

Appunti di Progettazione del Software

Canale 2 · Prof. Massimo Mecella

Capitolo 1: Il software industriale

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

Argomenti: Attori · Ciclo di vita · Manutenzione e sistemi legacy · Qualità del software

Indice

1	Il software industriale	1
2	Gli attori	1
3	Il ciclo di vita del software	3
4	Manutenzione e sistemi legacy	5
5	Le qualità del software	7
6	Perché l'orientamento agli oggetti	10
7	Esercizi	11
8	Soluzioni	12

1 Il software industriale

Definizione – Software industriale

Il **software industriale** è un prodotto dell'industria: come un'automobile, va progettato e costruito perché funzioni nell'uso reale con prestazioni stabilite. Si contrappone al **toy software**, il programma piccolo scritto per uso personale o come esercizio.

Il passaggio dall'uno all'altro richiede metodi e tecnologie che alzano il livello di astrazione, in particolare quelle **orientate agli oggetti** (Sezione 6).

2 Gli attori

Definizione – Attori della progettazione del software

Gli **attori** sono le figure che prendono parte alla costruzione di un sistema software. Si dividono in tre gruppi:

- chi lo chiede e lo usa: **committente, esperto del dominio, utente finale**;
- chi lo costruisce: **analista, progettista, programmatore**;
- chi lo mantiene dopo la consegna: **manutentore**.

Attore	Ruolo	Esempio (banca)
Committente	mette i soldi e il rischio d'impresa; dà i requisiti di alto livello, cioè gli obiettivi strategici	la banca, nella persona dei suoi vertici
Esperto del dominio	conosce il <i>dominio</i> , il mondo in cui lavorerà il sistema; traduce gli obiettivi del committente in specifiche	il direttore di filiale, un esperto di contabilità
Utente finale	usa il sistema	gli impiegati allo sportello e i clienti che usano l'app
Analista	comprende il problema e lo rappresenta in modo formale con un modello concettuale	
Progettista	trova la soluzione: <i>come</i> realizzare ciò che l'analisi ha descritto	
Programmatore	traduce il progetto in codice	
Manutentore	dopo l'entrata in produzione corregge, adegua ed evolve il sistema	

L'esperto del dominio è la figura con cui chi sviluppa lavora di più: va intervistato di continuo, e a lui si spiega in parole semplici che cosa si sta facendo, perché sa dire se ha senso. Buona parte di ciò che sa non è scritta da nessuna parte e va fatta emergere con le domande giuste.

Osservazione – L'utente finale quasi mai è il committente

Il committente è spesso una grande organizzazione; gli utenti finali sono i suoi impiegati o i suoi clienti. Un sistema può quindi soddisfare gli obiettivi di chi l'ha pagato e non le necessità di chi lo usa ogni giorno.

Nel progetto convergono così tre esigenze: gli obiettivi strategici del committente, la conoscenza dell'esperto del dominio e la comodità d'uso per gli utenti. Il sistema di gestione delle carriere universitarie, per esempio, registra i voti e stampa i certificati anche dopo molti anni, e per il committente fa ciò che deve; per docenti e studenti l'interfaccia resta macchinosa.

Osservazione – Esperienza d'uso e budget

Progettare perché gli utenti trovino il sistema semplice e gradevole (la **UX**, *user experience*) richiede lavoro, e il lavoro costa. Il committente, che bada al budget, può non potersi permettere tutto quello che gli utenti vorrebbero.

Nota – Ruoli fusi

Fino alla fine degli anni Novanta analista, progettista e programmatore erano persone diverse. Dopo il 2000 i ruoli si sono fusi, perché i linguaggi sono di livello sempre più alto e programmare richiede meno tempo, mentre chi progetta deve sapere come la soluzione diventerà codice. Oggi si chiede un'unica figura, l'**analista programmatore** o **DevOps**, che copre dallo sviluppo (*development*) alla messa in esercizio (*operations*).

Un caso completo con tutti gli attori è nell'Esercizio 1.

3 Il ciclo di vita del software

Definizione – Ciclo di vita del software (SDLC)

Il **ciclo di vita del software** (*Software Development Life Cycle*, SDLC) è la successione delle fasi di un sistema, dall'idea del committente al giorno in cui viene spento:

1. **studio di fattibilità e raccolta dei requisiti;**
2. **analisi dei requisiti:** *che cosa* l'applicazione deve fare;
3. **progetto e realizzazione:** *come* lo fa;
4. **verifica;**
5. **manutenzione.**

Studio di fattibilità. Si valutano costi e benefici dell'idea del committente, si pianificano attività e risorse (tempi, numero di sviluppatori), si sceglie l'ambiente di programmazione, hardware e software, e si raccolgono i requisiti di alto livello. Un errore qui può far fallire l'azienda. Il “nuovo prodotto” non è sempre un sistema nuovo: per una banca l'app mobile costruita sul sistema esistente è un nuovo prodotto.

Raccolta dei requisiti. Si fa con interviste a esperti del dominio e a potenziali utenti, e con altre tecniche. Il risultato è un **testo in linguaggio naturale** che descrive che cosa l'applicazione deve fare; è la forma del testo da cui parte anche un esercizio d'esame.

Raccolta e analisi dei requisiti sono fra le fasi più difficili: il committente spesso non sa esprimere ciò che vuole, e descrive una cosa diversa da quella che ha in mente.

Analisi dei requisiti. Analizzare vuol dire comprendere a fondo il problema: di che cosa parla l'applicazione e che cosa deve fare. Il risultato è lo **schema concettuale**, che descrive gli oggetti informativi del dominio (banche, conti correnti) e le funzioni del sistema. Si scrive in **UML** (*Unified Modeling Language*), lo standard di fatto per questi schemi.

Lo schema è *concettuale* perché resta sui concetti e non scende nei dettagli: dice che una persona ha una data di nascita, non se la data si rappresenta con giorno, mese e anno.

Progetto e realizzazione. Si definiscono l'architettura del programma (quante classi, come rappresentare i concetti) e le **strutture di rappresentazione**, poi si scrivono il codice e la **documentazione**. Senza documentazione il lavoro prodotto perde valore, perché nessun altro può riprenderlo.

Esempio – Il millennium bug

Negli anni Settanta, per risparmiare memoria, molti sistemi rappresentavano le date con sei cifre, due per il giorno, due per il mese e due per l'anno. Con il passaggio dal 1999 al 2000 l'anno **00** diventava ambiguo: 1900 o 2000? Una banca, per esempio, avrebbe potuto calcolare in modo sbagliato gli interessi dei conti. La rappresentazione corretta usava otto cifre. Negli anni Novanta la correzione richiese uno dei più grandi interventi di manutenzione della storia del software.

Una scelta di rappresentazione presa in fase di progetto può quindi costare moltissimo decenni dopo.

Verifica. Si controlla che il programma svolga il suo compito:

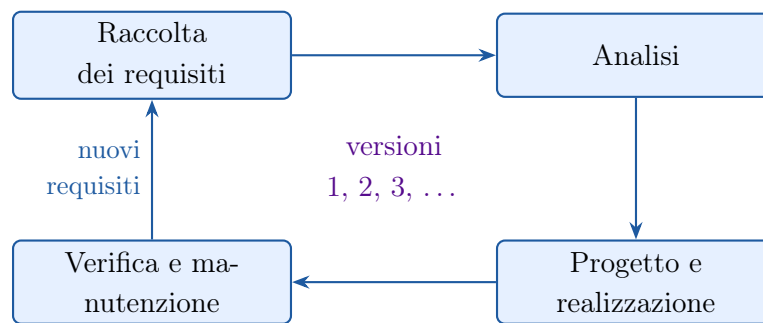
- **correttamente:** senza errori;
- **completamente:** tutto ciò che chiedono i requisiti è realizzato;
- **efficientemente:** in un tempo ragionevole.

Che un sistema sia corretto e completo non si dimostra come un teorema. Lo si prova e riprova finché la probabilità che contenga ancora errori, o che questi si manifestino spesso, diventa bassa. La verifica conta ancora di più quando il codice lo scrivono agenti software: la responsabilità del sistema resta di chi lo consegna.

Fase dello sviluppo		Tempo e risorse
Studio di fattibilità		circa 10%
Raccolta e analisi dei requisiti		circa 15%
Progetto e realizzazione		circa 25%
Verifica	almeno 50%, oggi fino al 60%	

Nei sistemi critici (impianti industriali, treni, centrali nucleari, semafori) la verifica può arrivare all'80% del tempo.

Le quattro fasi dello sviluppo occupano una parte piccola della vita del sistema; il resto è manutenzione (Sezione 4). Quando una manutenzione cambia i requisiti si torna alla loro analisi, e il ciclo ricomincia: per questo si parla di **modello a spirale**, in cui ogni giro produce una nuova versione.



Il modello a spirale: ogni giro del ciclo produce una nuova versione del programma.

4 Manutenzione e sistemi legacy

Definizione – Manutenzione

La **manutenzione** comprende gli interventi su un sistema dopo la sua entrata in produzione: correggerne gli errori, adeguarlo, migliorarlo. Su tutta la vita di un sistema lo sviluppo occupa al più il 10% del tempo e dei costi, la manutenzione circa il 90%.

La maggior parte di chi sviluppa software lavora quindi su sistemi esistenti, non su sistemi nuovi. Il sistema universitario di gestione delle carriere, sviluppato in un anno nel 1998, è stato mantenuto per quasi trent'anni.

Definizione – Tipi di manutenzione

- **Correttiva:** corregge un errore (*bug*) che si è manifestato.
- **Adeguativa:** adegua il sistema a un cambiamento esterno, legale o d'infrastruttura, senza cambiare ciò che fa.
- **Evolutiva:** cambia i requisiti ed estende le funzionalità.

Le prime due si indicano insieme con la sigla **MAC** (manutenzione adeguativa e correttiva), la terza con **MEV** (manutenzione evolutiva).

Tipo	Esempio
Correttiva	lo schermo di un aeroporto mostra la schermata di errore del sistema: il guasto va trovato e corretto
Adeguativa	il sistema scritto per Java 8 non funziona sulla versione successiva e va adattato; il sistema delle carriere va adeguato al passaggio dalle lauree quinquennali al 3+2; il millennium bug
Evolutiva	un sistema che funziona sul web deve funzionare anche come app mobile

La manutenzione evolutiva è la più frequente, perché i domini ancora senza informatica sono rarissimi: chi crede di costruire un sistema nuovo sta quasi sempre evolvendo uno esistente.

Definizione – Sistema legacy

Un **sistema legacy** è un sistema vecchio, spesso costruito decenni fa su *mainframe*, ancora in uso perché regge il cuore di un'attività. Nei sistemi di missione critica il nucleo che gestisce dati e transazioni è in genere quello costruito negli anni Settanta.

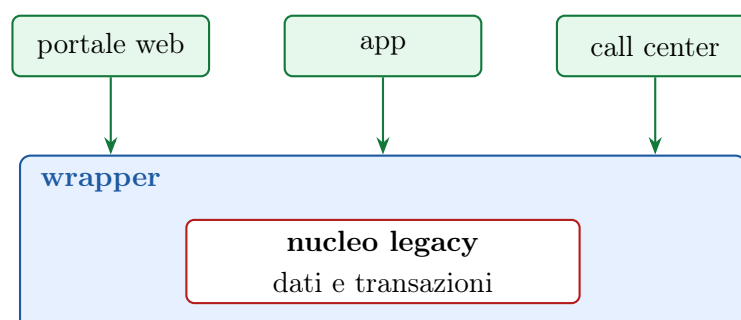
Una **transazione** è un'operazione che deve avvenire per intero o per niente: se pago il biglietto aereo, devo ottenere la prenotazione.

Osservazione – Perché i sistemi legacy non si riscrivono

Ogni errore scoperto in esercizio viene corretto. Un sistema usato e corretto per decenni ha quindi una probabilità molto bassa di contenere errori non ancora scoperti, o che questi si ripresentino. Riscriverlo per cambiare tecnologia vorrebbe dire perdere questa garanzia, e nessuna organizzazione corre il rischio.

Definizione – Wrapper

Un sistema legacy si **avvolge** (*wrap*) con strati di software moderno: il **wrapper** offre interfacce nuove (portale web, app, call center) e traduce ogni richiesta in una chiamata al vecchio nucleo, che resta invariato.



Un sistema legacy avvolto da un wrapper: i canali nuovi passano tutti per lo stesso nucleo.

Esempio – I codici degli aeroporti

Il nucleo dei sistemi di prenotazione aerea risale alla fine degli anni Sessanta, su mainframe con pochi kilobyte di memoria. Ogni aeroporto vi è identificato da tre lettere dell'alfabeto inglese: bastano per

$$26^3 = 17\,576$$

destinazioni, e occupano poco spazio. Per questo si scrive ancora **FCO** per Roma Fiumicino. Quando in un portale si digita “Roma”, il sito propone Fiumicino o Ciampino: il wrapper conosce la corrispondenza fra nomi e codici, ma la ricerca sul nucleo vuole il codice.

La manutenzione è anche la ragione per cui si impara a progettare: un sistema modulare, con codice documentato e nomi chiari, si capisce e si modifica senza smontarlo, anche da chi arriva in azienda vent'anni dopo. Chi rilegge il proprio codice dopo una settimana è già un manutentore.

5 Le qualità del software

Definizione – Qualità esterne e interne

Le **qualità esterne** sono visibili agli utenti, anche non specialisti, senza ispezionare il codice. Le **qualità interne** riguardano gli sviluppatori: le valutano gli specialisti e richiedono di conoscere la struttura del programma. La distinzione non è sempre netta.

Qualità esterne.

Qualità	Il sistema. . .
Correttezza	fa ciò per cui è stato progettato, con risultati coerenti con il dominio; se somma $2+2$ e dà 5 non è corretto, e lo vede chiunque
Affidabilità	è disponibile per la maggior parte del tempo e non si blocca mentre lo si usa: se i vincoli sono rispettati, i risultati arrivano
Robustezza	mantiene un comportamento accettabile anche in situazioni non previste dai requisiti, come un input non valido
Sicurezza (<i>security</i>)	protegge l'accesso alle informazioni e non concede accessi non autorizzati
Innocuità (<i>safety</i>)	non entra in stati pericolosi e non causa danni fisici a chi lo usa
Usabilità	è facile e gradevole da usare
Interoperabilità	scambia dati e collabora con altri sistemi, anche di fornitori diversi

Esempio – Robustezza

Si inserisce il 29 febbraio di un anno non bisestile. Un programma non robusto va in crash; uno robusto avvisa che la data non è valida e continua a funzionare.

L'interoperabilità si ottiene con **formati standard**, stabiliti da enti di standardizzazione e accettati da tutti: un sistema che deve essere conforme a uno standard va costruito leggendo quello standard.

Qualità interne.

Qualità	Il software. . .
Estendibilità	si adatta facilmente a nuove specifiche e cresce in funzionalità
Riusabilità	ha parti che si possono riusare in altre applicazioni
Efficienza	usa poche risorse: tempo di CPU, memoria, disco
Strutturazione	riflette nella sua struttura il problema e la soluzione; i moduli e i componenti si riconoscono
Modularità	è diviso in parti ben specificate e coese, che interagiscono
Comprensibilità	si capisce anche da chi non l'ha scritto: è documentato, commentato, con gli algoritmi spiegati
Verificabilità	si può testare, per controllare che faccia ciò che deve
Manutenibilità	si modifica facilmente; deriva dalle qualità precedenti
Portabilità	gira su piattaforme (hardware, sistema operativo) diverse da quella per cui è stato scritto

Strutturazione e modularità servono soprattutto alla manutenzione: chi sa che cosa fa ogni modulo interviene su una parte senza smontare il resto, come si sostituisce la fotocamera di un telefono senza cambiare il telefono.

La portabilità si ottiene con linguaggi interpretati o eseguiti da una macchina virtuale: lo stesso programma Python o Java gira su processori Intel con Windows e su processori ARM con macOS.

Osservazione – Misurare una qualità

Una qualità non si misura direttamente: è un concetto a più dimensioni. Si scelgono alcune proprietà misurabili in modo oggettivo, ciascuna con un indicatore numerico; il loro insieme dà una misura quantitativa, ma sempre indiretta.

Esempio – Misurare l'usabilità

Un indicatore è il questionario di soddisfazione con scala **Likert**, per esempio da 1 (molto insoddisfatto) a 10 (molto soddisfatto). Un contratto può legare un bonus al fornitore a una soglia: almeno l'80% dei rispondenti dà un voto di almeno 6. Un secondo indicatore è il tempo di passaggio da una pagina all'altra, letto nei log; l'usabilità complessiva li combina.

Osservazione – Qualità in contrasto

Non tutte le qualità si possono massimizzare insieme: bisogna trovare un bilanciamento.

- **Usabilità e sicurezza:** l'autenticazione a due fattori rende un accesso più sicuro e più lento.
- **Efficienza e portabilità:** Python, interpretato, è portabile ma più lento del C; il C è efficiente, ma va compilato per ogni sistema operativo e hardware.

6 Perché l'orientamento agli oggetti

L'approccio **orientato agli oggetti** favorisce tutte le qualità interne del software, ed è per questo il più usato nello sviluppo industriale. Indirettamente aiuta anche le qualità esterne: ottenere più facilmente quelle interne lascia tempo e budget per curare le altre.

Le tendenze dello sviluppo spingono nella stessa direzione:

- informazioni da gestire sempre più complesse e progetti sempre più grandi;
- utenti sempre più eterogenei;
- più bisogno di manutenzione;
- costi e tempi di produzione da contenere, con più qualità del prodotto finito.

La risposta è un approccio **modulare**, in particolare quello a oggetti.

Riepilogo

- **Software industriale:** prodotto dell'industria, da progettare come un'automobile; il contrario del toy software.
- **Attori:** committente, esperto del dominio, utente finale; analista, progettista, programmatore; manutentore.
- **Committente e utenti:** quasi mai coincidono; la UX costa e va pesata sul budget.
- **SDLC:** fattibilità e requisiti, analisi (che cosa), progetto e realizzazione (come), verifica, manutenzione; a spirale.
- **Verifica:** correttezza, completezza, efficienza; almeno metà dello sviluppo.
- **Manutenzione:** circa il 90% della vita; correttiva e adeguativa (MAC), evolutiva (MEV).
- **Sistemi legacy:** nucleo vecchio e affidabile, avvolto da un wrapper.
- **Qualità:** esterne per gli utenti, interne per gli sviluppatori; si misurano con indicatori e alcune sono in contrasto.

7 Esercizi

Esercizio 1 – Gli attori di un sistema per le contravvenzioni

Il Comune di XYZ intende automatizzare la gestione delle informazioni relative alle contravvenzioni elevate sul suo territorio. Vuole dotare ogni vigile di un dispositivo palmare con cui comunicare al sistema informatico il veicolo multato, il luogo e la natura dell'infrazione; il sistema notificherà la contravvenzione al cittadino per posta ordinaria. Il Comune bandisce una gara per la realizzazione e la manutenzione del sistema, vinta dalla Ditta ABC. Quali sono gli attori coinvolti?

Soluzione a pagina 12 →

8 Soluzioni

Soluzione dell'Esercizio 1 – Gli attori di un sistema per le contravvenzioni

- Committente: il Comune di XYZ, che paga il sistema.
- Esperto del dominio: un funzionario del Comune, o un altro professionista designato, esperto del Codice della Strada.
- Utenti finali: i vigili, che usano il palmare.
- Analista, progettista, programmatore, manutentore: personale della Ditta ABC, che realizza il sistema e ne cura la manutenzione.

Il cittadino riceve la notifica per posta e non usa il sistema: non è un utente finale.

← Torna all'Esercizio 1, pagina 11