Introduzione

C'era una volta un consulente, che andò a vedere un progetto di sviluppo per esaminare parte del codice che era stato scritto. Percorrendo la gerarchia delle classi al centro del sistema, la trovò molto pasticciata. Le classi di livello più alto facevano alcune assunzionii su come le classi avrebbero dovuto funzionare – assunzioni che venivano incorporate nel codice ereditato. Quel codice però non era adatto a tutte le sottoclassi e veniva aggirato molto pesantemente. Piccole modifiche alla superclasse avrebbero ridotto di molto il bisogno di aggirarla. In altri punti, un'intenzione della superclasse non era stata compresa correttamente e il comportamento presente nella superclasse veniva duplicato. In altri punti ancora, varie sottoclassi facevano la stessa cosa con codice che chiaramente avrebbe potuto essere spostato verso l'alto nella gerarchia.

Il consulente consigliò ai responsabili del progetto di riesaminare e ripulire il codice, ma i responsabili non ne furono entusiasti. Il codice sembrava funzionare e le scadenze incombevano. I responsabili del progetto dissero che ci avrebbero pensato in qualche altro momento in seguito.

Il consulente aveva fatto vedere che cosa succedeva anche ai programmatori che lavoravano alla gerarchia, i quali erano attenti e videro il problema. Sapevano che non era realmente colpa loro; a volte serve un nuovo paio di occhi per individuare il problema. Così i programmatori passarono un giorno o due a ripulire la gerarchia. Quando finirono, avevano eliminato metà del codice nella gerarchia, senza ridurne la funzionalità. Erano soddisfatti del risultato e scoprirono che era diventato più rapido e più facile sia aggiungere nuove classi sia usare le classi nel resto del sistema.

I responsabili del progetto non erano contenti. Le scadenze erano strette e c'era una gran quantità di lavoro da fare. Quei due programmatori avevano sprecato due giorni a fare del lavoro che non aggiungeva nulla alle molte funzionalità che il sistema doveva offrire nell'arco di pochi mesi. Il vecchio codice funzionava perfettamente. Sì, il design era un po' più "puro" e un po' più "pulito", ma il progetto doveva consegnare codice che funzionasse, non codice che facesse la felicità di un accademico. Il consulente suggerì che una analoga pulizia avrebbe dovuto essere fatta su altre parti centrali del sistema, il che avrebbe bloccato il progetto per una settimana o due. Tutto per dare un aspetto migliore al codice, non per fargli fare qualcosa che non facesse già.

Che cosa pensate di questa storia? Credete che il consulente avesse ragione a suggerire un'ulteriore pulizia? O siete seguaci del vecchio motto da ingegneri, "se funziona, non aggiustarlo"?

Devo ammettere di essere un po' di parte, in questo caso: quel consulente ero io. Sei mesi dopo, il progetto è fallito, in gran parte perché il codice era troppo complesso per il debug o per la messa a punto per ottenere prestazioni accettabili.
È stato chiamato un consulente, Kent Beck, per far ripartire il progetto – un esercizio che ha comportato riscrivere da zero quasi tutto il sistema. Ha fatto molte cose in modo diverso, ma uno dei cambiamenti più importanti è stato insistere su una ripulitura continua del codice con il refactoring. Il miglioramento nell'efficacia del team e il ruolo che vi ha svolto il refactoring sono ciò che mi ha spinto a scrivere la prima edizione di questo libro, così da poter trasmettere le conoscenze che Kent e altri hanno acquisito utilizzando il refactoring per migliorare la qualità del software.
Da allora il refactoring è diventato una parte accettata del vocabolario della programmazione. Il volume originale ha retto bene al tempo, ma diciotto anni sono tanti per un libro di programmazione, perciò ho pensato che fosse venuto il momento di riprenderlo in mano e rielaborarlo. Nel farlo, ho più o meno riscritto tutte le pagine ma, in un certo senso, è cambiato pochissimo. L'essenza del refactoring è sempre la stessa; la maggior parte delle rifattorizzazioni fondamentali restano sostanzialmente le stesse. Spero però che la riscrittura aiuti ancora più persone ad apprendere come applicare in modo efficace il refactoring.

Che cos'è il refactoring?

Refactoring è il processo di modifica di un sistema software in un modo che non alteri il comportamento esterno del codice ma ne migliori la struttura interna. È un modo rigoroso di ripulire il codice che riduce al minimo le probabilità di introdurvi degli errori. In sostanza, quando si rifattorizza, si migliora il design del codice dopo che è stato scritto. "Migliorare il design dopo è che è stato scritto" è un'espressione strana. Per gran parte della storia dello sviluppo del software, quasi tutti sono stati convinti che prima si definisce il design, il progetto, e solo dopo si inizia a scrivere codice. Con il tempo il codice verrà modificato e l'integrità del sistema (la sua struttura, sulla base di quel design) gradualmente sfuma. Il codice affonda lentamente dall'ingegneria all'hacking. Il refactoring è l'opposto di questa pratica. Con il refactoring possiamo prendere un cattivo design, anche caotico, e trasformarlo in codice ben strutturato. Ogni passo è semplice, addirittura semplicistico. Sposto un campo da una classe a un'altra, estraggo del codice da un metodo per farne un metodo a sé stante, oppure sposto del codice su o giù lungo la gerarchia. L'effetto cumulativo di questi piccoli cambiamenti può però migliorare radicalmente il design. È l'esatto inverso della nozione di decadimento del software.
Con il refactoring, l'equilibrio del lavoro cambia. Ho scoperto che il design, invece di verificarsi tutto all'inizio, avviene con continuità per tutto il corso dello sviluppo. Mentre costruisco il sistema, imparo come migliorare il design. Il risultato di questa interazione è un programma il cui design rimane di buona qualità mentre lo sviluppo continua.

Che cosa c'è in questo libro?

Questo libro è una guida al refactoring ed è scritto per il programmatore di professione. Il mio obiettivo è mostrarvi come rifattorizzare in modo controllato ed efficiente.

Imparerete a rifattorizzare in modo da non introdurre errori nel codice e migliorarne invece metodicamente la struttura.
Trovo difficile introdurre il refactoring con una discussione generale o una serie di definizioni, perciò il libro inizierà con un esempio. Il Capitolo 1 prende un piccolo programma con alcuni difetti progettuali comuni e lo rifattorizza facendolo diventare un programma più facile da capire e da modificare. Questo vi mostrerà sia il processo del refactoring sia varie rifattorizzazioni utili. Questo è il capitolo fondamentale da leggere, se volete capire che cosa è realmente il refactoring.
Nel Capitolo 2 parlerò più a fondo dei principi generali del refactoring, fornirò alcune definizioni, e chiarirò i motivi per rifattorizzare, oltre a illustrare anche alcuni dei problemi. Nel Capitolo 3, Kent Beck mi aiuterà a descrivere come trovare ciò che "puzza" nel codice e come fare pulizia con le rifattorizzazioni. Il testing ha un ruolo molto importante nel refactoring, perciò il Capitolo 4 descrive come costruire test dentro il codice. Il cuore del libro, cioè il catalogo delle rifattorizzazioni, occupa il resto del volume. Non si tratta di un catalogo esaustivo, ma tratta le rifattorizzazioni fondamentali di cui avrà realmente bisogno la maggior parte degli sviluppatori. Si è formato dagli appunti che ho preso quando ho imparato il refactoring verso la fine degli anni Novanta: uso ancora quegli appunti perché non li ricordo tutti a memoria. Quando voglio fare qualcosa, come *Split Phase*, il catalogo mi ricorda come farlo in modo sicuro, passo per passo. Spero che questa sia la parte del libro a cui ritornerete spesso.

Un libro "Web-first"

Il World Wide Web ha avuto un impatto enorme sulla nostra società, influenzando in particolare il modo in cui raccogliamo le informazioni. Quando ho scritto questo libro, la maggior parte della conoscenza sullo sviluppo di software veniva trasmessa attraverso testi a stampa; oggi raccolgo la maggior parte delle mie informazioni online. Questo ha presentato una sfida per autori come me: c'è ancora un ruolo per i libri, e come dovrebbero essere?
Credo che per libri come questo ci sia ancora un ruolo, ma che debbano cambiare. Il valore di un libro è contenere un grande corpus di conoscenza assemblato in modo coerente. Nello scrivere questo libro, ho cercato di trattare molte rifattorizzazioni diverse e di organizzarle in modo coerente e integrato.
Ma quella totalità integrata è un lavoro letterario astratto che, anche se rappresentato tradizionalmente da un libro di carta, non deve essere necessariamente così in futuro. La maggior parte dell'editoria vede ancora il libro di carta come rappresentazione primaria e, anche se abbiamo adottato con entusiasmo gli ebook, sono solo rappresentazioni elettroniche di un lavoro originale basato sulla struttura di un libro cartaceo.
Con questo libro esploro un approccio diverso. La forma canonica del volume è il suo sito web o la sua edizione web. L'accesso all'edizione web è incluso con l'acquisto delle versioni a stampa o in ebook (vedi la nota seguente a proposito della registrazione del vostro prodotto su InformIT). Il libro cartaceo è una selezione di materiali tratti dal sito web, organizzata in modo da avere un senso per la stampa. Non tenta di includere tutte le rifattorizzazioni sul sito web, in particolare dal momento che potrei aggiungere altre rifattorizzazioni all'edizione canonica in futuro.
Non so se state leggendo l'edizione web online, un ebook sul vostro telefono, una copia cartacea o qualche altra forma che non so immaginare mentre scrivo queste

righe. Farò del mio meglio perché questo lavoro sia utile, in qualunque modo desideriate assimilarlo.
Per l'accesso all'edizione web canonica, agli aggiornamenti o alle correzioni quando saranno disponibili, registrate la vostra copia sul sito `http://www.informit.com`. Per iniziare il processo di registrazione, andate alla pagina `informit.com/register` ed effettuate l'accesso (o create un account se ancora non ne avete uno). Inserite l'ISBN 9780134757599 e fate clic su *Submit*. Vi sarà posta una domanda per dimostrare di avere una copia del libro a stampa o dell'ebook a disposizione. Registrata la vostra copia, aprite la scheda *Digital Purchases* della pagina del vostro account e fate clic sul collegamento sotto il titolo per lanciare l'edizione web.

Esempi JavaScript

Come nella maggior parte delle aree tecniche dello sviluppo software, gli esempi di codice sono molto importanti per illustrare i concetti. Le rifattorizzazioni però si presentano in sostanza nello stesso modo in linguaggi diversi. A volte ci sono cose particolari a cui un linguaggio mi costringe a prestare attenzione, ma gli elementi fondamentali della rifattorizzazione restano identici.
Ho scelto JavaScript per illustrare queste rifattorizzazioni, perché avevo l'impressione che questo linguaggio sarebbe stato leggibile dalla maggior parte delle persone. Non dovreste trovare difficile, però, adattare le rifattorizzazioni a qualsiasi linguaggio usiate. Cercherò di non utilizzare gli aspetti più complicati del linguaggio, perciò dovreste poter seguire le rifattorizzazioni anche con una conoscenza superficiale di JavaScript. Il mio uso di questo linguaggio certamente non è una "sponsorizzazione".
Anche se uso JavaScript per i miei esempi, questo non significa che le tecniche nel libro siano confinate a JavaScript. La prima edizione usava Java, e molti programmatori l'hanno trovata utile anche se non hanno mai scritto una singola classe Java. Ho preso in considerazione di illustrare questa generalità utilizzando una dozzina di linguaggi diversi per gli esempi, ma poi ho pensato che avrebbe potuto essere fonte di troppa confusione per il lettore. In ogni caso, questo libro è scritto per chi programma, qualsiasi sia il linguaggio in cui lo fa. Presumo che il lettore possa assimilare i miei commenti generali e applicarli al linguaggio che usa; immagino che potrà prendere gli esempi in JavaScript e adattarli al proprio linguaggio.
Questo significa che, al di là della discussione di esempi specifici, quando parlo di "classe", "modulo", "funzione" e simili, uso questi termini nel senso generale della programmazione, non come termini specifici del modello del linguaggio JavaScript.
Il fatto che usi JavaScript come linguaggio per gli esempi significa anche che cercherò di evitare gli stili JavaScript che sono meno familiari a quanti non programmano regolarmente in questo linguaggio. Questo non è un libro sul "refactoring in JavaScript": è un libro sul refactoring in generale che casualmente usa JavaScript. Esistono molte rifattorizzazioni interessanti che sono specifiche di Javascript (come quelle da callback, a promise, ad async/await), ma sono fuori dal raggio d'azione di questo libro.

Chi dovrebbe leggere questo libro?

Questo libro si rivolge a un programmatore di professione, qualcuno che scrive software come lavoro. Gli esempi e le analisi comprendono molto codice, che va letto e compreso. Gli esempi sono in JavaScript, ma dovrebbero essere applicabili alla maggior parte dei linguaggi. Do per scontato che un programmatore abbia un po' di esperienza per apprezzare quello di cui si parla in questo libro, ma non presumo che le sue conoscenze siano molto ampie.

Anche se il destinatario principale è uno sviluppatore che vuole imparare qualcosa sul refactoring, il libro sarà utile anche per qualcuno che già conosce l'argomento: può essere usato come un ausilio per l'insegnamento. In queste pagine mi sono impegnato a fondo per spiegare come funzionino le diverse rifattorizzazioni, in modo che uno sviluppatore esperto possa usare questo materiale per fare da mentore ai suoi colleghi.

Anche se è concentrato sul codice, il refactoring ha un forte impatto sul design di un sistema. È determinante per designer e architetti che vogliano comprendere i principi del refactoring e usarli nei loro progetti. La cosa migliore è che il refactoring venga introdotto da uno sviluppatore rispettato ed esperto, che possa comprendere al meglio i principi alla base del processo e adattare quei principi allo specifico ambiente di lavoro. Questo è vero in particolare quando si usa un linguaggio diverso da JavaScript, perché dovrete adattare gli esempi ad altri linguaggi.

Ecco come ottenere il massimo da questo libro senza leggerlo tutto.

Se volete capire che cos'è il refactoring, leggete il Capitolo 1: l'esempio dovrebbe chiarire il processo.

Se volete capire perché dovete rifattorizzare, leggete i primi due capitoli. Vi diranno che cos'è il refactoring e perché lo dovete applicare.

Se volete scoprire dove dovete rifattorizzare, leggete il Capitolo 3: vi dice quali sono i segni che suggeriscono il bisogno di rifattorizzare.

Se volete effettivamente applicare il refactoring, leggete completamente i primi quattro capitoli, poi sfogliate il catalogo: leggetene abbastanza da capire, a grandi linee, che cosa contiene. Non è necessario capire tutti i dettagli. Quando avrete effettivamente bisogno di applicare una rifattorizzazione, leggetela in dettaglio e usatela come aiuto. Il catalogo è una sezione di consultazione, perciò probabilmente non vorrete leggerlo tutto di fila.

Una parte importante della scrittura di questo libro è stata dare un nome alle varie rifattorizzazioni. La terminologia ci aiuta a comunicare, in modo che quando uno sviluppatore consiglia a un altro di estrare del codice e metterlo in una funzione, o di suddividere in fasi separate qualche computazione, entrambi comprendano il riferimento a *Extract Function* e a *Split Phase*. Questo vocabolario aiuta anche a selezionare le rifattorizzazioni automatizzate.

Costruire su fondamenta gettate da altri

Devo dire subito che per questo libro ho un grande debito verso tutti quelli il cui lavoro nel corso degli anni Novanta ha portato allo sviluppo del campo del refactoring. Imparare dalla loro esperienza mi ha ispirato e spinto a scrivere la prima edizione e, anche se sono trascorsi molti anni, è importante che continui a riconoscere le fondamenta che

hanno gettato. Sarebbe stato giusto che uno di loro avesse scritto quella prima edizione, ma sono stato io quello che ha avuto il tempo e l'energia per farlo.
Due dei primi principali proponenti del refactoring sono stati Ward Cunningham e Kent Beck. Lo hanno usato come fondamento dello sviluppo, inizialmente, e hanno adattato i loro processi di sviluppo per sfruttarlo. In particolare, è stata la collaborazione con Kent che mi ha mostrato l'importanza del refactoring, un'ispirazione che ha portato direttamente a questo libro.
Ralph Johnson guida un gruppo all'Università dell'Illinois a Urbana-Champaign che è degno di nota per i suoi contributi pratici alla tecnologia a oggetti. Ralph è da molto tempo un alfiere del refactoring, e molti dei suoi studenti hanno dato contributi fondamentali a questo campo ai suoi inizi. Bill Opdyke ha sviluppato il primo lavoro scritto dettagliato sul refactoring con la sua tesi di dottorato. John Brant e Don Roberts sono andati oltre la scrittura e hanno creato il primo strumento automatizzato di refactoring, il Refactoring Browser, per la rifattorizzazione di programmi Smalltalk.
Molti hanno contribuito allo sviluppo del campo del refactoring dalla prima edizione di questo libro. In particolare, il lavoro di quanti hanno aggiunto rifattorizzazioni automatizzate agli strumenti di sviluppo hanno contribuito enormemente a rendere più facile la vita dei programmatori. Per me è facile dare per scontato di poter ricordare una funzione di uso frequente con una semplice sequenza di tasti, ma questa facilità si basa sugli sforzi dei team IDE il cui lavoro è di aiuto a tutti noi.

Ringraziamenti

Anche con tutte quelle ricerche su cui basarmi, ho avuto comunque bisogno di molto aiuto per scrivere questo libro. La prima edizione ha ricavato molto dall'esperienza e dall'incoraggiamento di Kent Beck. Mi ha fatto conoscere il refactoring, mi ha dato lo spunto per iniziare a scrivere appunti per documentare le rifattorizzazioni, e mi ha aiutato a trasformare il tutto in una prosa sensata. È stato lui ad avere l'idea dei Code Smells. Spesso penso che avrebbe scritto la prima edizione meglio di quanto abbia fatto io, se non fosse che stava scrivendo il libro fondamentale per l'Extreme Programming.
Tutti gli autori di libri tecnici che conosco citano il grande debito che hanno nei confronti dei revisori tecnici. Tutti abbiamo scritto lavori con grossi difetti che sono stati identificati da nostri pari con funzioni di revisori. Personalmente non svolgo molto lavoro di revisione tecnica, in parte perché non penso di essere molto bravo in questo campo, perciò provo grande ammirazione per quanti se lo accollano. Non si ricava nulla rivedendo il libro di qualcun altro, perciò si tratta di un grande atto di generosità.
Quando ho iniziato a lavorare seriamente sul libro, ho creato una mailing list di consiglieri che mi potessero dare dei feedback. Nell'andare avanti, inviavo bozze dei nuovi materiali a questo gruppo e chiedevo il loro parere. Per avere inviato i loro feedback alla mailing list vorrei ringraziare Arlo Belshee, Avdi Grimm, Beth Anders-Beck, Bill Wake, Brian Guthrie, Brian Marick, Chad Wathington, Dave Farley, David Rice, Don Roberts, Fred George, Giles Alexander, Greg Doench, Hugo Corbucci, Ivan Moore, James Shore, Jay Fields, Jessica Kerr, Joshua Kerievsky, Kevlin Henney, Luciano Ramalho, Marcos Brizeno, Michael Feathers, Patrick Kua, Pete Hodgson, Rebecca Parsons e Trisha Gee.
All'interno di questo gruppo, vorrei citare in particolare l'aiuto speciale che ho ricevuto su JavaScript da Beth Anders-Beck, James Shore e Pete Hodgson.

Una volta completata la prima stesura, l'ho fatta circolare per ulteriori revisioni, perché volevo che qualcuno la leggesse nella sua interezza con occhi nuovi. William Chargin e Michael Hunger mi hanno entrambi fornito commenti dettagliatissimi. Ho ricevuto molti commenti utili anche da Bob Martin e Scott Davis. Bill Wake ha aggiunto ai suoi contributi sulla mailing list una revisione completa della prima stesura.

I miei colleghi alla ThoughtWorks sono una fonte costante di idee e feedback su quello che scrivo. Sono innumerevoli le domande, i commenti e le osservazioni che hanno alimentato la riflessione e la scrittura di questo libro. Una delle cose più belle dell'essere un dipendente della ThoughtWorks è che mi consente di dedicare parecchio tempo alla scrittura. In particolare, mi sono preziose le conversazioni con Rebecca Parsons, la nostra CTO, e le idee che ne ricavo.

Alla Pearson, Greg Doench è il mio acquisition editor, in grado di risolvere i tanti problemi che sorgono sulla strada che porta alla pubblicazione di un libro. Julie Nahil è la mia production editor. Sono stato felice di lavorare ancora con Dmitry Kirsanov per la redazione e con Alina Kirsanova per la composizione e gli indici.