Capitolo 1
Introduzione

"Architecture is the set of design decisions that you wish you could get right early in a project, but that you are not necessarily more likely to get them right than any other."

(L'architettura rappresenta la serie di scelte sul design che si desidererebbero capire bene agli inizi di un progetto, ma che non hanno necessariamente maggiore probabilità di essere capite bene prima delle altre.)

RALPH JOHNSON

I molti libri esistenti sulle metodologie di progettazione e ingegneria del software dimostrano la necessità di definire nuovi percorsi e nuovi metodi per rendere l'attività di progettazione e sviluppo più affidabile e più consona alla crescente complessità delle architetture software. La voglia e la necessità di standardizzazione hanno portato all'accettazione di metodologie e strumenti come UML (Unified Modeling Language) considerate oggi parti integranti del processo di progettazione. Il costante mutamento di mezzi e tecnologie che si registra nel campo informatico porta però sempre a dubitare che ciò che viene ritenuto oggi valido lo sia effettivamente anche domani. In una situazione professionale in cui l'ultima tecnologia sembra sempre indispensabile e in cui i neo-laureati sembrano avvantaggiati rispetto ai loro predecessori, diventa fondamentale identificare e valorizzare, anche nell'ambito della progettazione software, quali sono gli elementi e le conoscenze più costanti nel tempo, su cui vale la pena investire e costruire la propria esperienza.

Fortunatamente, oltre la nebbia di tecnologie e nuovi tool, alla base di tutto rimangono pur sempre le capacità di design e di programmazione, che, sebbene fortemente influenzate da

giuste scelte organizzative e metodologiche, si basano sull'esperienza del programmatore e sulla sua maturata capacità di astrazione. Il concetto di esperienza ricopre un ruolo fondamentale in questo libro sugli aspetti pratici di progettazione e sviluppo software.

Tratteremo primariamente i temi dei modelli di design nella programmazione a oggetti, dei metodi di refactoring e del test automatico di unità. Questi elementi sono alla base delle metodologie di sviluppo cosiddette "agili" (Agile Software Development). Non ne tratteremo una in particolare, ma nel corso del libro avremo modo di riferirci varie volte a due tra le più conosciute: sia la più nota XP (Extreme Programming), sia la "meno estrema" TDD (Test-Driven Development).

Le metodologie usate durante la progettazione sono spesso influenzate dal linguaggio di programmazione utilizzato. Così, se per la maggior parte degli sviluppatori Smalltalk, molti concetti introdotti con le metodologie agili sono già noti, lo stesso non si può dire per molti programmatori Java, provenienti, o perlomeno influenzati, per buona parte, da C++.

Metodi agili

Senza voler entrare nel dettaglio delle metodologie agili, serve introdurre un paio di punti che stanno alla base delle stesse e che le contrappongono alle metodologie cosiddette "tradizionali". Prima di tutto due definizioni, estratte dai principi che stanno alla base della progettazione agile ([agile-manifesto]):

1. I metodi agili sono adattivi, in contrapposizione ai metodi predittivi.
   Molte metodologie di ingegneria software tendono a voler pianificare ogni dettaglio per lunghi periodi di tempo. Questo funziona bene solo fino a quando arriva la richiesta di una modifica. La loro natura è perciò quella di opporsi al cambiamento. I metodi agili, invece, sono aperti alle modifiche.

2. I metodi agili sono rivolti alla persona, piuttosto che ai processi.
   Un principio fondamentale nei metodi agili consiste nel fatto che i processi di sviluppo non hanno la pretesa di sostituire in nessun caso il team di sviluppo. Servono invece da supporto al lavoro di design e implementazione.

Una metafora spesso usata nel passato per cercare di descrivere e motivare i metodi di progettazione software tradizionali era quella dei metodi presi dall'ingegneria civile. In questo caso si tratta di prevedere tutto, fin nei minimi particolari, già in fase di progettazione (*design*), per trasformare la fase di realizzazione in una fase puramente esecutiva, in cui gli unici problemi possano essere unicamente ricondotti al contesto costruttivo.

In conseguenza dell'adozione di tale schema, i metodi di ignegneria software dividono il lavoro in due fasi distinte:

1. La fase di design, in cui ogni minimo dettaglio deve essere ipotizzato e ogni particolare richiesta del programma deve essere prevista.

2. La fase realizzativa, che rappresenta invece la realizzazione di tutti i dettagli specificati in precedenza.

Le domande da porsi a questo punto sono tre, suggerite in [Fowler]:

- È veramente possibile separare in modo netto l'attività di design da quella di codifica?
- Se sì, è possibile ottenere un design in grado di trasformare la fase realizzativa in una fase di costruzione completamente pianificata e prevista?
- Se sì, il costo di questa attività di pianificazione fin nei minimi dettagli è sufficientemente basso rispetto al resto, da giustificarne l'utilizzo sistematico?

Partiamo dall'ipotesi che sia possibile rispondere affermativamente alla prima domanda. L'esperienza di software designer e il fatto che non esistano metodi matematici in grado di dimostrare la correttezza di un design per un certo tipo di problema permettono di rispondere in modo negativo alla seconda domanda: non è cioè possibile ottenere un design in grado di trasformare la fase realizzativa in una fase di costruzione completamente pianificata e prevista.

Anche se però volessimo a tutti i costi rispondere in modo affermativo alla seconda, proviamo ad analizzare la terza domanda, confrontando la struttura dei costi di un progetto software con quello di un progetto di ingegneria civile. Nell'attività ingegneristica tradizionale il costo maggiore di un progetto consiste nella sua fase realizzativa (~90%). Diventa quindi evidente l'importanza di un design in grado di valutare ogni singolo dettaglio della realizzazione. Nell'ingegneria software, per grossi progetti, la fase realizzativa rappresenta invece la parte minore del progetto (da 15% a 40%). Non si giustifica quindi un investimento troppo grosso nella fase precedente, per risparmiare tempo nel 15% di fase realizzativa.

Ingegneria civile

È da quando ero studente che mi vengono regolarmente proposti confronti tra l'ingegneria del software e l'ingegneria civile, quasi si trattasse della stessa cosa. Stavo preparando una lezione di ingegneria del software quando mi imbattei in un articolo di Martin Fowler, in cui si affermava che nell'ingegneria civile la percentuale del costo di realizzazione rappresenta circa il 90% dei costi totali. Mi sembrava una di quelle cifre che uno scrive ben sapendo che il suo articolo verrà letto da persone poco interessate all'ingegneria civile, e che quindi non andranno a cercare conferma. Il caso vuole, però, che mio fratello sia proprio ingegnere civile… Chiesi subito conferma, attendendomi una risposta del tipo: "È difficile dirlo con certezza, dipende molto dai casi, dalle situazioni, ogni progetto ha una sua storia…". Con mia sorpresa, invece, mi confermò senza esitazione questo valore. Chi commissiona un ponte o un qualsiasi altro progetto edilizio di grosse dimensioni, sa in modo abbastanza preciso cosa vuole.

In informatica non sempre è così. Se poi, oltre alla grossa differenza in percentuali, consideriamo che il committente di software, nella maggior parte dei casi, all'avvio del progetto non sa ancora esattamente cosa desidera, e si aspetta di poter specificare meglio i requisiti nel corso della realizzazione, il quadro è completo: non è realistico fare confronti fra questi due campi.

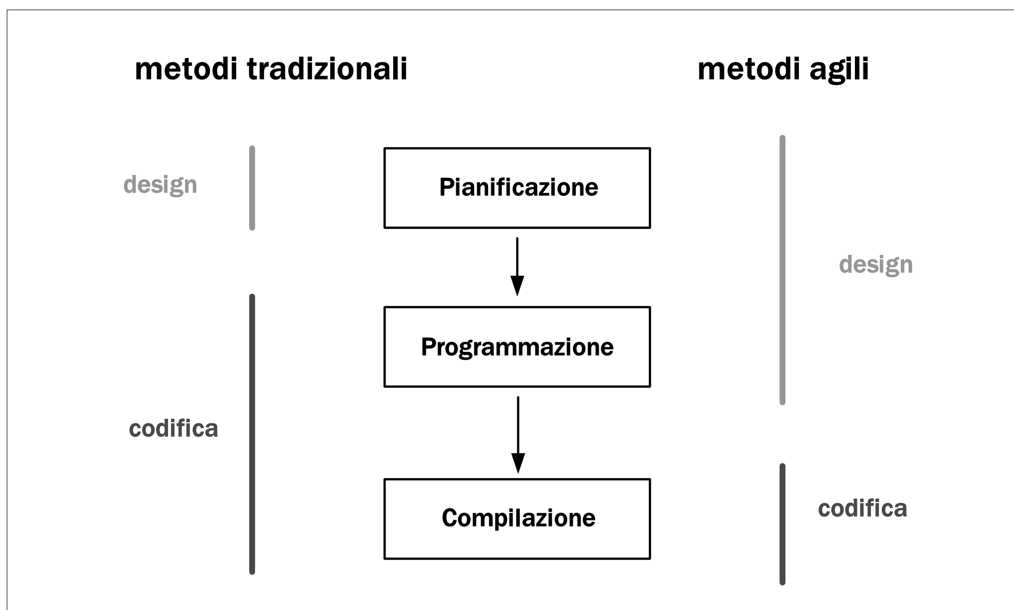

Figura 1.1 – Spostamento del design nella fase di programmazione.

Ma questo non è tutto. Nel 1992 Jack Reeves ([Reeves-1992]) ha suggerito che, in effetti, il codice sorgente, quindi la fase di programmazione, dev'essere considerata una fase intellettiva di design. Ha pubblicato questa sua affermazione quando è passato dall'esperienza di progettazione in C a quella in C++. A maggior ragione questo vale con l'utilizzo di linguaggi più dinamici ed evoluti, come Smalltalk, Lisp e anche Java, per fare alcuni esempi, soprattutto se aiutati da moderni ambienti di sviluppo. Per Reeves la fase di realizzazione si limita alla compilazione e al link, cioè a fasi completamente automatizzate, quindi ben al di sotto, in termini di costi, del 15% minimo considerato in precedenza. Questa visione della fase di realizzazione sottolinea ulteriormente (se ce ne fosse bisogno) la differenza tra pianificazione per progetti di ingegneria civile e pianificazione per progetti di ingegneria software. Infatti, sempre secondo questa visione:

- In progetti software la fase puramente realizzativa è da considerare completamente automatizzata e, di conseguenza, a costo bassissimo, o perlomeno velocemente ammortizzabile.

- Lo sforzo maggiore del progetto è il design, che comprende anche la programmazione, che richiede quindi persone di talento, formate e creative.

- Dobbiamo guardare con spirito critico e diffidenza alla facile metafora dell'ingegneria tradizionale quando parliamo di ingegneria software.

Capacità creative

Ho sempre avuto un interesse particolare per i linguaggi di programmazione e per il loro lato estetico. Questo non è sicuramente estraneo al fatto, fortunato, di aver avuto Niklaus Wirth, conosciuto per essere il padre del linguaggio Pascal, come professore durante gli anni di Modula-2 e Oberon ([Wirth-1987]).

L'illusione, un po' immodesta, che la programmazione richieda anche capacità creative mi ha accompagnato sin dall'inizio della mia attività. So di non essere l'unico a pensarla in questo modo e sono sicuro che l'avvicinamento della programmazione al design, avvenuta anche grazie ai design pattern, abbia dato a questa illusione un'ulteriore spinta…

Imprevedibilità

Uno degli elementi costanti nella progettazione software è l'imprevedibilità dei requisiti. A scanso di equivoci, diciamo subito che il cambiamento dei requisiti non deve per forza significare che la loro analisi sia stata fatta in modo troppo superficiale. Il business è infatti in continuo movimento; così anche un set di requisiti che sembrava studiato e ben definito può richiedere una modifica improvvisa. In fondo il software non è un ponte autostradale e il cliente ha il diritto di aspettarsi che le modifiche siano possibili in ogni momento, anche in fase realizzativa.

Questo non fa che confermare quanto detto in precedenza: se si deve accettare che i requisiti cambino con il tempo non ci si può aspettare di prevedere tutto nella prima fase.

Si tratta quindi di controllare l'imprevedibilità. Il modo migliore per controllare un processo in un ambiente non prevedibile è quello di costruire un sistema di feedback che ci mostri la situazione attuale a intervalli regolari. Si passa quindi in modo chiaro da uno sviluppo a cascata a uno di tipo iterativo (incrementale, prototipale, ecc.).

La chiave di questo tipo di sviluppo consiste nella continua produzione della versione finale con un subset di funzionalità. Tranne le funzionalità mancanti, la versione dev'essere già quella prevista come versione finale, integrata, e con i test di unità completi. Si capisce quindi come diventino determinanti i tre elementi principali trattati in questo libro: design pattern (flessibilità e riutilizzo di design), refactoring (evoluzione del codice) e test (controllo di consistenza ed evoluzione).

Design pattern

I primi studi e articoli sui design pattern risalgono alla fine degli anni Ottanta. Il primo libro che ha riassunto e divulgato questi studi è stato pubblicato per la prima volta nel 1995 e i suoi contenuti sono tutt'ora più che attuali.

I design pattern sono l'essenza dell'esperienza nella progettazione del software perché non solo descrivono architetture utilizzate nel passato, rendendole in questo modo riutilizzabili, ma permettono al programmatore neofita di sfruttare al meglio l'esperienza di "esperti" in questo campo. In un certo senso i design pattern rappresentano una condivisione di esperienza da parte di chi ha avuto l'occasione di approfondire un certo tipo di problema di progettazione. Questo approfondimento può essere sfruttato da tutti.

Metodi di refactoring

Refactoring significa modificare la struttura interna di un programma lasciando inalterata la sua funzionalità. Lo scopo è quello di migliorarne la comprensione, semplificarne la modificabilità e aumentare le possibilità di riutilizzo delle singole componenti. Un'operazione di refactoring diventa necessaria nel momento in cui viene richiesta un'estensione di funzionalità al programma e si desidera preparare il codice in modo che "accetti" senza problemi l'aggiunta. In altre parole il programma stesso deve essere preparato per includere l'estensione in modo omogeneo, se possibile seguendo uno degli obiettivi della programmazione ad oggetti, cioè quello di trasformare ogni futuro cambiamento in un'aggiunta.

In cosa consiste qui il valore dato dall'esperienza? In primo luogo refactoring significa metter mano nel codice, modificare la struttura, passando da stati inconsistenti, per arrivare di nuovo allo stato consistente di partenza. Modificare design e fattorizzare codice sono operazioni che fanno parte del bagaglio di esperienza di ogni informatico. In secondo luogo, passaggi complessi di refactoring hanno molto spesso l'obiettivo di introdurre un design pattern nell'architettura esistente, e i pattern, per definizione, nascono dall'esperienza.

Test di unità

Il terzo elemento pratico che si vuole enfatizzare in questo libro consiste nel test automatico di unità. Con questo si intende la possibilità di avere una sequenza di test sistematica, ripetibile e accumulabile nel tempo. La presenza di test di unità è garanzia non solo di affidabilità del programma, ma anche di fiducia del programmatore nel codice. Solo uno sviluppatore con una consistente serie di test che gli garantiscono l'affidabilità del programma è in grado di apportare modifiche nel codice e mandare lo stesso in produzione. L'utilizzo regolare di metodi di refactoring richiede una buona organizzazione del test. I test di unità rappresentano l'impalcatura necessaria per poter eseguire modifiche nel codice.

Obiettivi della progettazione

Se si mettono in relazione i tre elementi pratici di progettazione e sviluppo evidenziati sopra con alcuni riconosciuti obiettivi della progettazione software, si può evidenziare la loro importanza per l'ottenimento degli obiettivi stessi.

Affidabilità

Il test di unità serve a dimostrare l'affidabilità e la consistenza a livello di codice.

Modificabilità

Un software molto usato necessita di una continua serie di modifiche, sia per iniziativa del team di sviluppo, sia su richiesta dell'utente. Nel caso della modificabilità, i tre elementi considerati hanno tutti un ruolo importante. I design pattern descrivono architetture conosciute. Attraverso queste architetture vengono definite collaborazioni tra classi, che specificano in anticipo elementi di staticità e di dinamicità. La descrizione di questi elementi permette di capire in breve tempo dove vanno effettuate modifiche per un determinato scopo all'interno del codice.

I metodi di refactoring sono per definizione modifiche nel codice. Le modifiche vengono eseguite per preparare il codice a ulteriori aggiunte o per migliorarne il grado di leggibilità. Visto che lo scopo del test è quello di garantire la consistenza prima e dopo processi di refactoring, anche il test è da considerare un elemento fondamentale per l'obiettivo di modificabilità del codice.

Comprensione

Si parla qui solitamente di comprensione a livello di progetto, perché un progetto poco comprensibile frena la codifica. Visto che lo scopo del libro è quello di trattare la progettazione dal punto di vista della realizzazione, spostiamo il significato di comprensione a livello di design e codice. I design pattern sono senza dubbio un importante elemento di comprensione e di chiarezza del design. Non solo: l'utilizzo di pattern ci permette di documentare il design direttamente nel codice, evitando lunghi commenti di documentazione che rischiano di essere vecchi già dopo le prime modifiche. Definiscono infatti un vocabolario per parlare e discutere di design. Il refactoring stesso contribuisce alla chiarezza del programma. Uno dei motivi per iniziare un processo di refactoring è infatti il miglioramento della chiarezza nel codice.

Riutilizzo

L'obiettivo finale delle tecniche di riutilizzo è di giungere a disporre di un insieme di componenti elementari di alto livello da comporre per poter generare più comodamente le applicazioni richieste. I design pattern sono per definizione "elementi di riutilizzo di design", servono cioè a mettere a disposizione architetture da riutilizzare. Anche i passaggi di refactoring servono a questo scopo: sono infatti modifiche al codice e al design per aumentarne il grado di riutilizzo e adattabilità.

Conseguenze

L'utilizzo sistematico di design pattern, metodi di refactoring e test nel processo di sviluppo di software ha alcune conseguenze.

In primo luogo c'è una maggiore e più diretta interazione tra design e codice. Non si parte con l'idea di progettare per mesi ogni singolo dettaglio prima di passare all'implementazione, ma si parte da un design iniziale, ben analizzato, ma comunque da considerare iniziale, per poi passare velocemente alla programmazione, iniziando un processo iterativo continuo tra design e codice. In altre parole c'è minore separazione tra progettazione pura e codifica (metodi agili).

A questo contribuiscono naturalmente i pattern, con la loro visione del software, che rende l'attività di sviluppo più importante e interessante.

L'importanza rivalutata della programmazione e del ruolo del programmatore (*software engineer*), responsabile anche del design (*software architect*), porta ad una minore gerarchia all'interno del progetto, in cui la suddivisione dei compiti avviene in modo più orizzontale e meno verticale, e a una maggiore importanza della formazione del programmatore.