Principi di Design

- Linee guida per la progettazione software, spesso con radici remote
- Structured Design di Edward Yourdon e Larry Constantine (1975)
  - Definizione dettagliata di accoppiamento e coesione
- The Mythical Man-Month di Frederick Brooks (1975)
  - Introduce la legge di Brooks, rende popolare la legge di Conway
- 201 Principles of Software Development, Alan M. Davis (1995)
  - Una raccolta di principi con relativa succinta spiegazione
- The Pragmatic Programmer di Andrew Hunt, David Thomas (1999 / 2019)
  - Introduce i principi DRY, ETC
- Extreme Programming Explained di Kent Beck, Cynthia Andres (2000 / 2004)
  - Introduce il principio YAGNI
- Agile Software Development, Principles, Patterns, and Practices di Robert C. Martin (2002)
  - Introduce l'acronimo SOLID per cinque importanti principi
  - Revisione completa → Clean Architecture: A Craftsman's Guide to Software Structure and Design (2017)

ETC – Easy To Change

- È un meta-principio, alla base degli altri principi
  - Andrew Hunt e David Thomas (2019)
- Tutto cambia nello sviluppo software. I requisiti, il design, il team. Dobbiamo essere pronti ad adattarci ai cambiamenti
  - Vedi anche Extreme Programming, Kent Beck et al. (1999)
- Un buon design porta a scrivere codice facilmente modificabile
  - Vedi anche principi di basso accoppiamento e alta coesione

KISS – Keep it simple

- Preferisci il codice semplice
  - Più facile da scrivere, leggere, capire, modificare, spesso più elegante e più veloce in esecuzione
  - Più difficile che sia “bacato” e più facile da debuggare
- “A parità di condizioni, è preferibile la soluzione più semplice”
  - Principio di parsimonia – rasoio di Occam
- “Less is more”
  - Peter Behrens / Ludwig Mies van der Rohe ~1910
- “Make everything as simple as possible, but not simpler”
  - Albert Einstein, Oxford 1933
- “Make Simple Tasks Simple!”
  - Bjarne Stroustrup, CppCon 2014

Legge di Murphy

- Se qualcosa può andare storto, lo farà – Arthur Block 1977
  - Il Design for Errors è una sua formalizzazione, Alan Davis 1995
- Cerca di identificare tutto ciò che può andare storto prima che il programma sia rilasciato
- Potrebbe essere una buona idea usare:
  - Linguaggi di programmazione compilati e staticamente tipizzati
  - Un design basato su solidi principi e pattern
- Quando non è possibile impedire situazioni di errore → Fail Fast
- La robustezza del codice non dovrebbe andare a scapito della sua semplicità

DRY – Don’t repeat yourself

- Ogni elemento di conoscenza deve avere una singola, non ambigua e autorevole rappresentazione nel sistema
  - Andrew Hunt e David Thomas (1999)
- Le ripetizioni sono inerentemente pericolose – non solo nel codice
  - Es. un commento che è una parafrasi del codice
    - In seguito a un cambiamento del codice può diventare dannoso
  - Un cambiamento impatta su ogni duplicazione
    - Basta dimenticare di aggiornarne una per introdurre un errore
- Vedi l’approccio matematico del ricondursi a un caso precedente

Divide et Impera

- Affronta un problema dividendolo in sottoproblemi più semplici
  - La combinazione delle soluzioni parziali porta al risultato atteso
- Conduce spesso ad algoritmi ricorsivi, come mergesort, quicksort, …
- L’approccio è simile a quello delle dimostrazioni per induzione in matematica
  - Può portare a soluzioni eleganti, efficienti, facilmente parallelizzabili
- Richiede attenzione nell’identificare i casi base
  - Sottoproblemi considerati sufficientemente semplici da essere risolti direttamente
- Può essere utile tener traccia dei sottoproblemi generati
  - Per evitare duplicazioni inutili
  - Vedi ad esempio memoization o dynamic programming

Separation of Concern

- Dividi un programma in **moduli**, seguendo un approccio divide et impera
- Ogni modulo è relativo a uno specifico compito (interesse, ambito)
  - Un modulo può essere suddiviso in sottomoduli più specifici
- I moduli incapsulano informazioni e le rendono disponibili via una interfaccia ben definita
  - È quindi alla base del principio dell'Information Hiding
- Il codice che segue questo principio è più semplice da scrivere e mantenere
- Ad esempio, nello sviluppo Web si usano linguaggi diversi per gli ambiti specifici
  - HTML: struttura e contenuto
  - CSS: stile
  - JavaScript: interattività
- Importanti contributi sul tema già da Edsger W. Dijkstra (1974)

Low Coupling – High Cohesion

- L’accoppiamento tra moduli dovrebbe essere basso
  - Più due moduli sono correlati, più alto è il loro accoppiamento
  - Più alto è l’accoppiamento tra due moduli, più il cambiamento di uno si può propagare all’altro
- La coesione interna di un modulo dovrebbe essere alta
  - Più le componenti di un modulo sono correlate, più alta è la sua coesione
  - Una bassa coesione implica una alta complessità dei cambiamenti nel modulo
- Un basso accoppiamento e un alta coesione rendono più facili i cambiamenti
  - Sono forze contrastanti, occorre trovare il giusto bilanciamento
  - Potrebbero portare a una maggior complessità del sistema (in contrasto con KISS)
- Structured Design, Edward Yourdon e Larry Constantine (1975)
- Nel Pragmatic Programmer è definito come ortogonalità del sistema
  - E il disaccoppiamento è reso con una metaforica “timidezza” del modulo

Legge di Conway

- Il design di un sistema è determinato dalla struttura della comunicazione nell'organizzazione che lo progetta
  - Melvin Conway (1967)
- Il livello di accoppiamento tra moduli è determinato dal livello di comunicazione tra i responsabili del loro sviluppo
  - Se la comunicazione tra i responsabili dello sviluppo è facile, l'accoppiamento dei rispettivi moduli non viene percepito come un problema
- Una organizzazione più piccola tende ad essere più coesa ed efficace
  - Ottenendo risultati migliori in tempi più rapidi
- Accertati che l'organizzazione sia compatibile con l'architettura del prodotto
  - James Coplien in Organizational Patterns of Agile Software Development (2004)

Legge di Brooks

- L’aggiunta di personale a un progetto in ritardo ne aumenta il ritardo
  - Frederick Brooks, 1975, come *“oltraggiosa sovrasemplificazione”*
- I nuovi arrivati inizialmente riducono la produttività del sistema
  - Devono documentarsi su quanto è già stato fatto
  - Rallentano il lavoro degli altri partecipanti, che li devono supportare
- La mancanza di conoscenza porta alla produzione di codice di bassa qualità
  - Con associato aumento dei tempi di testing e debugging
- La comunicazione all’interno del sistema può diventare un problema
  - La relazione tra aumento del personale e i canali di comunicazione è quadratica
    - Se raddoppia il personale possiamo aspettarci che i tempi di comunicazione quadruplichino
  - Con le conseguenze evidenziate dalla legge di Conway

YAGNI

- You Ain’t Gonna Need It – non ne avrai bisogno (Kent Beck)
  - È inutile spendere tempo nello scrivere codice che non è richiesto
- Indica la pratica Extreme Programming del Simple Design
  - Fai la cosa più semplice che possa funzionare
  - Simile a KISS
- Riduce la complessità del codebase corrente, semplifica il testing
  - “Se non c’è non si rompe”
- Vedi anche Incremental Design nell’approccio agile

Design by contract

- Bertrand Meyer (1986), marchio registrato da Eiffel Software
  - Basato su lavori formali, come la logica di Hoare (1969)
- Ogni modulo specifica la sua interfaccia in modo formale, preciso, verificabile
  - Maggiori sono i limiti imposti, più semplice è il suo compito
- Ogni modulo rispetta un contratto in cui sono definite
  - Precondizioni: vere prima dell'esecuzione
    - Questa verifica può essere delegata al chiamante, se la possibile debolezza risultante è accettabile
  - Postcondizioni: vere al termine dell'esecuzione
  - Invarianti: sempre vere
- Alcuni linguaggi di programmazione supportano direttamente questo principio (Eiffel)
- Un approccio comune in altri linguaggi richiede l'uso di asserzioni (C / C++ / Java …)

Single Responsibility

- Primo principio **S**OLID
  - Robert C. Martin (2003)
- Deriva dalla legge di Conway e dalla Separation of Concern
  - Il modulo deve avere un solo attore (nel senso dello Use Case UML)
- Un modulo ha una sola responsabilità
  - Più semplice da capire, implementare, testare
  - Implica disaccoppiamento da altri moduli, maggior coesione del modulo
- Una richiesta di cambiamento dovrebbe impattare un solo modulo
  - Più localizzata, dovrebbe portare a cambiamenti più limitati nel codice
- Può portare alla creazione di molti piccoli moduli (vedi Façade per una soluzione)

Open/Closed

- Secondo principio SOLID
  - Object-Oriented Software Construction, Bertrand Meyer (1988, 1997)
- Una volta rilasciato, un modulo deve essere
  - Capace di evolvere, e dunque **aperto** all'estensione
  - Stabile nell'uso, e dunque **chiuso** al cambiamento *(se non per il bug fixing)*
- L'apertura può essere ottenuta via ereditarietà o aggregazione
  - Determinando un forte accoppiamento con la classe base
- Oppure si può abbinare il principio dell'Interface Segregation
  - Introducendo un livello di astrazione aggiuntivo

Liskov Substitution

- Terzo principio SOLID
  - Specifico per il design Object Oriented, Barbara Liskov (1988)
  - Simile a quanto stabilito dal design by contract riguardo all'ereditarietà
- Dove è atteso un oggetto di una certa classe, può essere usato un oggetto di classe derivata
- Rispetto alla classe base, nelle classi derivate:
  - Le precondizioni non possono essere più forti
  - Le postcondizioni non possono essere più deboli
  - Gli invarianti devono essere mantenuti
  - Le eventuali eccezioni generate non possono essere di tipo più specifico

Interface Segregation

- Quarto principio SOLID
  - Robert C. Martin (2003)
- Il chiamante non deve dipendere da metodi del chiamato che non usa
  - Un modulo dall’interfaccia troppo densa è di difficile comprensione
  - Simile a Single Responsibility, dal punto di vista del chiamante
- Si usino piccole interfacce che definiscono “ruoli”
  - Una classe concreta può implementare più interfacce
    - I suoi oggetti possono interpretare più ruoli
  - Un oggetto è gestito via un interfaccia specifica, il ruolo di interesse

Dependency Inversion

- Quinto principio SOLID – Robert C. Martin (1996)
- I moduli di alto livello non devono dipendere da quelli di basso livello
  - I dettagli di entrambi dovrebbero dipendere da astrazioni
- I dettagli implementativi dovrebbero essere definiti in termini astratti
  - Facendo uso di interfacce o classi astratte
    - Riduzione dell'accoppiamento
    - Maggior localizzazione dei cambiamenti

- Correlato a Inversion of Control
  - Enfasi sulla divisione dei compiti e debolezza della connessione tra i moduli