Chi trova un pattern trova un tesoro!

Estendere progetti legacy senza schiattarci ☠️

Marco Breveglieri

ROMA - 23 MAGGIO 2025

Marco Breveglieri

Software Developer 👨‍💻 | Trainer and Consultant 👨‍🏫 | Tech Content Creator 🎥 | Community lover 💜

Homepage
https://www.breveglieri.it

Blog tecnico
https://www.compilaquindiva.com

Delphi Podcast
https://www.delphipodcast.com

Canale Twitch
https://twitch.tv/compilaquindiva

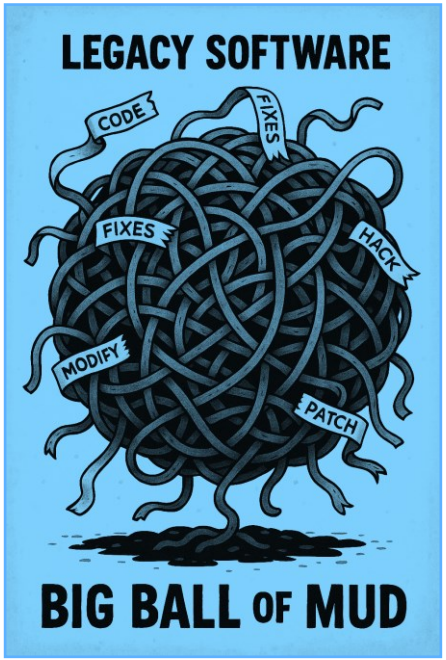
Tutti gli altri link:
👉 https://linktr.ee/marco.breveglieri


Iniziamo! 👌

Chi ha paura del Software Legacy?

lo, ad esempio. 😨

La famigerata «Big Ball of Mud»



- È un software diventato «caos totale» a causa della mancanza di disciplina architetturale
- È un problema comune nel software
- Può portare a un debito tecnico significativo
- Ha impatti (estremamente) negativi sull'intero processo di sviluppo

Come riconoscerla?

- **Architettura mancante**
  - Codice disorganizzato e difficile da capire
  - Principi architetturali indefiniti e «best practice» praticamente assenti
  - Confini poco chiari tra moduli e/o componenti non modulari
- **Alto accoppiamento**
  - Parti di codice molto interconnesse ad altre
  - Aree che ne impattano altre su modifiche
- **Manutenzione impegnativa**
  - Elevato tempo richiesto per fix e migliorie
- **Scalabilità limitata**
  - Impossibilità di scalare il sistema (senza revisioni significative)

Quindi...

...che si fa?

Si riscrive o si evolve?

Riscrivere da zero: i pro 👍

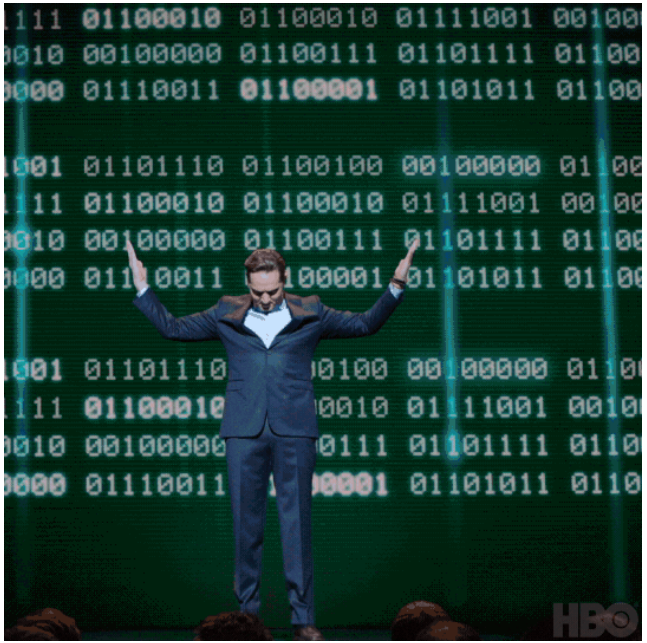
- **Libertà architetturale completa**
  - Tecnologie e paradigmi moderni
- **Rimozione del debito tecnico accumulato**
  - Nessun workaround ereditato
  - Si riparte da zero e «puliti»
- **Allineamento standard e compliance**
  - Più semplice rispettare le normative (es. GDPR, ISO, OWASP, ...)
- **Maggiore attrattività per il team**
  - I dev adorano scrivere codice nuovo
  - Strumenti e stack sono più «sexy»
- **Automazione sin dall'inizio**
  - CI/CD e Unit/Integration Test da subito

Riscrivere da zero: i contro 👎

- **Alti costi e tempi lunghi di realizzazione**
  - Si deve replicare tutto l'esistente
  - Non tutto è ben documentato
- **Regressioni funzionali molto probabili**
  - Casi limite non coperti dal nuovo
  - Logiche potrebbero andare perse
- **Distrazione del focus dal team**
  - Mantenere il vecchio sistema mentre si sviluppa il nuovo è frustrante
- **Mancanza di ROI immediato**
  - Valore per l'utente finale posticipato: si manifesta più avanti, spesso solo a progetto finito... se viene finito. 😁

Riscrivere da zero: quando farlo

- Il software è troppo instabile
- Il software è impossibile da manutenere
- Le tecnologie sono obsolete
- Il linguaggio e/o il compilatore non sono più supportati
- Il sistema non riflette più la realtà



Evolvere il legacy: i pro 👍

- **Rischio contenuto e controllabile**
  - Versione funzionante già in produzione
  - Sistema usabile durante l'evoluzione
- **Time-To-Value più rapido**
  - Priorità ai miglioramenti più tangibili
- **Conservazione della conoscenza**
  - Valorizzazione del codice già scritto, magari frutto di anni di lavoro
- **Impatto minore sull'utenza**
  - Continuità operativa è facilitata
  - Imprescindibile se «mission-critical»
- **Budget più flessibile**
  - Investimenti distribuiti nel tempo

Evolvere il legacy: i contro 👎

- **Complessità alta e crescente**
  - Sistema ibrido difficile da governare
  - Possibili effetti collaterali anche per singole modifiche isolate
- **Limiti delle tecnologie vecchi**
  - Ostacolo all'adozione di nuovi tool e pattern di sviluppo
- **Rischio di infiniti «rattoppi»**
  - Pericolo di procrastinare ulteriormente un «rewrite» necessario
- **DevOps e testing difficili**
  - Mancanza di testing e automazioni
  - Difficoltà a introdurli in seguito

Evolvere il legacy: quando farlo

- Il software è centrale per il business (e funziona bene)
- Il team conosce bene il codice esistente
- Si possono isolare i componenti e le funzionalità (con priorità a quelle strategiche)
- Il sistema può accogliere nuovi moduli e componenti

Di nuovo, quindi...

...riscrivere il software oppure evolverlo?

La risposta definitiva è...

Dipende.

Non voglio problemi, ma soluzioni.

Parliamone assieme.

Delphi: è già un'ottima scelta! 👌

- **Alta «backward-compatibility»!**
  - Forse la feature più concreta e apprezzata dagli sviluppatori ✨
- **Codice leggibile ed elegante** ✨
- **Ricco di risorse e strumenti utili**
  - RTTI, late-binding e serializzazione
  - Package (dinamici e non), interface
  - DUnitX già integrato per i test
  - VFI (Virtual Form Inheritance) ✨
  - Supporto a OOP (e altri paradigmi)

Ma i famosi «pattern»?

- Branch by Abstraction
- Facade Pattern
- Strangler Fig
- Anti-Corruption Layer
- Feature Flag
- Sidecar Pattern
- Gateway/Adapter Pattern
- Outbox Pattern
- [redacted]
- [redacted]

Ops, questi li ho inventati! 😁

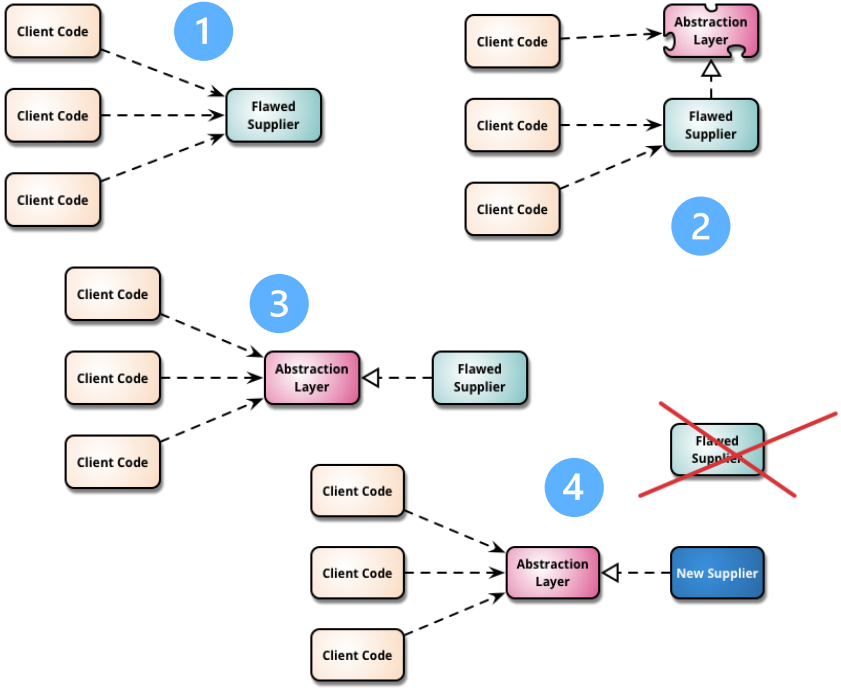

Demo time! 👍

Hai notato qualche pattern?

🧩 Branch by Abstraction

- Consente di lavorare parallelamente con meno interferenze
- Accelera lo sviluppo
- Mitiga i danni da dipendenze fallaci

👍 Vantaggi: efficiente, sicuro
👎 Svantaggi: ha un prezzo, codice a volte transitorio (vedi *«Transitional Architecture»*)



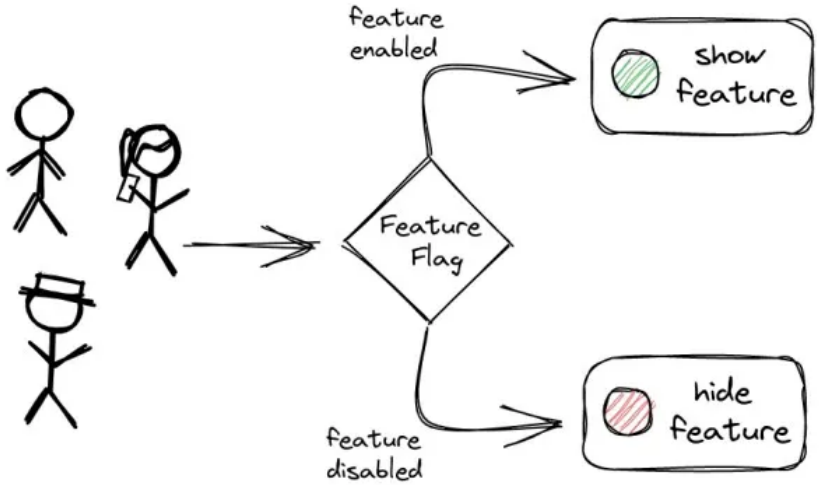
Fonte: https://martinfowler.com/bliki/BranchByAbstraction.html

🧩 Feature Flag

- Interruttori per controllare una determinata funzionalità
- Aiuta a testare nuove feature o implementazioni

👍 Vantaggi: roll-out controllati, test facilitato

👎 Svantaggi: complessità, possibili conflitti ed effetti collaterali



Fonte: https://medium.com/@suchibansal/system-design-question-build-a-feature-flag-library-81cf3886e9e

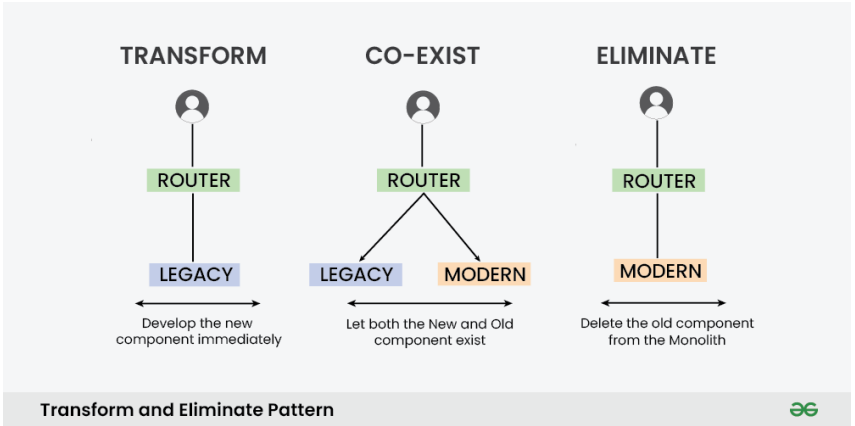
🧩 Strangler Fig

- Utile per migrare monoliti verso microservizi
- Sostituzione di parti di un'applicazione

👍 Vantaggi: gradualità, flessibilità, coesistenza

👎 Svantaggi: complesso, sfidante, potenzialmente impattante

Fonte: https://martinfowler.com/bliki/StranglerFigApplication.html



Fonte: https://www.geeksforgeeks.org/strangler-pattern-in-micro-services-system-design/

...e tantissimi altri! 

Prime conclusioni

🎯 Obiettivi e risultati

- Non serve riscrivere alcun codice per evolvere l'architettura
- L'organizzazione del codice rimane strutturata
- Il codice Delphi è sempre leggibile e mirato
- Abbiamo disaccoppiamento, «fallback», codice testabile

🔮 Ulteriori migliorie

- Feature Flag da DB o file INI
- Aggiunta di Integration Tests
- Timing delle performance
- Creazione di servizi accessori
  - Sidecar Pattern
  - Messaging e microservice
- Aggiunta di log, trace, metriche Prometheus*!

(*) ne parliamo nel pomeriggio! 😉

Ogni volta che...

- ...hai attivato una funzione solo per certi utenti tramite una colonna del database?
  Hai usato un Feature Flag Pattern
- ...hai scritto un codice «provvisorio» accanto a quello attuale, solo per testare una cosa senza rompere nulla?
  Stavi facendo Branch by Abstraction oppure la Strangler Fig
- ...hai creato un'applicazione esterna o un processo che ne arricchisce uno esistente?
  Quella è la forma base del Sidecar Pattern
- ...hai scritto dati in una tabella per consumarli da un altro processo?
  Benvenuto nell'Outbox Pattern!

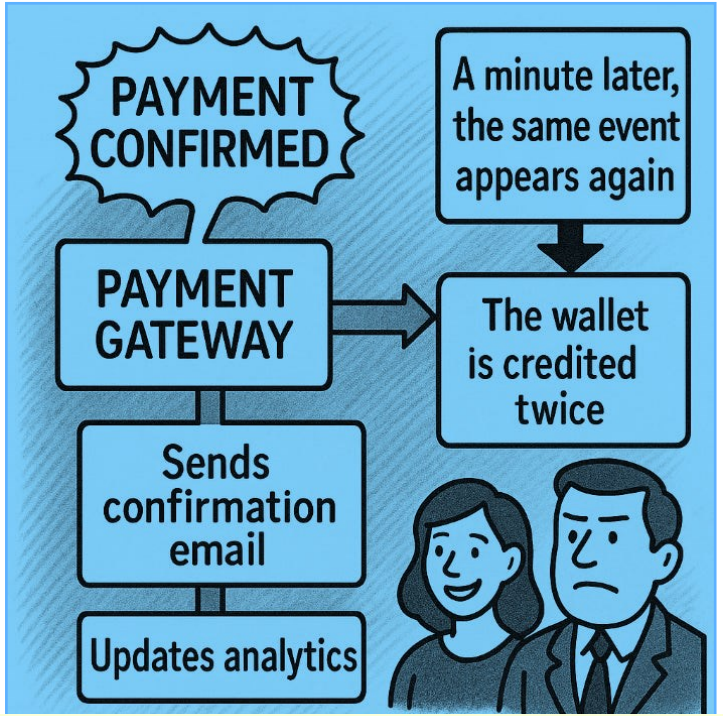
Un’esperienza reale!

Come un pattern può risolvere un problema e svoltare la giornata!
(in meglio, s’intende)

Scenario: 2 scritture, 1 problema!

- Supponiamo di avere
  - Salvataggio di un nuovo ordine nel DB
  - Invio dell'evento a un sistema esterno per notifiche o altre operazioni (es. SMTP, API, coda, ecc.)

- Cosa succede se
  - ...il salvataggio riesce ma l'invio fallisce? 😢
  - ...l'evento parte ma il DB fallisce subito dopo? 😭
  - ...e se provo a ritentare? 😵



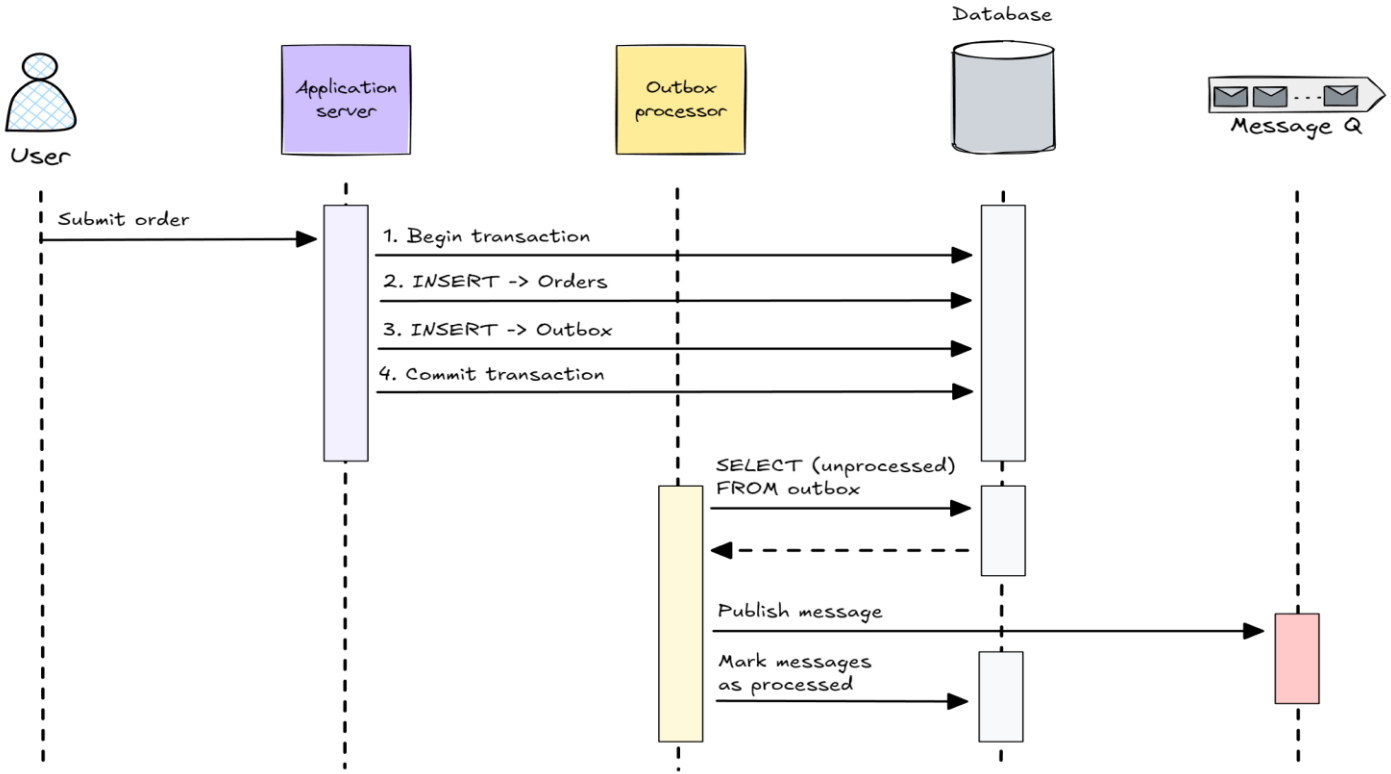
Fonte: https://newsletter.systemdesignclassroom.com/p/what-if-this-happens-twice

Altro scenario: evoluzione legacy!

La situazione che ci ritroviamo:

- Procedura che già fa «tante cose»
- Codice non proprio «a regola d'arte»
- Nuove dipendenze? Mmm... anche no!
- Tempistiche già abbastanza dilatate
- Sistema software «business critical»
- Cliente finale un po' suscettibile
- Implementazione passibile di ulteriori modifiche successive (l'analisi è ancora in corso)
- Ambiente di sviluppo e linguaggio non recentissimi (codice nuovo, nuovi tool!)

Outbox Pattern alla riscossa!



Fonte: https://www.milanjovanovic.tech/blog/implementing-the-outbox-pattern

Tabella di appoggio

```
CREATE TABLE outbox_messages (
  id INTEGER PRIMARY KEY,
  resource VARCHAR (255) NOT NULL
                  CONSTRAINT df_outbox_messages_resource DEFAULT (''),
  operation VARCHAR (255) NOT NULL
                  CONSTRAINT df_outbox_messages_operation DEFAULT (''),
  content JSONB NOT NULL
                  CONSTRAINT df_outbox_messages_content DEFAULT (''),
  occurred_on TIMESTAMP NOT NULL
                  CONSTRAINT df_outbox_messages_occurred_on DEFAULT (datetime()),
  processed_on TIMESTAMP,
  error TEXT NOT NULL
                  CONSTRAINT df_outbox_messages_error DEFAULT ('')
);
```

Esempio basato su database in formato SQLite 👆

Perché usarlo?

✅ **Affidabilità**
L'evento non viene perso

🔁 **Retry sicuri**
Fatti solo se necessario

🧪 **Debug e ispezione**
È tutto tracciato!

🚀 **Scalabilità**
Compatibile con architetture potenzialmente distribuite


Demo time! 👍

Conclusioni finali

Cosa portarsi a casa

Cosa portarsi a casa?

- Non sempre serve riscrivere tutto da zero
  - L'evoluzione è spesso più sostenibile del cosiddetto «big bang rewrite»
- Pattern architetturali mirati aiutano l'evoluzione
  - Strangler Fig, Outbox, Feature Flag, ecc.
  - Esplora il sito di Martin Fowler o cercali su Google per approfondirli ulteriormente
- Delphi ti offre tutti gli strumenti necessari
  - Interfacce, RTTI, gli elementi moderni del linguaggio
- Approccio incrementale = meno rischio, più controllo
  - Prediligi l'affiancamento del nuovo al vecchio, testalo a fondo e attivalo in produzione gradualmente.


Domande? ✋


Grazie! 🤗