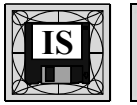
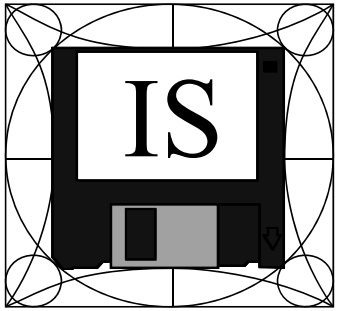


Progettazione *software*



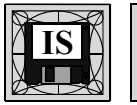
Ingegneria del Software

V. Ambriola, G.A. Cignoni
C. Montangero, L. Semini

Aggiornamenti di: T. Vardanega (UniPD)

Progettare prima di produrre

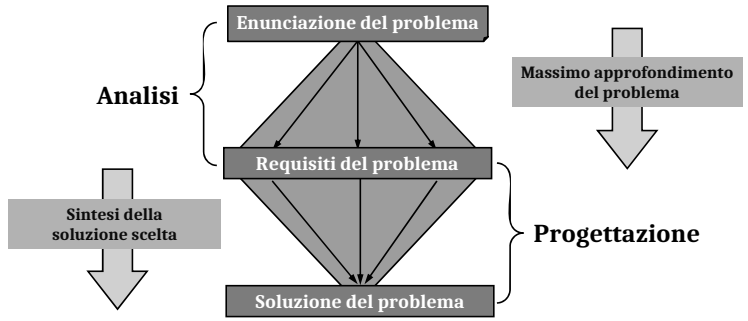
- La progettazione precede la produzione
  - Perseguendo la correttezza per costruzione
  - Invece che inseguendo la correttezza per correzione
- Progettare per
  - Dominare la complessità del prodotto ("*divide-et-impera*")
  - Organizzare e ripartire le responsabilità di realizzazione
  - Produrre in economia (efficienza)
  - Garantire qualità (efficacia)



Dall'analisi alla progettazione – 1

- L'analisi risponde alle domande: qual'è il problema / quale la cosa giusta da fare?
  - La risposta richiede
    - Comprensione del dominio
    - Discernimento di obiettivi, vincoli e requisiti
  - Attuazione di un approccio investigativo
- La progettazione risponde alla domanda: come farla giusta?
  - Per rispondere serve definire una soluzione soddisfacente per tutti gli *stakeholder*
    - Per farlo, non serve ancora il codice
    - I suoi prodotti documentano l'**architettura** scelta e i suoi modelli logici
  - Attuazione di un approccio sintetico

Dall'analisi alla progettazione – 2



Progettazione *software*

Dall'analisi alla progettazione – 3

- In "*On the role of scientific thought*" (1982), Edsger W. Dijkstra dice
  - *The task of "making a thing satisfying our needs", as a single responsibility, is split into two parts*
  1. *Stating the properties of a thing, by virtue of which it would satisfy our needs, and*
  2. *Making a thing that is guaranteed to have the stated properties*
- La prima (parte di) responsabilità è dell'analisi
- La seconda è di progettazione e codifica

Progettazione *software*

Obiettivi della progettazione – 1

- Soddisfare i requisiti con un sistema di qualità
- Definendo l'architettura logica del prodotto
  - Impiegando parti con specifica chiara e coesa
  - Realizzabili con risorse sostenibili e costi fissati
  - Organizzate in modo da facilitare cambiamenti futuri
- La scelta di una buona architettura facilita il successo
- Identificare soluzioni architetturali utili al caso e con parti riusabili

Progettazione *software*

Obiettivi della progettazione – 2

- Dominare la complessità del sistema
  - Suddividendo il sistema fino a che ciascuna sua parte abbia complessità trattabile
  - Perché la codifica di ogni singola parte possa essere compito rapido, fattibile e verificabile di un singolo individuo
- Conviene spingere la progettazione nel dettaglio
  - Capendo quando fermarsi
  - Nel momento in cui il costo di coordinamento delle parti supera il beneficio (di semplificazione) della ulteriore suddivisione
  - Più minute le componenti più complessa la loro orchestrazione

Progettazione *software*

Arte vs. architettura

- Ca. 1915, lo scrittore H.G. Wells (1866-1946), autore, tra l'altro, di "*The War of the Worlds*" (1898), scrive al collega H. James (1843-1916)

> *To you, literature – like painting – is an end, to me, literature – like architecture – is a means, it has a use*

- L'arte è un fine, l'architettura un mezzo

Una definizione di architettura

- La <u>decomposizione</u> del sistema in componenti
- L'<u>organizzazione</u> di tali componenti
  - Definizione di ruoli, responsabilità, interazioni (chi fa cosa e come)
- Le <u>interfacce</u> necessarie all'interazione tra le componenti tra loro e con l'ambiente
  - Come le componenti possono collaborare
- I <u>paradigmi di composizione</u> delle componenti
  - Regole, criteri, limiti, vincoli (anche a fini di manutenibilità)

ISO/IEC/IEEE 42010:2011 Systems and software engineering – Architecture description

Criteri guida

- Esistono più stili architetturali
  - Aderire a uno stile garantisce coerenza e consistenza
  - Le scelte architetturali determinano l'organizzazione dell'informazione e l'interazione tra le parti
- Molta letteratura sul tema
  - P.es.: https://msdn.microsoft.com/en-us/library/ee658098.aspx

An architectural style is a named collection of architectural design decisions that
- *are applicable in a given development context*
- *constrain architectural design decisions that are specific to a particular system within that context*
- *elicit beneficial qualities in each resulting system*

Qualità di una buona architettura – 1

- Sufficienza
  - È capace di soddisfare tutti i requisiti
- Comprensibilità
  - Può essere capita dagli *stakeholder*
- Modularità
  - È suddivisa in parti chiare e ben distinte
- Robustezza
  - È capace di sopportare ingressi diversi (giusti, sbagliati, tanti, pochi) dall'utente e dall'ambiente

Secondo D. Parnas (1972) ci sono due strade per "modularizzare"
1. Suddividere l'attività nei suoi blocchi logici principali (p.es. gli stadi di una *pipeline*)
2. Perseguire *information hiding*

La seconda si preoccupa di ridurre le perturbazione esterne causate da cambiamenti interni
La prima non ne è capace

Qualità di una buona architettura – 2

- Flessibilità
  - Permette modifiche a costo contenuto al variare dei requisiti
- Riusabilità
  - Sue parti possono essere utilmente impiegate in altre applicazioni
- Efficienza
  - Nel tempo, nello spazio, nelle comunicazioni
- Affidabilità (*reliability*)
  - È altamente probabile che svolga bene il suo compito quando utilizzata

Qualità di una buona architettura – 3

- **Disponibilità (*availability*)**
  - **Necessita di poco o nullo tempo di indisponibilità totale per manutenzione**
    - Non tutto il sistema deve essere interrotto se qualche sua parte è sotto intervento
- **Sicurezza rispetto a malfunzionamenti (*safety*)**
  - **È esente da malfunzionamenti gravi**
    - Il sistema dispone di un sufficiente grado di ridondanza per restare utilmente operativo anche in presenza di guasti locali
- **Sicurezza rispetto a intrusioni (*security*)**
  - **I suoi dati e le sue funzioni non sono vulnerabili a intrusioni**

Qualità di una buona architettura – 4

- **Semplicità** → Vedi approfondimenti
  - **Ogni parte contiene solo il necessario e niente di superfluo**
- **Incapsulazione (*information hiding*)** → Vedi approfondimenti
  - **L'interno delle componenti non è visibile dall'esterno**
- **Coesione** → Vedi approfondimenti
  - **Le parti che stanno insieme hanno gli stessi obiettivi**
- **Basso accoppiamento** → Vedi approfondimenti
  - **Parti distinte dipendono poco o niente le une dalle altre**

Semplicità

- **William Ockham (1285-1347/49)**

  - **"Pluralitas non est ponenda sine necessitate"**
  - **Le entità usate [per spiegare un fenomeno] non devono essere moltiplicate senza necessità**
  - **Principio noto come "il rasoio di Occam"**
- **Adottato da Isaac Newton (1643-1727) nella fisica**

  - **"*We are to admit no more causes of natural things than such that are both true and sufficient to explain their appearances*"**
  - **Quando hai due soluzioni equivalenti rispetto ai risultati scegli, la più semplice**
- **E poi anche da Albert Einstein (1879-1955)**
  - **"*Everything should be made as simple as possible, but not simpler*"**

Incapsulazione

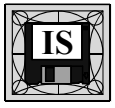
- **Le componenti sono "*black box*"**
  - **I suoi clienti ne conoscono solo l'interfaccia**
- **La loro specifica nasconde**
  - **Gli algoritmi e le strutture dati usati al loro interno**
- **Benefici**
  - **L'esterno non può fare assunzioni sull'interno**
  - **Cresce la manutenibilità**
  - **Al diminuire delle dipendenze aumenta la possibiltà di riuso**

Coesione – 1

- Proprietà interna di singole componenti
  - Funzionalità "vicine" devono stare nella stessa componente
  - La modularità spinge a decomporre il grande in piccolo
  - La ricerca di coesione fornisce un criterio di decomposizione e anche un limite inferiore a essa
- Va massimizzata per ottenere
  - Maggiore manutenibilità e riusabilità
  - Minore interdipendenza fra componenti
  - Maggiore comprensibilità dell'architettura del sistema

Coesione – 2

- Vi sono svariati tipi di coesione buona
- Funzionale, quando le parti concorrono al medesimo specifico compito
- Sequenziale, quando alcune azioni sono «vicine» ad altre per ordine di esecuzione e dunque conviene tenerle insieme
- Informativa, quando le parti agiscono sulla stessa unità di informazione
- Su tutte prevale quella che persegue *information hiding*

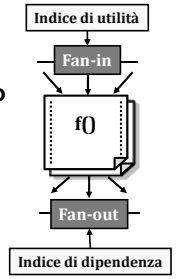


Accoppiamento – 1

- Parti diverse possono incorrere in un grado di dipendenza reciproca cattiva
  - Facendo assunzioni dall'esterno su come certe cose siano all'interno di altre (variabili, tipi, indirizzi, …)
  - Imponendo vincoli dall'esterno sull'interno di una parte (per ordine di azioni, uso di certi dati, formati, valori)
  - Condividendo frammenti delle stesse risorse (strutture dati)
- Un sistema è un insieme organizzato che ha bisogno di tutte le sue parti
  - E quindi ha un po' di accoppiamento
  - Ma la buona progettazione lo tiene basso

Accoppiamento – 2

- Proprietà esterna di componenti
  - Il grado U di utilizzo reciproco di M componenti
  - $U = M \times M$ è il massimo grado di accoppiamento
  - $U = \emptyset$ ne è il minimo
- Metriche: *fan-in* e *fan-out* strutturale
  - SFIN è indice di utilità → massimizzare
  - SFOUT è indice di dipendenza → minimizzare
- La buona progettazione produce componenti con SFIN elevato

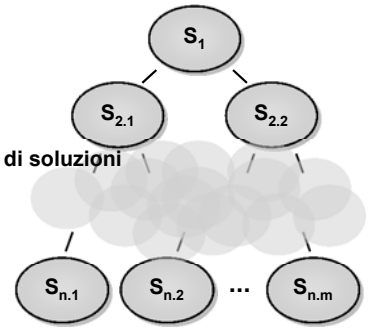


Riuso

- Capitalizzare sottosistemi già esistenti
  - Impiegandoli più volte per più prodotti
  - Ottenendo minor costo realizzativo
  - Ottenendo minor costo di verifica
- Problemi
  - Progettare per riuso è più difficile
    - Bisogna anticipare bisogni futuri
  - Progettare con riuso non è immediato
    - Bisogna minimizzare le modifiche alle componenti riusati per non perderne il valore
- Puro costo nel breve periodo
  - Diventa risparmio nel medio termine (quindi è un investimento)

Progettazione architetturale

- *Top-down* ↓
  - Decomposizione di problemi
  - Stile funzionale
- *Bottom-up* ↑
  - Composizione e specializzazione di soluzioni
  - Stile *object-oriented*
- *Meet-in-the-middle* ↑↓
  - Approccio intermedio
  - Il più frequentemente usato



Framework

- Insieme integrato di componenti SW prefabbricate
  - Nel mondo pre-OO erano chiamate librerie
  - Sono *bottom-up* perché fatti di codice già sviluppato
  - Sono anche *top-down* se impongono uno stile architetturale
- Utilissimi come base facilmente riusabile di diverse applicazioni entro un dato dominio
  - Molti importanti esempi nel mondo J2EE e JS
    - Spring (http://www.springsource.org/about) per architetture di business con MVC
    - Struts (http://struts.apache.org/) per Web Apps in stile MVC
    - Swing per GUI, ecc.

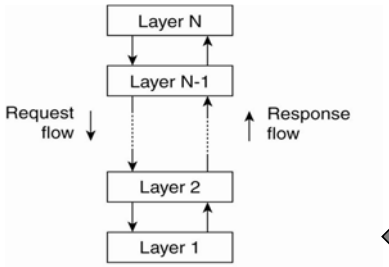
Design pattern architetturali

- Soluzione progettuale a problema ricorrente
  - Organizza una responsabilità architetturale lasciando gradi di libertà nel suo uso
    - Richiede precisa corrispondenza nel codice realizzativo
  - Rappresenta il corrispondente architetturale degli algoritmi
    - Che invece specificano procedimenti di soluzione
- Concetto promosso da C. Alexander (un vero architetto)
  - *The Timeless Way of Building*, Oxford University Press, 1979
- Rilevante nel SW a partire dalla pubblicazione di "*Design Patterns*" della GoF

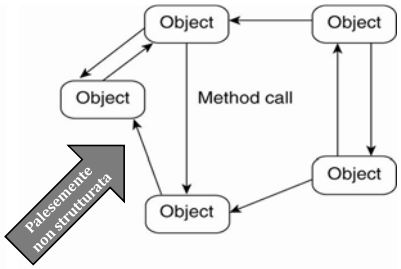
Pattern architetturali – 1

- Architettura "three-tier" (a livelli)
  - Strato della presentazione (GUI)
  - Strato della logica operativa (*business logic*)
  - Strato dell'organizzazione dei dati (DB)
- Variante multilivello (pila OSI e TCP/IP)
- Architettura produttore-consumatore
  - Collaborazione a *pipeline*

Esempi – 1



Architettura multilivello

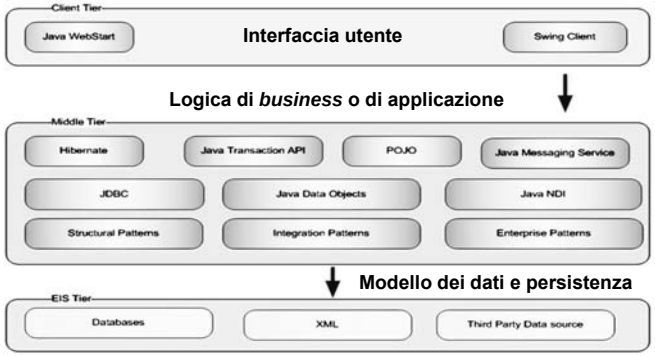


Architettura a oggetti

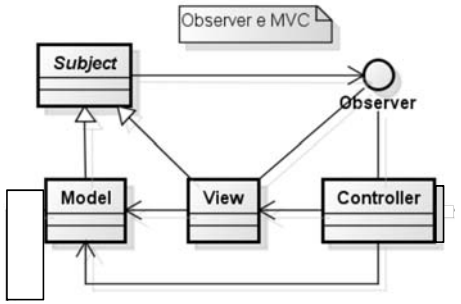
Tratto da: Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Esempi – 2

Architettura a 3 livelli



Esempi – 3



Questo pattern sarà oggetto di studio specifico

Progettazione di dettaglio: attività

- Definizione delle unità realizzative (moduli)
  - Un carico di lavoro realizzabile dal singolo programmatore
  - Corrispondente a una funzionalità (o responsabilità) ben definita
    - Componente terminale (non ulteriormente decomponibile) o loro aggregato
    - Insieme di entità (tipi, dati, operazioni) strettamente correlate
    - Raccolte insieme in un aggregato con chiara identità (p.es. *package*)
- Specifica delle unità come insieme di moduli
  - La corrispondenza è determinata dalle caratteristiche del linguaggio di programmazione utilizzato
  - Il modulo è la più piccola entità strutturale utilmente rappresentabile
- Ex-novo oppure tramite specializzazione di entità esistenti

Progettazione di dettaglio: obiettivi

- Le unità dell'architettura di dettaglio realizzano le componenti dell'architettura logica
  - Passo di decomposizione per organizzare il lavoro di programmazione
  - Tracciare la corrispondenza assicura congruenza con l'architettura di sistema
- Produrre la documentazione necessaria alla specifica di ogni unità
  - Perché la programmazione possa procedere in modo certo e disciplinato
  - Per assicurare tracciamento di requisiti alle unità
  - Per definire le configurazioni ammissibili del sistema
- Definire gli strumenti per le prove di unità
  - Casi di prova e componenti ausiliarie per la verifica unitaria e di integrazione

Documentazione

- IEEE 1016:1998 *Software Design Document*
  - Introduzione
    - Come nel documento AR (*Software Requirements Specification*)
  - Riferimenti normativi e informativi
  - Descrizione della decomposizione architetturale
    - Componenti (visione statica), processi (visione dinamica), dati
  - Descrizione delle dipendenze (tra componenti, processi, dati)
  - Descrizione delle interfacce (tra componenti, processi, dati)
  - Descrizione della progettazione di dettaglio

Stati di progresso per SEMAT – 1

- *Architecture selected*
  - Selezione di una architettura tecnicamente adatta al problema: accordo sui criteri di selezione
  - Selezione delle tecnologie necessarie
  - Decisioni su *buy, build, make*
- *Demonstrable*
  - Dimostrazione delle principali caratteristiche dell'architettura: gli *stakeholder* concordano
  - Decisione sulle principali interfacce e configurazioni di sistema

Stati di progresso per SEMAT – 2

- *Usable*
  - Il sistema è utilizzabile e ha le caratteristiche desiderate
  - Il sistema può essere operato dagli utenti
  - Le funzionalità e le prestazioni richieste sono state verificate e validate
  - La quantità di difetti residui è accettabile
- *Ready*
  - La documentazione per l'utente è pronta
  - Gli *stakeholder* hanno accettato il prodotto e vogliono che diventi operativo

Riferimenti

- D. Budgen, Software Design, Addison-Wesley
- C. Alexander, The origins of pattern theory, IEEE Software, settembre/ottobre 1999
- G. Booch, Object-oriented analysis and design, Addison-Wesley
- G. Booch, J. Rumbaugh, I. Jacobson, The UML user guide, Addison-Wesley
- C. Hofmeister, R. Nord, D. Soni, Applied Software Architecture, Addison-Wesley, 2000
- P. Krutchen, The Rational Unified Process, Addison-Wesley