Coesione

- La Coesione è una misura di quanto siano fortemente relate e mirate le responsabilità (o servizi offerti) di un modulo.
- Se ciascuna unità è responsabile di un singolo compito, diciamo che tale unità ha una alta coesione.
- Un'alta coesione è una proprietà desiderabile del codice, poiché
- permette:
  - Di comprendere meglio i ruoli di una classe
  - Di riusare una classe
  - Di manutenere una classe
  - Di limitare e focalizzare i cambiamenti nel codice
  - Utilizzare nomi appropriati, efficaci, comunicativi

Granularità della Coesione

- Coesione dei metodi
  - Un metodo dovrebbe essere responsabile di un solo compito ben definito
- Coesione delle classi
  - Ogni classe dovrebbe rappresentare un singolo concetto ben definito

Coesione - Il Single Responsibility Principle

- Una classe dovrebbe avere UNA sola responsabilità
- Una classe dovrebbe avere UN solo motivo per cambiare
- Una responsabilità è "una ragione per cambiare".
- Le responsabilità di una classe sono assi di cambiamento.
- Se ha due responsabilità, queste sono accoppiate nel progetto, e quindi devono cambiare insieme.
- Se una classe ha più di una responsabilità, può diventare fragile quando uno qualsiasi dei requisiti cambia

Coesione – Example

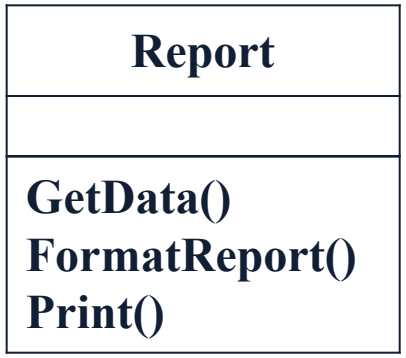
Report
GetData() FormatReport() Print()

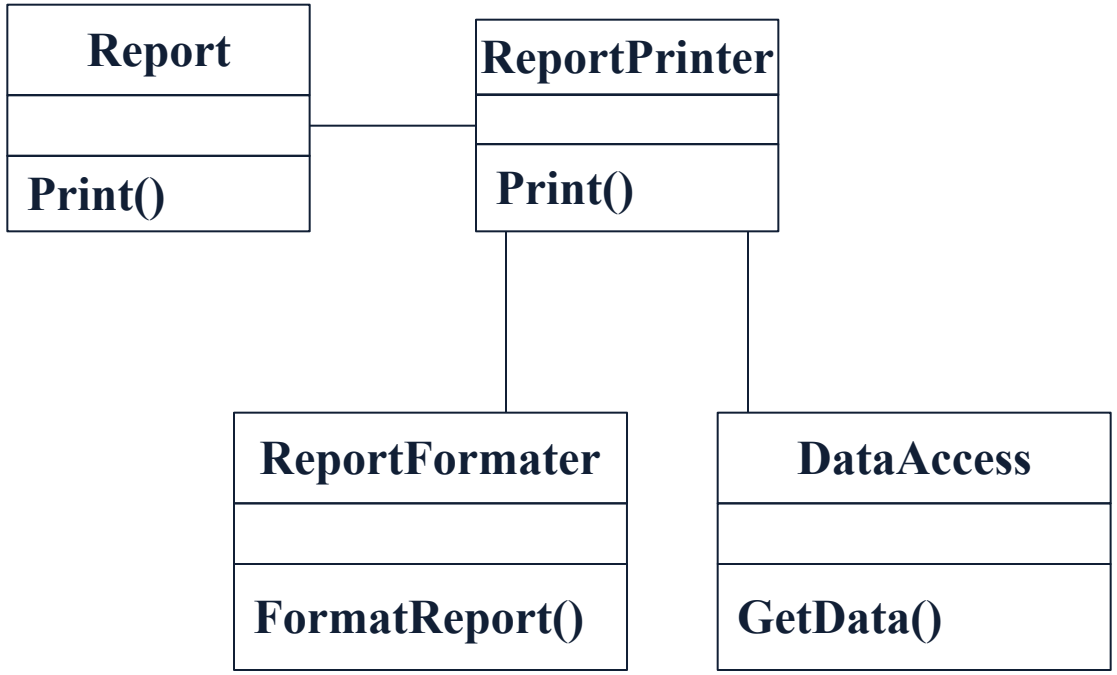
```
namespace Dimecasts.SOLID
{
    class Report
    {
        private IList<ReportDataElement> GetData()
        {
            Console.WriteLine("\nGetting Data...");
            return new List<ReportDataElement>() { new Repo
        }

        private void FormatReport()
        {
            Console.WriteLine("\nFormatting Report...");
        }

        public void Print()
        {
            GetData();
            FormatReport();
            Console.WriteLine("\nPrinting Report...");
        }
    }
}
```

Coesione – Example V2





Coesione – Example V2

```
public class ReportFormatter
{
    public void FormatReport()
    {
        Console.WriteLine("\nFormatting Report...");
    }
}
```

```
public class DataAccess
{
    public IList<ReportDataElement> GetData()
    {
        Console.WriteLine("\nGetting Data...");
        return new List<ReportDataElement>() { new ReportDataElement()
    }
}
```

Coesione – Example V2

```
public class ReportPrinter
{
    public void Print()
    {
        DataAccess dataAccess = new DataAccess();
        ReportFormatter reportFormatter = new ReportFormatter();
        dataAccess.GetData();
        reportFormatter.FormatReport();
        Console.WriteLine("\nPrinting Report...");
    }
}
```

```
class Report
{

    public void Print()
    {
        ReportPrinter reportPrinter = new ReportPrinter();
        reportPrinter.Print();
    }
}
```

Benefits of Cohesion

- Riutilizzo - Se tutte le tue classi seguono l'SRP, è più probabile che tu ne riutilizzi alcune
- Chiarezza - Il tuo codice è più pulito poiché le tue lezioni non fanno cose impreviste
- cose
- Denominazione - Poiché tutte le tue classi hanno un'unica responsabilità, scegliere un buon nome è facile
- Leggibilità - Grazie alla chiarezza, ai nomi migliori e ai file più brevi, la leggibilità è notevolmente migliorata.

Coesione

- Indicatori di un’alta coesione (è il nostro obiettivo)
  - Una classe ha delle resposabilità moderate, limitate ad una singola area funzionale
  - Collabora con altre classi per completare dei tasks
  - Ha un numero limitato di metodi, cioè di funzionalità altamente legate tra loro
  - Tutti i metodi sembrano appartenere ad uno stesso insieme, con un obiettivo globale simile
  - La classe è facile da comprendere

Accoppiamento

- L’accoppiamento è una misura su quanto fortemente una classe è connessa con /ha conoscenza di/ si basa su altre classi.
- Due o più classi si dicono accoppiate quando è impossibile riusare una senza dover riusare anche la/le altra/e
- Se due classi dipendono strettamente e per molti dettagli l’una dall’altra, diciamo che sono fortemente accoppiate.
  - In caso contrario diciamo che sono debolmente accoppiate
- Per un codice di qualità dobbiamo puntare ad un basso accoppiamento

Basso Accoppiamento

- Un basso accoppiamento è una proprietà desiderabile del codice, poiché permette di:
  - Capire il codice di una classe senza leggere i dettagli delle altre
  - Modificare una classe senza che le modifiche comportino conseguenze sulle

altre classi

    - Riusare facilmente una classe senza dover importare anche altre classi
  - Un basso accoppiamento migliora la manutenibilità del software

Basso Accoppiamento

- Un certo livello di accoppiamento è comunque insito nel concetto di scambio di messaggio del paradigma O-O
  - E’ quindi necessario per il funzionamento del sistema
  - Deve essere limitato ove possibile
- Linea Guida: definire le responsabilità delle classi in modo da ottenere un basso accoppiamento
  - Legge di Demetra (vedi slides successive)

Tipi di Accoppiamento in O-O

- Date 2 classi X e Y:
  - X ha un attributo di tipo Y
  - X ha un metodo che possiede un elemento (es: parametro, variabile locale, etc…) di tipo Y
  - X è una sottoclasse (eventualmente indiretta) di Y
  - X implementa un'interfaccia di tipo Y

- Lo scenario 3 è quello che porta al massimo accoppiamento in assoluto.

Esempio

```
class Traveler
{
Car c=new Car(); void
startJourney()
{
c.move();
}
}

class Car
{
void move()
{
// logic...
}
}
```

- Cosa succede se vogliamo riusare la nostra classe Traveler con un altro mezzo di trasporto che non sia Car?
- Cosa succede se vogliamo cambiare i metodi di Car?
- Traveler ha un attributo di tipo Car, e quindi un forte accoppiamento.

Esempio (cont.)

```
•class Traveler
•{
•Vehicle v;
•public void setV(Vehicle v)
•{
•this.v = v;
•}

•void startJourney()
•{
•v.move();
•}
•}

•Interface Vehicle
•{
•void move();
•}
```

```
class Car implements Vehicle
{
public void move()
{
// logic
}
}
```

```
class Bike implements Vehicle
{
public void move()
{
// logic
}
}
```

Spiegazione

- Nell'esempio, l'introduzione di un'interfaccia Vehicle ha spezzato completamente l'accoppiamento tra Traveler e Car.
- Conseguenze:
  - Traveler è ora accoppiato con un'interfaccia.
  - Tutte le implementazioni dell'interfaccia funzionano con Traveler, senza richiederne modifiche al codice
  - Sarà possibile usare in futuro implementazioni di Vehicle non ancora realizzate oggi (anticipare il cambiamento)
  - E' necessaria un'entità esterna che faccia l'inject della reale implementazione dell'interfaccia
  - E' aumentata la complessità del codice

Thinking about Interfaces

- Un membro del team di sviluppo Java che conosceva le interfacce ha deciso di renderle una parte importante di Java SDK. Ci sono centinaia di esempi nell'SDK J2SE del modo corretto di utilizzare le interfacce.
- Di seguito sono riportati due vantaggi principali derivanti dall'utilizzo delle interfacce:
- Un'interfaccia fornisce un mezzo per stabilire uno standard. Definisce un contratto che promuove il riuso. Se un oggetto implementa un'interfaccia, allora quell'oggetto promette di essere conforme a uno standard. Un oggetto che utilizza un altro oggetto è chiamato consumer. Un'interfaccia è un contratto tra un oggetto e il relativo consumatore.
- Un'interfaccia fornisce anche un livello di astrazione che rende i programmi più facili da capire. Le interfacce consentono agli sviluppatori di iniziare a parlare del modo generale in cui si comporta il codice senza dover entrare in molte specifiche dettagliate.

L'esempio del Paperboy

```
public class Customer {
 private String firstName;
 private String lastName;
 private Wallet myWallet;
 public String getFirstName(){
return firstName;
}
public String getLastName(){
 return lastName;
}
public Wallet getWallet(){
 return myWallet;
}
}
```

- public class Wallet { private float value;
- public float getTotalMoney() { return value;
- }
- public void setTotalMoney(float
- newValue){
- value = newValue;
- }
- public void addMoney(float deposit) { value += deposit;
- }
- public void subtractMoney(float debit)
- {
- value -= debit;
- }
- }

L'esempio del Paperboy

```
code from some method inside the Paperboy class...
payment = 2.00; // “I want my two dollars!”

   Wallet theWallet = myCustomer.getWallet(); if
      (theWallet.getTotalMoney() > payment) {
         theWallet.subtractMoney(payment);
                     } else {
        // come back later and get my money
                        }
```

Why Is This Bad?

- Traduciamo ciò che il codice sta effettivamente facendo in linguaggio reale:
- A quanto pare, quando il ragazzo dei giornali si ferma e chiede il pagamento, il cliente sta per lasciare che il ragazzo dei giornali tiri fuori il portafoglio dalla tasca posteriore e tiri fuori due dollari.
- Non so voi, ma raramente lascio che qualcuno gestisca il mio portafoglio. Ci sono una serie di problemi del "mondo reale" con questo, per non parlare del fatto che ci fidiamo del ragazzo dei giornali per essere onesto e prendere semplicemente ciò che gli è dovuto.
- Se il nostro futuro oggetto Wallet contiene carte di credito, il paperboy ha accesso anche a quelle... Ma il problema di base è che "il ragazzo dei giornali è esposto a più informazioni di quelle di cui ha bisogno".
- Questo è un concetto importante... La classe del "Paperboy" ora "sa" che il cliente ha un portafoglio e può manipolarlo. Quando compiliamo la classe Paperboy, avrà bisogno della classe Customer e della classe Wallet. Queste tre classi sono ora "strettamente accoppiate". Se cambiamo la classe Wallet, potremmo dover apportare modifiche a entrambe le altre classi.

Why Is This Bad?

C'è un altro problema classico che questo può creare... Cosa succede se il portafoglio del Cliente è stato rubato? Forse la scorsa notte un ladro ha preso la tasca del nostro esempio, e qualcun altro nel nostro team di sviluppo software ha deciso che un buon modo per modellare questo sarebbe stato impostare il portafoglio su null, in questo modo:

```
victim.setWallet(null);
```

- Sembra un'ipotesi ragionevole... Né la documentazione né il codice impongono alcun valore obbligatorio per il portafoglio, e c'è una certa "eleganza" nell'usare null per questa condizione. Ma cosa succede al nostro Paperboy? Tornando al codice...
- Il codice presuppone che ci sarà un portafoglio! Il nostro paperboy otterrà un'eccezione di runtime per la chiamata di un metodo su un puntatore nullo.
- Potremmo risolvere questo problema controllando la presenza di 'null' sul portafoglio prima di chiamare qualsiasi metodo su di esso, ma questo inizia a ingombrare il codice del paperboy ... La nostra descrizione in lingua reale sta diventando ancora peggiore...
- "Se il mio cliente ha un portafoglio, allora guarda quanto ha... se può pagarmi, prendilo"...

Improving The Original Code

```
public class Customer {
 private String firstName;
 private String lastName;
 private Wallet myWallet;
 public String getFirstName(){
return firstName;
}
public String getLastName(){
 return lastName;
}
```

```
  public float getPayment(float bill) {
   if (myWallet != null) {
  if (myWallet.getTotalMoney() > bill)
  {
  theWallet.subtractMoney(payment);
   return payment;
  }
  }
  }
}
```

The Customer no longer has a 'getWallet()' method,
but it does have a getPayment() method.

Improving the original code

```
// code from some method inside the Paperboy class...
    payment = 2.00;  // “I want my two dollars!”
    paidAmount = myCustomer.getPayment(payment);
if (paidAmount == payment) {
// say thank you and give customer a receipt
} else {
// come back later and get my money
}
```

Why is this better?

- Questa è una domanda lecita... Qualcuno a cui è piaciuto molto il primo pezzo di codice potrebbe obiettare che abbiamo solo reso il Cliente un oggetto più complesso. Questa è un'osservazione giusta, ma si parlerà in seguito degli svantaggi.
- Il primo motivo per cui questo è migliore è perché modella meglio lo scenario del mondo reale... Il codice Paperboy ora "chiede" al cliente un pagamento. Il paperboy non ha accesso diretto al portafoglio.
- Il secondo motivo per cui questo è migliore è perché la classe Wallet ora può cambiare, e il ragazzo dei giornali è completamente isolato da quel cambiamento.
- Se l'interfaccia del Wallet dovesse cambiare, il Cliente dovrebbe essere aggiornato, ma
- Questo è tutto...
- Finché l'interfaccia con il Cliente rimane la stessa, nessuno dei clienti del Cliente si preoccuperà di aver ricevuto un nuovo Portafoglio.
- Il codice sarà più gestibile, perché le modifiche non si "propagheranno" attraverso un progetto di grandi dimensioni.

Why is this better?

- La terza risposta, e probabilmente la più "orientata agli oggetti" è che ora siamo liberi di modificare l'implementazione di 'getPayment()'. Nel primo esempio, abbiamo ipotizzato che il Cliente avrebbe avuto un portafoglio. Ciò ha portato all'eccezione del puntatore nullo di cui abbiamo parlato. Nel mondo reale, però, quando il ragazzo di carta si presenta alla porta, il nostro cliente può effettivamente prendere i due dollari da un barattolo di spiccioli, cercare tra i cuscini del suo divano o prenderli in prestito dal suo coinquilino.
- Tutto questo è "Business Logic", e non interessa al ragazzo di carta... Tutto questo potrebbe essere implementato all'interno del metodo getPayment() e potrebbe cambiare in futuro, senza alcuna modifica al codice del paper boy.