Programmazione Orientata agli Oggetti

Qualità del codice:
Coesione e Accoppiamento

Contenuti

- Accoppiamento
- Coesione
- Introduzione a:
  - Testing
  - Refactoring

Il Software Evolve

- Il software evolve in continuazione
- ...inevitabilmente...
- Viene esteso, corretto, mantenuto, portato su altre piattaforme, adattato, ...
- Molte persone, in tempi diversi partecipano a questo processo
- Se il costo dell'evoluzione è troppo alto, il software viene gettato

Qualità del Codice

- La qualità del codice dipende da due fattori importanti:
  - Accoppiamento (coupling)
  - Coesione (cohesion)

Accoppiamento

- Due o più unità di un programma si dicono accoppiate quando è impossibile modificare una senza dover modificare anche le altre
- L'accoppiamento si riferisce ai legami tra unità separate e distinte di un programma
- Se due classi dipendono strettamente e per molti dettagli l'una dall'altra, diciamo che sono *strettamente accoppiate*
- Per un codice di qualità dobbiamo puntare ad un *basso accoppiamento*

Duplicazione del Codice

- “Codice Copia e Incolla”
- La duplicazione del codice
  - è sintomo di un cattivo progetto
  - porta facilmente alla propagazione di errori durante lo sviluppo
  - rende difficile la manutenzione
  - porta inevitabilmente alla introduzione di errori nelle attività di manutenzione
- E’ una forma elementare di accoppiamento

Basso Accoppiamento

- Un basso accoppiamento permette di:
  - Capire il codice di una classe senza leggere i dettagli delle altre
  - Modificare una classe senza che le modifiche comportino conseguenze sulle altre classi
- Quindi un basso accoppiamento migliora la manutenibilità del software

Coesione

- La coesione fa riferimento al numero e alla eterogeneità dei compiti di cui una singola unità è responsabile
- Se ciascuna unità è responsabile di un singolo compito, diciamo che tale unità possiede una *alta coesione*
- La coesione si applica alle classi e ai metodi (ed anche ai package!)
- Perseguiamo l’alta coesione

Alta Coesione

- Un’alta coesione favorisce:
  - La comprensione dei compiti di una classe o di un metodo
  - L’utilizzo di nomi appropriati, efficaci, comunicativi
  - Il riuso delle classi e dei metodi

Coesione

- Coesione dei metodi
  - Un metodo dovrebbe essere responsabile di un solo compito ben definito
- Coesione delle classi
  - Ogni classe dovrebbe rappresentare un singolo concetto ben definito

Coesione ed Accoppiamento

- Sono le due facce della stessa medaglia
  - L’alta coesione di una classe si ottiene perseguendo lo scarso accoppiamento di quella classe verso altre classi
  - Lo scarso accoppiamento di una classe verso le altre classi si ottiene perseguendo la sua alta coesione

Localizzare le Modifiche

- Basso accoppiamento e alta coesione portano ad una localizzazione delle modifiche
- Quando è necessario operare una modifica, il minor numero possibile di classi dovrebbero essere coinvolte

- La qualità del proprio codice si osserva proprio quando sorge l’esigenza di fare modifiche

Pensare in Avanti

- Quando progettiamo una classe dovremmo sforzarci di pensare a
  - quali cambiamenti potranno essere richiesti in futuro
  - come verrà usata la nostra classe dal programmatore-utilizzatore
- Le nostre scelte iniziali potrebbero facilitare l’evoluzione futura (su questo aspetto faremo esperienza nello studio di caso ed in altri esercizi)

Qualità Interna vs Qualità Esterna del Codice

- La qualità del codice può essere osservata da molti punti di vista ben distinti
- Due particolarmenti interessanti per noi
  - Quello degli utilizzatori finali: *qualità esterna*
  - Quello degli sviluppatori: *qualità interna*

- Gli utilizzatori finali possono fornire un giudizio di merito sulla capacità di un applicativo di rispondere ai requisiti ed alle proprie esigenze
- Solo gli sviluppatori possono valutare la manutenibilità del codice nel momento in cui nasce la necessità di modificarlo

“Refactoring”

- La manutenzione del software spesso richiede l’aggiunta di nuovo codice
- Le classi e i metodi tendono così a diventare più lunghi, a perdere in coesione, ad aumentare l’accoppiamento versi altre porzioni di codice
- A seguito delle modifiche, per mantenere un’alta coesione ed un basso accoppiamento, classi e metodi dovranno essere riorganizzate
- Questo processo di riorganizzazione del codice viene definito *“refactoring”*
- ✓ E’ il principale strumento che uno sviluppatore possiede per controllare la qualità **interna** del codice
  - Modifiche del codice che non alterano il funzionamento
  - Ma preparano il codice ad accogliere le modifiche future

Refactoring e Testing

- Quando modifichiamo il codice, è necessario isolare le conseguenze del refactoring da altri fattori
- Quindi la riorganizzazione delle classi e dei metodi (il refactoring) deve essere effettuata prima di introdurre nuove funzionalità
- Come contrastare la naturale paura di fare modifiche su un base di codice che si considerava funzionante?
- Per assicurarci di non aver introdotto nuovi errori, eseguiamo i test prima e dopo ogni azione di refactoring
  - Vedremo come organizzare metodicamente refactoring e testing

Linee Guida (1)

- Domande comuni
  - Quanto dovrebbe essere lunga una classe?
  - Quanto dovrebbe essere lungo un metodo?

- Possiamo rispondere in termini di coesione e accoppiamento

Linee Guida (2)

- Un metodo è troppo lungo se è responsabile di più di un compito logico
- Una classe è troppo complessa se rappresenta più di un concetto, se ha più di una responsabilità

- **Nota**: queste sono *linee guida* – solo attraverso l'esperienza si riescono a concretizzare

Caso di Studio: le classi `Diadia` e `Partita`

- Le classi **`Diadia`** e **`Partita`** sono particolarmente lunghe
- Se guardiamo il loro codice ci accorgiamo che hanno diverse (troppe!) responsabilità
  - **`Diadia`** implementa la logica di tutti i possibili comandi
  - **`Partita`** gestisce lo stato del gioco e crea il labirinto
- Questa miriade di responsabilità è indice di poca coesione

Aumentiamo la Coesione

- Possiamo iniziare a migliorare la coesione della classe **`Partita`** togliendole qualche responsabilità
- Iniziamo a togliere la responsabilità di creare e gestire il labirinto
- Affidiamo questa responsabilità ad una nuova classe appositamente introdotta: **`Labirinto`**

Esercizio

- Creare la classe **`Labirinto`** e modificare la classe **`Partita`** affinché non abbia la responsabilità della creazione del labirinto
  - Un labirinto ha una entrata (stanza di ingresso) ed una uscita (stanza vincente)
  - La classe **`Labirinto`** ha un metodo privato **`init()`** che inizializza il labirinto
- Provare ad eseguire il codice del gioco prima e dopo le modifiche e verificare che il comportamento sia rimasto invariato

Refactoring …

- Un programmatore OO deve avere forte senso critico, evidenziare i limiti delle proprie soluzioni sia in termini di funzionalità che di qualità del codice per risolverli attraverso disciplinati passi di refactoring
  - N.B. con un pizzico di attenzione al pericolo della sovra-ingegnerizzazione
    - ✓ bisogna risolvere i problemi di oggi del progetto! Poi, se c'è la possibilità, anche quelli che si prevedono
    - ✓ mai compromettere le soluzioni di **oggi** per bisogni che **forse** avremo in futuro

Refactoring …

- Concludiamo citando alcune problematiche risolvibili efficacemente solo dopo aver studiato il **polimorfismo**
  - Forte accoppiamento tra l'insieme dei comandi disponibili e la classe **`Diadia`**: risolvendo il primo homework è evidente che ogni volta che aggiungiamo/modifichiamo un comando dobbiamo agire su questa classe
    - Anche dopo i passi di refactoring precedenti la classe **`Diadia`** continua ad addossarsi diverse (troppe!) responsabilità
      - Implementa la logica del gioco
      - Implementa la logica di tutti i possibili comandi
  - Ma è possibile che la struttura del labirinto sia cablata all'interno della classe **`Labirinto`**? E pensando ad un gioco a più livelli di difficoltà con diversi labirinti?
  - Se volessi cambiare labirinto o comunque riutilizzare un oggetto istanza della classe **`Partita`** con diversi labirinti, potrei farlo?

Ricapitoliamo

- I programmi SW sono in continua evoluzione
- E’ importante facilitare e prevedere questa evoluzione
- La qualità del codice richiede molto più del corretto funzionamento di un programma in un preciso momento
  - “funziona!”… condizione necessaria, ma non sufficiente
  - in altri termini:
    - se funziona non è detto che vada bene
    - se non funziona certamente non va bene
- Il codice deve essere comprensibile e manutenibile

Ricapitoliamo

- Codice di buona qualità: “no copia e incolla”, alta coesione, basso accoppiamento
- Anche lo stile di codifica (identificatori, indentazione, spaziatura) è fondamentale (il programma è uno strumento di comunicazione)
  - codice ben scritto non ha bisogno di commenti!
- Il costo di manutenzione del software dipende fortemente dalla qualità del codice ed è ormai da tempo noto essere la voce di costo preponderante sul medio/lungo termine