3. Programación Orientada a Objetos (POO) Avanzado

3.3. Clases abstractas e Interfaces

Programación Modular y Orientación a Objetos

Felipe Ibañez y Juan Miguel Lopez
felipe.anfurrutia@ehu.es
juanmiguel.lopez@ehu.es
Dpto. de Lenguajes y Sistemas Informáticos
UPV/EHU

Basado en las transparencias de Juan Pavón Mestras
Dpto. de Ingeniería e Inteligencia Artificial
Universidad Complutense de Madrid

Conceptos

- Clases abstractas
- Herencia múltiple
- Interfaces

Objetivo

- Evitar duplicación de código
- Reutilizar la implementación
- Facilitar el mantenimiento de la implementación
- Reducir el acoplamiento (dependencias) entre clases

Simulación por computador

- Programas que permiten simular actividades del mundo real
  - El tráfico
  - El tiempo
  - Procesos nucleares
  - Las fluctuaciones de la bolsa
  - Políticas de distribución de agua
  - Etc.
- Simplifican el mundo real
  - Compromiso entre
    - grado de detalle (mayor exactitud)
    - recursos requeridos (proceso, memoria y tiempo de simulación)

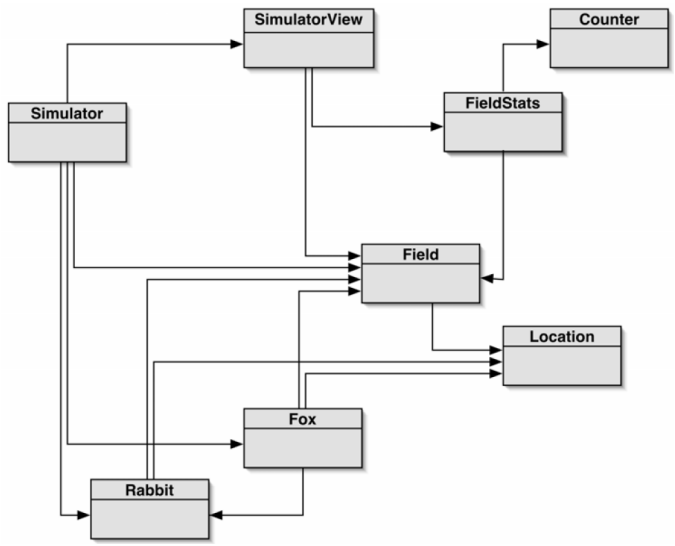
Simulación por computador

- Utilidad de la simulación
  - Ayuda a realizar predicciones
    - El tiempo
  - Facilita la experimentación
    - Más seguridad
    - Más barato
    - Más rápido
- Ejemplo
  - *¿Cómo le afectaría a la fauna del lugar si una autopista atravesara el parque nacional?*

Simulaciones predador-presa

- El balance entre las especies es delicado
  - Muchas presas implica mucha comida
  - Mucha comida anima a aumentar el número de predadores
  - Más predadores necesitan más presas
  - Menos presas significan menos comida
  - Menos comida significa…

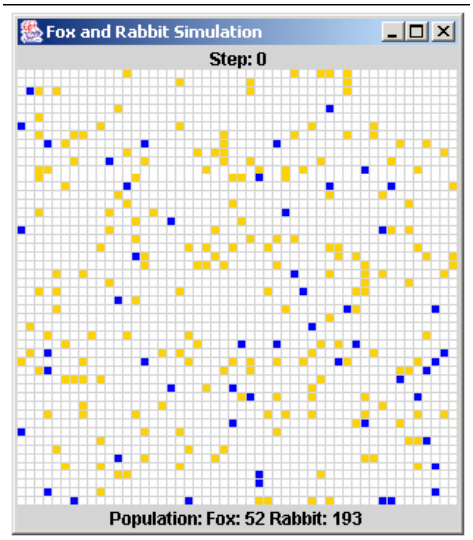
El proyecto de los zorros y los conejos



El proyecto de los zorros y los conejos (2)

- `Fox`
  - Modelo sencillo de un tipo de predador
- `Rabbit`
  - Modelo sencillo de un tipo de presa
- `Simulator`
  - Gestiona la tarea de simulación en general
  - Mantiene la colección de zorros y conejos
- `Field`
  - Representa un campo de dos dimensiones
- `Location`
  - Una posición en el campo de dos dimensiones
- `SimulatorView`, `FieldStats`, `Counter`
  - Mantienen estadísticas y presentan una visión del campo

Ejemplo de visualización



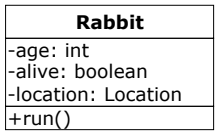
Estado de un conejo

Rabbit
-age: int -alive: boolean -location: Location

```
public class Rabbit {
    // Static fields omitted.

    // Individual characteristics (instance fields).
    // The rabbit's age.
    private int age;
    // Whether the rabbit is alive or not.
    private boolean alive;
    // The rabbit's position
    private Location location;

    //Methods omitted.
}
```

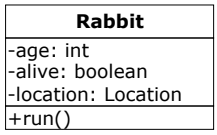
Comportamiento de un conejo

- Lo define el método `run()`
- A cada paso de la simulación (step) se incrementa la edad
  - En ese momento se mira si el conejo se muere
- A partir de cierta edad los conejos pueden dar a luz
  - Y así nacen nuevos conejitos

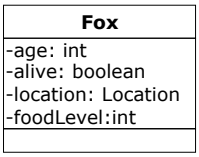


Simplificaciones sobre los conejos

- No se consideran géneros en los conejos
  - Todos son hembras: pueden tener más conejos
- El mismo conejo puede dar a luz en cada paso
- Todos los conejos mueren a la misma edad
- Otras?



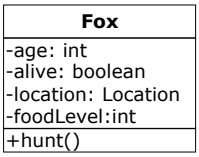
Estado de un zorro




```
public class Rabbit {
   // Static fields omitted.

   // Individual characteristics (instance fields).
   // The fox's age.
   private int age;
   // Whether the fox is alive or not.
   private boolean alive;
   // The fox's position
   private Location location;
   // The fox's food level, which is increased
   // by eating rabbits.
   private int foodLevel;

   //Methods omitted.
}
```

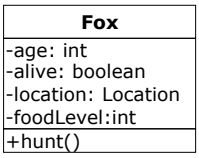
Comportamiento de un zorro

- Lo define el método `hunt()`
- Los zorros también crecen y se procrean
- Tienen hambre
- Y cazan para alimentarse en los lugares vecinos



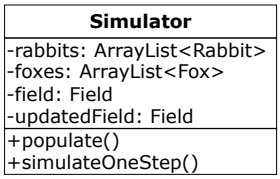
Configuración de los zorros

- Las mismas que para los conejos
- La caza y la comida se pueden modelar de varias maneras
  - ¿Debería ser el nivel de alimentación aditivo?
  - ¿Estará más o menos dispuesto a cazar un zorro si está hambriento?
- ¿Son siempre aceptables las simplificaciones?

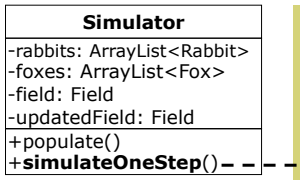


La clase Simulator

- Tres componentes clave:
  - Inicialización en el constructor
  - El método `populate()`
    - A cada animal se le asigna aleatoriamente una edad inicial
  - El método `simulateOneStep()`
    - Itera sobre las poblaciones de zorros y conejos
    - Utiliza dos objetos **Field** : **field** y **updatedField**



Actualización en cada paso




```
Iterator<Rabbit> it1 = rabbits.iterator();
while (it1.hasNext() ) {
    Rabbit rabbit = it1.next();
    rabbit.run(updatedField, newRabbits);
    if(! rabbit.isAlive()) {
        it1.remove();
    }
}
…
Iterator<Fox> it2 = foxes.iterator();
while (it2.hasNext() ) {
    Fox fox = it2.next();
    fox.hunt(field, updatedField, newFoxes);
    if(! fox.isAlive()) {
        it2.remove();
    }
}
```

Posibles mejoras

- El método `simulatedOneStep()` invoca código similar
- La clase `Simulator` está fuertemente acoplada a las clases específicas
  - Sabe demasiado sobre el comportamiento de los zorros y los conejos
- Las clases `Fox` y `Rabbit` tienen muchas similitudes pero no tienen una superclase en común

La superclase Animal

- Poner los atributos comunes en `Animal`:
  - `age`, `alive`, `location`
- Renombrar los métodos para encapsular la información:
  - `run()` y `hunt()` pasan a `act()`
- `Simulator` se puede desacoplar significativamente

```
Iterator<Animal> it = animals.iterator();
while (it.hasNext() ) {
   Animal animal= it.next();
   animal.act(field, updatedField, newAnimals);
   if(! animal.isAlive()) {
        it.remove();
   }
}
```

El método act de Animal

- La comprobación estática (compilación) requiere que haya un método `act()` en la clase `Animal`
- No hay una implementación compartida obvia
- Así que se define como abstracto:

```
abstract public void act(Field currentField,
                         Field updatedField,
                         List<Animal> newAnimals);
```

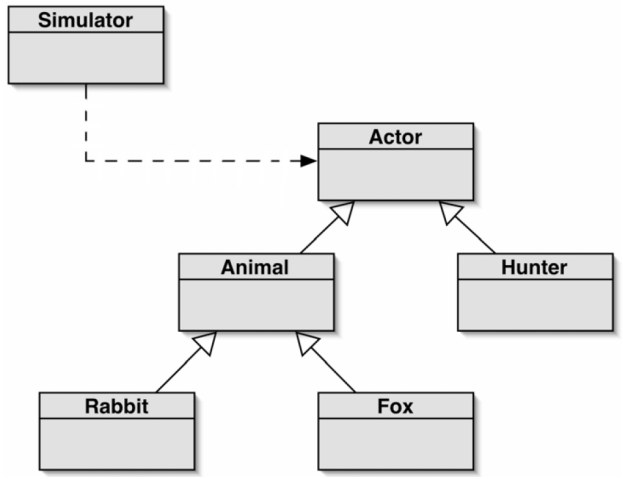
Métodos y clases abstractas

- Los métodos abstractos se indican con la palabra clave *abstract* al declararlos
- Los métodos abstractos no tienen cuerpo (implementación)
- Una clase con al menos un método abstracto es una clase abstracta
- Las clases abstractas no se pueden instanciar
  - Ej. ~~`Animal animal = new Animal();`~~ //ERROR
- La implementación de los métodos abstractos la realizarán las subclases concretas

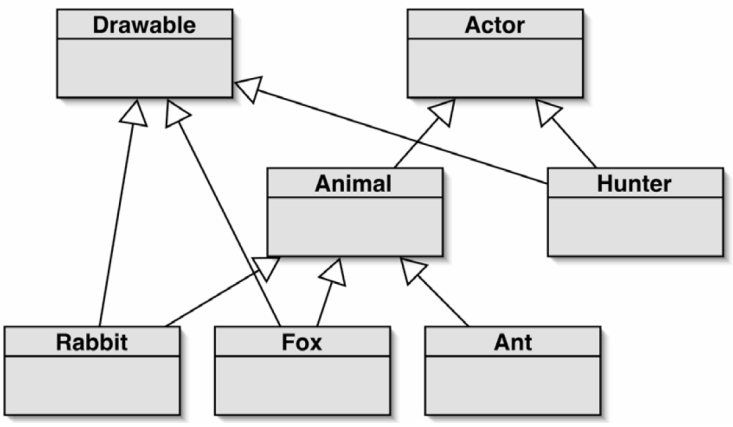
La clase Animal

```
public abstract class Animal{
   // fields omitted

   /**
    * Make this animal act - that is: make it do
    * whatever it wants/needs to do.
    */
   abstract public void act(Field currentField,
                            Field updatedField,
                            List<Animal> newAnimals);

   // other methods omitted
}
```

Mayor abstracción



Y si además los objetos se pueden dibujar



Herencia múltiple

- Cuando una clase hereda directamente de varias superclases
- Cada lenguaje tiene sus propias reglas
  - C++ lo permite
  - Java NO
- El problema surge en cómo resolver definiciones conflictivas
  - Por ejemplo métodos de dos superclases distintas con distinta implementación pero misma signatura
- ¿Cómo se puede resolver en Java cuando hace falta la herencia múltiple?
  - Con INTERFACES

La interfaz Actor

```
public interface Actor {
    /**
     * Perform the actor's daily behavior.
     * Transfer the actor to updatedField if it is
     * to participate in further steps of the simulation.
     * @param currentField The current state of the field.
     * @param updatedField The updated state of the field.
     * @param newActors New actors created as a result
     * of this actor's actions.
     */
    public void act(Field currentField, Field updatedField,
                    List<Actor> newActors);
}
```

Las clases implementan interfaces

```
public class Animal implements Actor {
...
}
```

```
public class Fox extends Animal implements Drawable {
...
}
```

```
public class Hunter implements Actor, Drawable {
...
}
```

Interfaces como tipos

- La implementación de clases no hereda código, pero…
- …las clases que implementan la interfaz son subtipos del tipo de interfaz
- Y se puede aplicar el polimorfismo con las interfaces de la misma manera que con las clases
  - Ejemplo:

```
...
Rabbit rabbit = new Rabbit();
Hunter hunter = new Hunter();

Actor actor = rabbit;
actor.act();
rabbit = (Rabbit) actor;
Drawable drawable = hunter;
drawable.draw();
if (drawable instanceof Hunter)
    hunter = (Hunter) drawable;
```

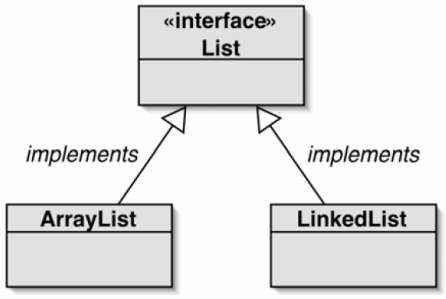
Interfaces

- No hay constructores
- Todos los métodos son abstractos
- Todos los métodos son públicos
- Todos los atributos son constantes

public static final

Interfaces (2)

- Permiten separar claramente la funcionalidad de la implementación
- Operaciones con sus parámetros y valores de retorno
- Los clientes interactúan independientemente de la imlementación
  - Pero pueden elegir implementaciones alternativas
  - Ejemplo:



Resumen

- La herencia permite compartir implementaciones
  - Clases abstractas y concretas
- La herencia permite compartir tipos
  - Clases e interfaces

- Los métodos abstractos permiten comprobación estática de tipos sin requerir una implementación
- Las clases abstractas son superclases incompletas
  - No se pueden instanciar
- Las clases abstractas soportan polimorfismo

- Las interfaces proporcionan una especificación sin implementación
  - Las interfaces son completamente abstractas
  - Las interfaces soportan polimorfismo
  - Las interfaces Java soportan ***herencia múltiple***