display(int depth);
}

public class Leaf : IComponent
{
    public string Name { get; set; }

    public void Add(IComponent component) => throw new InvalidOperationException("Cannot add to a leaf");

    public void Remove(IComponent component) => throw new InvalidOperationException("Cannot remove from a
leaf");

    public void Display(int depth)
    {
        Console.Write(new string(' ', depth * 2));
        Console.WriteLine(Name);
    }
}

public class Composite : IComponent
{
    private readonly List<IComponent> _children = new();

    public void Add(IComponent component) => _children.Add(component);
    public void Remove(IComponent component) => _children.Remove(component);
```

```
    public void Display(int depth)
    {
        Console.WriteLine($"Composite with {nameof(Composite)}");
        foreach (var child in _children)
            child.Display(depth + 1);
    }
}

var composite = new Composite();
composite.Add(new Leaf { Name = "Leaf 1" });
var subComposite = new Composite();
subComposite.Add(new Leaf { Name = "Leaf 2" });
composite.Add(subComposite);
composite.Display(0);
```

- **Use Case**: Useful for managing hierarchical structures like organizational charts or file systems.

13. State Pattern

- **Code Example**: Allows an object to alter its behavior when its internal state changes.

```
public interface IState
{
    void Handle();
}

public class ConcreteStateA : IState
{
    public void Handle() => Console.WriteLine("Handling in State A");
}

public class ConcreteStateB : IState
{
    public void Handle() => Console.WriteLine("Handling in State B");
}

public class Context
{
    private IState _state;

    public void SetState(IState state) => _state = state;
    public void Request() => _state.Handle();
}

var context = new Context();
context.SetState(new ConcreteStateA());
context.Request();

context.SetState(new ConcreteStateB());
context.Request();
```

- **Use Case**: Useful for modeling state-dependent behavior in applications like traffic lights or user sessions.

14. Flyweight Pattern

- **Code Example**: Minimizes memory usage by sharing as much data as possible among multiple similar objects.

```
public class FlyweightFactory
{
    private readonly Dictionary<string, Flyweight> _flyweights = new();

    public Flyweight GetFlyweight(string key)
    {
        if (!_flyweights.ContainsKey(key))
            _flyweights[key] = new Flyweight(key);
        return _flyweights[key];
    }
}

public class Flyweight
{
    private readonly string _intrinsicState;

    public Flyweight(string state) => _intrinsicState = state;
```

```
    public void Operation(int extrinsicState)
        => Console.WriteLine($"Flyweight: {_intrinsicState}, State: {extrinsicState}");
}

var factory = new FlyweightFactory();
var f1 = factory.GetFlyweight("A");
var f2 = factory.GetFlyweight("A");

Console.WriteLine(f1 == f2); // True
```

- **Use Case**: Useful for managing large numbers of objects, such as in graphics applications or text editors.

15. Interpreter Pattern

- **Code Example**: Defines a language and its interpreter to process sentences in that language.

```
public interface IExpression { bool Interpret(string context); }

public class TerminalExpression : IExpression
{
    private readonly string _data;

    public TerminalExpression(string data) => _data = data;

    public bool Interpret(string context) => context.Contains(_data);
}

public class OrExpression : IExpression
{
    private readonly IExpression _left;
    private readonly IExpression _right;

    public OrExpression(IExpression left, IExpression right)
    {
        _left = left;
        _right = right;
    }

    public bool Interpret(string context) => _left.Interpret(context) || _right.Interpret(context);
}

var expression = new OrExpression(
    new TerminalExpression("Hello"),
    new TerminalExpression("World")
);
Console.WriteLine(expression.Interpret("Hello World")); // True
```

- **Use Case**: Useful for implementing domain-specific languages or query interpreters.

16. Mediator Pattern

- **Code Example**: Encourages communication between classes through a mediator instead of directly.

```
public interface IMediator { void Send(string message, Colleague colleague); }

public class ConcreteMediator : IMediator
{
    private readonly List<Colleague> _colleagues = new();

    public void Send(string message, Colleague colleague)
        => _colleagues.ForEach(c => c.Receive(message));
}

public abstract class Colleague
{
    protected IMediator mediator;

    public void SetMediator(IMediator mediator) => this.mediator = mediator;
    public abstract void Receive(string message);
}

public class ConcreteColleagueA : Colleague
{
    public override void Receive(string message) => Console.WriteLine("Colleague A: " + message);
```

```
}

public class ConcreteColleagueB : Colleague
{
    public override void Receive(string message) => Console.WriteLine("Colleague B: " + message);
}

var mediator = new ConcreteMediator();
var colleagueA = new ConcreteColleagueA { mediator = mediator };
var colleagueB = new ConcreteColleagueB { mediator = mediator };

mediator.Send("Hello", colleagueA);
```

- **Use Case**: Useful for managing interactions in complex systems like chat applications or GUI components.

17. Memento Pattern

- **Code Example**: Captures and restores an object's previous state without revealing details of the state.

```
public class Originator
{
    private string _state;

    public void SetState(string state) => this._state = state;
    public IMemento Save() => new Memento(_state);
    public void Restore(IMemento memento) => _state = ((Memento)memento).State;

    public override string ToString() => $"Originator State: {_state}";
}

public interface IMemento { string State { get; } }

public class Memento : IMemento
{
    public string State { get; }

    public Memento(string state) => this.State = state;
}

var originator = new Originator();
originator.SetState("State1");
Console.WriteLine(originator);

IMemento memento = originator.Save();
originator.SetState("State2");
Console.WriteLine(originator);

originator.Restore(memento);
Console.WriteLine(originator);
```

- **Use Case**: Useful for implementing undo/redo functionality or checkpointing in applications.

18. Visitor Pattern

- **Code Example**: Adds methods to objects without changing their classes.

```
public interface IElement { void Accept(IVisitor visitor); }

public class ConcreteElementA : IElement
{
    public void Accept(IVisitor visitor) => visitor.visit(this);
}

public interface IVisitor
{
    void visit(ConcreteElementA element);
    void visit(ConcreteElementB element);
}

public class ConcreteVisitor : IVisitor
{
    public void visit(ConcreteElementA element) => Console.WriteLine("Visited A");
    public void visit(ConcreteElementB element) => Console.WriteLine("Visited B");
}
```

```
var elements = new List<IElement> { new ConcreteElementA(), new ConcreteElementB() };
var visitor = new ConcreteVisitor();
foreach (var element in elements)
    element.Accept(visitor);
```

- **Use Case**: Useful for operations that need to be applied to a hierarchy of elements, such as traversing an object structure.

19. Double Checked Locking Pattern

- **Code Example**: Ensures that a singleton is initialized only once, even in multi-threaded environments.

```
public sealed class Singleton
{
    private static volatile Singleton _instance;
    private static object _lock = new();

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (_instance == null)
            {
                lock (_lock)
                {
                    if (_instance == null)
                        _instance = new Singleton();
                }
            }
            return _instance;
        }
    }
}
```

- **Use Case**: Useful for ensuring thread-safe singleton creation in concurrent applications.

20. Thread Pool Pattern

- **Code Example**: Manages a pool of worker threads to handle asynchronous tasks.

```
public class ThreadPool
{
    private readonly BlockingCollection<Action> _queue = new();

    public void AddTask(Action task) => _queue.Add(task);

    public void Start()
    {
        foreach (var task in _queue.GetConsumingEnumerable())
            task();
    }
}

var pool = new ThreadPool();
pool.AddTask(() => Console.WriteLine("Task 1"));
pool.AddTask(() => Console.WriteLine("Task 2"));
pool.Start();
```

- **Use Case**: Useful for managing background tasks and improving application responsiveness.

21. Task Parallel Library (TPL) Pattern

- **Code Example**: Utilizes TPL for parallel execution.

```
var tasks = new List<Task>
{
    Task.Run(() => Console.WriteLine("Task 1")),
    Task.Run(() => Console.WriteLine("Task 2"))
};

await Task.WhenAll(tasks);
```

- **Use Case**: Useful for parallelizing operations to leverage multi-core processors.

22. Background Worker Pattern

- **Code Example**: Executes an operation on a background thread.

```
var worker = new BackgroundWorker();
worker.DoWork += (sender, e) =>
{
    Console.WriteLine("Background task running");
    Thread.Sleep(1000);
};

worker.RunWorkerAsync();
```

- **Use Case**: Useful for performing time-consuming operations without blocking the UI.

23. Proxy Pattern with Cancellation

- **Code Example**: Adds support for cancellation to a proxy.

```
public interface IOperation { Task DoWork(CancellationToken token); }
public class RealOperation : IOperation
{
    public async Task DoWork(CancellationToken token)
    {
        for (int i = 0; i < 10; i++)
        {
            if (token.IsCancellationRequested)
                break;
            Console.WriteLine($"Working: {i}");
            await Task.Delay(100);
        }
    }
}

public class CancellationProxy : IOperation
{
    private readonly RealOperation _real;

    public CancellationProxy(RealOperation real) => _real = real;

    public async Task DoWork(CancellationToken token)
        => await _real.DoWork(token);
}

var real = new RealOperation();
var proxy = new CancellationProxy(real);
CancellationTokenSource cts = new();

var task = proxy.DoWork(cts.Token);
Thread.Sleep(500);
cts.Cancel();
await task;
```

- **Use Case**: Useful for adding cancellation support to long-running operations.

24. Observer Pattern with Events

- **Code Example**: Uses events to implement the observer pattern.

```
public class Subject
{
    public event EventHandler <int> ValueChanged;

    private int _value;
    public int Value
    {
        get => _value;
        set
        {
            if (_value != value)
            {
```

```
                _value = value;
                ValueChanged?.Invoke(this, _value);
            }
        }
    }
}

public class Observer
{
    private readonly Subject _subject;

    public Observer(Subject subject)
    {
        _subject = subject;
        _subject.ValueChanged += OnValueChanged;
    }

    private void OnValueChanged(object sender, int value)
        => Console.WriteLine($"Value changed to: {value}");
}

var subject = new Subject();
var observer = new Observer(subject);
subject.Value = 10;
```

- **Use Case**: Useful for real-time updates in GUI applications or event-driven systems.

25. Command Pattern with History

- **Code Example**: Implements a command history to support undo operations.

```
public interface ICommand { void Execute(); void Undo(); }

public class AddCommand : ICommand
{
    private readonly List<int> _list;
    private int _value;

    public AddCommand(List<int> list, int value)
    {
        _list = list;
        _value = value;
    }

    public void Execute() => _list.Add(_value);
    public void Undo() => _list.Remove(_value);
}

public class History
{
    private readonly Stack<ICommand> _commands = new();

    public void Push(ICommand command) => _commands.Push(command);
    public void Undo() => _commands.Pop()?.Undo();
}

var list = new List<int>();
History history = new();

var addCommand = new AddCommand(list, 5);
addCommand.Execute();
history.Push(addCommand);

Console.WriteLine($"List: {string.Join(", ", list)}"); // 5
history.Undo();
Console.WriteLine($"List: {string.Join(", ", list)}"); // Empty
```

- **Use Case**: Useful for applications requiring undo functionality, such as text editors or CAD software.

26. Unit of Work Pattern

- **Code Example**: Manages a unit of work for data access.

```
public interface IUnitOfWork
{
```

```
    void Commit();
    void Rollback();
}

public class UnitOfWork : IUnitOfWork
{
    private bool _committed;

    public void Commit()
    {
        if (!_committed)
        {
            // Perform commit operations
            Console.WriteLine("Committing transaction");
            _committed = true;
        }
    }

    public void Rollback()
    {
        if (!_committed)
        {
            // Perform rollback operations
            Console.WriteLine("Rolling back transaction");
        }
    }
}

using (var uow = new UnitOfWork())
{
    // Perform operations
    uow.Commit();
}
```

- **Use Case**: Useful for managing transactions in data access layers.

27. Repository Pattern

- **Code Example**: Abstracts data access into a repository.

```
public interface IRepository<TEntity>
{
    void Add(TEntity entity);
    TEntity Get(int id);
    void Remove(TEntity entity);
}

public class Repository<TEntity> : IRepository<TEntity>
{
    private readonly List<TEntity> _entities = new();
    public void Add(TEntity entity) => _entities.Add(entity);
    public TEntity Get(int id) => throw new NotImplementedException(); // Implement ID-based retrieval
    public void Remove(TEntity entity) => _entities.Remove(entity);
}

var repo = new Repository<Customer>();
repo.Add(new Customer { Name = "Alice" });
```

- **Use Case**: Useful for encapsulating data access logic in enterprise applications.

28. Specification Pattern

- **Code Example**: Encapsulates business rules in reusable specifications.

```
public interface ISpecification<T>
{
    bool IsSatisfiedBy(T entity);
}

public class AndSpecification<T> : ISpecification<T>
{
    private readonly ISpecification<T> _left;
    private readonly ISpecification<T> _right;
```

```
    public AndSpecification(ISpecification<T> left, ISpecification<T> right)
    {
        _left = left;
        _right = right;
    }

    public bool IsSatisfiedBy(T entity) => _left.IsSatisfiedBy(entity) && _right.IsSatisfiedBy(entity);
}

public class Customer
{
    public int Age { get; set; }
    public string Name { get; set; }
}

public class AgeSpecification : ISpecification<Customer>
{
    private readonly int _minimumAge;

    public AgeSpecification(int minimumAge) => _minimumAge = minimumAge;

    public bool IsSatisfiedBy(Customer entity) => entity.Age >= _minimumAge;
}

var spec = new AndSpecification<Customer>(
    new AgeSpecification(21),
    new NameSpecification("Alice")
);

var customer = new Customer { Age = 25, Name = "Alice" };
Console.WriteLine(spec.IsSatisfiedBy(customer)); // True
```

- **Use Case**: Useful for encapsulating complex business rules and queries.

29. Domain Events Pattern

- **Code Example**: Publishes events within the domain model.

```
public interface IDomainEvent {}

public class OrderPlaced : IDomainEvent
{
    public Guid OrderId { get; }

    public OrderPlaced(Guid orderId) => OrderId = orderId;
}

public interface IDomainEventPublisher
{
    void Publish(IDomainEvent @event);
}

public class DomainEventPublisher : IDomainEventPublisher
{
    public void Publish(IDomainEvent @event)
    {
        // Implement event publishing logic
        Console.WriteLine($"Published: {@event.GetType().Name}");
    }
}

var event = new OrderPlaced(Guid.NewGuid());
IDomainEventPublisher publisher = new DomainEventPublisher();
publisher.Publish(event);
```

- **Use Case**: Useful for decoupling components in domain-driven design.

30. CQRS Pattern

- **Code Example**: Separates read and write operations.

```
public interface ICommandHandler<TCommand>
{
    void Handle(TCommand command);
}
```

```
public interface IQueryHandler<TQuery, TResult>
{
    TResult Handle(TQuery query);
}

public class PlaceOrderCommand { }

public class OrderPlacedHandler : ICommandHandler<PlaceOrderCommand>
{
    public void Handle(PlaceOrderCommand command) => Console.WriteLine("Order placed");
}

var command = new PlaceOrderCommand();
var handler = new OrderPlacedHandler();
handler.Handle(command);
```

- **Use Case**: Useful for improving performance and scalability in enterprise applications.

31. Hexagonal Architecture Pattern

- **Code Example**: Layers the application to isolate core business logic.

```
public interface IUseCase { void Execute(); }

public class CreateOrderUseCase : IUseCase
{
    private readonly IOrderRepository _repository;

    public CreateOrderUseCase(IOrderRepository repository) => _repository = repository;

    public void Execute() => _repository.Save(new Order());
}

public interface IOrderRepository { void Save(Order order); }
public class OrderRepository : IOrderRepository
{
    public void Save(Order order) => Console.WriteLine("Order saved");
}

var repository = new OrderRepository();
var useCase = new CreateOrderUseCase(repository);
useCase.Execute();
```

- **Use Case**: Useful for building maintainable and testable enterprise applications.

32. Event Sourcing Pattern

- **Code Example**: Stores application state as a sequence of events.

```
public interface IEventStore
{
    void Save(Guid aggregateId, IEnumerable<Event> events);
    IEnumerable<Event> Get(Guid aggregateId);
}

public class EventStore : IEventStore
{
    private readonly Dictionary<Guid, List<Event>> _events = new();

    public void Save(Guid aggregateId, IEnumerable<Event> events)
    {
        if (!_events.ContainsKey(aggregateId))
            _events[aggregateId] = new List<Event>();
        _events[aggregateId].AddRange(events);
    }

    public IEnumerable<Event> Get(Guid aggregateId) => _events.GetValueOrDefault(aggregateId);
}

public class Order
{
    private List<Event> _events = new();

    public void PlaceOrder() => Apply(new OrderPlaced());
```

```
    private void Apply(Event @event) => _events.Add(@event);
}

var store = new EventStore();
var order = new Order();
order.PlaceOrder();
store.Save(Guid.NewGuid(), order._events);
```

- **Use Case**: Useful for applications requiring full audit trails and versioning.

33. Command Bus Pattern

- **Code Example**: Routes commands to appropriate handlers.

```
public interface ICommandBus { void Send<T>(T command); }

public class CommandBus : ICommandBus
{
    private readonly Dictionary<Type, object> _handlers = new();

    public void RegisterHandler<T>(ICommandHandler<T> handler)
        => _handlers[typeof(T)] = handler;

    public void Send<T>(T command) where T : ICommand
    {
        if (_handlers.TryGetValue(typeof(T), out var handler))
            ((ICommandHandler<T>)handler).Handle(command);
    }
}

public interface ICommand { }

public class PlaceOrderCommand : ICommand {}

public interface ICommandHandler<T> where T : ICommand
{
    void Handle(T command);
}

public class PlaceOrderHandler : ICommandHandler<PlaceOrderCommand>
{
    public void Handle(PlaceOrderCommand command) => Console.WriteLine("Handled Place Order");
}

var bus = new CommandBus();
bus.RegisterHandler(new PlaceOrderHandler());
bus.Send(new PlaceOrderCommand());
```

- **Use Case**: Useful for building scalable and decoupled command processing systems.

34. Query Bus Pattern

- **Code Example**: Routes queries to appropriate handlers.

```
public interface IQueryBus { TResult Send<TResult>(IQuery<TResult> query); }

public class QueryBus : IQueryBus
{
    private readonly Dictionary<Type, object> _handlers = new();

    public void RegisterHandler<TQuery, TResult>(IQueryHandler<TQuery, TResult> handler)
        where TQuery : IQuery<TResult>
    {
        _handlers[typeof(TQuery)] = handler;
    }

    public TResult Send<TResult>(IQuery<TResult> query)
    {
        var type = typeof(TQuery);
        if (_handlers.TryGetValue(type, out var handler))
            return ((IQueryHandler<TQuery, TResult>)handler).Handle((TQuery)query);
        throw new InvalidOperationException("No handler registered");
    }
}
```

```
public interface IQuery<TResult> {}

public class GetOrderQuery : IQuery<Order> { }

public interface IQueryHandler<TQuery, TResult>
    where TQuery : IQuery<TResult>
{
    TResult Handle(TQuery query);
}

public class GetOrderHandler : IQueryHandler<GetOrderQuery, Order>
{
    public Order Handle(GetOrderQuery query) => new();
}

var bus = new QueryBus();
bus.RegisterHandler(new GetOrderHandler());
var result = bus.Send(new GetOrderQuery());
```

- **Use Case**: Useful for decoupling query processing from the rest of the application.

35. Ports and Adapters Pattern

- **Code Example**: Isolates the application core from external systems.

```
public interface IPort { void Request(); }

public class ApplicationCore
{
    private readonly IPort _inPort;

    public ApplicationCore(IPort inPort) => _inPort = inPort;

    public void Process() => _inPort.Request();
}

public class Adapter : IPort
{
    private readonly ExternalSystem _external;

    public Adapter(ExternalSystem external) => _external = external;

    public void Request() => _external.Execute();
}

public class ExternalSystem
{
    public void Execute() => Console.WriteLine("External system executed");
}

var external = new ExternalSystem();
var adapter = new Adapter(external);
var core = new ApplicationCore(adapter);
core.Process();
```

- **Use Case**: Useful for building maintainable and testable applications with external dependencies.

36. Sagas Pattern

- **Code Example**: Manages long-running transactions across multiple services.

```
public interface ISaga<T>
{
    void Handle(T message);
    bool IsComplete { get; }
}

public class OrderSaga : ISaga<OrderPlaced>
{
    public bool IsComplete { get; private set; }

    public void Handle(OrderPlaced message)
    {
        Console.WriteLine("Handling Order Saga");
        IsComplete = true;
```

```
    }
}

public class OrderPlaced { }

var saga = new OrderSaga();
saga.Handle(new OrderPlaced());
Console.WriteLine(saga.IsComplete); // True
```

- **Use Case**: Useful for coordinating long-running business processes in distributed systems.

37. Service Layer Pattern

- **Code Example**: Provides a layer of abstraction for business logic.

```
public interface IService { void DoWork(); }

public class Service : IService
{
    private readonly IUnitOfWork _unitOfWork;

    public Service(IUnitOfWork unitOfWork) => _unitOfWork = unitOfWork;

    public void DoWork()
    {
        // Business logic
        _unitOfWork.Commit();
    }
}

public interface IUnitOfWork { void Commit(); }

public class UnitOfWork : IUnitOfWork
{
    public void Commit() => Console.WriteLine("Committing units of work");
}

var unitOfWork = new UnitOfWork();
var service = new Service(unitOfWork);
service.DoWork();
```

- **Use Case**: Useful for encapsulating business logic and data access in enterprise applications.

38. Data Mapper Pattern

- **Code Example**: Maps between domain objects and database records.

```
public interface IDataMapper<T> where T : class, new()
{
    T Load(int id);
    void Save(T entity);
}

public class CustomerMapper : IDataMapper<Customer>
{
    public Customer Load(int id) => new();
    public void Save(Customer entity) => Console.WriteLine("Saved customer");
}

var mapper = new CustomerMapper();
var customer = mapper.Load(1);
mapper.Save(customer);
```

- **Use Case**: Useful for data persistence in domain-driven design.

39. Identity Map Pattern

- **Code Example**: Caches objects to prevent duplicate instances.

```
public interface IIdentityMap<TKey, TEntity> where TEntity : class
{
    TEntity Get(TKey key);
    void Put(TEntity entity);
}
```

```
public class IdentityMap<TKey, TEntity> : IIdentityMap<TKey, TEntity>
    where TEntity : class
{
    private readonly Dictionary<TKey, TEntity> _map = new();

    public TEntity Get(TKey key) => _map.GetValueOrDefault(key);
    public void Put(TEntity entity) => _map.Add(entity.GetKey(), entity);

    private TKey GetKey() { throw new NotImplementedException(); }
}

public class Customer
{
    public int Id { get; set; }
    public TKey GetKey() => Id;
}

var map = new IdentityMap<int, Customer>();
var customer = new Customer { Id = 1 };
map.Put(customer);
Console.WriteLine(map.Get(1) == customer); // True
```

- **Use Case**: Useful for managing object identity in data access layers.

40. Row Data Gateway Pattern

- **Code Example**: Represents a row in a data source.

```
public interface IRowDataGateway<T>
{
    T Find(int id);
    void Save(T entity);
}

public class CustomerGateway : IRowDataGateway<Customer>
{
    public Customer Find(int id) => new();
    public void Save(Customer entity) => Console.WriteLine("Saved customer");
}

var gateway = new CustomerGateway();
var customer = gateway.Find(1);
gateway.Save(customer);
```

- **Use Case**: Useful for simple data access operations.

41. Table Data Gateway Pattern

- **Code Example**: Represents a table in a data source.

```
public interface ITableDataGateway<T>
{
    void Insert(T entity);
    void Update(T entity);
    void Delete(int id);
}

public class CustomerGateway : ITableDataGateway<Customer>
{
    public void Insert(Customer entity) => Console.WriteLine("Inserted customer");
    public void Update(Customer entity) => Console.WriteLine("Updated customer");
    public void Delete(int id) => Console.WriteLine("Deleted customer with ID: " + id);
}

var gateway = new CustomerGateway();
gateway.Insert(new Customer());
```

- **Use Case**: Useful for managing operations on entire data tables.

42. Active Record Pattern

- **Code Example**: Combines data access logic with domain objects.

```
public class Customer : ActiveRecord<Customer>
{
    public int Id { get; set; }
    public string Name { get; set; }

    public void Save() => base.Save();
}

public abstract class ActiveRecord<T>
{
    protected virtual void Save() => Console.WriteLine("Saved " + typeof(T).Name);
}

var customer = new Customer { Name = "Alice" };
customer.Save();
```

- **Use Case**: Useful for simplifying data access in domain objects.

43. Data Transfer Object Pattern

- **Code Example**: Transfers data between layers without exposing domain objects.

```
public class CustomerDTO { public int Id { get; set; } }
public class Service
{
    private readonly ICustomerRepository _repository;

    public Service(ICustomerRepository repository) => _repository = repository;

    public CustomerDTO Get(int id)
    {
        var customer = _repository.GetById(id);
        return new CustomerDTO { Id = customer.Id };
    }
}

public interface ICustomerRepository
{
    Customer GetById(int id);
}

var service = new Service(new CustomerRepository());
var dto = service.Get(1);
```

- **Use Case**: Useful for decoupling data transfer from business logic.

44. Dependency Injection Pattern

- **Code Example**: Injects dependencies into classes.

```
public interface ILogger { void Log(string message); }

public class Logger : ILogger
{
    public void Log(string message) => Console.WriteLine(message);
}

public class Service
{
    private readonly ILogger _logger;

    public Service(ILogger logger) => _logger = logger;

    public void DoWork() => _logger.Log("Work done");
}

var logger = new Logger();
var service = new Service(logger);
service.DoWork();
```

- **Use Case**: Useful for building maintainable and testable applications.

45. Abstract Factory Pattern

- **Code Example**: Provides an interface for creating families of related objects.

```
public interface IAbstractFactory { IProduct CreateProduct(); }

public interface IProduct { void Use(); }

public class ConcreteFactory : IAbstractFactory
{
    public IProduct CreateProduct() => new ConcreteProduct();
}

public class ConcreteProduct : IProduct
{
    public void Use() => Console.WriteLine("Using concrete product");
}

var factory = new ConcreteFactory();
var product = factory.CreateProduct();
product.Use();
```

- **Use Case**: Useful for creating families of related objects in a decoupled way.

46. Builder Pattern

- **Code Example**: Separates object construction from its representation.

```
public interface IBuilder { void BuildPartA(); void BuildPartB(); Product GetResult(); }

public class ConcreteBuilder : IBuilder
{
    private readonly Product _product = new();

    public void BuildPartA() => _product.Add("Part A");
    public void BuildPartB() => _product.Add("Part B");

    public Product GetResult() => _product;
}

public class Director
{
    private readonly IBuilder _builder;

    public Director(IBuilder builder) => _builder = builder;

    public Product Construct()
    {
        _builder.BuildPartA();
        _builder.BuildPartB();
        return _builder.GetResult();
    }
}

var builder = new ConcreteBuilder();
var director = new Director(builder);
var product = director.Construct();
```

- **Use Case**: Useful for constructing complex objects with multiple parts.

47. Factory Method Pattern

- **Code Example**: Creates objects without specifying their exact class.

```
public abstract class Creator { public abstract Product FactoryMethod(); }

public class ConcreteCreator : Creator
{
    public override Product FactoryMethod() => new ConcreteProduct();
}

public abstract class Product { }
public class ConcreteProduct : Product {}

var creator = new ConcreteCreator();
var product = creator.FactoryMethod();
```

- **Use Case**: Useful for creating objects in a decoupled manner.

48. Singleton Pattern with Lazy Initialization

- **Code Example**: Ensures a class has only one instance, created on demand.

```
public sealed class Singleton
{
    private static readonly Lazy<Singleton> _instance = new(() => new Singleton());

    private Singleton() { }

    public static Singleton Instance => _instance.Value;
}

Console.WriteLine(Singleton.Instance);
```

- **Use Case**: Useful for controlling resource-intensive objects.

49. Prototype Pattern

- **Code Example**: Creates new objects by cloning existing ones.

```
public interface IPrototype { IPrototype Clone(); }

public class ConcretePrototype : IPrototype
{
    public IPrototype Clone() => new ConcretePrototype();
    public override string ToString() => nameof(ConcretePrototype);
}

var prototype = new ConcretePrototype();
var clone = prototype.Clone();
Console.WriteLine(clone);
```

- **Use Case**: Useful for creating objects that are expensive to create.

50. Adapter Pattern

- **Code Example**: Converts the interface of a class into another interface.

```
public interface ITarget { void Request(); }

public class Adaptee { public void SpecificRequest() => Console.WriteLine("Specific request"); }

public class Adapter : ITarget
{
    private readonly Adaptee _adaptee;

    public Adapter(Adaptee adaptee) => _adaptee = adaptee;

    public void Request() => _adaptee.SpecificRequest();
}

var adaptee = new Adaptee();
var adapter = new Adapter(adaptee);
adapter.Request();
```

- **Use Case**: Useful for integrating incompatible interfaces.

51. Decorator Pattern

- **Code Example**: Dynamically adds responsibilities to objects.

```
public interface IComponent { void Operation(); }

public class ConcreteComponent : IComponent
{
    public void Operation() => Console.WriteLine("Concrete component");
}

public class Decorator : IComponent
```

```
{
    protected readonly IComponent _component;

    public Decorator(IComponent component) => _component = component;
    public virtual void Operation() => _component.Operation();
}

public class ConcreteDecorator : Decorator
{
    public override void Operation()
    {
        base.Operation();
        Console.WriteLine("Decorator");
    }
}

var component = new ConcreteComponent();
var decorator = new ConcreteDecorator(component);
decorator.Operation();
```

- **Use Case**: Useful for adding responsibilities to objects dynamically.

52. Composite Pattern with Security

- **Code Example**: Composes objects and adds security checks.

```
public interface IComponent { void Display(); }

public class Leaf : IComponent
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine(Name);
}

public class SecureComposite : IComponent
{
    private readonly List<IComponent> _children = new();
    private readonly string _permission;

    public SecureComposite(string permission) => _permission = permission;

    public void Display()
    {
        if (CheckPermission())
            foreach (var child in _children)
                child.Display();
    }

    protected virtual bool CheckPermission() => !string.IsNullOrEmpty(_permission);
}

var composite = new SecureComposite("Admin");
composite._children.Add(new Leaf { Name = "Item 1" });
composite.Display();
```

- **Use Case**: Useful for managing security in hierarchical structures.

53. Decorator Pattern with Logging

- **Code Example**: Logs operations on decorated objects.

```
public interface IComponent { void Operation(); }

public class ConcreteComponent : IComponent
{
    public void Operation() => Console.WriteLine("Concrete component");
}

public class LoggingDecorator : IComponent
{
    private readonly IComponent _component;

    public LoggingDecorator(IComponent component) => _component = component;

    public void Operation()
```

```
    {
        Console.WriteLine("Operation started");
        _component.Operation();
        Console.WriteLine("Operation ended");
    }
}

var component = new ConcreteComponent();
var decorator = new LoggingDecorator(component);
decorator.Operation();
```

- **Use Case**: Useful for adding logging to objects dynamically.

54. Adapter Pattern with File Systems

- **Code Example**: Adapts different file systems to a common interface.

```
public interface IFileSystem { void Read(string path); }

public class LocalFileSystem : IFileSystem
{
    public void Read(string path) => Console.WriteLine("Reading local file: " + path);
}

public class CloudFileSystem
{
    public void Download(string path) => Console.WriteLine("Downloading from cloud: " + path);
}

public class CloudFileSystemAdapter : IFileSystem
{
    private readonly CloudFileSystem _cloud;

    public CloudFileSystemAdapter(CloudFileSystem cloud) => _cloud = cloud;

    public void Read(string path) => _cloud.Download(path);
}

var local = new LocalFileSystem();
var cloud = new CloudFileSystemAdapter(new CloudFileSystem());
local.Read("local.txt");
cloud.Read("cloud.txt");
```

- **Use Case**: Useful for integrating different file systems into a common interface.

55. Bridge Pattern

- **Code Example**: Decouples abstraction from implementation.

```
public interface IAbstraction
{
    IImplementor Implementor { get; set; }
    void Operation();
}

public abstract class Implementor { public abstract void DoSomething(); }

public class ConcreteAbstraction : IAbstraction
{
    public IImplementor Implementor { get; set; }

    public void Operation() => Implementor.DoSomething();
}

public class ConcreteImplementor : Implementor
{
    public override void DoSomething() => Console.WriteLine("Concrete implementor");
}

var abstraction = new ConcreteAbstraction();
abstraction.Implementor = new ConcreteImplementor();
abstraction.Operation();
```

- **Use Case**: Useful for decoupling interface from implementation.

56. Facade Pattern

- **Code Example**: Provides a simplified interface to a complex subsystem.

```
public interface ISubsystem1 { void DoSomething(); }
public interface ISubsystem2 { void DoSomethingElse(); }

public class Subsystem1 : ISubsystem1
{
    public void DoSomething() => Console.WriteLine("Subsystem 1");
}

public class Subsystem2 : ISubsystem2
{
    public void DoSomethingElse() => Console.WriteLine("Subsystem 2");
}

public class Facade
{
    private readonly ISubsystem1 _sub1;
    private readonly ISubsystem2 _sub2;

    public Facade(ISubsystem1 sub1, ISubsystem2 sub2)
    {
        _sub1 = sub1;
        _sub2 = sub2;
    }

    public void PerformOperation()
    {
        _sub1.DoSomething();
        _sub2.DoSomethingElse();
    }
}

var facade = new Facade(new Subsystem1(), new Subsystem2());
facade.PerformOperation();
```

- **Use Case**: Useful for simplifying access to complex subsystems.

57. Proxy Pattern for Remote Objects

- **Code Example**: Proxies a remote service.

```
public interface IService { void DoWork(); }

public class RemoteService : IService
{
    public void DoWork() => Console.WriteLine("Remote service");
}

public class ServiceProxy : IService
{
    private readonly IService _realService;

    public ServiceProxy(IService realService) => _realService = realService;

    public void DoWork() => _realService.DoWork();
}

var service = new RemoteService();
var proxy = new ServiceProxy(service);
proxy.DoWork();
```

- **Use Case**: Useful for accessing remote services locally.

58. Flyweight Pattern

- **Code Example**: Minimizes memory usage by sharing common object parts.

```
public class FlyweightFactory
{
    private readonly Dictionary<string, Flyweight> _flyweights = new();
```

```
    public Flyweight GetFlyweight(string key)
    {
        if (!_flyweights.ContainsKey(key))
            _flyweights[key] = new Flyweight();
        return _flyweights[key];
    }
}

public class Flyweight { }

var factory = new FlyweightFactory();
var fly1 = factory.GetFlyweight("A");
var fly2 = factory.GetFlyweight("A");
Console.WriteLine(fly1 == fly2); // True
```

- **Use Case**: Useful for managing large numbers of small objects efficiently.

59. State Pattern

- **Code Example**: Encapsulates state-specific behavior.

```
public interface IState { void Handle(); }

public class StateA : IState
{
    public void Handle() => Console.WriteLine("State A");
}

public class Context
{
    private IState _state;

    public void SetState(IState state) => _state = state;
    public void Request() => _state.Handle();
}

var context = new Context();
context.SetState(new StateA());
context.Request();
```

- **Use Case**: Useful for modeling state transitions in objects.

60. Strategy Pattern

- **Code Example**: Defines a family of algorithms and makes them interchangeable.

```
public interface IStrategy { void Execute(); }

public class StrategyA : IStrategy
{
    public void Execute() => Console.WriteLine("Strategy A");
}

public class Context
{
    private IStrategy _strategy;

    public void SetStrategy(IStrategy strategy) => _strategy = strategy;
    public void Execute() => _strategy.Execute();
}

var context = new Context();
context.SetStrategy(new StrategyA());
context.Execute();
```

- **Use Case**: Useful for selecting algorithms at runtime.

61. Observer Pattern with Events

- **Code Example**: Uses events to notify observers.

```
public class Subject
{
    public event EventHandler Changed;
```

```
    protected virtual void OnChanged() => Changed?.Invoke(this, EventArgs.Empty);
}

public class Observer
{
    private readonly Subject _subject;

    public Observer(Subject subject)
    {
        _subject = subject;
        _subject.Changed += HandleChanged;
    }

    private void HandleChanged(object sender, EventArgs e)
        => Console.WriteLine("Observer notified");
}

var subject = new Subject();
var observer = new Observer(subject);
subject.OnChanged();
```

- **Use Case**: Useful for real-time updates and event-driven systems.

62. Chain of Responsibility Pattern

- **Code Example**: Routes requests through a chain of handlers.

```
public interface IHandler { IHandler SetNext(IHandler handler); bool HandleRequest(int request); }

public class ConcreteHandler : IHandler
{
    private IHandler _next;

    public IHandler SetNext(IHandler handler) => _next = handler;
    public bool HandleRequest(int request)
    {
        if (request == 42) return true;
        if (_next != null) return _next.HandleRequest(request);
        return false;
    }
}

var handler1 = new ConcreteHandler();
var handler2 = new ConcreteHandler();
handler1.SetNext(handler2);

Console.WriteLine(handler1.HandleRequest(42)); // True
Console.WriteLine(handler1.HandleRequest(5)); // False
```

- **Use Case**: Useful for processing requests through multiple handlers.

63. Command Pattern with Menu Items

- **Code Example**: Implements a command hierarchy.

```
public interface ICommand { void Execute(); }

public class OpenCommand : ICommand
{
    public void Execute() => Console.WriteLine("Open command");
}

public class MenuItem
{
    private readonly ICommand _command;

    public MenuItem(ICommand command) => _command = command;
    public void Click() => _command.Execute();
}

var command = new OpenCommand();
var menuItem = new MenuItem(command);
menuItem.Click();
```

- **Use Case**: Useful for encapsulating actions in a menu-driven application.

64. Mediator Pattern

- **Code Example**: Coordinates communication between objects.

```
public interface IMediator { void Send(string message, string sender); }

public class Colleague
{
    private readonly IMediator _mediator;
    private string _name;

    public Colleague(IMediator mediator, string name)
    {
        _mediator = mediator;
        _name = name;
    }

    public void SendMessage(string message)
        => _mediator.Send(message, _name);
}

public class ChatMediator : IMediator
{
    public void Send(string message, string sender)
        => Console.WriteLine($"{sender}: {message}");
}

var mediator = new ChatMediator();
var colleague1 = new Colleague(mediator, "Alice");
var colleague2 = new Colleague(mediator, "Bob");

colleague1.SendMessage("Hello");
colleague2.SendMessage("Hi");
```

- **Use Case**: Useful for managing communication between multiple objects.

65. Iterator Pattern

- **Code Example**: Iterates over elements of a collection.

```
public interface IIterator { bool HasNext(); object Next(); }

public class ConcreteIterator : IIterator
{
    private readonly List<object> _list;
    private int _index;

    public ConcreteIterator(List<object> list) { _list = list; }

    public bool HasNext() => (_index < _list.Count);
    public object Next() => _list[_index++];
}

var list = new List<object> { 1, 2, 3 };
var iterator = new ConcreteIterator(list);
while (iterator.HasNext())
    Console.WriteLine(iterator.Next());
```

- **Use Case**: Useful for traversing collections without exposing underlying structure.

66. Template Method Pattern

- **Code Example**: Defines a skeleton of operations.

```
public abstract class TemplateMethod
{
    public void Template()
    {
        Step1();
        Step2();
    }
```

```
    protected abstract void Step1();
    protected abstract void Step2();
}

public class ConcreteTemplate : TemplateMethod
{
    protected override void Step1() => Console.WriteLine("Concrete step 1");
    protected override void Step2() => Console.WriteLine("Concrete step 2");
}

var template = new ConcreteTemplate();
template.Template();
```

- **Use Case**: Useful for defining workflows with reusable steps.

67. Composite Pattern with Recursive Structures

- **Code Example**: Composes objects into tree structures.

```
public interface IComponent { void Display(); }

public class Leaf : IComponent
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine(Name);
}

public class Composite : IComponent
{
    private readonly List<IComponent> _children = new();

    public void Display()
    {
        foreach (var child in _children)
            child.Display();
    }

    public void Add(IComponent component) => _children.Add(component);
}

var composite = new Composite();
composite.Add(new Leaf { Name = "Item 1" });
composite.Display();
```

- **Use Case**: Useful for managing tree structures.

68. Decorator Pattern with Composite

- **Code Example**: Combines decorator and composite patterns.

```
public interface IComponent { void Operation(); }

public class Leaf : IComponent
{
    public void Operation() => Console.WriteLine("Leaf operation");
}

public class Composite : IComponent
{
    private readonly List<IComponent> _children = new();

    public void Operation()
    {
        foreach (var child in _children)
            child.Operation();
    }

    public void Add(IComponent component) => _children.Add(component);
}

public class Decorator : IComponent
{
    private readonly IComponent _component;

    public Decorator(IComponent component) => _component = component;
```

```
    public virtual void Operation() => _component.Operation();
}

var composite = new Composite();
composite.Add(new Leaf());
composite.Add(new Decorator(new Leaf()));
composite.Operation();
```

- **Use Case**: Useful for decorating components within a composite structure.

69. State Pattern with Finite State Machine

- **Code Example**: Implements a finite state machine.

```
public interface IState { void Handle(); }

public class StartState : IState
{
    public void Handle() => Console.WriteLine("Starting");
}

public class StopState : IState
{
    public void Handle() => Console.WriteLine("Stopping");
}

public class Context
{
    private IState _state;

    public void SetState(IState state) => _state = state;
    public void Request() => _state.Handle();
}

var context = new Context();
context.SetState(new StartState());
context.Request();
context.SetState(new StopState());
context.Request();
```

- **Use Case**: Useful for modeling state transitions in complex systems.

70. Strategy Pattern with Sorting Algorithms

- **Code Example**: Implements different sorting strategies.

```
public interface ISortStrategy { void Sort(List<int> list); }

public class QuickSort : ISortStrategy
{
    public void Sort(List<int> list) => Console.WriteLine("Quick sort");
}

public class MergeSort : ISortStrategy
{
    public void Sort(List<int> list) => Console.WriteLine("Merge sort");
}

public class Context
{
    private ISortStrategy _strategy;

    public void SetStrategy(ISortStrategy strategy) => _strategy = strategy;
    public void Sort(List<int> list) => _strategy.Sort(list);
}

var context = new Context();
context.SetStrategy(new QuickSort());
context.Sort(new List<int>());
```

- **Use Case**: Useful for selecting algorithms at runtime.

71. Observer Pattern with GUI Updates

- **Code Example**: Observes changes in data to update UI.

```
public interface ISubject { void Attach(IObserver observer); void Detach(IObserver observer); void
Notify(); }

public interface IObserver { void Update(); }

public class DataSubject : ISubject
{
    private readonly List<IObserver> _observers = new();

    public void Attach(IObserver observer) => _observers.Add(observer);
    public void Detach(IObserver observer) => _observers.Remove(observer);
    public void Notify() => foreach (var observer in _observers) observer.Update();
}

public class GUIObserver : IObserver
{
    public void Update() => Console.WriteLine("GUI updated");
}

var subject = new DataSubject();
var observer1 = new GUIObserver();
var observer2 = new GUIObserver();

subject.Attach(observer1);
subject.Attach(observer2);
subject.Notify();
```

- **Use Case**: Useful for updating GUI components on data changes.

72. Command Pattern with Macro Recording

- **Code Example**: Records a series of commands to replay later.

```
public interface ICommand { void Execute(); }

public class Macro : ICommand
{
    private readonly List<ICommand> _commands = new();

    public void Add(ICommand command) => _commands.Add(command);
    public void Execute() => foreach (var cmd in _commands) cmd.Execute();
}

public class OpenCommand : ICommand
{
    public void Execute() => Console.WriteLine("Open");
}

var macro = new Macro();
macro.Add(new OpenCommand());
macro.Execute();
```

- **Use Case**: Useful for recording and replaying actions.

73. Chain of Responsibility Pattern with Exception Handling

- **Code Example**: Routes exceptions through handlers.

```
public interface IHandler { bool Handle(Exception e); }

public class ConcreteHandler : IHandler
{
    private IHandler _next;

    public bool Handle(Exception e)
    {
        if (e is NullReferenceException) return true;
        if (_next != null) return _next.Handle(e);
        return false;
    }
}

var handler1 = new ConcreteHandler();
```

```
var handler2 = new ConcreteHandler();
handler1._next = handler2;

Console.WriteLine(handler1.Handle(new NullReferenceException())); // True
```

- **Use Case**: Useful for handling exceptions through a chain of handlers.

74. Bridge Pattern with GUI Themes

- **Code Example**: Decouples GUI themes from their implementations.

```
public interface IAbstraction { void Operation(); }

public class ThemeAbstraction : IAbstraction
{
    private readonly IImplementor _implementor;

    public ThemeAbstraction(IImplementor implementor) => _implementor = implementor;
    public void Operation() => _implementor.DoSomething();
}

public interface IImplementor { void DoSomething(); }

public class DarkTheme : IImplementor
{
    public void DoSomething() => Console.WriteLine("Dark theme");
}

var abstraction = new ThemeAbstraction(new DarkTheme());
abstraction.Operation();
```

- **Use Case**: Useful for separating GUI themes from their implementations.

75. Decorator Pattern with UI Components

- **Code Example**: Decorates UI components with additional functionality.

```
public interface IComponent { void Display(); }

public class Button : IComponent
{
    public void Display() => Console.WriteLine("Button");
}

public class Decorator : IComponent
{
    private readonly IComponent _component;

    public Decorator(IComponent component) => _component = component;
    public virtual void Display() => _component.Display();
}

public class ThemedDecorator : Decorator
{
    private string _theme;

    public ThemedDecorator(IComponent component, string theme) : base(component)
    {
        _theme = theme;
    }

    public override void Display()
    {
        base.Display();
        Console.WriteLine(" with theme: " + _theme);
    }
}

var button = new Button();
var decorator = new ThemedDecorator(button, "Dark");
decorator.Display();
```

- **Use Case**: Useful for adding themes to UI components dynamically.

76. Proxy Pattern with Caching

- **Code Example**: Caches expensive operations.

```
public interface IService { string GetData(); }

public class RealService : IService
{
    public string GetData()
    {
        Console.WriteLine("Fetched data");
        return "Data";
    }
}

public class CachingProxy : IService
{
    private readonly IService _realService;
    private string _cache;

    public CachingProxy(IService realService) => _realService = realService;
    public string GetData()
    {
        if (_cache == null) _cache = _realService.GetData();
        return _cache;
    }
}

var service = new RealService();
var proxy = new CachingProxy(service);
Console.WriteLine(proxy.GetData());
```

- **Use Case**: Useful for caching expensive data retrieval operations.

77. Flyweight Pattern with Fonts

- **Code Example**: Shares font objects to save memory.

```
public class FlyweightFactory
{
    private readonly Dictionary<string, FontFlyweight> _flyweights = new();

    public FontFlyweight GetFont(string family) => _flyweights.TryGetValue(family, out var flyweight)
        ? flyweight
        : new FontFlyweight(family);

    private class FontFlyweight
    {
        public string Family { get; }

        public FontFlyweight(string family) => Family = family;
    }
}

var factory = new FlyweightFactory();
var font1 = factory.GetFont("Arial");
var font2 = factory.GetFont("Arial");
Console.WriteLine(font1 == font2); // True
```

- **Use Case**: Useful for managing many small objects efficiently.

78. State Pattern with Network Connections

- **Code Example**: Models network connection states.

```
public interface IState { void Connect(); }

public class Disconnected : IState
{
    public void Connect() => Console.WriteLine("Connecting...");
}

public class Connected : IState
{
```

```
    public void Connect() => Console.WriteLine("Already connected");
}

public class NetworkConnection
{
    private IState _state;

    public NetworkConnection() => _state = new Disconnected();
    public void RequestConnect() => _state.Connect();
}

var connection = new NetworkConnection();
connection.RequestConnect();
```

- **Use Case**: Useful for modeling state transitions in connections.

79. Strategy Pattern with Payment Methods

- **Code Example**: Implements different payment strategies.

```
public interface IPaymentStrategy { void Pay(decimal amount); }

public class CreditCard : IPaymentStrategy
{
    public void Pay(decimal amount) => Console.WriteLine("Paid via credit card: " + amount);
}

public class PayPal : IPaymentStrategy
{
    public void Pay(decimal amount) => Console.WriteLine("Paid via PayPal: " + amount);
}

public class Context
{
    private IPaymentStrategy _strategy;

    public void SetPaymentMethod(IPaymentStrategy strategy) => _strategy = strategy;
    public void ProcessPayment(decimal amount) => _strategy.Pay(amount);
}

var context = new Context();
context.SetPaymentMethod(new CreditCard());
context.ProcessPayment(100);
```

- **Use Case**: Useful for selecting payment methods dynamically.

80. Template Method Pattern with Document Processing

- **Code Example**: Defines a document processing workflow.

```
public abstract class DocumentProcessor
{
    public void ProcessDocument()
    {
        LoadDocument();
        ValidateDocument();
        SaveDocument();
    }

    protected abstract void LoadDocument();
    protected abstract void ValidateDocument();
    protected abstract void SaveDocument();
}

public class PDFProcessor : DocumentProcessor
{
    protected override void LoadDocument() => Console.WriteLine("Loading PDF");
    protected override void ValidateDocument() => Console.WriteLine("Validating PDF");
    protected override void SaveDocument() => Console.WriteLine("Saving PDF");
}

var processor = new PDFProcessor();
processor.ProcessDocument();
```

- **Use Case**: Useful for defining processing workflows.

81. Composite Pattern with File Systems

- **Code Example**: Composes files and directories.

```
public interface IFileSystemItem { void Display(); }

public class File : IFileSystemItem
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine("File: " + Name);
}

public class Directory : IFileSystemItem
{
    private readonly List<IFileSystemItem> _items = new();

    public void Display()
    {
        Console.WriteLine("Directory");
        foreach (var item in _items)
            item.Display();
    }

    public void Add(IFileSystemItem item) => _items.Add(item);
}

var directory = new Directory();
directory.Add(new File { Name = "File1" });
directory.Display();
```

- **Use Case**: Useful for managing file system structures.

82. Decorator Pattern with Logging and Security

- **Code Example**: Combines logging and security decorators.

```
public interface IComponent { void Operation(); }

public class ConcreteComponent : IComponent
{
    public void Operation() => Console.WriteLine("Concrete operation");
}

public class LoggingDecorator : IComponent
{
    private readonly IComponent _component;

    public LoggingDecorator(IComponent component) => _component = component;
    public void Operation()
    {
        Console.WriteLine("Log start");
        _component.Operation();
        Console.WriteLine("Log end");
    }
}

public class SecurityDecorator : IComponent
{
    private readonly IComponent _component;

    public SecurityDecorator(IComponent component) => _component = component;
    public void Operation()
    {
        Console.WriteLine("Security check");
        _component.Operation();
    }
}

var component = new ConcreteComponent();
component = new LoggingDecorator(component);
component = new SecurityDecorator(component);
component.Operation();
```

- **Use Case**: Useful for adding multiple responsibilities.

83. Bridge Pattern with GUI and Themes

- **Code Example**: Bridges GUI components with themes.

```
public interface IGUIComponent { void Render(); }

public abstract class Theme : IGUIComponent
{
    protected IGUIComponent _component;

    public void SetComponent(IGUIComponent component) => _component = component;
    public abstract void Render();
}

public class Button { public override string ToString() => "Button"; }

public class DarkTheme : Theme
{
    public override void Render() => Console.WriteLine("Rendering " + _component.ToString() + " with dark
theme");
}

var button = new Button();
var theme = new DarkTheme();
theme.SetComponent(button);
theme.Render();
```

- **Use Case**: Useful for separating themes from components.

84. Observer Pattern with Stock Updates

- **Code Example**: Observes stock price changes.

```
public interface ISubject { void Attach(IObserver observer); void Detach(IObserver observer); void
Notify(); }

public interface IObserver { void Update(double price); }

public class StockSubject : ISubject
{
    private readonly List<IObserver> _observers = new();
    private double _price;

    public void Attach(IObserver observer) => _observers.Add(observer);
    public void Detach(IObserver observer) => _observers.Remove(observer);
    public void Notify() => foreach (var observer in _observers) observer.Update(_price);
    public void SetPrice(double price)
    {
        _price = price;
        Notify();
    }
}

public class StockObserver : IObserver
{
    public void Update(double price) => Console.WriteLine("Stock updated to: " + price);
}

var subject = new StockSubject();
var observer1 = new StockObserver();
var observer2 = new StockObserver();

subject.Attach(observer1);
subject.Attach(observer2);

subject.SetPrice(100.5);
```

- **Use Case**: Useful for real-time stock updates.

85. Chain of Responsibility Pattern with Authorization

- **Code Example**: Routes authorization requests through roles.

```
public interface IHandler { bool Handle(string user, string permission); }
```

```
public class RoleHandler : IHandler
{
    private IHandler _next;
    private string _role;

    public RoleHandler(string role) => _role = role;

    public bool Handle(string user, string permission)
    {
        if (user == _role) return true;
        return _next?.Handle(user, permission) ?? false;
    }

    public IHandler SetNext(IHandler handler)
    {
        _next = handler;
        return this;
    }
}

var adminHandler = new RoleHandler("Admin");
var userHandler = new RoleHandler("User").SetNext(adminHandler);

Console.WriteLine(userHandler.Handle("Admin", "Write")); // True
Console.WriteLine(userHandler.Handle("User", "Read")); // True
```

- **Use Case**: Useful for authorization through role handling.

86. Command Pattern with Macro Recording

- **Code Example**: Records commands to replay later.

```
public interface ICommand { void Execute(); }

public class Macro : ICommand
{
    private readonly List<ICommand> _commands = new();

    public void Add(ICommand command) => _commands.Add(command);
    public void Execute() => foreach (var cmd in _commands) cmd.Execute();
}

public class OpenCommand : ICommand
{
    public void Execute() => Console.WriteLine("Open");
}

public class SaveCommand : ICommand
{
    public void Execute() => Console.WriteLine("Save");
}

var macro = new Macro();
macro.Add(new OpenCommand());
macro.Add(new SaveCommand());
macro.Execute();
```

- **Use Case**: Useful for recording and replaying sequences of actions.

87. Mediator Pattern with Chat Room

- **Code Example**: Coordinates messages in a chat room.

```
public interface IMediator { voidSendMessage(string message, string sender); }

public class ChatMediator : IMediator
{
    private readonly Dictionary<string, Participant> _participants = new();

    public void SendMessage(string message, string sender)
    {
        if (!_participants.ContainsKey(sender)) return;
        foreach (var participant in _participants.Values)
            if (participant.Name != sender) participant.Receive(message);
```

```
    }

    public void AddParticipant(Participant participant)
        => _participants[participant.Name] = participant;
}

public class Participant
{
    public string Name { get; set; }
    private IMediator _mediator;

    public Participant(IMediator mediator, string name)
    {
        _mediator = mediator;
        Name = name;
        mediator.AddParticipant(this);
    }

    public void SendMessage(string message) => _mediator.SendMessage(message, Name);
    public void Receive(string message) => Console.WriteLine($"{Name} received: {message}");
}

var mediator = new ChatMediator();
var alice = new Participant(mediator, "Alice");
var bob = new Participant(mediator, "Bob");

alice.SendMessage("Hello");
```

- **Use Case**: Useful for managing chat room communication.

88. Iterator Pattern with Custom Collection

- **Code Example**: Implements custom iteration.

```
public interface IIterator { bool HasNext(); object Next(); }

public class CustomCollection
{
    private readonly List<object> _items = new();

    public void Add(object item) => _items.Add(item);
    public IIterator CreateIterator() => new CustomIterator(_items);

    private class CustomIterator : IIterator
    {
        private readonly List<object> _list;
        private int _index;

        public CustomIterator(List<object> list) { _list = list; }

        public bool HasNext() => (_index < _list.Count);
        public object Next() => _list[_index++];
    }
}

var collection = new CustomCollection();
collection.Add(1);
collection.Add(2);

var iterator = collection.CreateIterator();
while (iterator.HasNext())
    Console.WriteLine(iterator.Next());
```

- **Use Case**: Useful for custom collection iteration.

89. Template Method Pattern with Reporting

- **Code Example**: Generates reports using a template.

```
public abstract class ReportGenerator
{
    public void GenerateReport()
    {
        Header();
        Body();
```

```
        Footer();
    }

    protected abstract void Header();
    protected abstract void Body();
    protected abstract void Footer();
}

public class SalesReport : ReportGenerator
{
    protected override void Header() => Console.WriteLine("Sales Report");
    protected override void Body() => Console.WriteLine("Sales data");
    protected override void Footer() => Console.WriteLine("End of report");
}

var generator = new SalesReport();
generator.GenerateReport();
```

- **Use Case**: Useful for generating reports with consistent structure.

90. Composite Pattern with Employee Hierarchy

- **Code Example**: Composes employees into a hierarchy.

```
public interface IEmployee { void Display(); }

public class Employee : IEmployee
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine(Name);
}

public class Department : IEmployee
{
    private readonly List<IEmployee> _employees = new();

    public void Display()
    {
        Console.WriteLine("Department");
        foreach (var employee in _employees)
            employee.Display();
    }

    public void Add(IEmployee employee) => _employees.Add(employee);
}

var department = new Department();
department.Add(new Employee { Name = "Alice" });
department.Display();
```

- **Use Case**: Useful for managing organizational structures.

91. Decorator Pattern with GUI Components

- **Code Example**: Enhances UI components with themes.

```
public interface IComponent { void Draw(); }

public class Button : IComponent
{
    public void Draw() => Console.WriteLine("Button");
}

public class ThemeDecorator : IComponent
{
    private readonly IComponent _component;
    private string _theme;

    public ThemeDecorator(IComponent component, string theme)
    {
        _component = component;
        _theme = theme;
    }

    public void Draw()
```

```
    {
        Console.WriteLine("Theming with: " + _theme);
        _component.Draw();
    }
}

var button = new Button();
var themedButton = new ThemeDecorator(button, "Dark");
themedButton.Draw();
```

- **Use Case**: Useful for theming UI components.

92. Proxy Pattern with Logging

- **Code Example**: Logs method calls on a service.

```
public interface IService { void DoWork(); }

public class RealService : IService
{
    public void DoWork() => Console.WriteLine("Doing work");
}

public class LoggingProxy : IService
{
    private readonly IService _realService;

    public LoggingProxy(IService realService) => _realService = realService;
    public void DoWork()
    {
        Console.WriteLine("Logging: Work started");
        _realService.DoWork();
        Console.WriteLine("Logging: Work ended");
    }
}

var service = new RealService();
var proxy = new LoggingProxy(service);
proxy.DoWork();
```

- **Use Case**: Useful for logging service method calls.

93. Flyweight Pattern with Character Rendering

- **Code Example**: Shares character rendering information.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        private char _char;

        public CharacterFlyweight(char c) => _char = c;
        public void Render() => Console.WriteLine(_char);
    }
}

var factory = new CharacterFlyweightFactory();
factory.GetCharacter('A').Render();
```

- **Use Case**: Useful for rendering characters efficiently.

94. State Pattern with Network Connection States

- **Code Example**: Manages connection states.

```
public interface IConnectionState { void Connect(); }

public class Disconnected : IConnectionState
```

```
{
    public void Connect() => Console.WriteLine("Connecting...");
}

public class Connected : IConnectionState
{
    public void Connect() => Console.WriteLine("Already connected");
}

public class NetworkConnection
{
    private IConnectionState _state;

    public NetworkConnection() => _state = new Disconnected();
    public void RequestConnect() => _state.Connect();
}

var connection = new NetworkConnection();
connection.RequestConnect();
```

- **Use Case**: Useful for managing network connections.

95. Strategy Pattern with Sorting

- **Code Example**: Implements different sorting strategies.

```
public interface ISortStrategy { void Sort(List<int> list); }

public class BubbleSort : ISortStrategy
{
    public void Sort(List<int> list)
    {
        for (int i = 0; i < list.Count; i++)
            for (int j = 0; j < list.Count - i - 1; j++)
                if (list[j] > list[j + 1]) Swap(list, j, j + 1);
    }

    private void Swap(List<int> list, int a, int b)
    {
        var temp = list[a];
        list[a] = list[b];
        list[b] = temp;
    }
}

public class QuickSort : ISortStrategy
{
    public void Sort(List<int> list) => QuickSortHelper(list, 0, list.Count - 1);

    private void QuickSortHelper(List<int> list, int low, int high)
    {
        if (low < high)
        {
            var pivot = Partition(list, low, high);
            QuickSortHelper(list, low, pivot - 1);
            QuickSortHelper(list, pivot + 1, high);
        }
    }

    private int Partition(List<int> list, int low, int high)
    {
        var pivotValue = list[high];
        var i = low - 1;
        for (var j = low; j < high; j++)
            if (list[j] <= pivotValue) Swap(list, ++i, j);
        Swap(list, i + 1, high);
        return i + 1;
    }

    private void Swap(List<int> list, int a, int b)
    {
        var temp = list[a];
        list[a] = list[b];
        list[b] = temp;
    }
}
```

```
var context = new Context();
context.SetStrategy(new BubbleSort());
var list = new List<int> { 3, 1, 4, 2 };
context.Sort(list);
```

- **Use Case**: Useful for selecting sorting algorithms dynamically.

96. Template Method Pattern with Game Framework

- **Code Example**: Defines a game loop.

```
public abstract class Game
{
    protected abstract void Initialize();
    protected abstract void LoadLevel();
    protected abstract void Update(float deltaTime);
    protected abstract void CleanUp();

    public void Run()
    {
        Initialize();
        LoadLevel();
        while (true) Update(0.1f);
        CleanUp();
    }
}

public class Platformer : Game
{
    protected override void Initialize() => Console.WriteLine("Initializing platformer");
    protected override void LoadLevel() => Console.WriteLine("Loading level");
    protected override void Update(float deltaTime) => Console.WriteLine("Updating game state");
    protected override void CleanUp() => Console.WriteLine("Cleaning up");
}

var game = new Platformer();
game.Run();
```

- **Use Case**: Useful for creating game frameworks.

97. Composite Pattern with Inventory Management

- **Code Example**: Composes inventory items.

```
public interface IInventoryItem { void Display(); }

public class Item : IInventoryItem
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine(Name);
}

public class Inventory : IInventoryItem
{
    private readonly List<IInventoryItem> _items = new();

    public void Display()
    {
        Console.WriteLine("Inventory");
        foreach (var item in _items) item.Display();
    }

    public void Add(IInventoryItem item) => _items.Add(item);
}

var inventory = new Inventory();
inventory.Add(new Item { Name = "Sword" });
inventory.Display();
```

- **Use Case**: Useful for managing inventories.

98. Decorator Pattern with Logging and Authentication

- **Code Example**: Decorates service calls with logging and auth.

```
public interface IService { string GetData(); }

public class RealService : IService
{
    public string GetData() => "Data";
}

public class LoggingDecorator : IService
{
    private readonly IService _service;

    public LoggingDecorator(IService service) => _service = service;
    public string GetData()
    {
        Console.WriteLine("Logging: Data requested");
        return _service.GetData();
    }
}

public class AuthDecorator : IService
{
    private readonly IService _service;
    public AuthDecorator(IService service) => _service = service;
    public string GetData()
    {
        Console.WriteLine("Auth: User authenticated");
        return _service.GetData();
    }
}

var service = new RealService();
service = new LoggingDecorator(service);
service = new AuthDecorator(service);

Console.WriteLine(service.GetData());
```

- **Use Case**: Useful for adding logging and authentication to services.

99. Bridge Pattern with Cars and Engines

- **Code Example**: Bridges car brands with engine types.

```
public interface ICar { void Drive(); }

public class Toyota : ICar
{
    private readonly IEngine _engine;
    public Toyota(IEngine engine) => _engine = engine;
    public void Drive() => Console.WriteLine("Toyota driving with " + _engine.GetType().Name);
}

public class Honda : ICar
{
    private readonly IEngine _engine;

    public Honda(IEngine engine) => _engine = engine;
    public void Drive() => Console.WriteLine("Honda driving with " + _engine.GetType().Name);
}

public interface IEngine { }

public class V8 : IEngine { }
public class Turbo : IEngine { }

var toyota = new Toyota(new V8());
var honda = new Honda(new Turbo());

toyota.Drive();
honda.Drive();
```

- **Use Case**: Useful for decoupling car brands from engine types.

100. Observer Pattern with Weather Station

- **Code Example**: Observes weather data changes.

```
public interface IWeatherSubject { void Attach(IObserver observer); void Detach(IObserver observer); void
Notify(); }

public interface IObserver { void Update(double temp, double humidity); }

public class WeatherStation : IWeatherSubject
{
    private readonly List<IObserver> _observers = new();
    private double _temp;
    private double _humidity;

    public void Attach(IObserver observer) => _observers.Add(observer);
    public void Detach(IObserver observer) => _observers.Remove(observer);
    public void Notify() => foreach (var observer in _observers) observer.Update(_temp, _humidity);
    public void SetMeasurements(double temp, double humidity)
    {
        _temp = temp;
        _humidity = humidity;
        Notify();
    }
}

public class TemperatureDisplay : IObserver
{
    public void Update(double temp, double humidity)
        => Console.WriteLine("Temperature: " + temp);
}

public class HumidityDisplay : IObserver
{
    public void Update(double temp, double humidity)
        => Console.WriteLine("Humidity: " + humidity);
}

var weatherStation = new WeatherStation();
var tempDisplay = new TemperatureDisplay();
var humidityDisplay = new HumidityDisplay();

weatherStation.Attach(tempDisplay);
weatherStation.Attach(humidityDisplay);

weatherStation.SetMeasurements(25, 60);
```

- **Use Case**: Useful for real-time weather data monitoring.

101. Chain of Responsibility Pattern with Loan Approval

- **Code Example**: Approves loans through a hierarchy.

```
public interface IApprover { bool Approve(int amount); }

public class Manager : IApprover
{
    private IApprover _next;
    private int _maxLimit = 1000;

    public bool Approve(int amount)
    {
        if (amount <= _maxLimit) return true;
        return _next?.Approve(amount) ?? false;
    }

    public IApprover SetNext(IApprover approver)
    {
        _next = approver;
        return this;
    }
}

public class Director : IApprover
{
    private IApprover _next;
```

```
    private int _maxLimit = 5000;

    public bool Approve(int amount)
    {
        if (amount <= _maxLimit) return true;
        return _next?.Approve(amount) ?? false;
    }

    public IApprover SetNext(IApprover approver)
    {
        _next = approver;
        return this;
    }
}

var manager = new Manager();
var director = new Director();

manager.SetNext(director);

Console.WriteLine(manager.Approve(800)); // Manager approves
Console.WriteLine(manager.Approve(3000)); // Director approves
```

- **Use Case**: Useful for hierarchical approval processes.

102. Command Pattern with Toggle Switch

- **Code Example**: Implements toggle functionality.

```
public interface ICommand { void Execute(); }

public class ToggleCommand : ICommand
{
    private IDevice _device;
    private bool _state;

    public ToggleCommand(IDevice device) => _device = device;

    public void Execute()
    {
        _state = !_state;
        if (_state) _device.TurnOn();
        else _device.TurnOff();
    }
}

public interface IDevice { void TurnOn(); void TurnOff(); }

public class Light : IDevice
{
    public void TurnOn() => Console.WriteLine("Light on");
    public void TurnOff() => Console.WriteLine("Light off");
}

var light = new Light();
var command = new ToggleCommand(light);

command.Execute(); // on
command.Execute(); // off
```

- **Use Case**: Useful for implementing toggle switches.

103. Mediator Pattern with Chat System

- **Code Example**: Coordinates chat messages between users.

```
public interface IMediator { voidSendMessage(string message, string sender); }

public class ChatMediator : IMediator
{
    private readonly Dictionary<string, Participant> _participants = new();

    public void SendMessage(string message, string sender)
    {
        if (!_participants.ContainsKey(sender)) return;
```

```
        foreach (var participant in _participants.Values)
            if (participant.Name != sender) participant.Receive(message);
    }

    public void AddParticipant(Participant participant)
        => _participants[participant.Name] = participant;
}

public class Participant
{
    public string Name { get; set; }
    private IMediator _mediator;

    public Participant(IMediator mediator, string name)
    {
        _mediator = mediator;
        Name = name;
        mediator.AddParticipant(this);
    }

    public void SendMessage(string message) => _mediator.SendMessage(message, Name);
    public void Receive(string message) => Console.WriteLine($"{Name} received: {message}");
}

var mediator = new ChatMediator();
var alice = new Participant(mediator, "Alice");
var bob = new Participant(mediator, "Bob");

alice.SendMessage("Hello");
```

- **Use Case**: Useful for chat systems where messages are sent to all participants except the sender.

104. Iterator Pattern with Custom Data Structures

- **Code Example**: Iterates over a custom list.

```
public interface IIterator { bool HasNext(); object Next(); }

public class CustomList<T>
{
    private readonly List<T> _items = new();

    public void Add(T item) => _items.Add(item);
    public IIterator GetEnumerator() => new CustomEnumerator(_items);

    private class CustomEnumerator : IIterator
    {
        private readonly List<T> _list;
        private int _index;

        public CustomEnumerator(List<T> list) { _list = list; }

        public bool HasNext() => (_index < _list.Count);
        public object Next() => _list[_index++];
    }
}

var list = new CustomList<int>();
list.Add(1);
list.Add(2);

var enumerator = list.GetEnumerator();
while (enumerator.HasNext())
    Console.WriteLine(enumerator.Next());
```

- **Use Case**: Useful for custom iteration over data structures.

105. Template Method Pattern with Image Processing

- **Code Example**: Defines image processing steps.

```
public abstract class ImageProcessor
{
    public void ProcessImage()
    {
```

```
        LoadImage();
        ApplyFilters();
        SaveImage();
    }

    protected abstract void LoadImage();
    protected abstract void ApplyFilters();
    protected abstract void SaveImage();
}

public class PhotoProcessor : ImageProcessor
{
    protected override void LoadImage() => Console.WriteLine("Loading photo");
    protected override void ApplyFilters()
    {
        Console.WriteLine("Applying color correction");
        Console.WriteLine("Sharpening image");
    }
    protected override void SaveImage() => Console.WriteLine("Saving as JPEG");
}

var processor = new PhotoProcessor();
processor.ProcessImage();
```

- **Use Case**: Useful for processing images through a consistent workflow.

106. Composite Pattern with Organizational Structure

- **Code Example**: Composes employees into departments.

```
public interface IEmployee { void Display(); }

public class Employee : IEmployee
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine(Name);
}

public class Department : IEmployee
{
    private readonly List<IEmployee> _employees = new();

    public void Display()
    {
        Console.WriteLine("Department");
        foreach (var employee in _employees)
            employee.Display();
    }

    public void Add(IEmployee employee) => _employees.Add(employee);
}

var department = new Department();
department.Add(new Employee { Name = "Alice" });
department.Display();
```

- **Use Case**: Useful for managing organizational structures.

107. Decorator Pattern with Network Connections

- **Code Example**: Enhances network connections with encryption.

```
public interface INetworkConnection { void Connect(); }

public class BasicConnection : INetworkConnection
{
    public void Connect() => Console.WriteLine("Connecting without encryption");
}

public class EncryptedConnection : INetworkConnection
{
    private readonly INetworkConnection _connection;

    public EncryptedConnection(INetworkConnection connection) => _connection = connection;
    public void Connect()
```

```
    {
        Console.WriteLine("Encrypting");
        _connection.Connect();
    }
}

var connection = new BasicConnection();
connection = new EncryptedConnection(connection);

connection.Connect();
```

- **Use Case**: Useful for adding encryption to network connections.

108. Flyweight Pattern with Character Set

- **Code Example**: Shares character rendering information.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');
Console.WriteLine(charA.Equals(charB)); // False
```

- **Use Case**: Useful for efficiently managing a large set of similar objects.

109. State Pattern with Elevator Floors

- **Code Example**: Manages elevator state changes.

```
public interface IElevatorState { void MoveUp(); }

public class GroundFloor : IElevatorState
{
    public void MoveUp() => Console.WriteLine("Going up from ground floor");
}

public class FirstFloor : IElevatorState
{
    public void MoveUp() => Console.WriteLine("Going up to second floor");
}

public class Elevator
{
    private IElevatorState _currentState;

    public Elevator() => _currentState = new GroundFloor();
    public void SetState(IElevatorState state) => _currentState = state;
    public void MoveUp() => _currentState.MoveUp();
}

var elevator = new Elevator();
elevator.SetState(new FirstFloor());
elevator.MoveUp();
```

- **Use Case**: Useful for managing state transitions in elevators.

110. Strategy Pattern with Authentication Methods

- **Code Example**: Implements different authentication strategies.

```
public interface IAuthStrategy { bool Authenticate(string username, string password); }

public class SimpleAuth : IAuthStrategy
{
    public bool Authenticate(string username, string password)
        => username == "user" && password == "pass";
}

public class OAuth : IAuthStrategy
{
    public bool Authenticate(string username, string password) => true; // Simplified OAuth logic
}

public class AuthContext
{
    private IAuthStrategy _strategy;

    public void SetStrategy(IAuthStrategy strategy) => _strategy = strategy;
    public bool Login(string username, string password) => _strategy.Authenticate(username, password);
}

var context = new AuthContext();
context.SetStrategy(new SimpleAuth());
Console.WriteLine(context.Login("user", "pass")); // True
```

- **Use Case**: Useful for selecting authentication methods dynamically.

111. Template Method Pattern with Software Development Life Cycle

- **Code Example**: Defines the SDLC process.

```
public abstract class SDLCProcess
{
    public void Run()
    {
        Plan();
        Develop();
        Test();
        Deploy();
        Maintain();
    }

    protected abstract void Plan();
    protected abstract void Develop();
    protected abstract void Test();
    protected abstract void Deploy();
    protected abstract void Maintain();
}

public class AgileSDLC : SDLCProcess
{
    protected override void Plan() => Console.WriteLine("Planning sprint");
    protected override void Develop() => Console.WriteLine("Developing features");
    protected override void Test() => Console.WriteLine("Testing in sprints");
    protected override void Deploy() => Console.WriteLine("Continuous deployment");
    protected override void Maintain() => Console.WriteLine("Iterative maintenance");
}

var sdlc = new AgileSDLC();
sdlc.Run();
```

- **Use Case**: Useful for defining and executing SDLC processes.

112. Composite Pattern with Menu Items

- **Code Example**: Composes menu items into a hierarchy.

```
public interface IMenuItem { void Display(); }

public class MenuItem : IMenuItem
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine(Name);
}
```

```
public class Menu : IMenuItem
{
    private readonly List<IMenuItem> _items = new();

    public void Display()
    {
        Console.WriteLine("Menu");
        foreach (var item in _items)
            item.Display();
    }

    public void Add(IMenuItem item) => _items.Add(item);
}

var menu = new Menu();
menu.Add(new MenuItem { Name = "Home" });
menu.Display();
```

- **Use Case**: Useful for managing menus and their items.

113. Decorator Pattern with File Compression

- **Code Example**: Decorates files with compression.

```
public interface IFile { void Open(); }

public class BasicFile : IFile
{
    public string Name { get; set; }
    public void Open() => Console.WriteLine("Opening " + Name);
}

public class CompressedFile : IFile
{
    private readonly IFile _file;

    public CompressedFile(IFile file) => _file = file;
    public void Open()
    {
        Console.WriteLine("Decompressing");
        _file.Open();
    }
}

var file = new BasicFile { Name = "data.txt" };
file = new CompressedFile(file);
file.Open();
```

- **Use Case**: Useful for adding compression to files.

114. Proxy Pattern with Image Loading

- **Code Example**: Lazily loads images.

```
public interface IImage { void Display(); }

public class RealImage : IImage
{
    private string _filename;

    public RealImage(string filename)
    {
        _filename = filename;
        LoadFromDisk();
    }

    private void LoadFromDisk() => Console.WriteLine("Loading " + _filename);
    public void Display() => Console.WriteLine("Displaying " + _filename);
}

public class ImageProxy : IImage
{
    private RealImage _realImage;
    private string _filename;
```

```
    public ImageProxy(string filename)
    {
        _filename = filename;
        _realImage = null;
    }

    public void Display()
    {
        if (_realImage == null) _realImage = new RealImage(_filename);
        _realImage.Display();
    }
}

var image = new ImageProxy("image.jpg");
image.Display(); // Loads and displays
image.Display(); // Displays without reloading
```

- **Use Case**: Useful for lazy loading of images to optimize performance.

115. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a large set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage when dealing with a large number of similar objects.

116. State Pattern with ATM States

- **Code Example**: Manages ATM states like idle, processing.

```
public interface IATMState { void InsertCard(); }

public class IdleState : IATMState
{
    public void InsertCard() => Console.WriteLine("Card inserted, moving to processing");
}

public class ProcessingState : IATMState
{
    public void InsertCard() => Console.WriteLine("Already processing, cannot insert another card");
}

public class ATM
{
    private IATMState _currentState;

    public ATM() => _currentState = new IdleState();
    public void SetState(IATMState state) => _currentState = state;
    public void InsertCard() => _currentState.InsertCard();
}

var atm = new ATM();
atm.InsertCard(); // Card inserted
```

- **Use Case**: Useful for managing state transitions in ATMs.

117. Strategy Pattern with Sorting Algorithms

- **Code Example**: Implements different sorting strategies.

```
public interface ISortStrategy { void Sort(List<int> list); }

public class QuickSort : ISortStrategy
{
    public void Sort(List<int> list)
    {
        if (list.Count <= 1) return;
        var pivot = list[list.Count / 2];
        var left = new List<int>();
        var right = new List<int>();

        foreach (var num in list)
            if (num < pivot) left.Add(num);
            else if (num > pivot) right.Add(num);

        Sort(left);
        Sort(right);

        list.Clear();
        left.ForEach(n => list.Add(n));
        list.Add(pivot);
        right.ForEach(n => list.Add(n));
    }
}

public class MergeSort : ISortStrategy
{
    public void Sort(List<int> list) { /* Implementation */ }
}

var context = new SortingContext(new QuickSort());
context.Sort(list);
```

- **Use Case**: Useful for selecting sorting algorithms dynamically.

118. Template Method Pattern with Test Automation

- **Code Example**: Defines a test execution framework.

```
public abstract class TestExecutor
{
    public void RunTests()
    {
        SetupEnvironment();
        ExecuteTests();
        TearDownEnvironment();
    }

    protected abstract void SetupEnvironment();
    protected abstract void ExecuteTests();
    protected abstract void TearDownEnvironment();
}

public class UnitTestExecutor : TestExecutor
{
    protected override void SetupEnvironment() => Console.WriteLine("Setting up unit test environment");
    protected override void ExecuteTests() => Console.WriteLine("Running unit tests");
    protected override void TearDownEnvironment() => Console.WriteLine("Tearing down unit test
environment");
}

var executor = new UnitTestExecutor();
executor.RunTests();
```

- **Use Case**: Useful for automating test execution with consistent setup and teardown.

119. Composite Pattern with File System

- **Code Example**: Composes files and directories.

```
public interface IFileSystemNode { void Display(); }

public class File : IFileSystemNode
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine(Name);
}

public class Directory : IFileSystemNode
{
    private readonly List<IFileSystemNode> _children = new();

    public void Display()
    {
        Console.WriteLine("Directory");
        foreach (var child in _children) child.Display();
    }

    public void Add(IFileSystemNode node) => _children.Add(node);
}

var directory = new Directory();
directory.Add(new File { Name = "file.txt" });
directory.Display();
```

- **Use Case**: Useful for managing file system structures.

120. Decorator Pattern with GUI Components

- **Code Example**: Enhances UI components with additional features.

```
public interface IGUIComponent { void Render(); }

public class Button : IGUIComponent
{
    public override string ToString() => "Button";
    public void Render() => Console.WriteLine(this);
}

public class ThemeDecorator : IGUIComponent
{
    private readonly IGUIComponent _component;
    private string _theme;

    public ThemeDecorator(IGUIComponent component, string theme)
    {
        _component = component;
        _theme = theme;
    }

    public void Render()
    {
        Console.WriteLine("Theming with: " + _theme);
        _component.Render();
    }
}

var button = new Button();
var themedButton = new ThemeDecorator(button, "Dark");
themedButton.Render();
```

- **Use Case**: Useful for theming UI components dynamically.

121. Proxy Pattern with Remote Data Access

- **Code Example**: Provides a local proxy for remote data.

```
public interface IRemoteService { string GetData(); }

public class RemoteService : IRemoteService
{
    public string GetData()
    {
```

```
        // Simulate network call
        Thread.Sleep(1000);
        return "Data from remote";
    }
}

public class RemoteProxy : IRemoteService
{
    private IRemoteService _realService;
    private string _data;

    public RemoteProxy() => _realService = null;

    public string GetData()
    {
        if (_realService == null)
        {
            _realService = new RemoteService();
            _data = _realService.GetData();
        }
        return _data;
    }
}

var proxy = new RemoteProxy();
Console.WriteLine(proxy.GetData()); // First call: remote service accessed
Console.WriteLine(proxy.GetData()); // Subsequent calls: data cached
```

- **Use Case**: Useful for optimizing access to remote services by caching results.

122. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a large number of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage by reusing shared data.

123. State Pattern with Order Processing

- **Code Example**: Manages order states like pending, shipped.

```
public interface IOrderState { void Process(); }

public class PendingState : IOrderState
{
    public void Process() => Console.WriteLine("Processing pending order");
}

public class ShippedState : IOrderState
{
    public void Process() => Console.WriteLine("Order already shipped");
}

public class Order
```

```
{
    private IOrderState _currentState;

    public Order() => _currentState = new PendingState();
    public void SetState(IOrderState state) => _currentState = state;
    public void Process() => _currentState.Process();
}

var order = new Order();
order.Process(); // Pending state processing
```

- **Use Case**: Useful for managing the lifecycle of orders through various states.

124. Strategy Pattern with Payment Methods

- **Code Example**: Implements different payment strategies.

```
public interface IPaymentStrategy { void Pay(decimal amount); }

public class CreditCardPayment : IPaymentStrategy
{
    public void Pay(decimal amount) => Console.WriteLine("Paid via credit card: " + amount);
}

public class PayPalPayment : IPaymentStrategy
{
    public void Pay(decimal amount) => Console.WriteLine("Paid via PayPal: " + amount);
}

public class PaymentContext
{
    private IPaymentStrategy _strategy;

    public void SetStrategy(IPaymentStrategy strategy) => _strategy = strategy;
    public void ProcessPayment(decimal amount) => _strategy.Pay(amount);
}

var context = new PaymentContext();
context.SetStrategy(new CreditCardPayment());
context.ProcessPayment(100);
```

- **Use Case**: Useful for selecting payment methods dynamically.

125. Template Method Pattern with Build Automation

- **Code Example**: Defines a build process.

```
public abstract class BuildProcess
{
    public void RunBuild()
    {
        CheckOutCode();
        Compile();
        Test();
        Deploy();
    }

    protected abstract void CheckOutCode();
    protected abstract void Compile();
    protected abstract void Test();
    protected abstract void Deploy();
}

public class CIPIPE : BuildProcess
{
    protected override void CheckOutCode() => Console.WriteLine("Checking out code from Git");
    protected override void Compile() => Console.WriteLine("Compiling with MSBuild");
    protected override void Test() => Console.WriteLine("Running unit tests");
    protected override void Deploy() => Console.WriteLine("Deploying to Azure");
}

var process = new CIPIPE();
process.RunBuild();
```

- **Use Case**: Useful for automating build processes with consistent steps.

126. Composite Pattern with Tree Structures

- **Code Example**: Composes nodes into a tree.

```
public interface ITreeNode { void Display(); }

public class LeafNode : ITreeNode
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine(Name);
}

public class CompositeNode : ITreeNode
{
    private readonly List<ITreeNode> _children = new();

    public void Display()
    {
        Console.WriteLine("Composite node");
        foreach (var child in _children) child.Display();
    }

    public void Add(ITreeNode node) => _children.Add(node);
}

var composite = new CompositeNode();
composite.Add(new LeafNode { Name = "Leaf 1" });
composite.Display();
```

- **Use Case**: Useful for managing hierarchical tree structures.

127. Decorator Pattern with Network Connections

- **Code Example**: Enhances network connections with additional features.

```
public interface INetworkConnection { void Connect(); }

public class BasicConnection : INetworkConnection
{
    public void Connect() => Console.WriteLine("Connecting without encryption");
}

public class EncryptedConnection : INetworkConnection
{
    private readonly INetworkConnection _connection;

    public EncryptedConnection(INetworkConnection connection) => _connection = connection;
    public void Connect()
    {
        Console.WriteLine("Encrypting");
        _connection.Connect();
    }
}

var connection = new BasicConnection();
connection = new EncryptedConnection(connection);
connection.Connect();
```

- **Use Case**: Useful for adding encryption to network connections dynamically.

128. Proxy Pattern with Image Loading

- **Code Example**: Lazily loads images to optimize performance.

```
public interface IImage { void Display(); }

public class RealImage : IImage
{
    private string _filename;

    public RealImage(string filename)
    {
        _filename = filename;
        LoadFromDisk();
```

```
    }

    private void LoadFromDisk() => Console.WriteLine("Loading " + _filename);
    public void Display() => Console.WriteLine("Displaying " + _filename);
}

public class ImageProxy : IImage
{
    private RealImage _realImage;
    private string _filename;

    public ImageProxy(string filename)
    {
        _filename = filename;
        _realImage = null;
    }

    public void Display()
    {
        if (_realImage == null) _realImage = new RealImage(_filename);
        _realImage.Display();
    }
}

var image = new ImageProxy("image.jpg");
image.Display(); // Loads and displays
image.Display(); // Displays without reloading
```

- **Use Case**: Useful for optimizing resource loading by deferring until necessary.

129. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages characters to save memory.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage by reusing shared character objects.

130. State Pattern with Traffic Lights

- **Code Example**: Manages traffic light state changes.

```
public interface ITrafficLightState { void Change(); }

public class RedState : ITrafficLightState
{
    public void Change() => Console.WriteLine("Turning to green");
}

public class GreenState : ITrafficLightState
{
    public void Change() => Console.WriteLine("Turning to red");
}

public class TrafficLight
```

```
{
    private ITrafficLightState _currentState;

    public TrafficLight() => _currentState = new RedState();
    public void SetState(ITrafficLightState state) => _currentState = state;
    public void Change() => _currentState.Change();
}

var light = new TrafficLight();
light.Change(); // Turns to green
```

- **Use Case**: Useful for managing state transitions in traffic lights or similar systems.

131. Strategy Pattern with Compression Methods

- **Code Example**: Implements different compression strategies.

```
public interface ICompressionStrategy { byte[] Compress(byte[] data); }

public class LZ77 : ICompressionStrategy
{
    public byte[] Compress(byte[] data)
    {
        // Simplified LZ77 compression logic
        return new byte[] { 0x01, 0x02 };
    }
}

public class Deflate : ICompressionStrategy
{
    public byte[] Compress(byte[] data)
    {
        // Simplified Deflate compression logic
        return new byte[] { 0x03, 0x04 };
    }
}

public class CompressionContext
{
    private ICompressionStrategy _strategy;

    public void SetStrategy(ICompressionStrategy strategy) => _strategy = strategy;
    public byte[] Compress(byte[] data) => _strategy.Compress(data);
}

var context = new CompressionContext();
context.SetStrategy(new LZ77());
byte[] result = context.Compress(Encoding.UTF8.GetBytes("Hello"));
```

- **Use Case**: Useful for selecting compression algorithms dynamically based on needs.

132. Template Method Pattern with Web Scraping

- **Code Example**: Defines a web scraping process.

```
public abstract class WebScraper
{
    public void Scrape()
    {
        FetchPage();
        ExtractData();
        SaveResults();
    }

    protected abstract void FetchPage();
    protected abstract void ExtractData();
    protected abstract void SaveResults();
}

public class AmazonScraper : WebScraper
{
    protected override void FetchPage() => Console.WriteLine("Fetching Amazon page");
    protected override void ExtractData() => Console.WriteLine("Extracting product info");
    protected override void SaveResults() => Console.WriteLine("Saving to database");
}
```

```
var scraper = new AmazonScraper();
scraper.Scrape();
```

- **Use Case**: Useful for automating web scraping tasks with consistent steps.

133. Composite Pattern with Music Playlists

- **Code Example**: Composes songs into playlists.

```
public interface IMediaItem { void Play(); }

public class Song : IMediaItem
{
    public string Title { get; set; }
    public void Play() => Console.WriteLine("Playing " + Title);
}

public class Playlist : IMediaItem
{
    private readonly List<IMediaItem> _songs = new();

    public void Play()
    {
        Console.WriteLine("Playing playlist");
        foreach (var song in _songs) song.Play();
    }

    public void Add(IMediaItem song) => _songs.Add(song);
}

var playlist = new Playlist();
playlist.Add(new Song { Title = "Bohemian Rhapsody" });
playlist.Play();
```

- **Use Case**: Useful for managing playlists and their constituent songs.

134. Decorator Pattern with GUI Components

- **Code Example**: Enhances UI components with additional behaviors.

```
public interface IGUIComponent { void Draw(); }

public class Button : IGUIComponent
{
    public string Text { get; set; }
    public void Draw() => Console.WriteLine("Drawing button: " + Text);
}

public class FocusDecorator : IGUIComponent
{
    private readonly IGUIComponent _component;

    public FocusDecorator(IGUIComponent component) => _component = component;
    public void Draw()
    {
        Console.WriteLine("Adding focus");
        _component.Draw();
    }
}

var button = new Button { Text = "Submit" };
button = new FocusDecorator(button);
button.Draw(); // Draws with focus
```

- **Use Case**: Useful for adding features to UI components dynamically.

135. Proxy Pattern with Database Access

- **Code Example**: Provides a proxy for database operations.

```
public interface IDatabase { void Query(string sql); }

public class RealDatabase : IDatabase
```

```
{
    public void Query(string sql)
    {
        // Simulate database query
        Console.WriteLine("Executing query: " + sql);
    }
}

public class DatabaseProxy : IDatabase
{
    private RealDatabase _realDb;
    private string _lastQuery;

    public DatabaseProxy() => _realDb = null;

    public void Query(string sql)
    {
        if (_realDb == null) _realDb = new RealDatabase();
        _lastQuery = sql;
        _realDb.Query(sql);
    }
}

var proxy = new DatabaseProxy();
proxy.Query("SELECT * FROM Users"); // Accesses real database
proxy.Query("SELECT * FROM Orders"); // Uses existing connection
```

- **Use Case**: Useful for optimizing database access by managing connections efficiently.

136. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage when dealing with numerous similar objects.

137. State Pattern with Vending Machine

- **Code Example**: Manages states like idle, processing payment.

```
public interface IVendingMachineState { void InsertCoin(); }

public class IdleState : IVendingMachineState
{
    public void InsertCoin() => Console.WriteLine("Coin inserted, ready to select item");
}

public class SelectingItemState : IVendingMachineState
{
    public void InsertCoin() => Console.WriteLine("Already selecting item");
}

public class VendingMachine
{
```

```
    private IVendingMachineState _currentState;

    public VendingMachine() => _currentState = new IdleState();
    public void SetState(IVendingMachineState state) => _currentState = state;
    public void InsertCoin() => _currentState.InsertCoin();
}

var machine = new VendingMachine();
machine.InsertCoin(); // Enters selecting item state
```

- **Use Case**: Useful for managing the operational states of vending machines.

138. Strategy Pattern with Sorting Algorithms

- **Code Example**: Implements different sorting strategies.

```
public interface ISortStrategy { void Sort(List<int> list); }

public class QuickSort : ISortStrategy
{
    public void Sort(List<int> list)
    {
        if (list.Count <= 1) return;
        var pivot = list[list.Count / 2];
        var left = new List<int>();
        var right = new List<int>();

        foreach (var num in list)
            if (num < pivot) left.Add(num);
            else if (num > pivot) right.Add(num);

        Sort(left);
        Sort(right);

        list.Clear();
        left.ForEach(n => list.Add(n));
        list.Add(pivot);
        right.ForEach(n => list.Add(n));
    }
}

public class MergeSort : ISortStrategy
{
    public void Sort(List<int> list) { /* Implementation */ }
}

var context = new SortingContext(new QuickSort());
context.Sort(list);
```

- **Use Case**: Useful for dynamically selecting sorting algorithms based on requirements.

139. Template Method Pattern with News Publishing

- **Code Example**: Defines the news publishing process.

```
public abstract class NewsPublisher
{
    public void Publish()
    {
        GatherNews();
        WriteArticle();
        EditArticle();
        PublishOnline();
    }

    protected abstract void GatherNews();
    protected abstract void WriteArticle();
    protected abstract void EditArticle();
    protected abstract void PublishOnline();
}

public class NewspaperPublisher : NewsPublisher
{
    protected override void GatherNews() => Console.WriteLine("Gathering news from reporters");
    protected override void WriteArticle() => Console.WriteLine("Writing article by journalist");
```

```
    protected override void EditArticle() => Console.WriteLine("Editing for accuracy and style");
    protected override void PublishOnline() => Console.WriteLine("Publishing on website");
}

var publisher = new NewspaperPublisher();
publisher.Publish();
```

- **Use Case**: Useful for automating the news publishing workflow consistently.

140. Composite Pattern with Family Trees

- **Code Example**: Composes family members into a tree structure.

```
public interface IFamilyMember { void Display(); }

public class Person : IFamilyMember
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine(Name);
}

public class FamilyTree : IFamilyMember
{
    private readonly List<IFamilyMember> _members = new();

    public void Display()
    {
        Console.WriteLine("Family Tree");
        foreach (var member in _members)
            member.Display();
    }

    public void Add(IFamilyMember member) => _members.Add(member);
}

var family = new FamilyTree();
family.Add(new Person { Name = "John" });
family.Display();
```

- **Use Case**: Useful for managing hierarchical family structures.

141. Decorator Pattern with Text Processing

- **Code Example**: Enhances text processing capabilities.

```
public interface ITextProcessor { string Process(string input); }

public class BasicProcessor : ITextProcessor
{
    public string Process(string input) => input.ToUpper();
}

public class AdvancedProcessor : ITextProcessor
{
    private readonly ITextProcessor _processor;

    public AdvancedProcessor(ITextProcessor processor) => _processor = processor;
    public string Process(string input)
    {
        var processed = _processor.Process(input);
        return processed.Replace(" ", "_");
    }
}

var processor = new BasicProcessor();
processor = new AdvancedProcessor(processor);
Console.WriteLine(processor.Process("Hello World")); // HELLO_WORLD
```

- **Use Case**: Useful for adding layers of processing to text dynamically.

142. Proxy Pattern with File Operations

- **Code Example**: Provides a proxy for file operations to add logging.

```
public interface IFile { void Read(); }

public class RealFile : IFile
{
    public string Name { get; set; }
    public void Read() => Console.WriteLine("Reading " + Name);
}

public class FileProxy : IFile
{
    private RealFile _realFile;
    private string _filename;

    public FileProxy(string filename)
    {
        _filename = filename;
        _realFile = null;
    }

    public void Read()
    {
        if (_realFile == null) _realFile = new RealFile { Name = _filename };
        Console.WriteLine("Logging access to " + _filename);
        _realFile.Read();
    }
}

var file = new FileProxy("data.txt");
file.Read(); // Logs and reads
```

- **Use Case**: Useful for adding logging or other features to file operations without changing the core functionality.

143. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory overhead when dealing with numerous similar objects.

144. State Pattern with Robot Movements

- **Code Example**: Manages robot movement states like moving, stopped.

```
public interface IRobotState { void Move(); }

public class MovingState : IRobotState
{
    public void Move() => Console.WriteLine("Already moving");
}

public class StoppedState : IRobotState
{
    public void Move() => Console.WriteLine("Starting to move");
```

```
}

public class Robot
{
    private IRobotState _currentState;

    public Robot() => _currentState = new StoppedState();
    public void SetState(IRobotState state) => _currentState = state;
    public void Move() => _currentState.Move();
}

var robot = new Robot();
robot.Move(); // Starts moving
```

- **Use Case**: Useful for managing the operational states of robots or automated systems.

145. Strategy Pattern with Machine Learning Models

- **Code Example**: Implements different machine learning strategies.

```
public interface IMachineLearningStrategy { void Train(); }

public class NeuralNetwork : IMachineLearningStrategy
{
    public void Train() => Console.WriteLine("Training neural network");
}

public class SupportVectorMachine : IMachineLearningStrategy
{
    public void Train() => Console.WriteLine("Training SVM");
}

public class MLContext
{
    private IMachineLearningStrategy _strategy;

    public void SetStrategy(IMachineLearningStrategy strategy) => _strategy = strategy;
    public void TrainModel() => _strategy.Train();
}

var context = new MLContext();
context.SetStrategy(new NeuralNetwork());
context.TrainModel(); // Trains neural network
```

- **Use Case**: Useful for selecting and switching between different machine learning algorithms dynamically.

146. Template Method Pattern with Software Updates

- **Code Example**: Defines the software update process.

```
public abstract class SoftwareUpdater
{
    public void Update()
    {
        CheckForUpdates();
        DownloadUpdate();
        ApplyUpdate();
        RestartSystem();
    }

    protected abstract void CheckForUpdates();
    protected abstract void DownloadUpdate();
    protected abstract void ApplyUpdate();
    protected abstract void RestartSystem();
}

public class OSUpdater : SoftwareUpdater
{
    protected override void CheckForUpdates() => Console.WriteLine("Checking for OS updates");
    protected override void DownloadUpdate() => Console.WriteLine("Downloading latest OS version");
    protected override void ApplyUpdate() => Console.WriteLine("Applying updates");
    protected override void RestartSystem() => Console.WriteLine("Rebooting system");
}

var updater = new OSUpdater();
```

```
updater.Update();
```

- **Use Case**: Useful for automating software updates with consistent steps.

147. Composite Pattern with Organizational Hierarchies

- **Code Example**: Composes employees into an organizational structure.

```
public interface IOrganizationNode { void Display(); }

public class Employee : IOrganizationNode
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine(Name);
}

public class Department : IOrganizationNode
{
    private readonly List<IOrganizationNode> _members = new();

    public void Display()
    {
        Console.WriteLine("Department");
        foreach (var member in _members) member.Display();
    }

    public void Add(IOrganizationNode node) => _members.Add(node);
}

var department = new Department();
department.Add(new Employee { Name = "Alice" });
department.Display();
```

- **Use Case**: Useful for managing organizational structures and hierarchies.

148. Decorator Pattern with Logging

- **Code Example**: Decorates methods to add logging functionality.

```
public interface ICalculator { int Add(int a, int b); }

public class BasicCalculator : ICalculator
{
    public int Add(int a, int b) => a + b;
}

public class LoggingDecorator : ICalculator
{
    private readonly ICalculator _calculator;
    public LoggingDecorator(ICalculator calculator) => _calculator = calculator;
    public int Add(int a, int b)
    {
        Console.WriteLine($"Adding {a} and {b}");
        return _calculator.Add(a, b);
    }
}

var calculator = new BasicCalculator();
calculator = new LoggingDecorator(calculator);
Console.WriteLine(calculator.Add(2, 3)); // Logs and returns 5
```

- **Use Case**: Useful for adding logging to methods without modifying their core functionality.

149. Proxy Pattern with Remote Control

- **Code Example**: Provides a local proxy to control remote devices.

```
public interface IRemoteDevice { void PowerOn(); }

public class Television : IRemoteDevice
{
    public void PowerOn() => Console.WriteLine("Television powered on");
}
```

```
public class RemoteControl : IRemoteDevice
{
    private Television _television;

    public void PowerOn()
    {
        if (_television == null) _television = new Television();
        Console.WriteLine("Sending power on command");
        _television.PowerOn();
    }
}

var remote = new RemoteControl();
remote.PowerOn(); // Controls television remotely
```

- **Use Case**: Useful for controlling remote devices through a proxy interface.

150. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages characters to save memory.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage when dealing with a large number of similar objects.

151. State Pattern with Elevator System

- **Code Example**: Manages elevator states like moving, stopped.

```
public interface IElevatorState { void MoveFloor(); }

public class DoorClosedState : IElevatorState
{
    public void MoveFloor() => Console.WriteLine("Elevator moving to next floor");
}

public class DoorOpenState : IElevatorState
{
    public void MoveFloor() => Console.WriteLine("Cannot move, doors are open");
}

public class Elevator
{
    private IElevatorState _currentState;

    public Elevator() => _currentState = new DoorClosedState();
    public void SetState(IElevatorState state) => _currentState = state;
    public void MoveFloor() => _currentState.MoveFloor();
}

var elevator = new Elevator();
elevator.MoveFloor(); // Moves to next floor
```

- **Use Case**: Useful for managing the operational states of elevators or similar systems.

152. Strategy Pattern with Payment Methods

- **Code Example**: Implements different payment strategies.

```
public interface IPaymentStrategy { void Pay(decimal amount); }

public class BitcoinPayment : IPaymentStrategy
{
    public void Pay(decimal amount) => Console.WriteLine("Paid via Bitcoin: " + amount);
}

public class EthereumPayment : IPaymentStrategy
{
    public void Pay(decimal amount) => Console.WriteLine("Paid via Ethereum: " + amount);
}

public class PaymentContext
{
    private IPaymentStrategy _strategy;

    public void SetStrategy(IPaymentStrategy strategy) => _strategy = strategy;
    public void ProcessPayment(decimal amount) => _strategy.Pay(amount);
}

var context = new PaymentContext();
context.SetStrategy(new BitcoinPayment());
context.ProcessPayment(0.05m); // Pays 0.05 BTC
```

- **Use Case**: Useful for selecting payment methods dynamically, especially in systems that support multiple payment types.

153. Template Method Pattern with Compiler pipelining

- **Code Example**: Defines the compiler pipeline process.

```
public abstract class CompilerPipeline
{
    public void Compile()
    {
        Lex();
        Parse();
        GenerateCode();
    }

    protected abstract void Lex();
    protected abstract void Parse();
    protected abstract void GenerateCode();
}

public class CSharpCompiler : CompilerPipeline
{
    protected override void Lex() => Console.WriteLine("Lexing C# code");
    protected override void Parse() => Console.WriteLine("Parsing into AST");
    protected override void GenerateCode() => Console.WriteLine("Generating IL bytecode");
}

var compiler = new CSharpCompiler();
compiler.Compile();
```

- **Use Case**: Useful for automating the compilation process with consistent steps across different languages.

154. Composite Pattern with Web Components

- **Code Example**: Composes UI components into a web page.

```
public interface IWebComponent { void Render(); }

public class ButtonComponent : IWebComponent
{
    public string Text { get; set; }
    public void Render() => Console.WriteLine($"Rendered button: {Text}");
}

public class WebPage : IWebComponent
{
```

```
    private readonly List<IWebComponent> _components = new();

    public void Render()
    {
        Console.WriteLine("Rendering web page");
        foreach (var component in _components) component.Render();
    }

    public void Add(IWebComponent component) => _components.Add(component);
}

var page = new WebPage();
page.Add(new ButtonComponent { Text = "Submit" });
page.Render(); // Renders all components
```

- **Use Case**: Useful for managing web pages and their constituent components in a hierarchical manner.

155. Decorator Pattern with Network Monitoring

- **Code Example**: Decorates network connections to add monitoring features.

```
public interface INetworkConnection { void Connect(); }

public class BasicConnection : INetworkConnection
{
    public void Connect() => Console.WriteLine("Establishing basic connection");
}

public class MonitoringDecorator : INetworkConnection
{
    private readonly INetworkConnection _connection;

    public MonitoringDecorator(INetworkConnection connection) => _connection = connection;
    public void Connect()
    {
        Console.WriteLine("Monitoring network usage");
        _connection.Connect();
    }
}

var connection = new BasicConnection();
connection = new MonitoringDecorator(connection);
connection.Connect(); // Connects with monitoring
```

- **Use Case**: Useful for adding network monitoring without altering existing connection logic.

156. Proxy Pattern with API Rate Limiting

- **Code Example**: Provides a proxy to limit API calls.

```
public interface IAPI { string GetData(); }

public class RealAPI : IAPI
{
    public string GetData()
    {
        // Simulate API call with rate limiting
        if (DateTime.Now.Second % 2 == 0)
            throw new Exception("Rate limit exceeded");
        return "Data from API";
    }
}

public class RateLimitingProxy : IAPI
{
    private RealAPI _realAPI;
    private int _callCount;

    public RateLimitingProxy()
    {
        _realAPI = new RealAPI();
        _callCount = 0;
    }

    public string GetData()
```

```
    {
        if (_callCount >= 5)
            throw new Exception("Rate limit exceeded");
        _callCount++;
        return _realAPI.GetData();
    }
}

var proxy = new RateLimitingProxy();
try
{
    for (int i = 0; i < 10; i++)
        Console.WriteLine(proxy.GetData());
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message); // Handles rate limit
}
```

- **Use Case**: Useful for implementing API rate limiting to prevent abuse or overuse.

157. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage by reusing character objects when possible.

158. State Pattern with Game Levels

- **Code Example**: Manages game level states like playing, paused.

```
public interface IGameState { void Play(); }

public class PlayingState : IGameState
{
    public void Play() => Console.WriteLine("Game is playing");
}

public class PausedState : IGameState
{
    public void Play() => Console.WriteLine("Game is paused");
}

public class Game
{
    private IGameState _currentState;

    public Game() => _currentState = new PlayingState();
    public void SetState(IGameState state) => _currentState = state;
    public void Play() => _currentState.Play();
}

var game = new Game();
```

```
game.Play(); // Game is playing
```

- **Use Case**: Useful for managing the states of games, allowing transitions between playing, paused, etc.

159. Strategy Pattern with Search Algorithms

- **Code Example**: Implements different search strategies.

```
public interface ISearchStrategy { int Search(int[] data, int target); }

public class LinearSearch : ISearchStrategy
{
    public int Search(int[] data, int target)
    {
        for (int i = 0; i < data.Length; i++)
            if (data[i] == target) return i;
        return -1;
    }
}

public class BinarySearch : ISearchStrategy
{
    public int Search(int[] data, int target)
    {
        int left = 0;
        int right = data.Length - 1;

        while (left <= right)
        {
            int mid = left + (right - left) / 2;
            if (data[mid] == target) return mid;
            if (data[mid] < target) left = mid + 1;
            else right = mid - 1;
        }
        return -1;
    }
}

public class SearchContext
{
    private ISearchStrategy _strategy;

    public void SetStrategy(ISearchStrategy strategy) => _strategy = strategy;
    public int Search(int[] data, int target) => _strategy.Search(data, target);
}

var context = new SearchContext();
context.SetStrategy(new BinarySearch());
int[] data = { 1, 2, 3, 4, 5 };
Console.WriteLine(context.Search(data, 3)); // Returns index 2
```

- **Use Case**: Useful for dynamically selecting search algorithms based on data structure or performance needs.

160. Template Method Pattern with Build Automation

- **Code Example**: Defines a build automation process.

```
public abstract class BuildAutomation
{
    public void RunBuild()
    {
        Clean();
        RestorePackages();
        BuildSolution();
        RunTests();
    }

    protected abstract void Clean();
    protected abstract void RestorePackages();
    protected abstract void BuildSolution();
    protected abstract void RunTests();
}

public class CIPIPE : BuildAutomation
{
```

```
    protected override void Clean() => Console.WriteLine("Cleaning build artifacts");
    protected override void RestorePackages() => Console.WriteLine("Restoring NuGet packages");
    protected override void BuildSolution() => Console.WriteLine("Building solution with MSBuild");
    protected override void RunTests() => Console.WriteLine("Running automated tests");
}

var automation = new CIPIPE();
automation.RunBuild(); // Executes the build process
```

- **Use Case**: Useful for creating consistent and repeatable build automation pipelines across projects.

161. Composite Pattern with Mathematical Expressions

- **Code Example**: Composes expressions into a tree structure.

```
public interface IExpression { int Evaluate(); }

public class Number : IExpression
{
    public int Value { get; set; }
    public int Evaluate() => Value;
}

public class Addition : IExpression
{
    private readonly IExpression _left;
    private readonly IExpression _right;

    public Addition(IExpression left, IExpression right)
    {
        _left = left;
        _right = right;
    }

    public int Evaluate() => _left.Evaluate() + _right.Evaluate();
}

var expression = new Addition(new Number { Value = 5 }, new Number { Value = 3 });
Console.WriteLine(expression.Evaluate()); // Outputs 8
```

- **Use Case**: Useful for representing and evaluating mathematical expressions in a hierarchical manner.

162. Decorator Pattern with Text Encryption

- **Code Example**: Decorates text processing to include encryption.

```
public interface ITextProcessor { string Process(string input); }

public class BasicEncryptor : ITextProcessor
{
    public string Process(string input) => new string(input.Reverse().ToArray());
}

public class AdvancedEncryptor : ITextProcessor
{
    private readonly ITextProcessor _encryptor;

    public AdvancedEncryptor(ITextProcessor encryptor) => _encryptor = encryptor;
    public string Process(string input)
    {
        var encrypted = _encryptor.Process(input);
        return encrypted + "Encrypted";
    }
}

var processor = new BasicEncryptor();
processor = new AdvancedEncryptor(processor);
Console.WriteLine(processor.Process("Hello")); // olleHEncrypted
```

- **Use Case**: Useful for adding layers of encryption to text processing without altering core functionality.

163. Proxy Pattern with File Sharing

- **Code Example**: Provides a proxy for secure file sharing.

```
public interface IFile { string Read(); }

public class SecureFile : IFile
{
    private readonly string _content;
    private readonly string _password;

    public SecureFile(string content, string password)
    {
        _content = content;
        _password = password;
    }

    public string Read(string password)
    {
        if (password != _password) throw new UnauthorizedAccessException();
        return _content;
    }
}

public class FileProxy : IFile
{
    private SecureFile _secureFile;
    private string _password;

    public FileProxy(SecureFile file, string password)
    {
        _secureFile = file;
        _password = password;
    }

    public string Read() => _secureFile.Read(_password);
}

var file = new SecureFile("Sensitive data", "secret");
var proxy = new FileProxy(file, "secret");
Console.WriteLine(proxy.Read()); // Accesses secure file
```

- **Use Case**: Useful for securely sharing files by validating access through a proxy.

164. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage when working with numerous similar objects.

165. State Pattern with Automobile Systems

- **Code Example**: Manages states like driving, parked.

```
public interface IAutomobileState { void Drive(); }
```

```
public class ParkedState : IAutomobileState
{
    public void Drive() => Console.WriteLine("Starting to drive");
}

public class DrivingState : IAutomobileState
{
    public void Drive() => Console.WriteLine("Already driving");
}

public class Automobile
{
    private IAutomobileState _currentState;

    public Automobile() => _currentState = new ParkedState();
    public void SetState(IAutomobileState state) => _currentState = state;
    public void Drive() => _currentState.Drive();
}

var car = new Automobile();
car.Drive(); // Starts driving
```

- **Use Case**: Useful for managing the operational states of automobiles or vehicle systems.

166. Strategy Pattern with Authentication Methods

- **Code Example**: Implements different authentication strategies.

```
public interface IAuthenticationStrategy { bool Authenticate(string username, string password); }

public class BasicAuth : IAuthenticationStrategy
{
    public bool Authenticate(string username, string password)
    {
        // Simplified authentication logic
        return username == "admin" && password == "secret";
    }
}

public class OAuth : IAuthenticationStrategy
{
    public bool Authenticate(string username, string password)
    {
        // Simplified OAuth logic
        return true;
    }
}

public class AuthContext
{
    private IAuthenticationStrategy _strategy;

    public void SetStrategy(IAuthenticationStrategy strategy) => _strategy = strategy;
    public bool Authenticate(string username, string password) => _strategy.Authenticate(username,
password);
}

var context = new AuthContext();
context.SetStrategy(new BasicAuth());
bool result = context.Authenticate("admin", "secret"); // Returns true
```

- **Use Case**: Useful for selecting different authentication methods based on user or system requirements.

167. Template Method Pattern with Report Generation

- **Code Example**: Defines the report generation process.

```
public abstract class ReportGenerator
{
    public void GenerateReport()
    {
        CollectData();
        ProcessData();
        GenerateOutput();
    }
```

```
    protected abstract void CollectData();
    protected abstract void ProcessData();
    protected abstract void GenerateOutput();
}

public class SalesReport : ReportGenerator
{
    protected override void CollectData() => Console.WriteLine("Collecting sales data");
    protected override void ProcessData() => Console.WriteLine("Processing sales figures");
    protected override void GenerateOutput() => Console.WriteLine("Generating PDF report");
}

var generator = new SalesReport();
generator.GenerateReport(); // Executes the entire report generation process
```

- **Use Case**: Useful for automating report generation with consistent steps across different types of reports.

168. Composite Pattern with Document Structures

- **Code Example**: Composes document elements into a structured format.

```
public interface IDocumentElement { void Render(); }

public class TextElement : IDocumentElement
{
    public string Text { get; set; }
    public void Render() => Console.WriteLine("Rendered text: " + Text);
}

public class Document : IDocumentElement
{
    private readonly List<IDocumentElement> _elements = new();

    public void Render()
    {
        Console.WriteLine("Rendering document");
        foreach (var element in _elements)
            element.Render();
    }

    public void Add(IDocumentElement element) => _elements.Add(element);
}

var doc = new Document();
doc.Add(new TextElement { Text = "Hello, World!" });
doc.Render(); // Renders all elements
```

- **Use Case**: Useful for managing document structures with various elements in a hierarchical manner.

169. Decorator Pattern with Aspect-Oriented Programming

- **Code Example**: Decorates methods to add cross-cutting concerns like logging or performance monitoring.

```
public interface IService { void DoWork(); }

public class BasicService : IService
{
    public void DoWork() => Console.WriteLine("Basic service operation");
}

public class LoggingDecorator : IService
{
    private readonly IService _service;

    public LoggingDecorator(IService service) => _service = service;
    public void DoWork()
    {
        Console.WriteLine("Starting operation");
        _service.DoWork();
        Console.WriteLine("Operation completed");
    }
}

var service = new BasicService();
```

```
service = new LoggingDecorator(service);
service.DoWork(); // Logs start and end of operation
```

- **Use Case**: Useful for implementing cross-cutting concerns like logging, monitoring, or security without modifying core business logic.

170. Proxy Pattern with Content Filtering

- **Code Example**: Provides a proxy to filter content before display.

```
public interface IContentProvider { string GetContent(); }

public class RealContent : IContentProvider
{
    public string GetContent() => "Some sensitive content here";
}

public class ContentFilterProxy : IContentProvider
{
    private RealContent _content;

    public string GetContent()
    {
        if (_content == null) _content = new RealContent();
        string filtered = _content.GetContent()
            .Replace("sensitive", "filtered")
            .Replace("password", "***");
        return filtered;
    }
}

var provider = new ContentFilterProxy();
Console.WriteLine(provider.GetContent()); // Displays filtered content
```

- **Use Case**: Useful for filtering or sanitizing content before it is displayed or processed further.

171. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for optimizing memory usage when dealing with numerous similar objects by sharing them where possible.

172. State Pattern with Network Connections

- **Code Example**: Manages network connection states like connected, disconnected.

```
public interface INetworkState { void Connect(); }

public class DisconnectedState : INetworkState
{
```

```
    public void Connect() => Console.WriteLine("Establishing network connection");
}

public class ConnectedState : INetworkState
{
    public void Connect() => Console.WriteLine("Already connected");
}

public class NetworkManager
{
    private INetworkState _currentState;

    public NetworkManager() => _currentState = new DisconnectedState();
    public void SetState(INetworkState state) => _currentState = state;
    public void Connect() => _currentState.Connect();
}

var manager = new NetworkManager();
manager.Connect(); // Connects to network
```

- **Use Case**: Useful for managing the states of network connections, allowing for transitions between connected and disconnected states.

173. Strategy Pattern with Compression Algorithms

- **Code Example**: Implements different compression strategies.

```
public interface ICompressionStrategy { byte[] Compress(byte[] data); }

public class GZipCompression : ICompressionStrategy
{
    public byte[] Compress(byte[] data)
    {
        // Simulate GZip compression
        return new byte[] { 0x1F, 0x8B };
    }
}

public class DeflateCompression : ICompressionStrategy
{
    public byte[] Compress(byte[] data)
    {
        // Simulate Deflate compression
        return new byte[] { 0x78, 0x9C };
    }
}

public class Compressor
{
    private ICompressionStrategy _strategy;
    public void SetStrategy(ICompressionStrategy strategy) => _strategy = strategy;
    public byte[] Compress(byte[] data) => _strategy.Compress(data);
}

var compressor = new Compressor();
compressor.SetStrategy(new GZipCompression());
byte[] compressed = compressor.Compress(Encoding.UTF8.GetBytes("Hello, World!"));
```

- **Use Case**: Useful for selecting different compression algorithms based on file type, size, or performance needs.

174. Template Method Pattern with Data Processing Pipelines

- **Code Example**: Defines a data processing pipeline.

```
public abstract class DataPipeline
{
    public void ProcessData()
    {
        ReadData();
        TransformData();
        WriteData();
    }

    protected abstract void ReadData();
```

```
    protected abstract void TransformData();
    protected abstract void WriteData();
}

public class ETLProcess : DataPipeline
{
    protected override void ReadData() => Console.WriteLine("Reading data from source");
    protected override void TransformData() => Console.WriteLine("Transforming data");
    protected override void WriteData() => Console.WriteLine("Writing transformed data to destination");
}

var pipeline = new ETLProcess();
pipeline.ProcessData(); // Executes the entire data processing workflow
```

- **Use Case**: Useful for creating consistent and repeatable data processing pipelines across different systems or datasets.

175. Composite Pattern with Tree Structures

- **Code Example**: Composes nodes into a tree structure.

```
public interface ITreeNode { void Display(); }

public class LeafNode : ITreeNode
{
    public string Value { get; set; }
    public void Display() => Console.WriteLine("Leaf: " + Value);
}

public class InternalNode : ITreeNode
{
    private readonly List<ITreeNode> _children = new();

    public void Display()
    {
        Console.WriteLine("Internal node");
        foreach (var child in _children)
            child.Display();
    }

    public void Add(ITreeNode node) => _children.Add(node);
}

var root = new InternalNode();
root.Add(new LeafNode { Value = "A" });
root.Display(); // Displays the entire tree structure
```

- **Use Case**: Useful for managing hierarchical data structures like trees, where each node can be either a leaf or an internal node containing other nodes.

176. Decorator Pattern with Performance Monitoring

- **Code Example**: Decorates methods to monitor their performance.

```
public interface IService { void DoWork(); }

public class BasicService : IService
{
    public void DoWork()
    {
        Thread.Sleep(100); // Simulate work
        Console.WriteLine("Work completed");
    }
}

public class PerformanceDecorator : IService
{
    private readonly IService _service;

    public PerformanceDecorator(IService service) => _service = service;
    public void DoWork()
    {
        var stopwatch = new Stopwatch();
        stopwatch.Start();
        _service.DoWork();
        stopwatch.Stop();
```

```
        Console.WriteLine($"Operation took {stopwatch.ElapsedMilliseconds} ms");
    }
}

var service = new BasicService();
service = new PerformanceDecorator(service);
service.DoWork(); // Monitors and displays performance
```

- **Use Case**: Useful for monitoring the performance of methods without altering their core functionality, aiding in optimization and debugging.

177. Proxy Pattern with Database Connections

- **Code Example**: Provides a proxy to manage database connections efficiently.

```
public interface IDatabase { void Query(string sql); }

public class RealDatabase : IDatabase
{
    public void Query(string sql) => Console.WriteLine("Executing query: " + sql);
}

public class ConnectionPoolProxy : IDatabase
{
    private RealDatabase _database;
    private int _connectionCount;

    public void Query(string sql)
    {
        if (_database == null) _database = new RealDatabase();
        Console.WriteLine("Retrieved connection from pool");
        _connectionCount++;
        if (_connectionCount > 10)
            throw new Exception("Connection limit reached");
        _database.Query(sql);
    }
}

var db = new ConnectionPoolProxy();
for (int i = 0; i < 15; i++)
    db.Query("SELECT * FROM table"); // Throws exception after 10 connections
```

- **Use Case**: Useful for managing database connections efficiently, preventing overuse and ensuring optimal resource utilization.

178. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage by sharing character objects where possible, particularly in scenarios with a large number of similar small objects.

179. State Pattern with User Sessions

- **Code Example**: Manages user session states like active, inactive.

```
public interface IUserSession { void AccessSystem(); }

public class ActiveSession : IUserSession
{
    public void AccessSystem() => Console.WriteLine("User has active session");
}

public class InactiveSession : IUserSession
{
    public void AccessSystem() => Console.WriteLine("User session expired, redirecting to login");
}

public class SessionManager
{
    private IUserSession _currentState;

    public SessionManager() => _currentState = new InactiveSession();
    public void SetState(IUserSession state) => _currentState = state;
    public void AccessSystem() => _currentState.AccessSystem();
}

var manager = new SessionManager();
manager.AccessSystem(); // Handles expired session
```

- **Use Case**: Useful for managing user sessions, allowing transitions between active and inactive states based on user activity or time limits.

180. Strategy Pattern with Sorting Algorithms

- **Code Example**: Implements different sorting strategies.

```
public interface ISortStrategy { void Sort(int[] array); }

public class BubbleSort : ISortStrategy
{
    public void Sort(int[] array)
    {
        int n = array.Length;
        for (int i = 0; i < n - 1; i++)
            for (int j = 0; j < n - i - 1; j++)
                if (array[j] > array[j + 1])
                    Swap(ref array[j], ref array[j + 1]);
    }

    private void Swap(ref int a, ref int b)
    {
        int temp = a;
        a = b;
        b = temp;
    }
}

public class QuickSort : ISortStrategy
{
    public void Sort(int[] array) => QuickSortRecursive(array, 0, array.Length - 1);

    private void QuickSortRecursive(int[] array, int low, int high)
    {
        if (low < high)
        {
            int pivotIndex = Partition(array, low, high);
            QuickSortRecursive(array, low, pivotIndex - 1);
            QuickSortRecursive(array, pivotIndex + 1, high);
        }
    }

    private int Partition(int[] array, int low, int high)
    {
        int pivot = array[high];
        int i = (low - 1);
        for (int j = low; j < high; j++)
```

```
            if (array[j] <= pivot)
                Swap(ref array[++i], ref array[j]);
        Swap(ref array[i + 1], ref array[high]);
        return i + 1;
    }

    private void Swap(ref int a, ref int b)
    {
        int temp = a;
        a = b;
        b = temp;
    }
}

public class Sorter
{
    private ISortStrategy _strategy;

    public void SetStrategy(ISortStrategy strategy) => _strategy = strategy;
    public void Sort(int[] array) => _strategy.Sort(array);
}

var sorter = new Sorter();
sorter.SetStrategy(new QuickSort());
int[] data = { 3, 1, 4, 1, 5, 9 };
sorter.Sort(data);
Console.WriteLine(string.Join(", ", data)); // Sorted array
```

- **Use Case**: Useful for selecting different sorting algorithms based on data size, performance requirements, or specific algorithm properties.

181. Template Method Pattern with Build Systems

- **Code Example**: Defines the build system process.

```
public abstract class BuildSystem
{
    public void ExecuteBuild()
    {
        CheckDependencies();
        RunTests();
        PackageOutput();
    }

    protected abstract void CheckDependencies();
    protected abstract void RunTests();
    protected abstract void PackageOutput();
}

public class MavenBuild : BuildSystem
{
    protected override void CheckDependencies() => Console.WriteLine("Resolving Maven dependencies");
    protected override void RunTests() => Console.WriteLine("Running JUnit tests");
    protected override void PackageOutput() => Console.WriteLine("Generating JAR package");
}

var system = new MavenBuild();
system.ExecuteBuild(); // Executes the build process
```

- **Use Case**: Useful for creating consistent and repeatable build processes across different projects or programming languages.

182. Composite Pattern with Scene Graphs

- **Code Example**: Composes graphical objects into a scene graph.

```
public interface ISceneNode { void Render(); }

public class Shape : ISceneNode
{
    public string Type { get; set; }
    public void Render() => Console.WriteLine("Rendered shape: " + Type);
}
```

```
public class SceneGraph : ISceneNode
{
    private readonly List<ISceneNode> _nodes = new();

    public void Render()
    {
        Console.WriteLine("Rendering scene");
        foreach (var node in _nodes)
            node.Render();
    }

    public void Add(ISceneNode node) => _nodes.Add(node);
}

var scene = new SceneGraph();
scene.Add(new Shape { Type = "Square" });
scene.Render(); // Renders all nodes in the graph
```

- **Use Case**: Useful for managing complex graphical scenes with hierarchical structures, allowing efficient rendering and manipulation of graphical objects.

183. Decorator Pattern with User Interface Components

- **Code Example**: Decorates UI components to add additional features.

```
public interface IUIComponent { void Render(); }

public class Button : IUIComponent
{
    public string Text { get; set; }
    public void Render() => Console.WriteLine("Button rendered with text: " + Text);
}

public class ThemeDecorator : IUIComponent
{
    private readonly IUIComponent _component;

    public ThemeDecorator(IUIComponent component) => _component = component;
    public void Render()
    {
        Console.WriteLine("Applying dark theme");
        _component.Render();
    }
}

var button = new Button { Text = "Click Me" };
button = new ThemeDecorator(button);
button.Render(); // Renders with dark theme
```

- **Use Case**: Useful for adding themes, styles, or additional features to UI components dynamically without modifying their core structure.

184. Proxy Pattern with Remote File Access

- **Code Example**: Provides a proxy to access remote files.

```
public interface IFile { string Read(); }

public class RemoteFile : IFile
{
    private readonly string _path;

    public RemoteFile(string path) => _path = path;
    public string Read() => $"Content of remote file {_path}";
}

public class FileProxy : IFile
{
    private RemoteFile _file;
    private string _cache;

    public FileProxy(RemoteFile file)
    {
        _file = file;
```

```
    }

    public string Read()
    {
        if (_cache == null)
            _cache = _file.Read();
        return _cache;
    }
}

var file = new RemoteFile("/remote/path");
var proxy = new FileProxy(file);
Console.WriteLine(proxy.Read()); // Accesses remote file through proxy
```

- **Use Case**: Useful for efficiently accessing remote files by caching their content and reducing unnecessary network calls.

185. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for optimizing memory usage by sharing character objects, especially in scenarios where a large number of similar objects are created.

186. State Pattern with Telephone Systems

- **Code Example**: Manages telephone states like idle, ringing.

```
public interface ITelephoneState { void HandleCall(); }

public class IdleState : ITelephoneState
{
    public void HandleCall() => Console.WriteLine("Handling incoming call");
}

public class RingingState : ITelephoneState
{
    public void HandleCall() => Console.WriteLine("Already ringing");
}

public class Phone
{
    private ITelephoneState _currentState;

    public Phone() => _currentState = new IdleState();
    public void SetState(ITelephoneState state) => _currentState = state;
    public void HandleCall() => _currentState.HandleCall();
}

var phone = new Phone();
phone.HandleCall(); // Handles incoming call
```

- **Use Case**: Useful for managing the states of telephone systems, allowing transitions between different operational states

based on user interactions or system events.

187. Strategy Pattern with Machine Learning Models

- **Code Example**: Implements different machine learning models.

```
public interface IModel { double Predict(double[] features); }

public class LinearRegression : IModel
{
    public double Predict(double[] features)
    {
        // Simplified linear regression prediction
        return 2 * features[0] + 3 * features[1];
    }
}

public class RandomForest : IModel
{
    public double Predict(double[] features)
    {
        // Simplified random forest prediction
        return 1.5 * features[0] + 2.5 * features[1];
    }
}

public class ModelSelector
{
    private IModel _model;

    public void SetModel(IModel model) => _model = model;
    public double Predict(double[] features) => _model.Predict(features);
}

var selector = new ModelSelector();
selector.SetModel(new RandomForest());
double prediction = selector.Predict(new double[] { 1, 2 });
Console.WriteLine(prediction); // Outputs 6.5
```

- **Use Case**: Useful for dynamically selecting different machine learning models based on performance, accuracy, or specific requirements.

188. Template Method Pattern with Algorithm Frameworks

- **Code Example**: Defines an algorithm framework.

```
public abstract class AlgorithmFramework
{
    public void ExecuteAlgorithm()
    {
        Initialize();
        Process();
        Cleanup();
    }

    protected abstract void Initialize();
    protected abstract void Process();
    protected abstract void Cleanup();
}

public class SortingAlgorithm : AlgorithmFramework
{
    private int[] _data;

    protected override void Initialize()
    {
        _data = new int[] { 5, 3, 8, 1, 2 };
        Console.WriteLine("Initialized sorting algorithm");
    }

    protected override void Process()
    {
        Array.Sort(_data);
        Console.WriteLine("Processed data: " + string.Join(", ", _data));
    }
```

```
    protected override void Cleanup()
    {
        _data = null;
        Console.WriteLine("Cleaned up resources");
    }
}

var algorithm = new SortingAlgorithm();
algorithm.ExecuteAlgorithm(); // Executes the entire process
```

- **Use Case**: Useful for creating consistent and repeatable algorithm frameworks, allowing different implementations to fit into the same structure.

189. Composite Pattern with Organizational Hierarchies

- **Code Example**: Composes employees into an organizational hierarchy.

```
public interface IEmployee { void Display(); }

public class Employee : IEmployee
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine("Employee: " + Name);
}

public class Department : IEmployee
{
    private readonly List<IEmployee> _members = new();

    public void Display()
    {
        Console.WriteLine("Department");
        foreach (var member in _members)
            member.Display();
    }

    public void Add(IEmployee employee) => _members.Add(employee);
}

var department = new Department();
department.Add(new Employee { Name = "Alice" });
department.Display(); // Displays entire department structure
```

- **Use Case**: Useful for managing organizational structures with hierarchical relationships, allowing efficient traversal and management of employees and departments.

190. Decorator Pattern with Email Notifications

- **Code Example**: Decorates email sending to add different notification features.

```
public interface IEmailService { void Send(string to, string message); }

public class BasicEmail : IEmailService
{
    public void Send(string to, string message) => Console.WriteLine($"Sending email to {to}: {message}");
}

public class PriorityDecorator : IEmailService
{
    private readonly IEmailService _email;

    public PriorityDecorator(IEmailService email) => _email = email;
    public void Send(string to, string message)
    {
        Console.WriteLine("Marking as priority");
        _email.Send(to, message);
    }
}

var email = new BasicEmail();
email = new PriorityDecorator(email);
email.Send("user@example.com", "Hello"); // Sends with priority
```

- **Use Case**: Useful for adding different notification features like priority, read receipts, or cc/bcc without altering the core email sending functionality.

191. Proxy Pattern with Caching Mechanisms

- **Code Example**: Provides a proxy to cache data for efficient access.

```
public interface IDataAccess { string GetData(string key); }

public class RealData : IDataAccess
{
    public string GetData(string key)
    {
        // Simulate data retrieval from a slow source
        Thread.Sleep(1000);
        return $"Data for {key}";
    }
}

public class CachingProxy : IDataAccess
{
    private RealData _realData;
    private Dictionary<string, string> _cache = new();
    private readonly object _lock = new();

    public string GetData(string key)
    {
        lock (_lock)
        {
            if (!_cache.TryGetValue(key, out var value))
            {
                _realData ??= new RealData();
                value = _realData.GetData(key);
                _cache.Add(key, value);
            }
            return value;
        }
    }
}

var data = new CachingProxy();
string result1 = data.GetData("key1"); // Retrieves from real source
string result2 = data.GetData("key1"); // Retrieves from cache
```

- **Use Case**: Useful for improving performance by caching frequently accessed data, reducing the load on backend systems and providing quicker responses to users.

192. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory consumption by sharing instances of frequently used small objects, such as

characters in text processing applications.

193. State Pattern with Software Updaters

- **Code Example**: Manages update states like checking, downloading.

```
public interface IUpdateState { void CheckForUpdates(); }

public class CheckingState : IUpdateState
{
    public void CheckForUpdates() => Console.WriteLine("Checking for updates...");
}

public class DownloadingState : IUpdateState
{
    public void CheckForUpdates() => Console.WriteLine("Downloading updates...");
}

public class SoftwareUpdater
{
    private IUpdateState _currentState;

    public SoftwareUpdater() => _currentState = new CheckingState();
    public void SetState(IUpdateState state) => _currentState = state;
    public void CheckForUpdates() => _currentState.CheckForUpdates();
}

var updater = new SoftwareUpdater();
updater.CheckForUpdates(); // Checks for updates
```

- **Use Case**: Useful for managing the states of a software updater, allowing transitions between different stages like checking, downloading, and installing updates.

194. Strategy Pattern with Payment Gateways

- **Code Example**: Implements different payment processing strategies.

```
public interface IPaymentStrategy { void ProcessPayment(decimal amount); }

public class CreditCard : IPaymentStrategy
{
    public void ProcessPayment(decimal amount) => Console.WriteLine($"Charging {amount:C} to credit
card");
}

public class PayPal : IPaymentStrategy
{
    public void ProcessPayment(decimal amount) => Console.WriteLine($"Transferring {amount:C} via
PayPal");
}

public class PaymentProcessor
{
    private IPaymentStrategy _strategy;

    public void SetStrategy(IPaymentStrategy strategy) => _strategy = strategy;
    public void ProcessPayment(decimal amount) => _strategy.ProcessPayment(amount);
}

var processor = new PaymentProcessor();
processor.SetStrategy(new PayPal());
processor.ProcessPayment(100.00m); // Processes payment via PayPal
```

- **Use Case**: Useful for selecting different payment methods based on user preference, transaction type, or security requirements.

195. Template Method Pattern with Testing Frameworks

- **Code Example**: Defines a testing framework process.

```
public abstract class TestFramework
{
    public void RunTests()
```

```
    {
        Initialize();
        Execute();
        Cleanup();
    }

    protected abstract void Initialize();
    protected abstract void Execute();
    protected abstract void Cleanup();
}

public class UnitTestRunner : TestFramework
{
    protected override void Initialize() => Console.WriteLine("Initializing unit tests");
    protected override void Execute() => Console.WriteLine("Executing unit test cases");
    protected override void Cleanup() => Console.WriteLine("Cleaning up resources");
}

var runner = new UnitTestRunner();
runner.RunTests(); // Executes the entire testing process
```

- **Use Case**: Useful for creating consistent and repeatable testing frameworks, allowing different test runners to fit into the same structure while executing their specific logic.

196. Composite Pattern with Song Playlists

- **Code Example**: Composes songs into a playlist structure.

```
public interface IMediaItem { void Play(); }

public class Song : IMediaItem
{
    public string Title { get; set; }
    public void Play() => Console.WriteLine("Playing song: " + Title);
}

public class Playlist : IMediaItem
{
    private readonly List<IMediaItem> _songs = new();

    public void Play()
    {
        Console.WriteLine("Playing playlist");
        foreach (var song in _songs)
            song.Play();
    }

    public void Add(IMediaItem song) => _songs.Add(song);
}

var playlist = new Playlist();
playlist.Add(new Song { Title = "Bohemian Rhapsody" });
playlist.Play(); // Plays all songs in the playlist
```

- **Use Case**: Useful for managing playlists with multiple songs, allowing efficient playback and manipulation of the entire collection as a single unit.

197. Decorator Pattern with Logging

- **Code Example**: Decorates methods to add logging functionality.

```
public interface IService { void DoWork(); }

public class BasicService : IService
{
    public void DoWork()
    {
        Thread.Sleep(500); // Simulate work
        Console.WriteLine("Work completed");
    }
}

public class LoggingDecorator : IService
{
```

```
    private readonly IService _service;

    public LoggingDecorator(IService service) => _service = service;
    public void DoWork()
    {
        Console.WriteLine("Starting operation");
        _service.DoWork();
        Console.WriteLine("Operation completed at " + DateTime.Now);
    }
}

var service = new BasicService();
service = new LoggingDecorator(service);
service.DoWork(); // Logs start and end of operation
```

- **Use Case**: Useful for adding logging to methods without altering their core functionality, aiding in debugging and monitoring.

198. Proxy Pattern with Content Delivery Networks (CDN)

- **Code Example**: Provides a proxy to serve content from a CDN.

```
public interface IContentDelivery { string GetResource(string url); }

public class RealServer : IContentDelivery
{
    public string GetResource(string url)
    {
        // Simulate resource retrieval from a remote server
        Thread.Sleep(1000);
        return $"Resource from {url}";
    }
}

public class CDNServer : IContentDelivery
{
    private RealServer _realServer;
    private Dictionary<string, string> _cache = new();

    public string GetResource(string url)
    {
        if (_cache.TryGetValue(url, out var resource))
            return resource;

        _realServer ??= new RealServer();
        string resource = _realServer.GetResource(url);
        _cache[url] = resource;
        return resource;
    }
}

var cdn = new CDNServer();
string resource1 = cdn.GetResource("/image.jpg"); // Retrieves from server
string resource2 = cdn.GetResource("/image.jpg"); // Serves from cache
```

- **Use Case**: Useful for improving content delivery performance by caching frequently accessed resources in a CDN, reducing latency and server load.

199. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
```

```
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage by sharing instances of frequently used characters, which is particularly beneficial in applications dealing with large text processing or displaying.

200. State Pattern with Video Player Controls

- **Code Example**: Manages video player states like playing, paused.

```
public interface IVideoPlayerState { void Play(); }

public class PausedState : IVideoPlayerState
{
    public void Play() => Console.WriteLine("Starting playback");
}

public class PlayingState : IVideoPlayerState
{
    public void Play() => Console.WriteLine("Already playing");
}

public class VideoPlayer
{
    private IVideoPlayerState _currentState;

    public VideoPlayer() => _currentState = new PausedState();
    public void SetState(IVideoPlayerState state) => _currentState = state;
    public void Play() => _currentState.Play();
}

var player = new VideoPlayer();
player.Play(); // Starts playback
```

- **Use Case**: Useful for managing the states of a video player, allowing transitions between playing and paused states based on user interactions or system events.

201. Strategy Pattern with Search Algorithms

- **Code Example**: Implements different search strategies.

```
public interface ISearchStrategy { int Search(int[] array, int target); }

public class LinearSearch : ISearchStrategy
{
    public int Search(int[] array, int target)
    {
        for (int i = 0; i < array.Length; i++)
            if (array[i] == target)
                return i;
        return -1;
    }
}

public class BinarySearch : ISearchStrategy
{
    public int Search(int[] array, int target)
    {
        Array.Sort(array);
        int left = 0;
        int right = array.Length - 1;

        while (left <= right)
        {
            int middle = left + (right - left) / 2;
            if (array[middle] == target)
                return middle;
```

```
            else if (array[middle] < target)
                left = middle + 1;
            else
                right = middle - 1;
        }

        return -1;
    }
}

public class Searcher
{
    private ISearchStrategy _strategy;

    public void SetStrategy(ISearchStrategy strategy) => _strategy = strategy;
    public int Search(int[] array, int target) => _strategy.Search(array, target);
}

var searcher = new Searcher();
searcher.SetStrategy(new BinarySearch());
int[] data = { 3, 1, 4, 1, 5, 9 };
int index = searcher.Search(data, 4);
Console.WriteLine(index); // Outputs the found index
```

- **Use Case**: Useful for selecting different search algorithms based on data characteristics, such as whether the data is sorted or not, to optimize performance and accuracy.

202. Template Method Pattern with Compiler Pipelines

- **Code Example**: Defines a compiler pipeline process.

```
public abstract class CompilerPipeline
{
    public void Compile()
    {
        Lex();
        Parse();
        GenerateCode();
    }

    protected abstract void Lex();
    protected abstract void Parse();
    protected abstract void GenerateCode();
}

public class CSharpCompiler : CompilerPipeline
{
    protected override void Lex() => Console.WriteLine("Lexing C# code");
    protected override void Parse() => Console.WriteLine("Parsing C# code");
    protected override void GenerateCode() => Console.WriteLine("Generating IL code");
}

var compiler = new CSharpCompiler();
compiler.Compile(); // Executes the entire compilation process
```

- **Use Case**: Useful for creating consistent and repeatable compiler pipelines, allowing different language compilers to fit into the same structure while handling their specific logic.

203. Composite Pattern with Furniture Assemblies

- **Code Example**: Composes parts into a furniture assembly.

```
public interface IFurniturePart { void Assemble(); }

public class Screw : IFurniturePart
{
    public void Assemble() => Console.WriteLine("Screw attached");
}

public class FurnitureAssembly : IFurniturePart
{
    private readonly List<IFurniturePart> _parts = new();

    public void Assemble()
```

```
    {
        Console.WriteLine("Assembling furniture");
        foreach (var part in _parts)
            part.Assemble();
    }

    public void Add(IFurniturePart part) => _parts.Add(part);
}

var assembly = new FurnitureAssembly();
assembly.Add(new Screw());
assembly.Assemble(); // Assembles all parts
```

- **Use Case**: Useful for managing complex assemblies by treating individual parts and sub-assemblies uniformly, allowing efficient assembly of components in a hierarchical manner.

204. Decorator Pattern with Authorization Checks

- **Code Example**: Decorates methods to add authorization checks.

```
public interface IService { void DoWork(); }

public class BasicService : IService
{
    public void DoWork() => Console.WriteLine("Performing work");
}

public class AuthorizationDecorator : IService
{
    private readonly IService _service;

    public AuthorizationDecorator(IService service) => _service = service;
    public void DoWork()
    {
        if (IsAuthorized())
            _service.DoWork();
        else
            throw new UnauthorizedAccessException("Access denied");
    }

    private bool IsAuthorized() => true; // Simplified check
}

var service = new BasicService();
service = new AuthorizationDecorator(service);
try
{
    service.DoWork(); // Performs work if authorized
}
catch (UnauthorizedAccessException ex)
{
    Console.WriteLine(ex.Message);
}
```

- **Use Case**: Useful for adding authorization checks to methods without altering their core functionality, enhancing security by controlling access based on user roles or permissions.

205. Proxy Pattern with Resource Loading

- **Code Example**: Provides a proxy to load resources on demand.

```
public interface IResourceLoader { object Load(string resource); }

public class RealResource : IResourceLoader
{
    public object Load(string resource)
    {
        // Simulate loading a heavy resource
        Thread.Sleep(2000);
        return $"Loaded {resource}";
    }
}

public class LazyLoader : IResourceLoader
```

```
{
    private RealResource _realResource;
    private Dictionary<string, object> _cache = new();

    public object Load(string resource)
    {
        if (_cache.TryGetValue(resource, out var loadedResource))
            return loadedResource;

        _realResource ??= new RealResource();
        object resource = _realResource.Load(resource);
        _cache[resource] = loadedResource;
        return loadedResource;
    }
}

var loader = new LazyLoader();
object resource1 = loader.Load("image.jpg"); // Loads on demand
object resource2 = loader.Load("image.jpg"); // Serves from cache
```

- **Use Case**: Useful for improving performance by loading resources only when needed and caching them for future use, reducing unnecessary resource consumption and enhancing user experience.

206. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage by sharing instances of frequently used characters, which is particularly beneficial in scenarios involving extensive text processing or display.

207. State Pattern with Elevator Systems

- **Code Example**: Manages elevator states like moving, stopped.

```
public interface IElevatorState { void MoveTo(int floor); }

public class MovingState : IElevatorState
{
    public void MoveTo(int floor) => Console.WriteLine("Moving to floor " + floor);
}

public class StoppedState : IElevatorState
{
    public void MoveTo(int floor) => Console.WriteLine("Starting to move to floor " + floor);
}

public class Elevator
{
    private IElevatorState _currentState;

    public Elevator() => _currentState = new StoppedState();
    public void SetState(IElevatorState state) => _currentState = state;
```

```
    public void MoveTo(int floor) => _currentState.MoveTo(floor);
}

var elevator = new Elevator();
elevator.MoveTo(5); // Handles movement based on current state
```

- **Use Case**: Useful for managing the states of an elevator system, allowing transitions between moving and stopped states based on user requests or sensor inputs.

208. Strategy Pattern with Encryption Methods

- **Code Example**: Implements different encryption strategies.

```
public interface IEncryptionStrategy { string Encrypt(string plaintext); }

public class AES : IEncryptionStrategy
{
    public string Encrypt(string plaintext) => $"AES encrypted: {plaintext}";
}

public class RSA : IEncryptionStrategy
{
    public string Encrypt(string plaintext) => $"RSA encrypted: {plaintext}";
}

public class Encryptor
{
    private IEncryptionStrategy _strategy;

    public void SetStrategy(IEncryptionStrategy strategy) => _strategy = strategy;
    public string Encrypt(string plaintext) => _strategy.Encrypt(plaintext);
}

var encryptor = new Encryptor();
encryptor.SetStrategy(new RSA());
string encrypted = encryptor.Encrypt("Secret message");
Console.WriteLine(encrypted); // Outputs RSA encrypted message
```

- **Use Case**: Useful for selecting different encryption methods based on security requirements, data type, or compatibility with third-party systems.

209. Template Method Pattern with Frameworks

- **Code Example**: Defines a framework process.

```
public abstract class Framework
{
    public void Execute()
    {
        Initialize();
        Run();
        Shutdown();
    }

    protected abstract void Initialize();
    protected abstract void Run();
    protected abstract void Shutdown();
}

public class WebServerFramework : Framework
{
    protected override void Initialize() => Console.WriteLine("Initializing web server");
    protected override void Run() => Console.WriteLine("Running web server");
    protected override void Shutdown() => Console.WriteLine("Shutting down web server");
}

var framework = new WebServerFramework();
framework.Execute(); // Executes the entire process
```

- **Use Case**: Useful for creating consistent and repeatable frameworks, allowing different implementations to fit into the same structure while handling their specific initialization, execution, and shutdown processes.

210. Composite Pattern with Scene Layers

- **Code Example**: Composes graphical layers into a scene.

```
public interface ISceneLayer { void Render(); }

public class Background : ISceneLayer
{
    public void Render() => Console.WriteLine("Rendering background");
}

public class Scene : ISceneLayer
{
    private readonly List<ISceneLayer> _layers = new();

    public void Render()
    {
        Console.WriteLine("Rendering scene");
        foreach (var layer in _layers)
            layer.Render();
    }

    public void Add(ISceneLayer layer) => _layers.Add(layer);
}

var scene = new Scene();
scene.Add(new Background());
scene.Render(); // Renders all layers
```

- **Use Case**: Useful for managing complex graphical scenes with multiple layers, allowing efficient rendering and manipulation of each layer as a separate component.

211. Decorator Pattern with Data Validation

- **Code Example**: Decorates data input to add validation checks.

```
public interface IDataInput { void Submit(); }

public class Form : IDataInput
{
    public void Submit() => Console.WriteLine("Form submitted");
}

public class ValidationDecorator : IDataInput
{
    private readonly IDataInput _input;

    public ValidationDecorator(IDataInput input) => _input = input;
    public void Submit()
    {
        if (IsValid())
            _input.Submit();
        else
            Console.WriteLine("Validation failed");
    }

    private bool IsValid() => true; // Simplified validation
}

var form = new Form();
form = new ValidationDecorator(form);
form.Submit(); // Submits form if valid
```

- **Use Case**: Useful for adding data validation checks to input handling without altering the core submission functionality, ensuring data integrity and preventing invalid submissions.

212. Proxy Pattern with API Rate Limiting

- **Code Example**: Provides a proxy to limit API calls.

```
public interface IAPI { string GetData(); }

public class RealAPI : IAPI
{
    private int _callCount = 0;
```

```
    public string GetData()
    {
        if (_callCount >= 10)
            throw new InvalidOperationException("Rate limit exceeded");

        _callCount++;
        return "Data from API";
    }
}

public class RateLimitProxy : IAPI
{
    private RealAPI _api;

    public string GetData()
    {
        try
        {
            if (_api == null)
                _api = new RealAPI();
            return _api.GetData();
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine(ex.Message);
            return null;
        }
    }
}

var api = new RateLimitProxy();
for (int i = 0; i < 15; i++)
    Console.WriteLine(api.GetData()); // Handles rate limiting
```

- **Use Case**: Useful for enforcing API rate limits by intercepting calls through a proxy, preventing excessive usage and ensuring compliance with service terms.

213. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for optimizing memory usage by sharing character instances, reducing the overhead of creating multiple identical objects in scenarios with high character usage.

214. State Pattern with File Transfer Protocols

- **Code Example**: Manages transfer states like connecting, transferring.

```
public interface ITransferState { void Transfer(); }

public class ConnectingState : ITransferState
{
```

```
    public void Transfer() => Console.WriteLine("Connecting...");
}

public class TransferringState : ITransferState
{
    public void Transfer() => Console.WriteLine("Transferring data...");
}

public class FileTransfer
{
    private ITransferState _currentState;

    public FileTransfer() => _currentState = new ConnectingState();
    public void SetState(ITransferState state) => _currentState = state;
    public void Transfer() => _ currentState. Transfer();
}

var transfer = new FileTransfer();
transfer.Transfer(); // Handles transfer based on state
```

- **Use Case**: Useful for managing the states of a file transfer process, allowing transitions between different stages like connecting, transferring, and completing based on protocol requirements.

215. Strategy Pattern with Sorting Orders

- **Code Example**: Implements different sorting orders.

```
public interface ISortStrategy { void Sort(int[] array); }

public class AscendingSort : ISortStrategy
{
    public void Sort(int[] array)
    {
        Array.Sort(array);
        Console.WriteLine("Sorted in ascending order");
    }
}

public class DescendingSort : ISortStrategy
{
    public void Sort(int[] array)
    {
        Array.Sort(array);
        Array.Reverse(array);
        Console.WriteLine("Sorted in descending order");
    }
}

public class Sorter
{
    private ISortStrategy _strategy;

    public void SetStrategy(ISortStrategy strategy) => _strategy = strategy;
    public void Sort(int[] array)
    {
        int[] copy = (int[])array.Clone();
        _strategy.Sort(copy);
        Console.WriteLine(string.Join(", ", copy));
    }
}

var sorter = new Sorter();
sorter.SetStrategy(new DescendingSort());
int[] data = { 3, 1, 4, 1, 5 };
sorter.Sort(data); // Sorts in descending order
```

- **Use Case**: Useful for selecting different sorting orders based on user preference or application requirements, providing flexibility in how data is organized and presented.

216. Template Method Pattern with Command-Line Tools

- **Code Example**: Defines a command-line tool process.

```
public abstract class CommandLineTool
```

```
{
    public void Execute()
    {
        ParseArguments();
        RunCommand();
        Cleanup();
    }

    protected abstract void ParseArguments();
    protected abstract void RunCommand();
    protected abstract void Cleanup();
}

public class FileProcessor : CommandLineTool
{
    protected override void ParseArguments() => Console.WriteLine("Parsing command-line arguments");
    protected override void RunCommand() => Console.WriteLine("Processing files");
    protected override void Cleanup() => Console.WriteLine("Cleaning up temporary files");
}

var tool = new FileProcessor();
tool.Execute(); // Executes the entire process
```

- **Use Case**: Useful for creating consistent and repeatable command-line tools, allowing different implementations to fit into the same structure while handling their specific argument parsing, execution, and cleanup processes.

217. Composite Pattern with Inventory Management

- **Code Example**: Composes items into an inventory structure.

```
public interface IInventoryItem { void Display(); }

public class Product : IInventoryItem
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine("Product: " + Name);
}

public class Inventory : IInventoryItem
{
    private readonly List<IInventoryItem> _items = new();

    public void Display()
    {
        Console.WriteLine("Inventory contents");
        foreach (var item in _items)
            item.Display();
    }

    public void Add(IInventoryItem item) => _items.Add(item);
}

var inventory = new Inventory();
inventory.Add(new Product { Name = "Laptop" });
inventory.Display(); // Displays all items
```

- **Use Case**: Useful for managing inventory by treating individual products and groups of products uniformly, allowing efficient display and manipulation of the entire collection.

218. Decorator Pattern with Error Handling

- **Code Example**: Decorates methods to add error handling.

```
public interface IService { void DoWork(); }

public class BasicService : IService
{
    public void DoWork() => Console.WriteLine("Performing work");
}

public class ErrorHandlingDecorator : IService
{
    private readonly IService _service;
```

```
    public ErrorHandlingDecorator(IService service) => _service = service;
    public void DoWork()
    {
        try
        {
            _service.DoWork();
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error occurred: " + ex.Message);
        }
    }
}

var service = new BasicService();
service = new ErrorHandlingDecorator(service);
service.DoWork(); // Handles errors gracefully
```

- **Use Case**: Useful for adding error handling to methods without altering their core functionality, improving robustness and providing better debugging information.

219. Proxy Pattern with Remote Control Systems

- **Code Example**: Provides a proxy to control remote devices.

```
public interface IRemoteControl { void PowerOn(); }

public class Television : IRemoteControl
{
    public void PowerOn() => Console.WriteLine("Television powered on");
}

public class RemoteControlProxy : IRemoteControl
{
    private Television _television;

    public void PowerOn()
    {
        if (_television == null)
            _television = new Television();
        _television.PowerOn();
    }
}

var remote = new RemoteControlProxy();
remote.PowerOn(); // Powers on television through proxy
```

- **Use Case**: Useful for controlling remote devices by providing a local interface that abstracts the complexity of direct communication, allowing seamless interaction with hardware across network boundaries.

220. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
```

```
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory consumption by sharing character instances, which is particularly beneficial in applications where characters are used extensively, such as text editors or rendering engines.

221. State Pattern with Traffic Lights

- **Code Example**: Manages traffic light states like red, green.

```
public interface ITrafficLightState { void Change(); }

public class RedState : ITrafficLightState
{
    public void Change() => Console.WriteLine("Changing to green light");
}

public class GreenState : ITrafficLightState
{
    public void Change() => Console.WriteLine("Changing to red light");
}

public class TrafficLight
{
    private ITrafficLightState _currentState;

    public TrafficLight() => _currentState = new RedState();
    public void SetState(ITrafficLightState state) => _currentState = state;
    public void Change() => _ currentState. Change();
}

var light = new TrafficLight();
light.Change(); // Changes state based on current state
```

- **Use Case**: Useful for managing the states of traffic lights, allowing transitions between different light colors based on predefined rules or sensors.

222. Strategy Pattern with Compression Algorithms

- **Code Example**: Implements different compression strategies.

```
public interface ICompressionStrategy { byte[] Compress(byte[] data); }

public class ZIP : ICompressionStrategy
{
    public byte[] Compress(byte[] data) => new byte[] { 0x1A, 0x2B }; // Simplified compression
}

public class GZIP : ICompressionStrategy
{
    public byte[] Compress(byte[] data) => new byte[] { 0x3C, 0x4D }; // Simplified compression
}

public class Compressor
{
    private ICompressionStrategy _strategy;

    public void SetStrategy(ICompressionStrategy strategy) => _strategy = strategy;
    public byte[] Compress(byte[] data) => _strategy.Compress(data);
}

var compressor = new Compressor();
compressor.SetStrategy(new GZIP());
byte[] compressed = compressor.Compress(Encoding.ASCII.GetBytes("Hello"));
Console.WriteLine(BitConverter.ToString(compressed)); // Outputs compressed bytes
```

- **Use Case**: Useful for selecting different compression algorithms based on file type, size, or desired compression ratio to optimize storage and transmission efficiency.

223. Template Method Pattern with Build Systems

- **Code Example**: Defines a build system process.

```
public abstract class BuildSystem
```

```
{
    public void Build()
    {
        Clean();
        Compile();
        Test();
    }

    protected abstract void Clean();
    protected abstract void Compile();
    protected abstract void Test();
}

public class CSharpBuildSystem : BuildSystem
{
    protected override void Clean() => Console.WriteLine("Cleaning C# project");
    protected override void Compile() => Console.WriteLine("Compiling C# code");
    protected override void Test() => Console.WriteLine("Running unit tests");
}

var build = new CSharpBuildSystem();
build.Build(); // Executes the entire process
```

- **Use Case**: Useful for creating consistent and repeatable build systems, allowing different programming languages or project types to fit into the same structure while handling their specific cleaning, compiling, and testing processes.

224. Composite Pattern with Folder Structures

- **Code Example**: Composes files and folders into a directory structure.

```
public interface IDirectoryItem { void Display(); }

public class File : IDirectoryItem
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine("File: " + Name);
}

public class Directory : IDirectoryItem
{
    private readonly List<IDirectoryItem> _items = new();

    public void Display()
    {
        Console.WriteLine("Directory contents");
        foreach (var item in _items)
            item.Display();
    }

    public void Add(IDirectoryItem item) => _items.Add(item);
}

var directory = new Directory();
directory.Add(new File { Name = "document.txt" });
directory.Display(); // Displays all items
```

- **Use Case**: Useful for managing file systems by treating individual files and directories uniformly, allowing efficient display and manipulation of the entire structure in a hierarchical manner.

225. Decorator Pattern with Performance Monitoring

- **Code Example**: Decorates methods to monitor their performance.

```
public interface IService { void DoWork(); }

public class BasicService : IService
{
    public void DoWork()
    {
        Thread.Sleep(500); // Simulate work
        Console.WriteLine("Work completed");
    }
}
```

```
public class PerformanceDecorator : IService
{
    private readonly IService _service;

    public PerformanceDecorator(IService service) => _service = service;
    public void DoWork()
    {
        Stopwatch stopwatch = new();
        stopwatch.Start();

        _service.DoWork();

        stopwatch.Stop();
        Console.WriteLine("Time taken: " + stopwatch.ElapsedMilliseconds + "ms");
    }
}

var service = new BasicService();
service = new PerformanceDecorator(service);
service.DoWork(); // Monitors performance
```

- **Use Case**: Useful for adding performance monitoring to methods without altering their core functionality, aiding in identifying bottlenecks and optimizing code execution.

226. Proxy Pattern with Lazy Initialization

- **Code Example**: Provides a proxy for lazy initialization of objects.

```
public interface IHeavyObject { void DoWork(); }

public class HeavyObject : IHeavyObject
{
    public HeavyObject()
    {
        Console.WriteLine("Heavy object initialized");
        // Simulate heavy initialization
        Thread.Sleep(2000);
    }

    public void DoWork() => Console.WriteLine("Heavy object working");
}

public class Proxy : IHeavyObject
{
    private HeavyObject _heavyObject;

    public void DoWork()
    {
        if (_heavyObject == null)
            _heavyObject = new HeavyObject();
        _heavyObject.DoWork();
    }
}

var proxy = new Proxy();
proxy.DoWork(); // Initializes and works heavy object
```

- **Use Case**: Useful for deferring the initialization of heavy or resource-intensive objects until they are actually needed, improving application startup time and reducing initial memory consumption.

227. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }
```

```
        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage by sharing instances of frequently used characters, which is particularly beneficial in applications where large amounts of text are processed or displayed.

228. State Pattern with Vending Machines

- **Code Example**: Manages vending machine states like idle, dispensing.

```
public interface IVendingMachineState { void Dispense(); }

public class IdleState : IVendingMachineState
{
    public void Dispense() => Console.WriteLine("Waiting for selection");
}

public class DispensingState : IVendingMachineState
{
    public void Dispense() => Console.WriteLine("Dispensing item");
}

public class VendingMachine
{
    private IVendingMachineState _currentState;

    public VendingMachine() => _currentState = new IdleState();
    public void SetState(IVendingMachineState state) => _currentState = state;
    public void Dispense() => _ currentState. Dispense();
}

var machine = new VendingMachine();
machine.Dispense(); // Handles dispensing based on state
```

- **Use Case**: Useful for managing the states of a vending machine, allowing transitions between idle and dispensing states based on user interactions or sensor inputs.

229. Strategy Pattern with Routing Algorithms

- **Code Example**: Implements different routing strategies.

```
public interface IRoutingStrategy { string GetRoute(string from, string to); }

public class ShortestPath : IRoutingStrategy
{
    public string GetRoute(string from, string to) => "Shortest path from " + from + " to " + to;
}

public class ScenicRoute : IRoutingStrategy
{
    public string GetRoute(string from, string to) => "Scenic route from " + from + " to " + to;
}

public class NavigationSystem
{
    private IRoutingStrategy _strategy;

    public void SetStrategy(IRoutingStrategy strategy) => _strategy = strategy;
    public string GetRoute(string from, string to)
        => _strategy.GetRoute(from, to);
}

var nav = new NavigationSystem();
nav.SetStrategy(new ScenicRoute());
string route = nav.GetRoute("New York", "Los Angeles");
```

```
Console.WriteLine(route); // Outputs scenic route
```

- **Use Case**: Useful for selecting different routing algorithms based on user preference or application requirements, providing flexibility in how paths are determined and displayed.

230. Template Method Pattern with Report Generators

- **Code Example**: Defines a report generation process.

```
public abstract class ReportGenerator
{
    public void GenerateReport()
    {
        CollectData();
        ProcessData();
        OutputReport();
    }

    protected abstract void CollectData();
    protected abstract void ProcessData();
    protected abstract void OutputReport();
}

public class SalesReport : ReportGenerator
{
    protected override void CollectData() => Console.WriteLine("Collecting sales data");
    protected override void ProcessData() => Console.WriteLine("Processing sales data");
    protected override void OutputReport() => Console.WriteLine("Generating sales report");
}

var report = new SalesReport();
report.GenerateReport(); // Executes the entire process
```

- **Use Case**: Useful for creating consistent and repeatable report generators, allowing different data sources or processing methods to fit into the same structure while handling their specific collection, processing, and output steps.

231. Composite Pattern with Music Albums

- **Code Example**: Composes tracks into an album structure.

```
public interface IMusicTrack { void Play(); }

public class Song : IMusicTrack
{
    public string Title { get; set; }
    public void Play() => Console.WriteLine("Playing song: " + Title);
}

public class Album : IMusicTrack
{
    private readonly List<IMusicTrack> _tracks = new();

    public void Play()
    {
        Console.WriteLine("Playing album");
        foreach (var track in _tracks)
            track.Play();
    }

    public void Add(IMusicTrack track) => _tracks.Add(track);
}

var album = new Album();
album.Add(new Song { Title = "Imagine" });
album.Play(); // Plays all tracks in the album
```

- **Use Case**: Useful for managing music albums by treating individual tracks and albums uniformly, allowing efficient playback and manipulation of the entire collection as a single unit.

232. Decorator Pattern with Data Transformation

- **Code Example**: Decorates data to add transformation logic.

```
public interface IDataProcessor { string Process(string input); }

public class BasicProcessor : IDataProcessor
{
    public string Process(string input) => input.ToUpper();
}

public class TransformationDecorator : IDataProcessor
{
    private readonly IDataProcessor _processor;

    public TransformationDecorator(IDataProcessor processor) => _processor = processor;
    public string Process(string input)
    {
        string transformed = _processor.Process(input);
        return $"Transformed: {transformed}";
    }
}

var processor = new BasicProcessor();
processor = new TransformationDecorator(processor);
string result = processor.Process("hello");
Console.WriteLine(result); // Outputs transformed uppercase
```

- **Use Case**: Useful for adding data transformation logic to processors without altering their core functionality, providing flexibility in how data is manipulated and presented.

233. Proxy Pattern with Remote Database Access

- **Code Example**: Provides a proxy to access a remote database.

```
public interface IDatabase { string Query(string sql); }

public class RemoteDatabase : IDatabase
{
    public string Query(string sql)
    {
        // Simulate remote database access
        Thread.Sleep(1000);
        return "Result from SQL: " + sql;
    }
}

public class DatabaseProxy : IDatabase
{
    private RemoteDatabase _remoteDb;

    public string Query(string sql)
    {
        if (_remoteDb == null)
            _remoteDb = new RemoteDatabase();
        return _remoteDb.Query(sql);
    }
}

var db = new DatabaseProxy();
string result = db.Query("SELECT * FROM Users");
Console.WriteLine(result); // Outputs query result
```

- **Use Case**: Useful for providing a local interface to access remote databases, abstracting the complexity of direct communication and improving application performance by deferring database connections until necessary.

234. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
```

```
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage by sharing instances of frequently used characters, which is particularly beneficial in applications involving extensive text operations or display.

235. State Pattern with Web Servers

- **Code Example**: Manages web server states like starting, running.

```
public interface IWebServerState { void Start(); }

public class StartingState : IWebServerState
{
    public void Start() => Console.WriteLine("Starting web server");
}

public class RunningState : IWebServerState
{
    public void Start() => Console.WriteLine("Web server already running");
}

public class WebServer
{
    private IWebServerState _currentState;

    public WebServer() => _currentState = new StartingState();
    public void SetState(IWebServerState state) => _currentState = state;
    public void Start() => _ currentState. Start();
}

var server = new WebServer();
server.Start(); // Handles starting based on state
```

- **Use Case**: Useful for managing the states of a web server, allowing transitions between starting and running states based on system status or configuration changes.

236. Strategy Pattern with Payment Methods

- **Code Example**: Implements different payment strategies.

```
public interface IPaymentStrategy { void Pay(decimal amount); }

public class CreditCard : IPaymentStrategy
{
    public void Pay(decimal amount) => Console.WriteLine("Paid with credit card: " + amount);
}

public class PayPal : IPaymentStrategy
{
    public void Pay(decimal amount) => Console.WriteLine("Paid with PayPal: " + amount);
}

public class PaymentProcessor
{
    private IPaymentStrategy _strategy;

    public void SetStrategy(IPaymentStrategy strategy) => _strategy = strategy;
    public void Pay(decimal amount)
        => _strategy.Pay(amount);
}

var processor = new PaymentProcessor();
```

```
processor.SetStrategy(new PayPal());
processor.Pay(100.00m); // Pays using PayPal
```

- **Use Case**: Useful for selecting different payment methods based on user preference or transaction requirements, providing flexible and secure options for financial transactions.

237. Template Method Pattern with Testing Frameworks

- **Code Example**: Defines a testing framework process.

```
public abstract class TestFramework
{
    public void RunTests()
    {
        SetUp();
        ExecuteTests();
        TearDown();
    }

    protected abstract void SetUp();
    protected abstract void ExecuteTests();
    protected abstract void TearDown();
}

public class UnitTestFramework : TestFramework
{
    protected override void SetUp() => Console.WriteLine("Setting up unit tests");
    protected override void ExecuteTests() => Console.WriteLine("Running unit tests");
    protected override void TearDown() => Console.WriteLine("Tearing down unit tests");
}

var framework = new UnitTestFramework();
framework.RunTests(); // Executes the entire process
```

- **Use Case**: Useful for creating consistent and repeatable testing frameworks, allowing different test types or configurations to fit into the same structure while handling their specific setup, execution, and teardown processes.

238. Composite Pattern with Network Devices

- **Code Example**: Composes devices into a network structure.

```
public interface INetworkDevice { void Connect(); }

public class Router : INetworkDevice
{
    public void Connect() => Console.WriteLine("Connecting router");
}

public class Network : INetworkDevice
{
    private readonly List<INetworkDevice> _devices = new();

    public void Connect()
    {
        Console.WriteLine("Connecting network");
        foreach (var device in _devices)
            device.Connect();
    }

    public void Add(INetworkDevice device) => _devices.Add(device);
}

var network = new Network();
network.Add(new Router());
network.Connect(); // Connects all devices
```

- **Use Case**: Useful for managing networks by treating individual devices and groups of devices uniformly, allowing efficient connection and manipulation of the entire network structure.

239. Decorator Pattern with Logging

- **Code Example**: Decorates methods to add logging functionality.

```
public interface IService { void DoWork(); }

public class BasicService : IService
{
    public void DoWork() => Console.WriteLine("Performing work");
}

public class LoggingDecorator : IService
{
    private readonly IService _service;

    public LoggingDecorator(IService service) => _service = service;
    public void DoWork()
    {
        Console.WriteLine("Starting work");
        _service.DoWork();
        Console.WriteLine("Work completed");
    }
}

var service = new BasicService();
service = new LoggingDecorator(service);
service.DoWork(); // Logs work process
```

- **Use Case**: Useful for adding logging to methods without altering their core functionality, providing visibility into method execution and aiding in debugging and auditing.

240. Proxy Pattern with File Access Control

- **Code Example**: Provides a proxy to control file access.

```
public interface IFileAccess { void Open(string path); }

public class File : IFileAccess
{
    public void Open(string path)
    {
        // Simulate file access check
        Console.WriteLine("Accessing file: " + path);
    }
}

public class FileAccessProxy : IFileAccess
{
    private File _file;

    public void Open(string path)
    {
        if (IsAuthorized())
            _file?.Open(path);
        else
            Console.WriteLine("Access denied");
    }

    private bool IsAuthorized() => true; // Simplified check

    public void Initialize()
    {
        if (_file == null)
            _file = new File();
    }
}

var proxy = new FileAccessProxy();
proxy.Initialize();
proxy.Open("secret.txt"); // Controls file access
```

- **Use Case**: Useful for enforcing file access control by intercepting requests through a proxy, preventing unauthorized access and ensuring data security.

241. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for optimizing memory usage by sharing character instances, which is particularly beneficial in scenarios with high text processing or display where multiple identical characters are used.

242. State Pattern with Game Levels

- **Code Example**: Manages game level states like playing, paused.

```
public interface IGameLevelState { void Play(); }

public class PlayingState : IGameLevelState
{
    public void Play() => Console.WriteLine("Playing game level");
}

public class PausedState : IGameLevelState
{
    public void Play() => Console.WriteLine("Resuming game level");
}

public class GameLevel
{
    private IGameLevelState _currentState;

    public GameLevel() => _currentState = new PlayingState();
    public void SetState(IGameLevelState state) => _currentState = state;
    public void Play() => _ currentState. Play();
}

var level = new GameLevel();
level.Play(); // Handles play based on state
```

- **Use Case**: Useful for managing the states of a game level, allowing transitions between playing and paused states based on user input or in-game events.

243. Strategy Pattern with Rating Systems

- **Code Example**: Implements different rating strategies.

```
public interface IRatingStrategy { int CalculateRating(int score); }

public class SimpleRating : IRatingStrategy
{
    public int CalculateRating(int score) => score / 10;
}

public class WeightedRating : IRatingStrategy
{
    public int CalculateRating(int score) => score * 2;
}

public class RatingCalculator
{
```

```
    private IRatingStrategy _strategy;

    public void SetStrategy(IRatingStrategy strategy) => _strategy = strategy;
    public int CalculateRating(int score)
        => _strategy.CalculateRating(score);
}

var calculator = new RatingCalculator();
calculator.SetStrategy(new WeightedRating());
int rating = calculator.CalculateRating(50);
Console.WriteLine(rating); // Outputs weighted rating
```

- **Use Case**: Useful for selecting different rating calculation methods based on system requirements or user preferences, providing flexibility in how scores are transformed into ratings.

244. Template Method Pattern with GUI Frameworks

- **Code Example**: Defines a GUI framework process.

```
public abstract class GUIFramework
{
    public void Render()
    {
        InitializeComponents();
        DrawElements();
        UpdateScreen();
    }

    protected abstract void InitializeComponents();
    protected abstract void DrawElements();
    protected abstract void UpdateScreen();
}

public class MainWindow : GUIFramework
{
    protected override void InitializeComponents() => Console.WriteLine("Initializing main window
components");
    protected override void DrawElements() => Console.WriteLine("Drawing main window elements");
    protected override void UpdateScreen() => Console.WriteLine("Updating main window screen");
}

var gui = new MainWindow();
gui.Render(); // Executes the entire process
```

- **Use Case**: Useful for creating consistent and repeatable GUI frameworks, allowing different components or rendering methods to fit into the same structure while handling their specific initialization, drawing, and updating processes.

245. Composite Pattern with Organizational Charts

- **Code Example**: Composes employees into an organizational structure.

```
public interface IOrganizationItem { void Display(); }

public class Employee : IOrganizationItem
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine("Employee: " + Name);
}

public class Department : IOrganizationItem
{
    private readonly List<IOrganizationItem> _members = new();

    public void Display()
    {
        Console.WriteLine("Department members");
        foreach (var member in _members)
            member.Display();
    }

    public void Add(IOrganizationItem member) => _members.Add(member);
}

var department = new Department();
```

```
department.Add(new Employee { Name = "John Doe" });
department.Display(); // Displays all members
```

- **Use Case**: Useful for managing organizational charts by treating individual employees and departments uniformly, allowing efficient display and manipulation of the entire structure in a hierarchical manner.

246. Decorator Pattern with Cache Optimization

- **Code Example**: Decorates data access to add caching.

```
public interface IDataAccess { string GetData(); }

public class Database : IDataAccess
{
    public string GetData()
    {
        // Simulate database access
        Console.WriteLine("Fetching data from database");
        return "Data from DB";
    }
}

public class CacheDecorator : IDataAccess
{
    private Database _database;
    private string _cachedData;

    public string GetData()
    {
        if (string.IsNullOrEmpty(_cachedData))
        {
            _cachedData = _database.GetData();
            Console.WriteLine("Caching data");
        }
        return _cachedData;
    }

    public CacheDecorator(Database database) => _database = database;
}

var db = new Database();
var cache = new CacheDecorator(db);
Console.WriteLine(cache.GetData()); // Caches and retrieves data
```

- **Use Case**: Useful for optimizing data access by caching frequently retrieved data, reducing redundant database queries and improving application performance.

247. Proxy Pattern with Print Job Management

- **Code Example**: Provides a proxy to manage print jobs.

```
public interface IPrinter { void Print(string document); }

public class Printer : IPrinter
{
    public void Print(string document)
    {
        // Simulate printing process
        Console.WriteLine("Printing document: " + document);
        Thread.Sleep(2000);
    }
}

public class PrintJobProxy : IPrinter
{
    private Printer _printer;

    public void Print(string document)
    {
        if (_printer == null)
            _printer = new Printer();
        _printer.Print(document);
    }
}
```

```
var proxy = new PrintJobProxy();
proxy.Print("Report.pdf"); // Manages print job through proxy
```

- **Use Case**: Useful for managing print jobs by providing a local interface to printers, abstracting the complexity of direct communication and allowing for queue management or job prioritization.

248. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage by sharing character instances, which is particularly beneficial in applications where characters are used extensively, such as word processors or graphical text displays.

249. State Pattern with Mobile Phone Modes

- **Code Example**: Manages mobile phone modes like normal, silent.

```
public interface IPhoneMode { void SetRingerVolume(int volume); }

public class NormalMode : IPhoneMode
{
    public void SetRingerVolume(int volume) => Console.WriteLine("Ringer volume set to normal: " +
volume);
}

public class SilentMode : IPhoneMode
{
    public void SetRingerVolume(int volume) => Console.WriteLine("Ringer volume set to silent");
}

public class MobilePhone
{
    private IPhoneMode _currentState;

    public MobilePhone() => _currentState = new NormalMode();
    public void SetState(IPhoneMode state) => _currentState = state;
    public void SetRingerVolume(int volume)
        => _ currentState. SetRingerVolume(volume);
}

var phone = new MobilePhone();
phone.SetState(new SilentMode());
phone.SetRingerVolume(10); // Handles volume based on state
```

- **Use Case**: Useful for managing the modes of a mobile phone, allowing transitions between normal and silent states based on user settings or environmental conditions.

250. Strategy Pattern with Encryption Methods

- **Code Example**: Implements different encryption strategies.

```
public interface IEncryptionStrategy { string Encrypt(string data); }

public class AES : IEncryptionStrategy
{
    public string Encrypt(string data) => "AES encrypted: " + data;
}

public class RSA : IEncryptionStrategy
{
    public string Encrypt(string data) => "RSA encrypted: " + data;
}

public class EncryptionManager
{
    private IEncryptionStrategy _strategy;

    public void SetStrategy(IEncryptionStrategy strategy) => _strategy = strategy;
    public string Encrypt(string data)
        => _strategy.Encrypt(data);
}

var manager = new EncryptionManager();
manager.SetStrategy(new RSA());
string encrypted = manager.Encrypt("Sensitive Data");
Console.WriteLine(encrypted); // Outputs RSA encrypted data
```

- **Use Case**: Useful for selecting different encryption algorithms based on security requirements or data sensitivity, providing robust and flexible options for protecting information.

251. Template Method Pattern with Build Pipelines

- **Code Example**: Defines a build pipeline process.

```
public abstract class BuildPipeline
{
    public void Run()
    {
        FetchCode();
        RestoreDependencies();
        Compile();
        Test();
        Deploy();
    }

    protected abstract void FetchCode();
    protected abstract void RestoreDependencies();
    protected abstract void Compile();
    protected abstract void Test();
    protected abstract void Deploy();
}

public class CIPIPELINE : BuildPipeline
{
    protected override void FetchCode() => Console.WriteLine("Fetching code from repository");
    protected override void RestoreDependencies() => Console.WriteLine("Restoring dependencies");
    protected override void Compile() => Console.WriteLine("Compiling code");
    protected override void Test() => Console.WriteLine("Running tests");
    protected override void Deploy() => Console.WriteLine("Deploying to production");
}

var pipeline = new CIPIPELINE();
pipeline.Run(); // Executes the entire process
```

- **Use Case**: Useful for creating consistent and repeatable build pipelines, allowing different steps or configurations to fit into the same structure while handling their specific fetching, restoring, compiling, testing, and deploying processes.

252. Composite Pattern with Library Catalogs

- **Code Example**: Composes books into a library structure.

```
public interface ILibraryItem { void Display(); }

public class Book : ILibraryItem
{
```

```
    public string Title { get; set; }
    public void Display() => Console.WriteLine("Book: " + Title);
}

public class Catalog : ILibraryItem
{
    private readonly List<ILibraryItem> _items = new();

    public void Display()
    {
        Console.WriteLine("Catalog contents");
        foreach (var item in _items)
            item.Display();
    }

    public void Add(ILibraryItem item) => _items.Add(item);
}

var catalog = new Catalog();
catalog.Add(new Book { Title = "1984" });
catalog.Display(); // Displays all items
```

- **Use Case**: Useful for managing library catalogs by treating individual books and collections of books uniformly, allowing efficient display and manipulation of the entire catalog in a hierarchical manner.

253. Decorator Pattern with Compression

- **Code Example**: Decorates data to add compression functionality.

```
public interface IDataWriter { void Write(string data); }

public class FileWriter : IDataWriter
{
    public void Write(string data) => Console.WriteLine("Writing to file: " + data);
}

public class CompressedWriterDecorator : IDataWriter
{
    private FileWriter _fileWriter;

    public void Write(string data)
    {
        string compressed = Compress(data);
        _fileWriter.Write(compressed);
    }

    private string Compress(string data) => "Compressed: " + data;

    public CompressedWriterDecorator(FileWriter fileWriter) => _fileWriter = fileWriter;
}

var writer = new FileWriter();
writer = new CompressedWriterDecorator(writer);
writer.Write("Hello, World!"); // Writes compressed data
```

- **Use Case**: Useful for adding compression to data writers without altering their core functionality, improving storage efficiency and reducing transmission times.

254. Proxy Pattern with Remote Process Control

- **Code Example**: Provides a proxy to control remote processes.

```
public interface IRemoteProcess { void Start(); }

public class RemoteService : IRemoteProcess
{
    public void Start()
    {
        // Simulate remote process start
        Console.WriteLine("Starting remote service");
        Thread.Sleep(1000);
    }
}
```

```
public class ProcessProxy : IRemoteProcess
{
    private RemoteService _remoteService;

    public void Start()
    {
        if (_remoteService == null)
            _remoteService = new RemoteService();
        _remoteService.Start();
    }
}

var proxy = new ProcessProxy();
proxy.Start(); // Controls remote process through proxy
```

- **Use Case**: Useful for providing a local interface to control remote processes, abstracting the complexity of direct communication and allowing for secure and efficient management of distributed systems.

255. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory consumption by sharing instances of frequently used characters, which is particularly beneficial in applications where large volumes of text are processed or displayed.

256. State Pattern with Air Conditioners

- **Code Example**: Manages air conditioner states like cooling, heating.

```
public interface IAirConditionerState { void AdjustTemperature(int temp); }

public class CoolingState : IAirConditionerState
{
    public void AdjustTemperature(int temp) => Console.WriteLine("Cooling to " + temp);
}

public class HeatingState : IAirConditionerState
{
    public void AdjustTemperature(int temp) => Console.WriteLine("Heating to " + temp);
}

public class AirConditioner
{
    private IAirConditionerState _currentState;

    public AirConditioner() => _currentState = new CoolingState();
    public void SetState(IAirConditionerState state) => _currentState = state;
    public void AdjustTemperature(int temp)
        => _ currentState. AdjustTemperature(temp);
}

var ac = new AirConditioner();
```

```
ac.SetState(new HeatingState());
ac.AdjustTemperature(70); // Adjusts temperature based on state
```

- **Use Case**: Useful for managing the states of an air conditioner, allowing transitions between cooling and heating states based on environmental conditions or user preferences.

257. Strategy Pattern with Sorting Algorithms

- **Code Example**: Implements different sorting strategies.

```
public interface ISortStrategy { int[] Sort(int[] data); }

public class BubbleSort : ISortStrategy
{
    public int[] Sort(int[] data)
    {
        // Simplified bubble sort implementation
        for (int i = 0; i < data.Length; i++)
            for (int j = 0; j < data.Length - i - 1; j++)
                if (data[j] > data[j + 1])
                    Swap(ref data[j], ref data[j + 1]);
        return data;
    }

    private void Swap(ref int a, ref int b)
    {
        int temp = a;
        a = b;
        b = temp;
    }
}

public class QuickSort : ISortStrategy
{
    public int[] Sort(int[] data)
    {
        // Simplified quick sort implementation
        QuickSortHelper(data, 0, data.Length - 1);
        return data;
    }

    private void QuickSortHelper(int[] array, int low, int high)
    {
        if (low < high)
        {
            int pivotIndex = Partition(array, low, high);
            QuickSortHelper(array, low, pivotIndex - 1);
            QuickSortHelper(array, pivotIndex + 1, high);
        }
    }

    private int Partition(int[] array, int low, int high)
    {
        int pivot = array[high];
        int i = low - 1;

        for (int j = low; j < high; j++)
            if (array[j] <= pivot)
                Swap(ref array[++i], ref array[j]);

        Swap(ref array[i + 1], ref array[high]);
        return i + 1;
    }

    private void Swap(ref int a, ref int b)
    {
        int temp = a;
        a = b;
        b = temp;
    }
}

public class Sorter
{
    private ISortStrategy _strategy;
```

```
    public void SetStrategy(ISortStrategy strategy) => _strategy = strategy;
    public int[] Sort(int[] data)
        => _strategy.Sort(data);
}

var sorter = new Sorter();
sorter.SetStrategy(new QuickSort());
int[] result = sorter.Sort(new int[] { 5, 3, 8, 1 });
Console.WriteLine(string.Join(", ", result)); // Outputs sorted array
```

- **Use Case**: Useful for selecting different sorting algorithms based on performance requirements or data size, providing efficient and flexible options for organizing collections of elements.

258. Template Method Pattern with Document Processing

- **Code Example**: Defines a document processing framework.

```
public abstract class DocumentProcessor
{
    public void ProcessDocument()
    {
        LoadDocument();
        ValidateFormat();
        ApplyTransformations();
        SaveDocument();
    }

    protected abstract void LoadDocument();
    protected abstract void ValidateFormat();
    protected abstract void ApplyTransformations();
    protected abstract void SaveDocument();
}

public class PDFProcessor : DocumentProcessor
{
    protected override void LoadDocument() => Console.WriteLine("Loading PDF document");
    protected override void ValidateFormat() => Console.WriteLine("Validating PDF format");
    protected override void ApplyTransformations() => Console.WriteLine("Applying transformations to
PDF");
    protected override void SaveDocument() => Console.WriteLine("Saving processed PDF");
}

var processor = new PDFProcessor();
processor.ProcessDocument(); // Executes the entire process
```

- **Use Case**: Useful for creating consistent and repeatable document processing frameworks, allowing different document types or transformations to fit into the same structure while handling their specific loading, validation, transformation, and saving steps.

259. Composite Pattern with Inventory Management

- **Code Example**: Composes items into an inventory structure.

```
public interface IInventoryItem { void Display(); }

public class Product : IInventoryItem
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine("Product: " + Name);
}

public class Inventory : IInventoryItem
{
    private readonly List<IInventoryItem> _items = new();

    public void Display()
    {
        Console.WriteLine("Inventory contents");
        foreach (var item in _items)
            item.Display();
    }

    public void Add(IInventoryItem item) => _items.Add(item);
}
```

```
var inventory = new Inventory();
inventory.Add(new Product { Name = "Laptop" });
inventory.Display(); // Displays all items
```

- **Use Case**: Useful for managing inventories by treating individual products and groups of products uniformly, allowing efficient display and manipulation of the entire inventory structure.

260. Decorator Pattern with Performance Counters

- **Code Example**: Decorates methods to add performance counters.

```
public interface IPerformanceTest { void RunTest(); }

public class BasicTest : IPerformanceTest
{
    public void RunTest() => Console.WriteLine("Running basic test");
}

public class PerformanceCounterDecorator : IPerformanceTest
{
    private BasicTest _test;
    private int _calls = 0;

    public void RunTest()
    {
        _calls++;
        Console.WriteLine("Performance counter: " + _calls);
        _test.RunTest();
    }

    public PerformanceCounterDecorator(BasicTest test) => _test = test;
}

var test = new BasicTest();
test = new PerformanceCounterDecorator(test);
test.RunTest(); // Counts and runs test
```

- **Use Case**: Useful for adding performance monitoring to methods without altering their core functionality, providing insights into method execution frequency and aiding in performance optimization.

261. Proxy Pattern with Remote API Access

- **Code Example**: Provides a proxy to access a remote API.

```
public interface IAPI { string GetData(string endpoint); }

public class RemoteAPI : IAPI
{
    public string GetData(string endpoint)
    {
        // Simulate remote API call
        Console.WriteLine("Calling API endpoint: " + endpoint);
        return "Data from " + endpoint;
    }
}

public class APIProxy : IAPI
{
    private RemoteAPI _remoteApi;

    public string GetData(string endpoint)
    {
        if (_remoteApi == null)
            _remoteApi = new RemoteAPI();
        return _remoteApi.GetData(endpoint);
    }
}

var api = new APIProxy();
string result = api.GetData("/users");
Console.WriteLine(result); // Outputs API data
```

- **Use Case**: Useful for providing a local interface to access remote APIs, abstracting the complexity of direct communication

and improving application performance by deferring API calls until necessary.

262. Flyweight Pattern with Character Set

- **Code Example**: Efficiently manages a set of characters.

```
public class CharacterFlyweightFactory
{
    private readonly Dictionary<char, CharacterFlyweight> _flyweights = new();

    public CharacterFlyweight GetCharacter(char c)
        => _flyweights.TryGetValue(c, out var flyweight) ? flyweight : new CharacterFlyweight(c);

    private class CharacterFlyweight
    {
        public char Char { get; }

        public CharacterFlyweight(char c) => Char = c;
    }
}

var factory = new CharacterFlyweightFactory();
var charA = factory.GetCharacter('A');
var charB = factory.GetCharacter('B');

Console.WriteLine(charA.Char); // A
Console.WriteLine(charB.Char); // B
```

- **Use Case**: Useful for reducing memory usage by sharing instances of frequently used characters, which is particularly beneficial in applications involving extensive text operations or display.

263. State Pattern with Traffic Lights

- **Code Example**: Manages traffic light states like red, green.

```
public interface ITrafficLightState { void ChangeColor(); }

public class RedLight : ITrafficLightState
{
    public void ChangeColor() => Console.WriteLine("Changing to green");
}

public class GreenLight : ITrafficLightState
{
    public void ChangeColor() => Console.WriteLine("Changing to red");
}

public class TrafficLight
{
    private ITrafficLightState _currentState;

    public TrafficLight() => _currentState = new RedLight();
    public void SetState(ITrafficLightState state) => _currentState = state;
    public void ChangeColor()
        => _ currentState. ChangeColor();
}

var light = new TrafficLight();
light.ChangeColor(); // Handles color change based on state
```

- **Use Case**: Useful for managing the states of a traffic light, allowing transitions between red and green states based on timing or external signals.

264. Strategy Pattern with Search Algorithms

- **Code Example**: Implements different search strategies.

```
public interface ISearchStrategy { int Search(int[] data, int target); }

public class LinearSearch : ISearchStrategy
{
    public int Search(int[] data, int target)
    {
```

```
        for (int i = 0; i < data.Length; i++)
            if (data[i] == target)
                return i;
        return -1;
    }
}

public class BinarySearch : ISearchStrategy
{
    public int Search(int[] data, int target)
    {
        int left = 0;
        int right = data.Length - 1;

        while (left <= right)
        {
            int mid = left + (right - left) / 2;

            if (data[mid] == target)
                return mid;
            else if (data[mid] < target)
                left = mid + 1;
            else
                right = mid - 1;
        }
        return -1;
    }
}

public class SearchManager
{
    private ISearchStrategy _strategy;

    public void SetStrategy(ISearchStrategy strategy) => _strategy = strategy;
    public int Search(int[] data, int target)
        => _strategy.Search(data, target);
}

var manager = new SearchManager();
manager.SetStrategy(new BinarySearch());
int[] array = { 1, 3, 5, 7, 9 };
int result = manager.Search(array, 5);
Console.WriteLine(result); // Outputs index of target
```

- **Use Case**: Useful for selecting different search algorithms based on data structure or performance requirements, providing efficient and flexible options for finding elements in collections.

265. Template Method Pattern with Email Campaigns

- **Code Example**: Defines an email campaign framework.

```
public abstract class EmailCampaign
{
    public void RunCampaign()
    {
        GatherContacts();
        ComposeMessage();
        SendEmails();
    }

    protected abstract void GatherContacts();
    protected abstract void ComposeMessage();
    protected abstract void SendEmails();
}

public class MarketingCampaign : EmailCampaign
{
    protected override void GatherContacts() => Console.WriteLine("Gathering marketing contacts");
    protected override void ComposeMessage() => Console.WriteLine("Composing marketing message");
    protected override void SendEmails() => Console.WriteLine("Sending marketing emails");
}

var campaign = new MarketingCampaign();
campaign.RunCampaign(); // Executes the entire process
```

- **Use Case**: Useful for creating consistent and repeatable email campaign frameworks, allowing different campaign types or strategies to fit into the same structure while handling their specific contact gathering, message composition, and email sending processes.

266. Composite Pattern with Family Trees

- **Code Example**: Composes family members into a tree structure.

```
public interface IFamilyMember { void Display(); }

public class Person : IFamilyMember
{
    public string Name { get; set; }
    public void Display() => Console.WriteLine("Family member: " + Name);
}

public class FamilyTree : IFamilyMember
{
    private readonly List<IFamilyMember> _members = new();

    public void Display()
    {
        Console.WriteLine("Family tree members");
        foreach (var member in _members)
            member.Display();
    }

    public void Add(IFamilyMember member) => _members.Add(member);
}

var family = new FamilyTree();
family.Add(new Person { Name = "John" });
family.Display(); // Displays all members
```

- **Use Case**: Useful for managing family trees by treating individual members and groups of members uniformly, allowing efficient display and manipulation of the entire tree structure.

267. Decorator Pattern with Data Validation

- **Code Example**: Decorates data access to add validation.

```
public interface IDataAccess { string GetData(); }

public class Database : IDataAccess
{
    public string GetData()
    {
        // Simulate database access
        return "Data from DB";
    }
}

public class ValidationDecorator : IDataAccess
{
    private Database _database;

    public string GetData()
    {
        string data = _database.GetData();
        if (IsValid(data))
            return data;
        else
            throw new InvalidOperationException("Data is invalid");
    }

    private bool IsValid(string data) => !string.IsNullOrEmpty(data);

    public ValidationDecorator(Database database) => _database = database;
}

var db = new Database();
var validator = new ValidationDecorator(db);
try
{
```

```
    Console.WriteLine(validator.GetData()); // Validates and retrieves data
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

- **Use Case**: Useful for adding data validation to accessors without altering their core functionality, ensuring data integrity and preventing invalid operations.