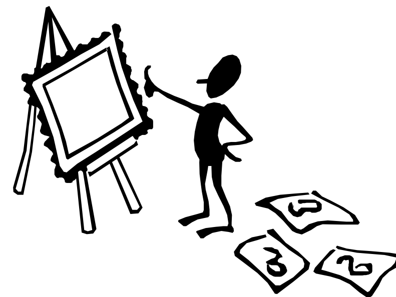
Introduzione all'Analisi e Design ad oggetti

Luca Lista

Il ciclo di vita del software

- Requirements
- Analysis
- Design
- Production
- Testing
- Maintenance



Il modello *Waterfall*

Requirements

Analysis

Design

Production

Testing

Il modello *Evolutionary*

Requirements

Analysis

Design

Production

Testing

Waterfall *vs* Evolutionary

Waterfall

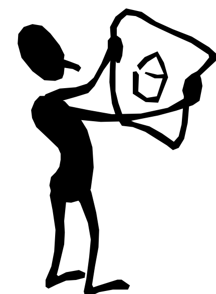
- Decomposizione completa del sistema dall'inizio
- Processo per singoli passi
- Integrazione alla fine

Un prodotto completo è disponibile solo alla fine

Evolutionary

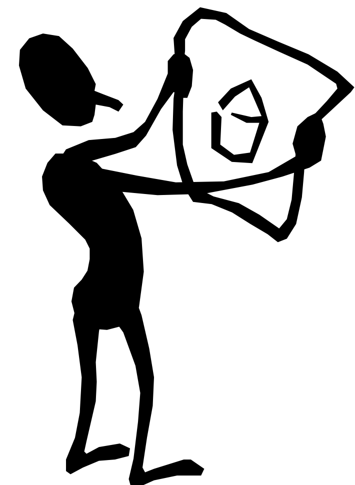
- Brevi cicli di sviluppo completi
- Scelte basate sulla valutazione dei rischi
- Integrazione continua

Un prototipo funzionante è disponibile sin dall'inizio



Analysis

- Comprensione e razionalizzazione dei requisiti
  - la formulazione dei requisiti è spesso incompleta, ambigua o imprecisa
  - chiarifica dei requisiti e verifica della consistenza
- Costruzione del modello
- Identificazione delle classi degli oggetti rilevanti per l’applicazione
  - Evitare riferimenti a strutture informatiche, come *array*, *liste*, etc. che saranno trattate più tardi
- Uso dell’astrazione
  - Rimandare i dettagli a una fase successiva
- Identificazione delle relazioni tra classi
- I nomi di classi, metodi e attributi sono importanti!



Come identificare le classi

- Dall’analisi testuale dei requisiti, le classi possono essere identificate con i più importanti sostantivi
- Possono specificare sia rappresentazioni di entità fisiche (impiegato, libro, ecc.) che concetti (superficie, traiettoria, pagamento, prestito, ecc.)
- Gli oggetti devono avere responsabilità chiare all’interno del modello
- Le responsabilità devono essere equamente distribuite tra le diverse classi

Errori da evitare

- Eliminare classi ridondanti
  - duplicati di classi già definite (stesse funzionalità: investitore, compratore, venditore), classi irrilevanti (scarse funzionalità: peso, altezza, prezzo), classi di sola implementazione (elenco libri, ecc.)
- Evitare classi con troppe responsabilità
  - Meglio spezzarle in classi più piccole, con ruoli più chiari e definiti
- Evitare classi che siano solo contenitori di dati
  - Esempio: solo metodi **`getX()`**, **`setX()`**: equivale ad aver dichiarato **`X`** attributo pubblico. La responsabilità sull'uso di **`X`** ce l'ha qualche altro oggetto!

Identificazione dei metodi e attributi

- I primi candidati metodi possono essere estratto dai principali verbi nella descrizione testuale dei requisiti
- Gli attributi rappresentano le principali proprietà degli oggetti
  - Utilizzare nuovi oggetti aggregati per le proprietà complesse che meritano un’identità indipendente (es.: l’auto di un autista, il conto corrente bancario di un risparmiatore)
  - Rimandare la descrizione degli attributi necessari per dettagli di implementazione
- Cercare operazioni comuni a più classi e verificare se sia possibile creare una classe astratta

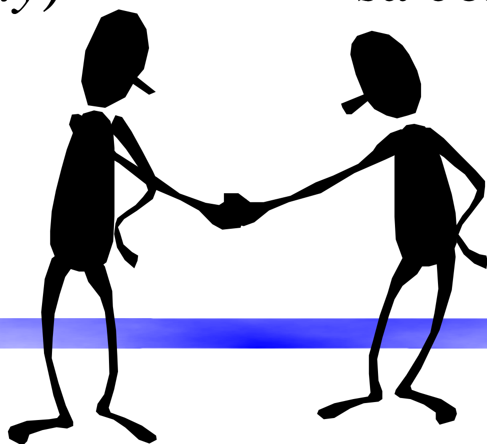
Rapporto tra *client* e *server*

Vista del *client*

- Il *client* sa di cosa ha bisogno, e che vuole richiedere al server
- Non gli interessa come il server fa il lavoro
- Non gli interessa se il lavoro lo fai il server o se lo delega (es.: *proxy*)

Vista del *server*

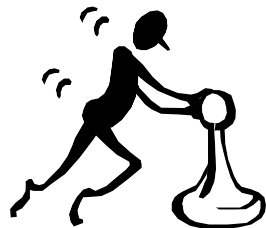
- Il server sa quale lavoro deve svolgere a richiesta
- Non gli interessa chi effettua la richiesta
- Se come svolgere il lavoro o a chi chiedere per svolgere le parti che non sa completare

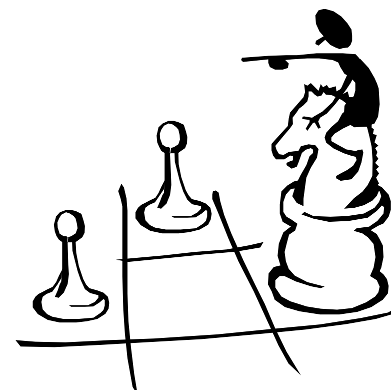


Assegnare Responsabilità

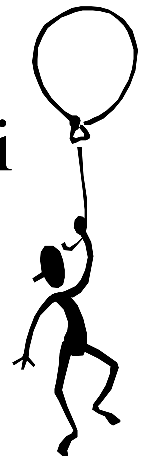
- Analizzare il ruolo dei vari oggetti
- Concentrarsi sul comportamento e non sulla rappresentazione
- Definire le interfacce (le operazioni che soddisfano le responsabilità) prima

Una corretta assegnazione delle responsabilità è la chiave di una buona modularità e riuso





Collaborazione tra classi



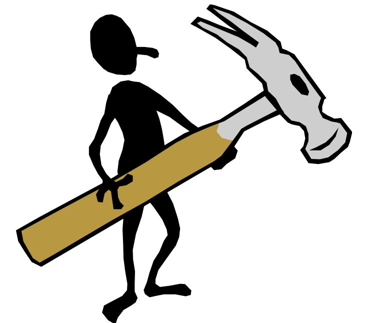
- Le responsabilità vanno suddivise tra i vari oggetti del sistema
- Non deve esistere un controllo centralizzato
  - Un oggetto che svolge tutte le operazioni e usa oggetti che siano solo contenitori di dati equivale a un programmare in termini di dati e funzioni!

- Un oggetto deve compiere le proprie responsabilità e delegare ad altri operazioni specifiche
  - Legge di Demeter: non usate oggetti lontani:

Invece di: `stockMarket.stocklist().add(stock);`

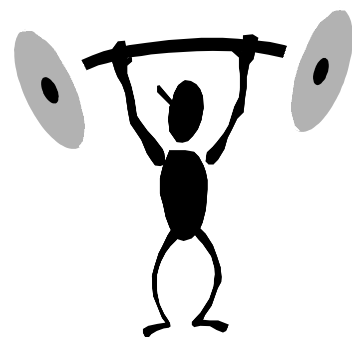
usare: `stockMarket.add(stock);`

Identificazione delle responsabilità



- Ogni classe deve tentate di rifiutare le responsabilità
  - Dovrei? (Non sono io che lo devo fare!)
  - Potrei? (Non ho le caratteristiche per farlo!)

- Cercate di fare poco lavoro
  - Se una classe ha dovuto accettare una responsabilità può cercare di far fare il lavoro a qualche altro oggetto



- Potenziate i collaboratori, non interferite

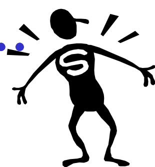
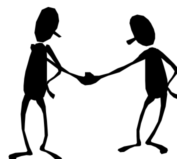
Identificazione delle relazioni

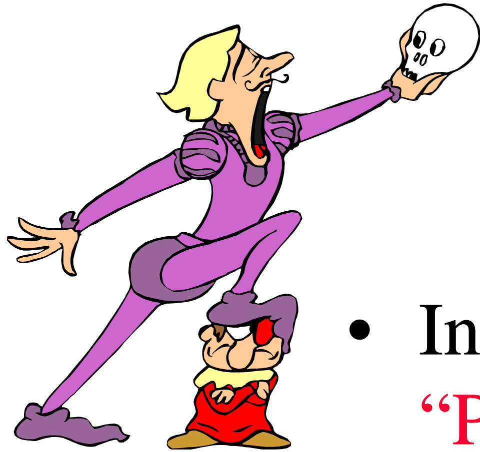
- Cercare collaborazioni
- Cercare aggregazioni
- Cercare generalizzazioni

Come un *client* conosce il suo *service provider*?

- Evitare le relazioni inutili e di eccedere nelle dipendenze
  - Ragnatele di dipendenze rendono rigido il codice!

Scelta delle possibili relazioni

Logiche		Implementazione
• Generalizzazione: E' un..		• Ereditarietà • Istanziazione di *template*
• Aggregazione: Ha		• Composizione by value
• Dipendenza: Conosce		• Composizione by reference



Avere o *Essere*

- In C++, l'ereditarietà pubblica segue il "Principio di Liskov":

sottotipi possono essere usati ogni volta che è richiesta la classe di base.

- Usare ereditarietà se è richiesto il polimorfismo
- Mai modificare il comportamento della classe di base per motivi indotti dalle sottoclassi concrete.

Ereditarietà e ri-uso del codice

- Attenzione! L'ereditarietà può essere usata per evitare di riscrivere codice che esiste già in altre classi.

  Questo non è OO ed è da evitare!

- L'aggregazione spesso risponde meglio a questa esigenza.
- Se il rapporto fra due oggetti è del tipo:
  - ***è*** allora si usa l'ereditarietà;
  - ***ha*** allora si usa l'aggregazione.

Avere o Essere

- `Investor`: è una `Strategy` o ha una `Strategy`
- `StockMarket`: è un `vector<Stock>` o ha un `vector<Stock>`

- `Auto`: ha 4 `Ruote` o è 4 volte una `Ruota` (impossiblie in C++, tipico in EIFFEL)

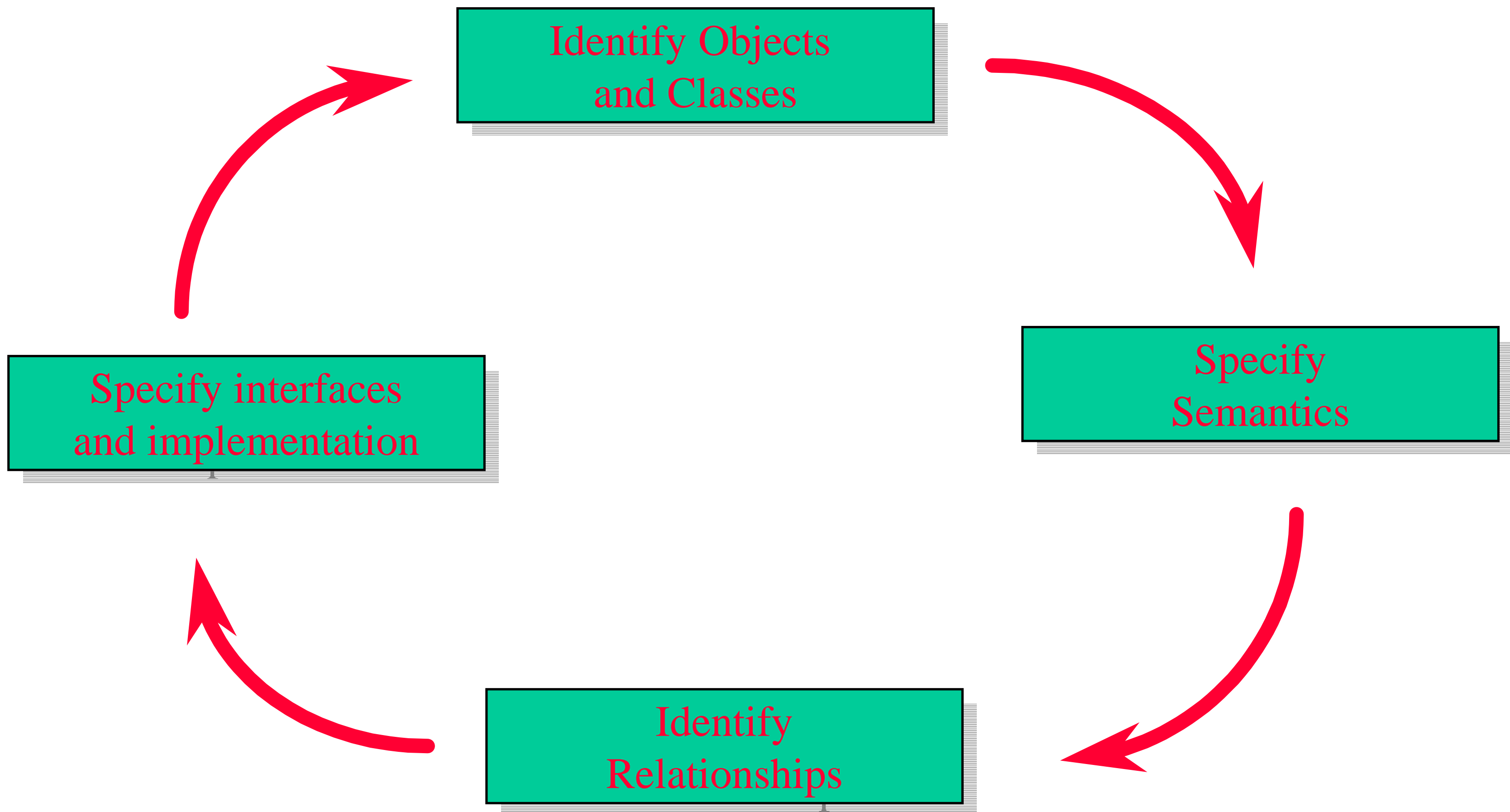
Composizione:
by value o *by reference*

- Tipi sempici (`int`, `double`, …): by value
- Parte dello stato di un oggetto:
  by value
- Un oggetto viene condiviso:
  by reference
- Allocato *run time*:
  by reference
- Usato polimorficamente:
  by reference

Design

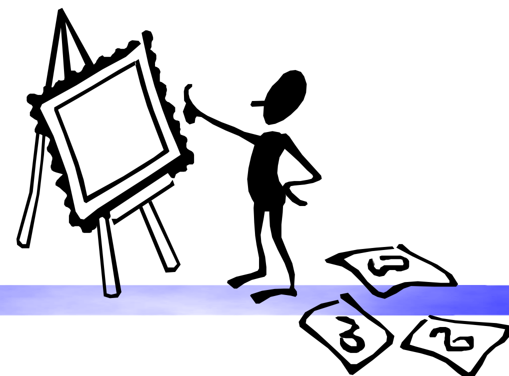
- Avvicinare il modello ad oggetti dell'analisi ai concetti informatici
- Aggiunta di nuovi oggetti interni necessari per l'implementazione
- Espressione delle operazioni identificate in fase di analisi in termini di algoritmi
- Suddivisione delle operazioni complesse in operazioni più semplici
  - da delegare ad altri oggetti
- Scelta delle strutture dati

Ciclo di Design



Iterazioni sul Design

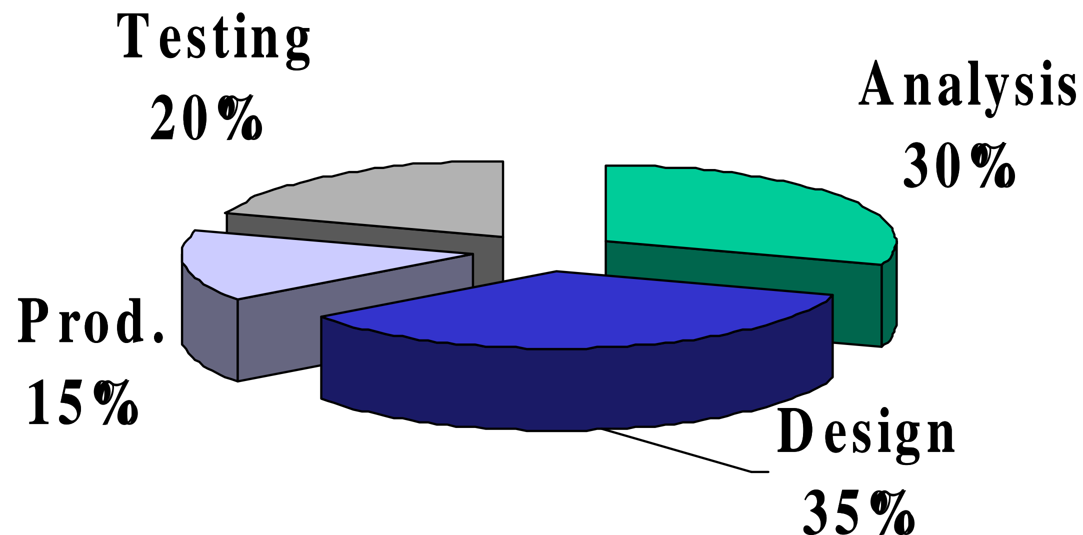
- Dopo ogni ciclo bisogna analizzare i rischi, la stabilità e la complessità delle classi
- Se una classe è troppo complessa conviene dividerla
- Ad ogni ciclo il numero di modifiche deve diminuire
- Architetture troppo complesse devono essere modularizzate



Produzione

- Codifica, implementazione del modello
- Non sopravvalutare questa fase: richiede minore sforzo se le fasi precedenti sono fatte bene

Suddivisione del tempo per il primo ciclo



Test

- Debugging: è ovvio… il codice non deve dare errori.
- Use cases: specificano il comportamento del sistema in una *regione*.
- Scenari: sono esempi concreti di *use cases*. Per definizione se tutti gli scenari sono soddisfatti correttamente il test è positivo.

Use Cases e Scenari

- Uno ***Use Case*** specifica alcuni dei comportamenti richiesti al sistema
- Uno ***Scenario*** è una realizzazione concreta di uno *use case* in una particolare circostanza
  - Scenari secondari possono essere usati per rappresentare una variazione di un tema di uno scenario principale
    (what if...)

Metodi di sviluppo del software

Un metodo comprende:

- Una notazione

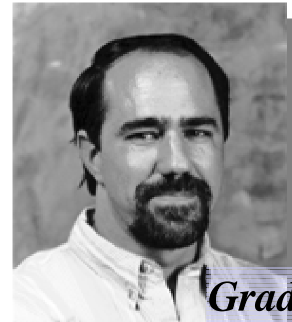
  mezzo comune per esprimere strategie e decisioni

- Un processo

  specifica come deve avvenire lo sviluppo

Metodi Object Oriented

- Booch Method
  by *Grady Booch*
- OMT
  by *Jim Rumbaugh*
- Objectory (Use Cases)
  by *Ivar Jacobson*
- CRC
  by *R.Wirfs-Brock*



Grady Booch

Jim Rumbaugh

Ivar Jacobson

- Di più recente introduzione: UML
  - uno standard OMG (Object Management Group), dal novembre 1997

Un esempio concreto

- Requisiti:
  - Realizzare una simulazione di un modello di mercato azionario. Diversi investitori si scambiano titoli su un mercato azionario. Gli investitori decidono se vendere o acquistare in base a proprie strategie.
  - Si vuole misurare
    - l’andamento dei titoli sul mercato
    - quali sono le strategie più redditizie

Possibili candidati classi

- Requisiti:
  - Realizzare una simulazione di un modello di mercato azionario. Diversi investitori si scambiano titoli su un mercato azionario. Gli investitori decidono se vendere o acquistare in base a proprie strategie.
  - Si vuole misurare
    - l'andamento dei titoli sul mercato
    - quali sono le strategie più redditizie

Identifichiamo classi e oggetti

- I primi candidati classi possono essere presi dal vocabolario del problema che stiamo analizzando. Esempio:
  - Investitore, Titolo, Ordine, Strategia

Analizziamo un paio di scenari

- Come decidere se vendere o acquistare un titolo
- Come gestire gli ordini sul mercato

Decidere se vendere o acquistare un titolo

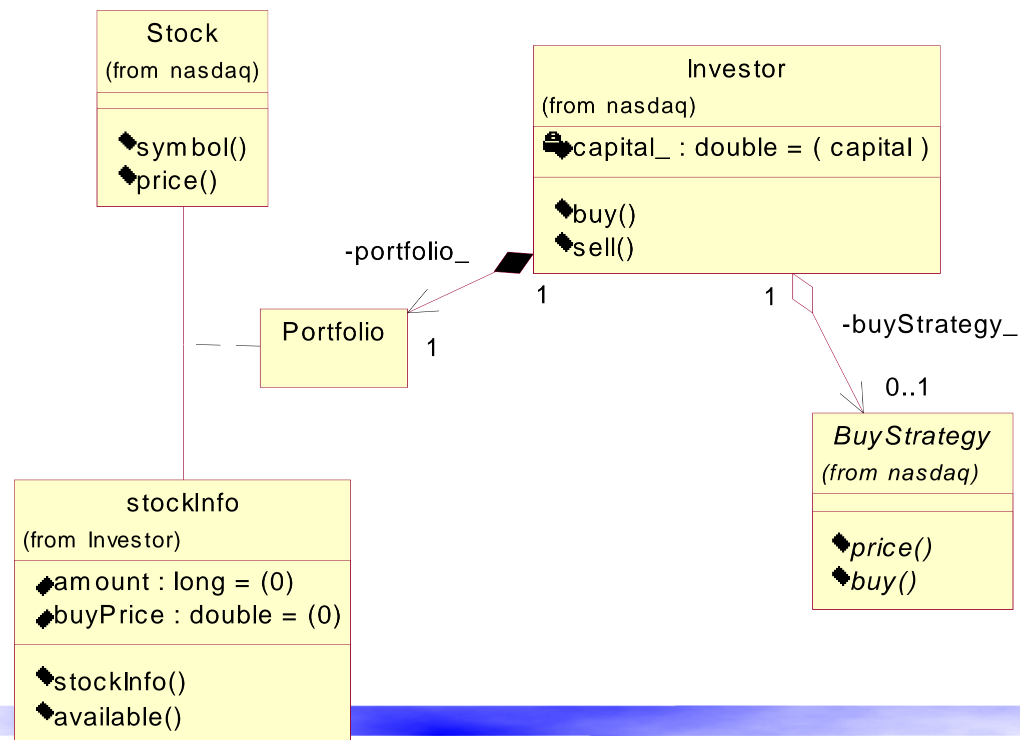
- Acquisire informazioni sul titolo
- Analizzare le prestazioni del titolo
  - Guadagno, perdita, storia….
- Analizzare i propri guadagni o le proprie perdite
- Decidere in base alla strategia
  - Quanto acquistare/vendere
  - A che prezzo piazzare l'ordine

Gestire gli ordini

- Controllare gli ordini relativi a ciascun titolo
- Verificare se ci sono ordini di vendita e di acquisto compatibili
  - L'offerta di vendita deve avere un prezzo minore o uguale a quello dell'offerta di acquisto
- Eseguire gli ordini
  - Possibilità di esecuzione parziale degli ordini
  - Gli ordini possono terminare ineseguiti al termine di una validità massima

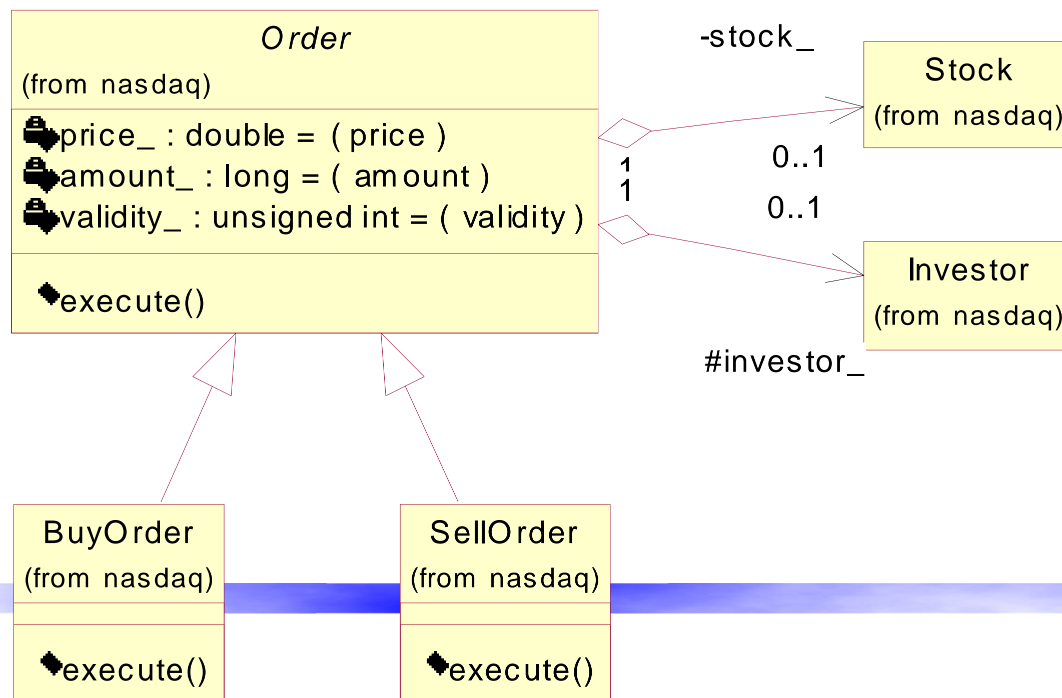
Identifichiamo alcune relazioni

- Un investitore
  - possiede una certa quantità di titoli
  - ha una certa strategia di acquisto



Relazioni della classe `Order`

- In ordine è relativo a un titolo
- Sa da chi investitore è stato immesso
- Può realizzare vendita o acquisto



Che significa acquistare o vendere

- Un investitore:
  - decide la quantità e il prezzo di azioni da vendere in base alla sua strategia
  - crea l'ordine relativo al titolo da acquistare/vendere
    - Si riserva di non spendere il capitale che può servire per eseguire un acquisto (altrimenti va in rosso…!)
    - Si riserva di non vendere azioni già impegnate in un ordine di vendita (venderebbe le stesse azioni due volte!)
  - Metterlo l'ordine sul mercato

Come viene emesso un ordine di acquisto

anInvestor : Investor

aStrategy : Buy Strategy

anOrder : Buy Order

nasdaq : Stock Market

buy(stock, capital, buyPrice, amountAvailable)

price( )

BuyOrder(this, stock, price, amount)

cost( )

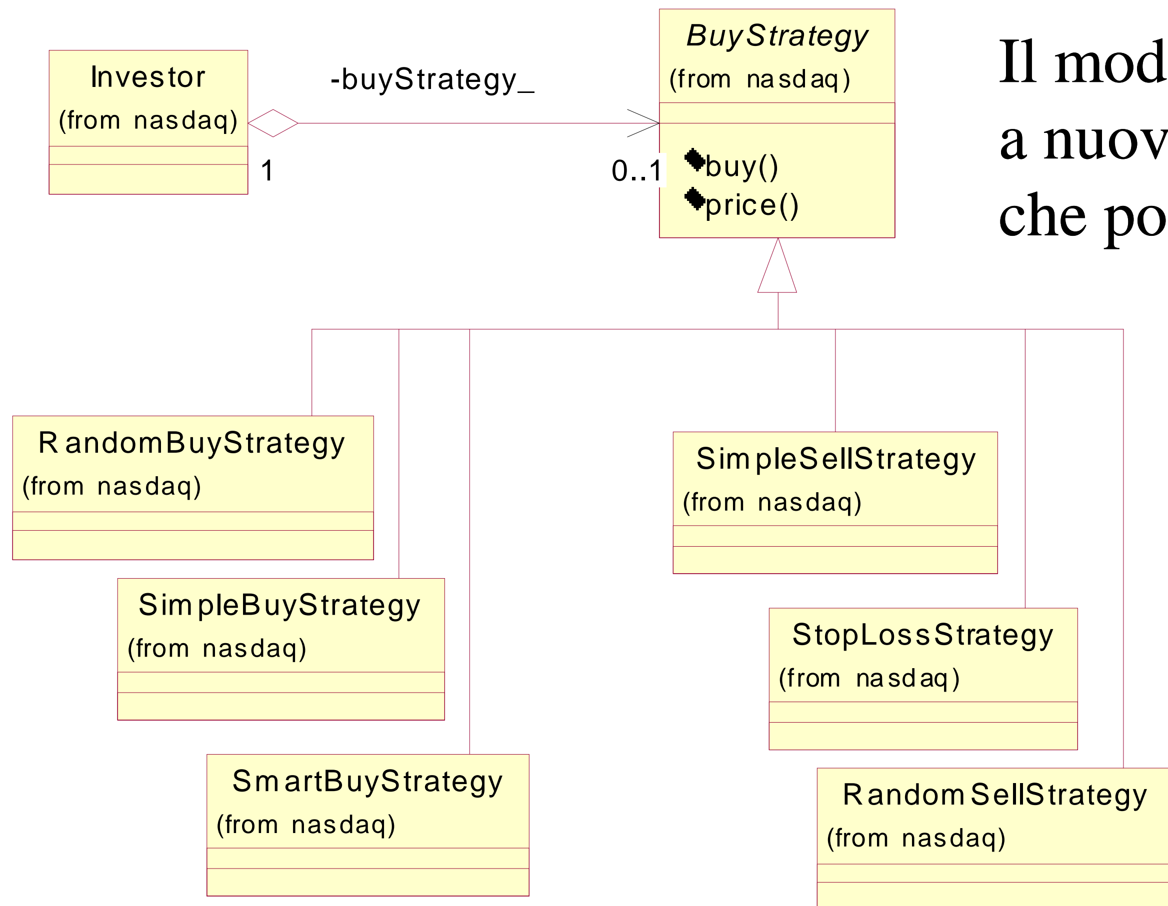
reserve(cost)

addBuy(anOrder)

Definiamo le strategie

- Strategie di acquisto:
  - Compro se il titolo sta salendo
    - Perché credo che continuerà a salire
  - Compro se il titolo è sceso
    - Perché credo che poi risalirà
  - Compro a caso
    - es.: simulo il caso in cui ho avuto notizie che mi hanno convinto
  - Compro altre azioni di un titolo che sta perdendo
    - Così medio il prezzo di acquisto, e più facilmente posso recuperare
- Strategie di vendita:
  - Vendo se ho guadagnato
    - Perché così realizzo il guadagno
  - Vendo per non perdere ulteriormente (Stop-loss)
    - Così non perdo ulteriormente
  - Vendo a caso
    - es.: simulo il caso in cui ho bisogno di soldi

Come sono organizzate le strategie



Il modello è estendibile
a nuove possibili strategie
che posso definire in seguito

Strategie multiple

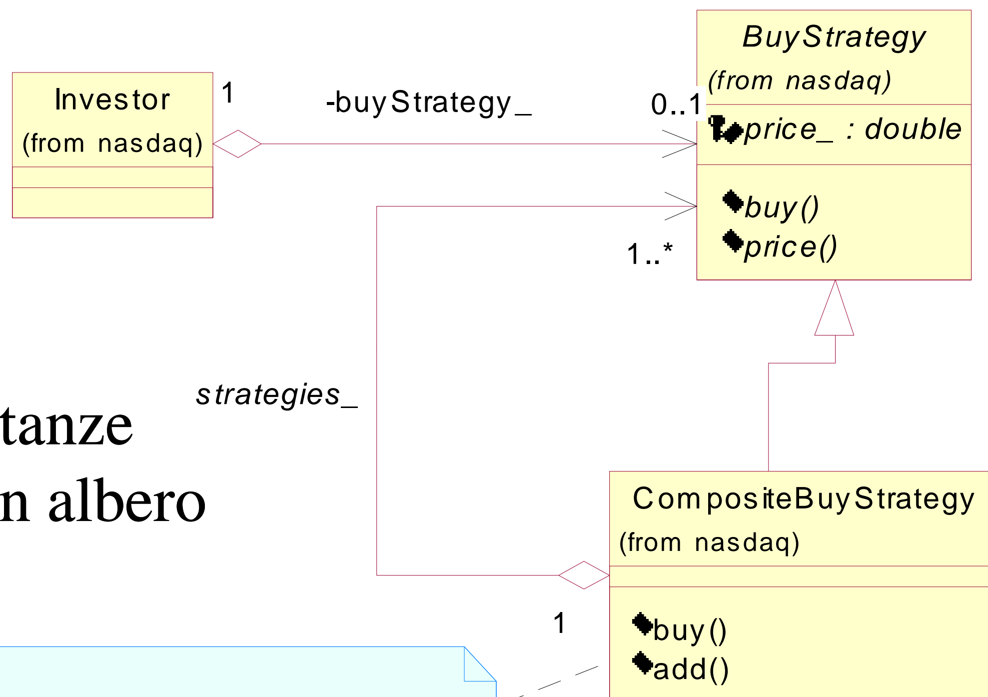
- Come gestire la possibilità di seguire più possibili strategie a seconda delle circostanze?

- Una strategia multipla è anch'essa una strategia
  - Astrazione!

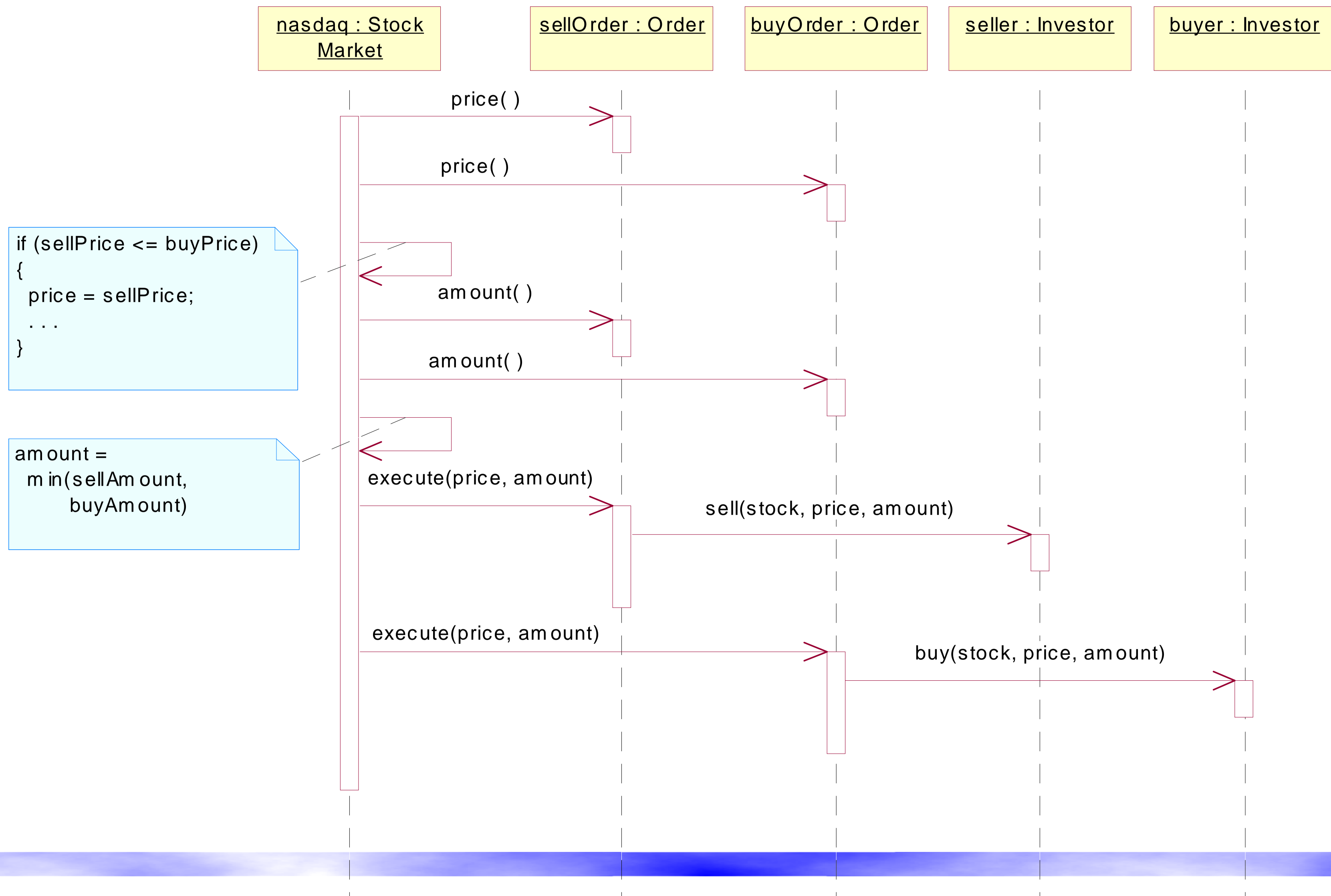
Strategie multiple

Questo modello di relazione si chiama *Composite Pattern.*

E' comune a molte circostanze nelle quale è necessario un albero di oggetti compositi



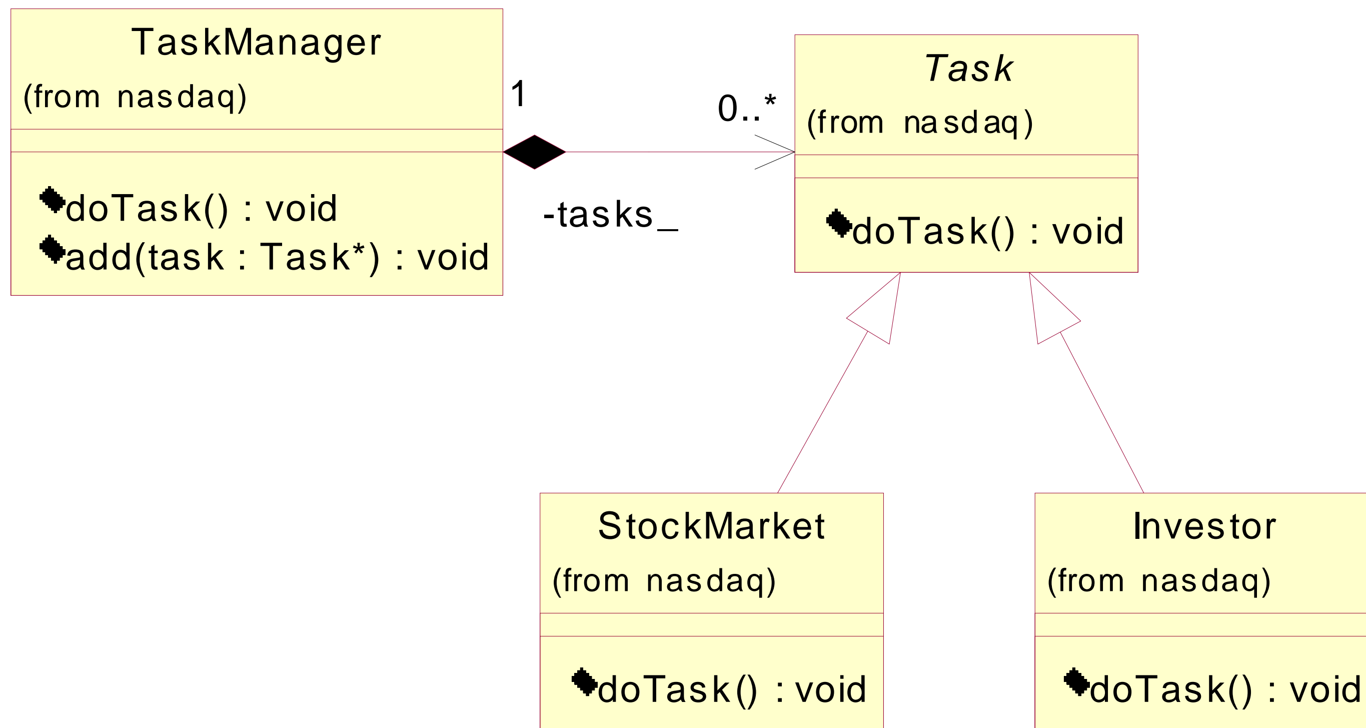
Come vengono gestiti gli ordini



Come vengono sincronizzate le operazioni

- Per simulare ogni “tornata” di acquisti e vendite:
  - i diversi investitori devono seguire le proprie strategie e decidere gli ordini
  - il mercato deve gestire gli ordini

Gestione simulata del “multitasking”



Come funziona il “multitasking”

Definizione delle implementazioni

```
#include "Task.h"
#include <map>
class Stock;
class BuyStrategy;
class StockMarket;

class Investor : public Task
{
public:
  Investor( double capital, StockMarket* );
  ~Investor();
  void setBuyStrategy( BuyStrategy * buy );
  void doTask();
  void print();
  bool buy ( Stock*, double price, long
    amount );
  bool sell( Stock*, double price, long
    amount );
  void reserve( double );
  void release( double );
  void reserve( Stock *, long );
  void release( Stock *, long );

  long amountAvailable( Stock * ) const;
```

```
private:
  double capital_;
  double reservedCapital_;
  StockMarket * market_;
  BuyStrategy * buyStrategy_;

  struct stockInfo
  {
    stockInfo() : buyPrice(0), amount(0),
   reserved(0) {}
    double buyPrice;
    long amount;
    long reserved;
    long available() const { return
   amount - reserved; }
  };

  map<Stock*, stockInfo> portfolio_;

  double buyPrice( Stock * ) const;
  long amount( Stock * ) const;
  long amountAvailable( Stock * ) const;
};
```

Implementazione metodi

```
void Investor::doTask()
{
  if ( market_ == 0 ) return;
  vector<Stock*>::const_iterator s;
  for( s = market_->begin(); s != market_->end(); s++ )
  {
    Stock * stock = *s;
    double buyPrice = Investor::buyPrice( stock );
    long amountAvailable = Investor::amountAvailable( stock );
    double capitalAvailable = capital_-  reservedCapital_;
    long buyAmount = buyStrategy_->buy( stock, capitalAvailable, buyPrice, amountAvailable );
    double price = buyStrategy_->price();
    if ( buyAmount < 0 )
    {
//        cout << "sell order: " << -buyAmount << " " << stock->symbol()
//           << " at " << price << endl;
      market_->addSell( new SellOrder( this, stock, price, -buyAmount ) );
    }
    else if ( buyAmount > 0 )
    {
//         cout << "buy order: " << buyAmount << " " << stock->symbol()
//            << " at " << price << endl;
      market_->addBuy( new BuyOrder( this, stock, price, buyAmount ) );
    }
  }
//  print();
}
```

Mettere tutto insieme

- Istanziare i titoli e il mercato
- Istanziare gli investitori
- Assegnare le strategie
  - i parametri sono determinati in base a numeri pseudocasuali
- Connettere i diversi oggetti
- … far partire il multitasking!

Il main program (1)

```
StockMarket nasdaq;
 Stock msft( "MSoft", 4.0);
 nasdaq.add( &msft );
 msft.setPrice( 5.0 );

 vector<Investor> investors;
 const unsigned int numberOfInvestors = 200;

 for( int k = 0; k < numberOfInvestors; k++ )
   investors.push_back( Investor( 100000,
   &nasdaq ) );

 const unsigned int initialBuyers = 100;
 for ( int j = 0; j < initialBuyers; j ++ )
 {
   investors[j].buy( &msft, msft.price(),
   20000 );
 }

 vector<Investor>::iterator i
 for( i = investors.begin(); i !=
   investors.end(); i++ )
   i->print();
```

```
for( vector<Investor>::iterator i =
    investors.begin(); i != investors.end();
    i++ )
  {
    CompositeBuyStrategy * strategy =
      new CompositeBuyStrategy;

    strategy->add( new StopLossStrategy
      ( 0.20 + drand48()*0.30 ) );
    strategy->add( new RandomBuyStrategy
      ( 0.05 + drand48()*0.15, 0.05 +
      drand48()*0.015, 0.02 + drand48()*0.48 )
      );
    strategy->add( new RandomSellStrategy
      ( 0.002 + drand48()*0.003, 0.05 +
      drand48()*0.015, 0.02 + drand48()*0.48 )
      );
    strategy->add( new SmartBuyStrategy
      ( 3, 0.010 + drand48()*0.020, 0.3 +
      drand48()*0.7 )  );
    strategy->add( new SimpleBuyStrategy
      ( 1, 0.005 + drand48()*0.015, 0.3 +
      drand48()*0.7 )  );
    strategy->add( new SimpleSellStrategy
      ( 0.1 + drand48()*0.20 ) );

    i->setBuyStrategy( strategy );
  }
```

Il main program (2)

```
TaskManager manager;

for( vector<Investor>::iterator i = investors.begin();
     i != investors.end(); i++ )
  manager.add( &*i );

manager.add( &nasdaq );
const unsigned int numberOfTasks = numberOfInvestors + 1;
const unsigned int numberOfTransactions = 5000;
const unsigned long iterations = (unsigned long)( numberOfTasks ) *
  (unsigned long)(numberOfTransactions);

for( unsigned long i = 0; i < iterations ; i++ )
  manager.doTask();

for( vector<Investor>::iterator i = investors.begin(); i !=
  investors.end(); i++ )
  i->print();
```

Risultati…

Quotazione ($?, €?)

9
8
7
6
5
4
3
2
1
0

speculazione

crollo

1
1001
2001

Tempo (giorni?)