Progettare visibilità

Sunday, 28 May 2023 15:22

- Visibilità = Capacità di un oggetto di poter vedere un altro oggetto
  - Si progetta un sistema come un sistema di oggetti software che comunicano tra di loro, quindi hanno bisogno di una visibilità
  - La visibilità si può ottenere:
    - Attributo, aka attributo pubblico
      - Quando B è un attributo di A
      - Permanente = Fino a che A e B esistono
    - Parametro, aka A ha un metodo e B lo chiama con il parametro
      - Visibilità temporanea = Appena finito il metodo non ce lo abbiamo più
      - Si passano le istanze
    - Locale, B è un oggetto locale di un metodo di A
      - B è dichiarato come oggetto locale di A
      - E' temporanea
      - Crea una nuova istanzia
      - E' possibile trasformare visibilità locale in visibilità parametro
    - Globale, si assegna ad una variabile globale (non esiste in java)
      - Però in java si può utilizzare in singenton
- Trasformare il progetto in codice
  Per farlo bisogna:
  - Definire classi ed interfacce
  - Variabili di istanza
  - Metodi e costruttori

  Si consiglia di scrivere il codice dalla classe con accoppiamento più basso (più indipendente)
- Refractoring e test
  - Lo sviluppo guidato dai test è una best practice
    - Chiamato sviluppo preceduto dai test
    - Prima test, e poi codice (SISI contaci che io inizio a fare la parte più noiosa all'inizio)
      - Ci aiuta ad avere più chiario ciò che vogliamo sviluppare
      - I test vengono realmente scritti (disolito vengono fatti alla fine)
      - Verifica automatica e dimostrabile

- Verifica automatica e dimostrabile
- Si ha una fiducia quando facciamo i cambiamenti

Tipologie di test:

- Unità
  - Testiamo singoli componenti in isolamento
  - Divisi in 4 sezioni:
    - Preparazione
      - Creato l'oggetto da verificare
    - Esecuzione
      - Si esegue
      - Bisogna identificare insieme di input che permettono di raggruppare tutti gli scenari = Partizione di equivalenza
        E nota: bisogna includere anche input che non dovrebbero risultare output corretti
        Immaginiamo le partizioni come un sottoinsieme dell'insieme "tutti i possibili input". Certe partizioni potrebbero intersecare altre partizioni

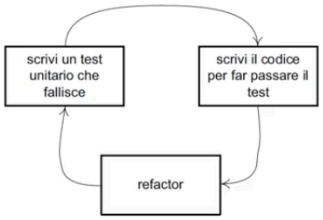
      Linee guida per testare input:
      - Testare casi limiti
      - Input che forzano tutti messaggi di errori
      - Input che fanno l'overflow
      - Ripetere tante volte stesso input
      - Overflow/Underflow
      - Null/Zeri

    - Verifica
      - I risultati ottenuti corrispono a ciò che desideriamo
    - Rilascio
      - Si ritorna alla situazione originale
      - Non deve fallire il test successivo per via di un qualcosa corrente
  - Ciò che permette:
    - NUnit per .NET
    - JUnit per java
    - Unittest per python (ciò che utilizzo io per i miei progetti!)
- Integrazione
  - Isolamento più unità
- Si

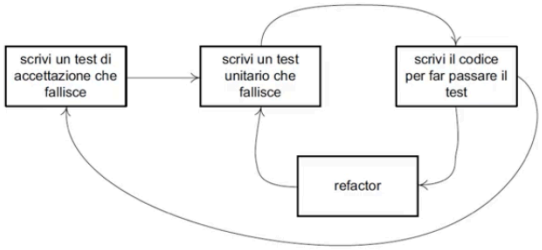
- Sistema
  - Tutto il sistema come scatola nera (come l'utente lo vede)
- Accettazione
  - Come l'utente finale recepisce il sistema

Noi useremo il test unitario.

- Cicli del TDD



Il ciclo di base del TDD per i test unitari.



Il doppio ciclo del TDD.

- Refractoring
  - Un metodo strutturato e disciplinato per riscrivere/ristrutturare codice esistente senza modificare il comportamento esterno. Bisogna eseguire testing passo dopo passo
  - Dovrebbe migliorare il codice
    - Rimuovere codice duplicato
    - + chiarezza
    - Abbreviare metodi lunghi
    - Non dovrebbe distruggere casi di test
  - Bisogna applicarlo:
    - Regola del tre
    - Aggiungere funzionalità
    - Dopo un bug
    - Durante la revisione del codice
  - Code smells (Cioè codici che dovrebbero avere un refractoring)
    - Long method
      - Un metodo con troppe linee di codice
      - I metodi lunghi sono difficili da comprendere, visualizzare e modificare
      - Per ridurre:
        - Extract method
          - Muovere linee di codice in una funzione
        - Replace templete query
          - Mi sembra una versione modificata/simile di sopra????
        - introduce parameter object
          - Sostituire parametri primitivi con oggetti
        - preserve whole object

            - ???? Passiamo un oggetto ad una funzione????
              Non approvo con certi dettagli siccome stai distruggendo la leggibilità del codice
        - Spostare l'intero metodo in un altro oggetto
            - Trasformare un metodo in una classe che è dentro ad una classe creata apposta (interessante
        - E' possibile spostare loop ed operatori condizionali
            - Un if è troppo lungo? Trasformiamolo in funzione
- Duplicate code
    - Lo stesso codice appare in molti luogi
    - Soluzione: usare extract method
- Feature envy
    - Un metodo accede ai dati di un altro oggetti più ai propri dati
    - Aka, non abbiamo applicato bene information expert
    - Usare move method
- Large class
    - Una classe contiene molti campi/metodi/linee di codice
    - Errore di progettazione. Alcuni metodi potrebbero apparire ad altre classi
    - Extract class/move method/Extract subclass
- Switch statement
    - Abbiamo un completto switch oppure un complesso if
      Aka una struttura con troppe casisiche
    - Soluzione: rimpiazzio condizione con poliformismo
      Oppure isolare lo switch con extract method e poi move method
- Data class
    - Una classe ha solo variabili e getter/setter
    - Problema: In certi casi questo potrebbe andare bene, secondo me
    - Soluzione: extract /movemethod
- Long parameter list
    - Troppi parametri passati
    - Problema: difficile da comprendere
    - Soluzione

- Soluzione:
  - Se alcuni parametri sono il risultato di altri metodi, chiamarlo
  - Chiamare l'oggetto intero
  - Unire parametri in 1 oggetto

- Shotgun surgery
  - Per apportare 1 modifica, bisogna apportare tante piccole modifiche in tante classi
  - Problema: La responsabilità è troppo sparpagliata
  - Soluzione: creare una nuova classe e fare una classe nel mezzo
- Comment
  - Un metodo è ricco di commenti esplicativi
  - Problema: codice dovrebbe essere autoesplicativo, se si hanno bisogno di lunghi commenti per la descrizione allora il codice non è chiaro
  - Soluzione:
    - Suddividere 1 espressione in sotto espressioni
    - E' possibile traspotare una sezione in un metodo
    - Se un metodo è già stato spostato, potremmo rinominarlo
    - Se c'è bisogno di affermare regole, introduce assertion

```
double getExpenseLimit() {
 // Deve avere un limite di spesa o
 // un progetto primario.
 return (expenseLimit != NULL_EXPENSE) ?
   expenseLimit :
   primaryProject.getMemberExpenseLimit();
}
```

```
double getExpenseLimit() {
 Assert.isTrue(expenseLimit != NULL_EXPENSE || primaryProject != null);
 return (expenseLimit != NULL_EXPENSE) ?
   expenseLimit:
   primaryProject.getMemberExpenseLimit();
}
```

Qui io c'ho problemi

- Refused bequest
  - Una sottoclasse utilizza solo alcuni metodi di un genitori
  - Problemi: gerarchia non corretta
  - Soluzione: rimuovere ereditarietà con ua delega Oppure fare una extract superclass

Oppure fare and extract superclass

Abbiamo finito APS :D