**INTRO**

**DESIGN PATTERN**: Coppia problema/soluzione che costituisce unità di riuso (soluzione elegante (con un nome) di uno specifico problema di design/programmazione OO che costituisce un'unità di riuso), non basata su un particolarer linguaggio OO.

Servono a: applicare principi di buona proj (high coesion, low coupling, …), a scrivere buoni design e a facilitare la comunicazone tra sviluppatori.

NB: esistono altre unità di riuso (librerie, COTS, frameworks, product lines, modelli)

**FRAMEWORK:** insieme di classi e interfacce cooperanti che realizzano un design riusabile e customizzabile per uno specifico dominio applicativo o tipologia di app. Dettano l’architettura e dunque si ha l’opposto delle librerie: con esse chiamavo le funzioni dal main, qui ho il main e devo creare le sottoclassi specifiche (“riempio buchi”)

**Design pattern vs. framework** più astratti: vanno implementati/adattati e non si possono usare “as-is”, più piccoli (come elementi architetturali) , di solito meno specializzati, non relativi a dominio applicativo specifico o tipologia di app

**DESIGN PATTERNS GRASP**: Per assegnazione di responsabilità nel sw

**Controller**: oggetto (oltre la UI) che riceve/coordina op di sys (evento input nel sistema, msg verso System nei Seq Diagr. → assegna la responsabilità (a seconda se la classe è un sistema piccolo o grande, ho il singolo use case o il sys intero)

**DESIGN PATTERN CLASSICI**: **CREAZIONALI**, STRUTTURALI, COMPORTAMENTALI

Descrizione: nome meaningful | problema (quando applicare il pattern + precond) | soluzione (spesso sketch), conseguenze .

**CREAZIONALI**: “chi crea un ogg di tipo A?” (per le creazioni, quando però c’è abbastanza complessità) ci si affida al Creator (se B aggrega/utilizza A o possiede dati di A)

STRUTTURALI: strutture tipiche (composizioni) / COMPORT.: interazione e respons. distrib.

**FACTORY**: nascondere la logica perchè potenzialmente complessa (x separare logica di creazione da applicativa, x avere gestioni della memoria x prestazioni): obj factory che crea

**ABSTRACT FACTORY**: soluzione x creare famiglie di prodotti affinché il client non debba specificare le classi concrete dei prodotti nel proprio codice. Si usa per rendere il sys indipendente da come i suoi prodotti sono creati, composti e rappresentati (sys config per diverse famiglie di prodotti disponibili (es: Window1, Scrollbar1 e Window2, Scrollbar2 MA non WIndow1, Scrollbar2) → multipiattaforma ma non eccessiva dipendenza da essa (la GUI non deve sapere quali classi concrete istanzia, e NON deve sbagliare accoppiamento).

→ GUI chiede una fabbrica (concreta) della fam1, è poi lei a dare (e accoppiare) i widget.

→ isola classi concrete, ez cambio di famiglia, ez consistenza dei prodotti di una famiglia

MA difficile aggiungere nuovi prodotti (dobbiamo modificare tutta l’architettura (concrFact, ..)

ADAPTER: Converte l’interfaccia di una classe in un’altra interfaccia che il cliente si aspetta

*ES: Vogliamo aggiungere X (simile ad Y). Usiamo la classe Y? Ha un’interfaccia diversa e non possiamo modificare Y (o non abbiamo il sorgente o dobbiamo usarla così com’è (es. perchè usata da altre classi)) → Soluzione: creare un adattatore e usare Y!*

Noi vediamo l’oggetto adattatore (basato su delega/composizione)

FACADE: Rendere ez l'accesso a sottosistemi che espongono interfacce complesse / fornire un’unica interfaccia per un insieme di funzionalità sparse → PRO: accoppiamento debole, nasconde componenti al cliente (se vuole il cliente può comunque accedere alle varie classi)  
TEMPLATE: Definisce lo scheletro di un algoritmo posponendo la definizione di alcuni passi a delle sottoclassi (si implementa l’invariante una sola volta) → PRO: riuso, inversione flusso di controllo (di solito sono le sottoclassi a chiamare le super, qui è il templateMethod a chiamare le sottoclassi), permette più sottoclassi concrete / CONTRO: importante chiarire le operazioni che vanno ridefinite.

OBSERVER: Gli osservatori (magari varie viste sullo stesso obj) si registrano presso l’oggetto osservato (subject) e si possono aggiungere e togliere dinamicamente (chiamando i metodi del subject “attach: add a lista in subject” e “detatch”). Quando l’oggetto osservato cambia stato, notifica tutti gli osservatori (notify: ciclo sulla lista degli osservatori, chiamando le loro update). Quando notificato, ogni osservatore decide cosa fare (update specifica).

→ dipendenza lasca 1-N / broadcast / accoppiamento debole (notifica obj senza assunzione a priori su che obj siano) MA modifica nell’osservato può scatenarla negli osservatori.

**MVC** (pattern/stile arch.): 1 modello (dati e funz. base → elab. dati), viste (mostrare info agli utenti → gestione OUT), controller(s) (gestione richieste utente → gestione IN). Un’interfaccia utente è formata da una vista e un controller. Segue possibile scenario di propagazione dell’input (contestuale a un meccanismo “observer” di propagazione):  
controller accetta evento, chiede al modello l’exec di un servizio, modello esegue (magari cambiando stato), modello notifica viste e relativi controller dei cambiamenti di stato (es. tramite Observer) → {ogni vista chiede al modello i dati cambiati e aggiorna la visualizzazione / anche i controller interrogano il modello per capire se devono abilitare o disabilitare certe funzionalità} → il controllo torna al controller originale

STATE: Permettere a un oggetto di cambiare il suo comportamento al variare del suo stato interno → implementazione State machine (estraendo la rappresentazione dello stato in classi esterne con una gerarchia + polimorfismo per variare il comportamento) →

PRO: incapsula stato in oggetto e transizioni in un altro (NO switch), comportamento x dipende solo da classe x (ed è facile cambiarlo e aggiungerne nuovi), CONTRO: + classi