Diseño de Objetos

Índice

1. Introducción

- En el diseño de sistemas se han identificado objetivos de diseño y subsistemas, y se han seleccionado estrategias
- El diseño de objetos es el proceso de añadir detalles al análisis y tomar decisiones de implementación:
  - Lenguaje de programación, estructuras de datos y algoritmos
- El diseñador de objetos debe elegir entre diferentes caminos para implementar el modelo de análisis con el objetivo de minimizar el tiempo de ejecución, memoria, y otras medidas de coste
- Los casos de uso y el modelo dinámico del análisis proporcionan operaciones para el modelo de objetos
- El diseño de objetos itera sobre los modelos, en particular el modelo de objetos, y refina los modelos
- El diseño de objetos sirve como base de implementación

Actividades principales del diseño de objetos

- Reutilización: Identificación de soluciones existentes
  - Uso de herencia
  - Componentes *Off-the-shelf* y objetos solución adicionales
  - Patrones de diseño
- Especificación de interfaces
  - Describe concretamente la interfaz de cada clase
- Reestructuración del modelo de objetos
  - Transforma el modelo de los objetos del diseño para mejorar su comprensión y extensibilidad
- Optimización del modelo de objetos
  - Transforma el modelo de los objetos del diseño para abordar criterios de rendimiento como tiempo de respuesta o utilización de memoria

2. Reusabilidad

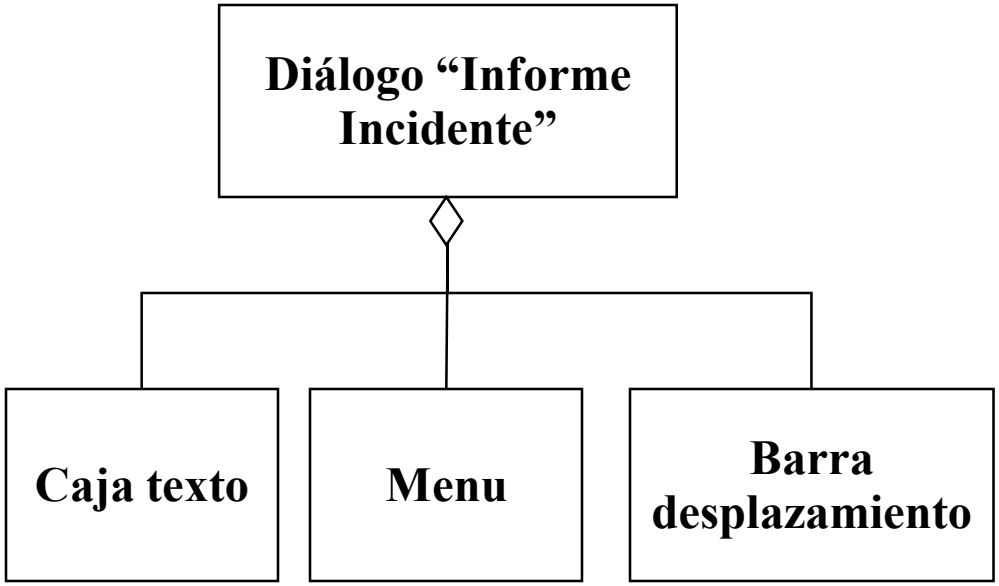
- Objetos del dominio de la aplicación vs objetos del dominio de la solución
- Objetos de aplicación (objetos dominio) representan conceptos del dominio que son relevantes para el sistema
  - Son identificados por los especialistas en el dominio de aplicación y por los usuarios finales
- Los objetos solución representan conceptos que no tienen una contrapartida en el dominio de la aplicación
  - Son identificados por los desarrolladores
  - Ejemplos:
    - almacenes de datos persistentes
    - objetos de interfaz de usuario (ej: componentes del GUI)
    - middleware para sistemas distribuidos
    - …

Objetos Aplicación vs Objetos Solución

Análisis
(lenguaje del dominio de aplicación)

Informe Incidente

Diseño de objetos
(lenguaje del dominio de solución)

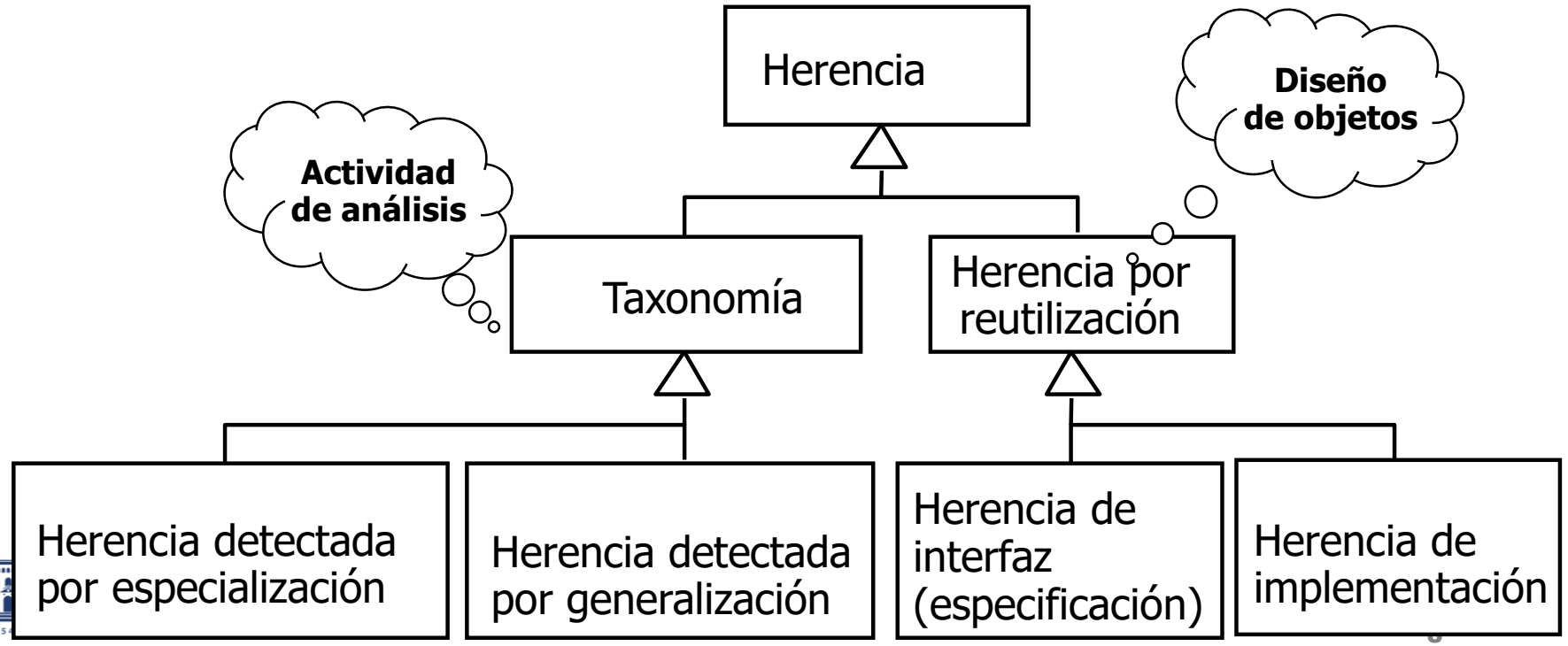


Reusabilidad (II)

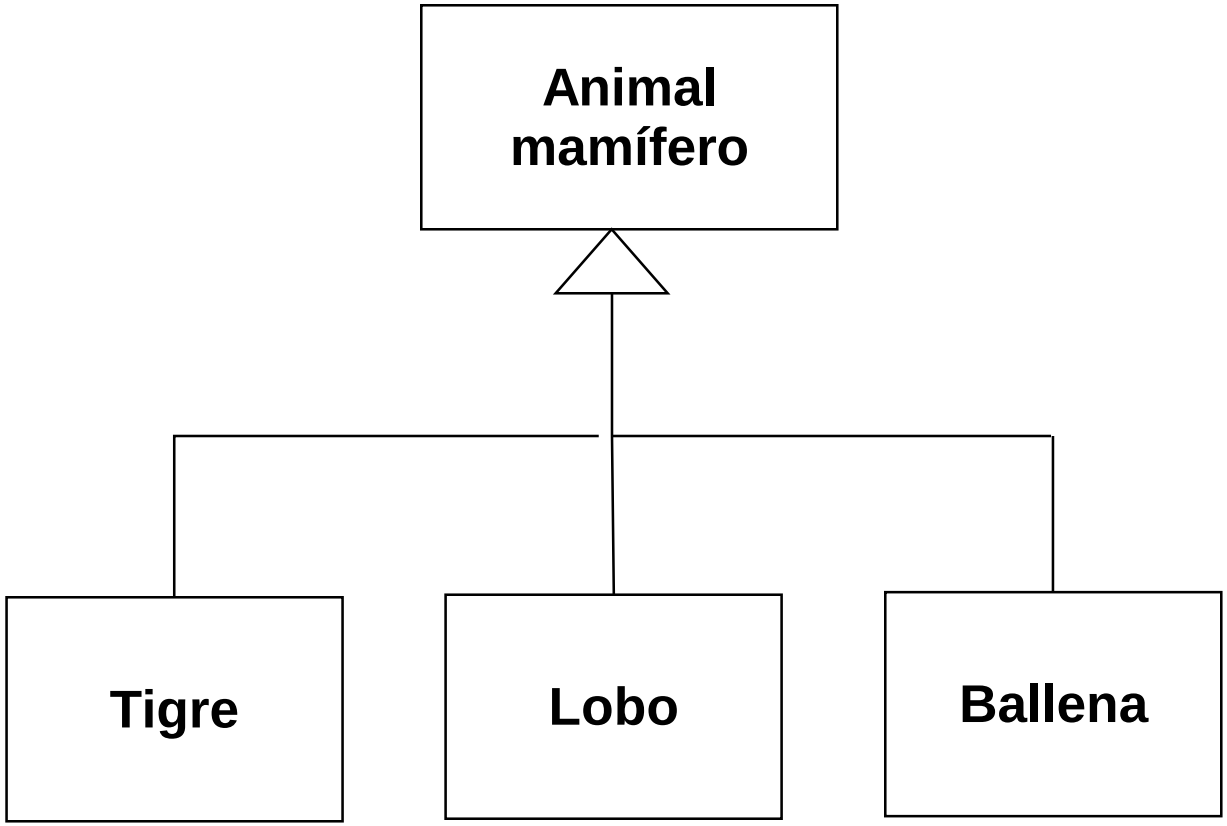
- Para diseñar la solución conviene aprovechar el conocimiento de diseño ya existente
  - Estrategias flexibles y reutilizables
- Posibles estrategias de reusabilidad
  - Herencia
  - Delegación
  - Patrones de diseño
  - Librerías, *frameworks*

2.1. Herencia

- La herencia se utiliza con dos objetivos diferentes:
  - Durante el análisis, para identificar los objetos dominio que están jerárquicamente relacionados en una taxonomía
  - Durante el diseño, para incrementar la reusabilidad, facilitar mantenimiento y extensión

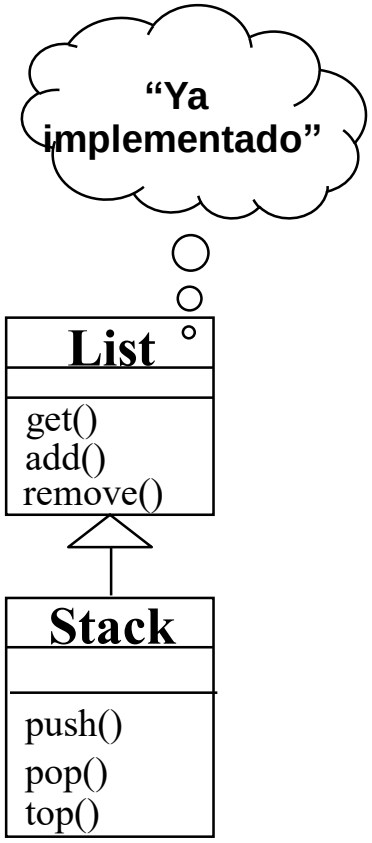


Ejemplo de taxonomía



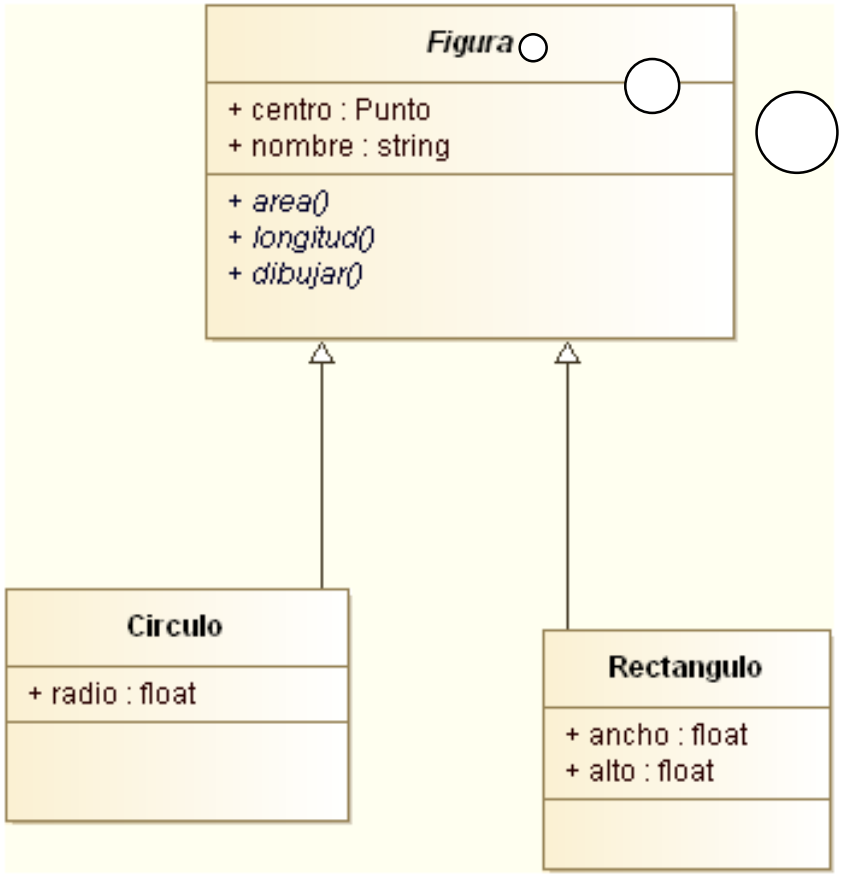
Herencia de implementación

- Objetivo: Extender la funcionalidad de una aplicación reusando la funcionalidad en la clase padre
  - Existe una clase muy similar ya implementada que proporciona prácticamente la misma implementación deseada
  - Heredamos de esa clase existente con algunas o todas las operaciones ya implementadas
- Ejemplo: tengo una clase Lista (*List*) y necesito una clase Pila (*Stack*). ¿Hacer que la clase *Stack* sea una subclase de *List* y proporcione los 3 métodos: *push()*, *pop()*, y *top()*?
- Problema de la herencia de implementación:
  - Alguna de las operaciones heredadas pueden mostrar un comportamiento no deseado
  - ¿Qué ocurre si el usuario llama a *remove()* en lugar de *pop()*?



Herencia de interfaz

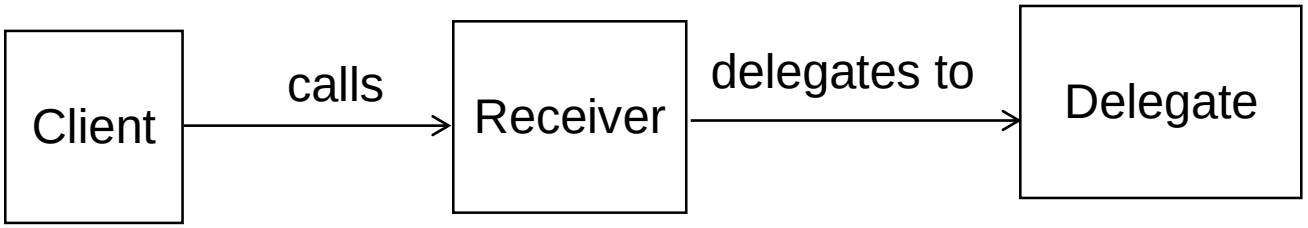
- También conocida como *subtyping*
- Heredar de una clase abstracta con todas las operaciones especificadas, pero no implementadas todavía



Notación UML:
Se utiliza cursiva para denotar clase o método abstracto (alternativamente también se puede añadir la palabra clave {abstract} a continuación)

2.2. Delegación

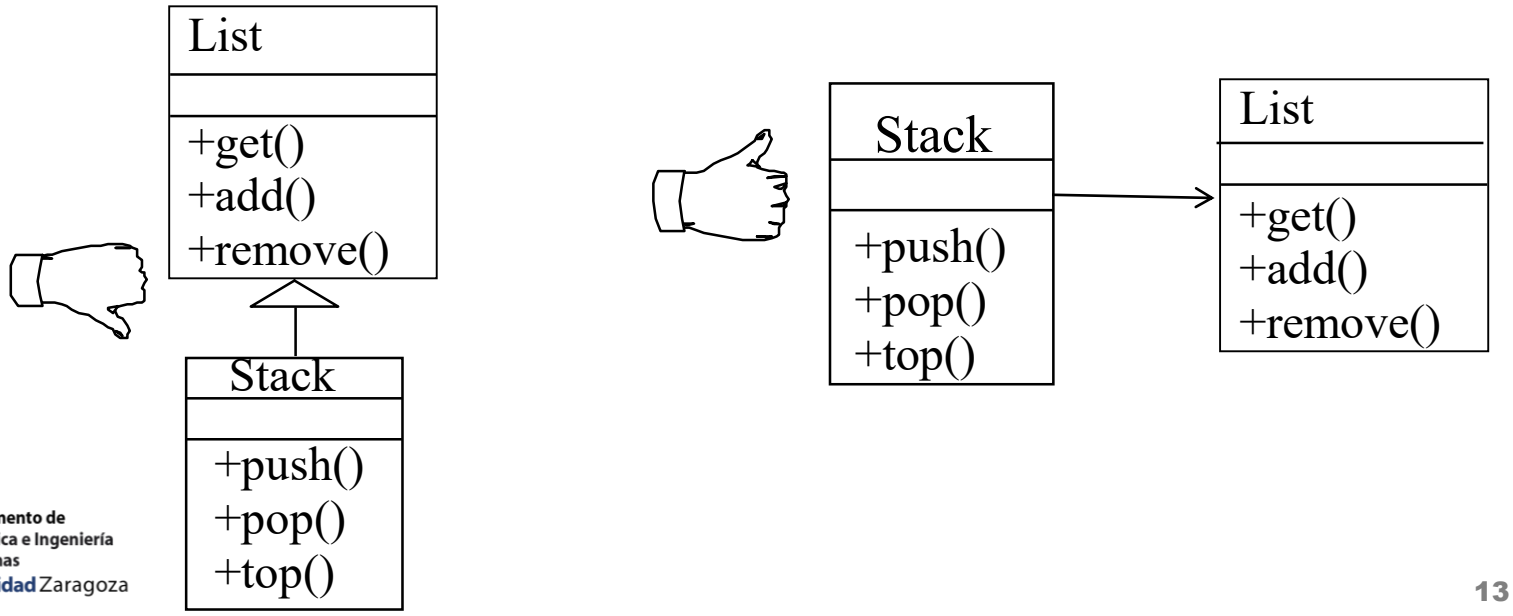
- Una alternativa a la herencia de implementación
- La delegación es una forma de hacer que la composición sea tan potente para la reutilización como la herencia
- En la delegación dos objetos están implicados en el tratamiento de una petición
  - El objeto receptor redirige las operaciones a su delegado
  - El desarrollador puede asegurarse de que el objeto receptor no permite que el cliente utilice mal al objeto delegado



Delegación en lugar de la herencia de implementación

- Herencia: extensión de una clase base con una nueva operación o sobre-escribiendo una operación
- Delegación: cachear una operación y enviarla a otro objeto

¿Cuál de los siguientes modelos es mejor para implementar una pila?



Comparación: delegación vs herencia

	Delegación	Herencia
Ventajas	• Flexibilidad: Cualquier objeto puede ser reemplazado en tiempo de ejecución por otro (con tal de que sea del mismo tipo)	• Uso directo • Soportado por muchos lenguajes de programación (pero normalmente solo herencia simple) • Fácil de implementar nueva funcionalidad
Inconvenientes	• Ineficiencia: Se encapsulan los objetos	• La herencia expone a una subclase a los detalles de su clase padre • Cualquier cambio en la implementación de la clase padre fuerza el cambio en la subclase (requiere recompilación de ambas)

- Usa herencia de interfaz en lugar de herencia de implementación
- Si estas tentado a usar la herencia de implementación, usa la delegación en su lugar

2.3. Patrones de diseño

- Son plantillas de soluciones que los desarrolladores han refinado a lo largo del tiempo para resolver un conjunto de problemas recurrentes
- Tienen 4 elementos principales
  - Nombre bien conocido
  - Motivación (descripción del problema)
  - Solución
    - Estructura (diagrama de clases)
    - Colaboraciones (diagramas de interacción)
  - Consecuencias (ventajas y desventajas del uso del patrón)
- Ver tema separado de patrones de diseño
- Muchos patrones usan una combinación de herencia y delegación

2.4. Librerías, *frameworks*

- Librerías de clases existentes (*off-the-self*)
  - Proporcionan estructuras de datos apropiadas para algoritmos
    - Contenedores, Vectores, Listas, Pilas, Colas, Conjuntos, Árboles ...
  - O gestionan tecnología específica
    - JSAPI (Java Speech API), JTAPI (Java Telephone API), JavaComm
  - Puede ser necesario ajustar las librerías para integrarlas en nuestra aplicación
    - Cambiar el API si se tiene acceso al código fuente
      - Operaciones complejas en base a operaciones de más bajo nivel (añadir clases internas y operaciones)
    - Usar el patrones de encapsulación (*Adapter* o *Bridge*) si no se puede modificar el código fuente

Infraestructura de desarrollo (*Frameworks*)

- Aplicaciones parcialmente reusables que se pueden especializar para desarrollar aplicaciones específicas
- Los *frameworks* se centran en facilitar
  - el desarrollo general de aplicaciones (*Infrastructure Frameworks*)
    - *Integrated Development Environments* (IDE) como Eclipse, IntelliJ, Android SDK, …
  - la utilización de tecnologías particulares (*Middleware Frameworks*)
    - *middleware* para desarrollar aplicaciones distribuidas (ej: DCOM, Java RMI, …), …
  - el desarrollo en dominios de aplicación específicos (*Enterprise Application Frameworks*)
    - gestión empresarial (*Enterprise Resource Planning* - ERP) (ej: SAP), gestión de contenidos (*Content Management Systems* – CMS) (ej: Drupal), …

Frameworks

- Los beneficios clave de las infraestructuras son la reusabilidad y la extensibilidad
  - La reusabilidad se aprovecha del conocimiento del dominio de aplicación y el esfuerzo previo de desarrolladores con experiencia
  - La extensibilidad se proporciona a través de métodos *hook* (extensibles), que se pueden reescribir por la aplicación que extiende el *framework*
    - Desacoplan las interfaces y comportamientos de un dominio de aplicación de las variaciones requeridas en un contexto particular

Librerías vs *Frameworks*

- Librerías de clases:
  - Son menos específicas en el dominio
  - Proporcionan un alcance menor de reusabilidad
  - Son pasivas; no hay restricciones sobre el flujo de control
- *Framework*:
  - Las clases cooperan para una familia de aplicaciones relacionadas
  - Son activos; afectan al flujo de control
- En la práctica, los desarrolladores utilizan ambas:
  - Los *frameworks* utilizan con frecuencia librerías de clases para simplificar el desarrollo
  - Los controladores/gestores de eventos en los *frameworks* utilizan librerías de clases para tareas básicas (ej: procesamiento de cadenas, gestión de ficheros, análisis numérico ….)

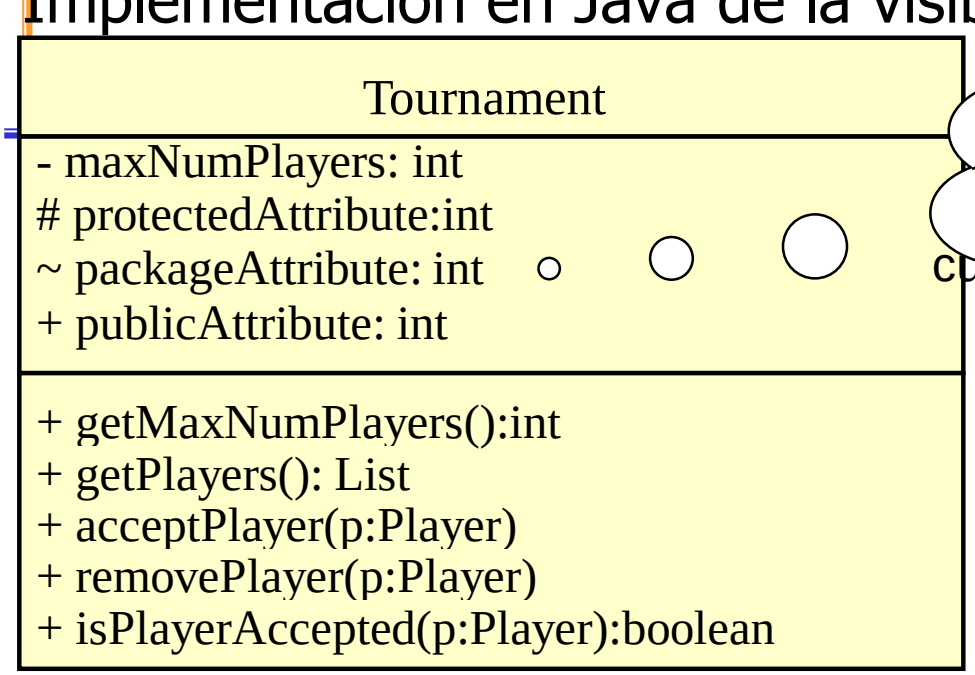
3. Especificación de interfaces

- Durante el análisis se identifican atributos y operaciones de los objetos de dominio sin especificar sus tipos o sus parámetros
- Durante el diseño de objetos:
  1. Se añade información de visibilidad
  2. Se añade tipo y signatura
  3. Se añaden contratos (restricciones sobre las clases)

3.1. Visibilidad

- Hay 3 roles diferentes para los desarrolladores durante el diseño de objetos
  - Usuario, implementador, creador de extensiones/subclases
- UML define 3 niveles de visibilidad:
  - Privada (Implementador de clases):
    - Un atributo privado solo puede ser accedido por la clase donde se define
    - Una operación privada solo puede ser invocada por la clase donde se define
    - Los atributos y operaciones privados no son accesibles por las subclases o cualquier otra clase
  - Protegido (creador de extensiones/subclases):
    - Un atributo o operación protegido puede ser accedido por la clase donde se define y cualquier descendiente de la clase
  - Publico (Usuario de la clase):
    - Un atributo u operación publico puede ser accedido por cualquier clase

Implementación en Java de la visibilidad UML



Miembros amistosos o de paquete (accesibles desde cualquier clase del mismo paquete)

```
public class Tournament {
  private int maxNumPlayers;
  protected int protectedAttribute;
  int packageAttribute;
  public int publicAttribute;
  public Tournament(League l, int maxNumPlayers)
  public int getMaxNumPlayers() {…};
  public List getPlayers() {…};
  public void acceptPlayer(Player p) {…};
  public void removePlayer(Player p) {…};
  public boolean isPlayerAccepted(Player p) {…};
}
```

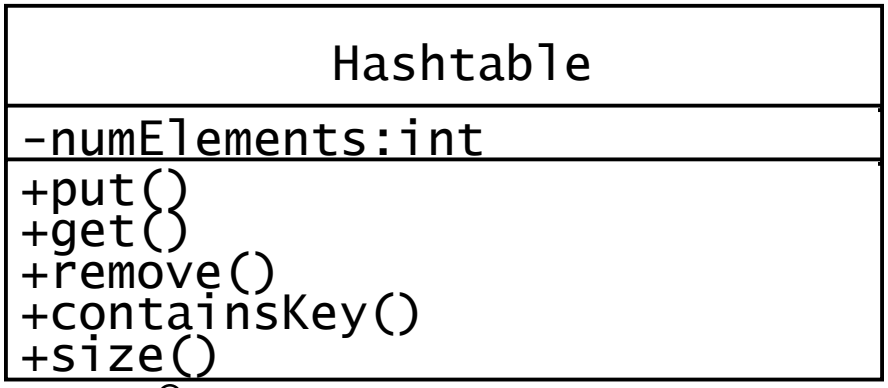
Recomendaciones para ocultación de información

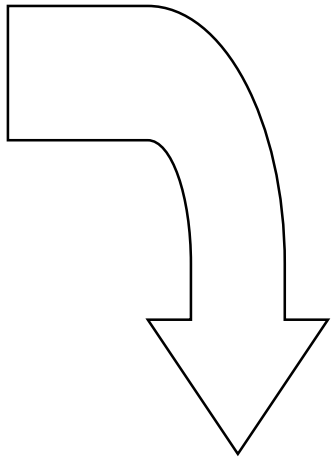
- Define cuidadosamente la interfaz pública de las clases así como de los subsistemas (*façade*)
- Aplica siempre el principio de "Necesita saber"
  - Solo si alguien necesita acceder a la información, hazla pública, pero siempre a través de canales bien definidos (de forma que seas siempre consciente del acceso)
- Cuanto menos se conozca sobre una operación
  - Será menos probable que se vea afectada por algún cambio
  - Será más fácil cambiar la clase
- Balanza: ocultación de información vs eficiencia
  - Acceder a un atributo privado mediante métodos de acceso intermedios puede ser demasiado lento (ej: sistema de tiempo real, o juegos)

Principios de diseño respecto a ocultación de información

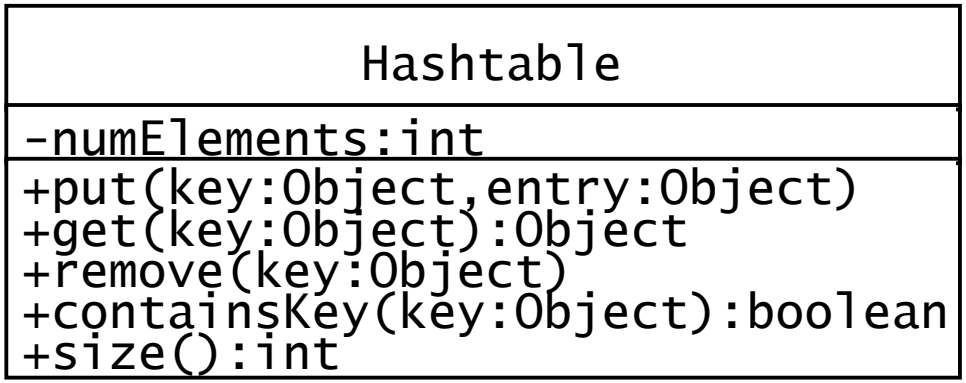
- Solo las operaciones de una clase deben manipular sus atributos
  - Acceso a atributos únicamente vía operaciones (ej: métodos *getXXX setXXX*)
- Ocultar los objetos en la frontera del subsistema
  - Definir interfaces que interactúan entre el sistema y el mundo externo, así como entre subsistemas
- No aplicar una operación al resultado de otra operación.
  - Escribir una nueva operación que combine ambas operaciones

3.2. Tipos y signatura





Los atributos y las operaciones sin información de tipos son aceptables durante el análisis

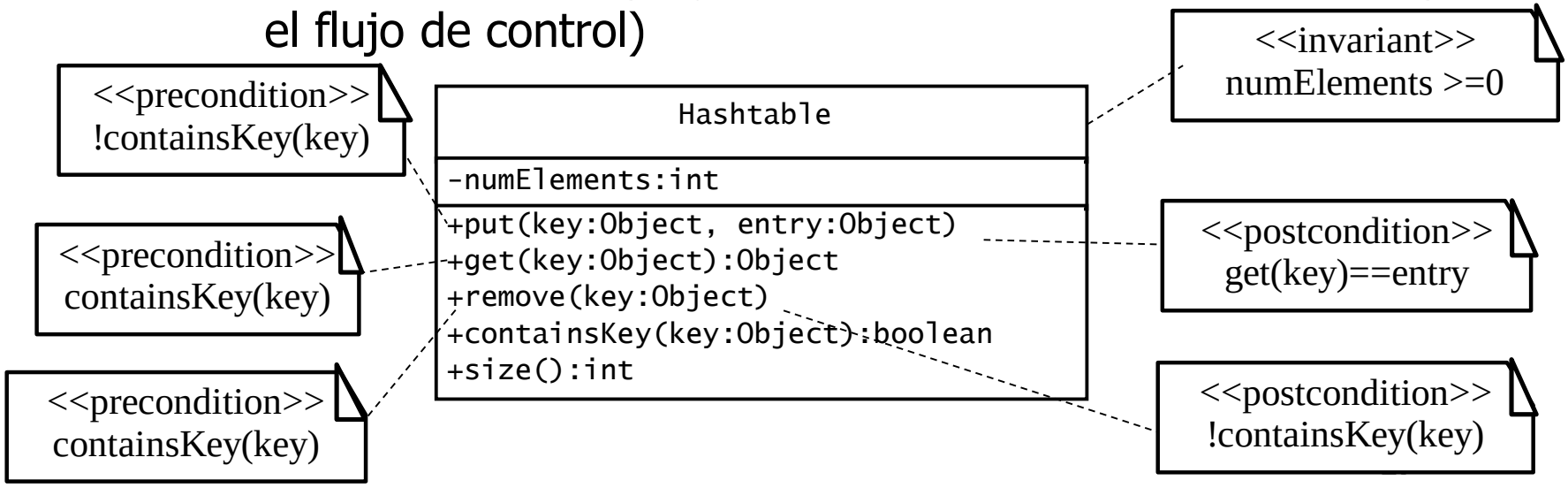


3.3. Contratos

- Los contratos sobre una clase permiten al invocador y al invocado compartir las mismas asunciones acerca de la clase
- Los contratos incluyen 3 tipos de restricciones:
- Invariante:
  - Un invariante que es siempre cierto para todas las instancias de una clase
  - Los invariantes son restricciones asociadas a clases o interfaces
- Precondición:
  - Las precondiciones son predicados asociados a una operación específica y deben ser ciertos antes de invocar a la operación.
  - Las precondiciones se usan para especificar restricciones que un invocador debe cumplir antes de llamar a una operación
- Postcondición:
  - Las postcondiciones son predicados asociados con una operación específica y que deben ser ciertos después de haber invocado a una operación
  - Las postcondiciones se usan para especificar restricciones que el objeto debe asegurar después de la invocación de la operación

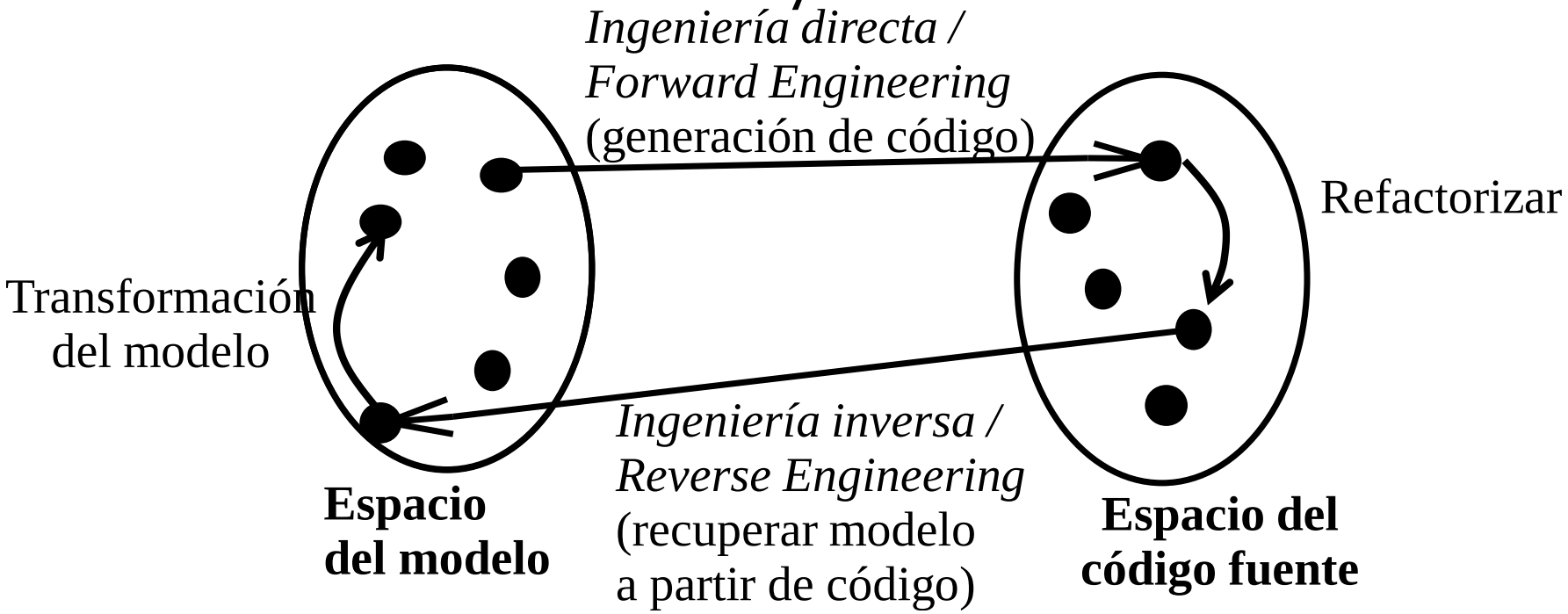
Expresión de restricciones en modelos UML

- OCL (*Object Constraint Language*)
  - https://www.omg.org/spec/OCL/
  - OCL permite que las restricciones se especifiquen formalmente sobre elementos del modelo individuales o sobre grupos de elementos del modelo
  - Una restricción se expresa como una expresión OCL devolviendo el valor verdad o falso
    - OCL no es un lenguaje procedimental (no puede restringir el flujo de control)



4. Transformando el diseño a la implementación

- El diseño de objetos se sitúa entre el diseño del sistema y la implementación
- Posibles transformaciones y modificaciones



Ingeniería de ida y vuelta (*Roundtrip Engineering*): *Forward Engineering + Reverse Engineering*

Reingeniería: reconstruir el modelo del sistema a partir del código para añadir nueva funcionalidad

Procedimiento disciplinado

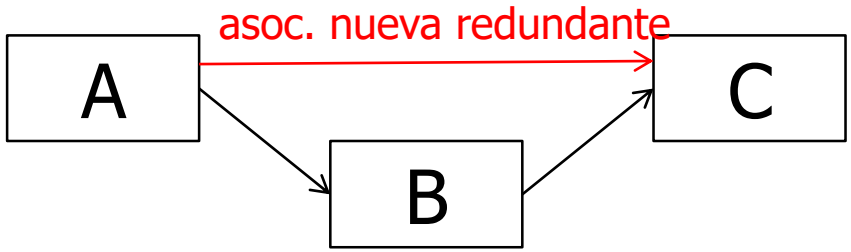
- Un diseño de objetos malo conduce a una pésima implementación del sistema
- Se debe adoptar un procedimiento disciplinado de transformación para evitar la degradación del sistema
  - Optimizar el modelo de objetos para conseguir los requisitos de eficiencia
  - Implementar el modelo de clases
    - Herencia, asociaciones, contratos, paquetes
  - Establecer la correspondencia de modelos de objetos al esquema de almacenamiento

4.1. Optimización del diseño de objetos

- Transformar el diseño de objetos para conseguir criterios de eficiencia como tiempo de respuesta o utilización de memoria
- Las optimizaciones de diseño son una parte importante en la fase del diseño de objetos:
  - El modelo del análisis (de requisitos) es semánticamente correcto pero a menudo demasiado ineficiente si se implementa directamente
- Actividades de optimización durante el diseño de objetos:
  - Añadir asociaciones redundantes vs eliminar caminos muertos
  - Conversión de clases en atributos
  - Almacenar atributos derivados
- Un diseñador de objetos debe hacer un balance entre eficiencia y claridad
  - Las optimizaciones oscurecen los modelos

Añadir asociaciones redundantes vs eliminar caminos muertos

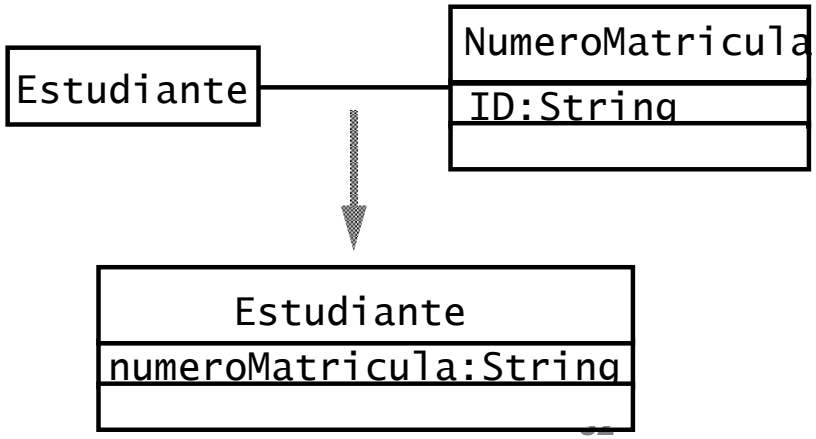
- Añadir asociaciones redundantes
  - Objetivo: minimizar el coste de los accesos
  - ¿Dónde?
    - ¿Cuáles son las operaciones más frecuentes? (lectura del sensor de datos)
    - ¿Con qué frecuencia se invoca a la operación? (30 veces al mes, cada 50 milisegundos)



- En otras ocasiones puede ocurrir lo contrario
  - Eliminación de caminos muertos en base a frecuencia de recorrido de caminos

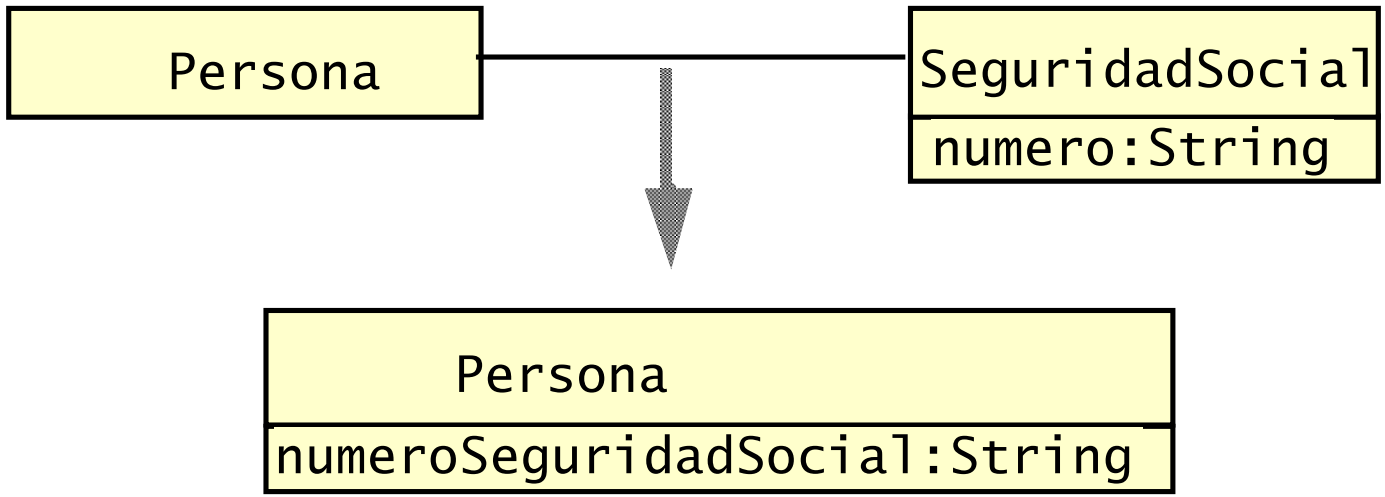
Conversión de clases en atributos (I)

- Compactar o no compactar: ¿Atributo o asociación?
- Decisiones del diseño de objetos:
  - Implementar una entidad como un atributo embebido
  - Implementar una entidad como una clase separada con asociaciones a otras clases
- Las asociaciones son más flexibles que los atributos pero con frecuencia introducen una indirección innecesaria
- Ejemplo
  - Cada estudiante recibe un número el primer día de la Universidad



Conversión de clases en atributos (II)

- ¿Compactar o no compactar?
  - Compacta una clase en un atributo si las únicas operaciones definidas sobre los atributos son *set()* y *get()*
    - Eliminar objetos que no implican un comportamiento especial
- Ejemplo: conversión de objetos en atributos



Almacenar atributos derivados

- Objetivo: Ahorrar tiempo de computación
- Ejemplo: Caché de base de datos o de resultados obtenidos después de cálculos costosos
- Problema con los atributos derivados:
  - Se deben actualizar cuando los valores base cambian
  - Hay 3 formas de tratar el problema de actualización:
    - Código explícito: El implementador determina los atributos derivados afectados (*push*)
    - Cálculo periódico: Recalcular los atributos derivados ocasionalmente (*pull*)
    - Valor activo: Un atributo puede tener asignados un conjuntos de valores dependientes que se actualizan automáticamente cuando el valor activo cambia (notificación, trigger/disparador de datos)

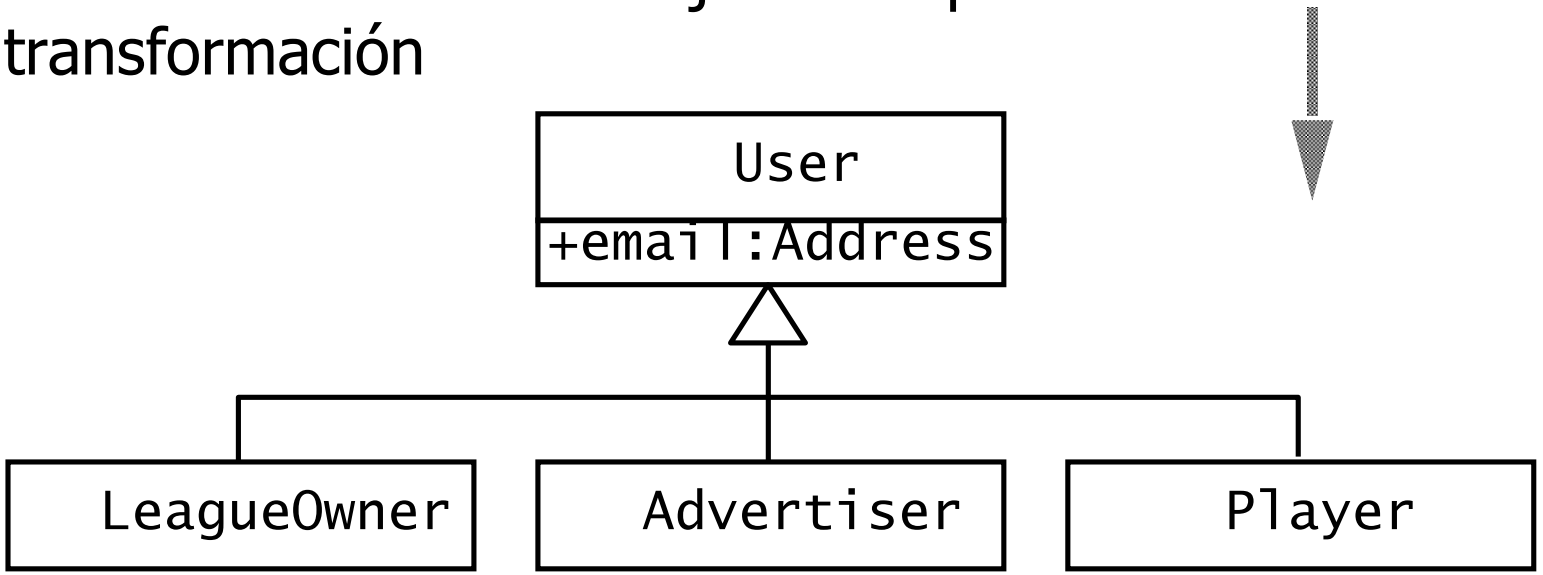
4.2. Implementación del modelo de clases
4.2.1. Preparar la herencia

- Abstraer comportamiento común de un grupo de clases
  - Si un conjunto de operaciones o atributos se repiten en 2 clases, estas pueden ser instancias de una clase más general
  - Tener cuidado de que se trata de una herencia de interfaz y no de una simple herencia de implementación
    - Si es herencia de implementación, recordar que es más interesante la delegación
- Reorganizar y ajustar clases y operaciones para facilitar la herencia

Ejemplo de abstracción

- Modelo original

- Modelo del diseño de objetos después de la transformación

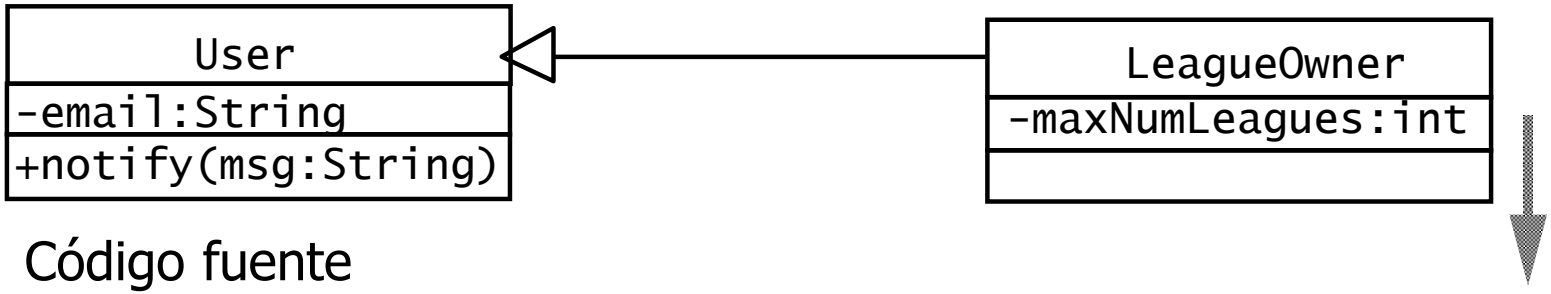


Reorganizar y ajustar clases y operaciones para facilitar la herencia

- Atributos/Operaciones similares en las clases tienen diferentes nombres
  - Renombra atributos y operaciones
- Todas las operaciones deben tener la misma signatura
  - Algunas operaciones tienen menos argumentos que otras
    - Utiliza la sobrecarga para crear operaciones con misma signatura que redirijan a operaciones originales con menos argumentos
  - Operaciones definidas en una clase pero no en otra
    - Utiliza funciones virtuales, y redefinición de funciones
    - En Java no existe ese problema: por defecto, todos los métodos instancia (que no son finales ni privados) son funciones virtuales (se pueden redefinir)

Ejemplo de Forward Engineering (generación de código)

- Modelo del diseño de objetos



- Código fuente

```
public class User {
  private String email;
  public String getEmail() {
    return email;
  }
  public void setEmail(String value){
    email = value;
  }
  public void notify(String msg) {
    // ....
  }
  /* Other methods omitted */
}
```

```
public class LeagueOwner extends User {
  private int maxNumLeagues;
  public int getMaxNumLeagues() {
    return maxNumLeagues;
  }
  public void setMaxNumLeagues (int value) {
                              maxNumLeagues = value;
  }
  /* Other methods omitted */
}
```

En ocasiones la herencia surge a posteriori

- Es posible que tras un proceso de refactorización (modificaciones sobre el código fuente) surja la herencia
- ¿Por qué son deseables las superclases?
  - Incrementan modularidad, extensibilidad y reusabilidad
    - Muchos patrones de diseño utilizan superclases
  - Mejoran la gestión de configuraciones
- Nota: ¡Aunque la herencia de interfaz se detecte durante la implementación, se deberían actualizar los modelos del diseño de objetos (ingeniería inversa)!

Ejemplo de refactorización (I)

```
public class Player {
  private String email;
  //...
}

public class LeagueOwner {
  private String eMail;
  //...
}

public class Advertiser {
  private String email_address;
  //...
}
```

```
// herencia (se encuentra campo común)
public class User {
  protected String email;
}

public class Player extends User {
  //...
}

public class LeagueOwner extends User {
  //...
}

public class Advertiser extends User {
  //...
}
```

Ejemplo de refactorización (II)

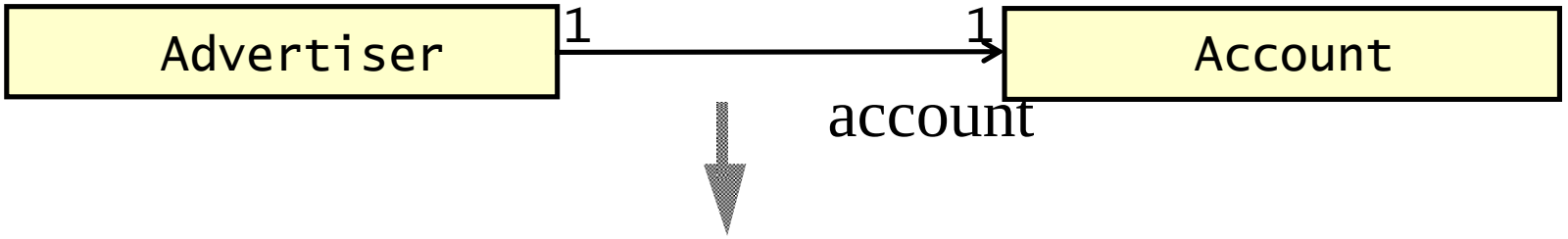
```
public class User {
  protected String email;
}
public class Player extends User {
  public Player(String email) {
    this.email = email;
  }
}
public class LeagueOwner extends User{
  public LeagueOwner(String email) {
    this.email = email;
  }
}
public class Advertiser extendsUser{
  public Advertiser(String email) {
    this.email = email;
  }
}
```

```
// se encuentra constructor común
public class User {
  private String email;
  public User(String email) {
    this.email = email;
  }
}
public class Player extends User {
  public Player(String email) {
    super(email);
  }
}
public class LeagueOwner extends User {
  public LeagueOwner(String email) {
    super(email);
  }
}
public class Advertiser extends User {
  public Advertiser(String email) {
    super(email);
  }
}
```

4.2.2. Reestructuración/Implementación de asociaciones

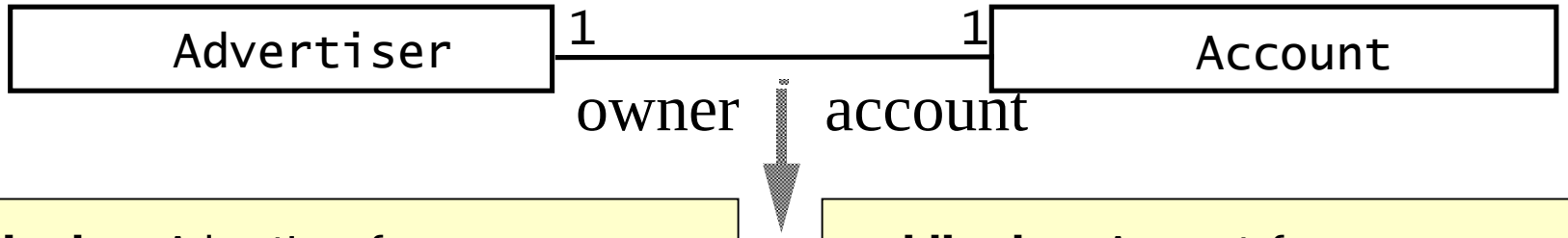
- Estrategia para implementar asociaciones:
  - Ser tan uniforme como sea posible
  - Decisión individual para cada asociación
- Ejemplo de implementación uniforme
  - Asociación 1-1
    - Los nombres de los roles se tratan como atributos en las clases y se traducen en referencias
  - Asociación 1-muchos
    - "Muchos ordenados": traducir a un vector / lista
    - "Muchos desordenados": traducir a un conjunto
  - Asociación calificada
    - Traducir a una tabla Hash

Asociaciones unidireccionales 1:1




```
public class Advertiser {
   private Account account;
   public Advertiser() {
       account = new Account();
   }
   public Account getAccount() {
       return account;
   }
}
```

Asociaciones bidireccionales 1:1



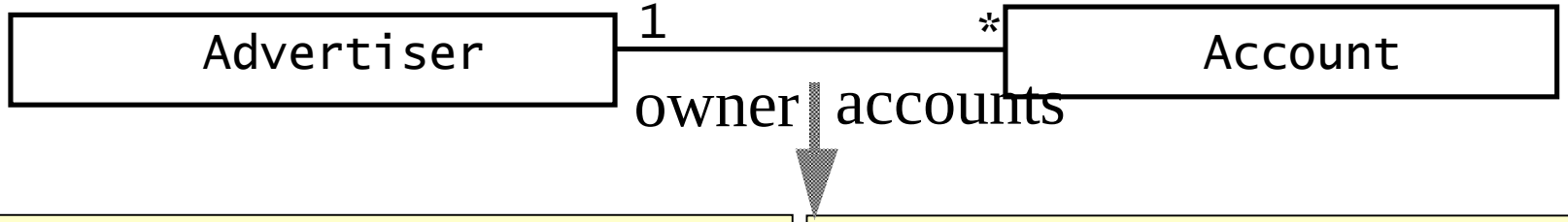

```
public class Advertiser {
  /* The account field is initialized
   * in the constructor and never
   * modified. */
  private Account account;

  public Advertiser() {
    account = new Account(this);
  }
  public Account getAccount() {
    return account;
  }
}
```

```
public class Account {
  /* The owner field is initialized
   * during the constructor and
   * never modified. */
  private Advertiser owner;

  public Account(Advertiser owner) {
    this.owner = owner;
  }
  public Advertiser getOwner() {
    return owner;
  }
}
```

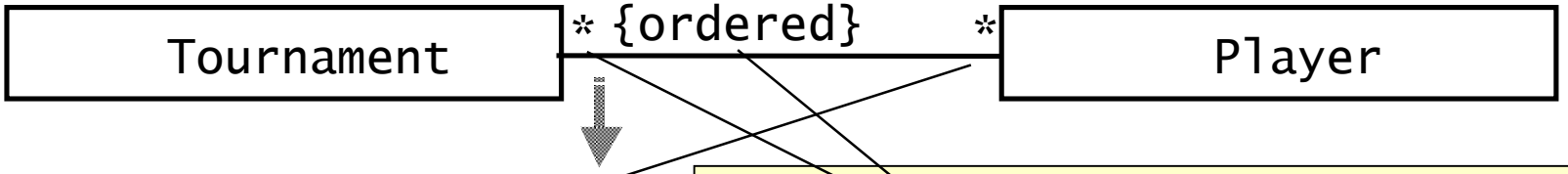
Asociaciones bidireccionales 1:n




```
public class Advertiser {
  private Set accounts; // sin elementos repetidos
  public Advertiser() {
    accounts = new HashSet();
  }
  public void addAccount(Account a) {
    accounts.add(a); // no afectan llamadas repetidas
    a.setOwner(this);
  }
 public void removeAccount(Account a) {
    accounts.remove(a);
    if ((a!=null)&&(a.getOwner()==this)) {
  // la eliminación se ha iniciado en esta clase
      a.setOwner(null);
    }
  }
}
```

```
public class Account {
  private Advertiser owner;
  public void setOwner(Advertiser newOwner) {
    if (owner != newOwner) {
      Advertiser old = owner;
      owner = newOwner;
      if (newOwner != null)
        newOwner.addAccount(this);
      if (old != null)
        old.removeAccount(this);
    }
  }
  public Advertiser getOwner() {
    return owner;
  }
}
```

Asociaciones bidireccionales, muchos a muchos




```
public class Tournament {
  private List players;
  public Tournament() {
    players = new ArrayList();
  }
  public void addPlayer(Player p) {
    if (!players.contains(p)) {
      players.add(p);
      p.addTournament(this);
    }
  }
}
```

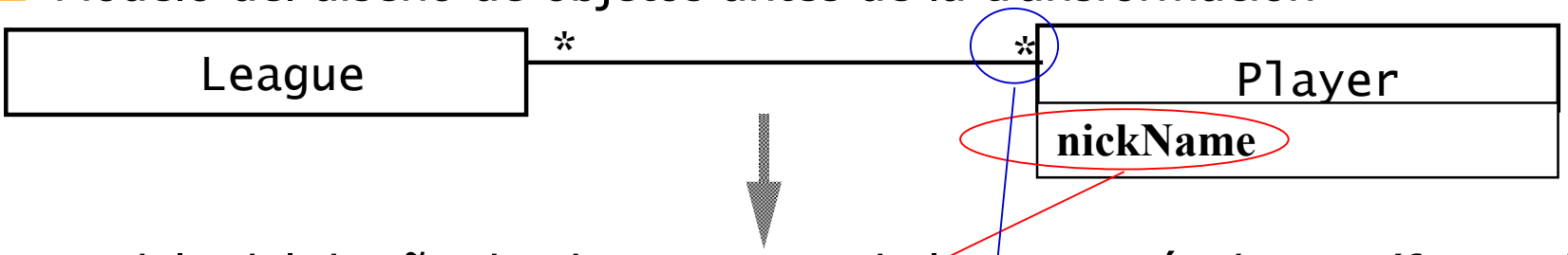
```
public class Player {
  private List tournaments;
  public Player() {
    tournaments = new ArrayList();
  }
  public void addTournament(Tournament t) {
    if (!tournaments.contains(t)) {
      tournaments.add(t);
      t.addPlayer(this);
    }
  }
}
```

Ejercicio: ¿Cómo implementar la eliminación de jugadores en la clase *Tournament* y la eliminación de torneos en la clase *Player*?

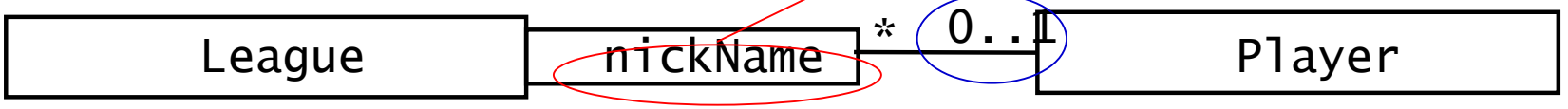
Solución en enunciado de ejercicio 4, examen de 1ª convocatoria del curso 2017-18.

Asociación bidireccional con calificador

- Se utilizan para reducir la multiplicidad de un muchos en una asociación 1:n o n:n
- El calificador de la asociación es un atributo de la clase en el lado "muchos" tal que ese nombre es único dentro del contexto de la asociación, no necesariamente único globalmente
- Modelo del diseño de objetos antes de la transformación



- Modelo del diseño de objetos antes de la ingeniería directa (forward)



- Código fuente después de la ingeniería directa (forward)

Asociación bidireccional con calificador

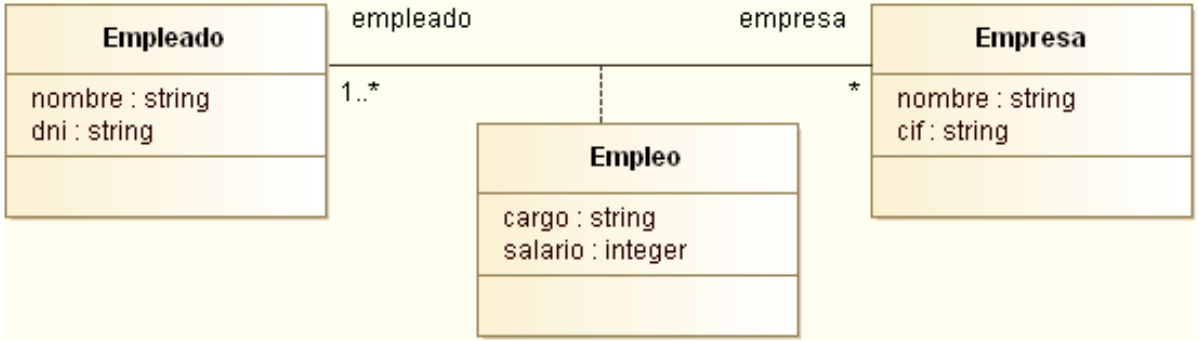
- Código fuente después de la ingeniería directa (forward)

```
public class League {
  private Map players;
  ...
  public void addPlayer
        (String nickName, Player p) {
    if (!players.containsKey(nickName)) {
      players.put(nickName, p);
      p.addLeague(nickName, this);
    }
  }
}
```

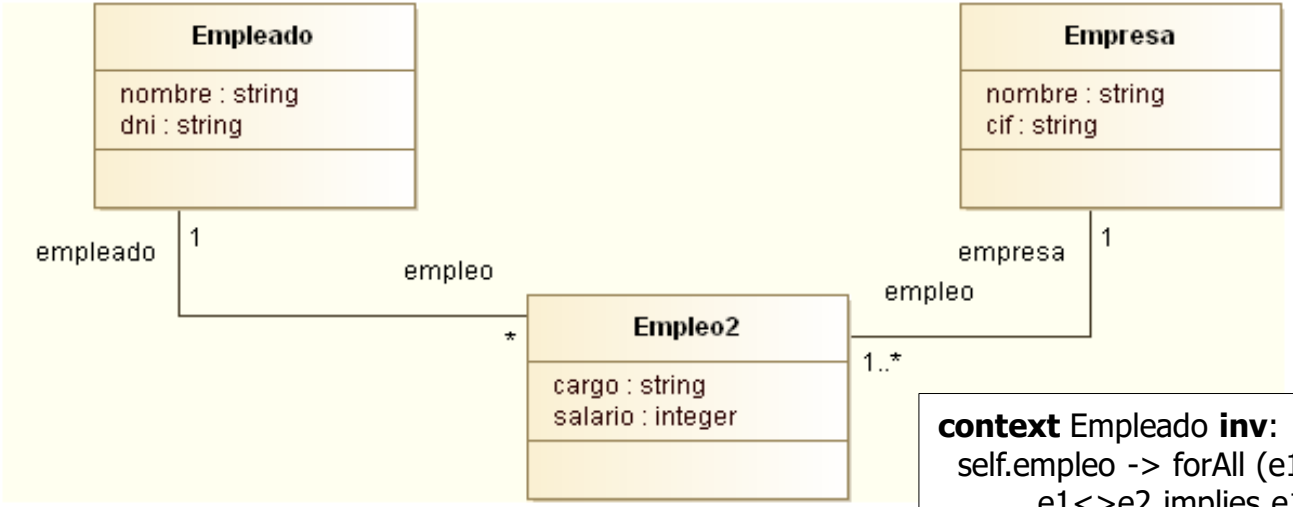
```
public class Player {
  private Map leagues;
  ...
  public void addLeague
        (String nickName, League l) {
    if (!leagues.containsKey(l)) {
      leagues.put(l, nickName);
      l.addPlayer(nickName, this);
    }
  }
}
```

Transformación de una clase asociación

- Modelo del diseño de objetos antes de la transformación



- Modelo de diseño de objetos después de la transformación: 1 clase y 2 asociaciones binarias (+ restricción OCL)

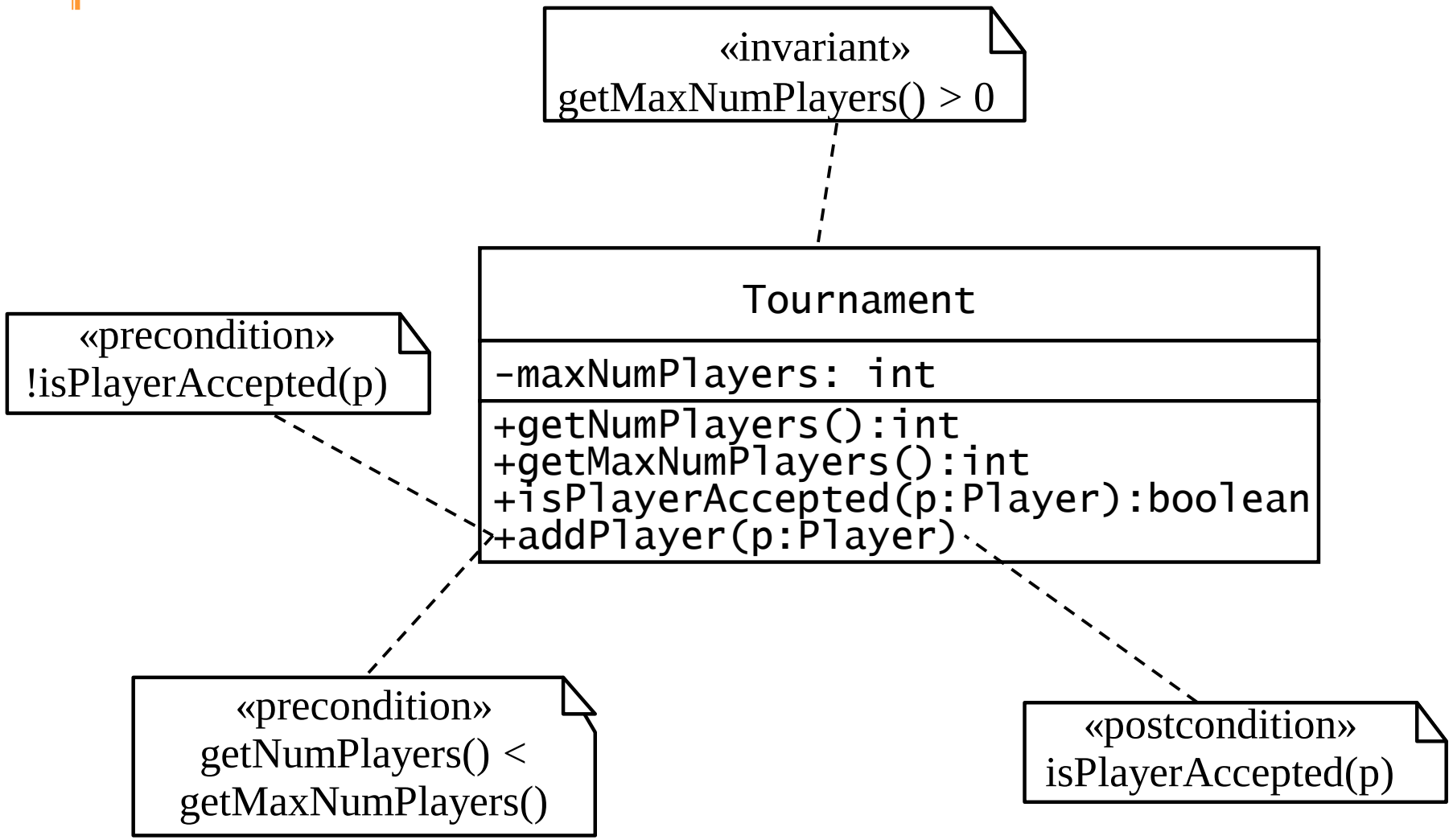


context Empleado **inv**:
self.empleo -> forAll (e1,e2: Empleo2 |
e1<>e2 implies e1.empresa.cif <> e2.empresa.cif)

4.2.3. Contratos

- Las excepciones como bloques constructivos para violaciones de contratos
- Muchos lenguajes orientados a objeto (incluido Java) no tiene soporte directo para contratos
- Sin embargo, podemos utilizar sus mecanismos de excepción como bloques constructivos para señalar y manejar violaciones de contratos
- En Java usamos el mecanismo *try-throw-catch*
- Ejemplo:
  - Pensemos que la operación *addPlayer()* de *Tournament* es invocada con un jugador que ya es parte del *Tournament*
  - En este caso *addPlayer()* debería lanzar una excepción del tipo *KnownPlayerException*

Una implementación completa del contrato *Tournament.addPlayer()*



```
public class Tournament {
  public void addPlayer(Player p)  throws KnownPlayerException, UnknownPlayerException,
                        IllegalNumPlayersException,IllegalMaxNumPlayersException {
    // chequear precondición: !isPlayerAccepted(p)
    if (isPlayerAccepted(p)) throw new KnownPlayerException(p);
    // chequear precondición: getNumPlayers() < getMaxNumPlayers()
    if (getNumPlayers()==getMaxNumPlayers()) throw new IllegalNumPlayersException(getNumPlayers());
     // add player
     ...
     // chequear postcondicion: isPlayerAccepted(p)
     if (!isPlayerAccepted(p))  throw new UnknownPlayerException(p);
     // chequear invariante: getMaxNumPlayers() > 0
     if (getMaxNumPlayers()<=0) throw new IllegalMaxNumPlayersException(getMaxNumPlayers());
   }
}
public class TournamentForm {// Formulario para la edición de jugadores
  private Tournament tournament;
  private ArrayList players;
  public void processPlayerApplications() {
   // procesar las peticiones de incluir jugadores
   for (Iteration i = players.iterator(); i.hasNext();) {
     try {        // Añadir jugador
       tournament.addPlayer((Player)i.next());
       } catch (KnownPlayerException e) {
                  // Si se ha producido una excepción, hacer log en la consola
                  ErrorConsole.log(e.getMessage());
       } catch (…) {…}
   }
  }
}
```

Ejemplo: mecanismo try-throw-catch de Java

Implementación de un contrato

- Para cada operación en el contrato, haz lo siguiente
- Comprobar precondición:
  - Comprobar la precondición antes de empezar el método con un test que lanza una excepción si la precondición es falsa
- Comprobar postcondición:
  - Comprobar la postcondición al final del método y lanzar una excepción si se viola el contrato
  - Si no se cumplen varias postcondiciones, lanzar una excepción solo con la primera violación
- Comprobar invariante
  - Comprobar invariantes a la misma vez que las postcondiciones
- Tratar la herencia:
  - Encapsular el código de comprobación de precondiciones y postcondiciones en métodos separados para que puedan ser invocados desde las subclases

Recomendaciones para mapear contratos a excepciones

- Se pragmático si no tienes mucho tiempo
- Omitir el código de comprobación de postcondiciones e invariantes
  - Normalmente es redundante con el código que realiza la funcionalidad de la clase
- Omitir el código de comprobación para métodos privados o protegidos
- Centrarse en componentes con la vida más larga
  - Centrarse en objetos entidad, no sobre objetos frontera asociados con la interfaz de usuario
- Reutiliza el código de comprobación de restricciones
  - Muchas operaciones tiene precondiciones similares
  - Encapsular código de comprobación de restricciones en métodos para que puedan compartir las mismas clases de excepciones

4.2.4. Paquetes

- Empaqueta el diseño en unidades físicas discretas para que puedan ser editadas, compiladas, lincadas, reutilizadas
- Construir módulos físicos
  - Idealmente, usar un paquete para cada subsistema
  - La descomposición de sistemas puede no ser buena para la implementación
- 2 principios de diseño para empaquetar
  - Minimizar el acoplamiento:
    - Las clases en relaciones cliente-proveedor están normalmente escasamente acopladas
    - Un gran número de parámetros en algunos métodos significa un acoplamiento fuerte (> 4-5)
    - Evitar datos globales (variables globales)
  - Maximizar la cohesión:
    - Clases estrechamente conectadas por asociaciones => mismo paquete

Recomendaciones de empaquetamiento

- Cada servicio de subsistema se hace disponible por uno o más objetos interfaz dentro del paquete
- Comienza con un objeto interfaz para cada servicio de subsistema
  - Intenta limitar el número de operaciones de interfaz (7+-2)
- Si el servicio de subsistema tiene muchas operaciones, reconsidera el número de objetos interfaz
- Si tienes muchos objetos interfaz, reconsidera el número de subsistemas
- Diferencia entre objetos interfaz y las interfaces en Java
  - Objeto interfaz: utilizado durante el análisis de requisitos, diseño del sistema y diseño de objetos. Denota el API de un servicio
  - Interfaz Java: Utilizada durante la implementación en Java (Una interfaz Java puede o no implementar un objeto interfaz)

4.3. Correspondencia de modelos de objetos al esquema de almacenamiento

- Los modelos de objetos UML se pueden mapear a bases de datos relacionales:
  - Hay alguna degradación porque todas las construcciones UML (elementos UML) deben ser mapeadas a la única construcción en bases de datos relacionales – la tabla
- Transformaciones similares a las ya vistas en asignaturas como Bases de Datos para la transformación de un modelo entidad relación a un modelo relacional

Transformación del modelo OO al modelo relacional

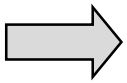
- Correspondencias: orientación a objeto – modelo relacional

Modelo OO	**Modelo relacional**
clase	se transforma en *relación (=tabla)*
atributo	se transforma en un *atributo(= columna)* de una relación(=tabla)
instancia de una clase	representa una *tupla(=fila)* en una relación(=tabla)
asociación uno-a-muchos	se implementa como una *clave ajena*
asociación muchos-a-muchos	se traduce en una *relación(=tabla)*
métodos	no tienen correspondencia

Correspondencia clase-relación

- Conjunto de atributos que permiten identificar unívocamente una instancia de una clase ⇒ *clave candidata* para identificar las tuplas de una relación
- Clave candidata que se utiliza realmente para la identificación de instancias ⇒ *clave primaria*

Libro
+titulo[1] : tpCadena +autor[1] : tpCadena +numReferencia[1] : entero +fechaPublicacion[1] : tpFecha +resumen[0..1] : tpCadena



notación gráfica

Libro	
PK	**numReferencia**
	titulo **autor** **fechaPublicacion** resumen

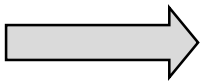
notación textual

LIBRO(titulo:tpCadena,
autor: tpCadena,
numReferencia: entero,
fechaPublicacion:tpFecha,
resumen:tpCadena)

CP:{numReferencia}
CAlt:{autor, titulo}
VNN:{fechaPublicacion}

User
+name:String +login:String +email:String

definir nuevo atributo para identificar tupla si es necesario



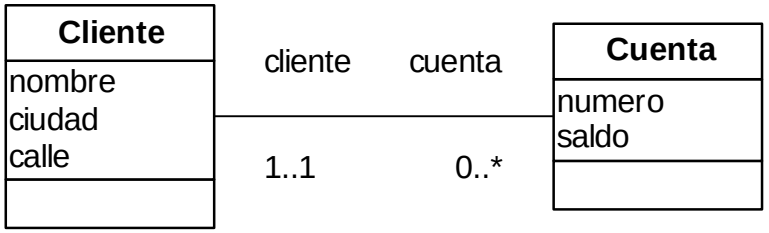
User	
PK	**id**
	login name email

User (ejemplo de tabla en BD)

id	name	login	email]
1	John Smith	jsmith	jsmith@gmail.com
2	Mary Jones	mjones	mjones@gmail.com

Transformación de asociaciones

- Varias posibilidades dependiendo del tipo de multiplicidad
- Si la asociación es de *uno a muchos*
  - Los atributos clave de la relación izquierda (correspondiente a la clase izquierda) se añaden a la relación que corresponde a la clase derecha (con multiplicidad máxima *muchos*)
  - Se especifica una clave ajena en la relación derecha
    - Clave ajena= atributo (o conjunto de atributos) que referencia a la clave primaria de otra tabla



Cliente	
PK	**nombre**
	calle ciudad

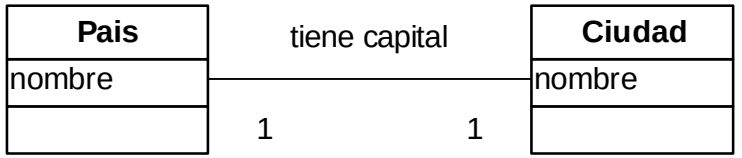
Cuenta	
PK	**numero**
FK1	**nombre** saldo

CLIENTE(nombre:tpCadena,
calle:tpCadena, ciudad:tpCadena)
CP: {nombre}

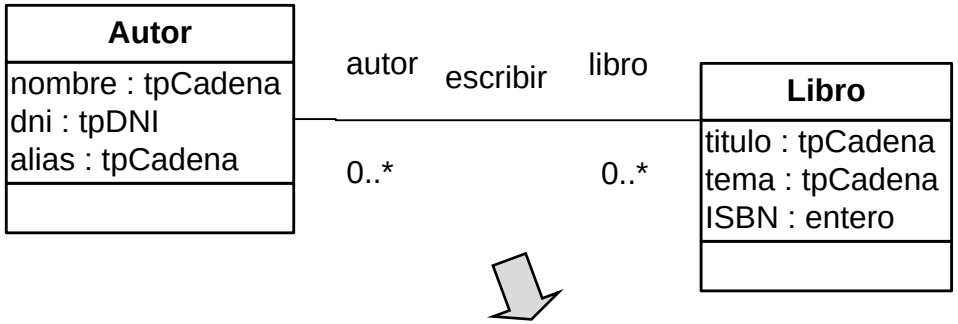
CUENTA(numero:entero,
saldo:entero,nombre:tpCadena)
CP:{numero}
VNN:{nombre}
CAj:{nombre} REFERENCIA a CLIENTE

Transformación de asociaciones

- Si la asociación es de *uno a uno* podemos elegir cualquiera de los extremos



- Si la asociación es de *muchos a muchos*
  - Construir nueva relación cuyos atributos son las claves primarias de las relaciones (=tablas) resultantes de transformar las clases que intervienen en la asociación



AUTOR(DNI:tpDNI,nombre:tpCadena
,alias:tpCadena)
CP:{DNI}

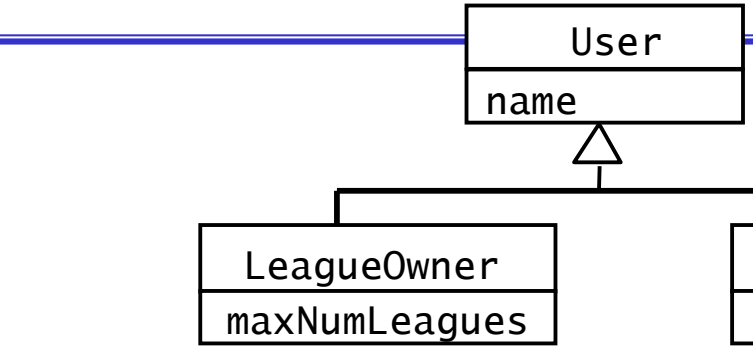
LIBRO(ISBN:entero,titulo:tpCadena
,tema:tpCadena)
CP:{ISBN}

ESCRIBIR(DNI:tpDNI,ISBN:entero)
CP:{DNI,ISBN}
CAj:{DNI} REFERENCIA a AUTOR
CAj:{ISBN} REFERENCIA a LIBRO

Transformación de la herencia

- Las bases de datos relacionales no soportan la herencia
- 2 posibilidades para almacenar la herencia en el esquema de base de datos relacional
  - Con una relación(=tabla) separada (*vertical mapping*)
    - Los atributos de la superclase y las subclases se almacenan en tablas diferentes
    - Ventaja: podemos añadir atributos a la superclase fácilmente añadiendo columnas a la tabla de la superclase
    - Desventaja: para recuperar los atributos de un objeto se requiere una operación *join*
  - Duplicando las columnas (*horizontal mapping*)
    - No hay tabla para la superclase
    - Cada subclase se almacena en una tabla conteniendo los atributos de la subclase y los atributos de la superclase
    - Ventaja: los objetos individuales no están fragmentados a través de un número de tablas, y por tanto las consultas son más rápidas
    - Desventaja: modificar el esquema de bases de datos es más complejo y propenso a errores

Transformación de la herencia



vertical mapping

horizontal mapping

User	
PK	**id**
	name ...

LeagueOwner	
PK,FK1	**id**
	maxNumLeagues

Player	
PK,FK1	**id**
	credits

LeagueOwner	
PK	**id**
	name ... maxNumLeagues

Player	
PK	**id**
	name ... credits

Tabla User

id	name	...
56	zoe	
79	john	

Tabla LeagueOwner

id	maxNumLeagues	...
56	12	

Tabla Player

id	credits	...
79	126	

Tabla LeagueOwner

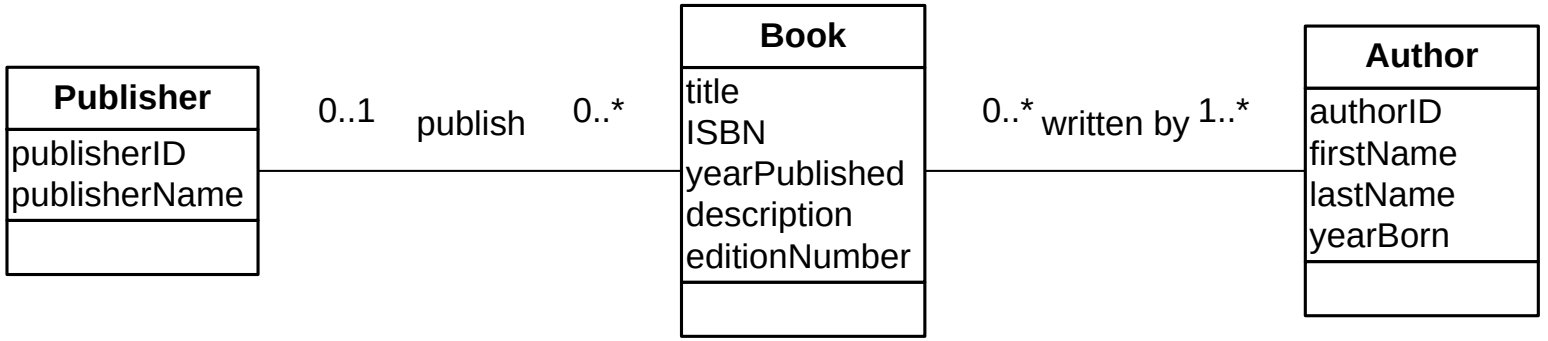
id	name	maxNumLeagues	...
56	zoe	12	

Tabla Player

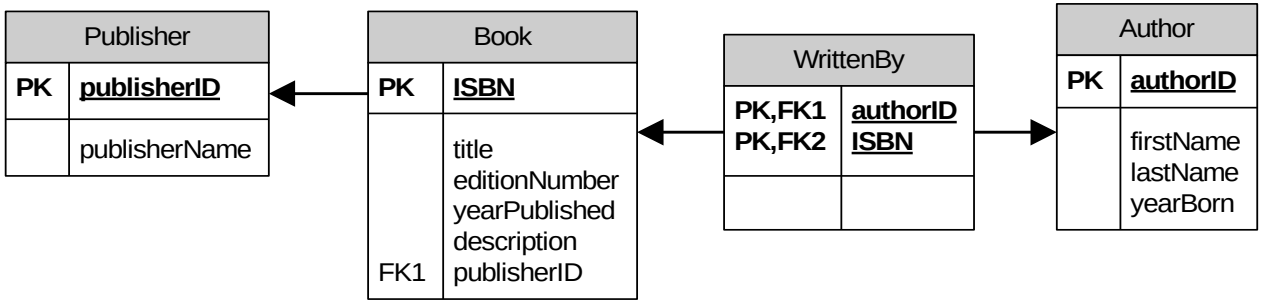
id	name	credits	...
79	john	126	

Ejemplo de una base de datos de libros

Diseño conceptual (modelo OO)



Diseño lógico (modelo relacional)



Creación de base de datos, diseño físico

```
CREATE TABLE PUBLISHER (
 PUBLISHERID VARCHAR(100)
 , PUBLISHERNAME VARCHAR(200)
 , CONSTRAINT PK_PUBLISHER PRIMARY KEY (PUBLISHERID)
);
```

```
CREATE TABLE BOOK ( … );

CREATE TABLE WRITTENBY ( … );

CREATE TABLE AUTHOR ( … );
```

4.4. Recomendaciones finales

- Para una transformación concreta utiliza la misma herramienta
  - Si utilizas una herramienta CASE para mapear asociaciones al código, utiliza la herramienta para cambiar las multiplicidades de la asociación
- Mantén los contratos en el código fuente, no en el modelo del diseño de objetos
  - Manteniendo la especificación como un comentario de código fuente, es más probable que se actualice ante cambios de código fuente
- Utiliza los mismos nombres para los mismos objetos
  - Si se cambia el nombre en el modelo, cambia el nombre en el código o en el esquema de base de datos
  - Proporciona trazabilidad entre modelos
- Utiliza una guía de estilo para las transformaciones
  - Haciendo explícitas las transformaciones en un manual, todos los desarrolladores pueden aplicar la transformación de la misma forma
- Añade la documentación del diseño de objetos en el código fuente y aprovecha herramientas de generación automática de documentación como JavaDoc