**INGEGNERIA DEL SOFTWARE PT.2 TESTO PPT**

* **DIVISIONE MVC:**

Il modello MVC è un'architettura software che divide un'applicazione in tre componenti principali: il Model (modello), la View (vista) e il Controller (controllore).  
Questa separazione mira a migliorare la modularità, la manutenibilità e la testabilità del software.

**Descrizione dei Componenti:**

1. **Model (Modello)**
   * Rappresenta i dati e la logica di business.
   * È responsabile di:
     + Gestire i dati dell'applicazione (spesso interfacciandosi con il database).
     + Applicare le regole di business.
   * Comunica direttamente con il controller, ma mai con la vista.

*Esempio*: In un'applicazione e-commerce, il modello potrebbe essere responsabile di gestire i dati di un prodotto.

1. **View (Vista)**
   * Si occupa della presentazione dei dati all'utente.
   * È responsabile di:
     + Mostrare informazioni prese dal modello.
     + Ricevere input dall'utente tramite interfacce grafiche o comandi.
   * Non contiene logica di business.

*Esempio*: Un'interfaccia che mostra la lista dei prodotti.

1. **Controller (Controllore)**
   * Agisce come intermediario tra il modello e la vista.
   * È responsabile di:
     + Gestire gli input dell'utente.
     + Decidere come aggiornare il modello e quale vista mostrare.

*Esempio*: Quando un utente clicca su un prodotto, il controller aggiorna il modello per segnare l'azione e carica una nuova vista.

**Vantaggi principali della suddivisione:**

1. **Alta coesione:** ogni componente (Controller e View) ha responsabilità chiare e ben definite.

I Controller gestiscono esclusivamente la logica applicativa legata a un particolare contesto (Manager, StoreManager, ecc.). Le View si occupano solo di mostrare i dati per quello specifico contesto.

1. **Manutenzione facilitata:** modifiche a un'area del progetto non impattano altre aree. Le aggiunte o le modifiche sono localizzate a un Controller e a una View specifica.
2. **Facilità di test:** è più facile scrivere test per ciascun Controller e View separatamente. Si evitano dipendenze inutili tra componenti non correlate.
3. **Scalabilità:** se il progetto cresce, puoi aggiungere nuovi Controller e View senza modificare le strutture esistenti.

**Limitazioni:**

1. **Curva di apprendimento**: Richiede un po' di tempo per essere compreso appieno, specialmente per i principianti.
2. **Overhead iniziale**: Può sembrare complesso in applicazioni semplici.

* **PATTERN GRASP:**
* **PATTERN SOLID:**
* **PATTERN GOF:**

**CONVERT EXCEPTION:**

Il Pattern Convert Exception consiste nel tradurre un'eccezione generata da una libreria o componente esterno in un'eccezione specifica e significativa per il contesto della tua applicazione.  
Questo approccio migliora la leggibilità e la manutenibilità del codice. Il suo obiettivo è offrire una gestione delle eccezioni che sia chiara, contestualizzata e consistente, riducendo la dipendenza da eccezioni generiche.

**Come funziona?**

1. **Cattura dell'eccezione originale**:  
   Utilizza un blocco try-catch per intercettare l'eccezione generata da un componente esterno.
2. **Conversione in un'eccezione personalizzata**:  
   Crea una nuova eccezione che rappresenti meglio il contesto applicativo e includi i dettagli dell'eccezione originale.
3. **Propagazione dell'eccezione convertita**:  
   L'eccezione convertita viene rilanciata o gestita in un livello superiore.

**Vantaggi dell'utilizzo di una ConvertException:**

1. **Specificità:** Fornisce un'indicazione chiara che l'errore è legato a un problema di conversione, rendendo più facile comprendere e debuggare il codice.
2. **Gestione degli errori:** Permette di gestire gli errori di conversione separatamente dagli altri tipi di eccezioni, consentendo una gestione degli errori più precisa e appropriata.
3. **Leggibilità del codice:** Migliora la leggibilità del codice rendendo esplicito lo scopo dell'eccezione, aiutando altri sviluppatori a comprendere il codice più rapidamente.
4. **Manutenibilità:** Rende il codice più manutenibile centralizzando la gestione degli errori di conversione, che possono essere aggiornati o modificati in un unico punto se necessario.
5. **Messaggi personalizzati:** Consente di fornire messaggi di errore personalizzati e contesto aggiuntivo specifico per gli errori di conversione, utili per il logging e il debugging.

**Limitazioni:**

1. **Complessità aggiuntiva**:  
   Introduce codice aggiuntivo, soprattutto in applicazioni semplici.
2. **Rischio di nascondere informazioni**:  
   Se non si propagano correttamente i dettagli dell'eccezione originale, si rischia di perdere informazioni importanti.

**ADAPTER:**

### Il Pattern Adapter è un design pattern strutturale che consente a due interfacce incompatibili di lavorare insieme, fungendo da "traduttore" tra di esse. Questo pattern è molto utile quando hai componenti esistenti che non possono essere modificati, ma che devono funzionare con nuovi sistemi. Il suo obiettivo è di integrare classi o interfacce incompatibili, permettere il riutilizzo di codice esistente senza modificarlo, ridurre la duplicazione di codice e migliorare la manutenibilità.

### **Tipi di Adapter:**

1. **Object Adapter**: Utilizza la **composizione** per adattare l'interfaccia di una classe a quella desiderata. Più flessibile perché lavora con oggetti, non con classi.
2. **Class Adapter**: Utilizza l'**ereditarietà** per adattare l'interfaccia. Può essere limitato dai vincoli dei linguaggi che non supportano l'ereditarietà multipla.

### **Come funziona?**

#### Struttura del Pattern Adapter:

1. **Target (Obiettivo)**: L'interfaccia richiesta dal sistema.
2. **Adaptee (Adattato)**: La classe esistente che deve essere adattata.
3. **Adapter**: La classe che implementa l'interfaccia Target e converte le richieste per l'Adaptee.

**Vantaggi pattern Adapter:**

1. **Disaccoppiamento:** Disaccoppia la classe *ManagerView* dall'implementazione specifica della classe *InputDati*, rendendo il codice più modulare e facile da mantenere.
2. **Flessibilità:** Consente di cambiare il meccanismo di input senza modificare la classe *ManagerView*. Ad esempio, è possibile passare da *InputDati* a un'altra libreria o metodo di input semplicemente creando un nuovo adapter.
3. **Testabilità:** Migliora la testabilità permettendo di simulare l'interfaccia *InputAdapter* nei test unitari, rendendo più semplice testare la classe *ManagerView* in isolamento.
4. **Principio di Responsabilità Singola:** Rispetta il Principio di Responsabilità Singola separando le preoccupazioni tra lettura dell'input e la sua elaborazione, rendendo il codice più pulito e organizzato.
5. **Riutilizzabilità:** Promuove la riutilizzabilità consentendo l'uso dello stesso adapter in diverse parti dell'applicazione dove è richiesto un input.

In sintesi, il pattern Adapter migliora la manutenibilità, la flessibilità e la testabilità del codice.

**Limitazioni:**

1. **Complessità aggiuntiva**:  
   Aggiunge uno strato intermedio che può complicare il sistema.
2. **Possibile perdita di prestazioni**:  
   A causa della traduzione delle chiamate tra interfacce.