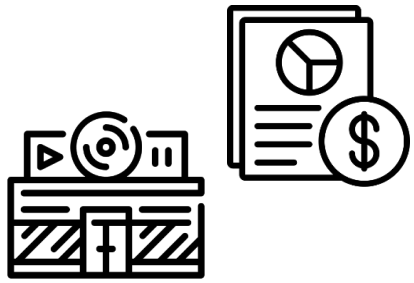
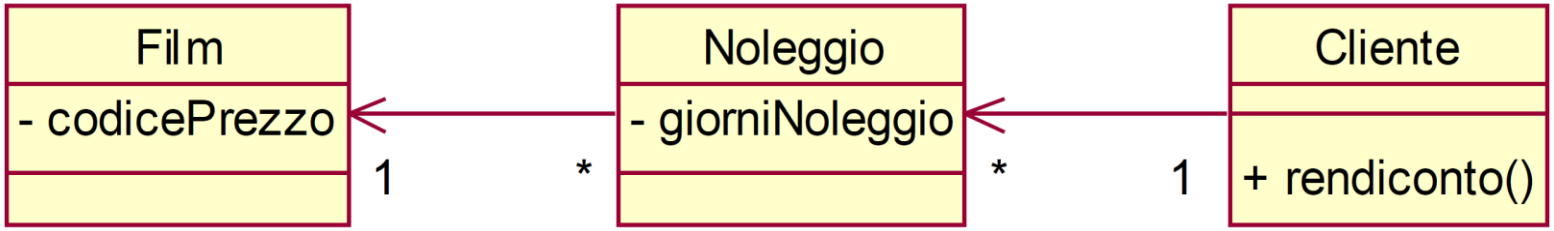
Laboratorio 4 - Refactoring

Fondamenti di Ingegneria del Software 2024-2025

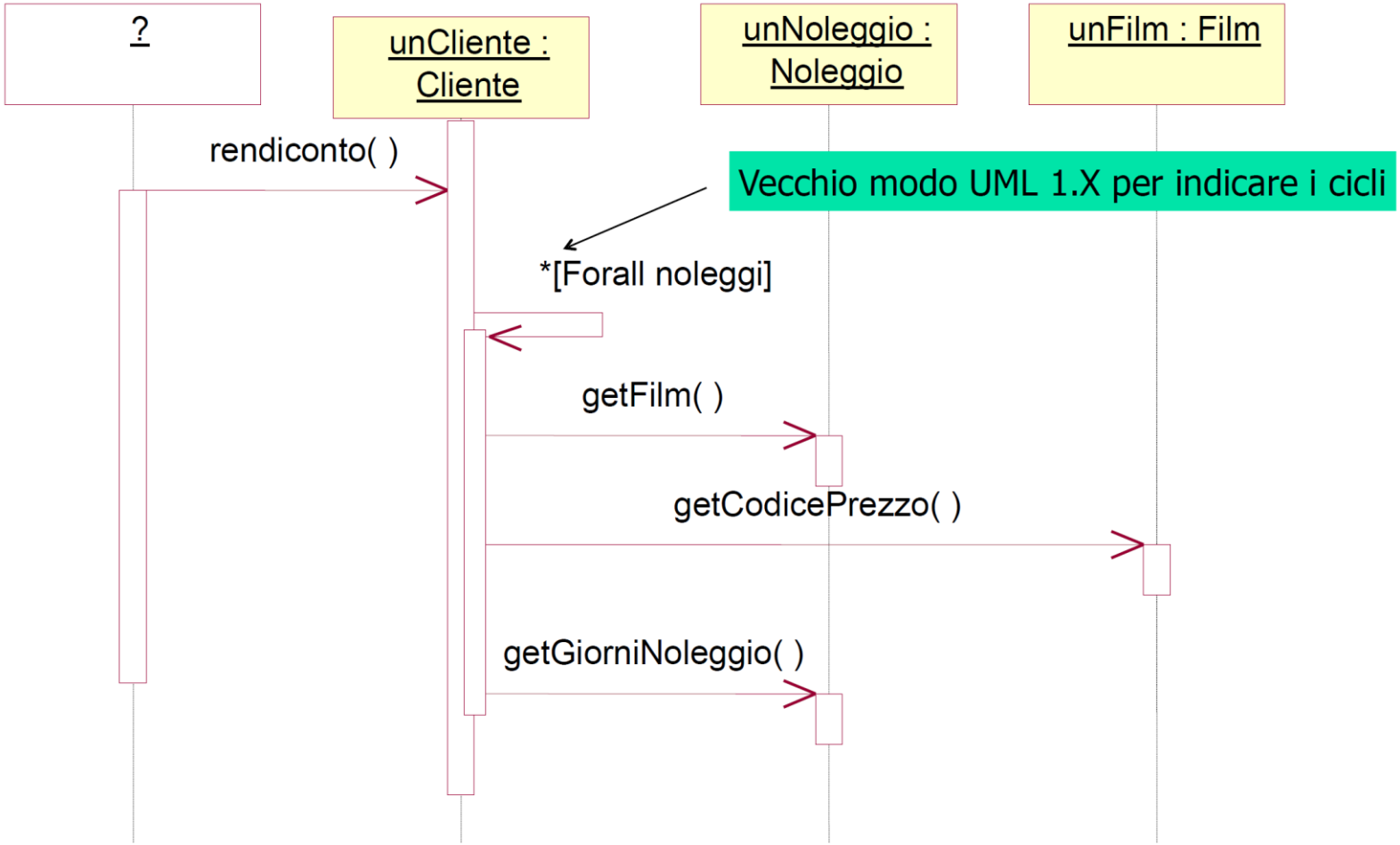
VideoStore: Class Diagram (v. 0)



- Software per calcolare il rendiconto di un cliente relativamente ai film noleggiati
- Tre tipologie di film determinano il prezzo del noleggio: **`Regolare`**, **`Bambini`**, **`Novità`**

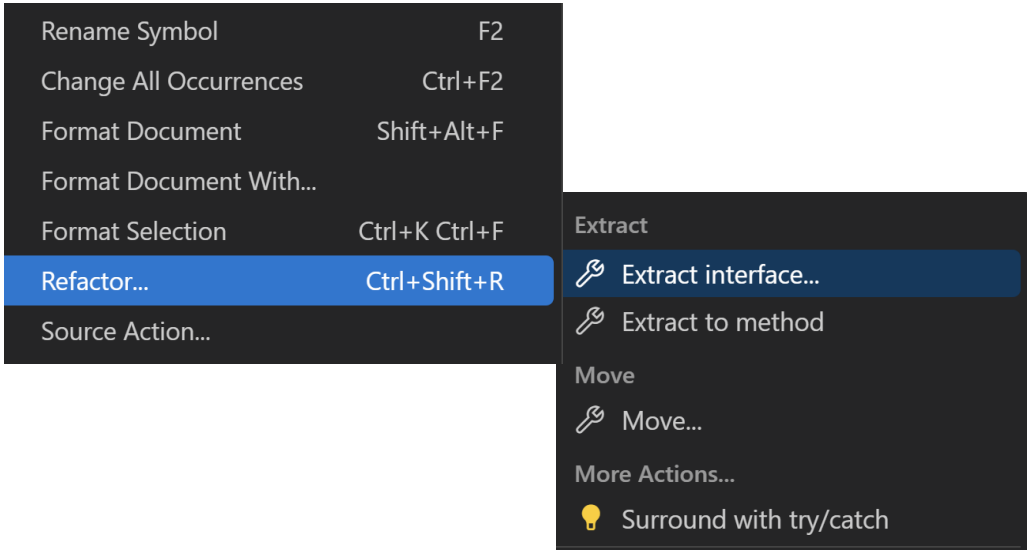


VIDEOSTORE: SEQ. DIAGRAM RENDICONTO (V. 0)



Refactoring in VS Code

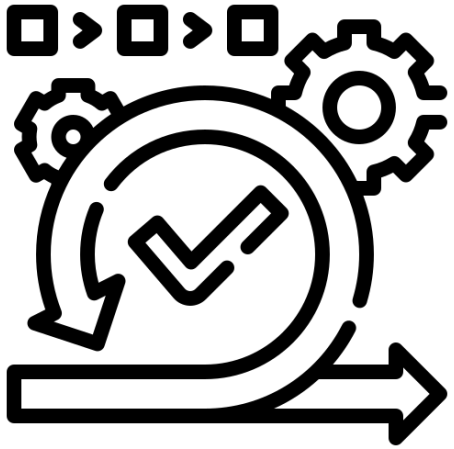
- In VS Code è possibile applicare refactoring automatici, anche non banali (es. **Extract Method**)
- Tuttavia, non sempre l'automazione produce il refactoring desiderato, quindi prestare attenzione!



Testare il Refactoring!

Verificheremo l'attività di refactoring mediante la classe di test **`TestRendiconto`** nel package **`test`**

- **Test OK** → il refactoring è stato implementato correttamente
- **Test KO** → il refactoring ha introdotto un problema



Refactoring #1: Extract Method

Obiettivo: ridurre la lunghezza di **`rendiconto()`** in **`Cliente`**

Passaggi:

- Selezionare il codice relativo allo switch/case in **`rendiconto()`**
- *Tasto destro* > *Refactor* > *Extract to method*
- Inserire **`ammontarePer`** come nome
- Rimuovere il parametro **`questoAmmontare`** dal nuovo metodo e sostituirlo con una variabile definita localmente

Effetto: verrà generato un nuovo metodo privato **`ammontarePer()`** in **`Cliente`**, invocato da **`rendiconto()`**, contenente la logica per il calcolo dell'ammontare di un noleggio

Rieseguire il caso di test dopo le modifiche!

Refactoring #2: Rename Variable

Obiettivo: dare una semantica più chiara al nome della variabile `questoAmmontare` in `ammontarePer()`

Passaggi:

- Selezionare la variabile `questoAmmontare` in `ammontarePer()`
- *Tasto destro* > *Rename Symbol*
- Inserire `risultato` come nome

Effetto: la variabile `questoAmmontare` cambierà nome e la modifica verrà propagata per ogni istanza del metodo

Rieseguire il caso di test dopo le modifiche!

Refactoring #3: Move Method

Obiettivo: assegnare la responsabilità del metodo **`ammontarePer()`** alla classe **`Noleggio`** anziché **`Cliente`** → usa informazioni di **`Noleggio`** e **`Film`** ma non di **`Cliente`**, quindi è nella classe sbagliata

Passaggi:

- Selezionare il metodo **`ammontarePer()`** in **`Cliente`**
- *Tasto destro > Refactor > Move*
- Selezionare la voce **`Noleggio`** suggerita

Effetto: il metodo **`ammontarePer()`** verrà spostato da **`Cliente`** a **`Noleggio`** e invocato in **`rendiconto()`** per ogni **`noleggio`**

Rieseguire il caso di test dopo le modifiche!

Refactoring #4: Rename Method

Obiettivo: dare una semantica più chiara al nome del metodo `ammontarePer()` in `Noleggio`

Passaggi:

- Selezionare il nome del metodo `ammontarePer()` in `Noleggio`
- *Tasto destro > Rename Symbol*
- Inserire `getAmmontare` come nome

Effetto: il metodo `ammontarePer()` cambierà nome e la modifica verrà propagata nella sua invocazione in `rendiconto()`

Rieseguire il caso di test dopo le modifiche!

Refactoring #5: Replace Temp with Query

Obiettivo: sostituire i riferimenti alla variabile temporanea **`questoAmmontare`** in **`rendiconto()`**, dal momento che salva un valore su cui non vengono effettuate modifiche, con chiamate al metodo **`getAmmontare()`** di **`Noleggio`**

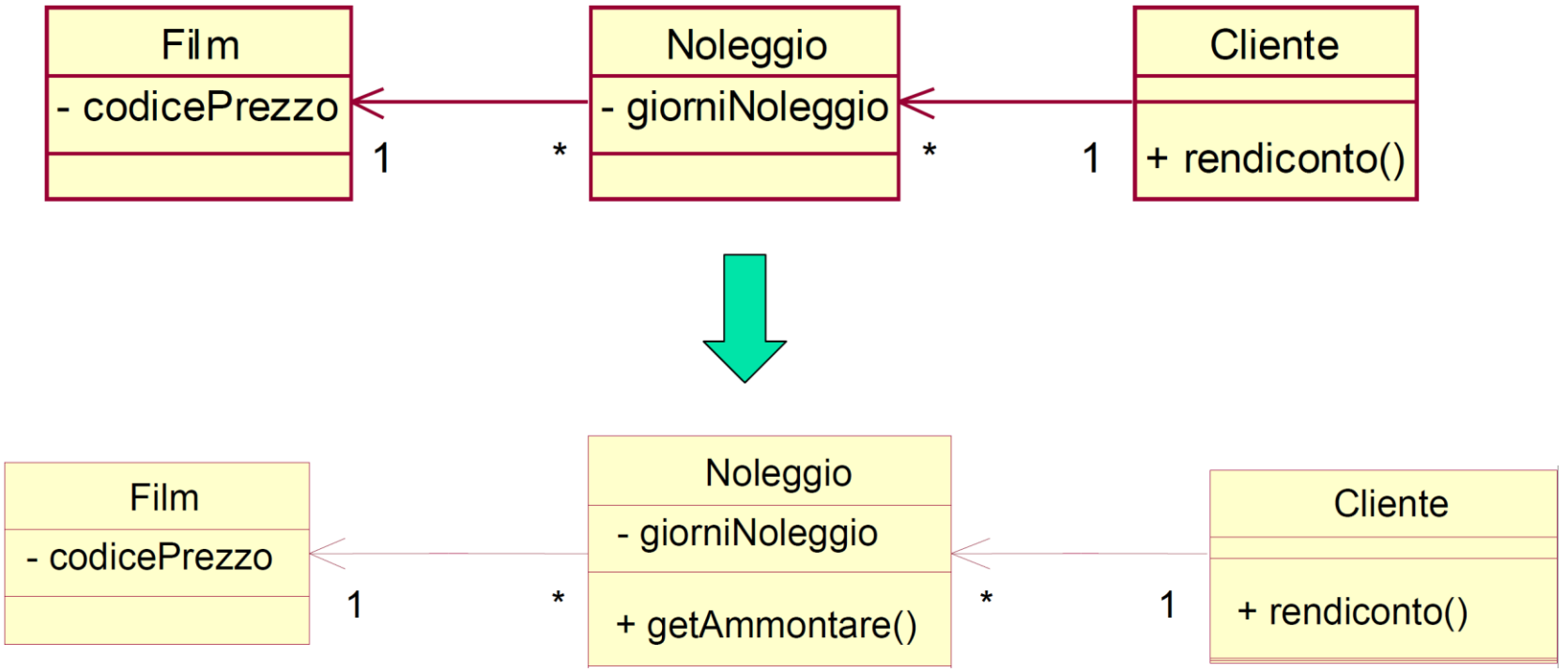
Passaggi:

- Rimuovere l'inizializzazione e l'assegnazione della variabile **`questoAmmontare`** in **`rendiconto()`**
- Sostituire ogni rimanente uso con chiamate al metodo **`getAmmontare()`** di **`Noleggio`**

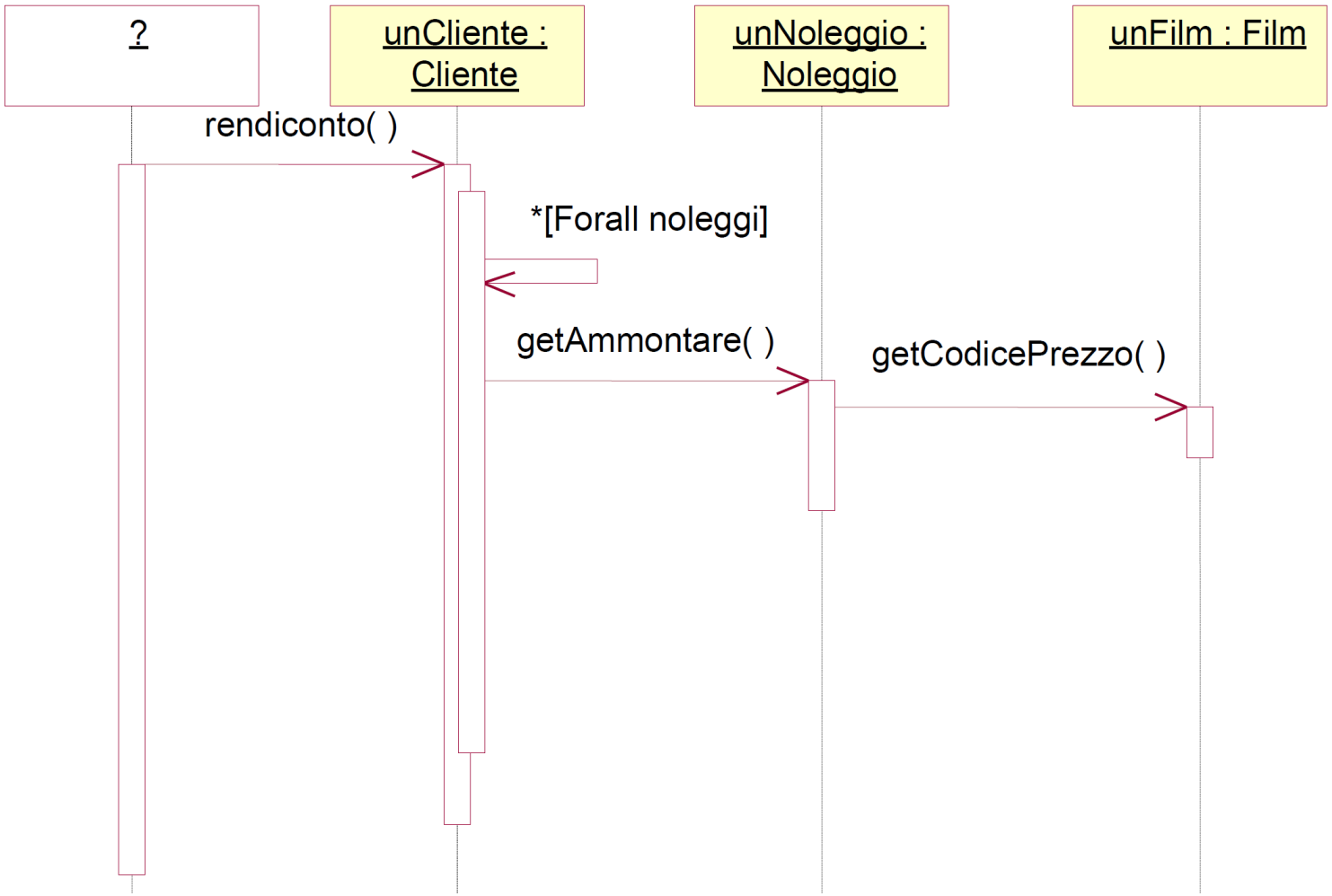
Effetto: La variabile **`questoAmmontare`** verrà rimossa e sostituita con invocazioni del metodo **`getAmmontare()`** di **`Noleggio`**

Rieseguire il caso di test dopo le modifiche!

VIDEOSTORE: CLASS DIAGRAM (V. 1)



VIDEOSTORE: SEQ. DIAGRAM RENDICONTO (V. 1)



Refactoring #6: Replace Temp with Query

Obiettivo: sostituire i riferimenti alla variabile temporanea **`ammontareTotale`** in **`rendiconto(),`** isolando la logica di calcolo del totale in un metodo specifico

Passaggi:

- Implementare un metodo **`private double getAmmontareTotale()`** in **`Cliente`**, che scorre su **`noleggi`** e restituisce l'ammontare totale
- In **`rendiconto()`**, rimuovere l'inizializzazione di **`ammontareTotale`** e il suo uso nel ciclo che calcola la somma dei totali, e sostituire l'uso della variabile nell'ultima istruzione prima del **`return`** con la chiamata a **`getAmmontareTotale()`**

Effetto: La variabile **`ammontareTotale`** verrà rimossa e sostituita con invocazioni del metodo **`getAmmontareTotale().`** L'operazione può sembrare inefficiente (più codice, più cicli), ma rende il codice più pulito e mantenibile

Rieseguire il caso di test dopo le modifiche!

Refactoring #7: Move Method

Obiettivo: assegnare la responsabilità del metodo **`getAmmontare()`** alla classe **`Film`** anziché **`Noleggio`** → usa informazioni di **`Film`**

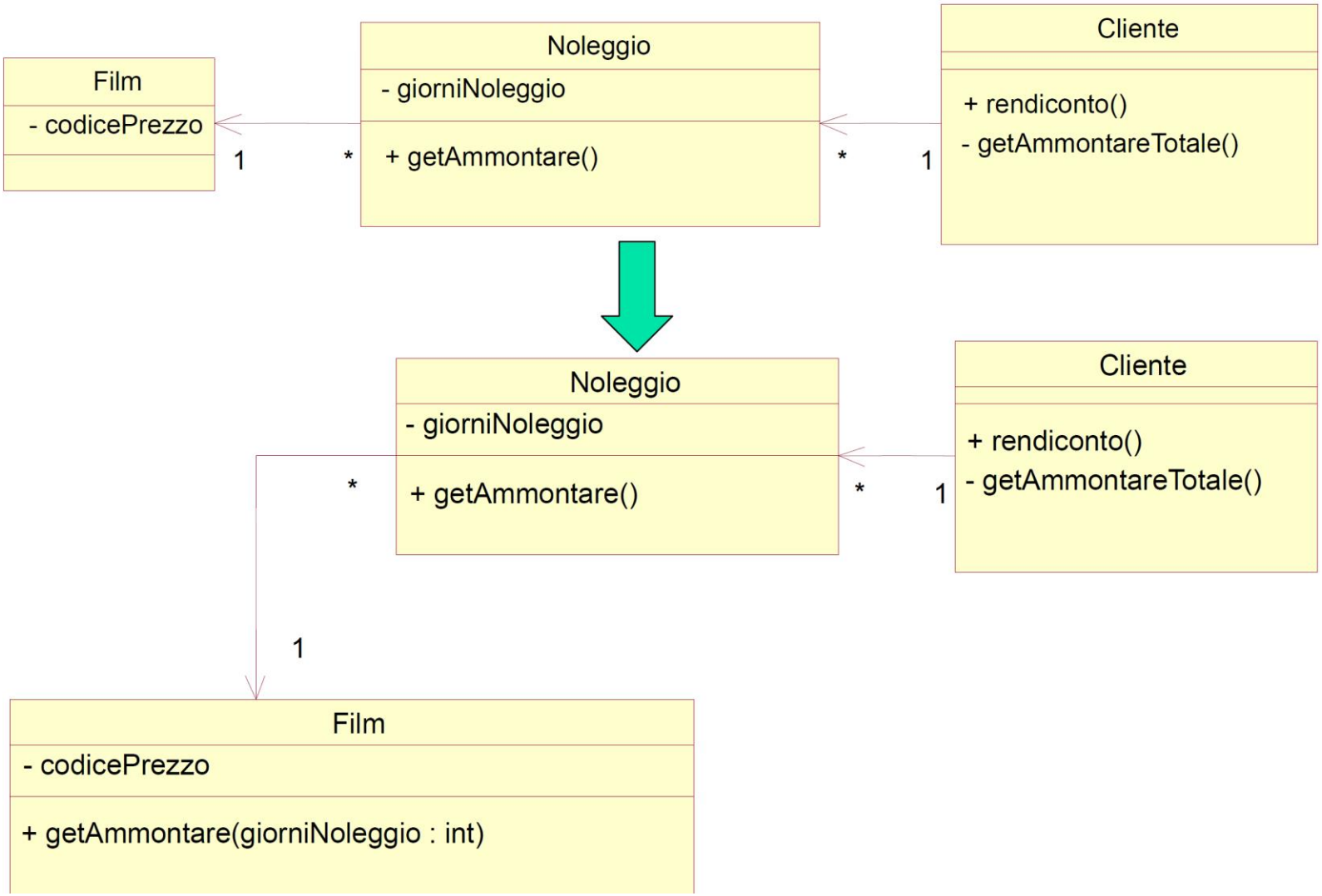
Passaggi:

- Spostare il metodo **`getAmmontare()`** da **`Noleggio`** a **`Film`**
- Modificare **`getAmmontare()`** in **`Film`**
  - Aggiungere alla segnatura il parametro **`int giorniNoleggio`**
  - Sostituire le chiamate a **`getGiorniNoleggio()`** con **`giorniNoleggio`**
  - Rimuovere **`getFilm()`** dalla condizione dello switch
- Definire un metodo **`double getAmmontare()`** in **`Noleggio`** che richiama **`getAmmontare()`** per il campo **`film`**, passandogli **`giorniNoleggio`**

Effetto: **`getAmmontare()`** verrà spostato da **`Noleggio`** a **`Film`**, introducendo in **`Noleggio`** un metodo delegante. Il refactoring automatico non è direttamente applicabile dato che non crea il metodo delegante in **`Noleggio`**!

Rieseguire il caso di test dopo le modifiche!

VIDEOSTORE: CLASS DIAGRAM (V. 2)



Refactoring #8: State Pattern

Obiettivo: sostituire la logica condizionale del metodo `getAmmontare()` in `Film` con il polimorfismo, al fine di garantire maggiore elasticità nella gestione di futuri comportamenti del software

Passaggi:

- Replace Type Code with State/Strategy (**Refactoring #8.1**)
- Move Method (**Refactoring #8.2**)
- Replace Conditional with Polymorphism (**Refactoring #8.3**)

Effetto: verrà implementata una gerarchia di classi per gestire in modo dinamico il calcolo dei prezzi dei vari noleggi

Refactoring #8.1: Replace Type Code with State/Strategy

Obiettivo: introdurre una gerarchia relativa al concetto di **`Prezzo`**

Passaggi:

- Creare 4 nuove classi: **`Prezzo`** (astratta), **`PrezzoRegolare`**, **`PrezzoBambini`** e **`PrezzoNovita`** (le ultime 3 estendono **`Prezzo`**)
  - Inserire in ogni classe un metodo **`int getCodicePrezzo()`** che ritorna il codice del prezzo, come indicato dai campi **`final`** in **`Film`**
    - es. **`getCodicePrezzo()`** in **`PrezzoBambini`** ritorna 2

Incluso un metodo astratto in **`Prezzo`**

- Modificare la classe **`Film`**
  - Sostituire il campo **`int codicePrezzo`** in **`Prezzo prezzo`**
  - Nel costruttore, rimuovere l'inizializzazione di **`codicePrezzo`** e invocare **`setCodicePrezzo(codicePrezzo)`**
  - Modificare **`setCodicePrezzo(codicePrezzo)`** introducendo una logica switch/case per creare l'istanza di **`prezzo`** corretta in base a **`codicePrezzo`**
    - es. **`switch(codicePrezzo) … case BAMBINI: prezzo = new PrezzoBambini(); …`**
  - Modificare **`getCodicePrezzo()`** richiamando il metodo omonimo di **`prezzo`**

Rieseguire il caso di test dopo le modifiche!

Refactoring #8.2: Move Method

Obiettivo: assegnare la responsabilità del metodo **`getAmmontare()`** alla classe **`Prezzo`** anziché **`Film`**

Passaggi:

- Spostare il metodo **`getAmmontare()`** da **`Film`** a **`Prezzo`**
- Definire un metodo **`double getAmmontare(int giorniNoleggio)`** in **`Film`** che richiama il metodo omonimo di **`Prezzo`**, passandogli il parametro **`giorniNoleggio`**

Il refactoring automatico non è direttamente applicabile dato che non crea il metodo delegante in **`Film`**!

Rieseguire il caso di test dopo le modifiche!

Refactoring #8.3: Replace Conditional with Polymorphism

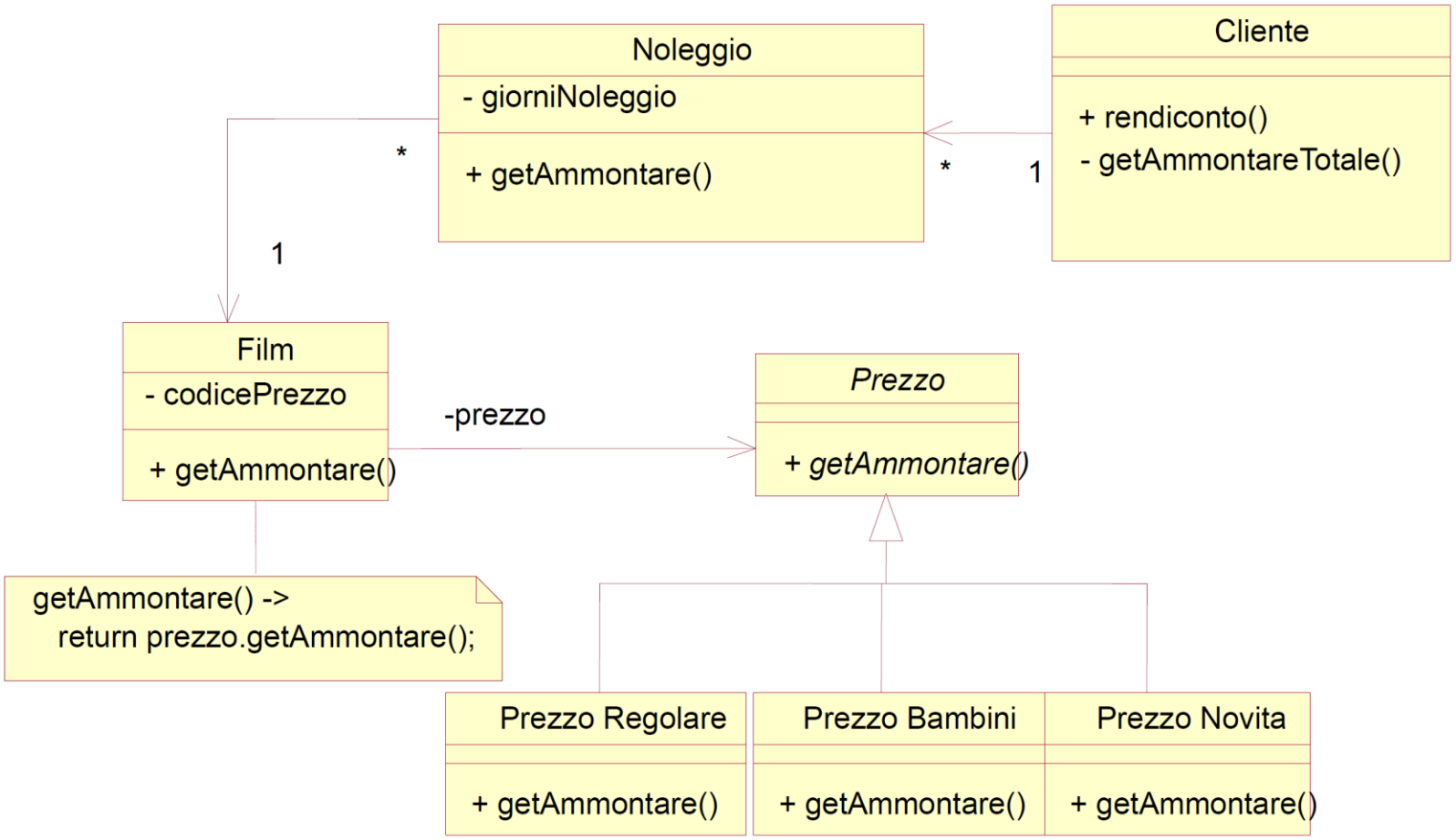
Obiettivo: sostituire la logica condizionale di **`getAmmontare()`** in **`Prezzo`** con il polimorfismo, grazie alla gerarchia introdotta

Passaggi

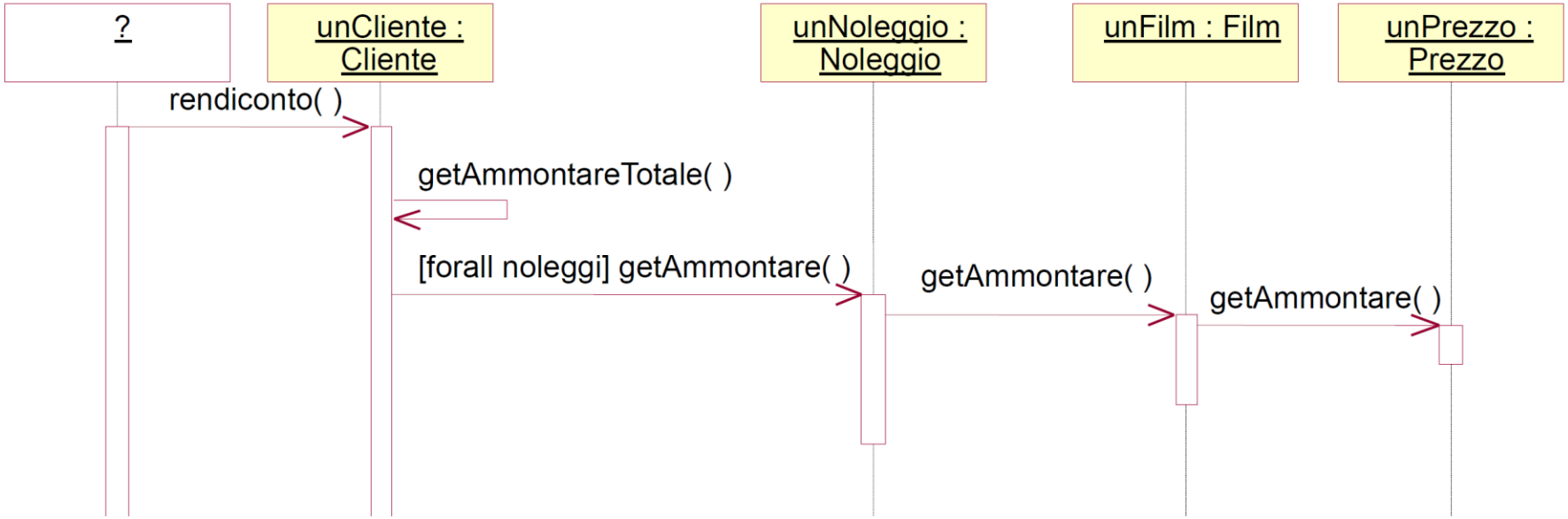
- Definire metodi vuoti **`double getAmmontare(int giorniNoleggio)`** nelle sottoclassi di **`Prezzo`** che sovrascrivano (**`@Override`**) il metodo omonimo di **`Prezzo`**
- Per ogni ramo dello switch di **`getAmmontare()`** in **`Prezzo`**, spostare e adattare il contenuto nel metodo della sottoclasse corrispondente
  - es. il ramo relativo al caso **`BAMBINI`** va spostato/adattato nel metodo **`getAmmontare()`** della classe **`PrezzoBambini`**
- Rendere astratto il metodo **`getAmmontare()`** di **`Prezzo`**, dal momento che verranno usate le implementazioni delle sottoclassi

Rieseguire il caso di test dopo le modifiche!

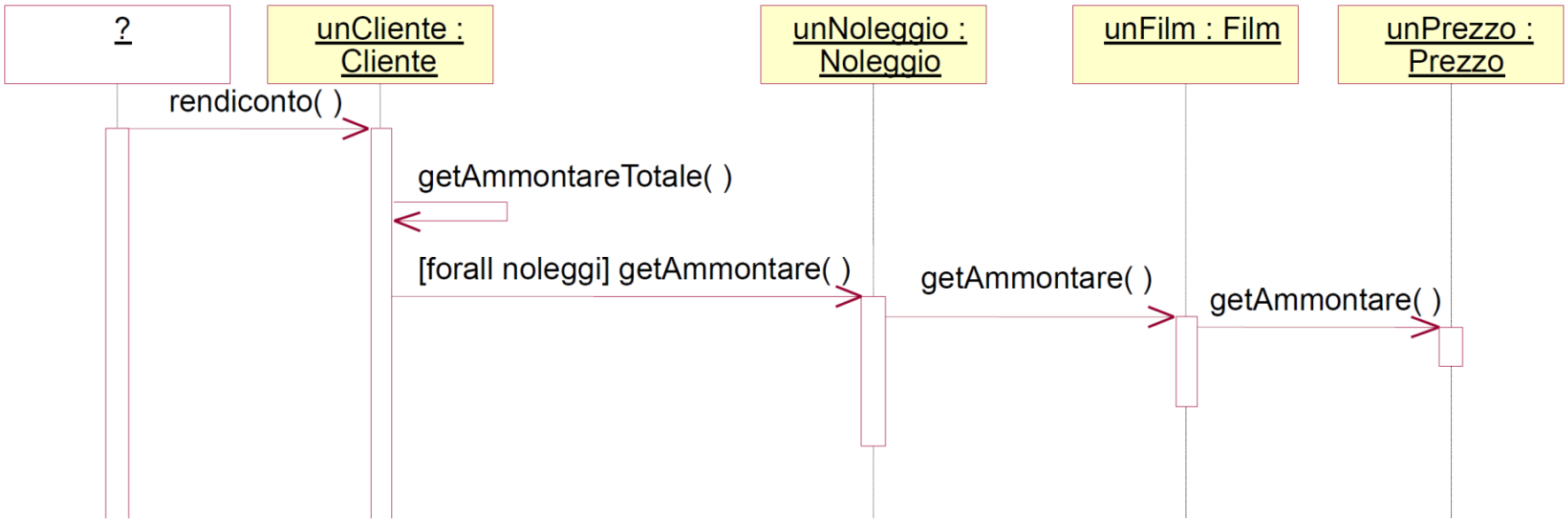
VideoStore: Class Diagram (v. 3)



VIDEOSTORE: SEQ. DIAGRAM RENDICONTO (V. 3)



VIDEOSTORE: SEQ. DIAGRAM RENDICONTO (V. 3)



Introdurre lo State Pattern è stato "costoso", ma ha permesso di avere codice più pulito e adatto a modifiche più agili future (es. aggiunta di comportamenti a seconda di nuove tipologie di prezzo)

Il consiglio è fare piccoli passi di refactoring alla volta e testare opportunamente il codice dopo ogni modifica!