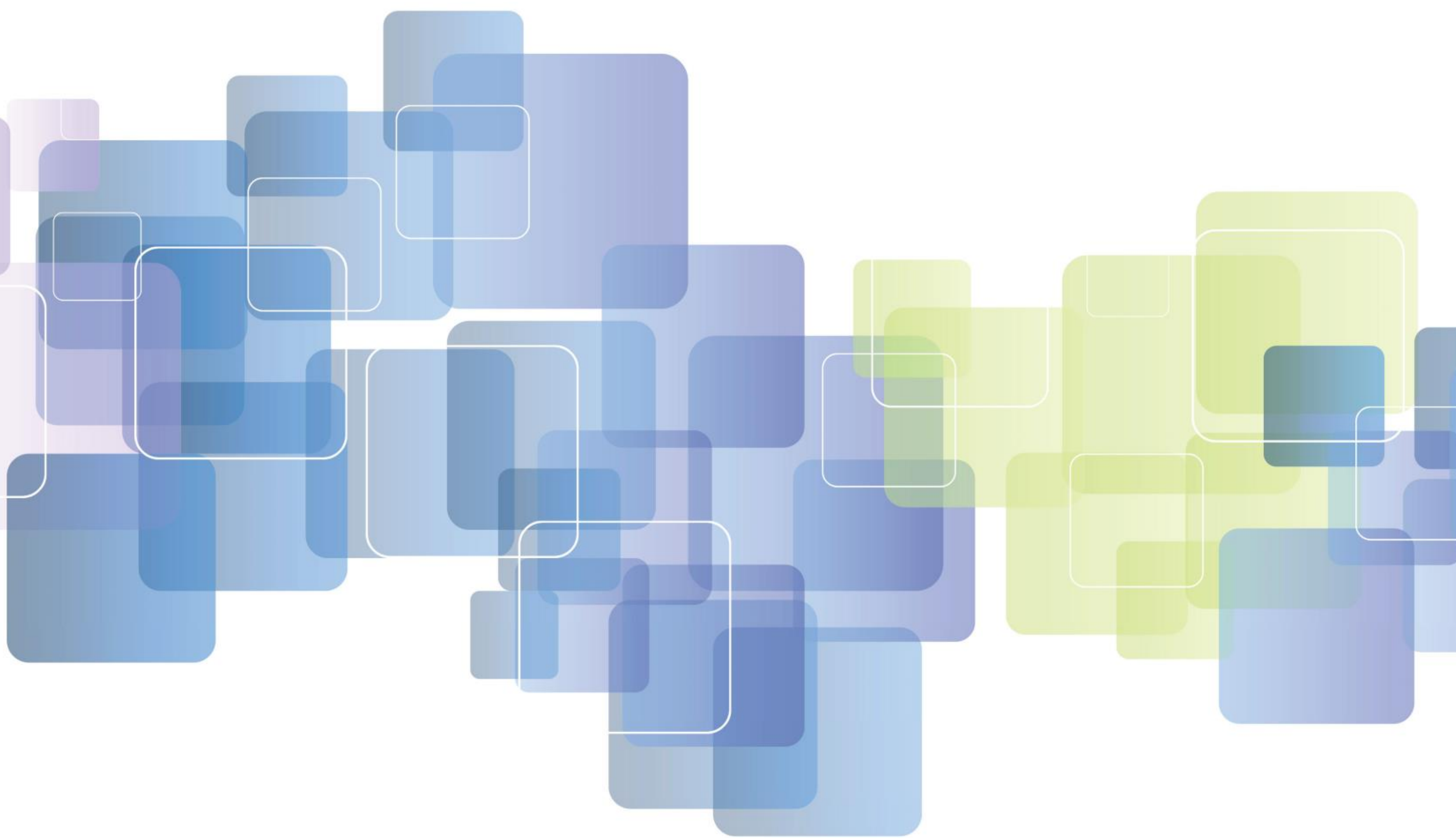
TecnetDati

Linee guida per la definizione dell'architettura logica di un'applicazione



L’importanza della architettura per la realizzazione di applicazioni solide!

Lo sviluppo del software ha visto, negli ultimi anni, un profondo cambiamento di prospettiva.
La realizzazione di applicazioni in ambienti centralizzati, monopiattaforma, con cicli di vita nell’ordine dei lustri è ormai un ricordo: oggi lo sviluppo avviene in ambienti distribuiti, in contesti tecnologici eterogenei e meno stabili che in passato, con necessità di rilascio pressanti e requisiti in continua evoluzione.
Ciò ha portato all’elaborazione di nuovi modi di realizzare le applicazioni, per cui spesso si parla di processo iterativo e incrementale, approccio object oriented all’analisi ed al disegno, sviluppo basato sui casi d’uso ecc.
Un possibile filo conduttore che accomuna i nuovi paradigmi di sviluppo è la ricerca di una valida architettura applicativa: in pratica oggi non è più sufficiente elaborare una soluzione software rispondente ai requisiti ma occorre, in aggiunta, realizzare una architettura applicativa robusta.
Gli scenari da affrontare spesso prevedono un rapido cambiamento dei requisiti da soddisfare, la necessità di far evolvere le applicazioni a partire da un nucleo iniziale, la distribuzione del carico elaborativo su più di una piattaforma, l’integrazione di componenti acquisite esternamente.
Per affrontare queste problematiche cercando, come si dice, di “uscirne vivi”, occorre strutturare bene l’applicazione (o, il altri termini, creare una valida architettura), in modo tale da poter accogliere i cambiamenti che tipicamente intercorrono, sia nei requisiti che nel contesto tecnologico, con un impatto il più limitato possibile su quanto già realizzato.

Caratteristiche generali di una buona architettura

C’è un sostanziale accordo sul principio secondo cui una valida architettura software è organizzata in strati di sottosistemi.
Ciò significa pensare l’applicazione come un insieme di strati logici, in cui ogni strato fornisce dei servizi a strati di più alto livello (tra cui, al top, c’è lo strato applicativo che fornisce i servizi agli utilizzatori del sistema) e, a sua volta, si appoggia ai servizi di strati di più basso livello.
Ogni strato, perché sia ben progettato, deve contenere sottosistemi che condividono lo stesso livello di astrazione e la stessa stabilità di interfaccia: gli strati più alti sono specifici di una o alcune applicazioni, sono soggetti a frequenti modifiche e vengono realizzati basandosi sugli strati a più basso livello che, invece, sono comuni a più applicazioni e tendenzialmente più stabili.

Tipi di strati

Definiamo, in generale, almeno tre strati in cui viene organizzata una applicazione:
- Lo strato dei sottosistemi specifici della applicazione, o *strato applicativo*
- Lo strato dei sottosistemi generali per più applicazioni (ad es. sicurezza, gestione errori, …)
- Lo strato dei servizi di base (ad es. middleware, GUI, DBMS, …)

Sottosistemi applicativi (es. gestione ordini, gestione fatturazione, gestione anagrafica clienti ecc.)	**Strato applicativo**
Sottosistemi indipendenti dal dominio (es. sicurezza, gestione errori, help, …)	**Strato dei sottosistemi generali**
GUI DBMS Middleware Networking Sistemi Operativi	**Strato dei servizi di base**

Per cercare di chiarire meglio il concetto, illustriamo un esempio tratto da una applicazione reale (che riguarda la gestione di tipologie di contratti), in cui gli strati logici ed i sottosistemi vengono rappresentati con il costrutto UML del package e le frecce tratteggiate tra i package rappresentano le relazioni di dipendenza tra i sottosistemi stessi.

Strato applicativo
Anagrafiche
Gestione contratto/variante
Inquiry e report
Gestione parametri contratto
Elaborazione flussi periodici

Servizi generali
Gestione errori
Gestione help
Gestione sicurezza

Servizi di base
Oracle
Java Virtual Machine
Java applet

Ben lungi dall'essere ottenuta gratis, l'architettura a strati di sottosistemi è un risultato della progettazione: se è vero che una GUI (Graphical User Interface) può facilmente essere ricondotta ad un sottosistema dello strato dei servizi di base che fornisce una serie di servizi (le sue API), per un db relazionale, invece, il discorso è più complesso e, in termini molto generali, si può dire che occorre creare o acquisire uno strato di software che fornisca questi servizi.

Il modo in cui i sottosistemi dello strato applicativo utilizzano i servizi forniti dagli strati più bassi stabilisce in che misura l'applicazione si lega alle componenti tecnologiche che forniscono questi servizi ed influenza direttamente caratteristiche della applicazione quali, ad esempio, la sua portabilità e le sue performances.

Scomposizione della applicazione in sottosistemi

Una buona architettura si basa sulla
- suddivisione (partizione) della applicazione in una collezione di sottosistemi
- stratificazione (layering) delle componenti (moduli, classi e oggetti) di ciascun sottosistema su più livelli

Affrontiamo qui il primo dei due punti, rimandando il secondo ad un paragrafo successivo.
In termini generali, un sottosistema ben progettato deve essere specializzato nello svolgere un certo compito (o, in altri termini, essere fortemente coeso), fornire un insieme di servizi (interfaccia) ben definito ed essere quanto più possibile indipendente da altri sottosistemi (debolmente connesso), in modo che i cambiamenti che intercorrono abbiano impatto limitato.

Alcuni validi motivi per suddividere una applicazione in sottosistemi sono:
- abbattere la complessità dei problemi legati alla realizzazione (secondo il principio del *Divide et Impera*, per cui risulta più semplice affrontare poche complessità per volta piuttosto che tutte insieme)
- facilitare la suddivisione dei compiti tra le persone del gruppo di progetto o tra i gruppi di progetto
- predisporre la successione dei rilasci sulla base della realizzazione dei sottosistemi
- isolare scelte o decisioni non ancora definite
- differenziare i servizi applicativi con un forte grado di riusabilità

Linee guida per scomporre una applicazione in sottosistemi

La scomposizione di una applicazione in sottosistemi può avvenire in modalità bottom-up o top-down [4].
La scomposizione in modalità bottom-up avviene a partire dalle classi del sistema, precedentemente individuate, raggruppando in uno stesso sottosistema classi strettamente correlate dal punto di vista funzionale.

Più in particolare vengono normalmente poste nello stesso sottosistema le classi che:

- appartengono allo stesso dominio
- eseguono diverse operazioni l'una sull'altra
- sono legate strutturalmente da relazioni di associazione, generalizzazione o specializzazione, aggregazione
- dipendono da uno stessa classe interfaccia o sono coinvolte in uno stesso caso d'uso

L'approccio top-down, invece, prevede che prima vengano identificati i principali sottosistemi, di questi vengano delineati i servizi che dovranno fornire e solo successivamente, analizzando più in dettaglio questi sottosistemi, vengano scoperte e progettate le classi che li compongono. In questo caso la linea guida per identificare i sottosistemi sono i casi d'uso, secondo la relazione per cui un sottosistema implementa uno o, se sono semplici, più casi d'uso a livello utente.

I meccanismi generali ed i comportamenti comuni individuati in una applicazione sono anch'essi validi candidati per divenire nuovi sottosistemi.

Gli approcci top-down e bottom-up non sono mutuamente esclusivi e possono essere applicati nel modo ritenuto più conveniente.

Interfacce e disaccoppiamento tra i sottosistemi

Importanza del disaccoppiamento tra i sottosistemi

I sottosistemi non vivono in isolamento ma, per fornire i loro servizi, utilizzano i servizi resi disponibili da altri sottosistemi.
Renderli indipendenti e robusti è un obiettivo per raggiungere il quale occorre progettare accuratamente il modo in cui i sottosistemi colloquiano, disaccoppiandoli quanto possibile e necessario.

Trattiamo il problema dividendolo in due aspetti: il disaccoppiamento dello strato applicativo dai sottosistemi degli strati dei servizi ed il disaccoppiamento tra i sottosistemi propri dello strato applicativo.
Per quanto riguarda il primo aspetto occorre stabilire quanto si vuole rendere indipendente l'applicazione dagli strati su cui si basa per i servizi comuni.
Se non si richiede l'indipendenza si possono direttamente utilizzare i servizi resi disponibili dagli strati più bassi. Se, ad esempio, prendiamo una applicazione installata su una piattaforma con GUI Windows, ciò significa che dalla nostra applicazione vengono richiamate direttamente le funzioni fornite da questa GUI.
Questo è il modo più semplice ed efficace di utilizzare la GUI ma se un domani occorresse, per esempio, portare l'applicazione su una piattaforma con GUI Motif, bisognerebbe mettere pesantemente mano su di essa.

Se, invece, si vuole rendere il più possibile indipendente l'applicazione dagli strati su cui si appoggia, occorre disaccoppiarla utilizzando le tecniche descritte nel seguito.
Per quanto riguarda il disaccoppiamento tra i sottosistemi propri della applicazione, la loro buona individuazione e progettazione limita già l'accoppiamento tra essi. Ulteriori esigenze di disaccoppiamento spesso vengono soddisfatte utilizzando pattern di progettazione.

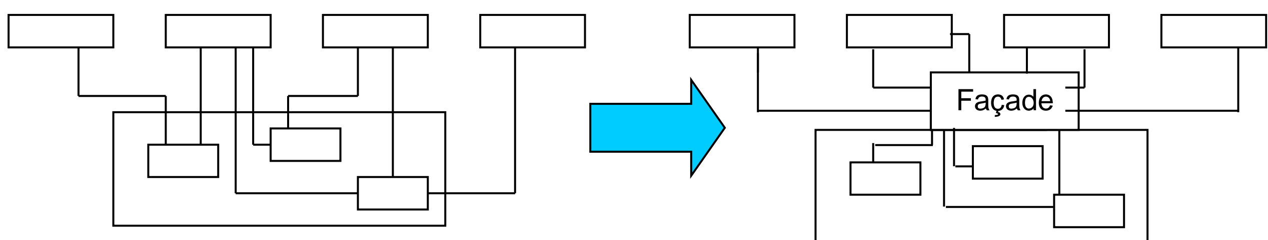
Interfaccia di un sottosistema

Per interfaccia di un sottosistema si intende l'insieme di servizi che questo rende disponibili verso l'esterno.
Poiché un sottosistema solitamente colloquia con diversi altri elementi del sistema (altri sottosistemi, oggetti, classi, …) ed è probabile che non si desideri dare a tutti questi elementi la stessa visibilità sui servizi, spesso vengono realizzate più interfacce per uno stesso sottosistema, ciascuna delle quali contiene solo i servizi che devono essere resi visibili agli elementi cui l'interfaccia è rivolta.

Pattern utili per interfacciare e disaccoppiare i sottosistemi

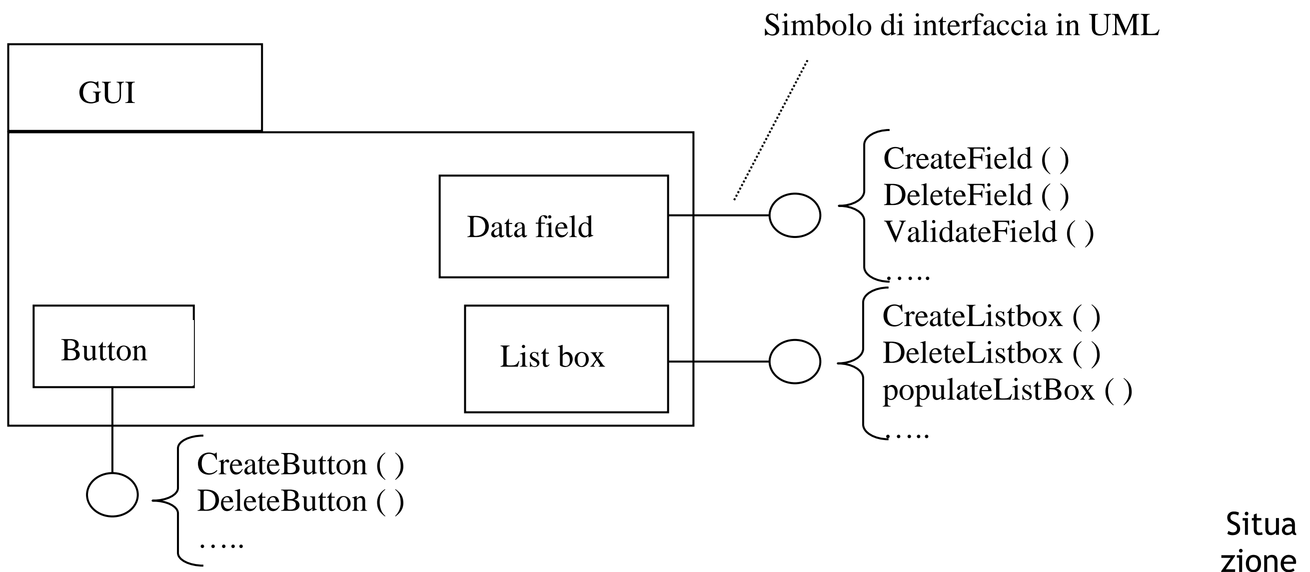
Esistono in letteratura molti pattern che risolvono problemi classici di progettazione.[2]
Di questi sono reperibili, anche in internet, diversi esempi pratici di utilizzo.
Per l'interfacciamento ed il disaccoppiamento dei sottosistemi, oltre ad utilizzare alcune notevoli proprietà dei linguaggi object-oriented (polimorfismo, incapsulazione, tipi), sono molto utili due pattern in particolare: "façade" e "publish-subscribe".

Il pattern "façade", in breve, prescrive di definire in un unico punto (un oggetto introdotto appositamente, chiamato façade) i servizi per accedere a un sottosistema.

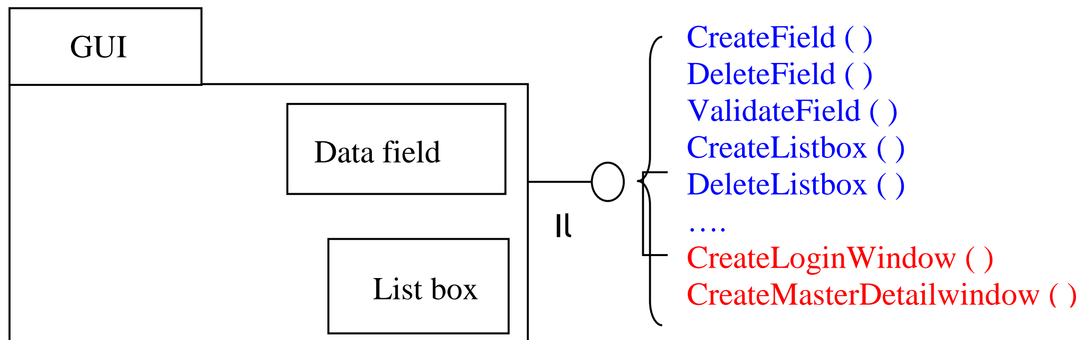


Così facendo gli elementi esterni non vedono più le singole componenti del sottosistema bensì la façade, col risultato che gli eventuali cambiamenti interni al sottosistema sono trasparenti all'esterno fintanto che non cambiano i servizi forniti dalla façade.
L'oggetto façade, per fornire i servizi che definisce, si appoggia ai servizi delle componenti del sottosistema.
Vediamo, come esempio, l'applicazione del pattern ad una GUI (Graphical User Interface).

Situazione iniziale, in cui le singole componenti del sottosistema esportano i loro servizi direttamente verso l'esterno.



Situazione successiva all'applicazione del pattern: il sottosistema GUI esporta una interfaccia che definisce una serie di servizi, alcuni dei quali sono gli stessi servizi "primari" forniti dalle singole componenti, mentre altri sono servizi diversi, tipicamente più potenti ed ottenuti basandosi sui servizi "primari".



Il pattern "publish-subscribe" (anche chiamato in altre fonti "observer" e "publisher-subscriber") risolve il problema di disaccoppiare l'oggetto che "pubblica" o genera un evento dagli oggetti interessati a questo evento.
Per far cio' viene introdotto un *Event Manager*, cioe' un oggetto che mantiene il mapping tra gli eventi e gli oggetti interessati a questi eventi.
Per pubblicare un evento, l'oggetto publisher invia un apposito messaggio, contenente tutti i dati dell'evento, all' *Event Manager*.
L' *Event Manager* notifica l'evento a tutti quegli oggetti che, precedentemente, si sono registrati presso di lui come interessati a quel tipo di evento (per la registrazione viene di solito utilizzato un meccanismo di *callback*), svincolando cosi' il publisher dal dover conoscere tutti i subscriber interessati ad un certo evento.
Questo pattern trova largo utilizzo in situazioni in cui, a fronte di un certo evento, due o piu' windows devono essere aggiornate contemporaneamente e, piu' in generale, quando in una applicazione si vuole disaccoppiare la business logic e la data logic dalla presentation logic.

Esistono molti altri pattern utili per interfacciare e disaccoppiare gli elementi di un sistema: rimandiamo a [1] e [2] per l'approfondimento.

Pattern architetturali per le applicazioni web

Mentre l'analisi si focalizza sui requisiti funzionali del sistema, ignorando i vincoli tecnici, il disegno architetturale deve fornire una soluzione tecnologica che tenga conto dei requisiti non funzionali del sistema (relativi all'usabilità, alle prestazioni, alla sicurezza, all'hardware,…) e costituisca l'input per la definizione dell'hardware necessario e la realizzazione del software.

Ovviamente il disegno architetturale è guidato in massima parte dalla tipologia dell'applicazione da realizzare.
Esistono oggi numerosi pattern architetturali, che forniscono un valido schema di riferimento per l'organizzazione strutturale delle diverse tipologie di sistemi software.
Qui ci occuperemo delle applicazioni web.

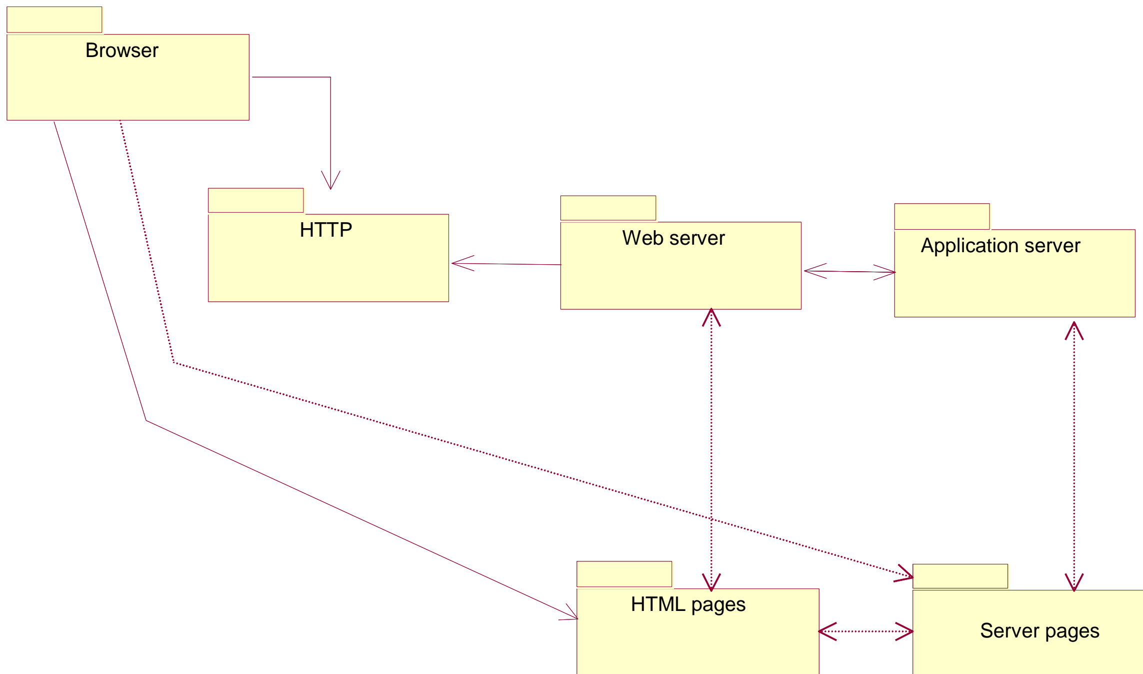
Per prima cosa, la definizione di un'applicazione web implica, come minimo, tre importanti componenti architetturali: il client browser, il web server e l'application server

Per essere più specifici, definiamo un'applicazione web come un sistema software client / server che ha, come minimo, i seguenti componenti architetturali: un HTML/XML browser su uno o più client che comunicano con un web server via HTTP e un'application server che gestisce la logica elaborativa.

I pattern più comuni per le applicazioni web sono tre:

- **Thin Web Client**, usato per lo più per le applicazioni basate su Internet, in cui il client è dotato esclusivamente di un browser HTML standard e tutta la logica di business è eseguita sul server; il protocollo di comunicazione client – server è HTTP
- **Thick Web Client**, dove una parte significativa della logica di business è eseguita sul client; a questo scopo il client usa generalmente HTML dinamico, applet Java o controlli ActiveX; la comunicazione con il server è sempre via HTTP
- **Web Delivery**, dove per la comunicazione client – server sono utilizzati altri protocolli oltre l'HTTP, ad es: IIOP e DCOM a supporto di un sistema a oggetti distribuito.

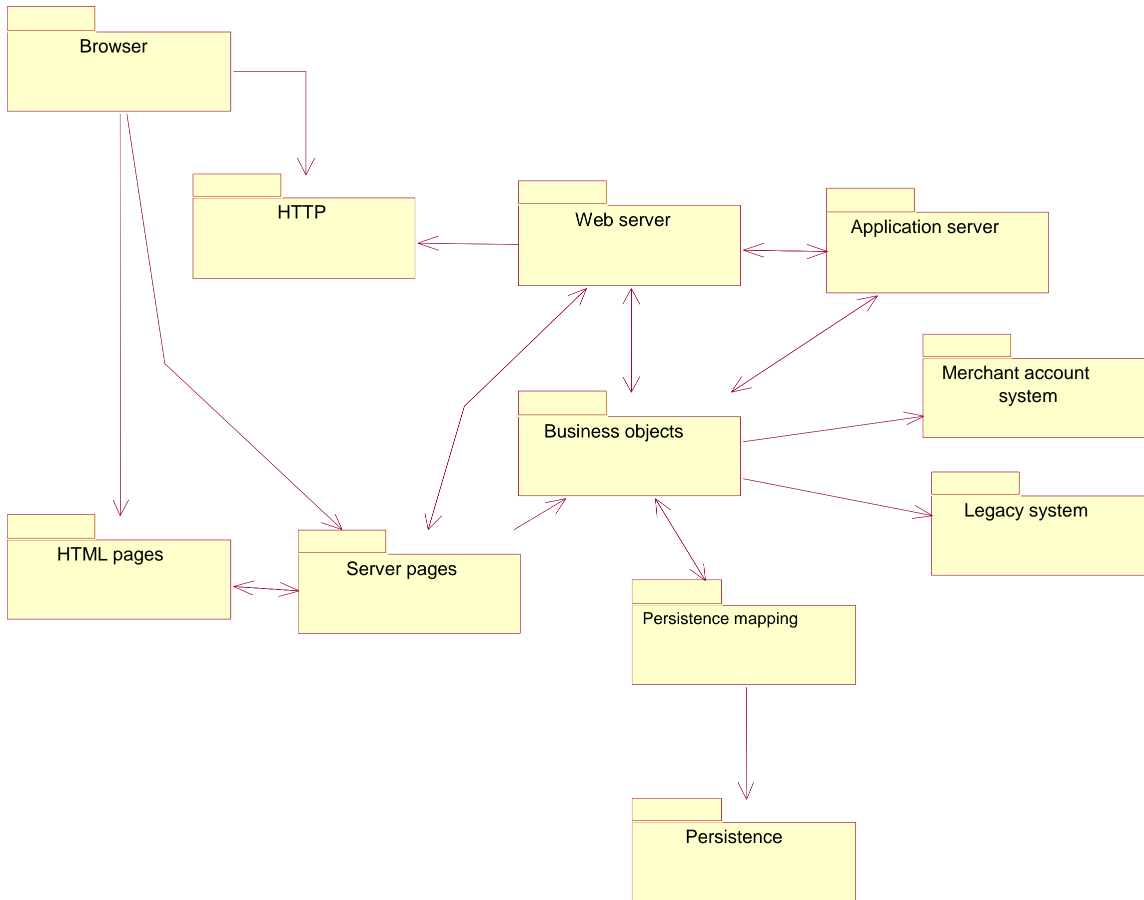
Ci soffermiamo di seguito su uno di questi pattern.
Il diagramma che segue illustra la vista logica dell'architettura prevista dal pattern ***Thin Web Client***.



Il pattern ***Thin Web Client*** illustra l'architettura di un'applicazione Internet in cui il client, dotato esclusivamente di un browser HTML standard, non esegue logica elaborativa delegando tale compito al server.

Le pagine web residenti sul server sono attivate a seguito di richieste provenienti dal client.
La pagina web costituisce l'elemento chiave per integrare il browser con le altre componenti del sistema.
Il client utilizza il protocollo HTTP per connettersi al server e richiedere una pagina web.
La pagina web può includere codice (script, componenti eseguibili, …) che deve essere eseguito prima di inviare il risultato al client.
In tal caso il web server chiama in causa l'application server. L'application server esegue il codice accedendo, se necessario, alle altre risorse lato server, quali ad esempio data base, sistemi legacy, altri servizi Internet, ecc…
Nel momento in cui la richiesta è stata evasa, il risultato prodotto viene inviato al client e la connessione s'intende terminata.

La vista logica fin qui descritta rappresenta uno schema architetturale ridotto al minimo con i soli componenti indispensabili, ma generalmente le applicazioni web presentano una serie di altri componenti importanti quali database, TP monitor, ecc…, così come si può osservare nella vista logica completata della figura seguente.



L'accesso al data base viene gestito mediante oggetti di business che incapsulano le regole di accesso ai dati. Tale scelta è auspicabile ancorché non tassativa. L'utilizzo di un RDBMS rappresenta oggi la scelta più comune per la gestione della persistenza.
Per questo motivo è stato previsto un ulteriore componente, denominato *Persistence Mapping*, che ha il compito di effettuare il mapping tra oggetti e strutture relazionali (materializzazione e de-materializzazione degli oggetti).

L'architettura proposta prevede inoltre l'integrazione del web sia con i sistemi legacy, sia, nel caso di applicazioni e-commerce, con un Merchant Account System che abilita le applicazioni web all'acquisizione di pagamenti mediante carta di credito.

Il disegno delle componenti interne delle applicazioni web (back end) segue le stesse regole adottate per il disegno di applicazioni mainframe o client/server. In particolare, le applicazioni web prevedono l'impiego di data base e TP monitor per le stesse ragioni indicate per gli altri tipi di applicazioni.

Organizzazione interna del sottosistema

Gli approcci tradizionali all'analisi ed al disegno Object-Oriented si concentrano sulla ricerca di un solo tipo di classi, tipicamente quelle che Ivar Jacobson (uno dei padri

dell'approccio Object-Oriented) identifica come classi Entità. Jacobson per primo ha introdotto la ricerca di tre tipi di classi: Interfaccia, Controllo ed Entità [3].
Le classi Interfaccia si occupano di gestire il colloquio tra il sistema e gli attori, garantendo la presentazione delle informazioni agli utenti e l'accettazione dei loro input al sistema.
Le classi Controllo contengono la logica necessaria per coordinare un servizio fornito dal sistema.
Le classi Entità contengono le informazioni tipiche del dominio del sistema che si sta realizzando e le operazioni strettamente legate a queste informazioni. Definiscono le regole di accesso ai dati e, normalmente, non prendono iniziative ma vengono chiamate in causa. Sono le prime che vengono individuate in fase di analisi.

Alcuni vantaggi che si ottengono da questo approccio sono:
una migliore distribuibilità della applicazione. Se, ad esempio, ci poniamo in ottica client/server a due livelli (un server con alcuni client collegati), tipicamente le classi Interfaccia sono destinate al client e quelle Entità al server, mentre le classi di Controllo vengono dislocate dove più opportuno.
la predisposizione per utilizzare la tecnologia più opportuna per realizzare le classi (ad es. 4GL per classi Interfaccia, PL/SQL per classi Entità, C per classi Controllo)
la localizzazione del cambiamento: classi di tipo diverso tendono a variare in modo diverso durante lo sviluppo (ad es. le classi Interfaccia tendono a variare più delle altre)
l'utilizzo di skill professionali specializzati per lo sviluppo delle varie tipologie di classi (ad es. uno specialista in linguaggi 4GL di solito non lo è anche in C o in Cobol e viceversa) .

Linee guida per la ricerca delle classi interfaccia, controllo ed entità

La ricerca delle tre tipologie di classi può essere condotta a livello di intera applicazione o di sottosistema, a seconda dell'approccio scelto per l'individuazione dei sottosistemi. Indipendentemente dal livello a cui viene effettuata, la ricerca avviene secondo le linee guida di seguito descritte.

Classi Interfaccia

Vengono definite in base alle caratteristiche degli attori, alla descrizione dei casi d'uso e dei requisiti.
In prima ipotesi si può definire, per ogni caso d'uso, una diversa classe interfaccia per ogni attore, tuttavia non è raro che un attore necessiti di più interfacce per interagire con il sistema o che un attore utilizzi una stessa interfaccia per eseguire più di un caso d'uso.

Classi Controllo

In prima ipotesi si può assegnare una classe di controllo per ogni caso d'uso.
Questa classe può essere

- una classe artificiale introdotta apposta per controllare il caso d'uso
- una classe che rappresenta il sistema intero
- una classe che rappresenta un'entità (oggetto o persona) che nella vita reale controlla le operazioni che si stanno considerando

Se il caso d'uso è molto semplice può essere superfluo introdurre una classe apposita e tutta la logica necessaria viene posta nelle classi entità.

Classi Entità
Vengono individuate in fase di analisi, *a partire dalla descrizione del sistema, dei requisiti e dei casi d'uso*, cercando di identificare le entità rilevanti nel dominio del sistema.

Mapping ad una piattaforma client-server a tre livelli
La distinzione delle classi del disegno in *interfaccia, controllo, entita'* prelude ad una stratificazione logica a tre livelli (layer):

Presentation layer	Interfaccia
Domain Model layer	Controllo
Persistent storage	Entità

La tipica allocazione degli strati logici su di una architettura fisica client/server a tre livelli (situazione tipica degli sviluppi attuali) prevede:
- Presentation layer sul client
- Domain Model layer sull'application server
- Persistent storage sul data server

Architettura delle classi: attribuzione delle responsabilità e definizione delle collaborazioni

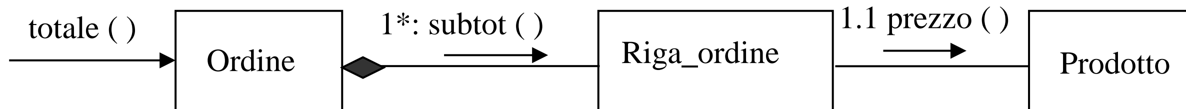
Concetto di responsabilità
La responsabilità complessiva di una classe è l'insieme di informazioni (attributi) e compiti (operazioni) che vengono assegnati ad una classe.
Assegnare correttamente le responsabilità ad una classe è importante perché così facendo si favorisce la realizzazione di una buona architettura interna del sistema.
Molte caratteristiche desiderate in una applicazione, quali ad esempio l'efficacia e la manutenibilità, vengono migliorate da una buona assegnazione delle responsabilità in quanto non si costringono le classi a svolgere compiti male assegnatigli, per i quali necessitano di recuperare informazioni e richiedere collaborazioni ad altre classi (magari poste su un'altra piattaforma in un ambiente distribuito), con la conseguenza di appesantire l'elaborazione, peggiorare la coesione ed aumentare il coupling.
Il problema quindi non è tanto il raggiungimento di un risultato funzionale, che può essere conseguito anche con una cattiva distribuzione di responsabilità, ma nella corretta attribuzione dei compiti alle classi per arrivare a questo risultato in modo ottimale.

Criteri per la assegnazione delle responsabilità

Esistono diversi principi (o linee guida) per assegnare correttamente le responsabilità: rimandiamo ad [1] per una trattazione completa di questi principi.
Qui citiamo solamente i due principi base per la assegnazione.

Il primo principio (che in [1] viene riportato come pattern “Expert”) è quello, intuitivo, di assegnare un compito alla classe che ha, possibilmente tutte, le informazioni necessarie per assolverlo e, quindi, non dovrà andarsele a procurare chiedendole ad altre classi.

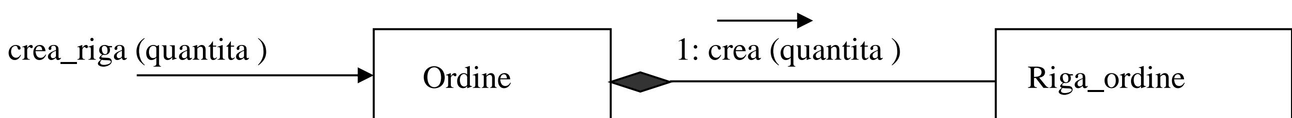
Esempio:
nell’esempio qui sotto riportato, la responsabilita’ di calcolare il totale di un ordine la assegno all’Ordine stesso, in quanto e’ lui che conosce tutte le righe di cui e’ composto. L’Ordine potra’ demandare ad ogni Riga_ordine il calcolo del subtotale.



Il secondo principio (che in [1] viene riportato come pattern “Creator”) è quello che riguarda la assegnazione della responsabilita’ della creazione degli oggetti: la creazione di una istanza di una classe A viene normalmente assegnata ad una classe B che ha almeno una delle seguenti relazioni con A (tali relazioni sono in ordine decrescente di importanza):

- B aggrega A
- B contiene A
- B registra A
- B usa A
- B possiede i dati necessari per inizializzare A

Sempre riferendosi all’esempio dell’Ordine, la responsabilita’ di creare nuove istanze di Riga_ordine andra’ assegnata ancora alla classe Ordine perche’ questa aggrega le righe.



Diagrammi di interazione

Per esprimere la collaborazione tra le classi e le relative responsabilità nell’ambito di un caso d’uso, da un punto di vista dinamico, si utilizzano i diagrammi di interazione.
Normalmente il diagramma di interazione descrive il comportamento di un singolo caso d’uso (un singolo scenario o più scenari).

Dei due diagrammi di interazione, diagramma di sequenza e diagramma di collaborazione, prendiamo qui in esame solo il primo (d'altra parte i due diagrammi sono isomorfi, si basano sulla stessa semantica, ossia due viste diverse sulle stesse informazioni; inoltre la maggior parte dei CASE basati su Uml consentono di trasformare in modo automatico un diagramma nell'altro).

Già in analisi si modellano i diagrammi di sequenza per allocare il comportamento del caso d'uso su più classi di oggetti, distinguendone la tipologia: classe interfaccia, controllo o entità.

Successivamente, nel corso del disegno, i diagrammi di sequenza elaborati durante l'analisi saranno estesi e affinati considerando le caratteristiche dell'architettura dell'applicazione e delle tecnologie adottate per lo sviluppo; generalmente ciò comporterà l'introduzione di nuove classi di oggetti e un'attribuzione più precisa di responsabilità e collaborazioni.in vista di obiettivi quali portabilità, manutenibilità, scalabilità, ecc…

Per seguire un esempio di tale evoluzione, riprendiamo qui il caso d'uso "Acquistare articoli" dell'applicazione "Acquisti online", basata su Internet, e il relativo diagramma di sequenza prodotto in analisi.

Descrizione del flusso di eventi del caso d'uso.

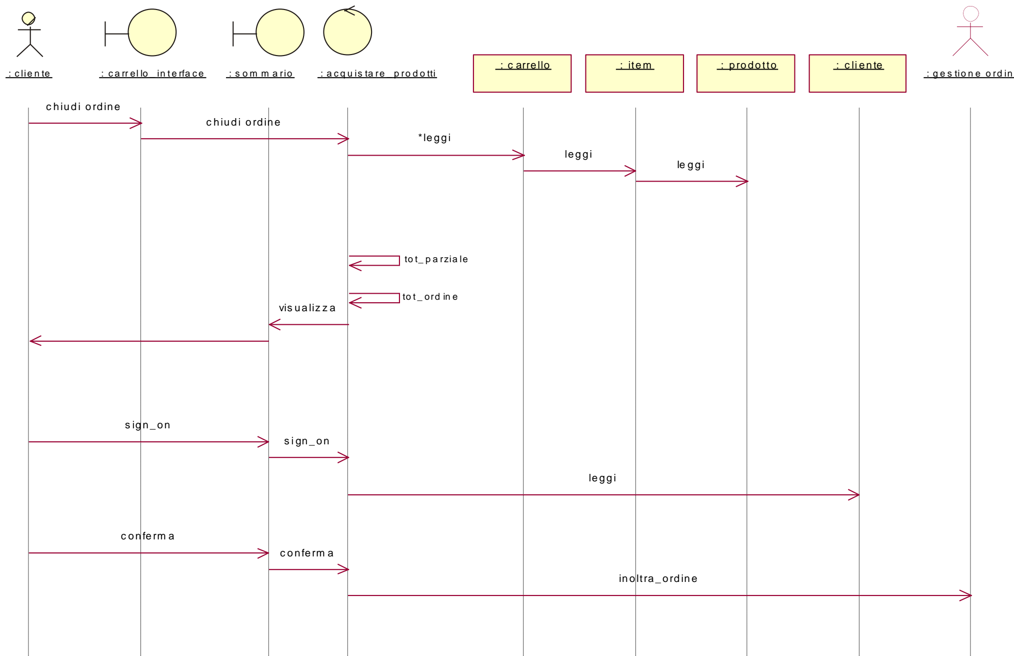
Principale:

1. Il caso d'uso inizia quando il cliente decide di acquistare i prodotti inclusi nel proprio "carrello della spesa" segnalando al sistema che l'ordine è completo
2. il sistema visualizza la lista dei prodotti da acquistare con l'indicazione del costo (sommario dell'ordine)
3. il cliente comunica al sistema i propri dati identificativi
4. il sistema visualizza i dati del cliente, evidenziando gli estremi della carta di credito su cui sarà addebitato il pagamento
5. il cliente conferma l'operazione e inoltra l'ordine di acquisto

Alternativi:

3.a se il cliente non è censito, deve registrare i propri dati per procedere all'acquisto

Diagramma di sequenza (analisi)

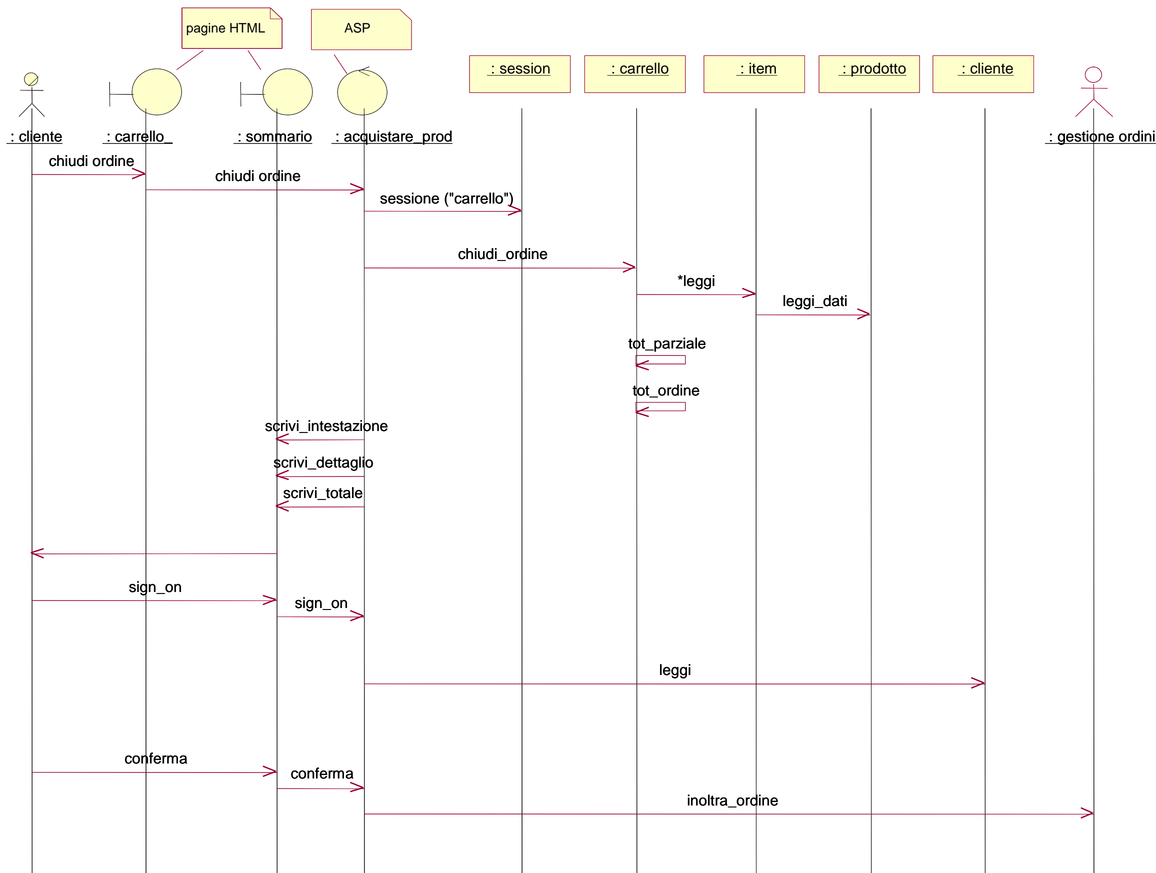


La revisione dei diagrammi di interazione ereditati dall'analisi prevede innanzitutto la separazione e la definizione delle interfacce utente e, più precisamente, delle pagine web.

Le pagine web rappresentano l'elemento chiave per connettere il browser con le altre componenti del sistema. Esistono diversi tipi di pagine web: pagine HTML, pagine web che contengono codice che viene eseguito sulla piattaforma server(script,…) e pagine web che contengono codice eseguito sulla piattaforma client (applet, …).

Tali aspetti possono essere documentati nei diagrammi UML introducendo nuovi tipi (o stereotipi) di oggetti.

Diagramma di sequenza (disegno)



Nel nostro diagramma, viene riportata l'indicazione che gli oggetti interfaccia daranno luogo a pagine web client, mentre l'oggetto di controllo corrisponderà a pagine web server (nel nostro esempio pagine ASP).

Compare inoltre un nuovo oggetto, Session, non presente nel diagramma prodotto in analisi.
La tecnologia ASP fornisce alcuni oggetti standard, particolarmente utili per l'implementazione di applicazioni web, quali ad esempio l'oggetto Session presente nel diagramma di disegno.
L'oggetto serve a mantenere lo stato di una sessione di lavoro utente, articolata in un insieme di richieste client a cui il server deve rispondere in modo coerente.
Il riferimento (Object Identifier) dell'oggetto Session utilizzato nell'ambito di una sessione di lavoro utente è mantenuto automaticamente dal sistema mediante l'utilizzo di cookie.

La pagina server Acquistare_prod acquisisce l'OID dell'oggetto Carrello associato alla sessione di lavoro specifica e gli invia il messaggio Chiudi_ordine. La maggior parte della logica elaborativa legata alla chiusura dell'ordine è incapsulata nell'oggetto di business Carrello.

La pagina server Acquistare_prod ha il compito di coordinare le attività nell'ambito della sessione di lavoro e di predisporre il riepilogo dei dati dell'ordine, un Sommario che è inviato al cliente in attesa di una sua conferma all'operazione.
Le responsabilità della pagina Acquistare_prod potrebbero ridursi ulteriormente introducendo un'altra pagina server, denominata ad esempio Predisporre_sommario, a cui delegare la predisposizione del sommario da visualizzare al cliente mediante il browser.

Bibliografia

[1] Craig Larman, "*Applying UML and Patterns: an introduction to Object-Oriented Analysis and Design*", Prentice-Hall, 1997

[2] Erich Gamma et al., "*Design Patterns: Elements of Reusable Object-Oriented Software*", Addison-Wesley, 1995

[3] Jacobson et al.,"*Object-Oriented Software Engineering – A Use Case Driven Approach*", Addison-Wesley, 1992

[4] Jacobson, Booch, Rumbaugh,"*The Unified Software Development Process*", Addison-Wesley, 199

Tecnet Dati s.r.l.
C.so Svizzera 185 –
10149 - Torino (TO), Italia
Tel.: +39 011 7718090 Fax.: +39 011 7718092
P.I. 05793500017 C.F. 09205650154
www.tecnetdati.com