Design Pattern Proxy

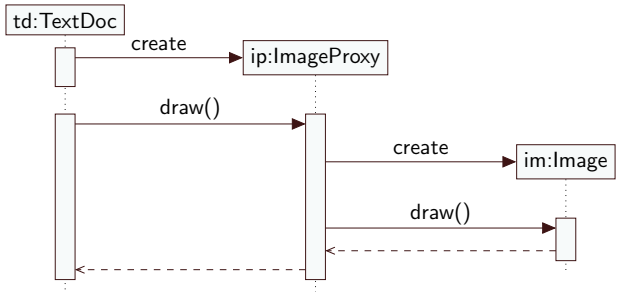
- **Intento**: Definire un sostituto o surrogato per un oggetto che controlla gli accessi all'oggetto target. I client comunicano con il surrogato anziché comunicare con l'oggetto target
- Esempi di uso del design pattern Proxy
  - All'apertura di un documento si vogliono ridurre i tempi di creazione di oggetti che contengono immagini, si usa un surrogato per le immagini non visibili
  - Si vuol controllare (ed in alcuni casi permettere) l'accesso ad un oggetto
  - Si vuol rendere più facile l'accesso ad un oggetto remoto

Design Pattern Proxy

- **Motivazione**: l'accesso all'oggetto target dovrebbe essere trasparente e semplice per il client
  - Ovvero, per l'es. dell'apertura documento, si vuol nascondere al client che l'immagine sia stata creata solo quando serve, così da non complicare gli oggetti client
  - Il client non deve cambiare il modo in cui chiama gli oggetti che usa

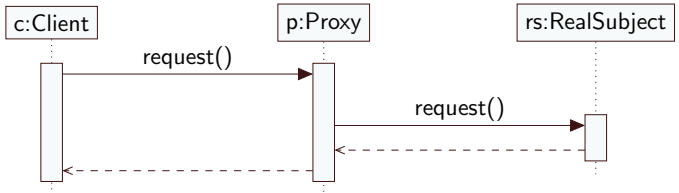
Proxy

- **Soluzione**: usare un oggetto Proxy che si frappone tra utilizzatori ed oggetto target. Il Proxy offre l'interfaccia dell'oggetto target ed aggiunge computazioni utili prima e dopo la chiamata (es. controllo dell'accesso)
- Es., il Proxy `ImageProxy` crea l'istanza di `Image` solo quando l'editor `TextDoc` invoca `draw()`



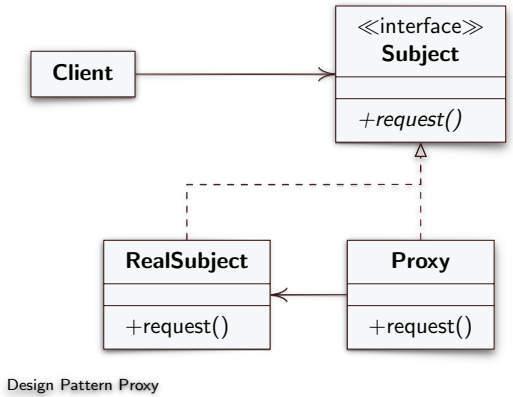
Proxy

- Componenti
  - Client usa il Proxy per accedere ad un certo servizio
  - RealSubject è l'oggetto target, quello che fornisce un servizio; è rappresentato dal Proxy
  - Proxy: (1) contiene il riferimento che permette l'accesso al RealSubject; (2) fornisce la stessa interfaccia del RealSubject; (3) fornisce l'accesso corretto al RealSubject
  - Subject definisce un'interfaccia per RealSubject e Proxy così che Proxy può essere usato al posto di RealSubject



Proxy

- Diagramma UML delle classi



Design Pattern Proxy

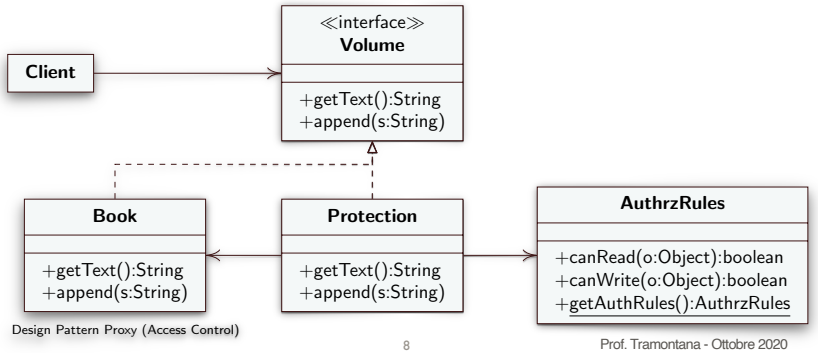

```java
public interface Subject {
   public void request();
}

public class RealSubject implements Subject {
   public void request() {
      // implements some appropriate service
   }
}

public class Proxy implements Subject {
   private RealSubject rs = null;

   public void request() {
      if (rs == null)
         rs = new RealSubject();
      // here some code that implements access control, caching,
      // synchronisation, remote access, etc.
      rs.request();
   }
}

public class Client {
   public static void main(String[] args) {
      Subject s = new Proxy();
      s.request();
   }
}
```


Protection Proxy

- Si ha bisogno di implementare politiche di protezione degli accessi a certi oggetti (o classi), quindi ai dati e alle risorse che questi rappresentano
- Es., si vuol concedere l'accesso, in scrittura o in lettura, dei campi delle istanze di una classe Book in base ad un insieme di regole, per es. identità dell'utente che ne fa richiesta, libro richiesto, parte del libro, operazione richiesta
  - La classe Book ha i metodi `getText()` e `append()`, rispettivamente per la lettura e la scrittura di dati dentro l'istanza di Book
  - Si inserisce una classe Proxy `Protection` fra richiedente e la classe Book per verificare i permessi di accesso. Il Proxy `Protection` accetterà gli accessi autorizzati e manderà le richieste all'istanza di Book, mentre non manderà le richieste non autorizzate

Protection Proxy

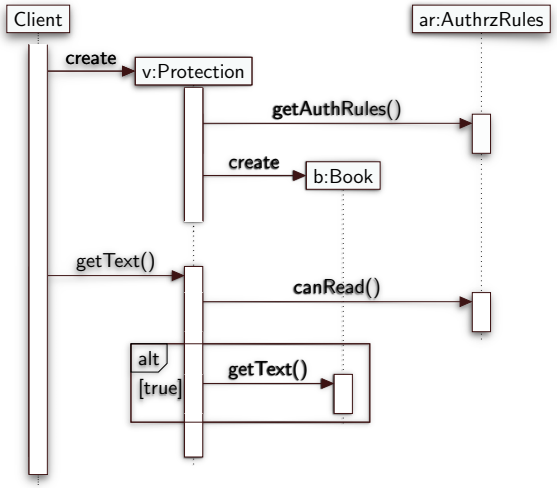
- Soluzione: l'interfaccia `Volume` è il Subject, la classe Book è il RealSubject, la classe `Protection` è il Proxy
  - La classe `AuthrzRules` contiene i permessi di accesso
- Diagramma UML delle classi



Design Pattern Proxy (Access Control)

Protection Proxy

- Diagramma UML di sequenza



Design Pattern Proxy (Access Control)


```java
public interface Volume { // Role Subject
   public String getText();
   public void append(String s);
}

public class Book implements Volume { // Role RealSubject
   private int current = 0;
   private List<String> content = new ArrayList<>();
   public Book() {
      // initialize book
   }
   @Override public String getText() {
      return content.get(current++);
   }
   @Override public void append(String s) {
      content.add(s);
   }
}

public class Client { // Role Client
   public static void main(String[] args) {
      Volume v = new Protection();
      System.out.println(v.getText());

      v.append("Hello world.");
   }
}
```



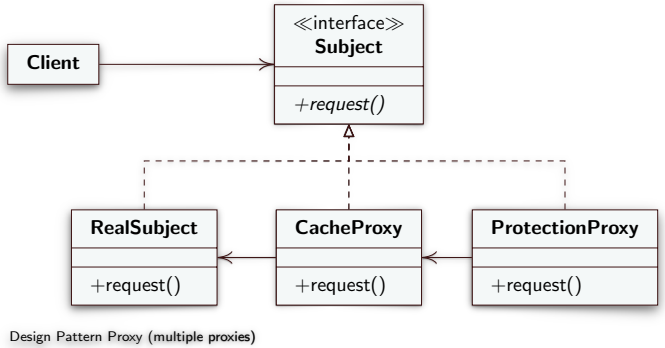
```java
public class Protection implements Volume {  // Role Proxy for pattern Proxy
    private Volume b;   // instance to guarded Book
    private AuthrzRules ar;
    public Protection() {
        ar = AuthrzRules.getAuthRules();
        b = new Book();
    }
    @Override public String getText() {  // security check
        if (ar.canRead(b)) return b.getText();
        System.out.println("Proxy: Read access has not been granted");
        return null;
    }
    @Override public void append(String s) { // security check
        if (ar.canWrite(b)) b.append(s);
        else System.out.println("Proxy: Write access has not been granted");
    }
}

public class AuthrzRules {
    private List<String> readpermits = new ArrayList<String>();
    private static AuthrzRules ar = new AuthrzRules();
    private AuthrzRules() {
        fillRules();
    }
    public static AuthrzRules getAuthRules() {
         return ar;
    }
    public boolean canRead(Object o) {
        return readpermits.contains(o.getClass().getName());
    }
    public void fillRules() {
        // select rules to be checked, e.g. according to user id, etc.
        readpermits.add("Book"); // in this example read permits are given to anyone
    }
}
```


Proxy

- Conseguenze
  - Il Proxy introduce una indirettezza
  - Ottimizza la creazione di oggetti onerosi
  - Permette di implementare, separatamente, politiche di protezione degli accessi (autenticazione e autorizzazione, o sincronizzazione tramite lock)
  - Un Cache Proxy può tenere risultati temporaneamente, e usato insieme al Remote Proxy può ottimizzare le prestazioni
  - Permette l'ottimizzazione copy-on-write, ovvero rimanda l'operazione di copia di un oggetto al momento in cui l'oggetto deve essere modificato, così da evitare il costo dell'operazione di copia se non necessaria o di sostenere il costo solo quando necessario
  - Nasconde il fatto che il RealSubject (nel Remote Proxy) risiede su un host remoto, quindi disaccoppia i client dalla locazione di oggetti remoti
  - Per ciascuna classe che si vuol *proteggere*, si implementa il corrispondente Proxy (la versione ad aspetti riduce il numero di Proxy necessari)

Proxy Multipli

- Posso avere vari Proxy per lo stesso RealSubject



Design Pattern Proxy (multiple proxies)