SOLID Principles

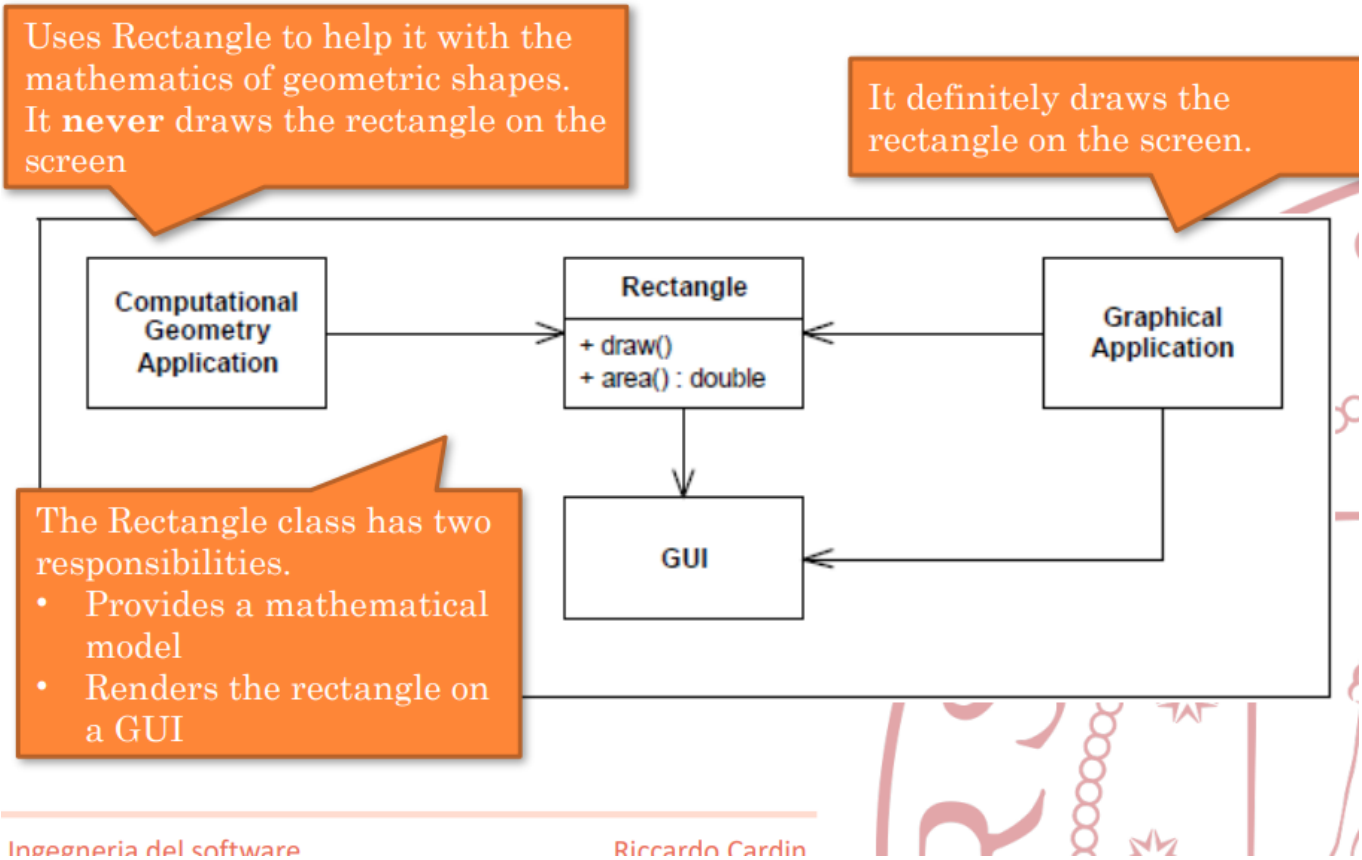
- Robert Martin

Single Responsibility Principle

- Una classe dovrebbe avere una sola responsabilità = avere solo una ragione per cambiare
- Elementi coesi = in grado di essere indipendenti gli uni dagli altri

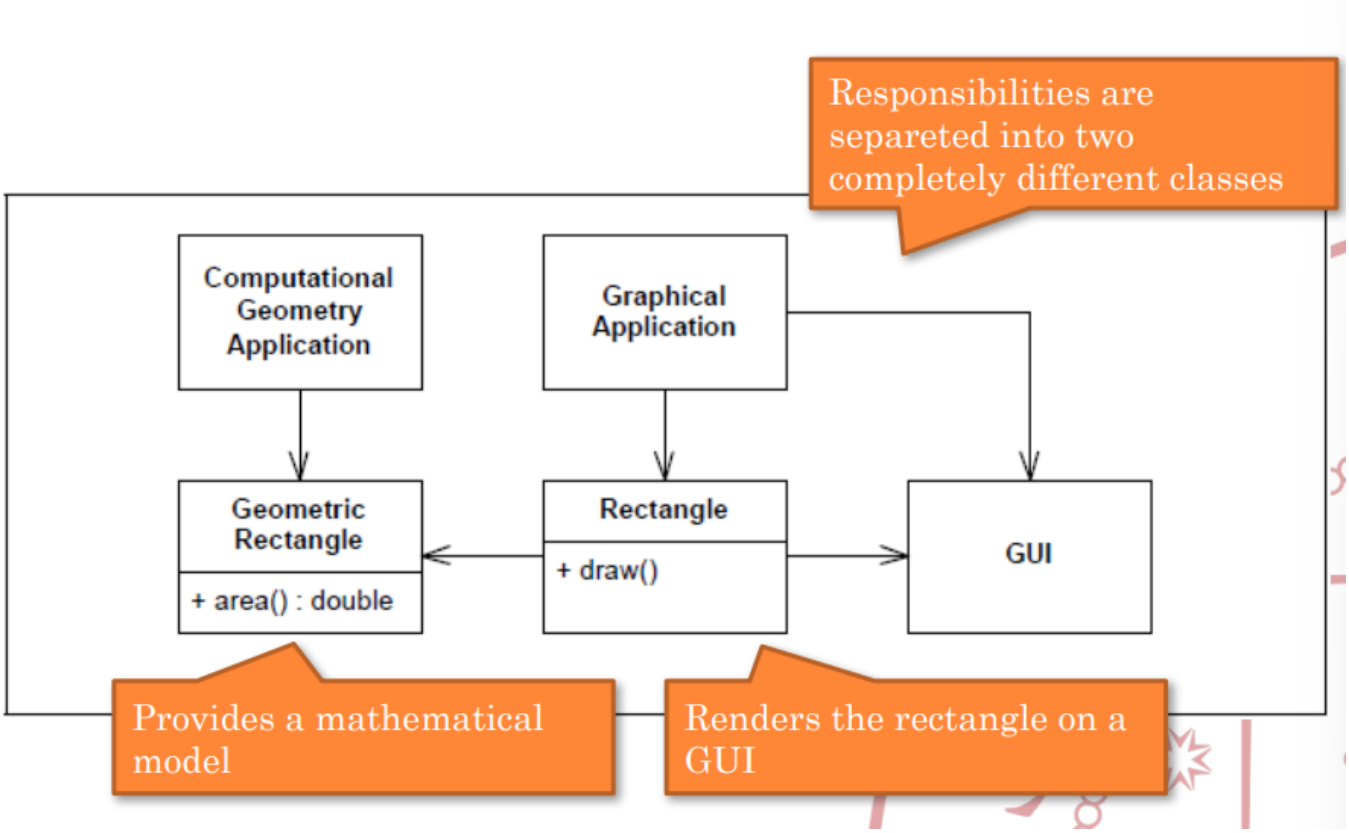
Problema:

- Mischiare metodi di gestione dati/GUI
- Mischiare metodi con diverse responsabilità



Soluzione:

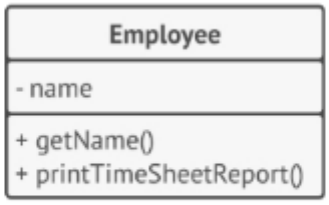
- Vado a spezzare le cose in più parti (gestendo modularmente)



Altro esempio:

- Due responsabilità mischiate nella classe Employee

The `Employee` class has several reasons to change. The first reason might be related to the main job of the class: managing employee data. However, there's another reason: the format of the timesheet report may change over time, requiring you to change the code within the class.

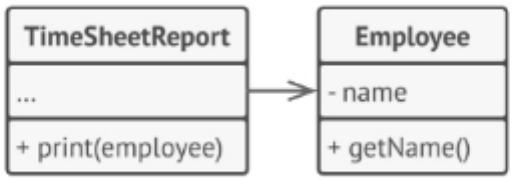


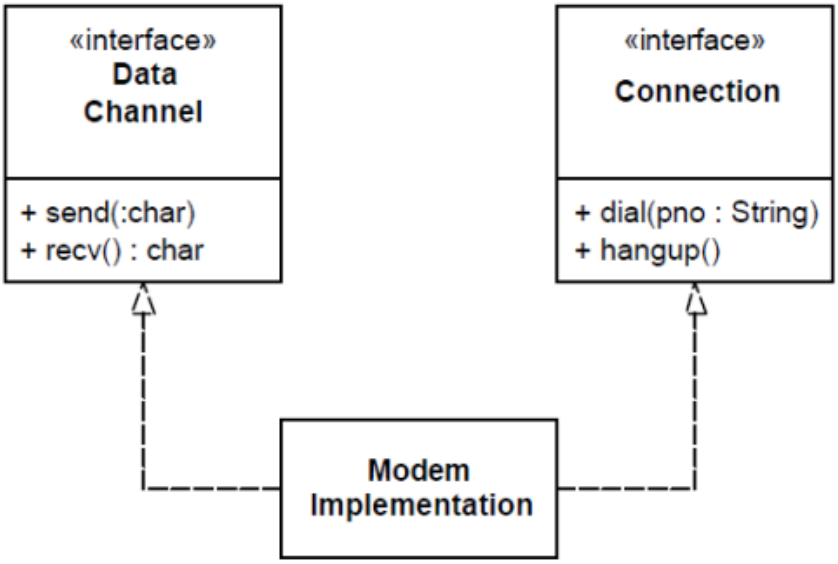
BEFORE: the class contains several different behaviors.

Soluzione:

- Spezzo le responsabilità in classi

Solve the problem by moving the behavior related to printing timesheet reports into a separate class. This change lets you move other report-related stuff to the new class.





- Eventually separate responsibilities avoids rigidity

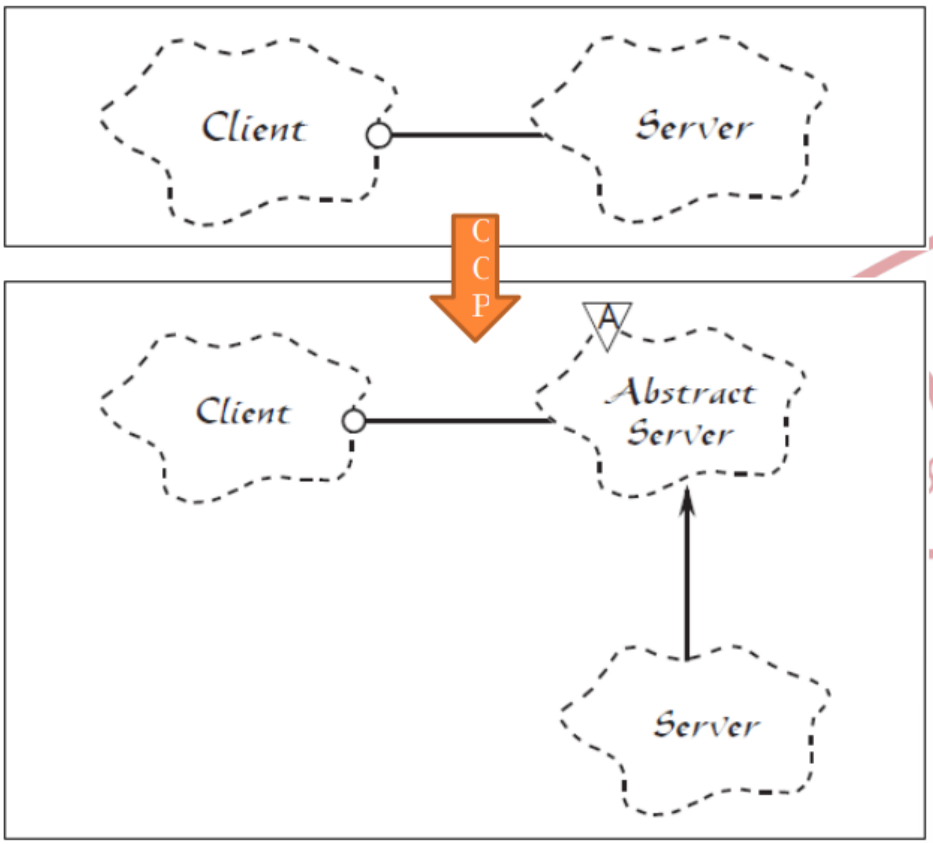
Open/Closed Principle

- Classi chiuse alla modifica ma aperte all'estensione
- Utilizziamo l'astrazione

Esempio: non dipendo dal contesto o dalle sottoclassi per Client/Server

`Client` class must be changed to name the new server class.

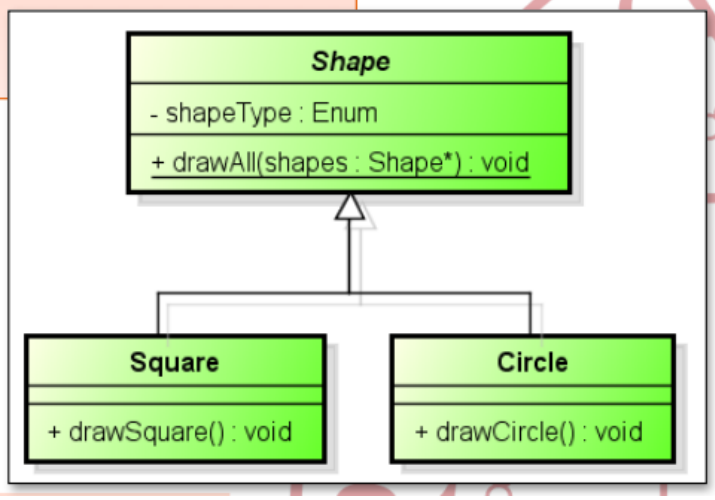
If we want `Client` objects to use a different server class, then a new derivative of the `AbstractServer` class can be created. The `Client` class can remain unchanged.



Problema: dipendiamo dalle forme (sottoclassi a basso livello) e quindi, se cambio da una parte, devo cambiare anche dall'altra

```
public static void drawAll(Shape[] shapes) {
    for (Shape shape : shapes) {
        switch (shape.shapeType) {
            case Square:
                ((Square) shape).drawSquare();
                break;
            case Circle:
                ((Circle) shape).drawCircle();
                break;
        }
    }
}
```

Does not conform to the open-closed principle because it cannot be closed against new kinds of shapes. If I wanted to extend this function, I would have to modify the function

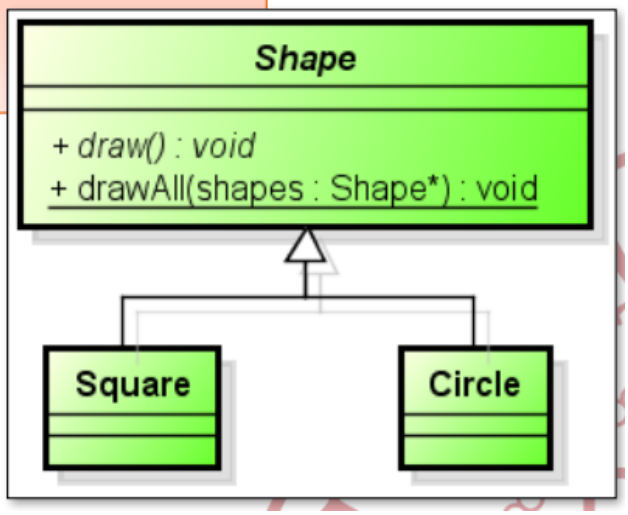


Soluzione:

- Spezzare le sottoclassi e usare l'astrazione (ereditarietà) in modo intelligente

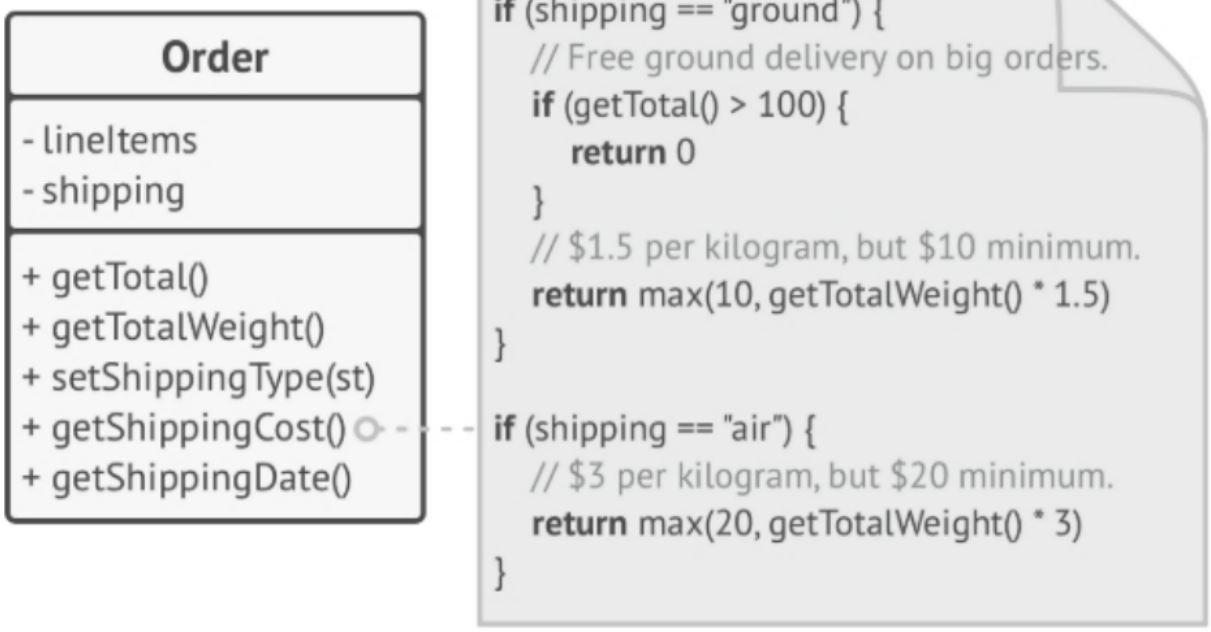
```
public static void drawAll(Shape[] shapes) {
    for (Shape shape : shapes) {
        shape.draw();
    }
}
```

Solution that conforms to open-close principle. To extend the behavior of the `drawAll` to draw a new kind of shape, all we need do is add a new derivative of the `Shape` class.



Altro esempio:

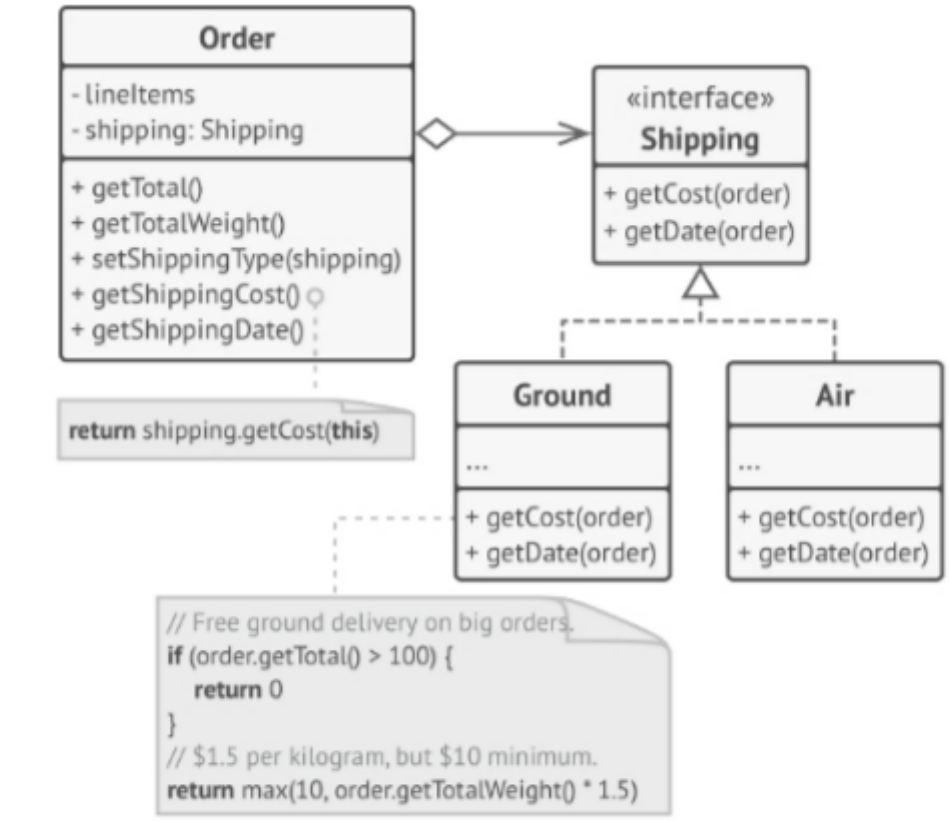
- La classe Ordine dipende dal tipo di consegna
- Duplicazione codice e gestione a basso livello



BEFORE: you have to change the `Order` class whenever you add a new shipping method to the app.

Soluzione:

- Tolgo il comportamento problematico
- Definisco un'interfaccia che gestisce



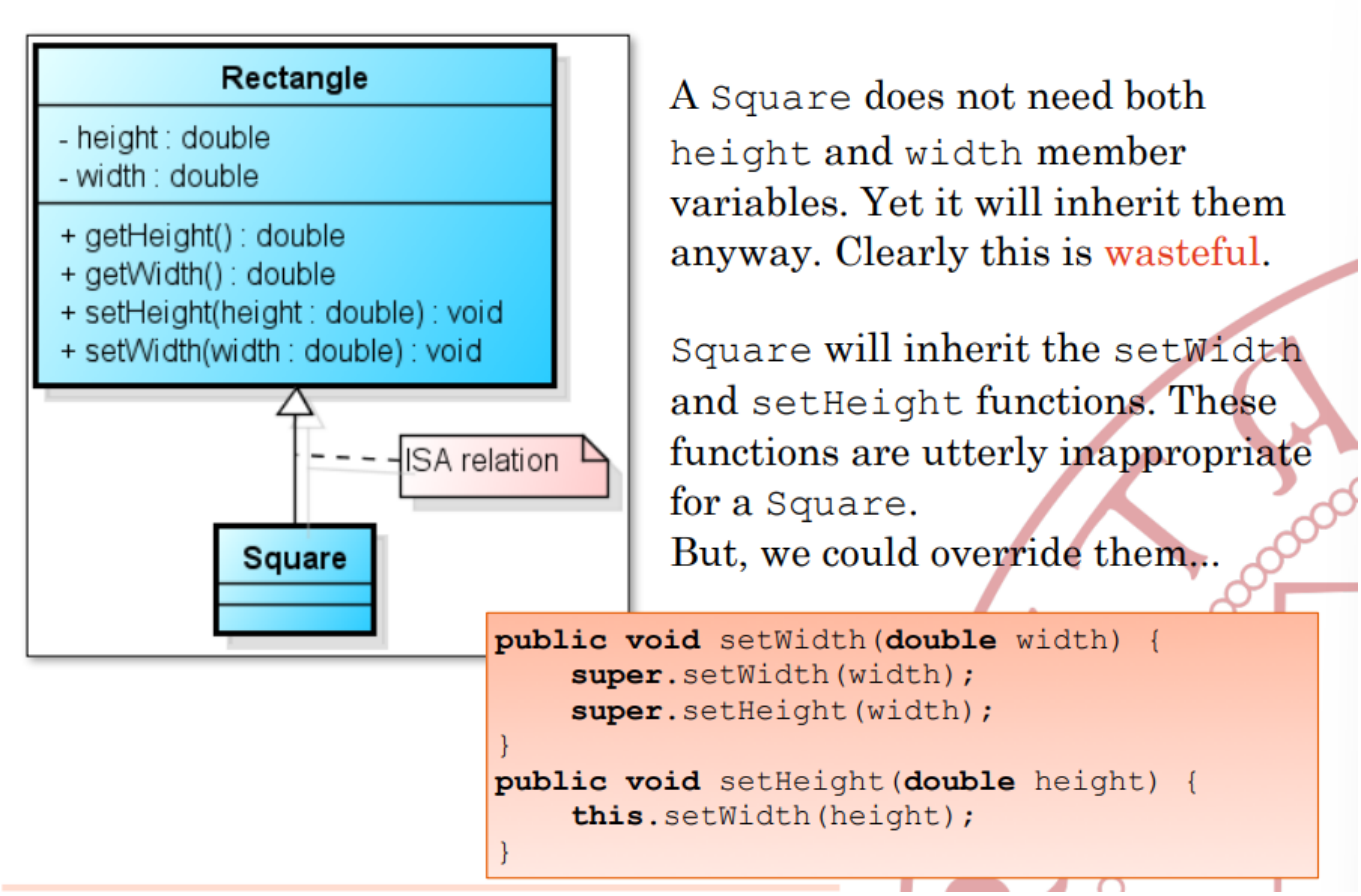
AFTER: adding a new shipping method doesn't require changing existing classes.

Liskov Substitution Principle

- Quando si estendono classi, dovremmo riuscire a passare oggetti della sottoclasse al posto degli oggetti della classe base senza rompere il codice/senza avere il problemi
- Passaggio "semplice" dalla sottoclasse alla superclasse "senza saperlo" = in modo liscio
- A livello pratico = uso dell'overriding

Problema:

- la sottoclasse contiene informazioni (es. larghezza e altezza) che non le servono
- violazione isolamento e della separation of concerns
- le superclassi non devono aggiungere comportamento per le sottoclassi



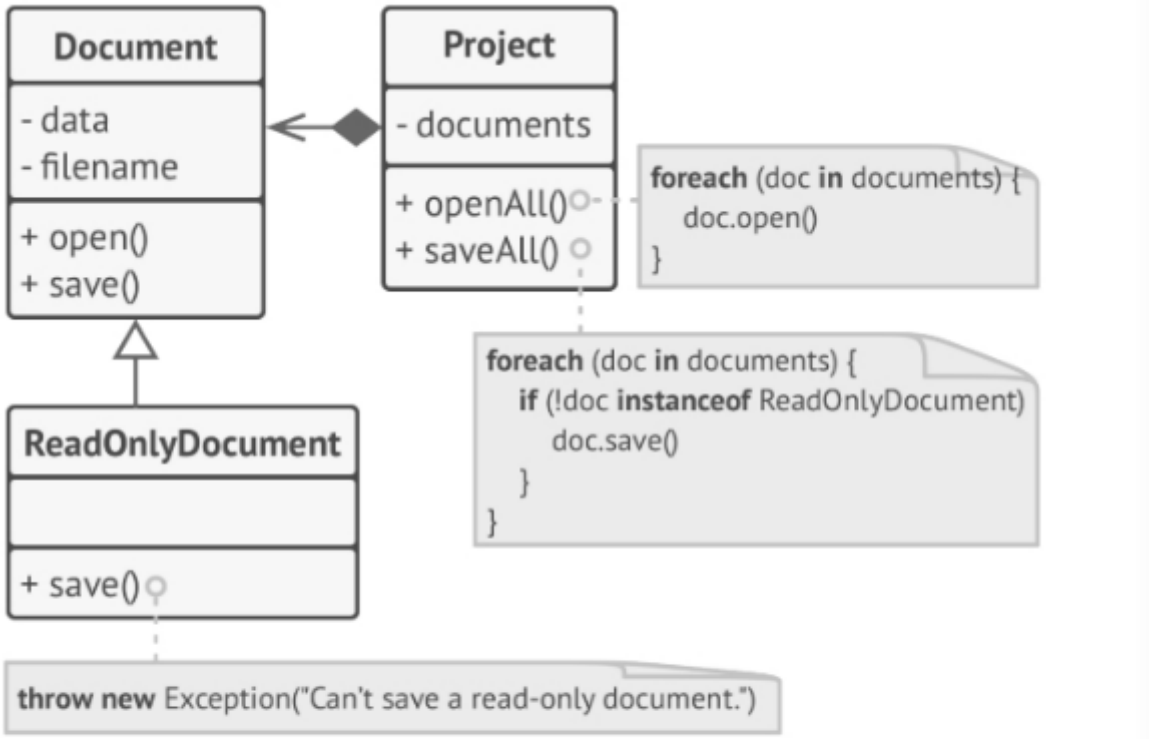
Soluzione:

- Utilizzo un'interfaccia che disegna in maniera tale che non dipendo né da Rectangle né da Square

Conseguenze:

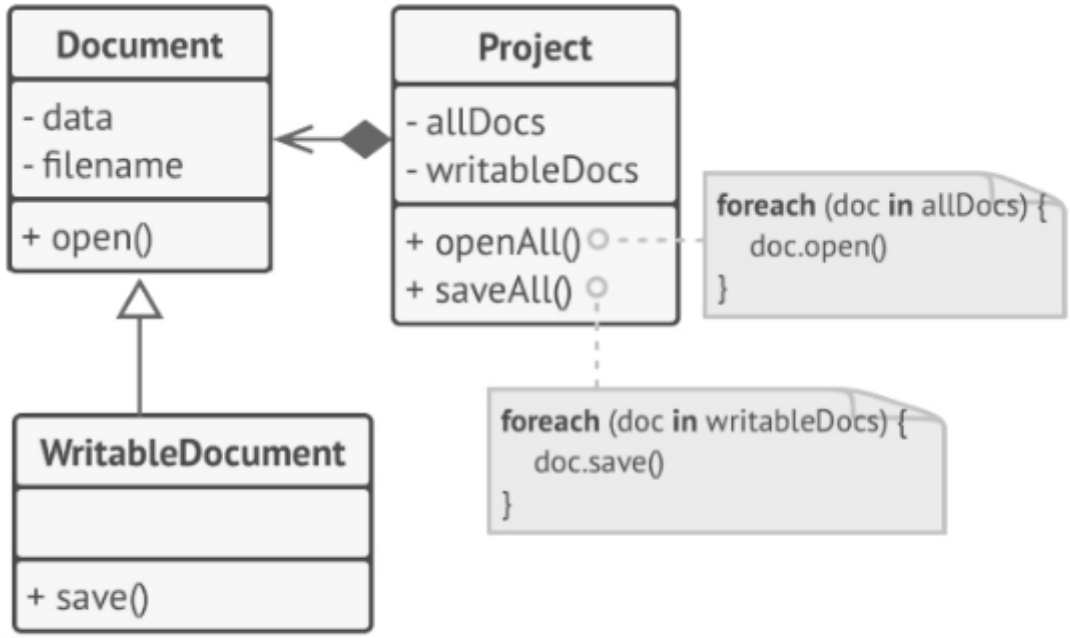
- Design by contract first (realizzo dei metodi per oggetti astratti con un comportamento specifico = contratti/interfacce)
- I parametri delle sottoclassi possono essere per astrazione facilmente sostituiti dalle superclassi
- Le sottoclassi lanciano eccezioni delle superclassi

Problema: per ereditarietà offriamo alla sottoclasse una funzionalità che non le serve



BEFORE: saving doesn't make sense in a read-only document, so the subclass tries to solve it by resetting the base behavior in the overridden method.

Soluzione: offro la funzionalità che mi serve solo per la classe desiderata oppure uso un'interfaccia esterna per quella specifica sottoclasse

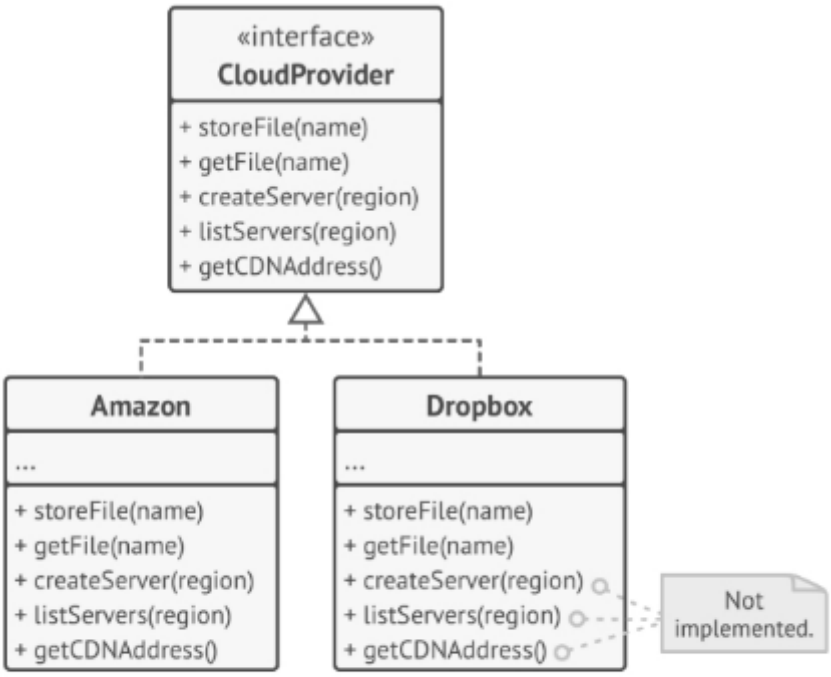


AFTER: the problem is solved after making the read-only document class the base class of the hierarchy.

Interface Segregation Principle

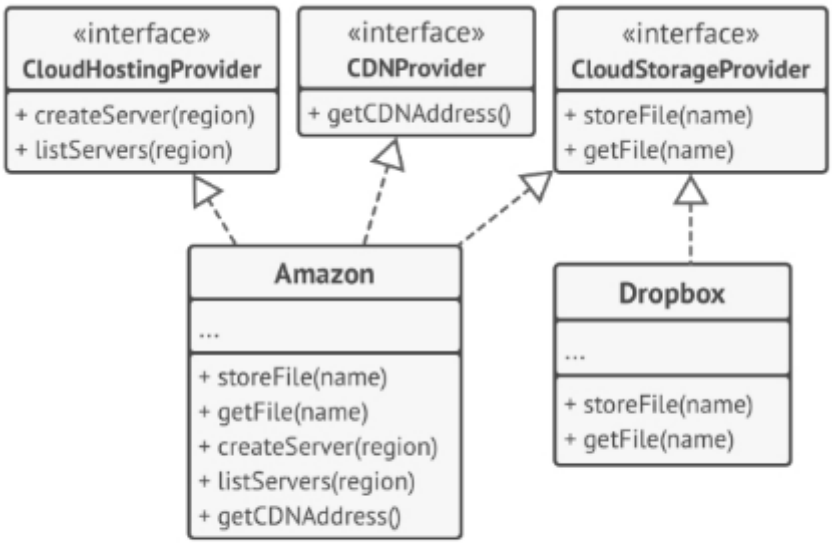
- I client non devono dipendere da metodi che non utilizzano
- Stiamo utilizzando l'ereditarietà nel modo sbagliato
- Le interfacce hanno dei metodi che non utilizzano (non sono chiuse alle modifiche
- Il Liskov delle interfacce, se noti

Problema: inquiniamo un'interfaccia con qualcosa che non usa (caso Dropbox)



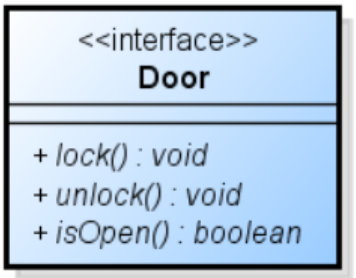
BEFORE: not all clients can satisfy the requirements of the bloated interface.

Soluzione: creo una serie di interfacce per gestire separatamente i metodi diversi tra le due classi, rendendo a grana fine i comportamenti



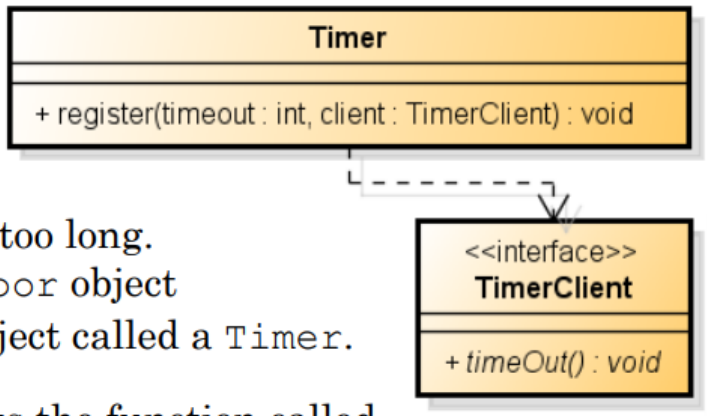
AFTER: one bloated interface is broken down into a set of more granular interfaces.

Esempio Cardin: non tutte le porte sono temporizzate ed offro qualcosa che non tutti usano



In this system there are `Door` objects that can be locked and unlocked, and which know whether they are open or closed.

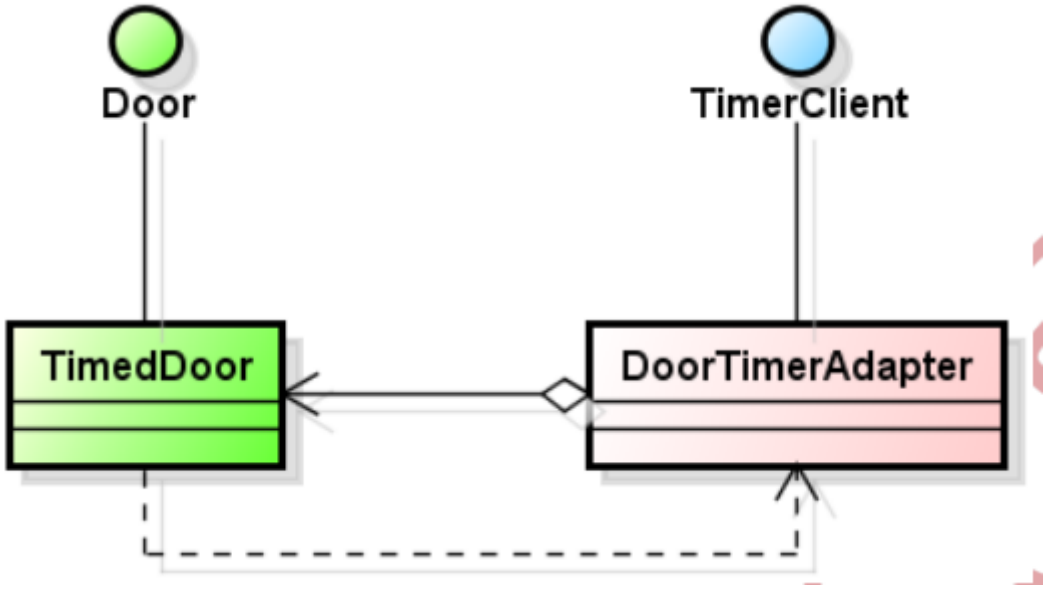
Clients used this interface to managed doors.



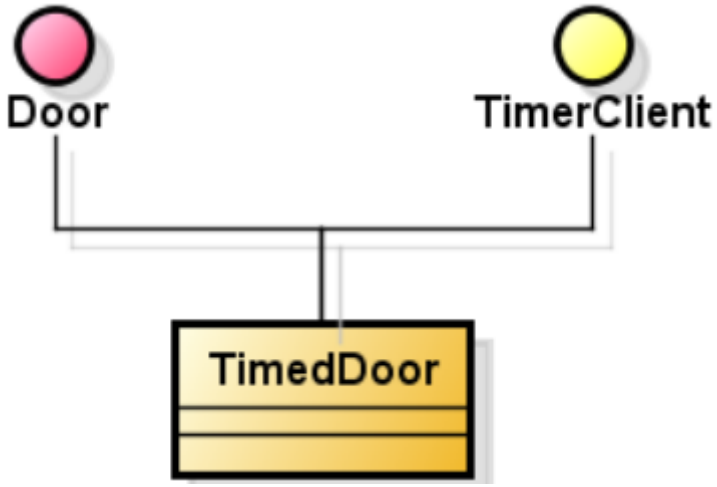
Now consider that one such implementation. `TimedDoor` needs to sound an alarm when the door has been left open for too long. In order to do this the `TimedDoor` object communicates with another object called a `Timer`.

`TimeClient` method represents the function called when the timeout expires

Soluzione: creo un'interfaccia separata per astrarre dal comportamento del client e fornire il metodo diverso come interfaccia a sé stante (primo esempio)



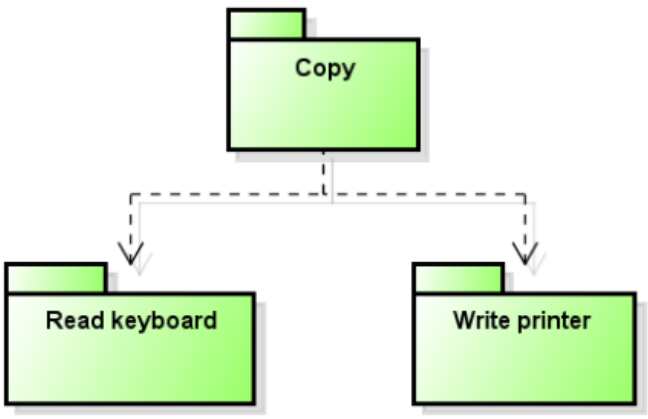
Secondo esempio: spezzo con ereditarietà in più sottoclassi



Dependency Inversion Principle

- Le classi ad alto livello non dipendono dalle classi a basso livello, ma dipendono solo dalle astrazioni
- Esistono dipendenza "da sopra a sotto" che dobbiamo risolvere

Problema: la classe "Copy" dipende da come vengono letti/scritti i file



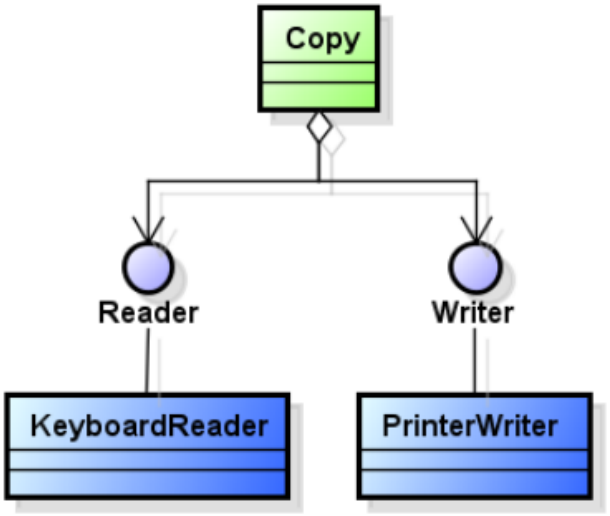
Consider a simple program that is charged with the task of copying characters typed on a keyboard to a printer.

"Read keyboard" and "Write printer" are quite reusable. However the "Copy" module is not reusable in any context that does not involve keyboard and printer

```
public void copy(OutputDevice dev) {
    int c;
    while ((c = readKeyboard()) != EOF)
        if (dev == PRINTER)
            writePrinter(c);
        else
            writeDisk(c);
}
```

Violates OCP

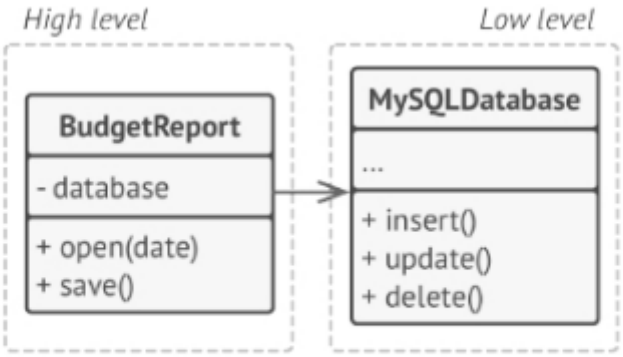
Soluzione: spezzo in interfacce i comportamenti per renderli indipendenti



We have performed dependency inversion. The dependencies have been inverted; the "Copy" class depends upon abstractions, and the detailed readers and writers depend upon the same abstractions.

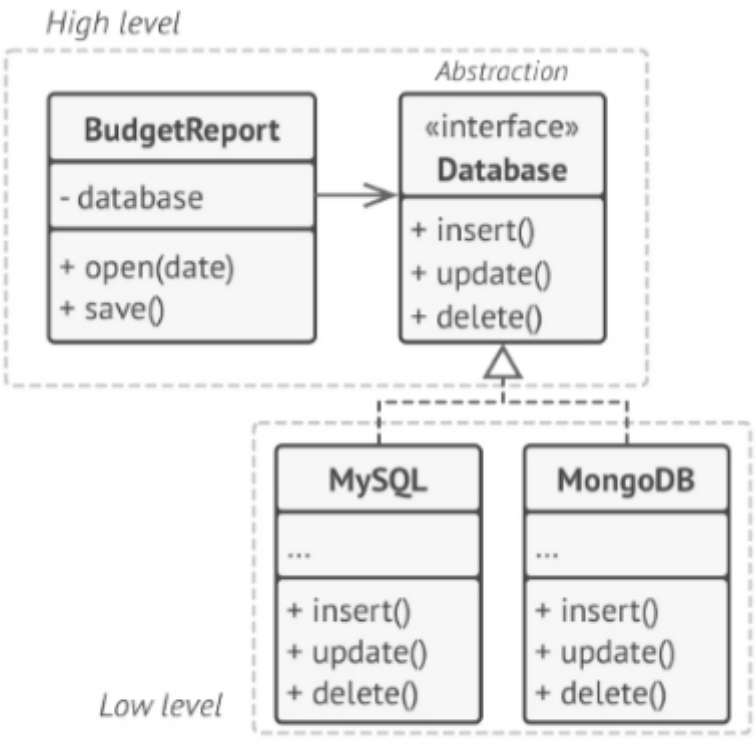
Now we can reuse the "Copy" class, independently of the "Keyboard Reader" and the "Printer Writer"

Problema: il report del budget (classe ad alto livello) dipende dal DB sottostante (classe a basso livello)



BEFORE: a high-level class depends on a low-level class.

Soluzione: creare un'interfaccia comune per gestire i diversi DB (e fare in modo che i comportamenti a basso livello siano separati gli uni dagli altri)



AFTER: low-level classes depend on a high-level abstraction.