Práctica 2

Tecnología de la Programación I

Parte I: *Lemmings Refactored*

Refactorización

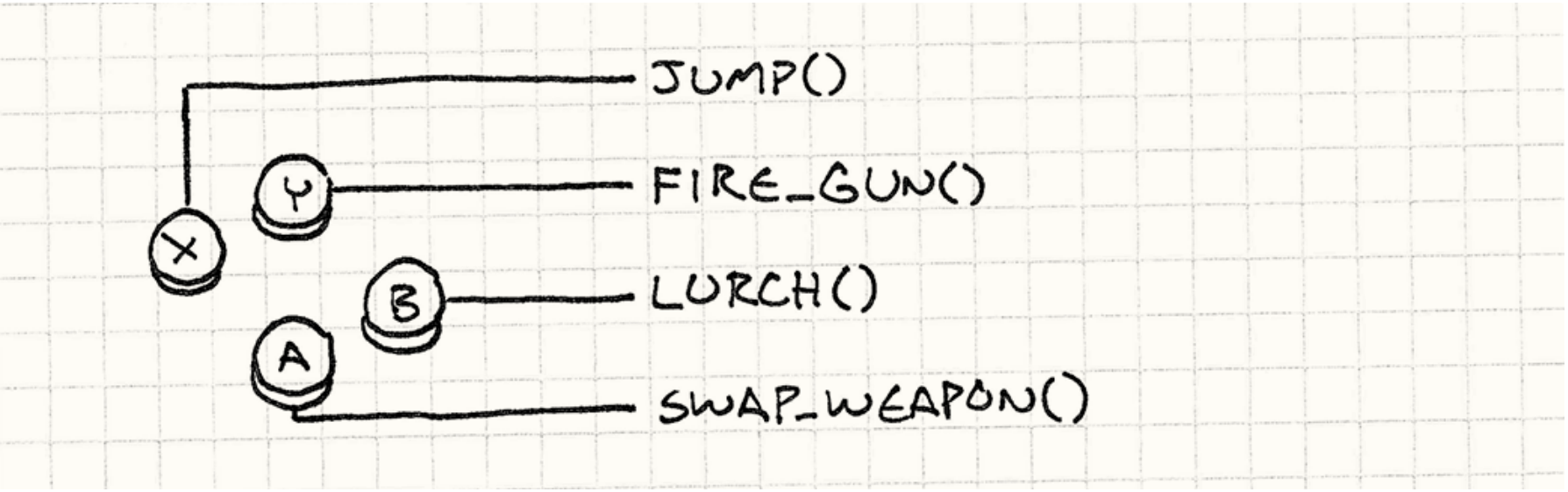
- Comandos del usuario
  - Modificamos el controlador distribuyendo su funcionalidad entre un conjunto de clases mejor estructuradas
    - Facilitará extensiones posteriores: añadir nuevos comandos
  - Versión simplificada del patrón de diseño *Command*
    - Idea general: **encapsular cada petición/acción del usuario en un objeto**
- Jerarquía de clases
  - Reorganización de los objetos del juego
    - Facilitará la extensión de la funcionalidad del juego
  - Utilización de clases abstractas e interfaces
- Contenedor único
  - Una sola estructura de datos para almacenar todos los objetos del juego

Beneficios:

- Facilidad para añadir nuevos comandos (código fácilmente ampliable)
- Bajo acoplamiento del código (separación de responsabilidades)
- Reutilización, escalabilidad y “testeabilidad”

El patrón de diseño *Command*

Leer entrada y llamar a la acción



El patrón de diseño *Command*

- La idea subyacente
  - "Cosificar" conceptos y convertirlos en datos (objetos)
  - Permite guardarlos en variables, pasarlos como parámetros, etc.
  - Viene a ser una llamada a un método empaquetada como objeto
    - Es decir, convertimos las peticiones en objetos
- Ejemplos de uso clásicos
  - Código que lee la entrada del usuario y la convierte en alguna acción
    - La orden del usuario se convierte en un objeto
    - Y este objeto es de una clase que implementa un método ejecutar()
    - Dependiendo del comando, su implementación será distinta
    - Entran en juego la herencia, el polimorfismo y la vinculación dinámica
  - Aplicaciones de escritorio basadas en ventanas
    - Con operaciones que, además, se pueden deshacer, y que pueden ser llamadas desde distintos contextos (menús contextuales, atajos, etc)

https://gameprogrammingpatterns.com/command.html

Bucle de juego de Controller

```
while (!game.isFinished()) {
  words = view.getPrompt();
  Command command = CommandGenerator.parse(words);

  if (command != null) {
    command.execute(game,view);
  else
    view.showErrorMessage(Messages.UNKNOWN_COMMAND);
}
```

leemos una acción de la consola

la “parseamos” para obtener el comando correspondiente (polimorfismo)

lo ejecutamos (vinculación dinámica)

Si no se puede parsear, mostramos un error

Idea clave: el controlador no sabe qué comando concreto se ejecutará, ni qué hará exactamente

Patrón de diseño de comportamiento

- Actores principales de nuestro patrón **Command**
  - `Controller` responsable de iniciar las solicitudes
    - No es responsable de crear el objeto
    - No sabe qué comando va a ejecutarse
    - Solo indica que se debe realizar una acción (ejecutar un comando)
  - `CommandGenerator`: crea los objetos-comandos concretos
  - *`Command`*: encapsula la información necesaria para ejecutar una acción y declara al menos un método abstracto para ejecutar un comando

    `execute(Game,GameView)`
  - **Comandos concretos**: implementan la acción trasmitiéndola a la lógica
    - Cada clase `XxxCommand` ejecuta una y solo una acción
  - `Game`: la clase que implementa la lógica y hace el trabajo real
  - `GameView`: la vista puede recibir peticiones del comando para mostrar cosas o pedir datos al usuario

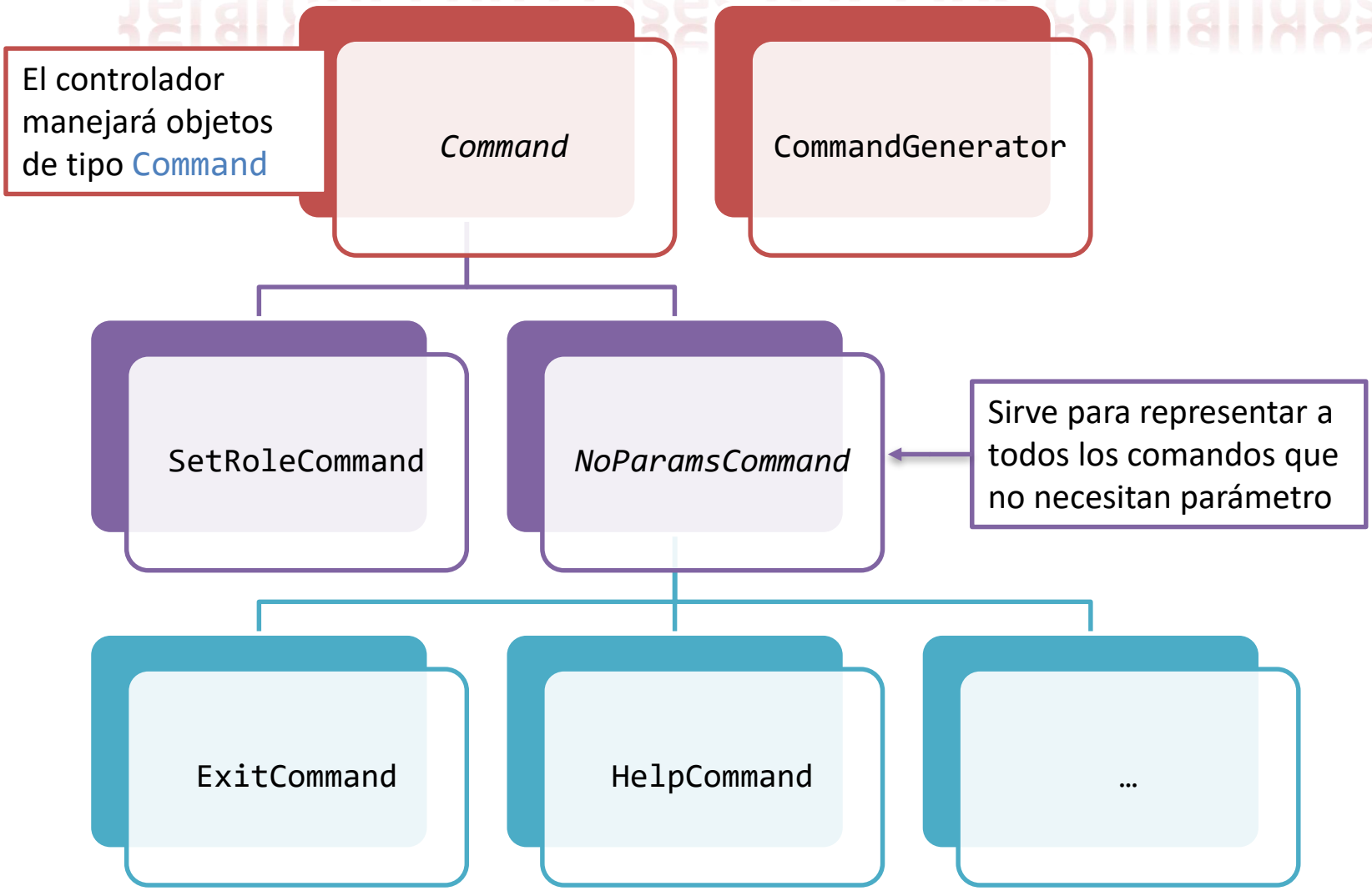
Implementando el patrón *Command*

- `Command`: clase abstracta que encapsula la funcionalidad común de todos los comandos concretos
  - Tres métodos básicos
    - `protected boolean matchCommand(String)` comprueba si una acción introducida por teclado se corresponde con la del comando
    - `public abstract Command parse(String[])` crea una instancia del comando, dados los parámetros proporcionados por el usuario
    - `public abstract void execute(Game,GameView)` ejecuta la acción

- Clase `NoParamsCommand`
  - Clase abstracta que hereda de `Command`
  - Sirve de base para todos los comandos que no tienen parámetros

- Clases para los comandos concretos

Jerarquía de clases para los comandos



Clase CommandGenerator

```
public class CommandGenerator {

 private static final List<Command> AVAILABLE_COMMANDS =
    Arrays.asList( new ExitCommand(), new HelpCommand(),
                   // …
    );
  public static Command parse(String[] commandWords) {
    for (Command c: availableCommands) {
      // TODO
    }
    return null;
  }

  public static String commandHelp() { // … }
}
```

Lista con los comandos concretos disponibles

`parse` encuentra el comando concreto asociado a la entrada del usuario, recorriendo la lista de comandos para determinar con cuál se corresponde la entrada del usuario, llamando al método `parse(String[])` de cada comando

Métodos clave de los comandos

- `execute` realiza la acción sobre *game* y *view*

- `parse(String[])` devuelve una instancia del comando concreto
  - Como cada comando procesa sus propios parámetros, este método devolverá `this` o creará una nueva instancia de la misma clase
    - De momento todos nuestros comandos son sin parámetros, de modo que pueden devolver `this`
  - En el caso de que los argumentos introducidos por el usuario no sean válidos para el comando, el método `parse(String[])` devolverá `null`

Clase abstracta Command

```
public abstract class Command {

  private final String name;
  protected String getName() {return name;}
  // Idem para shortcut, details y help


  public Command(String name,……) {……}

  public abstract void execute(Game game, GameView view);
  public abstract Command parse(String[] commandWords);

  protected boolean matchCommandName(String name) {
    return getShortcut().equalsIgnoreCase(name)
           || getName().equalsIgnoreCase(name);
  }

  public String helpText() {
    return getDetails() + " : " + getHelp() + "\n"; }
}
```

Clase abstracta NoParamsCommand

```
public abstract class NoParamsCommand extends Command {

 public NoParamsCommand (String name, String shortcut, ……){
    super(name, shortcut, ……);
 }

  @Override
  public Command parse(String[] commandWords) {
    // TODO
    return null;
  }
}
```

Todos los comandos sin parámetros se “parsean” igual

se comprueba que el usuario solamente ha introducido una palabra, y que esta coincide con el nombre o la abreviatura del comando

Clase concreta ExitCommand

```
public class ExitCommand extends NoParamsCommand {

 private static final NAME = Messages.COMMAND_EXIT_NAME;
  ……
 public ExitCommand (){
    super(NAME, SHORTCUT, ……);
 }

 @Override
 public void execute(Game game, GameView view) {
    game.exit();   // no hay que repintar
  }
}
```

Nueva clase `Messages` a sustituir por la que tenéis

Os damos bastante código

- Partid de una copia de vuestro proyecto
  - No uséis la “plantilla” como esqueleto con el que empezar
- Cuando proceda, id incorporando el código que os damos
  - Es una guía, más que una “plantilla”
- De esta parte, os damos
  - Nuevas clases completas
    - `Command, ExitCommand, HelpCommand`
  - Nuevas clases casi completas
    - `CommandGenerator, NoParamsCommand`
  - Clases a reemplazar
    - `Controller, Main, Messages` y todas las del paquete `view`

Beneficios:

- Facilidad para añadir nuevos objetos del juego
- Centralizar datos y métodos, eliminando código duplicado

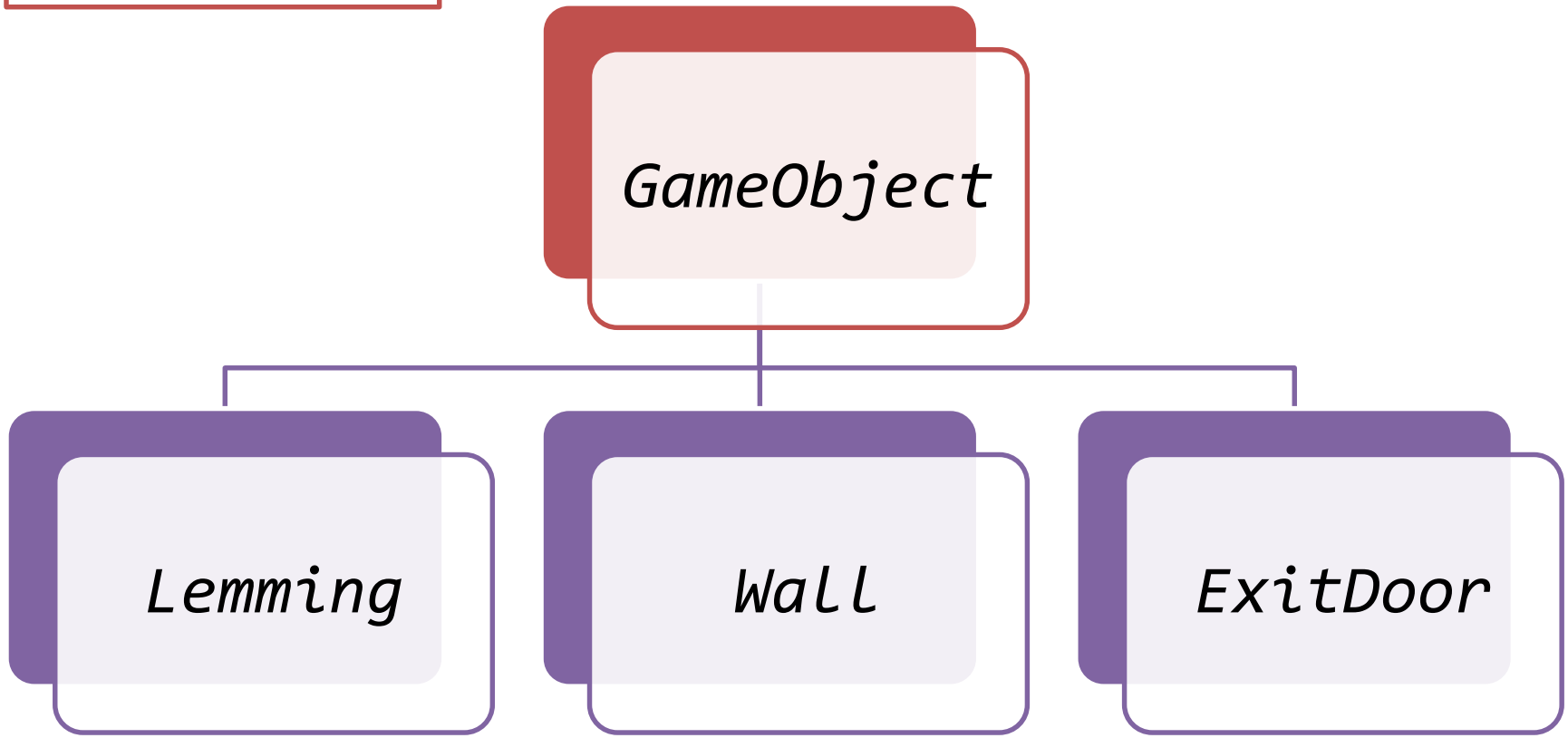
Herencia y polimorfismo

Nueva clase `GameObject`

- Clase abstracta `GameObject`
  - Todo lo común
    - Atributos: `game, pos, isAlive`
    - Incluye implementaciones de métodos con implementación coincidente en todos los objetos del juego
      - `isInPosition(Position), isAlive(), …`
    - Declara métodos con implementación no coincidente
      - `getIcon(), update(), isSolid()`, …
    - Incluso puede implementar alguno con implementación mayoritaria similar con la idea de redefinir solo cuando proceda
    - Puede ocurrir que la implementación de algunos de estos métodos sea vacía o con funcionalidad trivial
      - `Wall` no hace nada al actualizarse, por lo que la implementación de `update()` será vacía

Jerarquía de los objetos del juego

`Game` manejará objetos de tipo `GameObject`



Beneficios:

- Manejar los `GameObject` como una abstracción permite utilizar una única estructura de datos para almacenarlos, en lugar de tres listas

Contenedor de objetos del juego

Clase única contenedora de objetos

```
public class GameObjectContainer {
  private List<GameObject> objects;
  public GameObjectContainer() { objects = new ArrayList<>(); }

  public void add(GameObject object) { objects.add(object); }

  public void update() {
    for (int i = 0; i < objects.size(); i++) {
      GameObject object = objects.get(i);
      object.update();
    }
    removeDead();
  }

}
```

Usamos una colección genérica de Java

Polimorfismo

Vinculación dinámica

No usamos un for-each para evitar una excepción en caso de que la lista se modifique mientras se recorre

Beneficios:

- Evitar usos imprevistos de métodos públicos de Game

Interfaces de Game

Problema

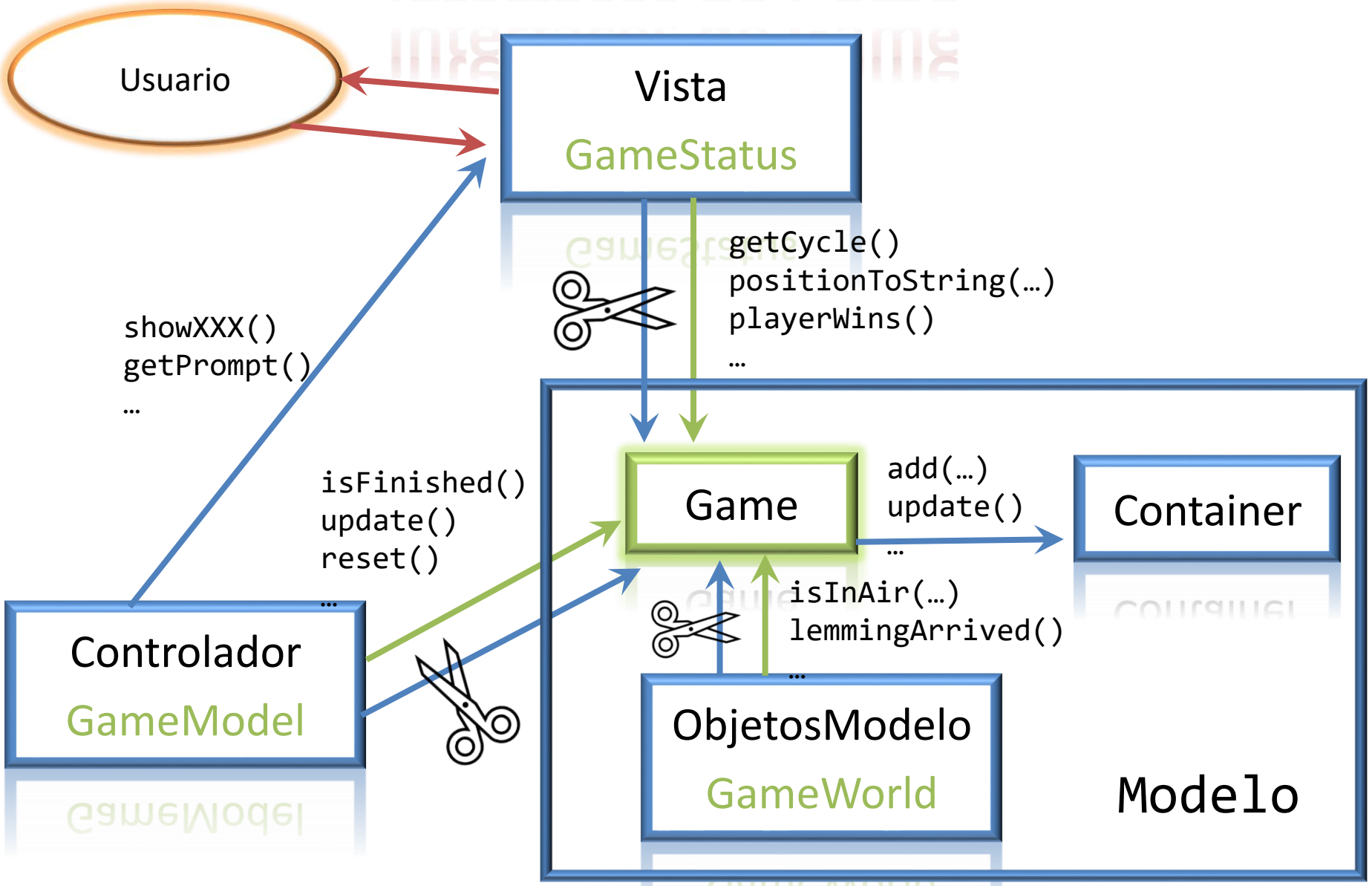
- Usamos los métodos de `Game` desde tres contextos diferentes
  - `Controller`: para enviar al juego las instrucciones del usuario (ahora a través del método `execute()` de los comandos)
    - Ejemplos: `reset()`, `exit()` o `update()`
  - `GameView`: para solicitar los datos que se deben mostrar
    - Ejemplos: `getCycle()`, `getRemainingLemmings()` o `positionToString(int, int)`
  - Objetos del juego: para que los objetos interactúen con su entorno
    - Ejemplos: `isInAir(Position)` o `lemmingDead()`
    - A este tipo de métodos se les suele llamar ***callbacks*** (acciones de retorno)
- Pero nada prohíbe que se usen de distinta manera
  - Por ejemplo, que un objeto del juego invoque a `reset` o que la vista invoque a `lemmingDead()`
    - Son métodos públicos de `Game` y no hay restricción en su utilización

Solución

- El anterior problema puede resolverse utilizando **interfaces**
  - Crearemos una interfaz distinta para cada uno de esos contextos
  - Cada interfaz proporcionará una **vista parcial** de `Game` con solo algunos de sus métodos

- Definiremos
  - `GameModel` para la visión del juego desde el controlador
    - Declarando los métodos que se pueden usar desde ahí
  - `GameStatus` para la visión del juego desde la vista *[os la damos completa]*
  - `GameWorld` para la visión del juego desde los objetos del juego
  - Una vez definidas las interfaces
    - Haremos que `Game` las implemente
    - Y en los contextos en los que se usaba `Game`, reemplazamos el tipo de referencia por la interfaz correspondiente

Interfaces de Game



Idea global

```
public interface GameModel { // Definir las interfaces

  public boolean isFinished();

  // PLAYER ACTIONS
  public boolean update();
  public void reset();
  // …
}
// Declarar que Game implementa las interfaces
public class Game implements GameModel, GameStatus, GameWorld {
  // …
}
// Y sustituir Game donde proceda por la interfaz adecuada
public abstract void execute(GameModel game, GameView view);
```

Os damos

```
public interface GameStatus {

  public String positionToString(int x, int y);
  public int getCycle();
  public int getRemainingLemmings();
  public boolean playerWin();
  public boolean playerLoses();
}

public class Game implements GameStatus { //… }

public abstract class GamePrinter {
  protected GameStatus game;
  public GamePrinter(GameStatus game) {
    this.game = game;
  } //…
```