Fase di Progettazione:
Principi di Progettazione e qualità di un progetto

Obiettivo della fase di progettazione non è solo la pianificazione del lavoro ma anche pensare alla qualità (manutenzione e riuso).
Lo scopo è quindi produrre un sistema che realizza i requisiti funzionali e di qualità, facilmente mantenibile (semplice ed economico), e riusabile (facile usarne parti per sistemi futuri).

Unità di progettazione: componenti e moduli

Componente (elemento a “run time”): ha un’interfaccia ben definita verso gli altri componenti, la progettazione dovrebbe facilitare composizione, riuso e manutenzione dei componenti.

Modulo (elemento a “design time”): unità di incapsulamento (permettono separazione tra interfaccia e corpo) e unità di compilazione (quando possono essere compilati separatamente).

Principi Generali

Information Hiding (vs Incapsulamento)

L’incapsulamento indica la proprietà degli oggetti di mantenere al loro interno sia gli attributi che i metodi (stato e comportamento).

Si parla di **Information Hiding** quando alcuni attributi e metodi possono anche essere nascosti all’interno dell’oggetto (invisibili ad altri oggetti, dichiarati privati).

L’incapsulamento permette l’information hiding ma non lo garantisce (variabile pubblica -> incapsulata ma non nascosta).

Per l’information hiding c’è

- una separazione tra interfaccia (visibile) e implementazione (invisibile / privata / nascosta)
- componenti o moduli come scatole nere
- sono mantenuti nascosti

Information Hiding: Interfaccia e Corpo

L'interfaccia di un'unità di progettazione esprime ciò che l'unità offre o richiede (elementi visibili alle altre unità).

Il **corpo** contiene l'implementazione degli elementi dell'interfaccia e realizza la semantica dell'unità (corpo -> nascosto).

Information Hiding: Vantaggi

I vantaggi dell'information hiding sono :

- **Comprensibilità:** non serve comprendere i dettagli implementativi per usare un'unità.
- **Manutenibilità:** si può cambiare implementazione di una unità senza cambiare le altre.
- **Lavoro in team:** la separazione corpo-interfaccia facilita lo sviluppo da parte di persone che lavorano in modo indipendente
- **Sicurezza**: i dati di un'unità possono essere modificati solo da funzioni interne alla stessa e non dall'esterno.

Information Hiding: Tecniche Note

Per realizzare Information Hiding creare variabili private a cui si accede solo con getters (accessors - get()) e setters (mutators - set()).

La get() non deve modificare lo stato dell'oggetto (side effects) ma restituire un dato come valore (restituire un clone).

La set() permette una modifica controllata delle proprietà di un oggetto.

Molti editor permettono di generarli in automatico per ogni attributo ma così facendo si presentano due svantaggi: potrebbero essere metodi non necessari o si potrebbe non voler permettere un accesso.

Astrazione sul controllo: Librerie

Per Modulo si intende una prima forma di astrazione effettuata sul flusso di controllo e il concetto di modulo è identificato con il concetto di subroutine o procedura.

Una procedura può effettivamente nascondere una scelta di progetto riguardante l'algoritmo utilizzato.
Le procedure in quanto astrazioni sul controllo sono utilizzate come parti di alcune classi di moduli, che prendono il nome di librerie (ad esempio, librerie di funzioni matematiche e grafiche).

Astrazione sui Dati: ADS

Un'astrazione di dato (o Struttura Dati Astratta, ADS) è un modo di incapsulare un dato in una rappresentazione tale da regolamentarne l'accesso e la modifica.

Non sono puramente funzionali (come le librerie) perché risultati dell'attivazione di un'operazione comportano modifiche al dato (cambia lo stato).

Coesione

E' una proprietà di un'unità di realizzazione (o sottoinsieme).
L'obiettivo di un progettista è creare sistemi coesi e quindi sistemi in cui tutti gli elementi di ogni unità di progettazione siano strettamente collegati tra loro.

Ci sono vari tipi e gradi di coesione:

- Coesione funzionale: raggruppa parti che collaborano per realizzare una funzionalità (situazione ideale).

- Coesione comunicativa: tra elementi che operano sugli stessi dati di input o contribuiscono agli stessi dati di output (non buono, non supporta il riuso).
  Esempio: aggiornare il record nel database e inviarlo alla stampante.

- Coesione procedurale: tra elementi che realizzano i passi di una procedura (azioni debolmente connesse, difficile il riuso).
  Esempio: calcola la media di uno studente, stampa la media di uno studente.

- Coesione temporale: tra azioni che devono essere fatte in uno stesso arco di tempo (azioni debolmente connesse, difficile il riuso).
  Esempio: azioni da fare all'apertura dell'anno accademico
  Soluzione preferibile: dividere le azioni in diverse unità, avere una routine che manda a tutte loro un evento di avvio.

- Coesione logica: tra elementi che sono logicamente correlati e non funzionalmente (operazioni correlate ma significativamente diverse - azioni debolmente connesse, difficile il riuso).

Esempio: un componente legge input da nastro, disco e rete. Le operazioni sono correlate, ma le funzioni sono significativamente diverse.

- Coesione accidentale: tra elementi non correlati ma piazzati assieme (peggiore forma di coesione).

Disaccoppiamento (uncoupling)

E' una proprietà di un insieme di unità di progettazione con il grado in cui un'unità è legata ad altre unità.
Il progettista ha l'obiettivo di creare sistemi disaccoppiati in cui le unità non sono strettamente legate tra loro.

Solid

Cinque principi di base di progettazione e programmazione OO

Solid:

- Single Responsibility Principle: una classe (o metodo) dovrebbe avere solo un motivo per cambiare (ad esempio la responsabilità).
  Se abbiamo 2 motivi per cambiare una classe allora dobbiamo dividere la classe in altre due. La classe deve essere funzionalmente coesa.

- Open Closed Principle: estendere una classe non dovrebbe comportare modifiche alla stessa. L'entità sw devono essere aperte per estensione ma chiuse per modifiche (possibile grazie all'uso di classi astratte e concrete).

- Liskov Substitution Principle: istanze di classi derivate possono essere usate al posto di istanze della classe base.

- Interface Segregation Principle: fate interfacce a grana fine e specifiche per ogni cliente. Questo principio dice che occorre prestare attenzione al modo in cui si scrivono le interfacce (mettendo solo metodi necessari e implementandoli).

- Dependency Inversion Principle: Principio di inversione della dipendenza, programma per l'interfaccia e non per l'implementazione.
  I moduli (alto o basso livello) e i dettagli devono dipendere dalle astrazioni (non da implementazioni concrete).

Grasp: Un’altra famiglia di principi di progettazione.

Progettazione guidata dalla realizzazione dei casi d’uso

Con realizzazione del caso d’uso si intende descrivere come un particolare caso d’uso è realizzato in termini di oggetti collaborativi
Per realizzarli si usano diagrammi di interazione e pattern e assegnando responsabilità alle classi.

Assegnare Responsabilità

Legate al dominio del problema, le responsabilità sono obblighi (di comportamento) che un oggetto ha.
Ne esistono due tipi:

- Fare (fare qualcosa, iniziare azioni di altri oggetti, ...).
- Conoscere (conoscere dati privati, oggetti correlati, dati che possono calcolare, ...).

I metodi sono implementati per soddisfare le responsabilità.
Il metodo GRASP si basa sulla loro assegnazione.

Qualità Analizzate

Si valutano le caratteristiche di stili architetturali in base a:

- Disponibilità
- Fault Tolerance
- Modificabilità
- Performance (efficienza)
- Scalabilità

Scalabilità

E’ definita come la capacità di aumentare il throughput (portata) dell’applicazione in proporzione all’aumento dell’hw utilizzato per ospitare l’applicazione.

Ci sono:

- scalabilità verticale (scale up): riferita all’aggiunta di memoria e cpu a un singolo nodo (utile con i db).

- scalabilità orizzontale (scale out): riferita all’aggiunta di nodi hw (utile con applicazioni web).

Esempi Qualità

Client-Server, 2 o N-tier

Disponibilità: I server di ogni tier(ordine, fila) possono essere replicati, quindi se uno fallisce c’è solo una minor QoS.

Fault Tolerance: Se un cliente sta comunicando con un server che fallisce, la maggior parte dei server reindirizza la richiesta a un server replicato in modo trasparente all’utente.

Modificabilità: Il disaccoppiamento e la coesione tipici di questa arch. favoriscono la modificabilità.

Performance: Performance ok, ma da tenere sott’occhio: numero di threads paralleli su ogni server, velocità delle comunicazioni tra server, volume dati scambiato.

Scalabilità: Basta replicare i server in ogni tier (quindi ok scale out). Unico collo di bottiglia l’eventuale base di dati che scala male orizzontalmente.

Pipes and Filters

Disponibilità: Avendo "pezzi di ricambio" (componenti e possibilità di connetterle) sufficienti a formare una catena.

Fault Tolerance: Occorre riparare una catena interrotta usando componenti replica.

Modificabilità: Si, se le modifiche interessano una o comunque poche componenti.

Performance: Dipende dalla capacità del canale di comunicazione e dalla performance del filtro più lento.

Scalabilità: Ok anche scale out.

Publish-Subscribe

Disponibilità: Si possono creare clusters di dispatcher.

Fault Tolerance: Si cerca un dispatcher replica.

Modificabilità: Si possono aggiungere publisher e subscribers a piacere. Unica attenzione al formato dei messaggi.

Performance: Ok. Ma compromesso tra velocità e altri requisiti tipo affidabilità e/o sicurezza.

Scalabilità: Ok scale out: con un cluster di dispatchers si può gestire un volume molto elevato di messaggi.

P2P

Disponibilità: Dipende dal numero di nodi in rete, ma si assume si.

Fault Tolerance: Gratis.

Modificabilità: Si, se dell'architettura interessa solo la parte di comunicazione.

Performance: Dipende dal numero di nodi connessi, dalla rete, dagli algoritmi. Per esempio BitTorrent ottimizza scaricando per primo il file/pezzo più raro.

Scalabilità: Gratis.

Coordinatore di Processi

Disponibilità: Il coordinatore è un punto critico: deve essere replicato se si vuole garantire disponibilità.

Fault Tolerance: Occorre specificare compensazione (cosa fare se un server fallisce).
Se fallisce il coordinatore: occorre ridirigere su un coordinatore replica.

Modificabilità: Posso modificare liberamente i servers purché non cambino le funzionalità esportate.

Performance: Il coordinatore deve essere in grado di servire più richieste concorrenti. La performance del processo è limitata dal server più lento. Se non tutti i server sono necessari, si usa un time-out.

Scalabilità: Replicando il coordinatore. Scala sia up che out.