Parte IV
Refactoring

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

(Qualsiasi stupido riesce a scrivere codice che possa essere compreso da un computer. I bravi programmatori scrivono codice che può essere compreso dagli umani.)

MARTIN FOWLER

Capitolo 21
Introduzione al refactoring

Refactoring significa "fattorizzare" il codice, riorganizzarlo, rivederlo per venire meglio incontro alle esigenze dello sviluppatore. Dopo un ciclo di refactoring, il funzionamento del codice rimane invariato. Non c'è aggiunta di funzionalità rispetto alla versione precedente. Semplicemente la struttura interna viene rivista e migliorata. Migliorata per ulteriori modifiche e per maggiore chiarezza. Il concetto di refactoring passa dalla modifica del nome di una variabile, a un radicale cambiamento dell'architettura del sistema. Si tratta di un argomento semplice, quasi banale, ma che dal lato pratico ha una forte influenza sulla vita del software.

In realtà il vero punto non è tanto il refactoring in quanto tale, la modifica del codice apportata dallo sviluppatore. La cosa essenziale è invece la definizione di una lista di refactoring comuni, in cui vengano spiegati i passaggi da eseguire, per ridurre al minimo la probabilità di inserire errori durante la loro implementazione. Questi passaggi stanno alla base dell'automatizzazione di un largo numero di refactoring.

Sistemi di sviluppo per Smalltalk offrono da anni la possibilità di intervenire nel codice attraverso metodi di refactoring automatizzati. Martin Fowler, dedicando un intero libro all'argomento [Fowler-1999] non ha quindi portato niente di nuovo a livello teorico. Ha però avuto il grosso merito di divulgare questo argomento a comunità molto più ampie (come quella Java) meno abituate a queste comodità. Sono così arrivati i primi sistemi di sviluppo con integrati i

primi refactoring automatizzati, prima per Java, poi per C#. I pionieri sono stati IntelliJ IDEA e Eclipse, seguiti poi da altri.

Impatto sul design

L'idea di refactoring deve però andare oltre le liste preconfezionate. La fase di refactoring è parte integrante dello sviluppo, automatizzata, dove possibile, o manuale, ancora nella maggior parte dei casi.

La modifica del nome di un package in un progetto Java è un'operazione che, se non viene automatizzata, può diventare lunga e tediosa, portando il codice ad attraversare stati di inconsistenza. È quindi importante che passaggi di questo tipo vengano automatizzati, eseguendo controlli incrociati di dipendenze tra i vari moduli di un sistema. Pur ammettendo, quindi, l'importanza di un refactoring del genere, è evidente che da un punto di vista concettuale la cosa non è così densa di significato.

Più interessanti sono allora i metodi di refactoring più difficilmente automatizzabili, quelli che hanno anche un impatto sul design dell'applicazione. Il fatto di considerarli parte dello sviluppo permette di relativizzare l'importanza del design iniziale, proprio come preconizzato dai metodi di sviluppo agile.

L'utilizzo sistematico di metodi di refactoring permette di sfatare il luogo comune, secondo cui prima viene il design completo, poi il codice che lo implementa. Abbiamo già visto in fase di introduzione come la differenza tra design e programmazione sia minima, entrambi vanno di pari passo. Se non fosse così avremmo il continuo tentativo di far evolvere il codice senza modificare l'architettura, ottenendo velocemente una struttura interna degradata.

Con la progettazione agile, prima viene il design "iniziale". Poi, durante l'implementazione del sistema si impara come migliorarlo e meglio adattarlo alle funzionalità richieste all'applicazione.

È necessario lavorare con cicli molto corti. Avere la capacità di eseguire refactoring significa essere in grado di adattare e ricreare continuamente il design durante l'implementazione. Un aspetto interessante a proposito di design: i refactoring più complessi hanno spesso come scopo finale l'aggiunta nel codice di un design pattern.

Utilizzo sistematico durante lo sviluppo

Vediamo di seguito alcuni aspetti positivi nell'utilizzo sistematico di cicli di refactoring.

Miglioramento del design

Senza refactoring il design di un software tende a decadere. Molte modifiche tendono a essere modifiche per scopi a corto termine, senza la necessaria comprensione del design, così il codice perde struttura e diventa in poco tempo illeggibile. Un esercizio regolare di refactoring aiuta a mantenere la struttura del codice.

Software più leggibile

La programmazione è sempre stata considerata comunicazione con il computer. Questo punto di vista non tiene però sufficientemente conto che il computer non è il solo a dover leggere e interpretare il codice. Altre persone dovranno leggerlo e capirlo in futuro. La cosa interessante è che spesso queste persone sono le stesse che hanno scritto il software… Scrivere un software leggibile è nel nostro stesso interesse.

Cosa importa se il compilatore impiega due cicli in più per compilare il codice, se la modifica che abbiamo introdotto migliora la leggibilità del nostro programma diminuendo il lavoro di ore o giorni a chi dovrà leggerlo? È normale pensare solo al calcolatore durante un primo approccio al programma. Ma in un secondo tempo ci si deve fermare ed eseguire alcuni cicli di refactoring per migliorare la struttura dell'applicazione e renderla più chiara e accessibile.

Si scoprono errori in anticipo

Ogni sviluppatore esperto sa che eseguendo regolarmente refactoring si aumenta in continuazione la comprensione del codice, si semplificano strutture e si trovano errori, spesso in anticipo rispetto alla loro apparizione in esecuzione. Eseguendo refactoring, può capitare di rimettere in discussione funzioni di test scritte in precedenza. Questa rimessa in discussione permette di verificare se effettivamente tutti i casi erano stati considerati.

Si programma più velocemente

Questo aspetto è una diretta conseguenza dei punti precedenti, anche se potrebbe sembrare il contrario. È chiaro che c'è un miglioramento della qualità: si migliora il design, la leggibilità e si correggono errori. Non c'è però un rallentamento dovuto a queste attività?

No, perché migliorare regolarmente il design serve a mantenere alta la velocità di sviluppo. Un design che si deteriora nel tempo rallenta in modo esponenziale lo sviluppo e aumenta la probabilità di inserire nuovi bug ad ogni modifica.

Refactoring e sistemi di sviluppo

Un buon sistema di sviluppo è essenziale per realizzare progetti in modo efficiente. A maggior ragione da quando alcuni ambienti integrati di sviluppo offrono tutta una serie di refactoring automatici. È chiaro che posso sviluppare in Java utilizzando un semplice editor, ma cosa succede se devo eseguire un'operazione che necessita l'aggiornamento di più file? Ad esempio, che conseguenze ha, in termini di tempo, la modifica del nome, o peggio della struttura, di una serie di package?

In un ambiente con refactoring la modifica dura pochi secondi, perché ogni dipendenza viene gestita dal sistema. Se programmo con un editor, l'operazione può durare ore. Conseguenza? Minore produttività e, soprattutto, design meno accurato, perché se per modificare la struttura di un package impieghiamo ore, è chiaro che la tendenza sarà quella di lasciare le cose come stanno… Avendo insegnato anche la programmazione ai primi semestri dei corsi universitari, sono consapevole dell'importanza di insegnare i fondamenti usando sistemi di sviluppo semplici. Lo

scopo è quello di rendere attenti sugli elementi di un ciclo di sviluppo (edit, compile, link, execute) e separare i problemi di programmazione pura e semplice da quelli di apprendimento di un ambiente di sviluppo. In questo approccio ci sono considerazioni di tipo didattico. Quindi, quando si impara un linguaggio, meglio farlo con un semplice editor, non con ambienti di sviluppo avanzati. Appena si inizia a lavorare a un progetto è però altrettanto essenziale far capire l'importanza di un buon ambiente di programmazione, proprio perché la produttività permessa da un semplice editor è insufficiente. Alcuni anni fa ricordo di aver sentito uno studente, durante la presentazione del suo lavoro di diploma, andar fiero per aver realizzato l'intero lavoro con "vi", uno degli editor più primitivi in ambiente Unix… Vorrei unicamente che si capisse che lavorare in questo modo è possibile, ma non deve essere considerato un vanto.

Passaggi di refactoring

Ciò che ha portato alla parziale automatizzazione di alcuni elementi di refactoring è stata la suddivisione di questi in passaggi. Ogni passaggio ha lo scopo di inserire una piccola modifica nel codice, permettendo però al più presto la compilazione, l'esecuzione del programma e l'esecuzione dei test.

Il test è un elemento essenziale. Una buona infrastruttura di test dà coraggio allo sviluppatore, permettendogli di eseguire modifiche nel codice funzionante, senza alcuna inibizione. È importante che chi sviluppa non sia frenato da paure nel modificare il codice: porterebbero ad evitare miglioramenti nel sorgente e, di conseguenza, a situazioni di design decaduto, senza più possibilità di intervento.

Il test serve a stabilire se, dopo i passaggi di refactoring, le funzionalità precedenti sono ancora le stesse. Obiettivo della suddivisione in passaggi è quello di rendere il più corti possibile gli stati di inconsistenza attraverso cui il codice dovrà forzatamente passare. Anche questi rappresentano un'ulteriore spinta per vincere la paura e dedicarsi anima e corpo al refactoring.

Il refactoring in quanto tale, sarà da considerare completo solo al termine di tutti i passaggi intermedi previsti.

Gestione del codice in team e integrazione continua

Non ne abbiamo parlato in modo esplicito nei capitoli del libro, ma tra gli aspetti essenziali nell'utilizzo di metodologie agili ci sono senza dubbio la capacità di gestire il codice all'interno di un gruppo e l'integrazione continua.

Nel primo caso si intende la possibilità data a tutti i membri del gruppo di intervenire sul codice del progetto, anche quello realizzato da altri, gestendo la versione principale in un archivio (*repository*) condiviso. Esistono parecchi prodotti, anche open source, che facilitano questo compito, obbligando nel contempo tutti i membri del team di sviluppo a una certa disciplina. Praticamente tutti i prodotti esistenti offrono, oltre alla condivisione di un repository centrale, la gestione delle diverse versioni e della storia dei cambiamenti effettuati nel tempo, importanti per il ripristino di situazioni antecedenti. I più noti sono CVS (Concurrent Versioning System) e Subversion.

Dal momento che tutti i membri del team fanno capo a un'unica versione centrale, diventa essenziale integrare le proprie modifiche (effettuate localmente, sulla propria macchina) nel

repository. Inoltre, affinché questa versione centrale possa corrispondere alla versione produttiva, è importante che ci siano test a sufficienza in grado di verificare che le modifiche (aggiunte o refactoring) inviate dai vari membri del team non abbiano compromesso la consistenza del programma.
Ci sono prodotti (uno su tutti, Cruise Control) che automatizzano questo passaggio di verifica nel repository, permettendo la cosiddetta "integrazione continua". Ad ogni nuovo arrivo sul repository, viene chiamato uno script, che, fra le altre cose, può far partire la sequenza prevista di test, inviando, ad esempio, un allarme via e-mail nel caso in cui ci fossero test che non passano.

Esempio

Consideriamo il caso di voler unire sotto una classe astratta in comune (superclass) due classi fino a questo momento indipendenti (refactoring: Extract Superclass). In figura 21.1 è riportato lo schema del design che si evolve:
Questi sono i passaggi necessari all'operazione.

- Creare una classe astratta vuota e fare ereditare le classi originali dalla nuova classe astratta.

- Compilare ed eseguire i test.

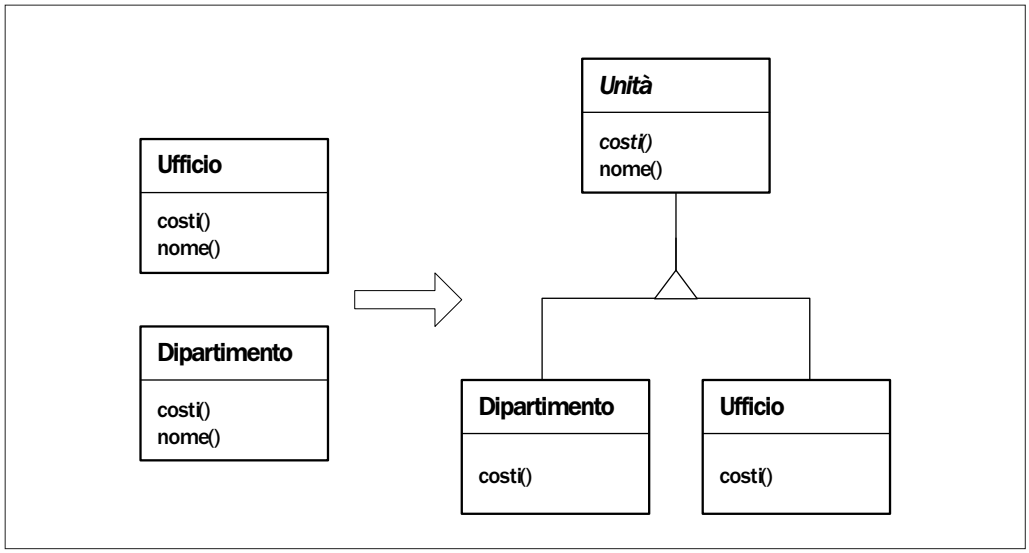

Figura 21.1 – Schema per Extract Superclass.

- Ad uno ad uno, spostare gli elementi in comune (campi, metodi, contenuto dei costruttori) nella superclasse.
- Compilare ed eseguire i test dopo ogni spostamento.
- Terminati questi passaggi, controllare l'utilizzo delle classi originali da parte di classi client. Se queste vengono utilizzate unicamente attraverso l'interfaccia comune stabilita dalla superclasse, allora si può modificare il tipo delle variabili da classe specifica a superclasse (ricordare: "program to the interface, not to the implementation").

Elenco di alcuni refactoring

Nel suo libro sui refactoring, Fowler presenta un elenco di 72 elementi. Lo scopo qui non è certo quello di riproporla in modo completo, anche perché ce ne sono alcuni molto complessi, che richiedono parecchie spiegazioni, e altri talmente semplici che non serve approfondirli, anche se il fatto di poterli avere automatizzati è essenziale per la produttività nello sviluppo.

Refactoring Extract

Vediamo dapprima una lista di refactoring non complicatissimi, ma estremamente utili, si tratta dei refactoring di tipo Extract, che hanno in comune l'idea di creare qualcosa di nuovo (classe, metodo, interfaccia) a partire dalla struttura esistente. Sono tutti automatizzabili:

Extract Class

Crea una nuova classe aggregata alla classe esistente e sposta alcuni campi dalla vecchia classe alla nuova.

Extract Hierarchy

Da una classe di partenza crea una gerarchia di classi, in cui ogni sottoclasse rappresenta un caso speciale. Abbiamo eseguito un passaggio del genere nell'esempio introduttivo del libro, quando abbiamo creato la gerarchia di `TipoSci`.

Extract Interface

Si estrae un'interfaccia (*interface*) da una classe esistente, per definire meglio una API e per separare meglio interfaccia da implementazione.

Extract Method

Si estrae una sequenza di istruzioni e si forma un metodo contenente la sequenza.

Extract Subclass

Se una classe ha elementi utilizzati unicamente in alcune istanze specifiche, si può estrarre una sottoclasse, utilizzata solo in queste istanze.

Extract Superclass

Se due classi indipendenti hanno elementi in comune, si può creare una superclasse e spostare gli elementi in comune nella nuova classe.

Refactoring Push/Pull

Una seconda serie di refactoring semplici è quella contrassegnata da Push e Pull:

Pull Up Constructor Body

Se si hanno più sottoclassi con costruttori simili, meglio riunire tutto nel costruttore della superclasse, richiamandolo da quello della sottoclasse. Questo refactoring sposta il contenuto (*body*) dei costruttori nel costruttore della superclasse.

Pull Up Field

Sposta nella superclasse un campo in comune tra tutte le sottoclassi.

Pull Up Method

Sposta nella superclasse un metodo in comune tra tutte le sottoclassi.

Push Down Field

Sposta dalla superclasse in una sottoclasse un campo usato solo nella sottoclasse coinvolta.

Push Down Method

Come sopra, ma con un metodo. Se una funzionalità di superclasse è rilevante unicamente in una sottoclasse, la si sposta in questo modo.

Refactoring complessi

Una lista di refactoring un po' più complessi è invece questa:

Replace Conditional with Polymorphism

Già vista nell'esempio introduttivo del libro. Permette di spostare ogni singolo braccio di un'espressione condizionale in un metodo di una sottoclasse, che riscrive il metodo originale, diventato abstract. Viene usato quando in un'espressione condizionale il comportamento dipende dal tipo dell'oggetto.

Replace Inheritance with Delegation

Permette di sostituire la relazione di ereditarietà con quella di composizione. Si usa quando una sottoclasse riutilizza solamente una minima parte della classe da cui eredita: in questo caso è consigliato usare una relazione di composizione.

Replace Delegation with Inheritance

Permette di sostituire la relazione di composizione con quella di ereditarietà. Si usa quando una classe ha bisogno di sempre più funzionalità da una sua classe delegata: in questo caso diventa comodo modificare il tipo di relazione.

Separate Domain from Presentation

Serve a separare gli elementi di una GUI dalla logica del programma. La complessità del refactoring dipende dalla complessità del programma da modificare. In ogni caso esistono passaggi generici, che verranno mostrati con un esempio più avanti nel libro.

In questo capitolo…

In questo capitolo introduttivo della Parte IV abbiamo visto che i metodi di refactoring sono elementi che hanno un ruolo importante durante i cicli di sviluppo. Permettono di preparare il codice a ulteriori aggiunte di funzionalità.

Una regolare attività di refactoring aiuta a mantenere il design pulito e aggiornato alle ultime modifiche di funzionalità. Un buon refactoring è possibile unicamente con una buona infrastruttura di test.