Progetto del Software

Appunti redatti

Iacopo Ruzzier

Ultimo aggiornamento: 31 marzo 2025

Indice

Introduzione

Il corso tratta metodologie di progetto del software, diversificate per le varie necessità e solitamente riconducibili a poche famiglie → strutturazione e automatizzazione del processo

vedremo

- documentazione (su sw di larga scala si lavora in molti) - come usare, modificare, testare, rilasciare i nostri artefatti sw; licensing
- requisiti, KPI, test-driven design
- tool di collaborazione
- design tools (UML, E/R d., ...)
- design pattern
- strutturazione codice (programmazione avanzata)
- alcuni use-cases

Parte I
Collaborative tools - Git & friends

1 VCS, git

vcs: sistema che traccia le modifiche fatte ad un progetto e permette di ritornare a stati precedenti

- permette lo sviluppo collaborativo (modifiche "firmate")
- obbliga a seguire flussi di sviluppo noti
- fornisce set di strumenti per automatizzare testing, integrazione, sviluppo (CI/CD)
- modo semplice per scrivere documentazione
- permette di concentrarsi sulla scrittura del codice "senza pensieri"

1.1 git

vcs più usato, basato su repo distribuiti (2005 torvalds, per supportare sviluppo kernel linux): a partire da repo remote contenenti la codebase ("origin") gli utenti clonano la repo in locale, la mantengono aggiornata tramite pull, la aggiornano tramite push

2 Workflow base

progetto come **sequenza di commit**: snapshot del codice in un dato istante, con commit identificati da hashcode, contenenti riferimento al commit precedente, e commento obbligatorio
→ i commit tracciano cambiamenti incrementali della codebase, con granularità a discrezione dei programmatori

- prima di un commit devo aggiungere i file nella staging area
- rinominare file significa elimino+riaggiungo

```
git clone <URL>
git log #log di tutti i commit fatti
git diff #per visualizzare le differenze tra due commit
git add <file>[<file>...] #staging
git commit # -m "msg"
git commit -a # bypassa staging, ma non considera i file appena aggiunti
              # non ancora tracciati
git commit --amend # ritorno al commit precedente se ho dimenticato
   qualcosa
git push [origin] [master]
git pull [origin] [master]
```

IMG schema git

pulling e auto-merging: la storia locale è aggiornata in base al timestamp del commit. in particolare viene modificata automaticamente includendo sia i cambiamenti locali che quelli remoti (merge in automatico)

2.1 Merge e conflitti

git lavora sulla singola riga → conflitto se viene modificata da più utenti diversi, con repo locale che resta in stato conflicting in caso di **merge conflicts** → vanno risolti localmente e va fatto il merge manuale (flag appropriati)

consigli:

- pull frequenti
- assicurarsi che il codice funzioni (testing automations)
- commit piccoli, suddivisi per area di lavoro (codice, makefile, script) → forza a mantenere lo spazio di lavoro pulito e strutturato

```
git checkout/reset # unstaging/deletion modifiche o commit locali
git revert # per ritornare ad un commit specifico
git cherry-pick # essendo i commit incrementali (tracciano la
# differenza rispetto al genitore), possiamo applicare la stessa
# logica ma rispetto ad un commit diverso
```

3 Struttura tipica di un progetto con versioning

tipicamente molto rigida

- branch principale contenente ultima versione rilasciata (e intera cronologia commit)
- branch multiple corrispondenti a sottoprogetti specifici
- libertà totale sulle branch, tipicam. regole aziendali (`develop`,`bugfix/`,`hotfix/`,`features/`,`.pb_<smth>`)
- push su main branch non permesso → fork di main o dev, e poi PR (gestita dal maintainer)
- regole di accesso e vari ruoli utente (a liv. repo e branch)

flow tipico:

1. dev clona una branch della repo remota aggiornata
2. dev inizia a lavorare, nuovi commit in lcoale, nel mentre commit nuovi anche in remoto
3. una volta pronto, il dev fa una pull da remoto, con la responsabilità di rendere consistenti i propri commit con la cronologia principale (implica retesting)
4. dopo il merge, si crea il "final commit", si pusha sul cloud e si fa una PR
5. (tipicamente) la richiesta viene accettata, e le modifiche sono applicate alla branch `Developer`
6. ultimo pull per rendere consistente la copia locale, e (tipicamente) eliminiamo l'altra branch

Parte II

The software design process

Parte III

Requisiti e specifiche

Parte IV

Documentazione - Notazioni e strumenti

Parte V

UML

Parte VI
System architecture and design

4 System design

Step in cui traduciamo le specifiche del cliente in specifiche tecnologiche per gli sviluppatori → output: **architettura del sistema**

- identifichiamo un insieme di moduli, ciascuno con una singola funzionalità specifica
- descriviamo i *contratti*, le interazioni tra di loro

4.1 Decomposizione in moduli

- decomponiamo prima in **sottosistemi**, che interagiscono ma **non dipendono** tra loro
- poi in moduli e sottomoduli, ciascuno con il loro servizio specifico
- poi in componenti (unità implementativa di base)

La scelta alla base della decomposizione sta ovviamente nella separazione dei servizi offerti, e nell'assegnamento del controllo (chi controlla cosa)

4.2 Moduli

- si raggruppano funzionalità in stretta relazione (es. CRUD relative agli account utente) → a livello di system design dobbiamo specificare chiaramente le **interfacce** verso altri moduli o l'esterno
- successivamente identifico le sotto-funzionalità

4.2.1 Relazioni tra moduli

Tipicamente i moduli espongono servizi usati da altri, sono composti da sottomoduli (per lavorare a livelli diversi di dettaglio) e possono dipendere da altri moduli (tipicamente seguendo dei diagrammi di sequenza per use-case specifici)

4.2.2 Strategia di partizionamento

Top down:

- parto dalle specifiche e dalla documentazione
- arrivo ai servizi, moduli, componenti eccetera

Bottom up:

- data-structure/functionality centric → quando ad es. parto da sistemi preesistenti

L'approccio reale è ovviamente ibrido

5 Pattern architetturali

5.1 Architettura client-server

Tipico di sistemi distribuiti, composto di

- 1+ server che offrono generici servizi
- + client che usano i servizi
- 1 network di comunicazione, che si presume sempre attivo

Comunicazione asimmetrica (requests e responses)
Pro:

- facile distribuire dati e responsabilità ?
- scalabilità in termini di client e di server

Contro:

- la scalabilità "costa" (vado a scalare semplicemente aumentando le risorse)
- bisogno di un naming service: i server devono essere conosciuti dai client!
- evidente situazione di dipendenza

5.2 Architettura multi-tier

immagine

5.3 Design del controllo

Controllo inteso come "chi fa cosa", ovvero chi possiede la logica che implementa i casi d'uso → centralizzato o decentralizzato, con chiare conseguenze sull'architettura del sistema

5.3.1 Controllo centralizzato (sincrono)

Un singolo sistema (es. server web) gestisce (serves) tutte le richieste

- dipende da altri sottosistemi
- tipico per frontend di web-app
- basato su comunicazioni sincrone (es. chiamate di funzione)

I pro e contro direttamente derivati sono **single point of access** e **single point of failure**

Master-Slave

Si presenta nei paradigmi multi-process e multi-thread, in ogni caso richiede comunicazione tra processi

5.3.2 Controllo event-based (asincrono)

Ciascun modulo o sottosistema lavora in maniera indipendente dagli altri, e si relaziona con l'esterno tramite comunicazioni asincrone
→ pro e contro: sistema distribuito, dunque più difficile da implementare ma con meno dipendenze tra moduli

Paradigma event-based: il message broker è il first-class citizen, ovvero il modulo al quale si interfacciano tutti gli altri e che gestisce tutte le comunicazioni → si sposa benissimo con un frontend che deve mandare richieste (e poi si mette in attesa) - se async non ha bisogno di tenere occupate le risorse del sistema

5.4 Specifiche per il singolo modulo

Una volta definito il modello architetturale, si decompongono le macroaree nei singoli moduli → comincio a decidere dettagli implementativi (es. dove si trovano le funzionalità, su che server...), seguendo alcuni principi base:

- loose-coupling: maggiore indipendenza possibile
- minima conoscenza inter-modulare tra sviluppatori
- high cohesion: i moduli raggruppano le funzionalità strettamente dipendenti

Per prima cosa dobbiamo definire i contratti (interfacce) tra i vari moduli (professionalmente si usa UML):

- che funzionalità espongo: es. update dell'età, rimozione utente...
- come le espongo: servizi o funzioni da chiamare, es. REST API
- parametri input-output: numero, tipo, eccetera

6 MVC - Model View Controller pattern