Laboratorio de Java

Estructura de la Información / Diseño de Estructuras de Datos

Anexo: ejemplos de interrelaciones de TADs

Índice de contenidos

1. Introducción

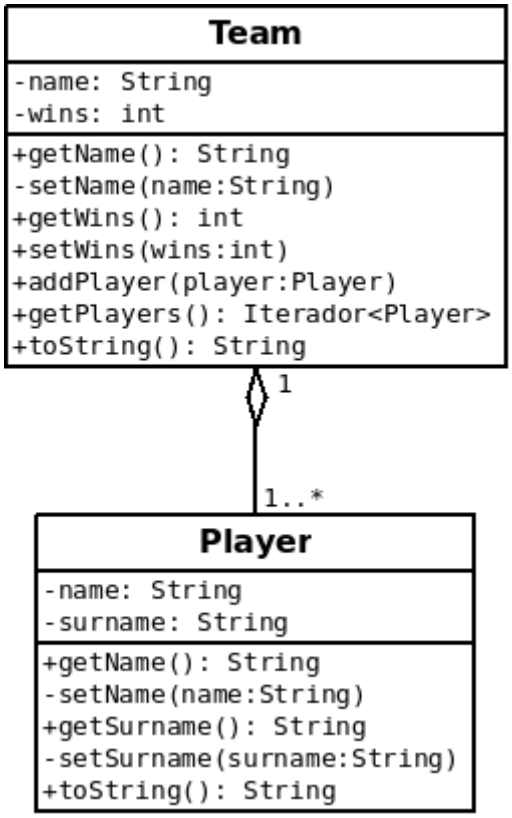
El actual documento tiene como objetivo mostrar una serie de ejemplos de interrelaciones de TADs. No se pretende entrar en profundidad en la toma de decisiones previa a cualquier elección de TADs, sino que en base a un escenario concreto, se quiere exponer de una forma práctica algunas relaciones que se pueden establecer entre TADs, usando la biblioteca de TADs como elemento de soporte para construir las nuevas estructuras.

2. Primer escenario

Se desea gestionar un conjunto de equipos de baloncesto, donde cada equipo está formado por un conjunto de jugadores.

2.1. Relación entre clases

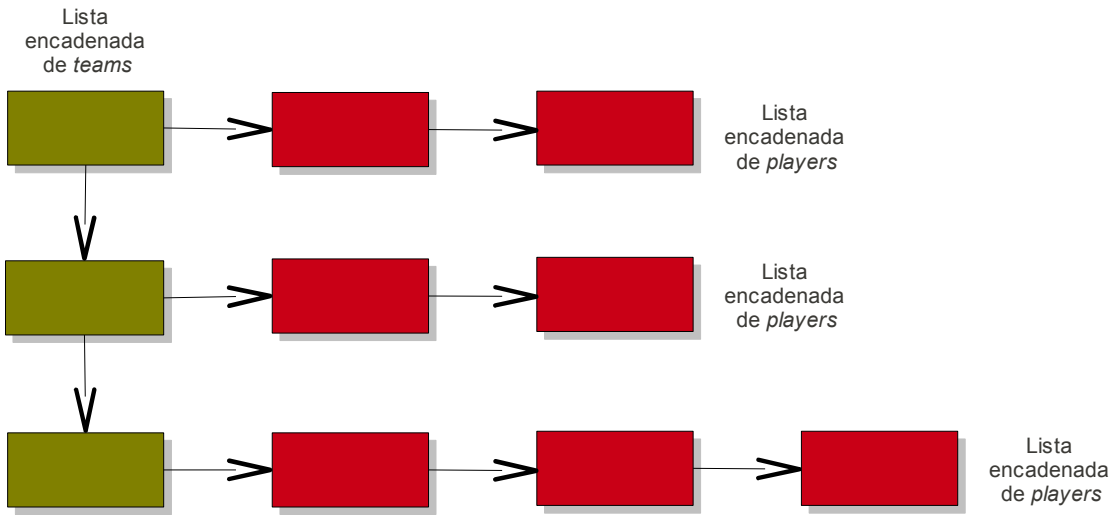
Se dispone de una clase `Team` para implementar un equipo de baloncesto, y una clase `Player` para instanciar los jugadores que forman el equipo.



Dada la relación de composición entre las clases, `Team` contendrá como atributo un TAD que a su vez contendrá un conjunto de tipo `Player`.

2.2. TADs

El primer escenario plantea una *lista encadenada* de equipos (objetos de tipo `Team`), de los cuales cada uno está compuesto por una *lista encadenada* de jugadores (objetos de tipo `Player`).



2.3. Implementación

Los pasos a seguir para implementar este escenario son:

1. Crear la clase `Player`, según el diagrama UML anterior:

```
public class Player {
	private String name;
	private String surname;

	public Player(String name, String surname) {
		setName(name);
		setSurname(surname);
	}

	public String getName() {
		return name;
	}

	private void setName(String name) {
		this.name = name;
	}

	public String getSurname() {
		return surname;
	}
```

```
	private void setSurname(String surname) {
		this.surname = surname;
	}

	public String toString() {
		return getSurname()+", "+getName();
	}
}
```

2. Crear la clase `Team`, según el diagrama UML anterior:

```
import uoc.ei.tads.Iterador;
import uoc.ei.tads.ListaEncadenada;

public class Team {

	private String name;
	private int wins;
	private ListaEncadenada<Player> players=new ListaEncadenada<Player>();

	public Team(String name) {
		setName(name);
	}

	public String getName() {
		return name;
	}

	private void setName(String name) {
		this.name = name;
	}

	public int getWins() {
		return wins;
	}

	public void setWins(int wins) {
		this.wins = wins;
	}

	public void addPlayer(Player player) {
		players.insertarAlFinal(player);
	}

	public Iterador<Player> getPlayers(){
		return players.elementos();
	}

	public String toString() {
		return getName()+", ["+getWins()+"]";
	}
}
```

Com se puede observar, `Team` contiene un atributo interno de tipo `ListaEncadenada`, en el que se guardaran los objetos de tipo `Player` que contiene el equipo:

```
private ListaEncadenada<Player> players=new ListaEncadenada<Player>();
```

Las referencias `<Player>` que se indican en `ListaEncadenada` definen qué tipo de objetos contendrá el TAD `ListaEncadenada`.

Para insertar una instancia `Player` dentro de la `ListaEncadenada` de `Team`, se usa el método `addPlayer(Player p)`, que a su vez realiza una llamada al método `insertarAlFinal(Player)` propio de `ListaEncadenada`.

3. Crear la clase ejecutable, que permitirá la validación del escenario planteado:

```
import uoc.ei.tads.Iterador;
import uoc.ei.tads.ListaEncadenada;

public class Example1 {

	public static void main(String args[]){

	/* Inicio inicialización */

		/* Se crean los equipos */
		Team chi=new Team("Chicago Bulls");
		Team ind=new Team("Indiana Pacers");
		Team mil=new Team("Milwaukee Bucks");
		Team cle=new Team("Cleveland Cavaliers");
		Team det=new Team("Detroit Pistons");

		/* Se asignan los equipos a la Central Division */
		ListaEncadenada<Team> centralDivision = new ListaEncadenada<Team>();
		centralDivision.insertarAlFinal(chi);
		centralDivision.insertarAlFinal(mil);
		centralDivision.insertarAlFinal(cle);
		centralDivision.insertarAlFinal(ind);
		centralDivision.insertarAlFinal(det);

		/* Se crean los jugadores */
		Player chi05=new Player("Carlos", "Boozer");
		Player chi01=new Player("Derrick", "Rose");
		Player chi13=new Player("Joakim", "Noah");
		Player chi09=new Player("Luol", "Deng");
		Player chi03=new Player("Omek", "Asik");
		Player cle02=new Player("Kyrie", "Irving");
		Player cle17=new Player("Anderson", "Varejao");
		Player cle04=new Player("Antawn", "Jamison");
		Player cle01=new Player("Daniel", "Gibson");
		Player cle33=new Player("Alonzo", "Gee");
		Player ind33=new Player("Danny", "Granger");
		Player ind02=new Player("Darren", "Collison");
		Player ind21=new Player("David", "West");
		Player ind17=new Player("Lou", "Amundson");
		Player ind55=new Player("Roy", "Hibbert");
```

```
		Player mil06=new Player("Andrew", "Bogut");
		Player mil00=new Player("Drew", "Gooden");
		Player mil07=new Player("Ersan", "Ilyasova");
		Player mil03=new Player("Brandon", "Jennings");
		Player mil05=new Player("Stephen", "Jackson");
		Player det08=new Player("Ben", "Gordon");
		Player det22=new Player("Tayshaun", "Prince");
		Player det07=new Player("Brandon", "Knight");
		Player det10=new Player("Greg", "Monroe");
		Player det03=new Player("Rodney", "Stuckey");

		/* Se asignan los jugadores a los correspondientes equipos */
		chi.addPlayer(chi05);
		chi.addPlayer(chi01);
		chi.addPlayer(chi13);
		chi.addPlayer(chi09);
		chi.addPlayer(chi03);
		cle.addPlayer(cle02);
		cle.addPlayer(cle17);
		cle.addPlayer(cle04);
		cle.addPlayer(cle01);
		cle.addPlayer(cle33);
		ind.addPlayer(ind33);
		ind.addPlayer(ind02);
		ind.addPlayer(ind21);
		ind.addPlayer(ind17);
		ind.addPlayer(ind55);
		mil.addPlayer(mil06);
		mil.addPlayer(mil00);
		mil.addPlayer(mil07);
		mil.addPlayer(mil03);
		mil.addPlayer(mil05);
		det.addPlayer(det08);
		det.addPlayer(det22);
		det.addPlayer(det07);
		det.addPlayer(det10);
		det.addPlayer(det03);

		/* Se inicializan los partidos ganados por los equipos */
		chi.setWins(33);
		mil.setWins(15);
		cle.setWins(14);
		ind.setWins(23);
		det.setWins(13);

	/* Fin inicialización */
	/* Inicio recorridos */

		/* Se muestra por pantalla los equipos de la Central Division */
		System.out.println("\nEquipos de la División Central:\n");
		Iterador<Team> teams=centralDivision.elementos();
		while (teams.haySiguiente()) {
			Team teamAux=teams.siguiente();
			System.out.println(teamAux.toString());
		}
```

```
		/* Se muestra por pantalla los jugadores definidos en la Central Division */
		System.out.println("\nJugadores de la División Central:\n");
		teams=centralDivision.elementos();
		while (teams.haySiguiente()) {
			Team teamAux=teams.siguiente();
			Iterador<Player> players=teamAux.getPlayers();
			while (players.haySiguiente()) {
				Player playerAux=players.siguiente();
				System.out.println(playerAux.toString());
			}
		}

		/* Se muestra por pantalla los jugadores definidos de la Central Division,
		 * agrupados por equipo */
		System.out.println("\nJugadores por equipo de la División Central:\n");
		teams=centralDivision.elementos();
		while (teams.haySiguiente()) {
			Team teamAux=teams.siguiente();
			System.out.println("\n"+teamAux.getName());
			System.out.println("------------------------");
			Iterador<Player> players=teamAux.getPlayers();
			while (players.haySiguiente()) {
				Player playerAux=players.siguiente();
				System.out.println("\t"+playerAux.toString());
			}
		}
	/* Fin recorridos */
	}
}
```

Si nos fijamos en el código fuente, el conjunto de `Team` quedan incluidos dentro de `ListaEncadenada<Team>`, de forma que este TAD solamente podrá contener objetos de tipo `Team`.

La salida que genera el primer bucle es:

```
Equipos de la División Central:

Chicago Bulls, [33]
Milwaukee Bucks, [15]
Cleveland Cavaliers, [14]
Indiana Pacers, [23]
Detroit Pistons, [13]
```

La salida del segundo bucle es:

```
Jugadores de la División Central:

Boozer, Carlos
Rose, Derrick
Noah, Joakim
Deng, Luol
Asik, Omek
Bogut, Andrew
Gooden, Drew
Ilyasova, Ersan
Jennings, Brandon
```

```
Jackson, Stephen
Irving, Kyrie
Varejao, Anderson
Jamison, Antawn
Gibson, Daniel
Gee, Alonzo
Granger, Danny
Collison, Darren
West, David
Amundson, Lou
Hibbert, Roy
Gordon, Ben
Prince, Tayshaun
Knight, Brandon
Monroe, Greg
Stuckey, Rodney
```

La del tercer bucle:

```
Jugadores por equipo de la División Central:

Chicago Bulls
------------------------
       Boozer, Carlos
       Rose, Derrick
       Noah, Joakim
       Deng, Luol
       Asik, Omek

Milwaukee Bucks
------------------------
       Bogut, Andrew
       Gooden, Drew
       Ilyasova, Ersan
       Jennings, Brandon
       Jackson, Stephen

Cleveland Cavaliers
------------------------
       Irving, Kyrie
       Varejao, Anderson
       Jamison, Antawn
       Gibson, Daniel
       Gee, Alonzo

Indiana Pacers
------------------------
       Granger, Danny
       Collison, Darren
       West, David
       Amundson, Lou
       Hibbert, Roy

Detroit Pistons
------------------------
       Gordon, Ben
       Prince, Tayshaun
       Knight, Brandon
       Monroe, Greg
       Stuckey, Rodney
```

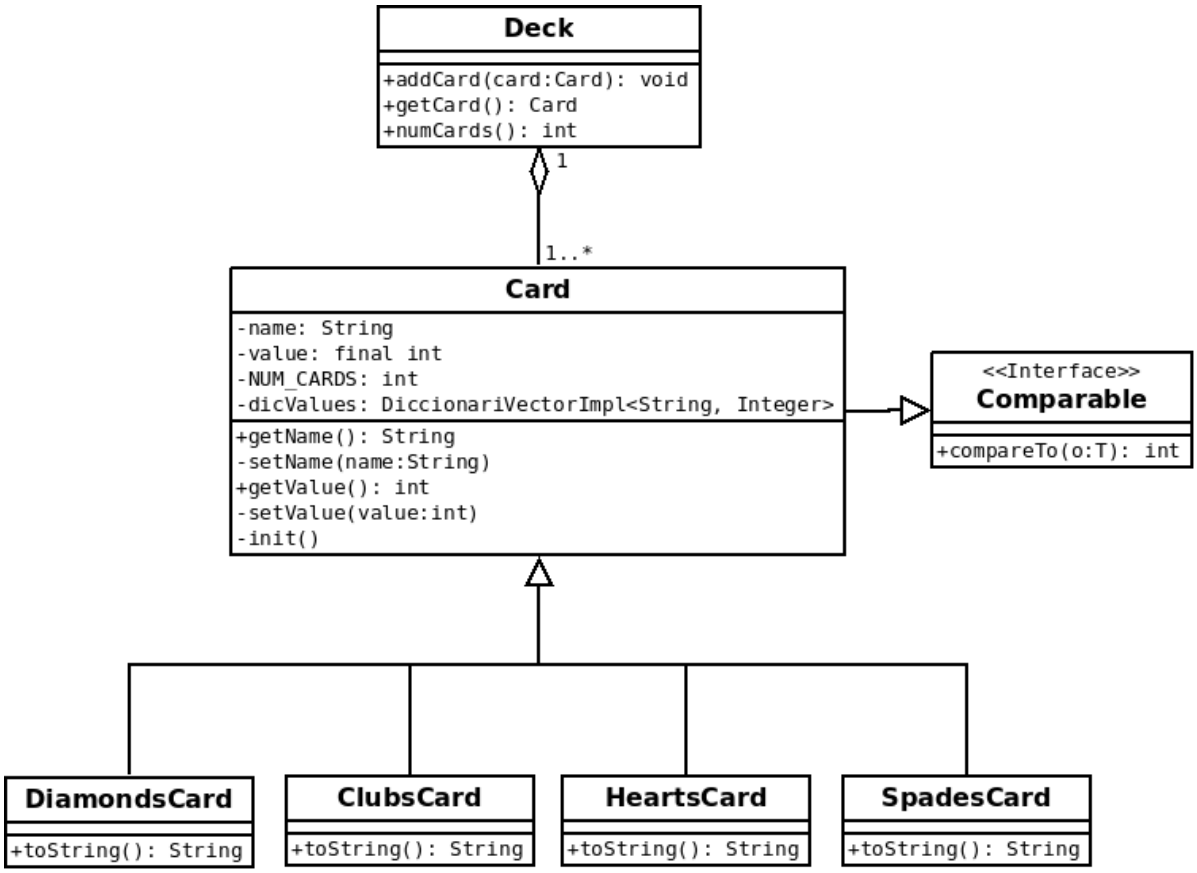
3. Segundo escenario

Se quiere simular las pilas de cartas que se generan cuando se está ordenando una baraja manualmente, cogiendo una carta cada vez para ponerla en la pila que le corresponde según su palo.

3.1. Relación entre clases

Se dispone de una clase `Card`, de la que heredan las clases `DiamondsCard`, `ClubsCard`, `HeartsCard` y `SpadesCard`. Todas las cartas de añadirán a la clase `Deck`, de póquer. La relación de composición entre clases implica que `Deck` tiene un atributo TAD con el conjunto de las `Card`.

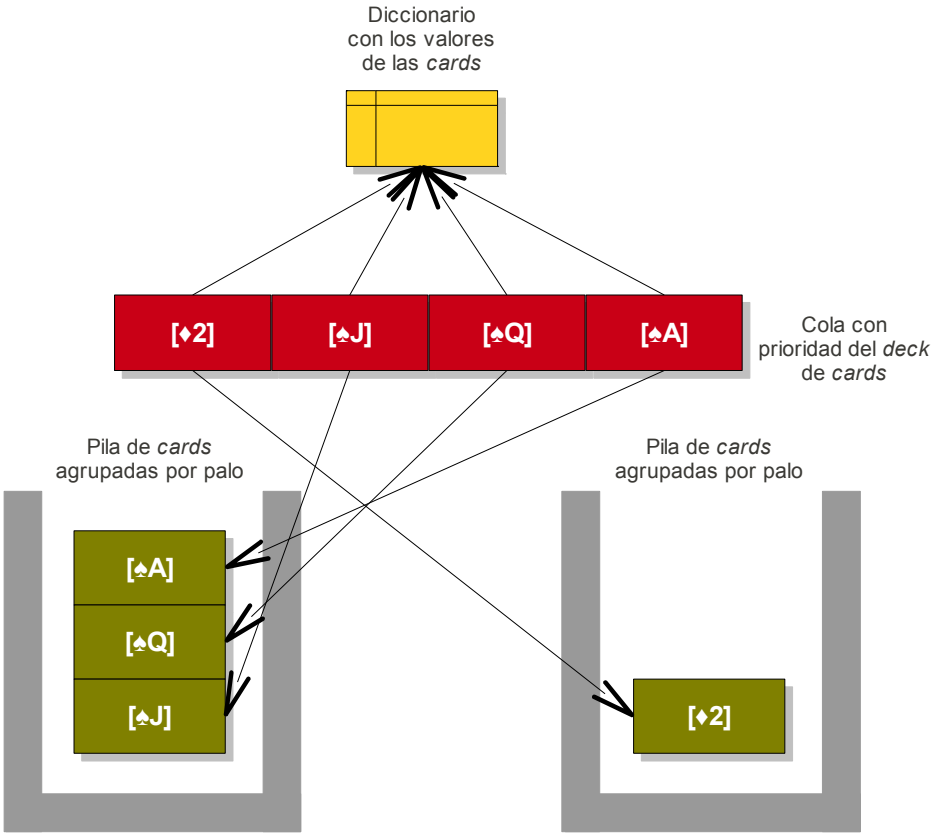
Además, `Card` implementa la *interficie* `Comparable`: el objetivo es permitir comparar dos instancias `Card` y así poder establecer una ordenación determinada entre ellas.



3.2. TADs

El segundo escenario utiliza el TAD *cola con prioridad* para contener las `Card`. La prioridad de la cola nos permitirá tener las cartas ordenadas ascendentemente según su valor. A continuación, se utilizarán cuatro pilas para agrupar las cartas según su palo; encolar y desencolar las cartas de las pilas provocará que las obtengamos en orden inverso.

Para poder establecer la relación entre el nombre de una `Card` y su valor, se ha usado un TAD de tipo *diccionario*.



3.3. Implementación

Los pasos a seguir para implementar el segundo escenario son:

1. Crear la clase `Card`, según el diagrama UML anterior:

```
import uoc.ei.tads.DiccionarioVectorImpl;

public class Card implements Comparable<Card> {

	private static final int NUM_CARDS=13;
	private int value;
	private String name;
	private static DiccionarioVectorImpl<String, Integer> dicValues = new
		DiccionarioVectorImpl<String, Integer>(NUM_CARDS);

	private void init() {
		dicValues.insertar(new String("2"), new Integer(1));
		dicValues.insertar(new String("3"), new Integer(2));
		dicValues.insertar(new String("4"), new Integer(3));
		dicValues.insertar(new String("5"), new Integer(4));
		dicValues.insertar(new String("6"), new Integer(5));
		dicValues.insertar(new String("7"), new Integer(6));
		dicValues.insertar(new String("8"), new Integer(7));
		dicValues.insertar(new String("9"), new Integer(8));
		dicValues.insertar(new String("10"), new Integer(9));
		dicValues.insertar(new String("J"), new Integer(10));
		dicValues.insertar(new String("Q"), new Integer(11));
		dicValues.insertar(new String("K"), new Integer(12));
		dicValues.insertar(new String("A"), new Integer(13));
	}

	public Card(String nameCard) {
		init();
		setName(nameCard);
		setValue(dicValues.consultar(nameCard));
	}

	public int getValue() {
		return value;
	}

	private void setValue(int value) {
		this.value = value;
	}

	public String getName() {
		return name;
	}

	private void setName(String name) {
		this.name = name;
	}

	public int compareTo(Card card) {
```

```
                return this.getValue()-card.getValue();
        }
}
```

2. Crear la clase hija `ClubsCard`:

```
public class ClubsCard extends Card {

        public ClubsCard(String nameCard){
                super(nameCard);
        }

        public String toString(){
                return "[♣"+this.getName()+"]";
        }
}
```

3. Crear la clase hija `DiamondsCard`:

```
public class DiamondsCard extends Card {

        public DiamondsCard(String nameCard){
                super(nameCard);
        }

        public String toString(){
                return "[♦"+this.getName()+"]";
        }
}
```

4. Crear la clase hija `SpadesCard`:

```
public class SpadesCard extends Card {

        public SpadesCard(String nameCard){
                super(nameCard);
        }

        public String toString(){
                return "[♠"+this.getName()+"]";
        }
}
```

5. Crear la clase hija `HeartsCard`:

```
public class HeartsCard extends Card {

        public HeartsCard(String nameCard){
```

```
		super(nameCard);
	}

	public String toString(){
		return "[♥"+this.getName()+"]";
	}
}
```

6. Crear la clase `Deck`, según el diagrama UML anterior:

```
import uoc.ei.tads.Cola;
import uoc.ei.tads.ColaConPrioridad;

public class Deck {

	private Cola<Card> cards=new ColaConPrioridad<Card>();

	public void addCard(Card card) {
		cards.encolar(card);
	}

	public Card getCard() {
		return cards.desencolar();
	}

	public int numCards() {
		return cards.numElems();
	}
}
```

En este caso, `Deck` contiene un atributo interno de tipo `ColaConPrioridad`, en el que se guardarán las `Card` de la baraja:

```
private Cola<Card> cards=new ColaConPrioridad<Card>();
```

7. Crear la clase ejecutable, que permitirá la validación del escenario planteado:

```
import uoc.ei.tads.Pila;
import uoc.ei.tads.PilaVectorImpl;

public class Example2 {

	public static void main(String args[]){

	/* Inicio inicialización */

		/* Se crea la baraja de cartas de póquer */
		Deck deck = new Deck();

		/* Se crean las cartas */
		Card h2=new HeartsCard("2");
```

```
		Card h3=new HeartsCard("3");
		Card hJ=new HeartsCard("J");
		Card c4=new ClubsCard("4");
		Card c6=new ClubsCard("6");
		Card sJ=new SpadesCard("J");
		Card sQ=new SpadesCard("Q");
		Card sK=new SpadesCard("K");
		Card d2=new DiamondsCard("2");
		Card dK=new DiamondsCard("K");
		Card dA=new DiamondsCard("A");

		/* Se añaden las cartas a la baraja */
		deck.addCard(h2);
		deck.addCard(sJ);
		deck.addCard(d2);
		deck.addCard(dK);
		deck.addCard(sK);
		deck.addCard(sQ);
		deck.addCard(h3);
		deck.addCard(c6);
		deck.addCard(hJ);
		deck.addCard(c4);
		deck.addCard(dA);

		/* Se crean cuatro pilas de cartas, una para cada uno de los palos
		 * de la baraja */
		Pila<ClubsCard> clubsCards=new PilaVectorImpl<ClubsCard>();
		Pila<DiamondsCard> diamondsCards=new PilaVectorImpl<DiamondsCard>();
		Pila<SpadesCard> spadesCards=new PilaVectorImpl<SpadesCard>();
		Pila<HeartsCard> heartsCards=new PilaVectorImpl<HeartsCard>();

	/* Fin inicialización */
	/* Inicio recorridos */

		/* Se muestra por pantalla cada una de las cartas de la baraja,
		 * de forma ordenada. Además, en función de la naturaleza de cada
		 * carta, se añade a la pila correspondiente del mismo palo
		 */
		System.out.print("Total cartas :\n\t");
		while (deck.numCards()>0) {
			Card aux=deck.getCard();
			System.out.print(aux.toString()+" ");
			if (aux instanceof ClubsCard) {
				clubsCards.apilar((ClubsCard)aux);
			} else if (aux instanceof DiamondsCard) {
				diamondsCards.apilar((DiamondsCard)aux);
			} else if (aux instanceof SpadesCard) {
				spadesCards.apilar((SpadesCard)aux);
			} else if (aux instanceof HeartsCard) {
				heartsCards.apilar((HeartsCard)aux);
			}
		}

		/* Se desapilan las cartas diamond y se muestran por pantalla */
		System.out.print("\n\nDiamons desapiladas :\n\t");
		while (!diamondsCards.estaVacio()) {
```

```
                System.out.print(diamondsCards.desapilar()+" ");
            }

            /* Se desapilan las cartas spades y se muestran por pantalla */
            System.out.print("\n\nSpades desapiladas :\n\t");
            while (!spadesCards.estaVacio()) {
                System.out.print(spadesCards.desapilar()+" ");
            }

            /* Se desapilan las cartas hearts y se muestran por pantalla */
            System.out.print("\n\nHearts desapiladas :\n\t");
            while (!heartsCards.estaVacio()) {
                System.out.print(heartsCards.desapilar()+" ");
            }

            /* Se desapilan las cartas clubs y se muestran por pantalla */
            System.out.print("\n\nClubs desapiladas :\n\t");
            while (!clubsCards.estaVacio()) {
                System.out.print(clubsCards.desapilar()+" ");
            }
        /* Fin recorridos */
        }
}
```

La salida que genera el primer recorrido es:

```
Total cartas :
      [♥2] [♦2] [♥3] [♣4] [♣6] [♥J] [♠J] [♠Q] [♦K] [♠K] [♦A]
```

Como se puede observar, aunque se inserten las `Card` al `Deck` de forma desordenada, al recorrer la *cola con prioridad* se muestran en orden. El orden ascendente o descendente queda definido por el valor positivo o negativo que retorna el método `compareTo(Card card)` de `Card`.

Los siguientes cuatro recorridos corresponden a la acción de desapilar las `Card` de las colas para cada palo de la baraja:

```
Diamons desapiladas :
      [♦A] [♦K] [♦2]

Spades desapiladas :
      [♠K] [♠Q] [♠J]

Hearts desapiladas :
      [♥J] [♥3] [♥2]

Clubs desapiladas :
      [♣6] [♣4]
```