Pattern di disegno software

Matteo Duranti

matteo.duranti@pg.infn.it

(cfr. Gamma, Helm, Johnson e Vlissides - Design Patterns: Elements of Reusable Object-Oriented Software [http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf (legale?!)] https://en.wikipedia.org/wiki/Design_Patterns)

Design pattern

Nel “disegno” di un software è importante utilizzare delle “soluzioni”:

- siano chiare
- siano funzionali
- rendano il sw facilmente modificabile e riusabile (non sia tutto *hard-codato* e quindi impossibile da cambiare senza praticamente riscriverlo)

→ tipicamente queste soluzioni sono “ricorrenti” e indipendenti dal linguaggio utilizzato

Design pattern

In generale questo si ottiene:

- ragionando in termini di *client* (il pezzo di codice che "necessita" di utilizzare un algoritmo) e *server* (il pezzo di software che offre quell'algoritmo) come *oggetti* del software finale;
- creando delle 'interfacce' agli algoritmi utilizzati: "incapsulando" tutto è possibile disaccoppiare *client* e *server*;
- realizzando il software finale come *composizione di oggetti* invece che come un'unica *implementazione*

Design pattern

- Creazionali:
  creano gli oggetti invece di lasciare che questi vengano instanziati direttamente;
- Strutturali:
  servono a collegare più interfacce o oggetti fra loro;
- Comportamentali:
  gestiscono la comunicazione fra diverse interfacce o oggetti;

Design pattern

- La "bibbia" dei pattern è il celebre libro del 1994:
  Design Patterns: Elements of Reusable Object-Oriented Software
  della GoF (Gang of Four: E.Gamma, R.Helm, R.Johnson, J.Vlissides)
- Il loro utilizzo è spesso "utile" sono per veri sviluppatori sw
- Nella lezione verrano descritti brevemente alcuni dei pattern che è comune trovare anche nel sw utilizzato e sviluppato nella Ricerca in Fisica

Design pattern

- Strutturali:
  - Adapter (wrapper)

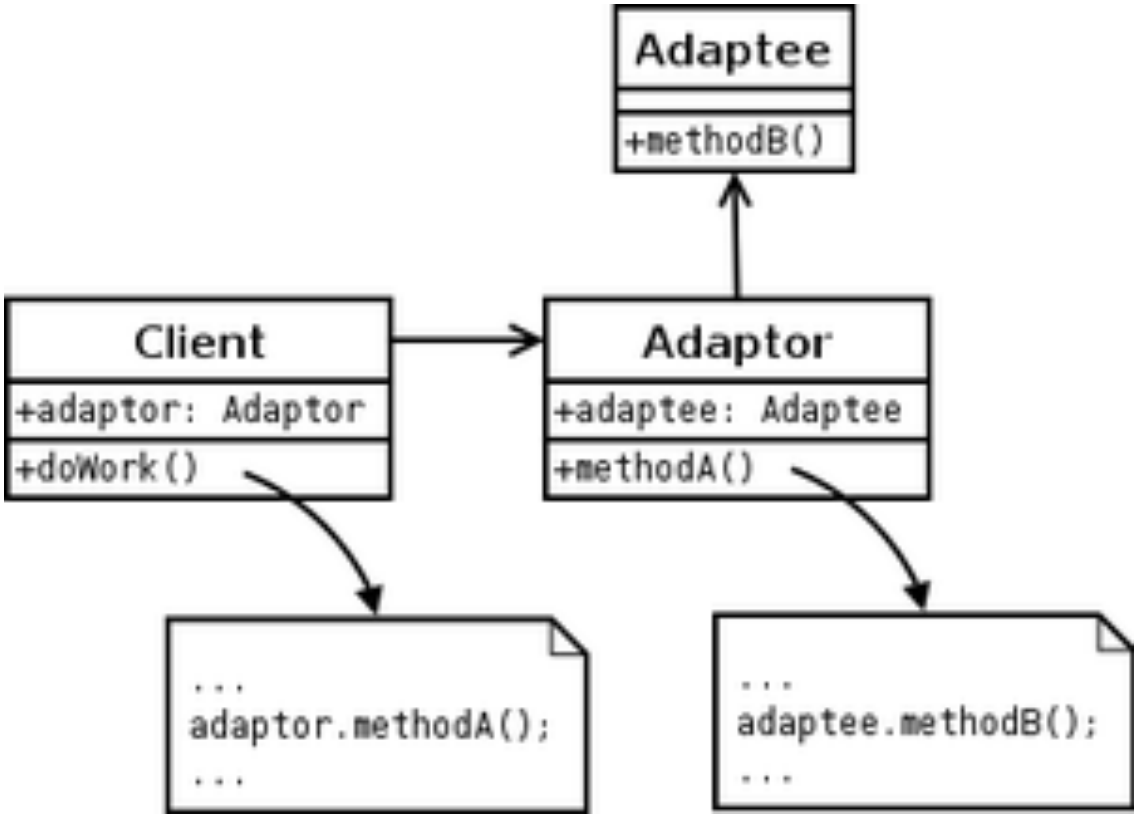
Adapter (wrapper)

Il *client* deve utilizzare il *methodB()*** del *server* (*Adaptee*).

E' conveniente (non in termini di perfomances!) mettere un layer (*Adaptor*), che funge anche da wrapper: "espone" *methodA()*, "possiede" il vero server (*Adaptee*, di cui, ad esempio, ha un'istanza, *adaptee*).

Il client utilizza *Adaptor::methodA()* mentre poi internamente viene utilizzato *Adaptee::methodB()*

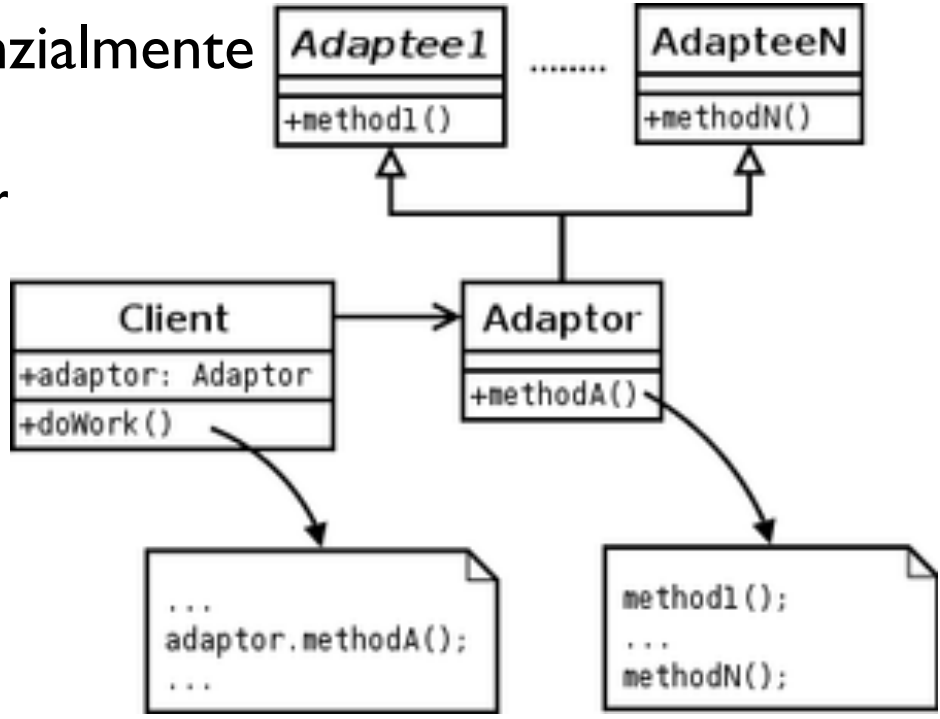
** i nomi sono ovviamente esempi



Adapter (wrapper)

Questo è utile per (almeno) due cose:

- *Adaptee* ha un' interfaccia "complessa" (i.e. che, ad esempio, accetta molti parametri) in quanto di utilizzo molto più generale
  → al *client* viene "esposta" un'interfaccia molto più "semplice" e aderente all'utilizzo particolare del progetto;
- In futuro potremmo avere altri *server* (*Adaptee1*, *Adaptee2*, etc…) potenzialmente con interfacce diverse
  → possiamo scegliere quale *server* utilizzare (magari a *runtime* e non in compilazione) e non dobbiamo cambiare minimamente l'implementazione del client (che continuerà ad utilizzare *Adaptor::methodA()*)



Adapter (wrapper)

- Un utilizzo molto comune è anche quando ci viene dato il sorgente di un codice (che magari è complesso e difficilmente modificabile) che fa una cosa e abbiamo un programma (magari anch'esso complesso e difficilmente modificabile), che lo deve utilizzare: creare un wrapper consente, potenzialmente, di non modificare nessuno dei due codici pre-esistenti;

- L'interfaccia a differenti generatori di numeri random, oggetto di una delle esercitazioni, era un esempio di *wrapper*;

Adapter (wrapper)

```
class RndGen {

private:
  unsigned long long int random;
  int algo;
  TRandom* trand;

public:
  RndGen(unsigned long long int seed=345);

  inline void SetAlgorithm(int algorithm) { algo=algorithm; };
  void SetSeed(unsigned long long int seed);
  inline unsigned long long int GetRandom() { return random; };
  unsigned long long int GetRandomAndThrowANewOne();

private:
  unsigned long long int GetRandomAndThrowANewOneSimple();
  unsigned long long int GetRandomAndThrowANewOneLCG();
  unsigned long long int GetRandomAndThrowANewOneTRandom();
};

RndGen::RndGen(unsigned long long int seed) {

  algo=1;
  trand  = new TRandom();

  SetSeed(seed);

  return;
}
```

la classe RndGen “espone” il metodo *GetRandomAndThrowANewOne()* che, a seconda di *algo*, chiama un differente generatore/algoritmo

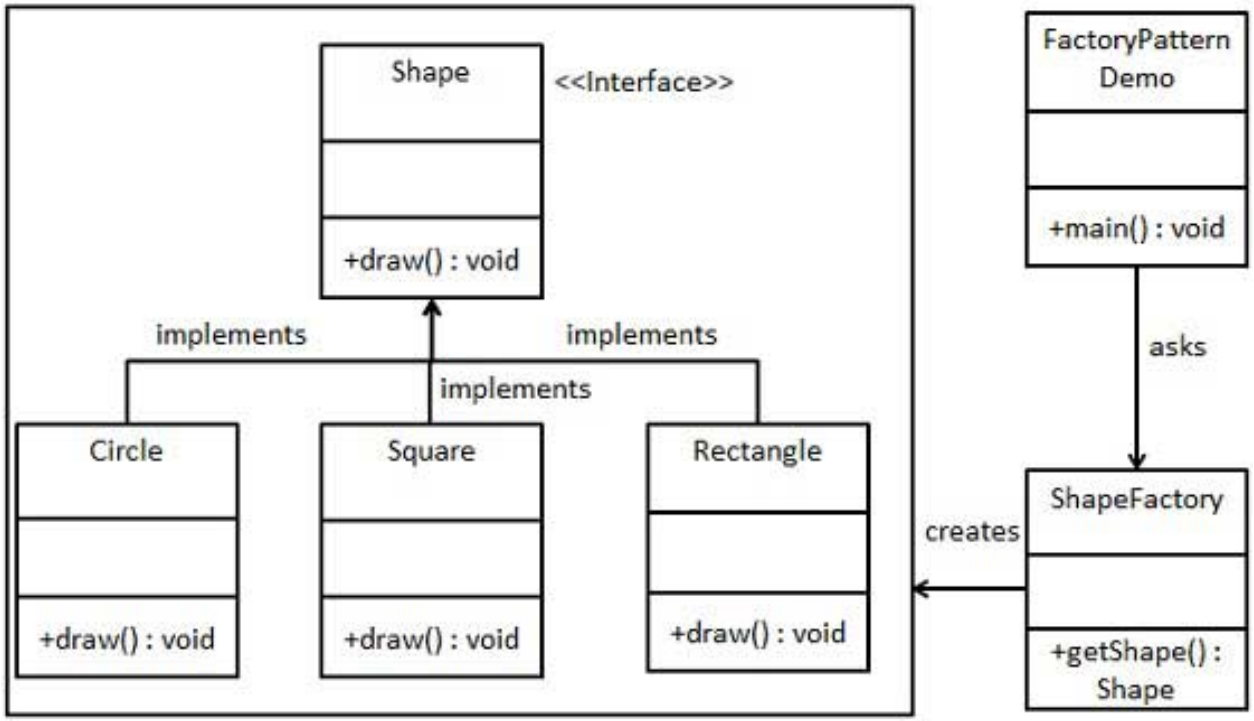
Design pattern

- Creazionali:
  - Factory
  - Abstract factory
  - Singleton

Factory

- E' un oggetto che *crea* (i.e. *istanzia*) altri oggetti, evitando al *client* di dover conoscere la classe esatta e la corrispettiva sintassi

- Anche qui il concetto si "confonde" con quello di *wrapper*, ma la specificità della *factory* è nel ruolo di creazione di altri oggetti



Factory

```
class IPerson { string GetName(){}; }

class Villager: IPerson {
    public string GetName() { return "Village Person”; }
}

class CityPerson: IPerson {
    public string GetName() { return "City Person”; }
}

enum PersonType {
    Rural,
    Urban
}

class Factory {
    public IPerson GetPerson(PersonType type){
        switch (type) {
            case PersonType.Rural:
                return new Villager();
            case PersonType.Urban:
                return new CityPerson();
        }
    }
}

int main() {
   Factory _factory;
   IPerson _person = _factory.GetPerson(Rural);
   cout<<_person.GetName();
}
```

il programma (*client*) “vede” solamente la Factory su cui opera utilizzando solamente un metodo “generico” (i.e. *GetPerson()*) che restituisce un oggetto “generico” (i.e. *IPerson*).

Abstract factory

- Incapsula la creazione di differenti classi concrete dentro un'interfaccia generica, esattamente come la Factory;

- Implementa diverse *factory*, ognuna specializzata nella creazione di oggetti diversi;

- In C++ si può ottenere "semplicemente" tramite l'utilizzo dell'ereditarietà;

Abstract factory

```
class IButton { void Paint(); }
class IGUIFactory { IButton CreateButton(){}; }
class WinFactory: IGUIFactory {
    public: IButton CreateButton() { return new WinButton(); }
}
class OSXFactory: IGUIFactory {
    public: IButton CreateButton() { return new OSXButton(); }
}
class WinButton: IButton {
    public: void Paint() { do_something_Win(); }
}
class OSXButton: IButton {
    public: void Paint() { do_something_OSX(); }
}

int main() {
    var appearance = Settings.Appearance;
    IGUIFactory factory;
    switch (appearance) {
        case Appearance.Win:
            factory = new WinFactory();
            break;
        case Appearance.OSX:
            factory = new OSXFactory();
            break;
    }

    var button = factory.CreateButton();
    button.Paint();
    return 0;
}
```

il programma (*client*) instanzia la *Factory* (generica, astratta) che in realtà è una factory specializzata (nell'esempio *WinFactory()* o *OSXFactory()*), in base ad una scelta condizionale e poi usa quella che, in maniera del tutto trasparente, fa le operazioni, specializzate, necessarie

Abstract Factory vs Factory

- I due concetti sono, ovviamente. molto simili
- La *abstract factory* è una *factory* astratta che implementa diverse *factory*
- Esempi:
  - esempio "concreto" di factory:
    una fabbrica di giocatoli "produce giocattoli", e ha diversi reparti per creare giocattoli diversi;
  - esempio "concreto" di abstract factory:
    un trapano (*abstract factory)* è un oggetto che serve "a fare i buchi". In realtà i buchi li "sa fare" ogni singola punta (*factory*), ognuna di una dimensione diversa, mentre il trapano non li "sa fare";

Singleton

- Restringe l'istanziazione di una classe ad un singolo oggetto
- Praticamente "appare" come se fosse una "variabile globale"
- E' utile quando è comodo/richiesto avere un'unico oggetto in tutti il progetto (ad esempio evitate di dover passare il puntatore a questo oggetto fra le varie funzioni). Ad esempio una *factory* può essere un *singleton*
- In C++ si può implementare facilmente rendendo *private* il costruttore e "esponendo" solo una funzione di accesso all'unico oggetto istanziato

Singleton

```
class Singleton {
  private:
    static Singleton *p_inst;
    Singleton();

  public:
    static Singleton* gethead(){
      if (!p_inst){
        p_inst = new Singleton();
      }

      return p_inst;
    }
    void DoSomething();
};
```

```
Singleton* p_inst = NULL;

int main() {
    Singleton::gethead()->DoSomething();
    return 0;
}
```

Il costruttore e il puntatore “a sé stesso” sono *private*.
L’unico “punto di accesso”, per il *client*, è la funzione che restituisce (in questo caso crea, se non esiste) il puntatore “a sé stesso”

In questa implementazione, chiamata *lazy implementation* (implementazione “pigra”), l’oggetto:
- può solo venire creato nell’*heap* (i.e. “*new Singleton()*”);
- viceversa viene creato (i.e. occupa memoria) solamente se viene utilizzato

Questa implementazione presenta problemi in caso di calcolo parallelo

Singleton

```
class Singleton {
 private:
    Singleton();

  public:
    static Singleton& gethead(){
        static Singleton inst;
        return inst;
    }
    void DoSomething();
};
```

```
int main() {
    Singleton::gethead().DoSomething();
    return 0;
}
```

Il costruttore è *private*.
L'unico "punto di accesso", per il *client*, è la funzione che restituisce l'unico oggetto, *static*, esistente

In questa implementazione, l'oggetto:
- può solo venire creato nello *stack* (i.e. "*Singleton _sing*");
- è *static*, quindi è creato (i.e. viene usata memoria) anche se non utilizzato

Questa implementazione è più facilmente utilizzabile in ambienti di calcolo parallelo

Design pattern

- Comportamentali:
  - Iterator

Iterator

- E' usato per attraversare, in maniera ordinata, un contenitore (ad esempio un *std::vector*) di oggetti di un qualsiasi tipo
- Permette di accere agli elementi nel contenitore
- Permette di implementare un *loop* sugli oggetti nel contenitore senza la necessità di una sintassi specifica per ogni tipo di contenitore/classe
- In C++ è built-in all'interno dei contenitori *standard* (i.e. delle *STL*), come *vector*, *map*, *multimap*, etc…

Iterator

```
#include <iterator>     // std::iterator, std::input_iterator_tag

class MyIterator: public std::iterator<std::input_iterator_tag, int> {
  int* p;
public:
  MyIterator(int* x) :p(x) {}
  MyIterator(const MyIterator& mit) : p(mit.p) {}
  MyIterator& operator++() {++p;return *this;}
  MyIterator operator++(int) {MyIterator tmp(*this); operator++(); return tmp;}
  bool operator==(const MyIterator& rhs) const {return p==rhs.p;}
  bool operator!=(const MyIterator& rhs) const {return p!=rhs.p;}
  int& operator*() {return *p;}
};

int main () {
  int numbers[]={10,20,30,40,50};
  MyIterator from(numbers);
  MyIterator until(numbers+5);
  for (MyIterator it=from; it!=until; it++)
    std::cout << *it << ' ';
  std::cout << '\n';

  return 0;
}
```

Il loop sugli elementi di un “array” (potrebbero essere di tipo qualsiasi, sia gli elementi che l’”array”) è fatto con una sintassi “generale”

Iterator

- Rende molto "facile", ad esempio, l'*ordinamento* degli elementi, di tipo generico, di un contenitore generico:

```
class Skyscraper {
public:
    Skyscraper(const std::string& name, double height);
    double getHeight() const {return m_height;}
    void print() const;

private:
    std::string m_name;
    double m_height;
};

bool operator<(const Skyscraper &s1, const Skyscraper &s2);
```

```
Skyscraper::Skyscraper(const std::string &name, double height):
    m_name(name), m_height(height) {}

void Skyscraper::print() const {
    std::cout << this->m_name << " "
              << this->m_height << '\n';
}

bool operator<(const Skyscraper &s1, const Skyscraper &s2){
    if(s1.getHeight() < s2.getHeight()) return true;
    else return false;
}
```

Iterator

- Rende molto “facile”, ad esempio, l’*ordinamento* degli elementi, di tipo generico, di un contenitore generico:

```
int main()
{
    std::vector<Skyscraper> skyscrapers;
    skyscrapers.push_back(Skyscraper("Empire State", 381));
    skyscrapers.push_back(Skyscraper("Petronas", 452));
    skyscrapers.push_back(Skyscraper("Burj Khalifa", 828));
    skyscrapers.push_back(Skyscraper("Taipei", 509));

    std::sort(skyscrapers.begin(),skyscrapers.end());
    for(unsigned int i = 0; i < skyscrapers.size(); i++)
        skyscrapers.at(i).print();

    return 0;
}
```

std::sort beneficia di un modo standard di fare il l*oop* sugli eventi (e dell’overloading dell’operatore ‘<‘)

```
Empire State 381
Petronas 452
Taipei 509
Burj Khalifa 828
```