Progettazione

Progettazione ad oggetti

- Gli oggetti sono astrazioni di entità reali o di sistema
  - In alcuni casi, ci potrebbe essere una corrispondenza ovvia tra entità del mondo reale e oggetti
- Gli oggetti sono indipendenti e incapsulano il loro stato
  - Le funzionalità del sistema sono espresse attraverso le operazioni offerte dagli oggetti
  - Le aree dati condivise sono limitate e gli oggetti comunicano attraverso messaggi o invocazioni di metodo
  - Gli oggetti possono essere distribuiti e possono essere eseguiti in sequenza o in parallelo
- Gli oggetti sono potenzialmente componenti riusabili

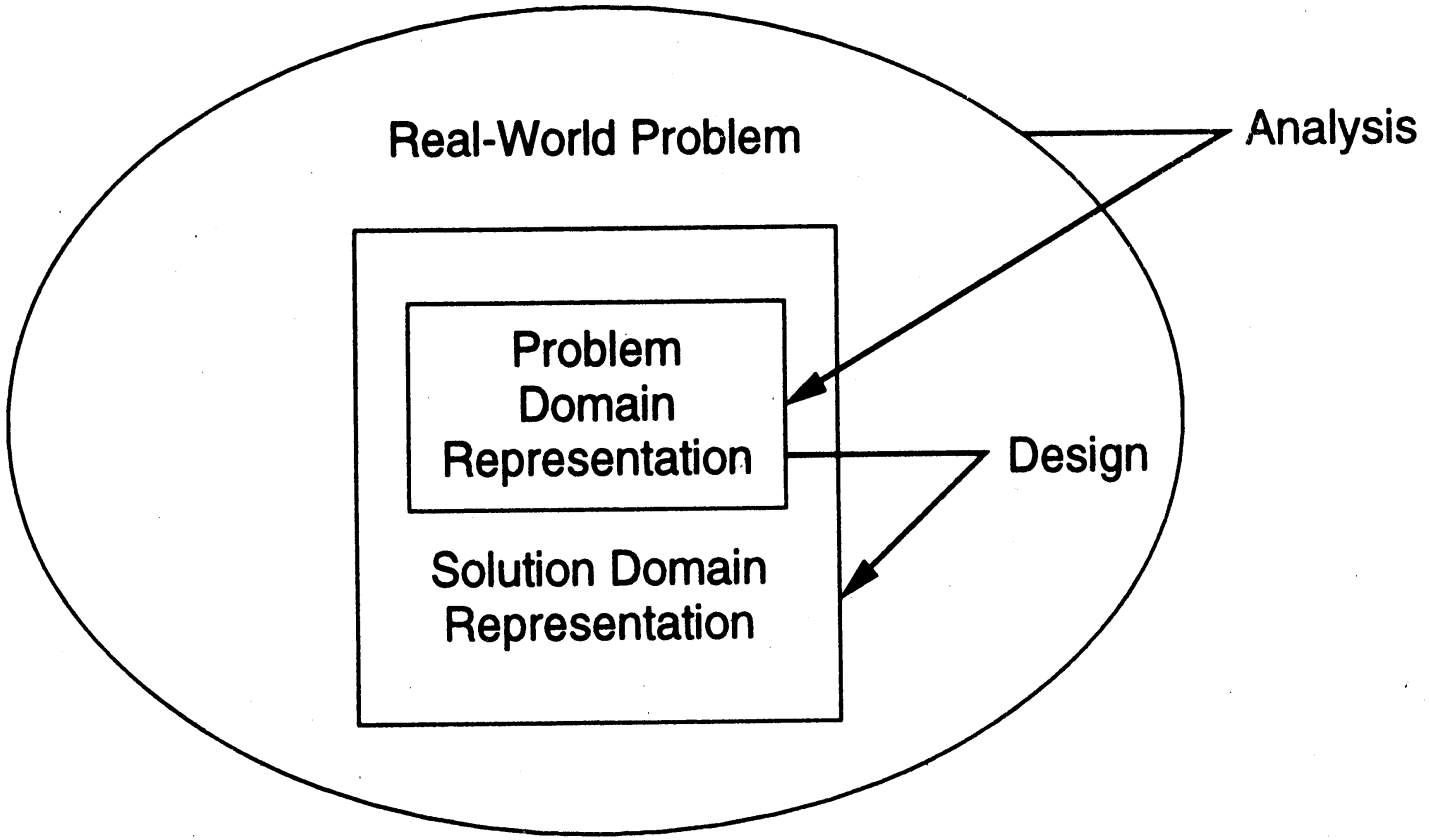
Come identificare gli oggetti

- L'identificazione degli oggetti (classi) è la parte più difficile della progettazione a oggetti
- Non esiste una "formula magica"
  - Ci si basa su euristiche, abilità, **esperienza** e conoscenza del **dominio applicativo**
- L'identificazione degli oggetti/classi è un processo iterativo
  - Solitamente non si finisce con una sola iterazione
  - Distinzione tra oggetti/classi del dominio e della soluzione

Approcci possibili

- Approccio grammaticale basato su una descrizione del sistema in linguaggio naturale
- Identificazione degli “elementi tangibili” del dominio applicativo
- Approccio “comportamentale” per identificare gli oggetti in funzione di chi partecipa a quale comportamento
- Approccio basato su scenari per identificare gli oggetti, attributi e metodi in ogni scenario
- Modellazione dei dati (schema ER) completata con le operazioni
- Altri metodi …

Analisi e progettazione



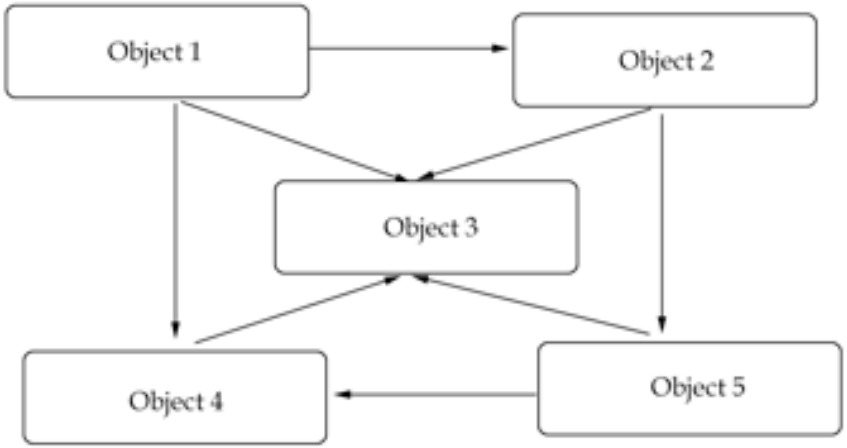
Progettazione

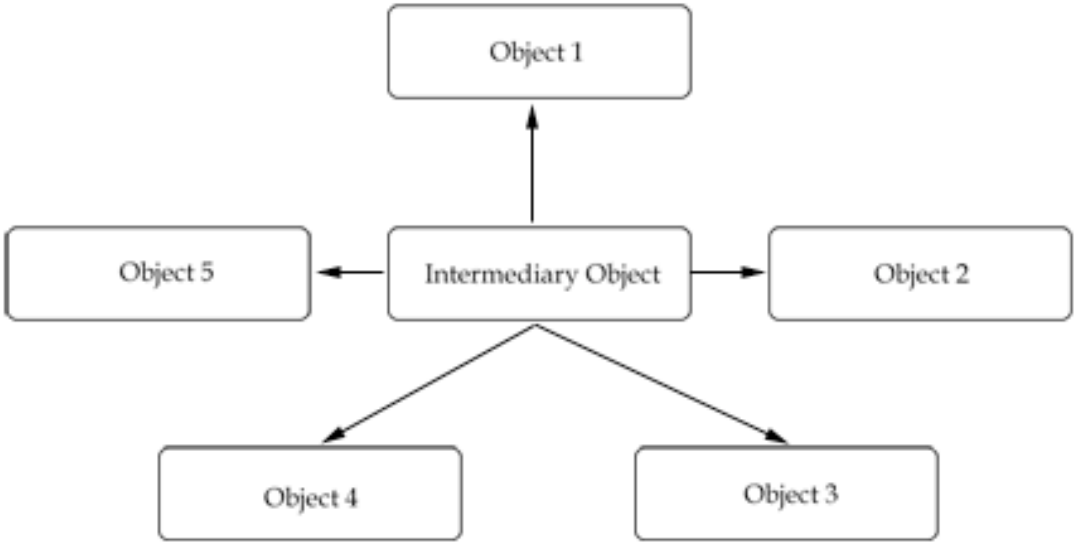
- La “correttezza” del progetto è fondamentale
  - …ma possiamo anche valutarlo in funzione della sua efficienza, modificabilità, stabilità, …
- Possiamo valutare un progetto usando
  - **Accoppiamento**
  - **Coesione**
  - Principio **open-closed**
- Modulo e classe saranno sinonimi

Accoppiamento (coupling)

- Cattura il grado di interconnessione tra moduli
- Un alto grado di accoppiamento significa
  - Alta interdipendenza tra i moduli
  - Difficoltà di modifica del singolo modulo
- Un **basso accoppiamento** è requisito fondamentale per creare un sistema comprensibile e modificabile

Esempio


Object 1
Object 2
Object 3
Object 4
Object 5


Object 1
Object 5
Intermediary Object
Object 2
Object 4
Object 3

Coesione

- Concetto intra-modulo
- Si concentra sul perché gli elementi stanno nello stesso modulo
  - Solo gli elementi fortemente correlati dovrebbero stare nello stesso modulo
  - Il modulo rappresenterebbe una chiara astrazione e sarebbe più semplice da capire
- Alta coesione solitamente porta a basso accoppiamento
- Tre tipi di coesione: **method**, **class**, e **inheritance**

Sintomi di un progetto “mal fatto”

- **Rigidità** è la tendenza del software ad essere difficile da cambiare, anche in modo semplice
  - Ogni cambiamento provoca una cascata di cambiamenti in sequenza nei moduli dipendenti
- **Fragilità** è la tendenza del software a “rompersi” in molti punti ogni volta che viene cambiato
  - Spesso il problema si verifica in parti del programma non logicamente correlate al cambiamento
- **Immobilità** è l’incapacità a riusare software da altri progetti o da altre parti dello stesso progetto
- **Viscosità** si verifica quando l’uso di scorciatoie (hack) è più facile dell’uso dei metodi che rispettano il progetto
  - È facile fare la cosa sbagliata, difficile fare quella giusta

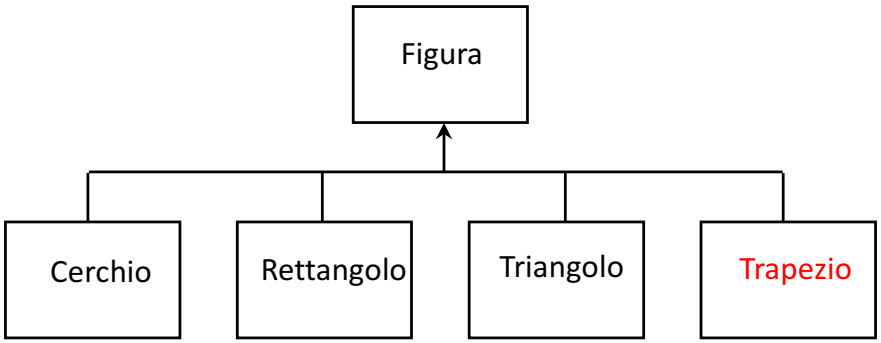
Gestione delle dipendenze

- I quattro sintomi precedenti sono causati, direttamente o indirettamente, da dipendenze improprie tra i moduli
  - Quando si ha un degrado nella struttura (architettura) del software, si ha anche un degrado della sua manutenibilità
- Le dipendenze tra moduli vanno gestite attraverso appositi "**firewall**"
  - Le dipendenze non devono propagarsi oltre i firewall

Principio Open-Closed

- Un modulo dovrebbe essere **aperto alle estensioni**, ma **chiuso alle modifiche**
- Derivato dal lavoro di Bertrand Meyer, in sostanza, dice
  - Dovremmo sempre scrivere le nostre classi in modo che siano estendibili, senza che debbano essere modificate
- Vogliamo essere capaci di cambiare il comportamento delle classi senza cambiarne il codice sorgente
- Il vantaggio in manutenzione è ovvio: se non modifico, non faccio errori
- Principi base: ereditarietà, overriding, polimorfismo e binding dinamico

Esempio ben noto



typedef struct …Figura;

String figure[100];

figure[1] = “rettangolo”;
figure[2] = “triangolo”;
figure[3] = “cerchio”;
figure[4] = “trapezio”;

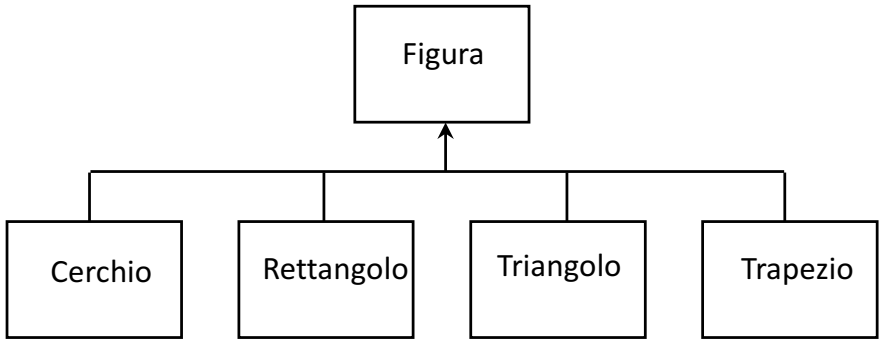
```
void disegnaTutto(Figura figure[]) {
   for (i= 0;  i<100; i++) {
      if (figure[i] è  “rettangolo”) “disegna rettangolo”
      if (figure[i] è “triangolo”) “disegna triangolo”
      if (figure[i] è  “cerchio”) “disegna cerchio”
      if (figure[i] è  “trapezio”) “disegna trapezio”
   }
}
```

Il codice già definito deve cambiare per supportare il nuovo tipo!

Open/closed?

- La progettazione tradizionale non rispetta il principio open-closed
  - Il modulo che implementa disegnaTutto è aperto a estensioni, ma solo tramite modifica: non è chiuso rispetto alle modifiche
- Come si può fare a rispettare open/closed?

Aggiungiamo la classe Trapezio




```
Figura[] figure = new Figura[100];

figure[1] = new Rettangolo();
figure[2] = new Triangolo();
figure[3] = new Cerchio();
figure[4] = new Trapezio();

disegnaTutto(figure);
```

```
public static void disegnaTutto(Figura[] figure) {
    for (i= 0;  i<100;i++)
        figure[i].disegna();
}
```

Tramite polimorfismo e binding dinamico, il codice non deve cambiare all'aggiunta di un nuovo tipo!

Estendibile se….

- Estendibilità tramite ereditarietà è garantita solo se possiamo usare oggetti di una sottoclasse che sono “sostituibili” a quelli della sopraclasse
- Infatti gli oggetti della classe devono rispettare il “contratto” della superclasse
  - Ad esempio, la disegna() deve contenere codice che disegna l’oggetto this sullo schermo
  - Ogni sottoclasse di Figura deve definire una disegna() che abbia lo stesso effetto!
  - Ma se si definisse disegna() in modo diverso?
    - Ad esempio, se disegna() di Trapezio fosse definita in modo che disegna this solo se l’area è maggiore di 20? Va ancora bene?

Liskov Substitution Principle

- Gli oggetti della sottoclasse devono **rispettare il contratto** della superclasse
  - Significa che il comportamento della sottoclasse deve essere “compatibile” con la specifica della sopraclasse
  - Ma il metodo può essere esteso per coprire ulteriori casi… senza cambiare i casi che la sopraclasse richiedeva di coprire!
- Moduli che usano oggetti di un tipo devono potere usare oggetti di un tipo derivato senza accorgersi della differenza
  - Il contratto della superclasse deve essere onorato da tutte le sottoclassi

Esempi

- Quadrato e Rettangolo
- Ortaggio e OrtaggioStagionale
- Persona e Studente

Dependency Inversion Principle

- Dipendere dalle astrazioni, non dagli elementi concreti
- In Java, vuol dire
  - Dipendere da interfacce e classi astratte, non da metodi e classi concrete
  - Principio fondante del concetto di progettazione per componenti (component design)
    - CORBA, EJB, …

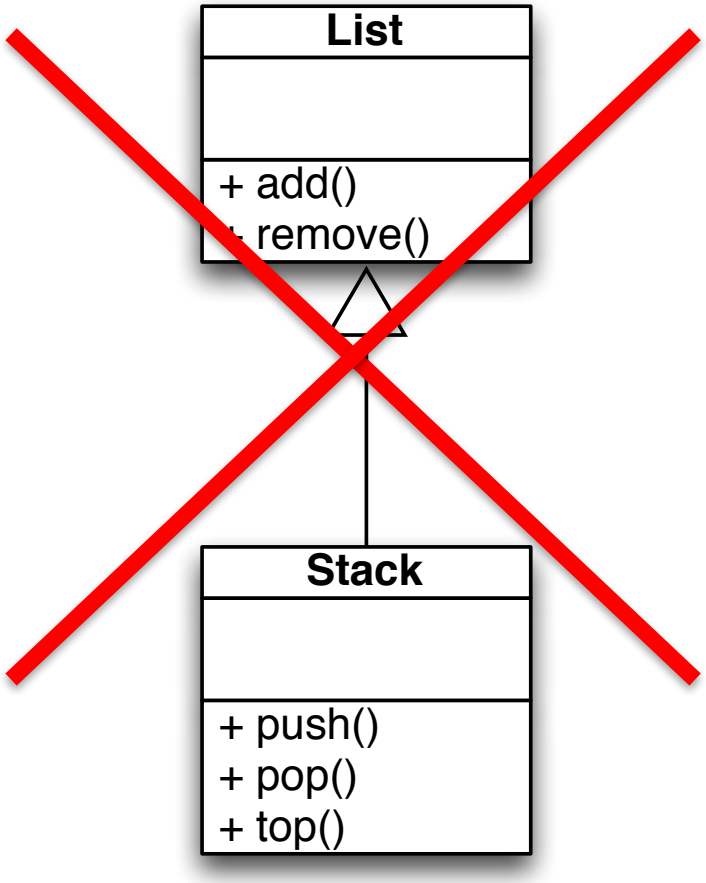
… in altre parole

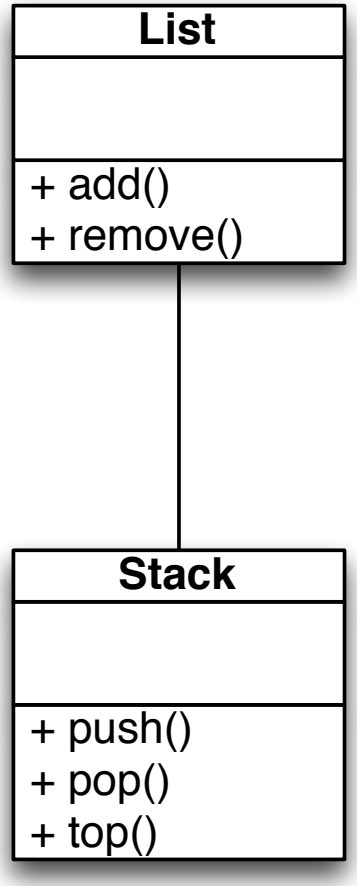
- Ogni classe C che si ritiene possa essere estesa in futuro, dovrebbe essere definite come sottotipo di un'interfaccia o di una classe astratta A
- Tutte le volte che non è strettamente indispensabile riferirsi ad oggetti della classe concreta C, è meglio riferirsi invece a oggetti il cui tipo statico è A, non C
- In questo modo, sarà più facile in seguito modificare il codice che utilizza le funzioni per utilizzare invece di C altre classi sottotipi di A

Minimizzazione dell'interfaccia

- Se non esiste una ragione forte per dire che un metodo è pubblico, lo dichiariamo privato
- Definiamo solo metodo getter (e non setter) per i campi della classe se possibile
- Non dobbiamo replicare completamente la struttura dati contenuta in una classe nella sua interfaccia
  - Cambiare la struttura dati facilmente porterà a dover cambiare anche l'interfaccia
  - L'Information hiding è di fatto vanificato…

Ereditarietà e delega





Ulteriori principi

- The **Interface Segregation Principle**: many client-specific interfaces are better than one general purpose interface
- The **Reuse/Release Equivalency Principle**: the granule of reuse is the same as the granule of release; only components that are released through a tracking system can be effectively reused
- The **Acyclic Dependencies Principle**: the dependency structure for released components must be a directed acyclic graph; there can be no cycles
- The **Stable Dependencies Principle**: dependencies between released categories must run in the direction of stability; the dependee must be more stable than the depender
- The **Stable Abstractions Principle**: the more stable a class category is, the more it must consist of abstract classes; a completely stable category should consist of nothing but abstract classes

Anti-pattern

- Un anti-pattern è una soluzione usata spesso, ma che dovrebbe essere evitata
  - È un errore comune
  - Dovrebbe fornire suggerimenti su come migliorare il codice o evitare errori noti
- Gli anti-pattern non si applicano solo alla progettazione a oggetti

Esempio

- Classe Blob
  - Una sola classe enorme che contiene la maggior parte della logica applicativa
  - Deriva spesso dall'adozione di una progettazione procedurale

Refactoring

- È abbastanza difficile fare la cosa giusta al primo colpo
  - Spesso le soluzioni trovate richiedono **refactoring**
  - Ovvero il miglioramento del progetto del codice esistente **senza cambiarne il comportamento**
- Miglioramenti al codice/architettura
  - Piccoli miglioramenti
  - Test di regressione continuo
    - Framework Junit

Refactoring

- Il tempo e la manutenzione potrebbero rendere disordinato il codice
- Codice disordinato riduce le possibilità di manutenzione
- Il refactoring rinfresca/pulisce il codice senza cambiarne le funzionalità
  - Migliora la coesione e riduce l'accoppiamento
  - Applica i pattern
  - Rimuove gli anti-pattern
  - Ottimizza le prestazioni (velocità e memoria)
  - Aggiunge commenti (magari attraverso JavaDocs)

Quando fare refactoring

- Non esiste regola precisa
- Quando si intravedono problemi
  - Parti di codice particolarmente difficili
  - Parti troppo complesse
  - Parti con anti-pattern evidenti
- Meglio fare refactoring prima di aggiungere nuove funzionalità o cambiare le esistenti

Problemi noti (I)

- **Codice duplicato**
  - Estrarlo, parametrizzarlo e farlo diventare un metodo di servizio
  - Codice simile in sottoclassi correlate
    - Spostare il codice comune nella superclasse
- **Metodi lunghi**
  - Se il codice di un metodo diventa troppo lungo per poterlo capire facilmente, bisogna estrarne delle parti come metodi di servizio

Metodi di piccole dimensioni

- Metodi di dimensioni limitate possono essere letti e compresi in modo abbastanza agevole
  - Metodi brevi e chiari richiedono pochi commenti o possono non richiederne affatto
  - Di solito si riesce a mantenere la maggior parte dei metodi al di sotto delle 20 righe di codice
  - Raramente è necessario scrivere metodi più lunghi di 40 righe

Problemi noti (II)

- **Funzionalità e dati**
  - Se un metodo usa principalmente i dati di un'altra classe, bisogna spostare il metodo nella classe in cui sono definiti i dati
- Uso eccessivo di **switch**
  - Spesso indicano un tipo
  - L'introduzione di nuovi tipi richiede la modifica degli switch relativi
  - Bisognerebbe usare ereditarità, enum, pattern

Problemi noti (III)

- **Blocchi di dati**
  - Dati che sono solitamente usati insieme
  - Potrebbero portare a una lunga lista di parametri (altro problema)
  - Bisognerebbe introdurre una classe per contenere i dati correlati
- **Commenti**
  - Parti di codice con commenti significativi potrebbero diventare metodi con un nome auto-esplicativo