Prototype

Il pattern **Prototype** permette di creare nuovi oggetti clonando un **prototipo** esistente invece di istanziarli da zero. Immagina di avere un oggetto configurato e volerne altri simili: con Prototype il **client** può ottenere nuove istanze senza conoscere i dettagli di creazione, copiando direttamente un oggetto già inizializzato. Questo è utile quando la creazione di un oggetto è costosa o complessa, oppure quando vuoi mantenere un certo stato di default in ogni copia.

La chiave del Prototype è definire un metodo di **clonazione** (ad esempio `Clone()`) in un'interfaccia o classe base, implementato poi dalle classi concrete per restituire una copia di se stesse. A runtime il client invoca `Clone()` sul prototipo invece di usare `new`, ottenendo un nuovo oggetto con lo stesso stato. Bisogna fare attenzione a clonare correttamente gli oggetti complessi: se contengono riferimenti ad altri oggetti interni, potrebbe servire una **clonazione profonda** (*deep copy*) per duplicarli interamente, altrimenti con una **clonazione superficiale** (*shallow copy*) il clone condividerà parti di stato col prototipo originale.

```
abstract class Shape {
    public abstract Shape Clone();
    /* metodo prototipo: definisce l'interfaccia per clonare */
}

class Circle : Shape {
    public int Radius;
    public Circle(int r) { Radius = r; }
    public override Shape Clone() {
        // ConcretePrototype: crea una nuova istanza copiando lo stato
        return new Circle(this.Radius);
    }
}

// Utilizzo del Prototype:
Circle cerchio1 = new Circle(5);
Circle cerchio2 = (Circle) cerchio1.Clone();
Console.WriteLine(cerchio2.Radius);  // Output: 5 (cerchio2 ha lo stesso
raggio di cerchio1)
```

Builder

Il pattern **Builder** aiuta a costruire un **oggetto complesso** passo per passo, separando la logica di costruzione dalla rappresentazione finale. Quando la creazione di un oggetto richiede molti passi (ad esempio impostare vari campi o sottocomponenti), Builder permette di aggiungere configurazioni gradualmente invece di avere costruttori con troppi parametri. Il codice diventa più leggibile: è come seguire una “ricetta” in cui aggiungiamo ingredienti uno alla volta, utile per oggetti come configurazioni, assemblaggi di parti (es. costruire un computer o una pizza personalizzata) senza dover ricordare l'ordine esatto di tutti i parametri.

In pratica, il Builder espone metodi per configurare o aggiungere parti dell'oggetto e un metodo finale (ad esempio `Build()` o `Crea`) per ottenere l'oggetto costruito. Spesso i metodi del builder restituiscono `this` in modo da poterli concatenare fluentemente (**interfaccia fluente**), ad esempio `builder.AggiungiX(...).ImpostaY(...)` ecc. Ciò rende il codice simile a un linguaggio naturale e facile da seguire. Il pattern è utile anche perché lo stesso processo di costruzione può creare varianti differenti dell'oggetto: basta cambiare il builder o alcuni passi, mantenendo invariato il codice client che dirige la costruzione.

```
class Pizza {
    public string Impasto;
    public string Salsa;
    public string Condimento;
    // Classe che rappresenta il "prodotto" finale
}

class PizzaBuilder {
    private Pizza _pizza = new Pizza();
    public PizzaBuilder ImpostaImpasto(string tipoImpasto) {
        _pizza.Impasto = tipoImpasto;
        return this;  // permette la chiamata concatenata
    }
    public PizzaBuilder AggiungiSalsa(string tipoSalsa) {
        _pizza.Salsa = tipoSalsa;
        return this;
    }
    public PizzaBuilder AggiungiCondimento(string cond) {
        _pizza.Condimento = cond;
        return this;
    }
    public Pizza CreaPizza() {
        return _pizza;  // restituisce l'oggetto costruito
    }
}

// Utilizzo del Builder:
Pizza pizza = new PizzaBuilder()
    .ImpostaImpasto("sottile")
    .AggiungiSalsa("pomodoro")
    .AggiungiCondimento("mozzarella")
    .CreaPizza();
// pizza ora ha Impasto = "sottile", Salsa = "pomodoro", Condimento =
"mozzarella"
```

Object Pool

Il pattern **Object Pool** gestisce un insieme riutilizzabile di oggetti pronti all'uso, anziché crearne di nuovi ogni volta che servono. È utile quando gli oggetti sono costosi da creare (per esempio connessioni a database, thread, socket) e ne richiedi molti in momenti diversi: invece di istanziare e distruggere continuamente, li prendi da un **pool** (piscina) di istanze pre-costruite. In altre parole, il pool funziona da

fabbrica + cache: fornisce un oggetto libero quando richiesto e lo conserva quando non serve più, così può essere **riusato** successivamente.

Il funzionamento tipico è: il pool mantiene una collezione (lista, coda, ecc.) di oggetti disponibili. Quando il client chiede un oggetto (`Acquire`), se ce n'è uno libero il pool lo consegna immediatamente (magari dopo averlo inizializzato allo stato giusto); se non ce ne sono, ne crea uno nuovo. Quando il client ha finito di usarlo, restituisce l'oggetto al pool tramite `Release` invece di distruggerlo. In questo modo si limita il numero totale di istanze attive e si riduce il costo di creazione, migliorando le prestazioni in scenari ad alta frequenza di utilizzo di oggetti.

```
class Connection {
    public Guid Id = Guid.NewGuid();
    /* Classe risorsa costosa (es. connessione DB), identificata da un Id */
}

static class ConnectionPool {
    private static List<Connection> _available = new List<Connection>();
    public static Connection Acquire() {
        if (_available.Count > 0) {
            // Riuso un oggetto esistente dal pool
            Connection conn = _available[0];
            _available.RemoveAt(0);
            Console.WriteLine("Riuso connessione esistente " + conn.Id);
            return conn;
        } else {
            // Pool vuoto: creo una nuova connessione
            Connection conn = new Connection();
            Console.WriteLine("Creo nuova connessione " + conn.Id);
            return conn;
        }
    }
    public static void Release(Connection conn) {
        // L'oggetto viene restituito al pool per futuri utilizzi
        Console.WriteLine("Rilascio connessione " + conn.Id + " al pool");
        _available.Add(conn);
    }
}

// Utilizzo dell'Object Pool:
Connection c1 = ConnectionPool.Acquire();
Connection c2 = ConnectionPool.Acquire();
ConnectionPool.Release(c1);
Connection c3 = ConnectionPool.Acquire();  // c3 potrebbe riutilizzare c1
```

Bridge

Il pattern **Bridge** separa un'**astrazione** dalla sua **implementazione** permettendo di svilupparle in modo indipendente. Invece di avere un'unica gerarchia di classi che combina entrambe (rischiando un'esplosione di sottoclassi per ogni variante), si crea una struttura a due gerarchie: una per le

astrazioni e una per le implementazioni, collegate tra loro da un riferimento. Un esempio comune è il rapporto tra un dispositivo e il suo controller: immagina un telecomando (astrazione) che può controllare diversi dispositivi (implementazioni come TV, radio, lettore DVD). Il telecomando definisce funzioni generiche (accendi, spegni, cambia canale...) e contiene un riferimento a un'interfaccia dispositivo; i dispositivi concreti (TV, Radio, ecc.) implementano quell'interfaccia. In questo modo, aggiungere un nuovo tipo di dispositivo o una nuova variante di telecomando non richiede modifiche radicali all'altro lato.

Il Bridge quindi favorisce la **combinazione flessibile** di astrazione e implementazione: l'astrazione chiama metodi sull'interfaccia implementor, ignorando i dettagli specifici. Si può estendere la classe astratta (per esempio un telecomando avanzato) senza toccare le implementazioni dei dispositivi, e viceversa introdurre nuovi dispositivi senza cambiare il codice dei controlli. Questo riduce l'accoppiamento e evita duplicazione di codice. In sintesi, l'**astrazione** (parte alta) e l'**implementazione** (parte bassa) comunicano attraverso un ponte (il riferimento all'interfaccia), mantenendosi separate ma funzionando insieme.

```
interface IDevice {
    void Accendi();
    void Spegni();
    // Interfaccia comune per vari dispositivi (implementazione)
}

class TV : IDevice {
    public void Accendi() { Console.WriteLine("TV accesa"); }
    public void Spegni() { Console.WriteLine("TV spenta"); }
}
class Radio : IDevice {
    public void Accendi() { Console.WriteLine("Radio accesa"); }
    public void Spegni() { Console.WriteLine("Radio spenta"); }
}

class RemoteControl {
    protected IDevice device;
    public RemoteControl(IDevice dev) { this.device = dev; }
    public void AccendiDispositivo() { device.Accendi(); /* delega
all'implementazione */ }
    public void SpegniDispositivo() { device.Spegni(); }
    // Classe astratta (telecomando) che usa l'interfaccia IDevice per
operare sul dispositivo
}

// Utilizzo del Bridge:
RemoteControl telecomando = new RemoteControl(new TV());
telecomando.AccendiDispositivo();  // Output: "TV accesa"
telecomando = new RemoteControl(new Radio());
telecomando.AccendiDispositivo();  // Output: "Radio accesa"
// Il telecomando (astrazione) funziona con qualunque IDevice
(implementazione) semplicemente cambiando riferimento
```

Composite

Il pattern **Composite** permette di comporre oggetti in strutture ad **albero** per rappresentare gerarchie parte-tutto, così da trattare oggetti singoli e composti nello stesso modo. L'idea è avere un'interfaccia comune (`Component`) per sia gli oggetti **foglia** (elementi indivisibili) sia i **composti** (che contengono altri elementi). In questo modo, il client può invocare operazioni sull'oggetto a livello alto senza preoccuparsi se è un singolo elemento o un contenitore: ad esempio, in una struttura di file e cartelle, sia il file che la cartella possono avere un metodo `Stampa()`. Chiamando `Stampa()` su una cartella, questa stamperà il proprio nome e poi chiamerà `Stampa()` su ogni elemento contenuto, ricorsivamente, proprio grazie al Composite.

In un Composite, ogni elemento composito memorizza e gestisce una collezione dei propri figli (che possono essere foglie o ulteriori composti). Si possono aggiungere o rimuovere elementi tramite metodi come `Aggiungi()` e `Rimuovi()`. Quando viene eseguita un'operazione su un composito, tipicamente esso la **propaga** a tutti i figli. Questo pattern semplifica l'uso di strutture gerarchiche complesse con codice uniforme: il client interagisce con l'interfaccia comune senza distinguere esplicitamente i casi, ottenendo un sistema flessibile per rappresentare organigrammi, scene grafiche (forme dentro forme), menu con sottovoci, ecc.

```
interface IFileSystemItem {
    void Stampa(string prefisso);
    // Interfaccia comune per file (foglie) e directory (nodi composti)
}

class FileItem : IFileSystemItem {
    private string nome;
    public FileItem(string nome) { this.nome = nome; }
    public void Stampa(string prefisso) {
        Console.WriteLine(prefisso + nome);
    }
    // Foglia: stampa semplicemente il proprio nome
}

class DirectoryItem : IFileSystemItem {
    private string nome;
    private List<IFileSystemItem> figli = new List<IFileSystemItem>();
    public DirectoryItem(string nome) { this.nome = nome; }
    public void Aggiungi(IFileSystemItem item) {
        figli.Add(item);
    }
    public void Stampa(string prefisso) {
        Console.WriteLine(prefisso + nome + "/");
        foreach (var f in figli) {
            f.Stampa(prefisso + "   ");
        }
    }
    // Composito: stampa il nome e poi delega la stampa ai figli, con
indentazione
}
```

```
// Utilizzo del Composite (struttura file/cartelle):
DirectoryItem cartellaRoot = new DirectoryItem("root");
cartellaRoot.Aggiungi(new FileItem("file1.txt"));
DirectoryItem sottocartella = new DirectoryItem("docs");
sottocartella.Aggiungi(new FileItem("cv.pdf"));
sottocartella.Aggiungi(new FileItem("note.txt"));
cartellaRoot.Aggiungi(sottocartella);
cartellaRoot.Stampa("");
/* Output:
root/
   file1.txt
   docs/
      cv.pdf
      note.txt
*/
```

Flyweight

Il pattern **Flyweight** (letteralmente "peso piuma") mira a minimizzare l'uso di memoria condividendo il più possibile lo stato tra oggetti simili. Quando c'è un grandissimo numero di oggetti di piccole dimensioni che hanno molti dati in comune, Flyweight evita di avere duplicati in memoria mantenendo un unico oggetto per rappresentare elementi identici. Un esempio classico è la gestione dei caratteri di testo in un editor: invece di creare un oggetto separato per ogni lettera "A" presente nel documento (occupando molta memoria), si può riutilizzare un singolo oggetto condiviso che rappresenta la lettera "A" e referenziarlo ovunque serva quella lettera. Così gli **oggetti flyweight** possono essere usati in molti punti, ma istanziati una volta sola.

Tecnicamente, si suddivide lo stato dell'oggetto in due categorie: **stato intrinseco** e **stato estrinseco**. Lo stato intrinseco è la parte che rimane uguale per ogni oggetto condiviso (ad esempio, nel caso delle lettere, il carattere, il font, etc.) ed è memorizzato all'interno del flyweight. Lo stato estrinseco invece varia per ogni occorrenza (ad esempio la posizione della lettera nel documento, il colore se varia) e non viene memorizzato nel flyweight, ma passato dall'esterno quando serve. Si utilizza una **Flyweight Factory** per gestire gli oggetti condivisi: quando il client richiede un oggetto, la factory controlla se ne esiste già uno identico; se sì lo restituisce, altrimenti ne crea uno nuovo, assicurando che ci sia una singola istanza per ogni configurazione di stato intrinseco.

```
class Letter {
    public char Symbol;
    public Letter(char symbol) { this.Symbol = symbol; }
    // Classe leggera che rappresenta una lettera; potrebbe contenere anche
font, dimensione, ecc.
}

class LetterFactory {
    private static Dictionary<char, Letter> _cache = new Dictionary<char,
Letter>();
    public static Letter GetLetter(char symbol) {
        if (!_cache.ContainsKey(symbol)) {
            _cache[symbol] = new Letter(symbol);
```

```
            Console.WriteLine("Creo oggetto Letter per '" + symbol + "'");
        } else {
            Console.WriteLine("Riutilizzo oggetto Letter per '" + symbol +
"'");
        }
        return _cache[symbol];
    }
}

// Utilizzo del Flyweight:
Letter a1 = LetterFactory.GetLetter('a');
Letter a2 = LetterFactory.GetLetter('a');
Letter b1 = LetterFactory.GetLetter('b');
// a1 e a2 sono lo stesso oggetto condiviso per 'a'
Console.WriteLine(Object.ReferenceEquals(a1, a2));  // Output: True
```

Proxy

Il pattern **Proxy** fornisce un sostituto o rappresentante (**proxy**) di un altro oggetto per controllarne l'accesso. Invece di interagire direttamente con un oggetto "costoso" o che ha accesso ristretto, il client parla con il proxy, che implementa la stessa interfaccia del soggetto reale e si occupa di inoltrare le richieste ad esso aggiungendo eventualmente logica aggiuntiva. Questo permette, ad esempio, di implementare il **caricamento pigro** (lazy loading) di risorse pesanti: il proxy crea/carica l'oggetto reale solo quando è strettamente necessario. Oppure si può usare un proxy di protezione per verificare i permessi di accesso prima di delegare a un oggetto sensibile.

Si possono avere diversi tipi di Proxy a seconda dello scopo: **virtual proxy** per ritardare l'istanza reale finché non serve (ad esempio caricando un'immagine dal disco solo al primo `Display()`), **remote proxy** per rappresentare oggetti che risiedono su un altro spazio di indirizzamento o server, **protection proxy** per aggiungere controlli di sicurezza, ecc. Il vantaggio è che il client resta ignaro se sta parlando col proxy o col vero oggetto — dal suo punto di vista l'interfaccia è la stessa — mentre dietro le quinte il proxy decide se creare l'oggetto, memorizzarlo in cache, controllare accessi o altre logiche di contorno.

```
interface IImage {
    void Display();
    // Interfaccia comune sia per l'immagine reale che per il proxy
}

class RealImage : IImage {
    private string fileName;
    public RealImage(string fileName) {
        this.fileName = fileName;
        LoadFromDisk();
    }
    private void LoadFromDisk() {
        Console.WriteLine("Caricamento dell'immagine da disco: " + fileName);
    }
    public void Display() {
        Console.WriteLine("Visualizzazione immagine: " + fileName);
```

```
    }
    // Classe dell'oggetto reale, con caricamento costoso nel costruttore
}

class ImageProxy : IImage {
    private string fileName;
    private RealImage realImage;
    public ImageProxy(string fileName) {
        this.fileName = fileName;
    }
    public void Display() {
        if (realImage == null) {
            // Istanziazione ritardata: creo RealImage solo al primo utilizzo
            realImage = new RealImage(fileName);
        }
        realImage.Display();
    }
    // Proxy che mantiene un riferimento pigro all'immagine reale
}

// Utilizzo del Proxy:
IImage img = new ImageProxy("foto.jpg");
img.Display();  // La prima chiamata carica l'immagine da disco e poi la
mostra
img.Display();  // Le chiamate successive mostrano l'immagine senza
ricaricarla
```

Chain of Responsibility

Il pattern **Chain of Responsibility** (Catena di responsabilità) organizza una serie di **gestori** in cascata, dove ciascun handler ha la possibilità di trattare una certa **richiesta** oppure passarla al successivo nella catena. Invece di vincolare una richiesta a un singolo oggetto, si crea una catena di più oggetti che possono collaborare per gestirla. Il vantaggio è che il mittente della richiesta non conosce quale (o quali) dei possibili handler la gestiranno effettivamente: potenzialmente tutti hanno la chance di farlo finché uno "si assume la responsabilità". Un esempio comune è un sistema di assistenza: la richiesta di supporto passa dal livello 1 (es. help desk) al livello 2 e così via finché qualcuno risolve il problema. Allo stesso modo, in un motore di gioco un evento potrebbe essere offerto in sequenza a vari moduli finché uno lo consuma.

Implementare la catena significa tipicamente avere una classe base `Handler` con un riferimento al **prossimo** handler nella catena. Ogni handler concreto overridea un metodo (ad esempio `Gestisci(richiesta)`) controllando se è in grado di gestire quel tipo di richiesta. Se sì, la elabora e magari la catena termina lì; se no, passa la palla chiamando il metodo del handler successivo. Il client configura il collegamento tra i vari handler (decidendo l'ordine della catena) e poi invia la richiesta al primo della lista. Questo pattern facilita l'aggiunta o modifica di handler senza cambiare il codice del mittente; inoltre favorisce un design flessibile dove le responsabilità possono essere suddivise in più classi invece che concentrate in una sola con tanti `if/else`.

```
abstract class Handler {
    protected Handler next;
    public void ImpostaProssimo(Handler prossimo) {
        this.next = prossimo;
    }
    public virtual void Gestisci(int richiesta) {
        if (next != null) {
            next.Gestisci(richiesta);
        }
    }
    // Classe base Handler con riferimento al successivo e comportamento di
default (passa oltre)
}

class NegativeHandler : Handler {
    public override void Gestisci(int richiesta) {
        if (richiesta < 0) {
            Console.WriteLine("NegativeHandler gestisce il numero negativo "
+ richiesta);
        } else {
            base.Gestisci(richiesta);
        }
    }
}
class ZeroHandler : Handler {
    public override void Gestisci(int richiesta) {
        if (richiesta == 0) {
            Console.WriteLine("ZeroHandler gestisce lo zero");
        } else {
            base.Gestisci(richiesta);
        }
    }
}
class PositiveHandler : Handler {
    public override void Gestisci(int richiesta) {
        if (richiesta > 0) {
            Console.WriteLine("PositiveHandler gestisce il numero positivo "
+ richiesta);
        } else {
            base.Gestisci(richiesta);
        }
    }
}
// Tre handler concreti che gestiscono numeri negativi, zero e positivi
rispettivamente

// Impostazione della catena: Negative -> Zero -> Positive
Handler neg = new NegativeHandler();
Handler zero = new ZeroHandler();
Handler pos = new PositiveHandler();
```

```
neg.ImpostaProssimo(zero);
zero.ImpostaProssimo(pos);

// Invio di varie richieste alla catena:
neg.Gestisci(42);   // gestito da PositiveHandler
neg.Gestisci(0);    // gestito da ZeroHandler
neg.Gestisci(-7);   // gestito da NegativeHandler
```

Iterator

Il pattern **Iterator** fornisce un modo per **iterare** sugli elementi di una collezione in modo sequenziale senza esporre la rappresentazione interna della collezione stessa. In altre parole, l'Iterator astrae il meccanismo di attraversamento, permettendo al client di scorrere una struttura dati complessa (che potrebbe essere un array, una lista collegata, un albero, ecc.) usando sempre la stessa interfaccia semplice (`Next()`, `HasNext()`), senza bisogno di conoscere come gli elementi sono memorizzati. In C# questo concetto è integrato tramite `IEnumerable`/`IEnumerator` e il costrutto `foreach` (che dietro le quinte usa un iteratore), ma qui lo illustreremo a livello concettuale.

Con Iterator si definisce di solito una classe interna o separata che mantiene lo stato corrente dell'iterazione (ad esempio un indice per un array, un puntatore a nodo corrente per una lista, ecc.). La collezione (detta **aggregato**) espone un metodo tipo `CreaIterator()` che istanzia e restituisce un nuovo oggetto iteratore collegato ad essa. Il client userà l'iteratore per scorrere gli elementi chiamando ad esempio `HasNext()` per vedere se ci sono altri elementi e `Next()` per passare al prossimo elemento. In questo modo la logica di iterazione è separata dalla struttura dati: puoi avere collezioni diverse con iteratori diversi ma intercambiabili nel modo d'uso. Questo favorisce l'**incapsulamento** e la single responsibility: la collezione si occupa dei dati, l'iteratore di come attraversarli.

```
class MyCollection {
    private int[] items;
    public MyCollection(int[] items) {
        this.items = items;
    }
    public MyIterator CreaIterator() {
        return new MyIterator(this);
    }
    // Metodo per ottenere un iteratore collegato a questa collezione

    // Metodi di accesso usati dall'iteratore (potrebbero essere non pubblici
in realt\u00e0):
    public int GetElement(int index) { return items[index]; }
    public int Count { get { return items.Length; } }
}

class MyIterator {
    private MyCollection collection;
    private int index = 0;
    public MyIterator(MyCollection collection) {
        this.collection = collection;
    }
```

```
    public bool HasNext() {
        return index < collection.Count;
    }
    public int Next() {
        return collection.GetElement(index++);
    }
    // Questa classe incapsula la logica di scorrimento della collezione
}

// Utilizzo dell'Iterator:
MyCollection numeri = new MyCollection(new int[] { 10, 20, 30 });
MyIterator iter = numeri.CreaIterator();
while (iter.HasNext()) {
    Console.WriteLine(iter.Next());
}
/* Output:
10
20
30
*/
```

Mediator

Il pattern **Mediator** introduce un oggetto intermediario (il mediatore) per gestire in maniera centralizzata le interazioni tra un insieme di oggetti. Invece che comunicare direttamente tra loro (creando un groviglio di dipendenze incrociate), gli oggetti partecipanti - chiamati **colleghi** - si registrano presso un mediatore e inviano i messaggi a lui; sarà poi il mediatore a inoltrare tali messaggi agli altri colleghi interessati. In questo modo, la logica di coordinamento è concentrata in un solo posto, facilitando la manutenzione e l'evoluzione delle interazioni complesse. Un classico esempio è una chat room: gli utenti non conoscono direttamente gli altri utenti in linea, ma comunicano inviando messaggi alla chat room, che pensa a distribuirli.

Il mediatore quindi **riduce l'accoppiamento**: ogni collega parla solo con il mediatore, ignorando i dettagli degli altri. Per implementarlo, di solito si crea una classe Mediator con metodi tipo `Notify` o `Send`, e ciascun collega quando deve inviare qualcosa chiama quei metodi invece di chiamare direttamente un altro collega. Il Mediator contiene la logica per decidere chi deve ricevere il messaggio o che azione intraprendere. Tornando all'esempio della chat: la chat room tiene una lista di utenti registrati; quando un utente manda un messaggio, la chat lo inoltra a tutti gli altri. In un programma GUI, un Mediator potrebbe essere una finestra di dialogo che gestisce i vari controlli (quando clicchi un pulsante, la finestra mediatrice decide di abilitare/disabilitare altri controlli, ecc.). Questo approccio semplifica la comunicazione molti-a-molti orchestrandola attraverso un singolo oggetto.

```
class ChatRoom {
    private List<User> _users = new List<User>();
    public void Register(User user) {
        _users.Add(user);
        user.SetChatRoom(this);
    }
    public void Send(string from, string message) {
```

```
        foreach (var u in _users) {
            if (u.Name != from) {
                u.Receive(from, message);
            }
        }
    }
    // Mediator: tiene traccia degli utenti e inoltra i messaggi
}

class User {
    public string Name { get; }
    private ChatRoom _chat;
    public User(string name) { Name = name; }
    public void SetChatRoom(ChatRoom chat) {
        _chat = chat;
    }
    public void Send(string message) {
        _chat.Send(Name, message);
    }
    public void Receive(string from, string message) {
        Console.WriteLine(from + " a " + Name + ": " + message);
    }
    // Collega: invia messaggi tramite il mediatore e riceve messaggi dallo
stesso
}

// Utilizzo del Mediator (chat room):
ChatRoom chat = new ChatRoom();
User alice = new User("Alice");
User bob = new User("Bob");
chat.Register(alice);
chat.Register(bob);
alice.Send("Ciao Bob!");  // Solo Bob riceve il messaggio tramite il
mediatore
```

Memento

Il pattern **Memento** consente di catturare e salvare lo stato interno di un oggetto, per poi poterlo **ripristinare** in un momento successivo, il tutto senza violare l'incapsulamento (cioè senza esporre direttamente i dettagli dello stato). In pratica si tratta di creare un oggetto dedicato (chiamato Memento) che funge da **snapshot** dello stato dell'Originator (l'oggetto di cui vogliamo salvare lo stato). Un esempio tipico è un editor di testo con funzionalità di *undo*: quando salviamo la versione corrente del documento, stiamo creando un Memento contenente il testo attuale; se poi l'utente annulla l'ultima modifica, il programma ripristina lo stato precedente del documento estraendo i dati dal Memento memorizzato.

Nel Memento pattern ci sono tre ruoli principali. L'**Originator** è l'oggetto che ha uno stato interno da salvare (es. l'editor di testo). Esso sa come creare un Memento del proprio stato e come ripristinare lo stato da un Memento. Il **Memento** è l'oggetto semplice e immutabile che contiene lo stato salvato; spesso espone poco o nulla all'esterno, magari solo a chi l'ha creato. Il **Caretaker** (custode) è l'entità che

chiede all'Originator di salvare lo stato e conserva i Memento (ad esempio una pila di stati per l'undo). Quando serve un ripristino, il Caretaker passa all'Originator il Memento appropriato. Un aspetto importante è che il Caretaker non deve conoscere i dettagli dello stato nel Memento (principio di incapsulamento): tratta il Memento come un oggetto opaco. In code design, questo si traduce spesso nel rendere i dati interni del Memento accessibili solo all'Originator, magari facendo del Memento una classe innestata o usando accorgimenti di visibilità.

```
class Editor {
    private string _text;
    public Editor(string text) { _text = text; }
    public void SetText(string text) { _text = text; }
    public string GetText() { return _text; }
    public EditorMemento Save() {
        return new EditorMemento(_text);
    }
    public void Restore(EditorMemento memento) {
        _text = memento.GetState();
    }
    // Originator: sa salvare e ripristinare il proprio stato
}

class EditorMemento {
    private readonly string _state;
    public EditorMemento(string state) { _state = state; }
    public string GetState() { return _state; }
    // Memento: incapsula lo stato (qui il testo) in sola lettura
}

// Utilizzo del Memento (undo semplice):
Editor editor = new Editor("Testo iniziale");
EditorMemento salvataggio = editor.Save();    // salva lo stato corrente
editor.SetText("Testo modificato");
Console.WriteLine(editor.GetText());          // Output: "Testo modificato"
editor.Restore(salvataggio);                  // ripristina lo stato salvato
Console.WriteLine(editor.GetText());          // Output: "Testo iniziale"
```

State

Il pattern **State** permette a un oggetto di modificare il proprio comportamento al cambiare del suo **stato interno**, come se cambiasse “classe” a runtime. Invece di inserire nella classe tante variabili di stato e lunghi switch/if per ogni operazione dipendente dallo stato, si delegano tali comportamenti a oggetti separati che rappresentano gli **stati** possibili. Il contesto (l'oggetto principale) contiene un riferimento a un oggetto di stato corrente e quando viene richiesta un'operazione, la gira all'oggetto stato, il quale la esegue in modo appropriato e può decidere di cambiare lo stato nel contesto. Questo porta a un design più pulito dove ogni classe stato gestisce solo le transizioni e azioni per quello specifico stato, senza if/else sparsi.

Ad esempio, pensa a una lampadina che può essere in due stati: **Spenta** o **Accesa**. Senza pattern State, la classe Lampadina avrebbe un booleano tipo `isOn` e ogni metodo controllerebbe questo flag per

decidere cosa fare. Con il pattern State, invece, avremo due classi separate (`StatoSpento` e `StatoAcceso`) che implementano un'interfaccia comune (ad esempio `IState` con un metodo `PremiInterruttore`). La lampadina (il contesto) all'inizio contiene un riferimento a un oggetto `StatoSpento`. Quando qualcuno preme l'interruttore, la lampadina delega all'oggetto stato corrente l'operazione: l'oggetto `StatoSpento` gestirà l'evento dicendo "accendi la luce" e cambierà lo stato della lampadina a `StatoAcceso`. La prossima pressione verrà gestita dall'oggetto `StatoAcceso`, che spegnerà la luce e riporterà lo stato a Spento. Questo pattern facilita l'aggiunta di nuovi stati o cambi di logica di transizione senza toccare l'intera classe principale, rispettando il principio aperto/chiuso.

```
interface IState {
    void PremiInterruttore(Lampadina lamp);
    // Interfaccia comune per tutti gli stati
}

class Lampadina {
    public IState Stato { get; set; }
    public Lampadina() {
        Stato = new SpentaState();  // stato iniziale
    }
    public void PremiInterruttore() {
        // Delego il comportamento allo stato attuale
        Stato.PremiInterruttore(this);
    }
    // Contesto: mantiene un riferimento allo stato corrente e lo delega
}

class SpentaState : IState {
    public void PremiInterruttore(Lampadina lamp) {
        Console.WriteLine("La lampadina si ACCENDE.");
        lamp.Stato = new AccesaState();
    }
}
class AccesaState : IState {
    public void PremiInterruttore(Lampadina lamp) {
        Console.WriteLine("La lampadina si SPEGNE.");
        lamp.Stato = new SpentaState();
    }
}
// Stati concreti: implementano il comportamento per lo specifico stato e
cambiano lo stato del contesto quando serve

// Utilizzo del State:
Lampadina lamp = new Lampadina();
lamp.PremiInterruttore();  // Output: "La lampadina si ACCENDE." (stato
cambia a Accesa)
lamp.PremiInterruttore();  // Output: "La lampadina si SPEGNE." (stato torna
a Spenta)
```

Template Method

Il pattern **Template Method** definisce la **struttura di un algoritmo** nell'ambito di un metodo, ma delega alcuni passi dettagliati alle sottoclassi. In altre parole, una classe astratta implementa un metodo template che rappresenta lo scheletro dell'operazione complessa, chiamando in sequenza vari metodi (alcuni con un'implementazione di default, altri astratti da ridefinire). Le sottoclassi estendono questa classe e forniscono le implementazioni specifiche per i passi variabili, senza alterare l'ordine generale delle cose. Questo è utile quando si hanno più varianti di un processo con una logica comune: il Template Method consente di mettere la logica condivisa in alto e lasciare che le differenze vengano gestite in basso, rispettando l'idea di codice riutilizzabile e aperto all'estensione.

Un esempio concreto: supponiamo di dover definire un processo di elaborazione dati che prevede sempre tre passi – leggere i dati, processarli, salvare il risultato – ma la parte di "processamento" cambia a seconda del tipo di dati (numeri, testo, immagini, ecc.). Possiamo creare una classe astratta `ElaboratoreDati` con un metodo `Elabora()` che chiama nell'ordine `LeggiDati()`, `ProcessaDati()` e `SalvaDati()`. `LeggiDati()` e `SalvaDati()` possono avere una implementazione standard (es. leggere da una fonte comune e scrivere su output comune), mentre `ProcessaDati()` è astratto e verrà definito dalle sottoclassi come `ElaboratoreNumeri`, `ElaboratoreTesto`, ecc. In questo modo ogni sottoclasse si concentra solo sul passo specifico, ma beneficia dell'implementazione generale del template method che garantisce la sequenza corretta delle operazioni.

```
abstract class ElaboratoreDati {
    public void Elabora() {
        LeggiDati();
        ProcessaDati();
        SalvaDati();
    }
    protected void LeggiDati() {
        Console.WriteLine("Leggo i dati dall'origine");
    }
    protected abstract void ProcessaDati();
    protected void SalvaDati() {
        Console.WriteLine("Salvo i dati nell'output");
    }
    // Classe base astratta con Template Method (Elabora) e passi fissi/
variabili
}

class ElaboratoreNumeri : ElaboratoreDati {
    protected override void ProcessaDati() {
        Console.WriteLine("Processo dati numerici (es. calcolo media)");
    }
}
class ElaboratoreTesto : ElaboratoreDati {
    protected override void ProcessaDati() {
        Console.WriteLine("Processo dati testuali (es. rimuovo spazi
extra)");
    }
}
```

```
// Sottoclassi che implementano il passo variabile specifico (ProcessaDati)

// Utilizzo del Template Method:
ElaboratoreDati proc = new ElaboratoreNumeri();
proc.Elabora();
/* Output:
Leggo i dati dall'origine
Processo dati numerici (es. calcolo media)
Salvo i dati nell'output
*/
```

Visitor

Il pattern **Visitor** permette di definire nuove operazioni su una struttura di oggetti senza modificarne le classi. Si tratta di creare un **oggetto esterno** (il visitatore) che contiene la logica di un'operazione, e di far "accettare" questo visitatore agli elementi della struttura in modo che la giusta funzione di visita venga chiamata. In termini semplici, invece di avere metodi come `CalcolaCosto()`, `StampaDettagli()` dentro di ogni classe di oggetto su cui vuoi operare, crei delle classi Visitor separate (es. `CalcolaCostoVisitor`, `StampaDettagliVisitor`) che implementano tali operazioni per ogni tipo di oggetto possibile. Gli oggetti della struttura (chiamati **elementi**) allora hanno un metodo `Accept(IVisitor)` che chiama `visitor.Visit(this)` passando se stessi. In questo modo puoi aggiungere facilmente una nuova operazione definendo un nuovo Visitor, senza toccare le classi esistenti degli elementi (che restano aperte all'estensione ma chiuse alla modifica, secondo il principio OCP).

Il Visitor è utile quando hai una struttura dati complessa (es. un albero di elementi di diverso tipo, come nodi di un AST, oggetti grafici in una scena, elementi di fatturazione, ecc.) e vuoi eseguire operazioni differenti su quegli elementi senza intasarne le classi con ogni possibile metodo. La doppia chiamata `Accept/Visit` garantisce che venga invocato il metodo appropriato del Visitor in base al tipo concreto dell'elemento (meccanismo di **double dispatch**). Ad esempio, se abbiamo classi `Cerchio` e `Rettangolo` e vogliamo aggiungere un'operazione di calcolo dell'area, creiamo un `AreaVisitor` con metodi `Visit(Cerchio)` e `Visit(Rettangolo)` che effettuano i calcoli. I nostri elementi implementano `Accept(visitor)` chiamando il metodo specifico. Così il codice che calcola l'area totale di una lista di forme può essere scritto senza *if* o *instanceof*, semplicemente iterando sulle forme e chiamando `forma.Accept(areaVisitor)`. Domani, se volessimo aggiungere un'operazione di disegno o di esportazione, creeremmo un nuovo Visitor senza dover modificare le classi `Cerchio` e `Rettangolo`.

```
interface IShape {
    void Accept(IShapeVisitor visitor);
    // Interfaccia per gli elementi: definisce il metodo Accept
}

class Cerchio : IShape {
    public double Raggio;
    public Cerchio(double r) { Raggio = r; }
    public void Accept(IShapeVisitor visitor) {
        visitor.VisitCerchio(this);
    }
```

```
}
class Rettangolo : IShape {
    public double Base, Altezza;
    public Rettangolo(double b, double h) { Base = b; Altezza = h; }
    public void Accept(IShapeVisitor visitor) {
        visitor.VisitRettangolo(this);
    }
}
// Elementi concreti (Cerchio, Rettangolo) che implementano Accept chiamando
il Visitor appropriato

interface IShapeVisitor {
    void VisitCerchio(Cerchio c);
    void VisitRettangolo(Rettangolo r);
    // Interfaccia Visitor: un metodo per ogni tipo di elemento da visitare
}

class CalcolaAreaVisitor : IShapeVisitor {
    public double AreaTotale = 0;
    public void VisitCerchio(Cerchio c) {
        AreaTotale += Math.PI * c.Raggio * c.Raggio;
    }
    public void VisitRettangolo(Rettangolo r) {
        AreaTotale += r.Base * r.Altezza;
    }
}
// Visitor concreto: implementa la logica (qui somma le aree) per ogni tipo
di elemento

// Utilizzo del Visitor:
List<IShape> forme = new List<IShape> {
    new Cerchio(3),
    new Rettangolo(4, 5)
};
CalcolaAreaVisitor visitor = new CalcolaAreaVisitor();
foreach (IShape forma in forme) {
    forma.Accept(visitor);
}
Console.WriteLine("Area totale: " + visitor.AreaTotale);
// Output: somma delle aree di tutte le forme (Area totale: 37.2743...)
```
