

La qualità di un'applicazione dipende da specifiche priorità organizzative.

Un buon progetto deve essere il più affidabile, efficiente e manutenibile possibile, oltre ad essere poco costoso.

Un brutto progetto invece può essere:

Rigido: è difficile da modificare, tant'è che una singola modifica porta a una cascata di cambiamenti nei moduli dipendenti.

Fragile: potrebbe rompersi inaspettatamente da qualche parte ogni volta che avviene un cambiamento. E' impossibile mantenere questo software: è meglio reingegnerizzarlo piuttosto che modificarlo.

Immobile: non si può riutilizzare perché per non scrivere qualche metodo ci si deve portare dietro un bagaglio non indifferente di dipendenze, quindi non si riutilizza niente e si scrive il codice da capo.

Viscoso: quando si scopre un problema ci sono 2 modi per metterlo a posto: un modo semplice e veloce (che mantiene il design) e uno più laborioso che modifica tutta l'architettura. E' ovvio che il secondo metodo è più stabile e laborioso.

Principi di design:

Single Responsibility Principle: ogni classe deve avere una ed una sola responsabilità: se ne ha più di una, è necessario scindere le classi.

Open/Close Principle: ogni classe deve essere aperta alle estensioni e chiusa alle modifiche. Questo significa che deve poter essere implementata a piacere, ma senza modificare il codice sorgente. Questo può avvenire tramite l'utilizzo di interfacce o classi astratte che vengono implementate da classi concrete che, anche se vengono modificate, non obbligano a ricompilare tutto (interfaccia più tutte le classi che la implementano) ma solamente la classe interessata.

Liskov Substitution Principle: deve essere possibile per il cliente utilizzare le sottoclassi senza accorgersene, continuando a funzionare correttamente. Questo accade perché nelle sottoclassi le precondizioni devono essere meno stringenti (o uguali) e le post-condizioni sono più stringenti (o uguali) per permettere al cliente di non notare la differenza: se il chiamante garantisce il verificarsi delle precondizioni, il metodo deve garantire, a meno di eccezioni, il verificarsi delle post-condizioni.

Dependency Inversion Principle: i moduli di più basso livello, che dipendono da moduli di più alto livello che dipendono a loro volta da astrazioni, sono i più soggetti ai cambiamenti e possono essere tranquillamente modificati perché non c'è nulla che dipende da essi. In questo modo si è invertita la direzione delle dipendenze: i cambiamenti si propagano dal basso (classi concrete) verso l'alto (classi di più alto livello, fino alle astrazioni). In questo modo i dettagli sono isolati tra di loro e permettono che i cambiamenti si propaghino tra le classi e le dipendenze, e permette una maggiore riusabilità dei moduli.

Interface Segregation Principle: è inutile che tanti client dipendano da un'unica interfaccia, quando non ne utilizzano tutti i metodi: è quindi più utile avere diversi clienti che dipendono da diverse interfacce specializzate che da un'unica interfaccia general purpose. In questo caso, quando avvengono cambiamenti nelle interfacce, solo le classi interessate subiscono dei cambiamenti/aggiunte, e quelle che non implementano/utilizzano quei metodi non si accorgono di nulla.