Durante l'Object Design identifichiamo e perfezioniamo gli oggetti della soluzione per realizzare i sottosistemi definiti durante la progettazione del sistema. Quindi la compressione di ogni oggetto facente parte del sottosistema diventa sempre più approfondita: specifichiamo le firme dei tipi e la visibilità di ciascuna delle operazioni e descriviamo le condizioni in cui un'operazione può essere invocata e quelle in cui l'operazione solleva un'eccezione.
Poiché il sistem design era basato sull'identificazione di grandi blocchi di lavoro da assegnare ai vari team o sviluppatori, il focus di Object Design è basato sulla specifica dei confini tra gli oggetti.
L'obiettivo delle specifiche dell'interfaccia servono a :

- comunicare chiaramente e in un modo preciso i dettagli del sistema di livello sempre più basso tra i vari sviluppatori.
- Descrivere precisamente l'interfaccia di ogni oggetti cosi che non ci sia necessità di lavoro di integrazione per oggetti realizzati da diversi sviluppatori

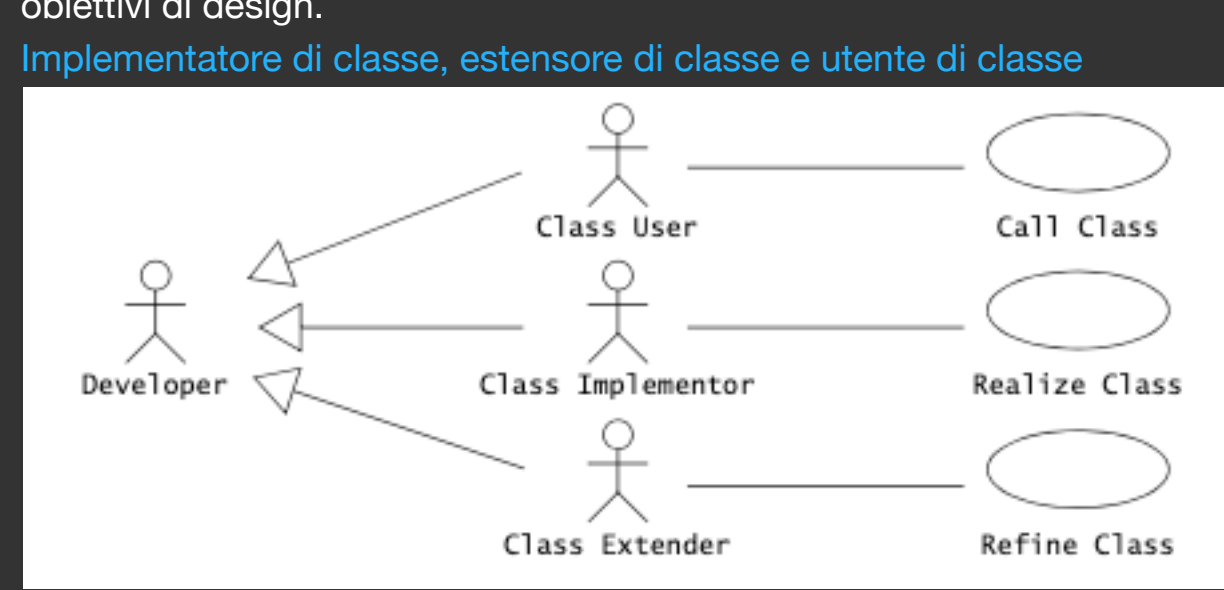
Le attività di specifica dell'interfaccia della progettazione di oggetti includono:

- Identificare gli attributi e le operazioni mancanti: esaminiamo ogni servizio del sottosistema e ogni oggetto di analisi mancanti. Identifichiamo le operazioni e gli attributi mancanti necessari per realizzare il servizio del sottosistema. Raffiniamo l'attuale modello e lo aumentiamo con delle operazioni.
- Specificare le firme e la viabilità del tipo: decidiamo quali operazioni sono disponibili per altri oggetti e sottosistemi e quali vengono utilizzati solo all'interno di un sottosistema. Specifichiamo anche il tipo di ritorno di ciascun operazione, nonché il numero e il tipo dei suoi parametri. L'obiettivo di questa attività è ridurre l'accoppiamento tra sottosistemi e fornire un'interfaccia piccola e semplice.
- Specificare le invarianti:
- Specificare le pre-condizioni e le post-condizioni:descriviamo in termini di vincoli il comportamento delle operazione fornite da ciascun oggetto. In particolare, per ogni operazione, descriviamo le condizioni che devono essere soddisfate prima che venga invocata l'operazione e una specifica del risultato dopo che l'operazione ritorna.

Useremo OCL (Object Constraint Language) come linguaggio per specificare invariati, pre-condizioni e post-condizioni. Approfitteremo quindi delle euristiche e linee guida che ci facilitano alla scrittura di vincoli leggibili.
Panoramica delle specifiche dell'interfaccia
Sono state presi molte decisioni sul sistema che hanno determinato la creazione di una vasta gamma di modelli:

- Il modello a oggetti di analisi descrive gli oggetti entità,boundary e control che sono visibili all'utente. Include attributi e operazioni per ciascun oggetto.
- La decomposizione del sistema: descrive come questi oggetti sono suddivisi in pezzi coesivi che sono realizzati da diversi team di sviluppatori. Ogni sottosistema include descrizioni di servizi di alto livello che indicano quale funzionalità fornisce agli altri.
- Il mapping Hardware/Software identifica i componenti che compongono la macchina virtuale su cui costruiamo oggetti soluzione (come classi e API definite da componenti esistenti)
- Use Case Boundary: descrivono, dal punto di vista dell'utente, i casi amministrativi ed eccezionali gestiti dal sistema.
- Design Pattern: descrivono i modelli di progettazione parziale degli oggetti che affrontano problemi di progettazione specifici.

Tutti i modelli sin qui costruiti forniscono una "visione parziale del sistema", molti pezzi mancano e altri sono da raffinare. L'obiettivo del Object Design è quello di produrre un modello Object Design che integri tutte le informazioni in modo coerente e preciso.
L'Object Design Document (**ODD**) contiene la specifica di ogni classe per supportare lo scambio di informazioni consentendo di prendere decisioni consistenti sia tra i vari sviluppatori che con gli obiettivi di design.
Implementatore di classe, estensore di classe e utente di classe



Finora per questi tre sviluppatori li abbiamo trattati allo stesso modo. Ma adesso bisogna differenziali poiché ognuno di loro visualizza l'interfaccia da un punto di vista radicalmente diverse:

- L'implementatore di classe è responsabile della realizzazione della classe in esame. Gli implementatori di classe progettano le strutture dati interne e implementato il codice per ogni operazioni pubblica. Per essi, la specifica dell'interfaccia è un incarico di lavoro
- L'Utente della classe richiama le operazioni fornite dalla classe in esame durante la realizzazione di un'altra classe, chiamata classe client. Per gli utenti della classe, la specifica dell'interfaccia rivela il limite della classe in termini di servizi che fornisce e delle ipotesi che fa sulla classe client.
- L'estensore di classe sviluppa specializzazioni della classe in esame. Si concentrano su versioni specializzate degli servizi stessi. Per essi, la specifica dell'interfaccia specifica sia il comportamento corrente della classe sia eventuali vincoli sui servii forniti dalla classe specializzata.

Esempio:

Consideriamo la classe astratta del gioco ARENA. Lo sviluppatore responsabile della realizzazione della classe Game è un implementatole di classe. Le classi League e Tournament invocano le operazioni fornite dall'interfaccia Game per organizzare e avviare le partite quindi gli sviluppatori che hanno realizzato tali classe sono utenti di classe di Game. Le classi TitTacToe e Chess sono classi specializzati quindi gli sviluppatori di tali classe sono estensori di classe di Game.
Tipi, firme e visibilità
Durante l'analisi, abbiamo trovato gli attributi e le operazioni senza specificare i tipi e i parametri. Quindi in tale fase andremo a raffinarli aggiungendo informazioni sul tipo,sulla visibilità e sui contratti.

- Tipi e signature
  - Il tipo di un attributo specifica l'intervallo di valori che l'attributo può assumere e le operazioni che possono essere applicati all'attributo.
    - Esempio: L'attributo maxNumPlayers della classe Tournament in Arena; rappresenta il numero massimo di giocatori che possono essere accettati in un determinato torneo. Il tipo è int quindi un numero intero. Le operazioni che possono giocarci con questo tipo sono confronto, somma, sottrazione o moltiplicazioni.
  - I parametri di operazioni e i valori di ritorno vengono digitati nello stesso modo degli attributi. Il tipo limita l'intervallo di valori che può assumere il parametro o il valore restituito.
    - Esempio: accetPlayer(Player p): void prende un parametro di tipo Player e non restituisce nessun valore di ritorno.
    - Esempio: getMaxNumPlayers(): int non prende nessun parametro e restituisce un valore di ritorno di tipo int.
- Visibilità
  - Di solito gli sviluppatori non sono autorizzati ad accendere a delle determinate operazione definite su una classe. Da qui ci aiuta la visibilità, un meccanismo per specificare se l'attributo o l'operazione possono essere utilizzati o meno da altre classi. UML definisce tre livelli di visibilità:
    - Privato (Class Implementor) indicata con "-" :
      - Un attributo privato e un'operazione privata sono accessibili solo dalla classe in cui è definito.
      - Gli attributi e le operazioni private non sono accessibili da sottoclassi o classi chiamati.
    - Protetto (Class Extender) indicata con "#" :
      - Un attributo o un'operazione protetta è accessibile solo dalla classe in cui è definita e da qualsiasi discendete di quella classe. Non sono accessibile da nessun'altra classe.
    - Pubblico (Class User) indicata con "+" :
      - Un attributo o un'operazione pubblica è accessibile da qualsiasi classe.
- Contratti
  - Le informazioni sul tipo spesso non sono sufficienti per esprimere appieno l'informazioni che possiede. Per esempio: il tipo dedicato alla variabile "maxNumPlayers consente di ricevere come input dei valori negativi, valori che non assume nessun significato logico per l'applicazione in questione(ARENA). Per risolvere si usano i contratti.
  - Un contratto specifica i vincoli che l'utente della classe deve soddisfare prima di utilizzare la classe. Includono tre tipi di vincoli:
    - Un invariante è un predicato che è sempre vero per tutte le istanze. Sono associati a classi o interfaccie.
      - Esempio: il maxNumPlayers dovrebbe avere un numero positivo poichè se fosse stato creato un Torneo con maxNumPlayers=0,allora acceptPlayer() violerebbe sempre il suo contratto e il Torneo non potrebbe mai iniziare.
    - Una precondizione è un predicato che deve essere vero prima che venga invocata un'operazione. Sono associati a metodi.
    - Un postcondizione è un predicato che deve essere vero dopo che è stata invocata un'operazione. Sono associati a metodi.

Torneo
- maxNumPlayers:int
+ getMaxNumPlayers():int + getNumPlayers(): int + getPlayers():list +acceptPlayer(p:Player):void +removePlayer(p:Player):void +isPlayerAccepted(p:Player):boolean

Un'esempio di invariante per la classe Torneo è che il numero di massimo dei giocatori dovrebbe essere positivo. Quindi se un torneo viene creato con maxNumPlayer pari a 0 o valori negativi violerà il contratto causando cosi l'eccezione. Per esprimere tale contesto:

```
t.getMaxNumPlayers()>0 // t è un torneo
```

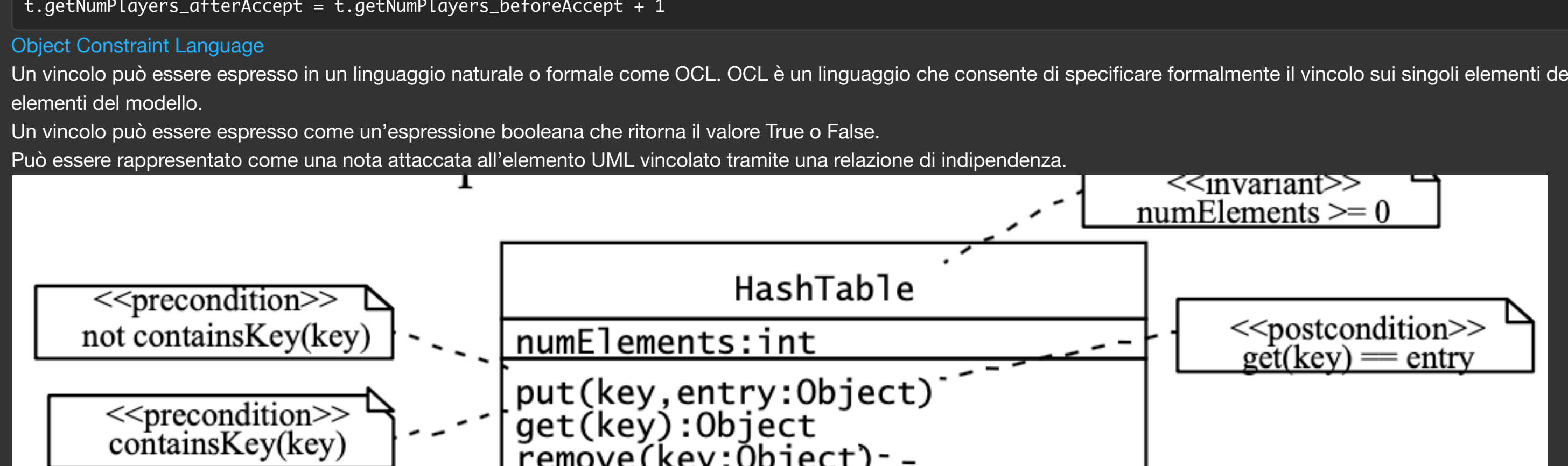
Un esempio di condizione preliminare per il metodo accetPlayer è che il giocatore da aggiungere non è stato ancora accettato nel torneo e che il torneo non ha ancora raggiunto il numero massimo di giocatori.

```
!t.isPlayerAccepted(p) && t.getNumPlayers()<t.getMaxNumPlayers() // t è un torneo mentre p è un giocatore.
```

Un esempio di postcondizione per il metodo accetPlayer() è che il numero corrente di giocatori deve essere uno in più del numero di giocatori prima dell'invocazione di acceptPlayer().

```
t.getNumPlayers_afterAccept = t.getNumPlayers_beforeAccept + 1
```

Object Constraint Language
Un vincolo può essere espresso in un linguaggio naturale o formale come OCL. OCL è un linguaggio che consente di specificare formalmente il vincolo sui singoli elementi del modello o gruppi di elementi del modello.
Un vincolo può essere espresso come un'espressione booleana che ritorna il valore True o False.
Può essere rappresentato come una nota attaccata all'elemento UML vincolato tramite una relazione di indipendenza.



Però rappresentarli in questo modo, tramite delle note, può causare il disordine quindi si preferisce rappresentarli tali vincoli in formato testuali:
L'invariante per la classe Torneo la quale specifica che l'attributo maxNumPlayers deve essere positivo può essere espresso in questa forma:

```
context Tournament inv:
    self.getMaxNumPlayers()>0
```

La parola chiave "context" indica l'entità a cui si applica l'espressione, seguito poi da una delle parole chiave "inv", "pre" e "post" che che corrispondono rispettivamente gli stereotipi UML <<invariante>>, <<precondizione>> e <<postcondizione>>.

Per precondizioni e postcondizioni, il contesto dell'espressione OCL è un'operazione. I parametri passati all'operazione possono essere utilizzati come variabili nell'espressione. Ad esempio: consideriamo la seguente precondizione sull'operazione acceptPlayer sul Torneo:

```
context Tournament::acceptPlayer(p:Player) pre:
    !isPlayerAccepted(p)
```

Poiché si tratta di una condizione preliminare, il vincolo deve essere True prima di poter essere invocato l'operazione acceptPlayer. Quindi tale vincolo esprime: acceptPlayer(p) presume che p non sia stato ancora accettato nel Torneo". Se ci sono più di una precondizione per una data operazione, è necessario che tutte queste precondizioni siano vere prima di poter essere eseguita tale operazione. Ad esempio, possiamo anche affermare che il Torneo non deve ancora raggiunto il numero massimo di giocatori prima di invocare acceptPlayer():

```
context Tournament::acceptPlayer(p:Player) pre:
    getNumPlayers() < getMaxNumPlayers()
```

Le postcondinzione sono scritte allo stesso modo delle precondizione con la differenza che cambia la parola chiave: pre—>post dove indica il vincolo che viene valutato dopo il ritorno dell'operazione.
Ad esempio, la seguente postcondizione su acceptPlayer (p) afferma che il giocatore p dovrebbe essere noto al Torneo dopo la restituzione di acceptPlayer( ):

```
context Tournament::acceptPlayer(p:Player) post:
    isPlayerAccepted(p)
```

Per le postcondizioni, spesso è necessario fare riferimento al valore di un attributo prima e dopo l'esecuzione dell'operazione. A tale scopo, il suffisso @pre indica il valore di sé o un attributo prima dell'esecuzione dell'operazione. Ad esempio, se vogliamo affermare che il numero di giocatori nel torneo aumenta di uno con l'invocazione di acceptPlayer (), dobbiamo fare riferimento al valore di getNumPlayers () prima e dopo l'invocazione di acceptPlayer (). Possiamo scrivere il seguente postcondition:

```
context Tournament::acceptPlayer(p:Player) post:
    getNumPlayers() = self@pre.getNumPlayers() + 1
```

@pre.getNumPlayers sta a indicare il valore che viene ricevuto dall'invocazione getNumPlayers() prima dell'invocazione dell'operazione acceptPlayer() e getNumPlayers() indica il valore di ritorno dalla stessa operazione dopo aver invocato l'operazione acceptPlayer().

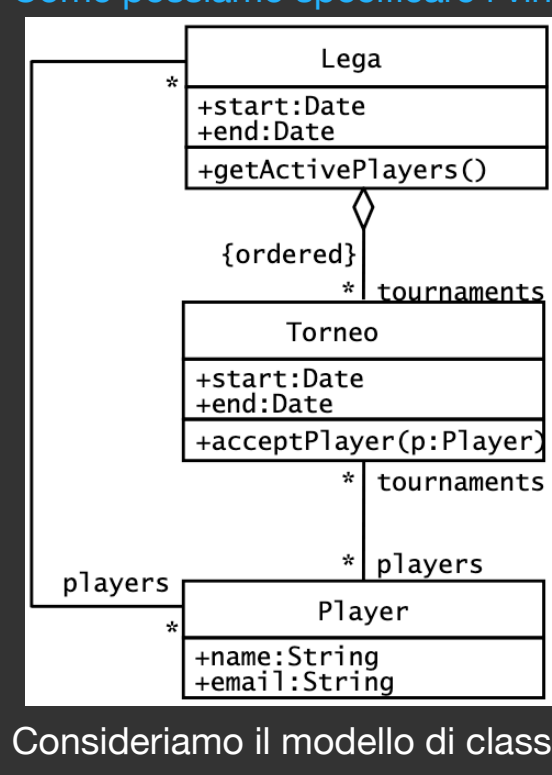
Con il metodo removePlayers si possono scrivere dei seguenti vincoli:

```
context Tournament::removePlayer(p:Player) pre:
    isPlayerAccepted(p)
context Tournament::removePlayer(p:Player) post:
    !isPlayerAccepted(p)
context Tournament::removePlayer(p:Player) post:
    getNumPlayers() = self@pre.getNumPlayers() - 1
```

La figura mostrata in seguito mostra il codice implementato rispettando i vincoli nominati in precedenza:

```
public class Torneo {

    /** Il Massimo numero di players * è sempre positivo.
     *@invariant maxNumPlayers > 0 */
     private int maxNumPlayers;

     /** La List di players contiene * referimenti ai Players che *
     sono registrati al Torneo. */
    private List players;

    /** Restituisce in numero corrente di players nel torneo. */
    public int getNumPlayers() {
     ...}

     /** Restituisce in numero massimo * di players nel torneo. */
    public int getMaxNumPlayers(){
    ...}

    /** L'operazione acceptPlayer()
     * assume che il player
     * specificato non è stato ancora
     * accettato nel Torneo.
     * @pre not isPlayerAccepted(p)
     * @pre getNumPlayers()<maxNumPlayers * @post isPlayerAccepted(p)
     * @post getNumPlayers() =
     * @pre.getNumPlayers() + 1
     */
    public void acceptPlayer (Player p) {
    ...}

    /** L'operazione removePlayer() * assume che il player
     * specificato è al momento nel
     Torneo.
     * @pre isPlayerAccepted(p)
     * @post not isPlayerAccepted(p)
     * @post getNumPlayers() =@pre.getNumPlayers() - 1 */
    public void removePlayer(Player p) {
    ...}
}
```

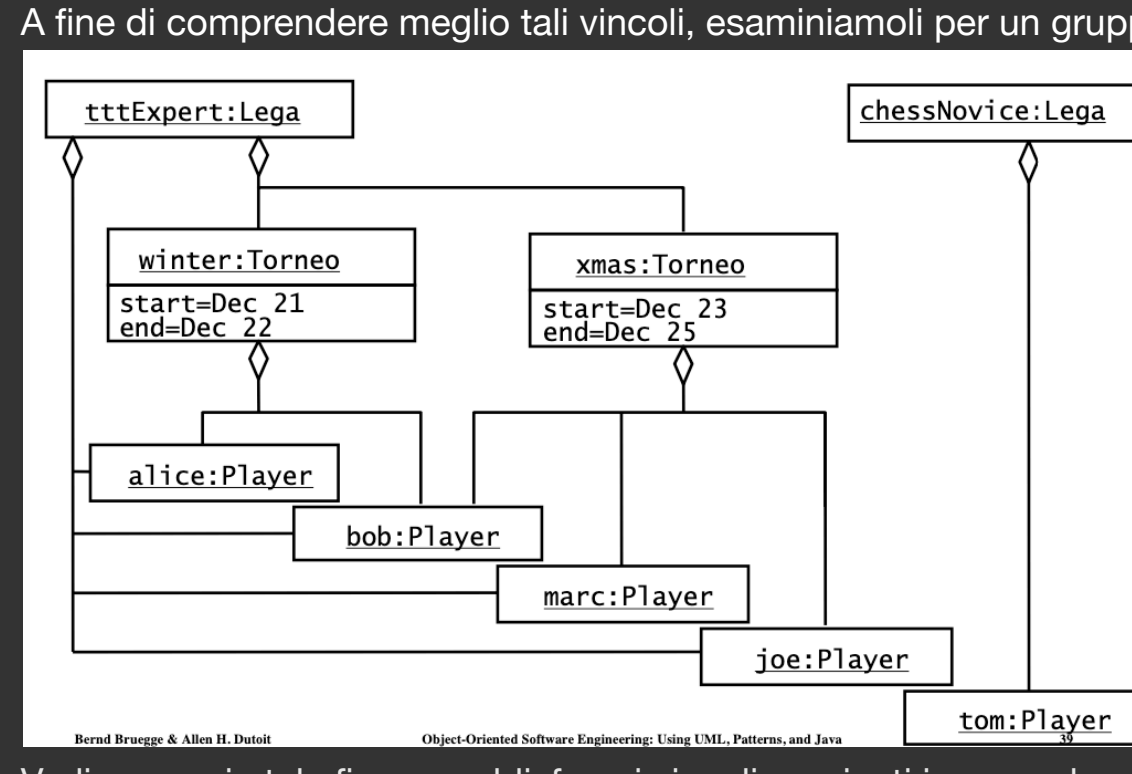
Come possiamo specificare i vincoli che coinvolgono più di una classe?



Consideriamo il modello di classe nella quale rappresenta le associazioni tra le classi League, Torneo e Giocatore.
Supponiamo di voler perfezionare il modello con i seguenti vincoli:

1. La durata prevista di un torneo deve essere inferiore a una settimana.
2. I giocatori possono essere accettati in un Torneo solo se sono già registrati nella Lega corrispondente.
3. Il numero di giocatori attivi in una lega è quello che ha preso parte ad almeno un torneo della lega.

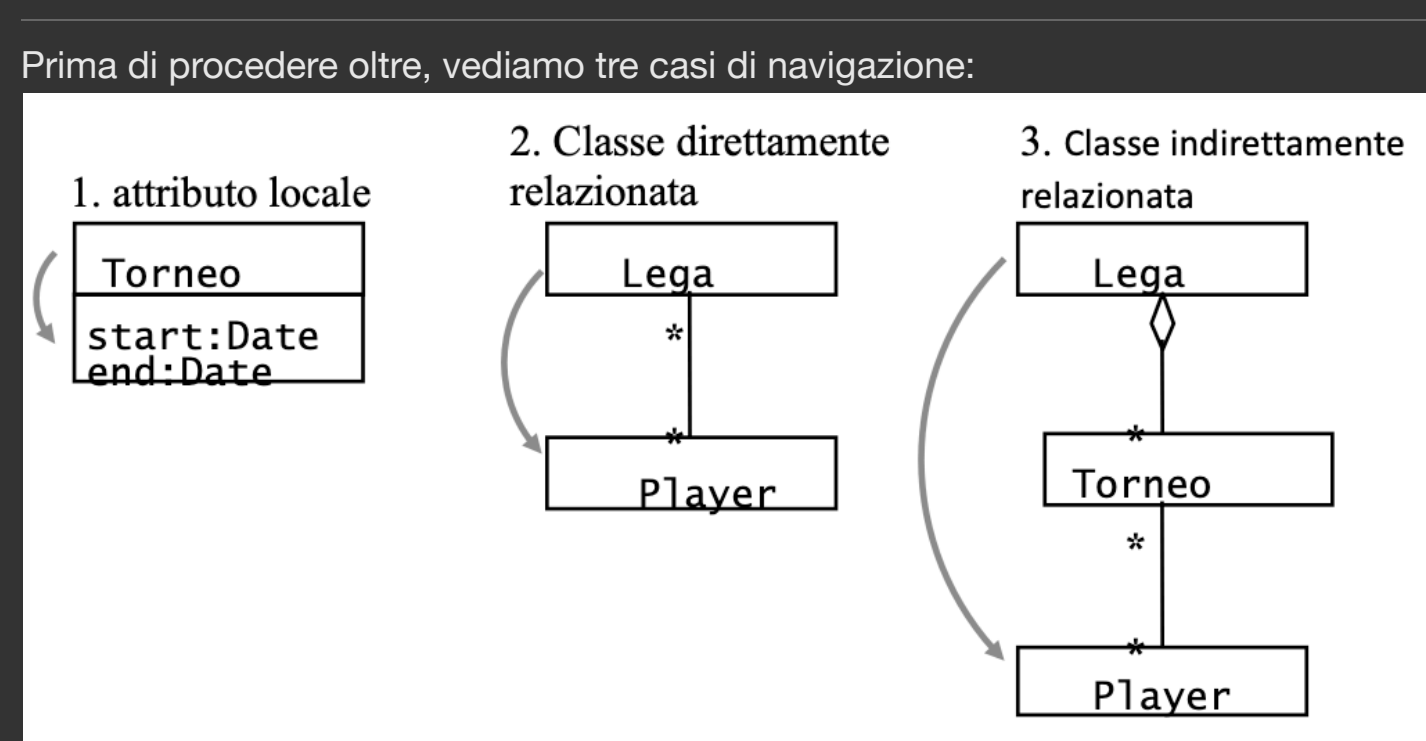
A fine di comprendere meglio tali vincoli, esaminiamoli per un gruppo specifico di istanze:



Vediamo se in tale figura soddisfano i vincoli nominati in precedenza:

1. Winter: torneo dura due giorni e xmas:torneo tre giorni, quindi entrambi meno di una settimana.
2. Tutti i giocatori di winter:torneo e mas:torneo sono associati a tttExpert:Lega. Il giocatore zoe, tuttavia, non fa parte di tttExpert:Lega e non partecipa a nessuno dei tornei.
3. tttExpert:Lega ha quattro giocatori attivi mentre il chessNovice:Lega non ne ha nessuno perché Zoe non partecipa a nessun torneo.

Prima di procedere oltre, vediamo tre casi di navigazione:



- Attributo locale: il vincolo implica un attributo locale per la classe di interesse (primo vincolo, gli attributi start e end di Torneo).
- Classe direttamente relazionata: l'espressione implica la navigazione di una singola associazione a una classe direttamente carrellata (secondo vincolo, i Players di Torneo, Lega di un Torneo).
- Classe indirettamente relazionata: il vincolo prevede la navigazione di una serie di associazioni verso una classe indirettamente collegata. (terzo vincolo, i Players di tutti i Tornei di una Lega).

Quando abbiamo a che fare con associazioni molti a molti, l'espressione lega.players può riferirsi a molti oggetti. Per far fronte a questa situazione, OCL fornisce tipi di dati aggiuntivi chiamate "collezioni":

1. I set OCL vengono utilizzati durante la navigazioni di una singola associazione.
   - Navigando l'associazione player di winter:Torneo otteniamo l'insieme {alice,bob}
   - Navigando nell'associazione player di tttExpert:Lega otteniamo l'insieme {alice, bob,marc,joe}
   - Nel caso in cui la molteplicità è pari a 1, produce un oggetto, non un insieme.
2. OCL sequences: vengono utilizzati durante la navigazione di una singola associazione "ordinata".
   - Navigando l'associazione da tttExpert:Lega otteniamo la sequenza [winter:Torneo e xmas:Torneo], in cui gli elementi hanno indici 1 e 2 rispettivamente.
3. OLC bags (multisite—>multinsiemi): sono usati per accumulare oggetti quando si accede a oggetti indirettamente correlati. Esempio: per determinare quali Player sono attivi nella tttExpert:Lega navighiamo:
   - l'associazione da tttExpert:lega;
   - l'associazione Players da winter:Torneo;
   - l'associazione Players da xmas:Torneo;
     - ottenendo il bag{alice,bob,bob,marc,joe};
       - P.S:da notare come i bag possono contenere più di una volta lo stesso oggetto.
   - Se non siamo interessati ad ottenere il numero di occorrenze di ogni oggetto nel bag, allora il bag può essere convertito in un insieme usando l'operazione asSet(collection).

OCL fornisce molte operazioni per l'accesso alle raccolte. I più utilizzati sono:

- size: restituisce il numero di elementi nella raccolta;
- includes(object): restituisce True se l'oggetto è nella raccolta.
- select(expression): restituisce una raccolta che contiene solo gli elementi di collezione originale per cui l'espressione è vera.
- union(collection):restituisce la collezione contenente sia gli elementi della collezione originale sia quelli della collezione specificata come parametro.
- interesection(collection): restituisce la collezione contenente solo gli elementi che appartengono sia alla collezione originale sia alla collezione specificata come parametro.
- asSet(collection): restituisce un insieme contenente gli elementi che appartengono alla collezione.

Esprimiamo i vincoli:
1° vincolo: la durata di un torneo deve essere minore di una settimana:

```
context Tournament inv:
    self.end - self.start <= 7
```

2° vincolo: I Players possono essere accettati in un Torneo solo se sono già registrati con la Lega corrispondente:

```
context Tournament::acceptPlayer(p:Player) pre:
    lega.players->includes(p)
```

Per arrivare alla classe Players, dobbiamo prima navigare attraverso l'associazione tra Torneo e Lega, quindi l'associazione tra Lega e Giocatore.

1. Ci riferiamo alla classe della lega utilizzando il nome del ruolo associato all'associazione o, se non è disponibile alcun nome, utilizziamo il nome della classe correlata con la prima lettera in minuscolo—>lega.
2. Navigheremo l'associazione dei giocatori sulla Lega, dove si traduce in un set a causa della molteplicità molti a molti—>lega.players.
3. Usiamo l'operazione include() su questo set per verificare se il giocatore p è noto alla Lega—>lega—>lega.players->inlcudes(p).

3° vincolo:Il numero di Players attivi in una Lega sono quelli che hanno preso parte ad almeno un Torneo della Lega:

```
context League::getActivePlayers:Set post:
    result = tournaments.players->asSet()
```

L'espressione tournament.players contiene la concatenazione di tutti i giocatori relativi alla lega attuale. Come risultato di questa concatenazione, gli elementi possono apparire più volte. Per rimuove i duplicati in questo bag, possiamo convertire il bag in un set usando l'operazione OCL asSet.
OCL Qualificatori: forAll and exists
Due operazioni aggiuntive sulle raccolte ci consentono di scorrere le raccolte e testare le espressioni su ciascun elemento:

- forAll(variable|expression): è True se expression è True per tutti gli elementi nella raccolta.
- exist(variable|expression): è True se esiste almeno un elemento nella raccolta per expression è vero.

Ad esempio, per garantire che tutte le partite di un torneo si svolgano entro il periodo di tempo del torneo, possiamo testare ripetutamente le date di inizio di tutte le partite su Torneo usando forAll:

```
1. context Tournament inv:
    matches->forAll(m:Match | m.start.after(start) and m.end.before(end))
```

L'operazione OCL exists() è simile a forAll(), tranne per il fatto che le espressioni valutate su ciascun elemento sono ORed, ovvero solo un elemento deve soddisfare l'espressione affinché l'operazione exists() restituisca True.
Ad esempio per garantire che ogni torneo organizzi almeno una partita il primo giorno del torneo:

```
1. context Tournament inv:
    matches->exists(m:Match | m.start.equals(start))
```

Gestione dell'Object Design
Esistono due principali sfide di gestione durante la progettazione degli oggetti:

- Maggiore complessità della comunicazione.
  - Il numero di partecipanti coinvolti in questa fase di sviluppo aumenta notevolmente. E' necessario assicurare che le decisione prese siano in accordo con gli obiettivi del progetto.
- Consistenza con decisioni e documenti precedenti.
  - Gli sviluppatori spesso non apprezzano completamente le conseguenze dell'analisi e delle decisioni di Sistem Design prima di Object Design. Quindi possono mettere in discussione alcune di queste decisioni e rivalutarle. Bisogna assicurare di mantenere un registro di queste decisioni riviste e far si che tutti i documenti riflettano lo stato attuale dello sviluppo.

Ci viene in aiuto ODD.
Object Design Document (ODD)
Descrive i compromessi di progettazione degli oggetti fatti dagli sviluppatori, le linee guida seguite per le interfacce dei sottosistemi, la scomposizione dei sottosistemi in pacchetti e classi e le interfacce di classe. Viene utilizzato per scambiare informazioni sull'interfaccia tra i team e come riferimento durante i test. E' rivolto a:

- Architetti del sistema(ovvero gli sviluppatori che partecipazione alla progettazione del sistema).
- Sviluppatori che implementano ciascun sottosistema.
- Tester.

Object Design Document

1. Introduzione
   1. 1. Object Design Trade Offs.
   2. .2 Linee guida per l'implementazione.
   3. .3 Definizioni, acronimi e abbreviazioni.
   4. .4 Riferimenti.
2. Packages.
3. Class Interfaces.
4. Class Diagram

Glossario.

L'introduzione descrive i compromessi generali fatti dagli sviluppatori (ad esempio, spazio di memoria vs tempo di risposta), linee guida e convenzioni(convezioni di denominazione, casi limite, meccanismi di gestione delle eccezioni) e una panoramica del documento.
Le linee guida per la documentazione e le convezioni di codifica sono dei fattori che possono migliorare notevolmente la comunicazione tra gli sviluppatori.
Includono una serie di regole:

1. Le classi sono nominati con nomi singolari.
2. I metodi sono denominati con frasi verbali, campi e parametri con frasi di nomi.
3. Lo stato dell'errore viene restituito tramite un'eccezione, non un valore di ritorno.
4. Le raccolte e i contenitori hanno un metodo iterator() che restituisce un iterator.
5. Gli iterativi restituiti dai metodi iterator() sono robusti per la rimozione degli elementi.

Packages descrive la scomposizione dei sottosistemi in pacchetti e l'organizzazione dei file del codice. Ciò include una panoramica di ciascun pacchetto, le sue dipendenze con altri pacchetti e il suo utilizzo previsto.
Class Interfaces descrive le classi e le loro interfacce pubbliche. Ciò include una panoramica di ogni classe, le sue dipendenze con altre classi e pacchetti, i suoi attributi pubblici, le operazioni e le eccezioni che possono sollevare.

La versione iniziale di ODD può essere scritta subito dopo che la decomposizione del sottosistema è stabile. L'ODD viene aggiornato ogni volta che nuove interfacce diventano disponibili o quelle esistenti vengono riviste. Anche se il sottosistema non è ancora funzionante, avere un'interfaccia di codice sorgente consente agli sviluppatori di codificare più facilmente sottosistemi dipendenti e comunicare in modo inequivocabile. In questa fase, gli sviluppatori di solito scoprono parametri mancanti e nuovi casi limite. Lo sviluppo dell'ODD è diverso da altri documenti, poiché sono coinvolti più partecipanti e il documento viene rivisto più frequentemente. Per soddisfare un alto tasso di cambiamento e molti sviluppatori, gli ultimi tre sezioni possono essere generate da uno strumento dai commenti del codice sorgente ovvero "Javadoc".