	FACTORY METHOD	PROTOTYPE	ABSTRACT FACTORY	SINGLETONE	ADAPTER	FACADE
INTENTO	*Interfaccia* per un oggetto, le sottoclassi decidono quale classe istanziare.	Tipo di oggetti specificato con istanze prototipali (nuovi cloni di altre per mano del client)	Fornire interfaccia per famiglie di oggetti correlate senza specificare le classi concrete.	Un'istanza per classe accessibile globalmente.	Converte l'interfaccia in quella attesa dal Client (*compatibilità*).	Fornire un'interfaccia unificata per rendere il sistema facile da usare.
PROBLEMA	Il metodo Factory() incapsula la conoscenza su quale classe creare (di base un framework conosce la classe astratta non istanziabile per creare oggetti)	Sistema indipendente dal prodotto che usa (classe concreta nota a run-time)	Per sistemi adattabili a più contesti. Indipendenza dal tipo di prodotti; sistema configurabile con una delle famiglie.	Alcune classi devono avere una sola istanza.	Capita che una classe sia inutilizzabile per incompatibilità con l'interfaccia. Si potrebbe voler decidere che metodo chiamare senza dirlo al chiamante.	Spesso si hanno tante classi con funzioni correlate e l'insieme delle interfacce può essere complesso, anche per capire quali sono essenziali alla comunicazione con i client.
SOLUZIONE	*Product*->interfaccia; *ConcreteProduct*-> implementazione; *Creator*-> ha il factory() che torna un Product; *ConcreteCreator*-> ha factory(), sceglie che Concrete Product istanziare e lo torna	Interfaccia *Cloneable* (esiste). Due tipi di copie: 1-*Shallow*: riferimento agli stessi attributi dell'oggetto; 2-*Deep*: nuovi attributi per clone.	Creo un'interfaccia/ classe astratta per componente del sistema e una classe concreta per contesto. L'app può rivolgersi all'interfaccia.	Classe Singleton implementa *getInstance()* [static] che torna l'unica istanza creata. La classe è responsabile della creazione e il costruttore è *privato* (no creazione con *new*)	**Object Adapter** con: 1-*Target*: interfaccia attesa; 2-*Client*: usa oggetti di Target; 3-*Adaptee*: oggetto di libreria da adattare; 4-*Adapter*: converte la chiamata del client all'interfaccia della classe di libreria, implementa Target e tiene riferimento a Adaptee.	**Facade** fornisce tale interfaccia ai client e nasconde gli oggetti del sottosistema, invocandone i metodi. Tutto *private/protected*, Facade *public*. Le classi possono essere annidate nella classe Facade.
CONSEGUENZE	Le classi conoscono e lavorano con *Product/ ConcreteProduct*	Prototipi sostituibili a run-time con struttura interna nascosta. A volte è utile un **Prototype Manager**	Rimanda la creazione dei prodotti ad un'istanza della sottoclasse; usa un oggetto in cui incapsula più prodotti; famiglia variabile ma aggiunta di nuovi prodotti difficile.	Singleton controlla gli accessi all'istanza e il numero di istanze create, meglio di usare static per operazioni e variabili. Modificare Singleton per cambiare il num di istanze create.	Client e Adaptee indipendenti, ma Adapter può cambiare il comportamento di Adaptee; si possono aggiungere test; si può implementare la tecnica **Lazy Initialization**; ogni invocazione del Client ne crea un''altra di Adapter (rallentamento e complessità).	Promuove l'accoppiamento debole tra sottosistema e client; riduce dipendenze di compilazione.

Prototype Manager: registro di prototipi clonabili, torna il riferimento ad un prototipo associato ad una chiave.

Class/Object Adapter: è poco pratico adattare le interfacce sottoclassandole tutte, ma si adeguano a quella del client. La Class Adapter usa l'ereditarietà, adatta le classi ai suoi ascendenti, permette l'override dei metodi della classe adattata. Un Object Adapter usa la composizione e consente l'uso di un solo Adapter per più adattati, rende complicato la ridefinizione dei comportamenti dato che non può sapere con quale classe specifica si stia lavorando.

Lazy Initialization: si attende l'invocazione di un metodo di Adaptee prima di istanziarlo.

Facade: gruppi di classi collaboratrici per realizzare set di compiti specifici. Un sottosistema deve essere una buona astrazione per una parte precisa del sistema.

	TEMPLATE METHOD	STRATEGY	STATE	OBSERVER	MVC	MEDIATOR
INTENTO	Codificare l'algoritmo di un'operazione delegando la implementazione, ridefinibile senza alterazioni, di alcuni passi alle sottoclassi	Definire una famiglia di algoritmi correlati, incapsularli singolarmente (intercambiabili)	Permette ad un oggetto di alterare il comportamento al cambiamento di stato.	Definire una dipendenza tra oggetti in modo che al cambio di stato di uno, i dipendenti cambiano anche.	Pattern per applicazioni interattive con: **Model:** funz. principali e dati; **View:** mostra i dati; **Controller:** comprende input.	Definire interfaccia per la comunicazione di un gruppo di oggetti che interagisce, evita un'interazione diretta tra oggetti, modificando le interazioni indipendentemente.
PROBLEMA	Application Framework con classi astratte con logica applicativa che opera su prodotti diversi.	Voglio avere versioni diverse dell'algoritmo, manipolare i dati senza l'accesso del client, ho classi correlate diverse per i modi in cui attivano lo scopo.	Stati senza variabili condivisibili in più contesti (si possono creare con Singleton per istanziarli una sola volta).	Sistema partizionato in classi che collaborano, si mantiene la consistenza di oggetti con relazioni.	Le interfacce possono cambiare, le stesse info sono rappresentate in finestre diverse, i cambiamenti dell'interfaccia dovrebbero essere facili.	Gli oggetti potrebbero interagire con tutti gli altri rendendo il sistema difficile da riusare e cambiare tutto il comportamento.
SOLUZIONE	Definisco la struttura origine lasciando alcuni passi astratti. Lascio le parti variabili alle sottoclassi, elimino la duplicazione di comportamenti comuni. Operazioni primitive protected, metodi template senza override.	Definisco le interfacce di *Strategia* e *Contesto* e come interagiscono. Posso usare **inner class** (classi membri di altre).	Inserire ogni ramo condizionale in una classe separata con il polimorfismo. Context definisce l'interfaccia del client e ha un'istanza di ConcreteState, che definisce lo stato corrente. State è un'interfaccia con il comportamento associato ad uno stato di Context.	Operazioni di *update()* [Observer, ConcreteObserver] e *notify()* [Subject] per passare il dato sullo stato, cambio gestito e fatto con *getState()* e *setState()* [ConcreteSubject]. ***Java.util.Observer e Observable***	*Model:* registra View e Controller e li avvisa dei cambiamenti di dati; *View:* associata ad un Controller, lo inizializza e mostra i dati letti da Model; *Controller:* riceve gli input come eventi e li traduce in richieste che invia a Model e avvisa View.	Isolare le comunicazioni con una classe *Mediator*. *ConcreteMediator* implementa il comportamento cooperativo e coordina oggetti *Colleague,* che conoscono e comunicano con Mediator. *ConcreteColleague* scambiano richieste con Mediator.
CONSEGUENZE	Inventa le strutture di controllo (il padre esegue metodi delle sottoclassi).	Ho una famiglia di algoritmi, elimino istruzioni condizionali con *strategy.do()* . I client devono conoscere le strategie.	Il comportamento di uno stato è *localizzato* in ConcreteState e si partiziona il comportamento di stati diversi. Si possono aggiungere nuovi stati con sottoclassi; la logica di cambiamento di stato è separata.	Subject conosce solo Observer. ConcreteObserver e ConcreteSubject sono separati e facili da usare. La notify non dice agli Observ che cambia.	Lavoro suddiviso in tre componenti, più facile da comprendere e da gestire.	Oggetti facili da implementare e mantenere, i Colleague sono riusabili, il Mediator no.

Template Method -> passi: scrivo il codice in un metodo unico, lo divido con i commenti, metto ogni parte in un metodo, riscrivo il template invocando i metodi, ripeto…

Java.util.Observable: Subject con metodo notifyObservers(), ha un flag setChanged() per il cambiamento di stato nei metodi di ConcreteSubject.

Java.util.Observer: metodo update() con possibile argomento che identifica l'oggetto causa dell'aggiornamento.

	DECORATOR	COMPOSITE	BRIDGE	CHAINRESPONSIBILITY	COMMAND	COMMAND PROCESSOR
INTENTO	Aggiungere responsabilità agli oggetti con una componente, le funzionalità si possono aggiungere a catena.	Gestire strutture ad oggetti **composite**.	Dividere e variare indipendentemente astrazione e implementazione.	Dividere mandante e ricevente di una richiesta, gestita da più oggetti. Concatenazione di oggetti fino al gestore.	Incapsulare le richieste in un oggetto.	Command con l'aggiunta di un gestore degli oggetti che salva tutte le istanze, schedula l'esecuzione e può conservarli anche dopo per UNDO.
PROBLEMA	Aggiungere/togliere combinazioni di responsabilità a singoli oggetti e non a tutta la classe.	Raggruppare piccoli elementi per farne uno grande. Si usa l'interfaccia Component. Si ha una struttura ad albero.	Per un'astrazione con più implementazioni si usa ereditarietà (dipendenza)	Volendosi interfacciare con l'app, non si comunicherà subito con l'oggetto finale, si arriverà a lui.	Capita di avere problemi non definibile una volta per tutte di base, serve voler supportare applicazioni future.	
SOLUZIONE	*Component*->interfaccia con operazione che lega gli oggetti; *ConcreteComponent*-> oggetto a cui dare responsabilità; *Decorator*-> component con riferimento a un altro per inoltrare richiesta; *ConcreteDecorator*-> implementa responsabilità per il component.	*Component*-> interfaccia/classe astratta per gli elementi, ha operazioni comuni ad oggetti semplici; *Leaf*-> elemento semplice sottoclasse di Component; *Composite*-> sottoclasse di Component con elementi contenitori, ha operazioni per gestire i Leaf e quelle di Component.	*Abstraction*-> interfaccia client con riferimento a Implementor a cui inoltra richieste; *RedifinedAbstraction* ->estende interfaccia; *Implementor*-> interfaccia classi della implementazione, ha operazioni primitive; *ConcreteImplementor* -> implementa Implementor con operazioni concrete.	*Handler*-> interfaccia delle richieste e si riferisce al successore; *ConcreteHandler*-> sottoclasse di Handler, gestisce se può le richieste, altrimenti le passa al successore. Si usa se più oggetti possono gestire la stessa richiesta, ma non si sa chi a priori.	Classi: *Invoker , Command, ConcreteCommand* e *Receiver* (con cui si interfaccia il client). Utile per non definire callback staticamente; avere definizione, invocazione ed esecuzione in tempi diversi; per un log delle richieste; UNDO delle modifiche con log; per un sistema transazionale.	Si usa il Singleton per imporre un gestore unico. *CommandProcessor*-> si interfaccia con il client e passa i comandi al Command, o può annullarli.
CONSEGUENZE	Più flessibilità, aggiunta multipla di responsabilità, tante classi per un compito. Il comportamento cambia a run-time (come STATE), ma cambia effettivamente (tipo) e aggiungo funzionalità.	Composizione di elementi ricorsiva, un client può ricevere un composite, si possono aggiungere nuovi elementi. Si devono aggiungere controlli a run-time.	Implementazione modificabile a run-time, le gerarchie possono evolvere indipendentemente.	Chi richiede non conosce il ricevente e viceversa, si può cambiare la catena a run-time. Non c'è garanzia che una richiesta venga gestita.	Divisione tra oggetto che invoca e che esegue, si possono cambiare/ aggiungere nuovi comandi senza cambiare le classi esistenti. Si possono creare comandi complessi assemblandoli in un oggetto composite.	Flessibilità nell'esecuzione delle richieste, nel numero e nelle funzionalità di esse. I comandi si possono conservare per ripeterli in seguito o annullare, … Perdita d'efficienza per indirezione e aumento di classi. Complessità maggiore.

Composite: composto da altri oggetti.