Capitolo 1

Che cosa si intende per struttura e architettura?

In questo capitolo

- **L'obiettivo?**
- **Un caso di studio**
- **Conclusioni**

È stata fatta molta confusione sui termini struttura e architettura nel corso degli anni. Che cosa si intende per struttura? Che cosa si intende, invece, per architettura? Quali sono le differenze?
Uno degli obiettivi di questo libro è quello di far piazza pulita di tutta questa confusione e di definire, una volta per tutte, che cosa si intende per struttura e architettura. Per iniziare, vi dirò che non vi è alcuna differenza. *Proprio nessuna.*
La parola "architettura" viene spesso usata nel contesto di qualcosa che si trova a un livello superiore, distinto dai dettagli di basso livello, mentre "struttura" sembra implicare strutture e decisioni a un livello più basso. Ma questa distinzione non ha senso considerando quello che fa davvero un architetto.
Considerate l'architetto che ha progettato la mia casa. Questa casa ha un'architettura? Naturalmente sì. E che cos'è questa architettura? Be', è la forma della casa, il suo aspetto esteriore, l'elevazione e la disposizione degli spazi e delle stanze. Ma osservando i disegni prodotti dal mio architetto, noto un immensa quantità di dettagli di basso livello. Vedo la posizione delle prese di corrente, degli interruttori e delle luci. Vedo quali interruttori controllano quali luci. Vedo la posizione del camino e le dimensioni e la posizione dello scaldabagno e della pompa a immersione. Vedo una rappresentazione dettagliata di come verranno collocate le pareti, i soffitti e le fondamenta.
In breve, vedo tutti i piccoli dettagli che supportano tutte le decisioni di alto livello. Vedo anche che questi dettagli di basso livello e queste decisioni di alto livello fanno parte dell'intera struttura della mia casa.
E così è anche con la progettazione del software. I dettagli di basso livello e la struttura di alto livello fanno parte dello stesso insieme. Essi costituiscono un unico insieme che definisce la forma del sistema. Non potete avere gli uni senza l'altra; in effetti, non esiste una linea che li separi. Vi è semplicemente un *continuum* di decisioni dai livelli più elevati a quelli più bassi.

L'obiettivo?

E l'obiettivo di tali decisioni? L'obiettivo della buona progettazione del software? Tale obiettivo non è altro che la mia descrizione utopistica:

> L'obiettivo dell'architettura del software è quello di ridurre al minimo le risorse umane necessarie per realizzare e manutenere il sistema in questione.

La qualità della struttura si misura semplicemente valutando l'impegno necessario per corrispondere alle esigenze del cliente. Se tale impegno è contenuto e rimane contenuto per tutta la vita del sistema, la struttura è buona. Se l'impegno cresce a ogni nuova release, la struttura è cattiva. Tutto molto semplice.

Un caso di studio

Come esempio, considerate il seguente caso di studio. È costituito da veri dati di una vera azienda che preferisce rimanere anonima.

Innanzitutto, osserviamo la crescita del personale Engineering. Sono sicuro che concordiate sul fatto che questa tendenza è molto incoraggiante. Una crescita come quella rappresentata nella Figura 1.1 deve certamente indicare un significativo successo!

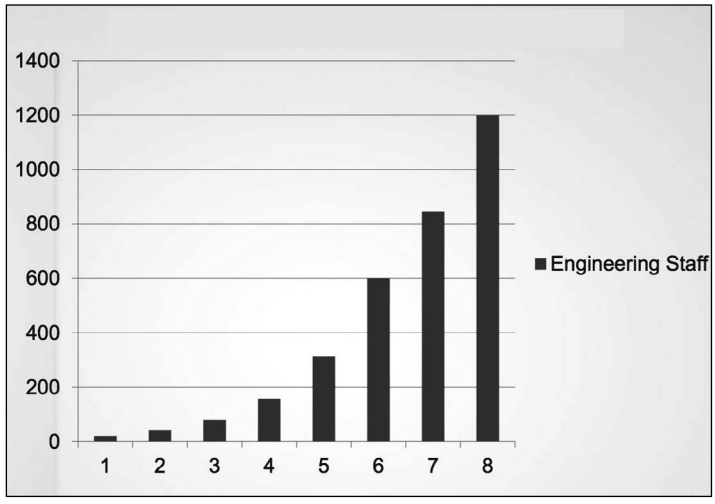

Figura 1.1 Crescita del personale nella categoria Engineering.
Tratta da una presentazione di Jason Gorman.

Ora però osserviamo la produttività dell'azienda nello stesso arco di tempo, misurata semplicemente in righe di codice (Figura 1.2).

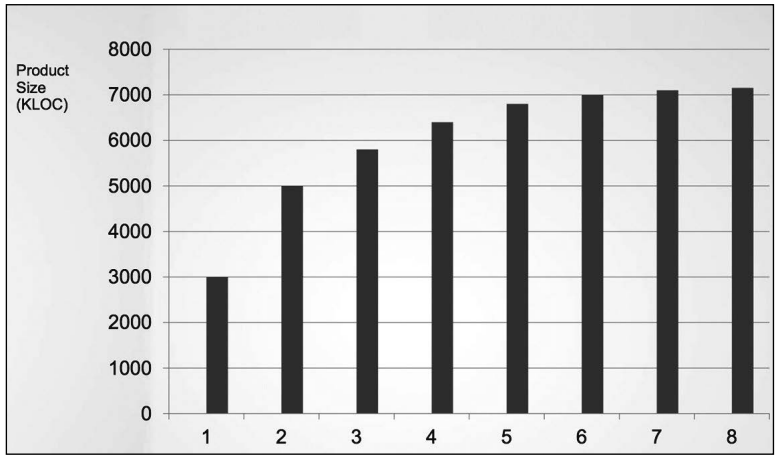

Figura 1.2 Produttività nello stesso arco di tempo.

È evidente che qualcosa non va. Anche se ogni release è supportata da un numero sempre crescente di sviluppatori, la crescita in termini di sviluppo di codice tende a un asintoto. Ora ecco il grafico davvero più spaventoso: la Figura 1.3 presenta il costo per riga di codice nel corso del tempo.

Questi trend non sono sostenibili. Non importa quanti profitti tragga ancora l'azienda: queste curve eroderanno in modo catastrofico i profitti dal modello di business e porteranno l'azienda a uno stallo, ma più probabilmente a un vero collasso.
Che cosa ha provocato questo evidente calo di produttività? Perché lo sviluppo del codice è diventato 40 volte più costoso nella release 8 rispetto alla release 1?

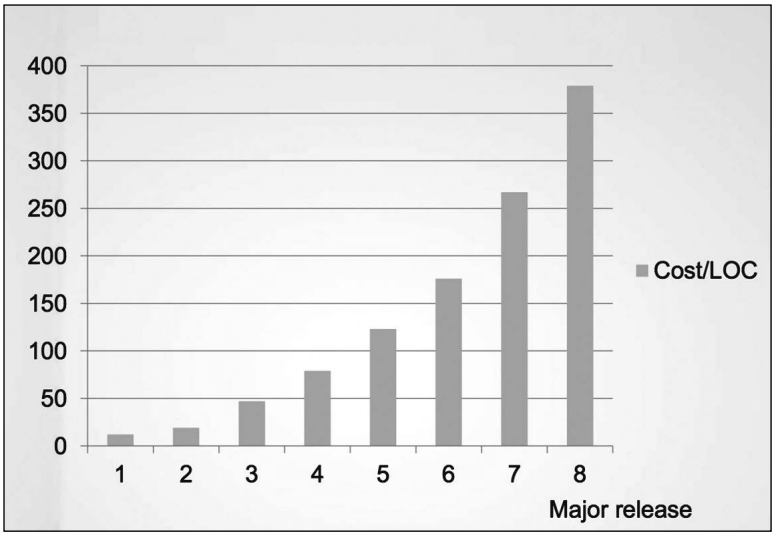

Figura 1.3 Costo per riga di codice nel corso del tempo.

Cronaca di un disastro annunciato

I dati che state osservando preannunciano un disastro. Quando i sistemi vengono messi insieme in fretta e furia, quando il semplice numero di programmatori è l'unica variabile di output e quando si tiene in poca (o nulla) considerazione la pulizia del codice o il design della struttura, potete benissimo immaginare dove sta portando questa curva.
La Figura 1.4 rappresenta le cose dal punto di vista degli sviluppatori. Iniziarono da una produttività di quasi il 100 percento, ma a ogni release la loro produttività ha subito un declino. Entro la quarta release, era chiaro che la loro produttività stava precipitando, con un approccio asintotico allo zero.
Dal punto di vista degli sviluppatori, questo è terribilmente frustrante, perché tutti stanno lavorando sodo. Nessuno ha ridotto il proprio impegno.
E ciononostante, pur con tutto il loro eroismo, gli straordinari e la dedizione, semplicemente non riescono più a procedere. Tutto il loro impegno, anziché essere rivolto allo sviluppo di funzionalità è ora consumato dalla sistemazione del groviglio di codice. Il loro lavoro, attualmente, è diventato spostare il groviglio da un posto all'altro e poi ancora e ancora, per fare posto a una piccolissima aggiunta.

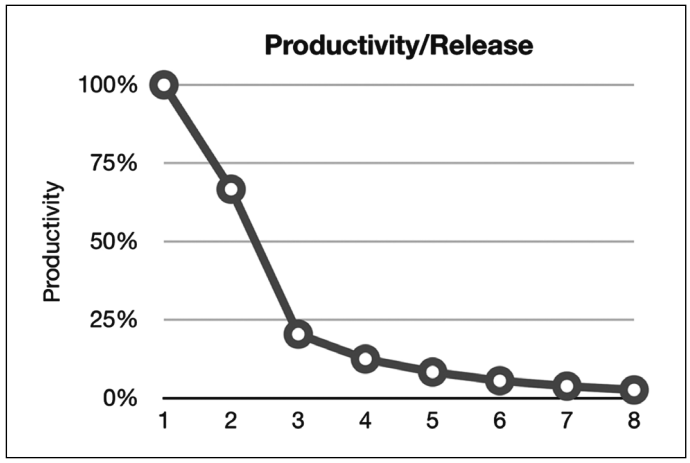

Figura 1.4 Produttività per release.

Il punto di vista della dirigenza

Se pensate che *questo* sia "male", immaginate che cosa ne deve pensare la dirigenza! Considerate la Figura 1.5, che rappresenta i costi di sviluppo mensili per lo stesso periodo.

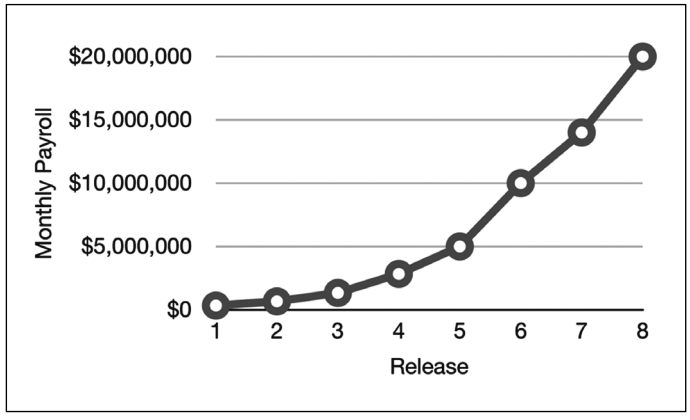

Figura 1.5 Costi di sviluppo mensili per release.

La release 1 è stata ottenuta con una spesa mensile di alcune centinaia di migliaia di dollari. La seconda release è costata qualche centinaio di migliaia in più. Entro l'ottava release, la spesa mensile era di 20 milioni, in ascesa.

Osservare anche solo questo grafico è preoccupante. Chiaramente sta accadendo qualcosa di tragico. Si spera che i profitti superino i costi e pertanto giustifichino le spese. Ma da qualsiasi prospettiva la si guardi, questa curva è solo fonte di preoccupazioni.

Ma ora confrontate la curva della Figura 1.5 con le righe di codice scritte per release nella Figura 1.2. Quelle poche centinaia di migliaia di dollari al mese hanno aggiunto molte funzionalità, mentre i 20 milioni di dollari finali non hanno aggiunto pratica-

mente nulla! Qualsiasi direttore finanziario, osservando questi due grafici capirebbe che è richiesta un'azione immediata per tentare di sventare il disastro.
Ma, esattamente, quale azione? Che cosa è andato storto? Che cosa ha causato questo incredibile declino della produttività? Che cosa può fare la dirigenza, a parte prendersela con gli sviluppatori?

Che cosa è andato storto?

Circa 2600 anni fa, Esopo raccontò la storia della lepre e della tartaruga. La morale di quella storia è stata affermata molte volte e in molti modi differenti.

- "La calma e la determinazione premiano."
- "La corsa non va al più veloce, né la battaglia al più forte."
- "Più fretta, meno velocità."

La storia è piena di esempi di stupidità e di sopravvalutazioni. La lepre, così fiduciosa della sua velocità, non prende sul serio la corsa e così si addormenta mentre la tartaruga taglia il traguardo.
Gli sviluppatori, spesso si sentono in una corsa di questo tipo e sopravvalutano le proprie forze. È vero, loro non dormono, tutt'altro. La maggior parte degli sviluppatori di oggi si impegna al massimo. Ma una parte del loro cervello *dorme*, quella parte che sa che nel codice la pulizia, l'ordine e la buona progettazione *contano*.
Questi sviluppatori credono al solito vecchio equivoco: "Possiamo sempre sistemarlo dopo; l'importante è metterlo sul mercato!". Naturalmente, le cose poi non vengono sistemate, perché le pressioni del mercato non calano mai. Uscire subito sul mercato significa semplicemente che avete orde di concorrenti alle calcagna e che per rimanere davanti a loro dovete correre più forte che potete.
E così gli sviluppatori non cambiano mai modalità di lavoro. Non possono tornare indietro a ripulire il codice, perché devono preparare la prossima funzionalità e poi quella successiva, quella dopo ancora e così via. E così il groviglio di codice cresce e la produttività continua nel suo approccio asintotico allo zero.
Così come la lepre aveva sopravvalutato la sua velocità, altrettanto gli sviluppatori sopravvalutano la loro capacità di rimanere produttivi. Ma i difetti presenti nel codice e che minano la loro produttività non dormono mai e restano in agguato. Se non vengono risolti, sono in grado di ridurre la produttività a zero nel giro di pochi mesi.
La più grande bugia alla quale gli sviluppatori credono è che del codice mal realizzato li faccia andare più velocemente nel breve periodo e dia qualche lieve problema a lungo termine. Gli sviluppatori che accettano questa bugia hanno la stessa fiducia della lepre nella propria capacità di cambiare modalità operativa, trovando in futuro il tempo di risolvere i problemi, ma commettono anche un semplice errore. Il fatto è che il *programmare male è sempre più lento del rimanere puliti*, qualsiasi scala si usi per misurare questa velocità.
Considerate i risultati di un interessante esperimento svolto da Jason Gorman e rappresentato nella Figura 1.6. Jason ha condotto questo test nell'arco di sei giorni. Ogni giorno completava un semplice programma di conversione di numeri interi in numerali romani. Sapeva che il suo lavoro era completo quando i suoi test di accettazione venivano passati. Ogni giorno l'operazione richiedeva poco meno di 30 minuti. I giorni dispari (primo, terzo e quinto), Jason usava una ben nota disciplina chiamata sviluppo TDD (Test-Driven Development). Gli altri tre giorni, scriveva il codice senza adottare tale disciplina.

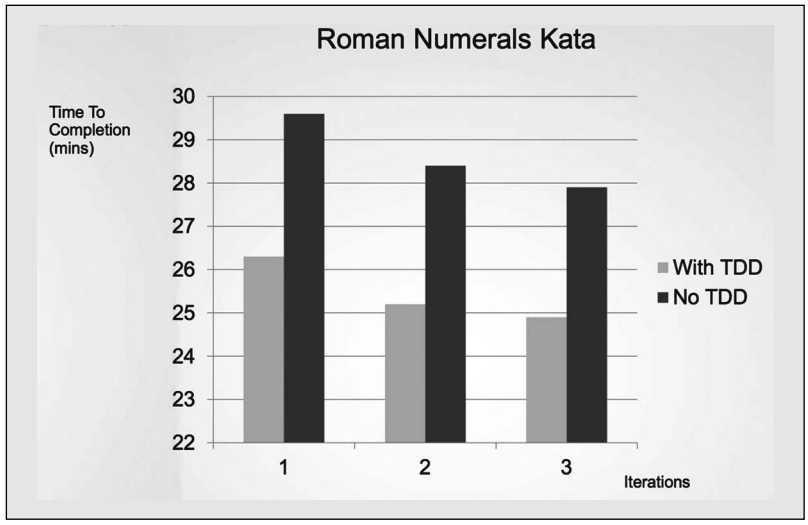

Figura 1.6 Tempo di completamento delle iterazioni e uso o meno dello sviluppo TDD.

Innanzitutto, notate la curva di apprendimento evidente nella Figura 1.6. Il lavoro, negli ultimi giorni, viene completato più rapidamente che nei primi. Notate anche che il lavoro nei "giorni TDD" è stato approssimativamente il 10 percento più rapido che negli altri giorni e che anche il più lento dei giorni TDD è stato comunque più veloce del più veloce dei giorni non-TDD.
Qualcuno potrebbe osservare questi risultati e pensare che siano notevoli. Ma per coloro che sono stati "scottati" dall'eccesso di fiducia della lepre, il risultato è solo naturale, perché conoscono questa semplice verità dello sviluppo di software:

> L'unico modo per procedere rapidamente, è fare le cose per bene.

E questa è anche la risposta del dilemma della dirigenza aziendale. L'unico modo per invertire il declino in termini di produttività e l'incremento nei costi consiste nel fare in modo che gli sviluppatori la smettano di ragionare come lepri e inizino ad assumersi la responsabilità per il groviglio che hanno prodotto.
Gli sviluppatori possono pensare che la risposta sia ripartire dall'inizio e riprogettare l'intero sistema, ma questo significherebbe ancora ragionare da lepri. Lo stesso eccesso di fiducia che ha condotto al groviglio di codice, ora dice loro che potrebbero rifarlo meglio, se solo avessero la possibilità di ripartire. La realtà è molto meno rosea:

> La sopravvalutazione guiderà la riprogettazione verso lo stesso esito del progetto originario.

Conclusioni

In assoluto, l'opzione migliore è che la società di sviluppo riconosca ed eviti ogni forma di sopravvalutazione e inizi a prendere sul serio la qualità dell'architettura del suo software.

Per prendere seriamente l'architettura del software, occorre sapere che cos'è una buona architettura del software. Per realizzare un sistema con una struttura, un'architettura che minimizzi l'impegno e massimizzi la produttività, occorre capire quali attributi dell'architettura di un sistema conducono a tale esito.
Ed esattamente questo è lo scopo di questo libro. Spiega che cosa si intende per architettura "pulita", per consentire agli sviluppatori di software di realizzare sistemi che si mantengano redditizi per tutto il loro arco di vita.