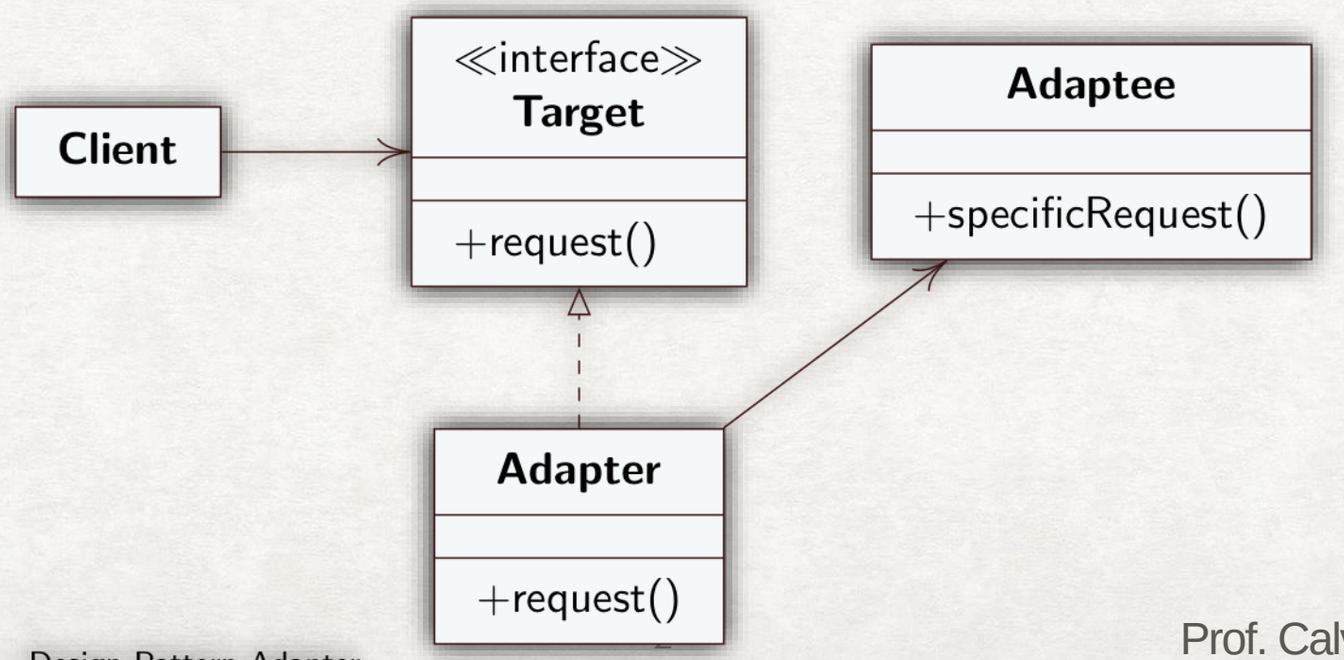
Design pattern Adapter

- Intento: Converte l'interfaccia di una classe in un'altra interfaccia che i client si aspettano. Adapter permette ad alcune classi di interagire, eliminando il problema di interfacce incompatibili
- Problema
  - Alcune volte una classe di una libreria non può essere usata poiché incompatibile con l'interfaccia che si aspetta l'applicazione. Ovvero nome metodo, parametri, tipo parametri, di chiamate all'interno dell'applicazione non sono corrispondenti a quelli offerti da una classe di libreria
- Non è possibile cambiare l'interfaccia della libreria, poiché non si ha il sorgente (comunque non conviene cambiarla)
- Non è possibile cambiare l'applicazione (il chiamante), e si può voler cambiare quale metodo chiamare, senza renderlo noto al chiamante

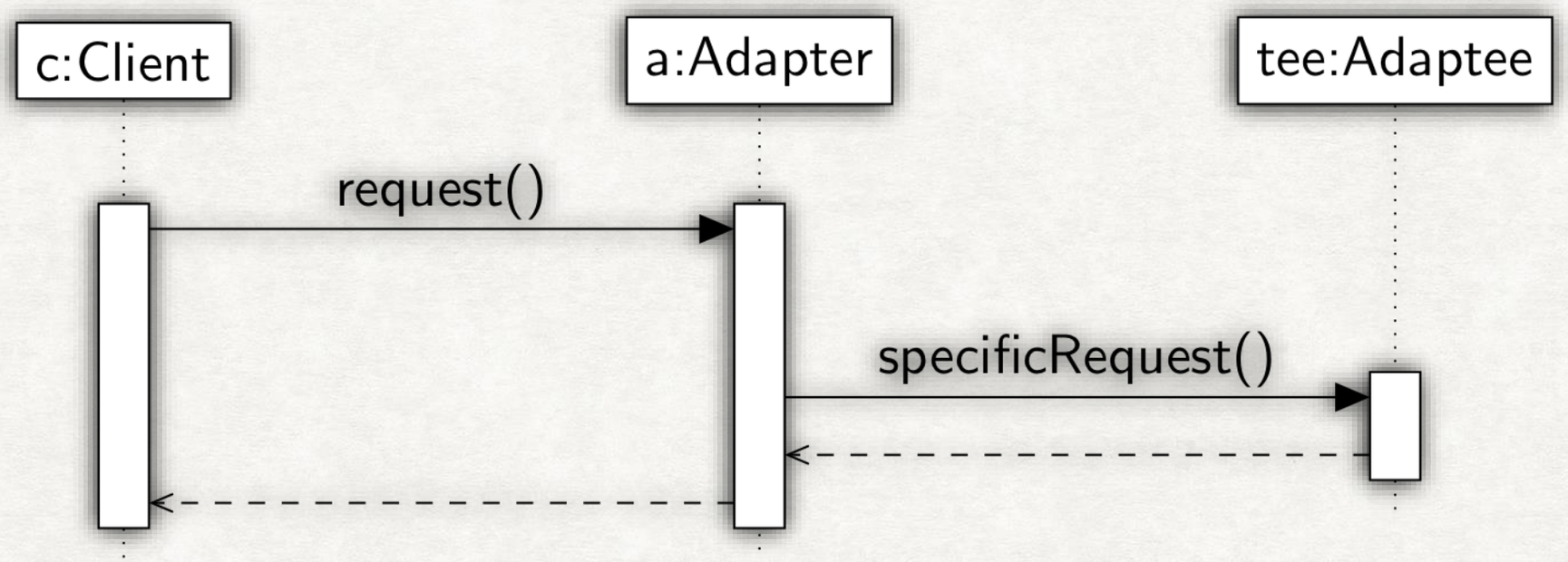
Design pattern Adapter

- Soluzione Object Adapter
  - Target è l'interfaccia che il chiamante si aspetta
  - Client usa oggetti che sono conformi all'interfaccia Target
  - Adaptee è l'oggetto di libreria che necessita l'adattamento
  - Adapter converte, ovvero adatta, la chiamata che fa una classe client all'interfaccia della classe di libreria. Il chiamante (Client) usa l'Adapter come se fosse l'oggetto di libreria. La classe con ruolo Adapter implementa Target, tiene il riferimento all'oggetto di libreria (Adaptee) e sa come invocarlo, ovvero chiama i metodi di Adaptee



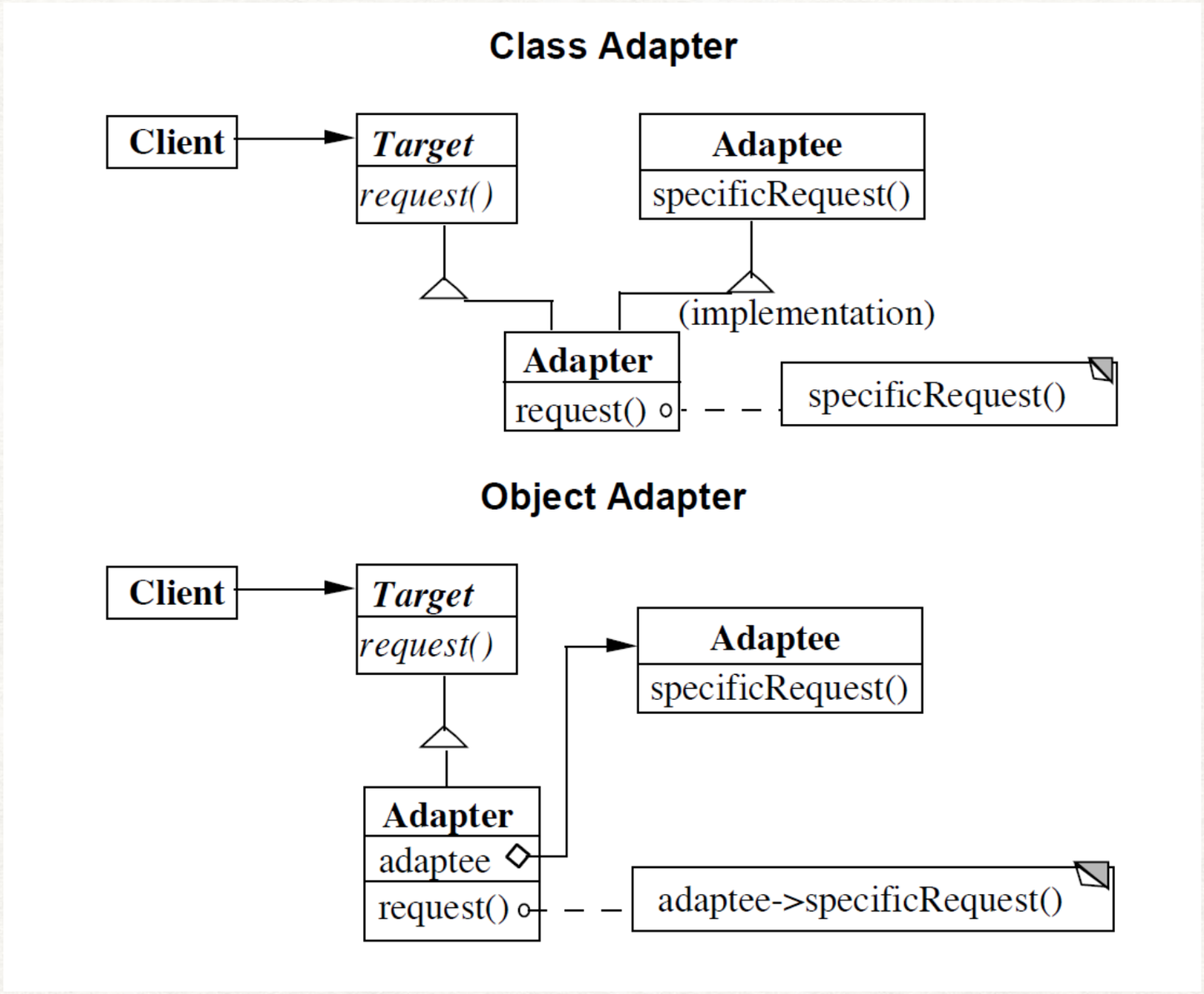
Design pattern Adapter

- Diagramma di sequenza della soluzione Object Adapter



Object adapter

- Ci sono due forme di Adapter
  - Class Adapter
  - Object Adapter



- Quando vuoi riusare molte classi esistenti è poco pratico adattare le interfacce sottoclassandole tutte
- Un ***object adapter*** può adeguare l'interfaccia di varie classi a quella del client senza sottoclassarle

Class Adapter Example

```
class OldSquarePeg {
      public void squarePegOperation() { do something }
}

Interface RoundPeg {
      public void roundPegOperation();
}

class PegAdapter extends OldSquarePeg, implements RoundPeg {
      public void roundPegOperation() {
                  //add some corners;
                  squarePegOperation();
      }
}

void clientMethod() {
            RoundPeg aPeg = new PegAdapter();
            aPeg.roundPegOperation();
}
```

Object Adapter Example

```
class OldSquarePeg{
    public void squarePegOperation() { do something }
}

interface RoundPeg {
    public void roundPegOperation();
}

class PegAdapter implements RoundPeg {
    private OldSquarePeg square;

    public PegAdapter() { square = new OldSquarePeg; }

    void roundPegOperation() {
        //add some corners;
        square.squarePegOperation();
    }
}
```

Class/Object adapter

Un class adapter usa l'ereditarietà, per cui:

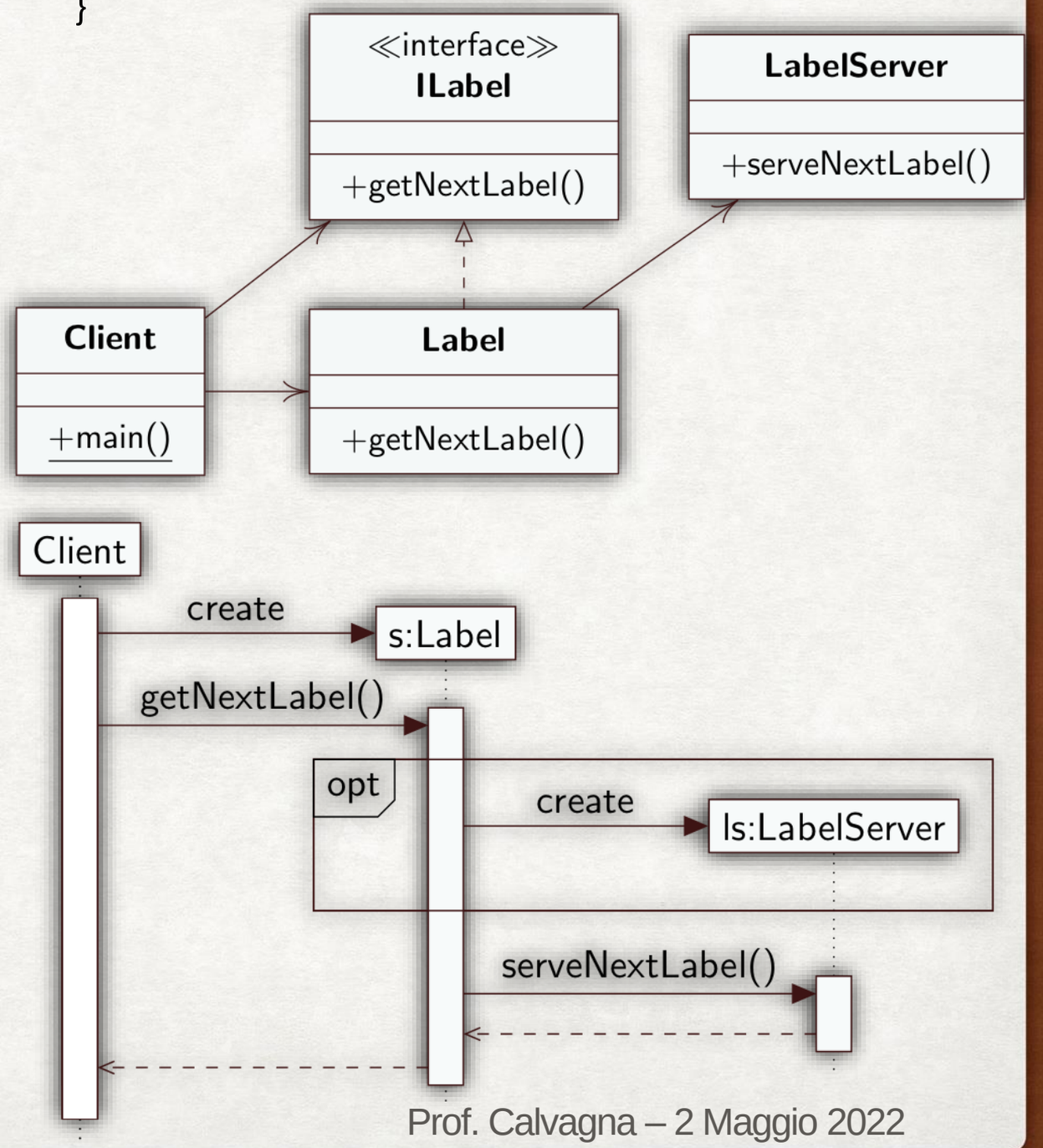
- adatta solo una classe a tutti i sui ascendenti, non alle sottoclassi
- Consente all'adapter di ridefinire parte dell'implementazione della classe *adattata (override)*
- Non introduce indirezione

Un object adapter usa la composizione, per cui :

- Consente di usare un solo adapter per più *adattati*
- Rende problematico ridefinire il comportamento degli adattati perchè…
- L'adapter può non sapere esattamente con quale classe specifica sta lavorando

```java
public interface ILabel { // Target
  public String getNextLabel();
}

// Adapter
public class Label implements ILabel {
  private LabelServer ls;
  private String p;
  public Label(String prefix) {
    p = prefix;
  }
  public String getNextLabel() {
    // lazy initialisation
    if (null == ls)
                    ls = new LabelServer(p);
    return ls.serveNextLabel();
  }
}

public class Client {
  public static void main(String args[]) {
    ILabel s = new Label("LAB");
    String l = s.getNextLabel();
    if (l.equals("LAB1"))
              System.out.println("Test 1:Passed");
    else
      System.out.println("Test1:Failed");
  }
}
```

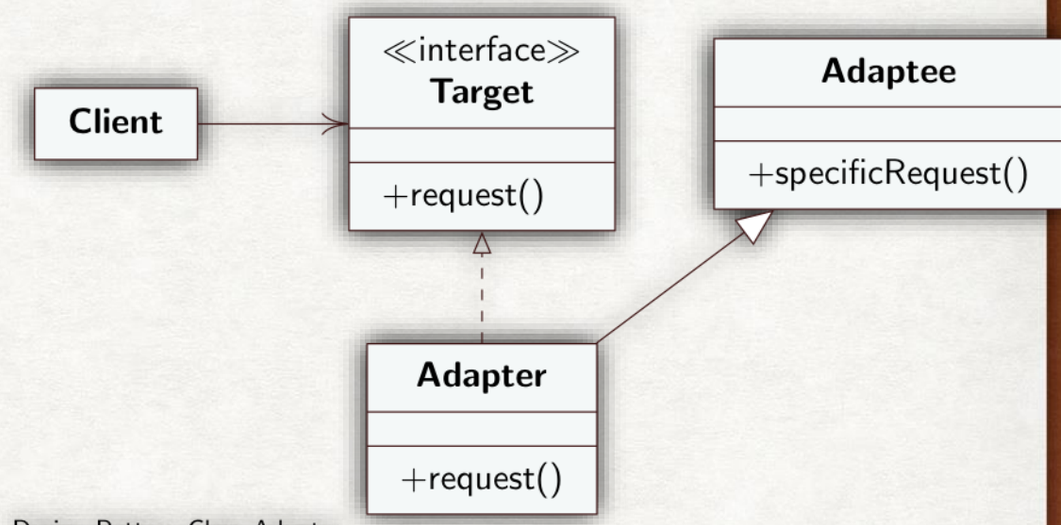
```java
public class LabelServer { // Adaptee
  private int labelNum = 1;
  private String labelPrefix;
  public LabelServer(String prefix) {
    labelPrefix = prefix;
  }
  public String serveNextLabel() {
    return labelPrefix + labelNum++;
  }
}
```



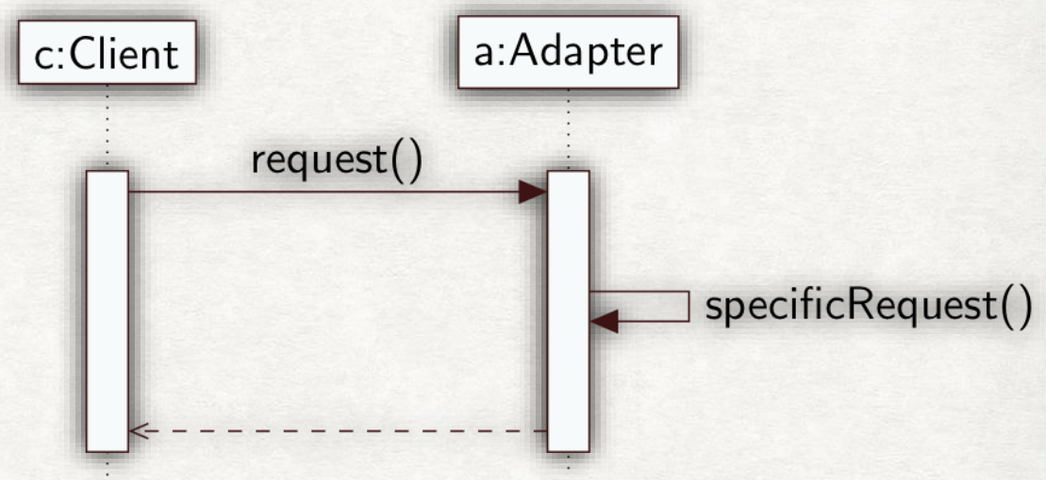
Design pattern Adapter

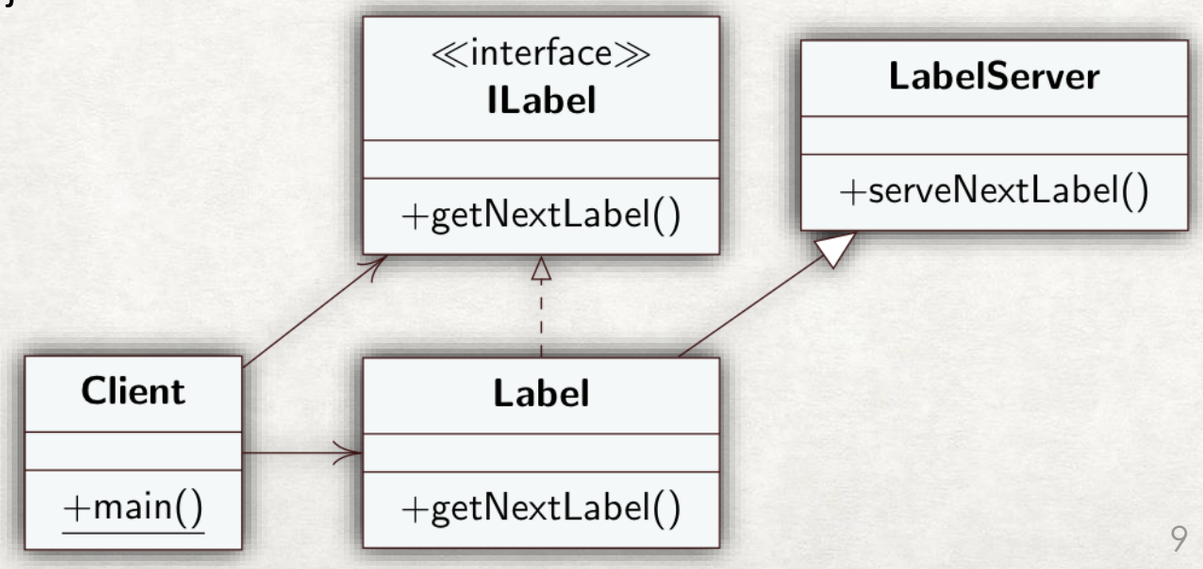
- Soluzione Class Adapter
  - La classe con il ruolo Adapter è sottoclasse di Adaptee

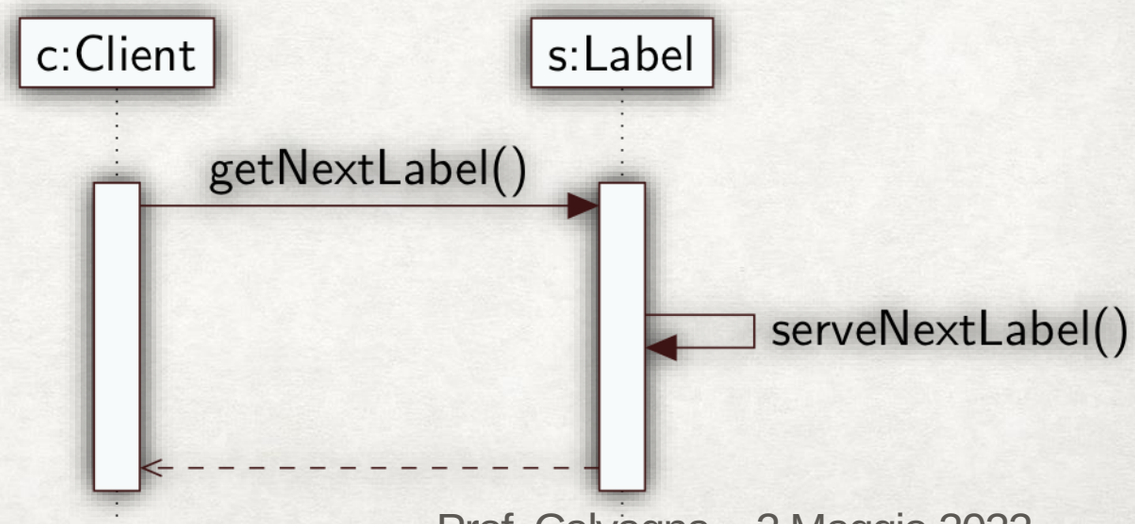
```
public class Label extends LabelServer
        implements ILabel { // Adapter

    public Label(String prefix) {
        super(prefix);
    }

    public String getNextLabel() {
        return serveNextLabel();
    }
}
```



Design Pattern Class Adapter







- Esempio applicazione completa Label Service
- https://www.dmi.unict.it/tramonta/se/oop/appLabel.html

Design pattern Adapter

- Variante design pattern Adapter a due vie
  - Definizione: la classe con ruolo Adapter fornisce l'interfaccia di Target e l'interfaccia di Adaptee
  - Realizzazione: la soluzione Class Adapter è un Adapter a due vie
- Conseguenze del design pattern Adapter
  - Client e classe di libreria Adaptee rimangono indipendenti. Il ruolo Adapter può cambiare il comportamento dell'Adaptee
  - Può aggiungere test di precondizioni e postcondizioni [Precondizioni: cosa si deve soddisfare prima di eseguire. Postcondizioni: cosa è verificato se tutto è andato bene]
  - L'Object Adapter può implementare la tecnica di Lazy Initialisation (si aspetta che avvenga un'invocazione a un metodo di Adaptee prima di istanziarlo)
  - Il design pattern Adapter aggiunge un livello di indirettezza. Ogni invocazione del Client ne scatena un'altra fatta dal ruolo Adapter. Possibile rallentamento (trascurabile), e codice più difficile da comprendere

FACADE

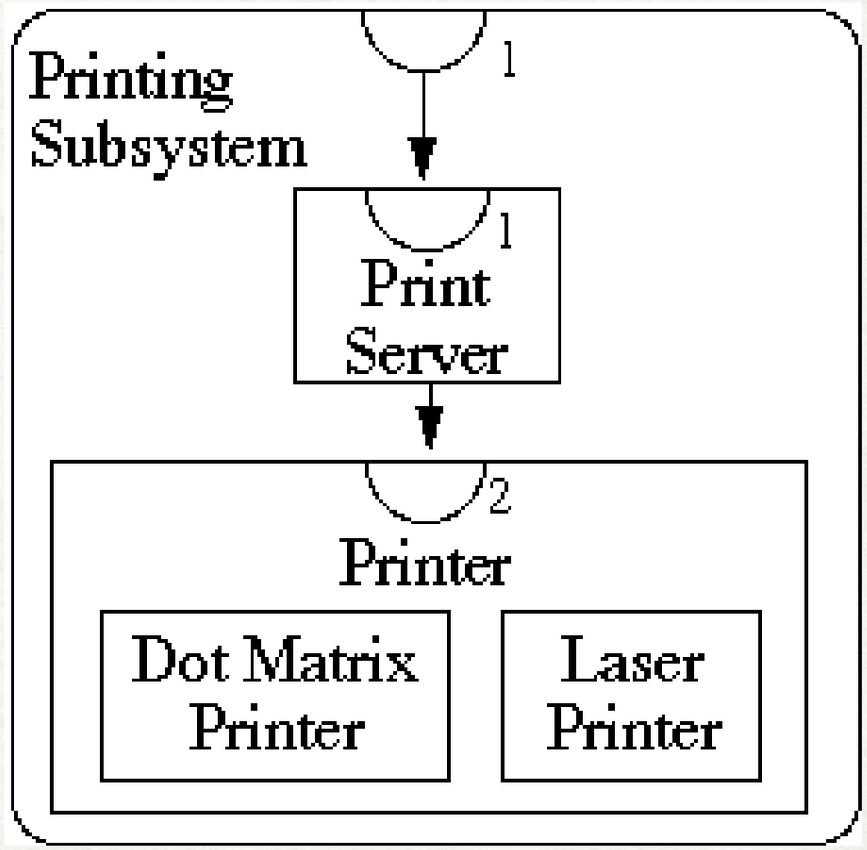
Sottosistema
Sono gruppi di classi, ed eventualmente altri sottosistemi, che collaborano tra loro per supportare la realizzazione di un set di compiti specifici

Non c'è differenza concettuale tra il raggruppamento di responsabilità in una classe e quello in un sottosistema di classi

E' solo una questione di scala

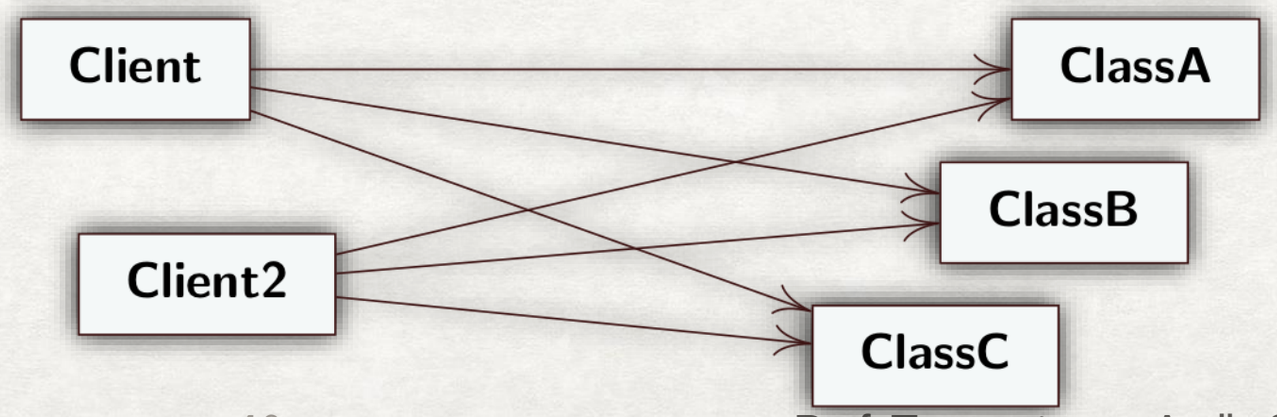
Un sottosistema deve essere una buona astrazione per una parte ben precisa e del sistema

Ci dovrebbe essere meno interazione (dipendenza) possible tra i diversi sottosistemi



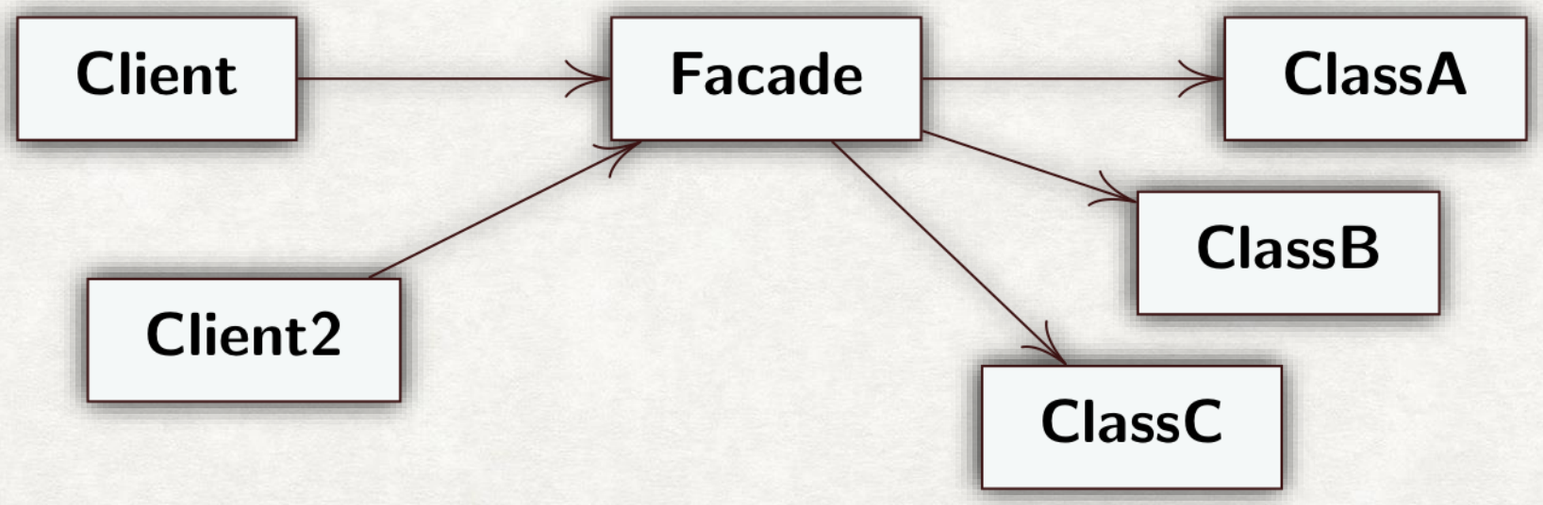
Design pattern Facade

- Intento: Fornire un'interfaccia unificata al posto di un insieme di interfacce in un sottosistema (consistente di un insieme di classi). Definire un'interfaccia di alto livello (semplificata) che rende il sottosistema più facile da usare
- Problema
  - Spesso si hanno tante classi che svolgono funzioni correlate e l'insieme delle interfacce può essere complesso
- Può essere difficile capire qual è l'interfaccia essenziale ai client per l'insieme di classi
- Si vogliono ridurre le comunicazioni e le dipendenze dirette fra i client ed il sottosistema



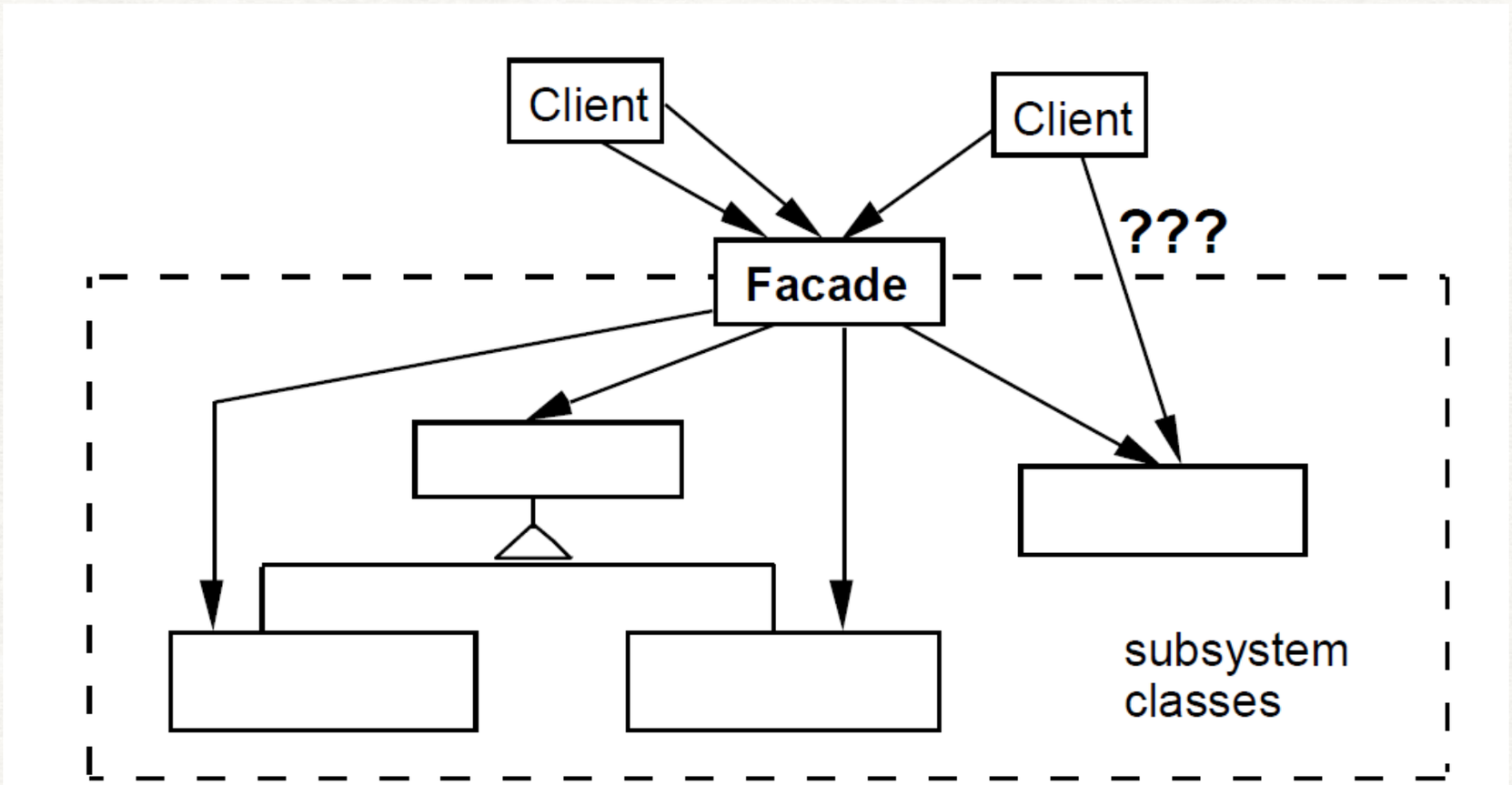
Design pattern Facade

- Soluzione
  - Facade fornisce un'unica interfaccia semplificata ai client e nasconde gli oggetti del sottosistema, questo riduce la complessità dell'interfaccia e quindi delle chiamate. Facade invoca i metodi degli oggetti che nasconde
  - Client interagisce solo con l'oggetto Facade



FAÇADE

Creo una classe che fa da facciata-interfaccia al sottosistema
I client si interfacciano col façade per interagire col sottosistema



In Java:
Default class Access Modifier

Private, Protected (*default?*) per i membri
No ***public***

Solo per Façade ***public*** class e/o membri

Modificatori di accesso in Java

- Solo *public* e *no-modifier* per le classi
- Tutti per i membri

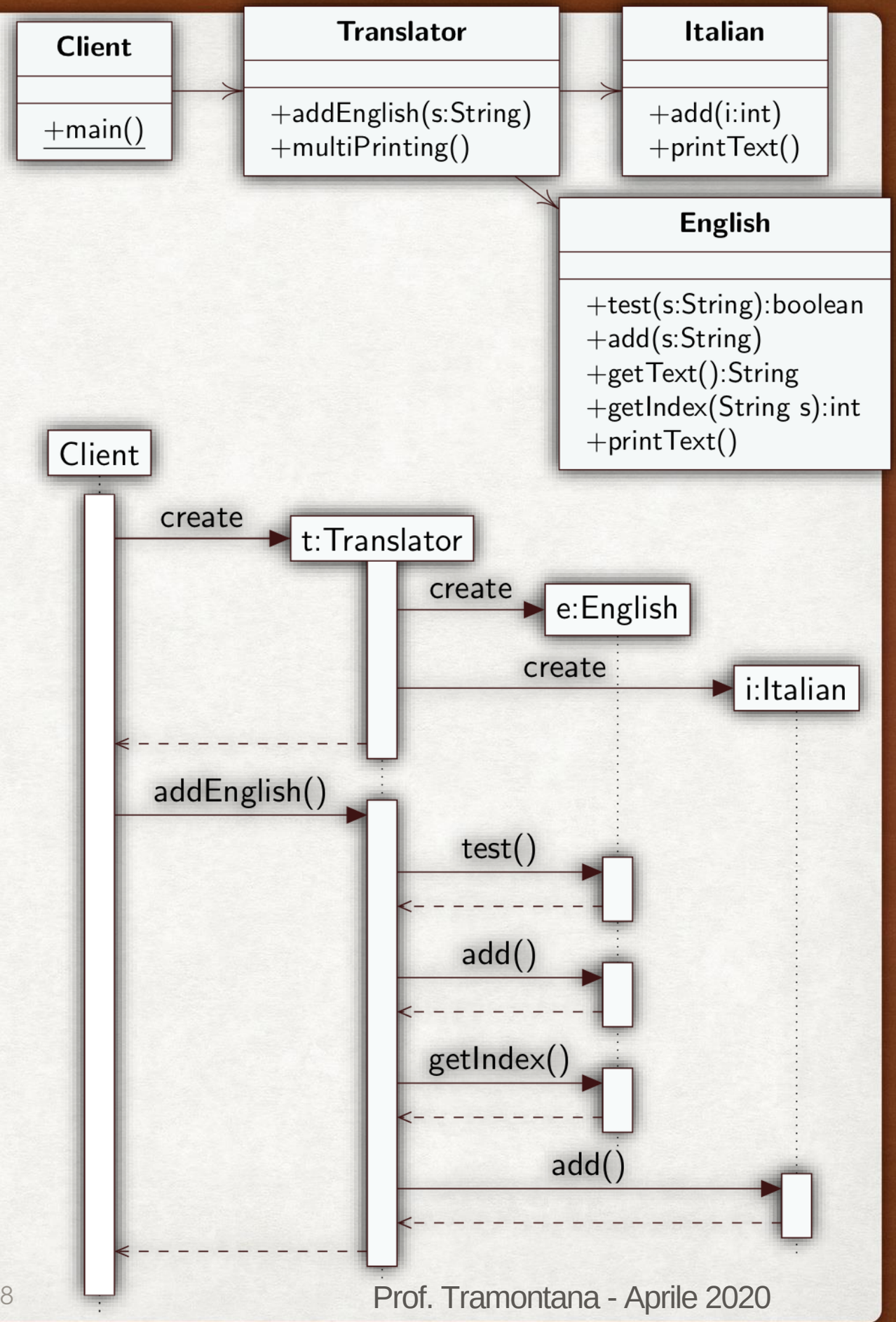
Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Design pattern Facade

- Conseguenze
  - Nasconde ai client l'implementazione del sottosistema
  - Promuove l'accoppiamento debole tra sottosistema e client
  - Riduce le dipendenze di compilazione in sistemi grandi. Se si cambia una classe del sottosistema, si può ricompilare la parte di sottosistema fino al facade, quindi non i vari client
  - Non previene l'uso di client più complessi, quando occorre, che accedono ad oggetti del sottosistema
- Implementazione
  - Per rendere gli oggetti del sottosistema non accessibili al client le corrispondenti classi possono essere annidate dentro la classe Façade
  - Non deve aggiungere funzionalità non presenti già nel sottosistema

```java
public class Client {
	public static void main(String args[]) {
		Translator t = new Translator();
		t.addEnglish("Hello");
		t.multiPrinting();
	}
}

public class Translator { // Ruolo Facade
	private English e = new English();
	private Italian i = new Italian();

	public void addEnglish(String s) {
		if (e.test(s)) {
			e.add(s);
			i.add(e.getIndex(s));
		}
	}

	public void multiPrinting() {
		System.out.print("Italiano:  ");
		i.printText();
		System.out.print("English:   ");
		e.printText();
	}
}
```



```java
public class English {
	private String text = " ";
	private List<String> d =
	Arrays.asList("Alright", "Hello",
		"Understood", "Yes");

	public boolean test(String s) {
		return d.contains(s);
	}

	public void add(String s) {
		text = text + " " + s;
	}

	public String getText() {
		return text;
	}

	public int getIndex(String s) {
		return d.indexOf(s);
	}

	public void printText() {
		System.out.println(text);
	}
}
```

```java
public class Italian {
	private String text = " ";
	private List<String> d =
	Arrays.asList("Va bene", "Ciao",
		"Capito", "Sì");

	public void add(int i) {
		text = text + " " + d.get(i);
	}

	public void printText() {
		System.out.println(text);
	}
}
```

- Esempio applicazione completa Libreria
- https://www.dmi.unict.it/tramonta/se/oop/appLibri.html