Architectural Design and Software Evolution

Matteo Cadoni

Dipartimento Tecnologie Innovative, SUPSI

DTI.M-I4040.22:Ingegneria del *software*

Docente: Giancarlo Corti

12-06-2022

Abstract

In questo articolo vengono tratti i temi della *progettazione* di *architetture* e dell'*evoluzione software*, due temi che ritengo molto legati tra di loro. Il contesto in cui si muove il report è l'ingegneria del *software*, materia che si occupa di tutti gli aspetti legati alla produzione *software*. Il problema affrontato è il fatto di dover progettare *software* che sia facile da modificare e che impieghi al meglio le *risorse*. La *progettazione* di *architetture* è un processo che si occupa della *progettazione* di una *struttura* organizzativa del *sistema*. L'input di questo processo sono i requisiti e l'output è la *struttura* architettonica di come interagiscono i componenti. La rappresentazione è spesso informale poiché deve essere semplice comunicare con gli stakeholders. Ci sono diversi pattern architettonici come il modello a strati o client-server. L'*evoluzione* del *software* si occupa della manutenzione, miglioramento e aggiunta di nuove funzioni nel prodotto. Il raport tratta principalmente come le *architetture* favoriscano l'*evoluzione* e la manutenibilità del *sistema*, oltre che portare vantaggi nell'impiego di *risorse* e budget.

keywords: *progettazione,software, architetture, evoluzione,sistema, struttura, risorse*

Architectural Design and Software Evolution

Questo articolo tratterà brevemente del processo di *progettazione* delle *architetture software* e del legame che hanno con l'*evoluzione software*. Inizialmente esporrò le motivazione per cui mi hanno portato a scrivere questo lavoro e il contesto dei temi trattati, l'ingegneria del *software*. Successivamente entrerò più nel dettaglio sulla *progettazione* delle *architetture*, come vengono rappresentati e parlerò dei pattern architetturali principali. Successivamente parlerò dell'*evoluzione* del *software* e delle sue caratteristiche principali. Si prosegue con il confronto tra i due temi principali nel quale esporrò i perché li ritengo molto connessi come argomenti. Termina con le conclusioni con alcune considerazioni personali.

Motivazione

Questo paper tratta di "*Progettazione* di *Architetture* ed *Evoluzione Software*" poiché penso che siano due argomenti molto legati e ho potuto provare e usufruire dei vantaggi che le *architetture* offrono durante lo sviluppo. Ulteriore motivo di questo articolo è che è parte di valutazione del corso di ingegneria del *software*. Il contesto del report è l'ingegneria del *software*: disciplina dell'ingegneria che si occupa di tutti gli aspetti della produzione *software*, dalle prime fasi di sviluppo fino alla manutenzione del *sistema* dopo la distribuzione (Sommerville, 2016, p. 20). Quando parliamo di architectural design si parla di quella parte di *progettazione* che si occupa della comprensione dell'organizzazione del *sistema software* e della sua *progettazione* nella sua *struttura* complessiva. (Sommerville, 2016, p. 169)

Con *evoluzione software* si intende l'insieme delle operazioni di manutenzione, miglioramento e aggiunta di nuove funzionalità di un prodotto *software*.

Problema

Quando sviluppiamo un *sistema* o un *software* dobbiamo pensare che esso dovrà essere cambiato, aggiungendo funzionalità e integrare nuovi sistemi. Tutto questo deve essere fatto seguendo delle linee guida che ci permettono di costruire un *software* solido. Queste linee guida sono dette architetture e in questo articolo andremo a conoscere cosa sono alcuni tipologie, come vengono progettate e che rapporto hanno con l'*evoluzione*.

Stato dell'arte

Cos'è l'Architectural design

L'architectural design si occupa della comprensione dell'organizzazione del *sistema software* e della sua *progettazione* nella sua *struttura* complessiva. (Sommerville, 2016, p.169) Questa fase di *progettazione* si pone come passo successivo all'ingegneria dei requisiti, nella quale si raccolgono le informazioni necessarie per la realizzazione del *software*. L'output che si otterrà sarà modello architettonico del *sistema*, che rappresenterà come il *sistema* deve essere

organizzato e come ogni componente interagisce con gli altri. In questo tipo di *progettazione* il *sistema* incrementale non è usato poiché bisogna avere un idea generale e completa su come il *sistema* deve funzionare. Tuttavia i cambiamenti di un architettura, nonostante la facilità di modifica dei componenti, è molto costosa perché richiede la modifica di molti componenti in base ai cambiamenti. Quando parliamo di architectural design bisogna pensare al *sistema* in modo astratto e capire quali componenti fanno parte dell'architettura principale. Si può progettare un *software* tramite 2 livelli di astrazione chiamati: architecture in the small e architecture in the large. Il primo si occupa di *architetture* di un singoli programmi, composti da più componenti. Il secondo livello, invece, si occupa di *architetture* complesse per sistemi enterprise che contengono altri sistemi, programmi e componenti di quest'ultimi. Questi sistemi possono essere sistemi distribuiti, quindi ulteriormente complessi poiché necessitano anche della comunicazione in rete. Le *architetture software* sono importanti perché influenzano le prestazioni, la robustezza, la distribuibilità e la manutenibilità di un *sistema*. Secondo Len Bass, dal libro *software architecture in practice*, la documentazione e la *progettazione* di un'architettura *software* porta 3 vantaggi:

1. Comunicazione con gli stakeholders: l'architettura essendo ad un alto livello di presentazione è facilmente comprensibile e dunque si ha una migliore comunicazione tra i vari stakeholders
2. Analisi: la realizzazione di un'architettura, soprattutto se in una fase iniziale, necessita di analisi per individuare eventuali punti critici nei requisiti come prestazioni, affidabilità e manutenibilità.
3. Riutilizzo su vasta scala: il modello di un architettura essendo compatto, facilmente gestibile e riutilizzabile. Quest'ultimo punto consenti di utilizzare la stessa architettura che soddisfa gli stessi requisiti per applicazioni diverse.(Bass, Clements, and Kazman 2012).

Per rappresentare le *architetture* si utilizzano schemi a blocchi, questi sono molto efficaci dal punto di vista della comprensione di chiunque poiché rappresentano il *sistema* in modo semplice senza andare nel dettaglio. Un altro vantaggio di questo metodo di rappresentazione è il tempo impiegato per realizzarli, molto minore rispetto a una documentazione dettagliata, portando inoltre, un minor costo per il progetto. (Sommerville, 2016, p. 169-170)

Le decisioni nel design di architetture

Durante la *progettazione* vi sono numerose strutture decisionali che influenzano significativamente il *sistema* e il processo di sviluppo. La *progettazione* viene eseguita seguendo dei pattern, ossia una descrizione di un *sistema* organizzativo. La scelta di un architettura deve essere fatta basandosi sui requisiti. Successivamente bisogna decomporre la macro *struttura* in sotto strutture, ciò va fatto seguendo una strategia di decomposizione. In fine bisogna sviluppare

un modello generale di controllo delle relazione tra i vari componenti del *sistema*. La presenza di relazioni tra caratteristiche non funzionali e *architetture software* comporta la scelta di basare l'architettura su i seguenti requisiti:

- Performance: se le performance sono un requisito critico, allora l'architettura dovrà essere basata su componenti di piccole dimensioni per le operazioni critiche e risiedere sulla stessa macchina.
- Protezione: se la protezione delle *risorse* è un requisito critico, allora si dovrà adottare una *struttura* a livelli nei quali le *risorse* più critiche andranno a risiedere degli strati più interni, i quali dovranno avere un alto livello di sicurezza e validazione.
- Sicurezza: se la sicurezza è un requisito critico, allora bisognerà fare in modo che operazioni legate alla sicurezza siano co-localizzate in un singolo componente o in un numero ridotto di componenti.
- Disponibilità: se la disponibilità è un requisito critico, allora bisognerà progettare ed includere dei componenti di ridondanza.
- Manutenibilità: se la manutenibilità è un requisito critico, bisogna progettare i componenti in modo da poterli cambiare facilmente.

L'*evoluzione* di un architettura non è facilmente valutabile poiché bisogna valutare in che modo il *sistema* soddisfa i requisiti funzionali e non funzionali. (Sommerville, 2016, p. 171-173)

Rappresentare le architetture

la rappresentazione completa di un'architettura su un singolo diagramma è impossibile, poiché la rappresentazione grafica serve per avere una visione generale del *sistema*. (Sommerville, 2016, p. 173) Per documentare un architettura, dunque, si utilizzano 4 viste:

1. Vista logica: mostra le chiavi di astrazione del *sistema* come oggetti o classi di oggetti
2. Vista dei processi: mostra come il *sistema* compone i processi durante l'esecuzione
3. Vista di sviluppo: mostra come è decomposto il *software* per lo sviluppo
4. Vista fisica: mostra il *sistema* hardware e come sono distribuiti i componenti *software* attraverso i processori e il *sistema*.

La rappresentazione delle *architetture* a livello concettuale viene usata per spiegare la *struttura* del *sistema* agli stakeholders, e non è quasi mai necessario avere una rappresentazione completa dell'architettura. Lo strumento di rappresentazione è spesso oggetto di discussione, infatti l'uso di UML non è sempre fatto in modo preciso, ma in modo informale.

(Sommerville, 2016, p. 173-175)

Secondo Sommerville (2016) UML non è utile per il processo di *progettazione* e preferisce usare notazioni informali più semplici e rapide da comprendere (p.175). Su questo punto mi trovo d'accordo, poiché ritengo che rappresentare ciò che si vuole sviluppare debba essere comprensibile e veloce da spiegare. Tuttavia sono stati sviluppati linguaggi specifici per le *architetture*, ADLs, ma risultano spesso difficili da comprendere e utilizzare. In conclusione di questo capitolo, la documentazione dettagliata molto spesso non ha molta utilità e quindi può essere una perdita di tempo e budget la realizzazione di questi documenti. La rappresentazione dei sistemi deve essere in primo luogo uno strumento utile per la comunicazione con gli stakeholders e quindi non bisogna basarsi su una rappresentazione formale e completa. (Sommerville, 2016, p. 175)

Pattern architetturali

La *progettazione* di un architettura, come detto in precedenza, sfrutta dei pattern, ossia una serie di punti che descrivono la *struttura* organizzativa.I pattern vengono definiti in modo standard adottando una descrizione scritta mischiata a diagrammi. (Sommerville, 2016, p. 175)

Queste procedure possono essere viste come un insieme di good-practice che sono state testate in diversi sistemi e ambienti. Tra i pattern principali troviamo:

- Layered *architetture*
- Repository *architetture*
- Client-server *architetture*

Le *architetture* a livelli hanno come scopo quello di distribuire le responsabilità delle operazioni favorendo il cambiamento di un singolo livello senza dover modificare l'intera architettura.

Lo svantaggio di questo pattern è la suddivisione dei ruoli dei vari livelli, e in certi casi le performance possono essere problematica poiché ogni livello deve essere elaborato. (Sommerville, 2016, p. 176)

L'architettura basata su repository, invece, consente la gestione di dati in una repository centrale che permetta la condivisione dei dati da altri moduli. Questo tipo di architettura la ritroviamo degli IDE dove ogni tool recupera i dati dalla repository del progetto, ad esempio la cartella del progetto. Le repository *architetture* vengono utilizzate quando si deve generare una grosso volume di dati che devono essere immagazzinati a lungo termine. Lo svantaggio è che se la repository è un punto singolo, quindi in caso di errore (crash, blackout) tutto il *sistema* sarà bloccato, quindi bisogna anche predisporre un *sistema* di backup.

Le *architetture* client-server sono composte da due attori principali, i server che forniscono i servizi e i client che erogano quest'ultimi. Si usano nel caso in cui si abbiano dati condivisi come database; vi possono essere più server che possono variare in base al carico del *sistema*. Anche in

questo caso le problematiche di rete e backup dei sistemi vanno gestite per mantenere l'operabilità dei servizi. (Sommerville, 2016, p. 177-182)

Evoluzione software

I *software* sono prodotti che non vengono sviluppati una volta e successivamente non vengono aggiornati. Infatti si tratta di prodotti che evolvono con l'introduzione di nuove funzionalità e supporto per nuove piattaforme, per rimanere al passo con il mercato.

L'*evoluzione* di un *software*, specialmente in ambito enterprise, è un processo parecchio costoso poiché in quest'ultimo ambito si hanno prodotti che si appoggiano ad altri sistemi.

Durante l'*evoluzione* di un *sistema* i requisiti di installazione, il business e l'ambiente di lavoro cambiano e dunque i cambiamenti devono essere introdotti in aggiornamenti a intervalli regolari. L'ingegneria del *software*, dunque, è un processo a spirale tra requisiti, design, implementazione e testing. i hanno i 4 punti: Specification, Implementation, Operation e Validation che caratterizzano l'ingegneria del *software* che si ripetono a spirale per ogni release del *sistema*.

(Sommerville, 2016, p. 256)

L'*evoluzione software* di sistemi custom segue un processo diverso da quanto descritto in precedenza. Per questo tipo di sistemi il cliente paga un azienda per sviluppare il programma, ma successivamente si prende la responsabilità del supporto e delle scelte di *evoluzione*, ciò comporta una certa discontinuità negli aggiornamenti. Un alternativa al modello a spirale è la vista del ciclo di vita di una *sistema software*. Esso si compone di4 macro fasi che un *software* vive, dunque dallo sviluppo, seguito dall'*evoluzione* e il servicing, questa ultima fase avviene quando l'*evoluzione* non è più conveniente in termini di costi e dunque si fanno solo piccoli aggiornamenti per rendere il *sistema* funzionante, senza l'aggiunta di nuove funzionalità. Il processo di cambiamento successivo al rilascio del *software* è detto manutenzione *software*. Il ciclo di vita termina con il ritiro del *software*, c'è avviene quando non viene più utilizzato per motivi hardware, costi oppure per il trasferimento su un nuovo *sistema*.

(Sommerville, 2016, p. 256-258)

L'*evoluzione* dei *software* esiste dall'alba dell'informatica, infatti molte aziende hanno sostituito più volte i propri sistemi per rimanere dietro al business. Molti di questi sistemi esistono ancora oggi e sono detti sistemi legacy. Questi sistemi usano linguaggi e/o tecnologie dismesse per lo sviluppo di nuovi sistemi. Lo svantaggio di questi sistemi è che la manutenzione spesso è scadente. Questi sistemi non vengono cambiati per diverse ragioni come costi e rischi di varia natura. (Sommerville, 2016, p. 262)

Quando parliamo di *evoluzione* come detto in precedenza bisogna parlare anche di manutenzione. Infatti esistono diversi tipi di manutenzione[1]:

- Manutenzione correttiva

  consiste nella correzione di errori scoperti nelle release

- Manutenzione adattiva

  consiste nella modifica applicando cambiamenti dopo il rilascio

- Manutenzione di miglioramento

  consiste nel migliorare le prestazioni e la manutenibilità del sorgente

- Manutenzione di prevenzione

  consiste nella correzioni di errori prima che diventino problematici dopo il rilascio

Tra le altre pratiche di di gestione del codice citiamo la re-ingegnerizzazione, una pratica che consente di restaurare la *struttura* del *software* e la redazione di una nuova documentazione. Essa viene usata spesso per aggiornare i sistemi legacy nonostante la loro complessità nella comprensione e del cambiamento. Questo processo rende quest'ultimi sistemi più comprensibili e facili da mantenere. Tuttavia il *software* reengineering ha dei limiti su quanto si possa migliorare il prodotto. (Sommerville, 2016, p. 276)

Esiste un ulteriore pratica chiamata refactoring, un processo che consente di rallentare il degrado *struttura*le del *software* attraverso il cambiamento. Quest'ultimo processo migliora la *struttura*, rendendola più comprensibile, e non vengono aggiunte funzionalità rispetto al sorgente originario. Il vantaggio del refactoring è che utilizza metodi predisposti al cambiamento, e quindi rendendo più agile lo sviluppo futuro. Il *software* refactoring e il *software* reengineering possono sembrare la stessa operazione, ma la re-ingegnerizzazione è un processo che avviene dopo il rilascio e che ha dei costi di manutenzione in crescita. Mentre il refactoring è un processo continuo mirato al miglioramento del processo di sviluppo e di *evoluzione*.

(Sommerville, 2016, p. 278-279)

Esiste un ulteriore processo nella re-ingegnierizzazione del *software* detta rearchitecting. Questo processo è usato per documentare una *struttura* obsoleta di un legacy system complesso. L'obbiettivo è quello di recuperare la *struttura* in modo da poter evolvere il *sistema* in futuro. (Galal Galal-Edeen, Nissreen El-Saber, 2007)

1 *Software* evolution.(21, Novembre 2021).in *wikipedia.* https://en.wikipedia.org/wiki/*Software*_evolution

(consultato in data: 10-06-2022)

Confronto e valutazione

Abbiamo visto come le *architetture struttura*no i sistemi in modo che siano predisposti al cambiamento. Oggi con le tecniche di refactoring e re-ingegnerizzazione del codice si ha la possibilità di trasformare *software* vecchi con strutture vecchie e obsolete in sistemi più agili da mantenere e dunque con una miglior tendenza all'*evoluzione*. Un esempio di pattern architetturale che si presta all'*evoluzione* è l'architettura a livelli; infatti, se si vuole adottare una tecnologia differente o migliorarla in un determinato livello sarà necessario cambiare solo il livello di riferimento senza andare a cambiare tutto il *sistema*. Ciò porta anche dei vantaggi in termini di costi e *risorse* nella fase di *evoluzione* del *software*. L'*evoluzione software* e le *architetture* sono due elementi che si sostengono a vicenda, infatti le *architetture* consentono un'*evoluzione* efficiente e stabile, mentre l'*evoluzione*, e tutte le attività che la costituiscono come processo, come il *software* reengineering e il *software* rearchitecting consentono di riportare *software* con strutture corrotte, sistemi legacy, a una *struttura* moderna, documentata e che consenta un ulteriore *evoluzione* del *sistema*.

Conclusione

Le *architetture software* e l'*evoluzione* sono due ambiti molto legati. Lo sviluppo di *software* che sfruttano le *architetture*, come quella a strati, consentono di evolversi in modo agile, senza dover modificare altri componenti o riprogettare l'intero *sistema*. Questo consente di risparmiare *risorse* e budget, che oggi giorno si cerca di ottimizzare sempre di più. Anche il processo di manutenzione tramite un architettura viene semplificato ed ottimizzato portando parecchi vantaggi economici e tecnici. Nonostante la mia esperienza con lo sviluppo tramite pattern architetturali sia limita, in quel poco di esperienza ho toccato con mano alcuni vantaggi della separazione del *software* in piccoli blocchi e posso dire che ottimizza parecchio il processo di individuazione di bugs ed errori e accelera il processo di correzione. Uno sviluppo disordinato senza punti di riferimento come un'architettura rischia di essere una via problematica e confusionaria portando costi inutili durante la manutenzione e sviluppo del *sistema* .

Riferimenti

Sommerville, I.Pearson, ed., (2016), *Software Engineering, Global Edition*.

Galal, H., Galal-Edeen, Shouman, M., Nissreen, A.-G. and Elsaber (2007), Approaches to *software* Rearchitecting, *in* .