LEZIONE 25
PROGETTAZIONE CON RESPONSABILITÀ
&&
INTRODUZIONE AI PATTERN

Ingegneria del Software e Progettazione Web

Università degli Studi di Tor Vergata - Roma

Guglielmo De Angelis
guglielmo.deangelis@isti.cnr.it

cosa è UML :

"*In short, the Unified Modeling Language (UML) provides industry standard mechanisms for visualizing, specifying, constructing, and documenting software systems.*"

cosa è UML

"*In short, the Unified Modeling Language (UML) provides industry standard mechanisms for visualizing, specifying, constructing, and documenting software systems.*"

- UML è un linguaggio
- UML non un impone l'adesione ad uno specifico processo di sviluppo del software
- la conoscenza "sintattica" di UML non è funzione di un'appropriata applicazione dei principi O.O.

UML: mezzo non soluzione

- l'esperienza nella modellazione O.O. suggerisce
  - principi generali
  - soluzioni ricorrenti
  - aspetti da considerare in funzione:
    - del contesto
    - della fase del processo adottato
    - degli obiettivi da raggiungere
- tuttavia spesso è intesa come una “forma creativa” basata su abilità personali e attitudini derivanti dallo studio e dalla pratica
  - una forma di artigianato?!?!?

RDD – 1

- un modo di pensare alla progettazione O.O.
  - **R**esponsibility **D**riven **D**esign
- gli oggetti software sono associati ad una descrizione delle loro responsabilità
  - obblighi sulla struttura o sul comportamento di un oggetto in relazione al suo ruolo all'interno del sistema
  - sono assegnate alle classi ed agli oggetti dal progettista

RDD – 2

- definire elementi del sistema
  - responsabilità
    - di fare
      - istanziare un oggetto, computare un calcolo,
      - dare inizio alle attività di altri oggetti
      - controllare e coordinare parte delle attività di altri oggetti
    - di conoscere
      - i propri dati privati incapsulati
      - gli oggetti correlati
      - cose che possono essere derivate o calcolate
  - ruoli
  - collaborazioni

GRASP

- **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**rinciples (o **P**atterns)
  - principi di supporto all'apprendimento/progettazione O.O.
  - si basano sui concetti base di RDD

- GRASP vs GoF
  - GRASP sono principi generali che guidano le scelte progettuali attraverso l'assegnamento di responsabilità
  - GoF sono soluzioni tipiche a problemi ricorrenti (vedi lezioni 27 e 30)
    - idee di progettazione avanzate

GRASP

- **G**eneral **R**esponsibility **A**ssignme nciples (o **P**atterns)
  - principi di suppor ne O.O.
  - si basan
- C
  - scelte progettuali
    a ilità
  - Gu blemi ricorrenti (vedi lezioni 27 e 30)
    - i nzate

non sempre è possibile l'applicazione contemporanea di tutti i principi discussi nel seguito. E' compito del progettista valutare le differenti soluzioni e scegliere cosa prediligere

GRASP

- principali GRASP considerati :
  - creator
  - information expert
  - low coupling
  - high coesion
  - controller
  - principio conoscenza minima (legge di Demetra)

creator

- Problema
  - chi crea/istanzia un oggetto relativo ad una classe A?
- Soluzione
  - assegna alla classe B la responsabilità di creare un'istanza della classe A se una delle seguenti condizioni è vera
    - B compone/aggrega un insieme di oggetti di tipo A
    - B registra A presso un oggetto che ha la responsabilità di “contenitore” di istanze
    - B utilizza “in modo esclusivo” A, cioè solo istanze di tipo B sono responsabile di interrogare e/o modificare istanze di tipo A
    - B possiede tutti i dati per poter inizializzare A
  - la scelta tra soluzioni alternative dovrebbe essere fatta preferendo quelle soluzioni che massimizzano il numero di condizioni vere

information expert

- Problema
  - qual è il principio base per assegnare responsabilità ad oggetti?
- Soluzione
  - assegna una responsabilità alla classe che possiede le informazioni necessarie per soddisfarle
  - capire il ruolo che una determinata classe sta svolgendo nel sistema e quali informazioni è in grado di recuperare/mantenere
    - definire degli “esperti di informazione” aiuta a trasformare associazioni in aggregazioni o composizioni, ed a dettagliare l'insieme di operazioni fornite da una classe/interfaccia

low coupling – 1

- accoppiamento: misura del grado di dipendenza esistente tra classi distinte di uno stesso sistema software
  - stima quanto fortemente l'implementazione di una classe è connessa ad altre, quanto ne conosce, o quanto dipende da esse
  - vedi lezioni 6, 9, 10, 17

- Problema
  - come ridurre l'impatto dei cambiamenti?
- Soluzione
  - assegna responsabilità in modo tale che l'accoppiamento (quello non necessario!!!!) rimanga basso
    - alto grado di accoppiamento → soluzione fortemente dipendente da scelte implementative
  - usa questo principio per valutare soluzioni alternative

low coupling – 2

- Note
  - information expert “sostiene” low coupling
    - “sostiene” = “condizione necessaria ma non sufficiente”
  - le forme più comuni di coupling tra `TypeX` e `TypeY`
    - `TypeX` ha un attributo `TypeY` o è associato a `TypeY`
    - `TypeX` richiama servizi di un oggetto `TypeY`
    - `TypeX` ha una operazione il cui metodo associato include il riferimento ad una istanza di `TypeY` (e.g. variabili locali, parametri o tipi di ritorno nei metodi invocati, etc)
    - `TypeX` è sottoclasse (diretta o indiretta) di `TypeY`
    - `TypeY` è un'interfaccia, che `TypeX` implementa

high coesion – 1

- coesione : misura del grado di correlazione tra le operazioni di un elemento software dal punto di vista funzionale
  - stima quanto fortemente
    - la definizione di una classe sia concettualmente affine ad altre
    - le operazioni fornite dalla classe siano consistenti con le responsabilità associate alla classe
  - implica una misura del “carico di lavoro” potenziale (i.e. responsabilità) assegnato ad una classe
  - vedi lezioni 6, 9, 10, 17

- Problema
  - come mantenere oggetti focalizzati, comprensibili, gestibili e che implichino “low coupling”?
- Soluzione
  - assegna responsabilità in modo tale che la coesione rimanga alta
    - alto grado di coesione → soluzione ben strutturata nel dominio del problema da risolvere
  - usare classi che definiscono un insieme (relativamente) limitato di operazioni, riferenti ad operazioni altamente correlate
  - usa questo principio per valutare scelte alternative

high coesion – 2

- esempi di gradi di coesione funzionale
  - coesione bassa
    - una classe è la sola responsabile di molte cose in aree diverse
    - una classe ha da sola la responsabilità di un compito complesso in una sola area funzionale
  - coesione alta
    - una classe ha responsabilità moderate in un'unica area funzionale e collabora con altre classi per svolgere i suoi compiti
  - coesione moderata
    - una classe A ha da sola la responsabilità di leggere in un insieme limitato e scorrelato di aree funzionali diverse che sono logicamente legate al concetto modellato da A

controller

- Problema
  - qual è il primo oggetto oltre lo strato UI che riceve e coordina una operazione nel sistema?
- Soluzione
  - assegna la responsabilità a un oggetto che rappresenta
    - il punto di accesso al “sistema complessivo”
      - applicazione del GoF Façade (vedi lezione 30)
    - uno scenario di un caso d'uso all'interno del quale si verifica l'operazione di un sistema (i.e. controller di sessioni)

principio della conoscenza minima – 1

- è una linea guida di progettazione più che di analisi
  - spesso riferito come Legge di Demetra
- “una classe dovrebbe conoscere solo ciò che le è strettamente necessario”
  - ogni unità di programma dovrebbe conoscere solo poche altre unità di programma strettamente correlate;
  - ogni unità di programma dovrebbe interagire solo con le unità che conosce direttamente.

principio della conoscenza minima – 2

```
public class Car {

  private Driver d;

  private Vector<Passengers> p;

  public Car(Context context){
    this.d = context.driver;
    this.p = context.getPassengers();

  }

}
```

principio della conoscenza minima – 2

```
public class Car {

  private Driver d;

  private Vector<Passengers> p;

  public Car(Context context){
    this.d = context.driver;
    this.p = context.getPassengers();

  }

}
```

- questo esempio viola la legge di Demetra
  - `Car` deve esplicitamente sapere che `Context` ha :
    - un attributo `driver`
    - un metodo `getPassengers`
  - questa soluzione impatta sul
    - coupling delle classi del sistema
    - principio di information hiding
  - i cambiamenti di `Context` incidono direttamente su `Car`, anche se le due classi apparentemente dovrebbero essere scorrelate

principio della conoscenza minima – 3

```
public class Car {

  private Driver d;

  private Vector<Passengers> p;

  public Car(Context context){
    this.d = context.driver;
    this.p = context.getPassengers();

  }

}
```

- soluzione appropriata che rispetta la legge di Demetra

```
public class Car {

  private Driver d;

  private Vector<Passengers> p;

  public Car(Driver driver){
    this(driver, null);

  }

  public Car(Driver driver, Vector<Passengers> listOfPassengers){
    this.d = driver;
    this.p = listOfPassengers;

  }

}
```

in conclusione RDD

- un modo di pensare alla progettazione O.O.
  - **R**esponsibility **D**riven **D**esign
- gli oggetti software sono definiti in funzione di una loro responsabilità
  - assegnata dal progettista
  - obblighi sulla struttura o sul comportamento
    - responsabilità di fare
    - responsabilità di conoscere
    - responsabilità di collaborare

bibliografia di riferimento

- “Design Patterns – Elementi per il riuso di software a oggetti”, Gamma, Helm, Johnson, Vlissides (GoF). Addison-Wesley (1995).
  - Design Patterns: Elements of Reusable Object-Oriented Software
- “Applicare UML e i Pattern – Analisi e progettazione orientata agli oggetti”, Larman. 3za Edizione. Pearson (2005).
  - “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development”