JAVA: `Threads`

- **`Thread` (hilo, tarea) es la clase base de *Java* para definir hilos de ejecución concurrentes dentro de un mismo programa.**
- **En Java, como lenguaje O.O., el concepto de concurrencia está asociado a los objetos:**
  - ✓ Son los objetos los que actúan concurrentemente con otros.
- **Las clases de objetos (hilos) que puedan actuar concurrentemente deben extender la clase `Thread`.**
  - ✓ Ejemplo:

```
class miClaseConcurrente extends Thread {...}
```

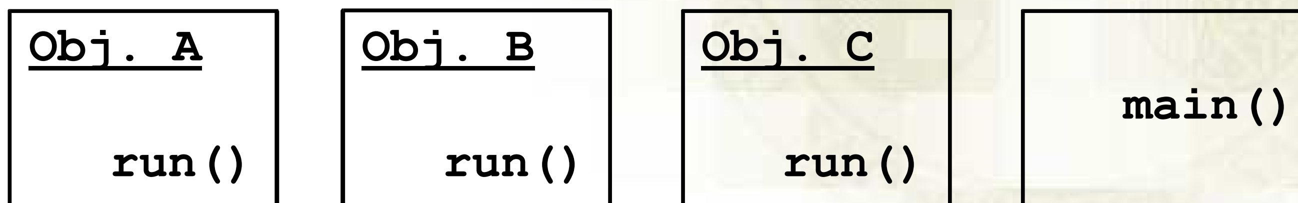
La clase `Thread`

- **Las clases derivadas de `Thread` deben de incluir un método:**

```
public void run ()
```

  - ✓ Este método especifica realmente la tarea a realizar.

- **La ejecución del método `run` de un `Thread` puede realizarse <u>concurrentemente</u> con otros métodos `run` de otros `Thread` y con el método `main`.**



Iniciar una tarea: `start`

- **El inicio de la ejecución de una tarea se realiza mediante el método `start()` heredado de `Thread`.**

- **`start()` es un método especial que invoca a `run()` y devuelve inmediatamente el control a la tarea que lo ha llamado.**

Ejemplo: Lanzar tareas concurrentes en Java

```
class claseConcurrente extends Thread {
    //...métodos
    //...atributos
    public void run () {
        //sentencias
    }
}
```

La tarea que será concurrente.

```
class miPrograma {
    public static void main (String[] args) {
        claseConcurrente tarea1, tarea2;
        tarea1 = new claseConcurrente();
        tarea2 = new claseConcurrente();
        //más sentencias
        tarea1.start();
        tarea2.start();
        //más sentencias;
}
```

Declaración y creación de objetos → NORMAL

Iniciar `tarea1` concurrentemente con `main()`. Invoca a `tarea1.run()` y vuelve **inmediatamente**.

Idem para `tarea2`.

Creación/Ejecución de hilos en Java: Resumen

- **Un objeto concurrente pertenece a una clase que extiende `Thread`.**
- **Hay que redefinir el método `run()` que especifica la tarea concurrente.**
- **La ejecución de la tarea concurrente se realiza mediante el método `start()` (heredado de `Thread`).**

Ejemplo: Hilo

La clase cuyos objetos pueden ser concurrentes

La tarea que será concurrente.

Detiene la ejecución del Hilo ("duerme")

```
class Hilo extends Thread {
    public Hilo (String str) {
        super(str);
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            }
             catch (InterruptedException e) {}
        }
        System.out.println("FIN! " + getName());
    }

}
```

Ejemplo: Test

```
public class Demo {
    public static void main (String[] args) {
        Hilo uno, dos;

        uno = new Hilo("Jamaica");
        dos = new Hilo("Fiji");

        uno.start();
        dos.start();

        System.out.println("main no hace nada");
    }
}
```

Ejemplo: Ejecución

```
public class Demo {
    public static void main (String[] args) {
          Hilo uno, dos;

          uno = new Hilo("Jamaica");
          dos = new Hilo("Fiji");

          uno.start();
          dos.start();

          System.out.println("main no hace nada");
    }
}
```

```
main no hace nada
0 Jamaica
0 Fiji
1 Jamaica
1 Fiji
2 Jamaica
2 Fiji
3 Jamaica
3 Fiji
4 Fiji
5 Fiji
4 Jamaica
6 Fiji
7 Fiji
5 Jamaica
8 Fiji
6 Jamaica
9 Fiji
7 Jamaica
8 Jamaica
FIN! Fiji
9 Jamaica
FIN! Jamaica
```

Ejemplo: Ejecución (start/run)

```
public class Demo {
    public static void main (String[] args) {
          Hilo uno, dos;

          uno = new Hilo("Jamaica");
          dos = new Hilo("Fiji");

          uno.run();
          dos.run();

          System.out.println("main no hace nada");
    }
}
```

```
0 Jamaica
1 Jamaica
2 Jamaica
3 Jamaica
4 Jamaica
5 Jamaica
6 Jamaica
7 Jamaica
8 Jamaica
9 Jamaica
FIN! Jamaica
0 Fiji
1 Fiji
2 Fiji
3 Fiji
4 Fiji
5 Fiji
6 Fiji
7 Fiji
8 Fiji
9 Fiji
FIN! Fiji
main no hace nada
```

Prioridades

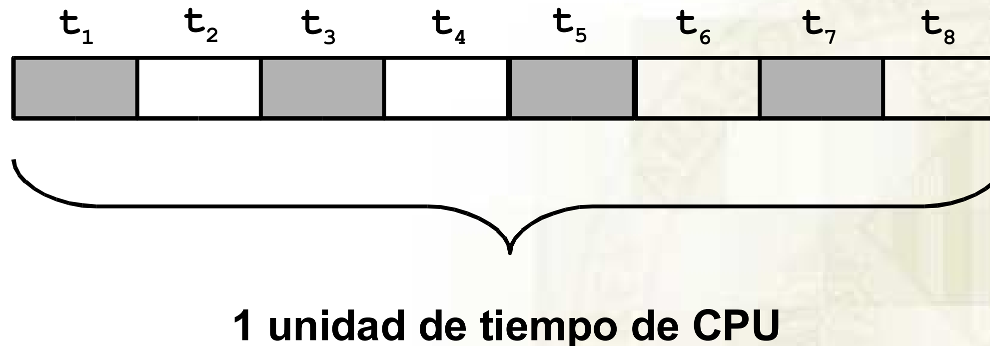
- **El criterio para la ejecución de múltiples hilos sobre una CPU se basa en prioridades:**
  - ✓ El sistema (JRE) selecciona para ejecutar en cada instante el hilo (ejecutable) con prioridad más alta.
    - A igualdad de prioridad, selección arbitraria.
- **Los hilos heredan la prioridad del hilo que los crea.**
- **La prioridad se puede cambiar:**

```
void setPriority (long x)
long getPriority ( )
```

- **Un hilo de prioridad baja sólo podrá ejecutarse cuando todos los hilos de prioridad superior pasen al estado "No ejecutable"**

Reparto de tiempo (*Time Slicing*)

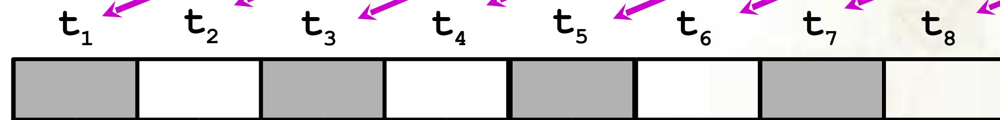
- **Método de asignación de tiempo de CPU a diferentes hilos de igual (o mayor) prioridad.**



- **Cada intervalo de tiempo (t_i) se selecciona un hilo a ejecutar en la CPU (prioridad >= al actual).**
- **Un hilo puede ceder su tiempo:**

```
static void yield ( )
```

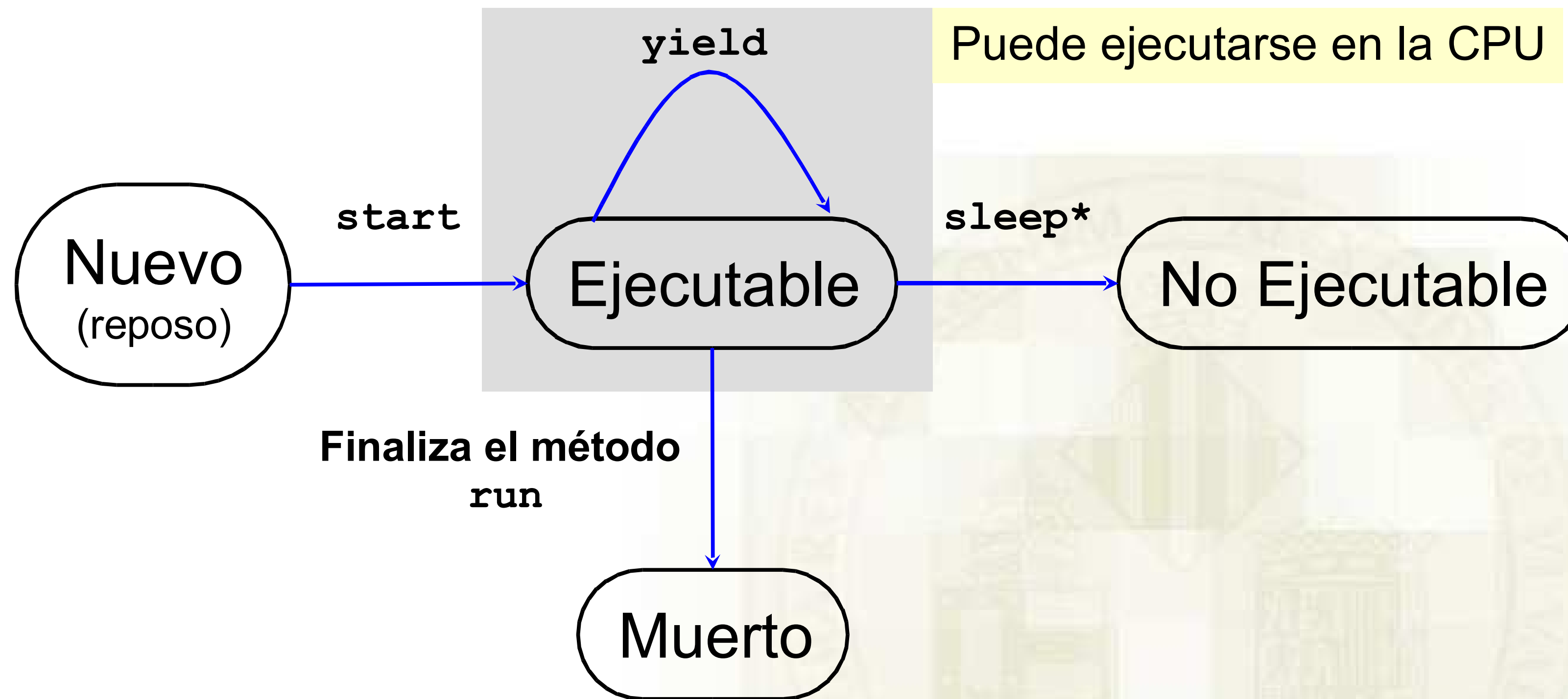
Reparto de tiempo (*Time Slicing*) (2)



1 unidad de tiempo de CPU

```
main no hace nada
0 Jamaica
0 Fiji
1 Jamaica
1 Fiji
2 Jamaica
2 Fiji
3 Jamaica
3 Fiji
4 Fiji
5 Fiji
4 Jamaica
6 Fiji
7 Fiji
5 Jamaica
8 Fiji
6 Jamaica
9 Fiji
7 Jamaica
8 Jamaica
FIN! Fiji
9 Jamaica
FIN! Jamaica
```

Ciclo de vida de un “hilo”



- **Un hilo está “vivo” si se encuentra en el estado “Ejecutable” o “No ejecutable”.**

```
boolean isAlive ( )
```

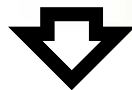
Definición de hilos: Interfaz `Runnable`

- **<u>Problema</u>: ¿Qué ocurre si se desea hacer concurrentes los objetos de una clase que hereda de otra, que no es `Thread`?**

  <u>Ejemplo</u>: `class Circulo extends Figura {...}`

  No puede heredar también de `Thread` → *Java* prohíbe la herencia múltiple.

- **<u>Solución</u>: *Java* admite heredar de 1 sola clase pero implementar múltiples interfaces → Proporciona un interfaz para clases concurrentes**



`Runnable`

Utilización del interfaz `Runnable`

- **Mínimas diferencias respecto al uso de `Thread`.**
- **La clase que se desee hacer concurrente debe de implementar `Runnable`:**

```
class Circulo extends Figura implements Runnable {...}
```

- **La clase debe de implementar el método `run()`, igual que con `Thread`.**

Utilización del interfaz `Runnable` (2)

- El lanzamiento de la tarea se hace a través de un `Thread`:

```
Circulo c = new Circulo();

Thread elHilo = new Thread ( c );

elHilo.start();
```

No se puede hacer directamente `c.start()`, porque `c` NO es un hilo (`Thread`)

Se crea un hilo de ejecución y se indica que el objeto `c` se va a ejecutar en él.

Se ejecuta concurrentemente el método run del objeto asociado con elHilo.

- El mecanismo es más indirecto que heredando de `Thread`, pero evita el problema de la herencia múltiple.

Revisión Ejemplo: Hilo2

```
class Hilo2 implements Runnable {
    private Thread miHilo = null;

    public Hilo2 (String str) {
        miHilo = new Thread (this, str);
        miHilo.start();
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + miHilo.getName());
            try {
                Thread.sleep((long)(Math.random() * 1000));
            }
             catch (InterruptedException e) {}
        }
        System.out.println("FIN! " + miHilo.getName());
    }

}
```

La clase implementa el interfaz "Ejecutable"

El Thread sobre el que se va a ejecutar

La tarea que será concurrente.

Revisión Ejemplo: Test2

```
public class Demo2 {
    public static void main (String[] args) {
          Hilo2 uno, dos;

          uno = new Hilo2("Jamaica");
          dos = new Hilo2("Fiji");

          System.out.println("main no hace nada");
    }
}
```

Crea y lanza los 2 hilos

Revisión Ejemplo: Ejecución

```
public class Demo2 {
    public static void main (String[] args) {
          Hilo2 uno, dos;

          uno = new Hilo2("Jamaica");
          dos = new Hilo2("Fiji");

          System.out.println("main no hace nada");
    }
}
```

```
main no hace nada
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Fiji
3 Fiji
2 Jamaica
3 Jamaica
4 Fiji
4 Jamaica
5 Jamaica
6 Jamaica
7 Jamaica
5 Fiji
6 Fiji
8 Jamaica
9 Jamaica
7 Fiji
FIN! Jamaica
8 Fiji
9 Fiji
FIN! Fiji
```

Sincronización de hilos en Java

- **Se trata básicamente de una sincronización basada en el concepto de *monitor*, aunque no existe ese tipo de datos como predefinido.**
  - ✓ No hace falta. Java asocia un monitor a todos los objetos.

- **El problema de la exclusividad de acceso a un objeto es tratado de manera automática por Java mediante la definición de métodos (o bloques) *`synchronized`*.**

- **El programador debe preocuparse de tratar la sincronización tipo Cooperación, que se realiza mediante mediante las operaciones `esperar/avisar` típicas.**

Definición de monitores en Java

- **Un monitor NO es un `Thread`, es un objeto "normal" al que pueden acceder varios `Thread`.**

- **Un monitor permite hacer un objeto "de acceso seguro" que:**
  - ✓ Asegura el uso excluyente del objeto → en cada instante sólo una tarea está haciendo uso de él.
    - Los método públicos se definen como `synchronized`.
  - ✓ Sincroniza el uso adecuado del objeto por parte de diferentes tareas.
    - Métodos `wait()` y `notify()`, `notifyAll()` de `Object`.

Bloqueo de objetos: `synchronized`

- **Permite asegurar el acceso competitivo a un objeto.**
- **Se pueden "sincronizar" métodos o bloques de sentencias.**
- **`synchronized` asegura que:**
  - ✓ Mientras un hilo esté ejecutando un método (bloque) sincronizado de un objeto, ningún otro hilo podrá ejecutar un método (bloque) sincronizado del mismo objeto.
- **La gestión de colas de tareas para este tipo de acceso es transparente para el programador.**

Sincronización Cooperativa: `wait - notify`

- **La clase `Object` tiene asociado un monitor y, por tanto, las operaciones de control:**
  - ✓ `wait ()`: El hilo actual se espera hasta que otro hilo envíe una notificación (notify) al mismo objeto.
  - ✓ `notify ()`: Activa un hilo que está esperando en el monitor del objeto.
  - ✓ `notifyAll()`: Activa todos los hilos que están esperando en el monitor del objeto.
    - Los hilos “compiten” por el objeto, sólo uno obtiene el bloqueo.

Ejemplo: Cola monitorizada

- **Se define un buffer de enteros mediante una cola circular.**

```
class Cola {
   private int [] _datos;
   private int _sigEnt, _sigSal, _ocupados, _tamano;

   public Cola ( int tam ) {
      _datos = new int [ tam ];
      _tamano = tam;
      _ocupados = 0;
      _sigEnt = 1;
      _sigSal = 1;
   }

   public synchronized void almacenar ( int x ) {...}

   public synchronized int extraer () {...}
}
```

Declaraciones típicas para esta Estructura de Datos

Antes de realizar la operación se va a asegurar de que no hay otro hilo usando el objeto Cola.

Ejemplo: Cola monitorizada (2)

```
public synchronized void almacenar ( int x ) {
    try {
        while ( _ocupados == _tamano ) wait ();
        _datos [ _sigEnt ] = x;
        _sigEnt = ( _sigEnt + 1 ) % _tamano;
        _ocupados++;
        notify();
    }
    catch ( InterruptedException e ) {}
}
```

Insertar en el buffer

No hay espacio → esperar

¡He terminado! Despertar a un hilo que esté esperando

wait puede lanzar Excepción

```
public synchronized int extraer () {
    int x = 0;
    try {
        while ( _ocupados == 0 ) wait();
        x = _datos [ _sigSal ];
        _sigSal = ( _sigSal + 1 ) % _tamano;
        _ocupados--;
        notify();
    }
    catch ( InterruptedException e ) {}
    return x;
}
```

Extraer del buffer

No hay datos → esperar

¡He terminado! Despertar a un hilo que esté esperando

Ejemplo: Tareas accediendo al buffer

```
//CLASE PRODUCTOR, ACCEDE PARA ALMACENAR
//Genera datos (int) y los almacena en
//el buffer indefinidamente.
class Productor extends Thread {

    private Cola _buffer;

    public Productor ( Cola c ) {
        _buffer = c;
    }

    public void run () {
        int valor = 0;
        while ( true ) {
            _buffer.almacenar ( valor );
            valor++;
        }
    }
}
```

El buffer donde va a almacenar, se asigna en el constructor.

Como el buffer está monitorizado, la tarea se despreocupa del sincronismo con otras tareas.

Ejemplo: Tareas accediendo al buffer (2)

```
//CLASE CONSUMIDOR, ACCEDE PARA EXTRAER
//Extrae datos (int) del buffer indefinidamente.
class Consumidor extends Thread {

   private Cola _buffer;

   public Consumidor ( Cola c ) {
       _buffer = c;
   }

   public void run () {
       int dato;
       while ( true ) {
           dato = _buffer.extraer ();
           System.out.println( dato );
       }
   }
}
```

El buffer donde va a almacenar, se asigna en el constructor.

Ejemplo: Tareas accediendo al buffer (3)

```
//El PROGRAMA que maneja el buffer y las tareas
class programaConHilos {

   public static void main (String [] args) {

      Cola el_buffer;      //buffer monitor
      Productor p;         //1 Tarea Productor
      Consumidor [] c;     //Varias Tareas Consumidor

      el_buffer = new Cola (50);
      p = new Productor ( el_buffer );
      c = new Consumidor [3];
      for (int i = 0; i < c.length; i++)
         c[i] = new Consumidor ( el_buffer );

      p.start();
      for (int i = 0; i < c.length; i++)
         c[i].start();
   }
}
```

Declarar referencias

Crear los objetos

Ejecutar las tareas

5 hilos concurrentes:
- **1 Productor**
- **3 Consumidores**
- **main()**

Ejecución

```
public static void main (String [] args) {
//...
   el_buffer = new Cola (5);
///...
}

class Consumidor extends Thread {
//...
   public void run () {
      int dato;
      while ( dato < 10 ) {
         dato = _buffer.extraer ();
         System.out.println( dato );
      }
   }
}
```

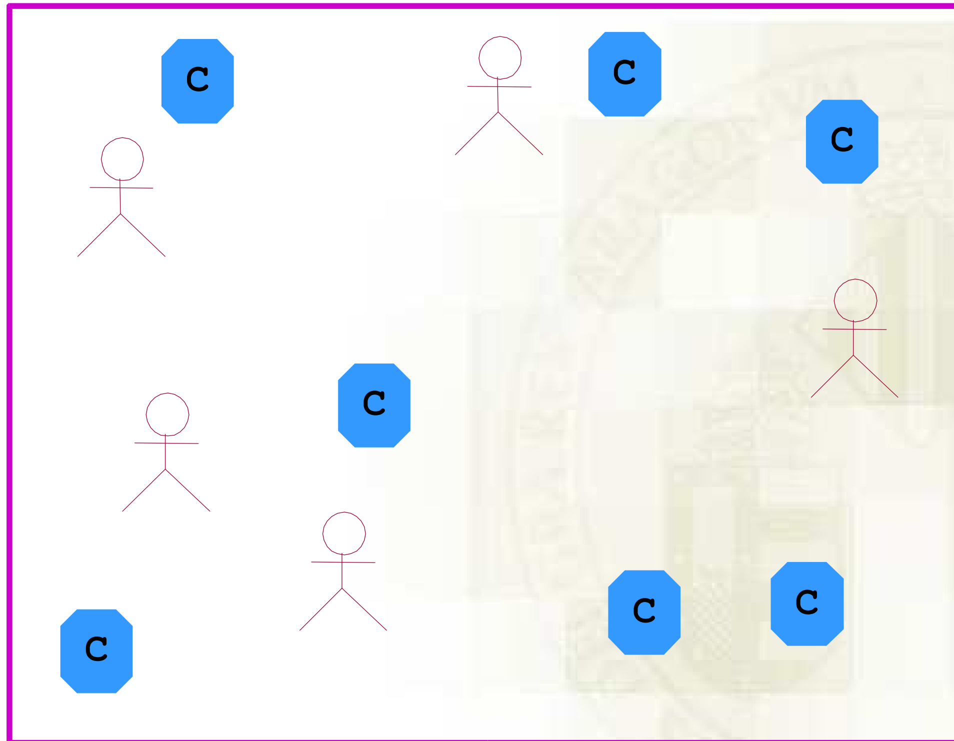
```
Productor 0
Productor 1
Productor 2
Productor 3
Productor 4
Productor 5
Consum0 : 0
Consum0 : 1
Consum0 : 2
Productor 6
Consum1 : 3
Consum2 : 4
Consum0 : 5
Productor 7
Consum1 : 6
Consum2 : 7
Productor 8
Consum1 : 8
Productor 9
Consum1 : 9
Consum0 : 10
```

```
Consum0 : 10
FIN: Consum0
Productor 10
Productor 11
Productor 12
Productor 13
Consum1 : 11
FIN: Consum1
Consum2 : 12
FIN: Consum2
Productor 14
Productor 15
Productor 16
Productor 17

Kill
```

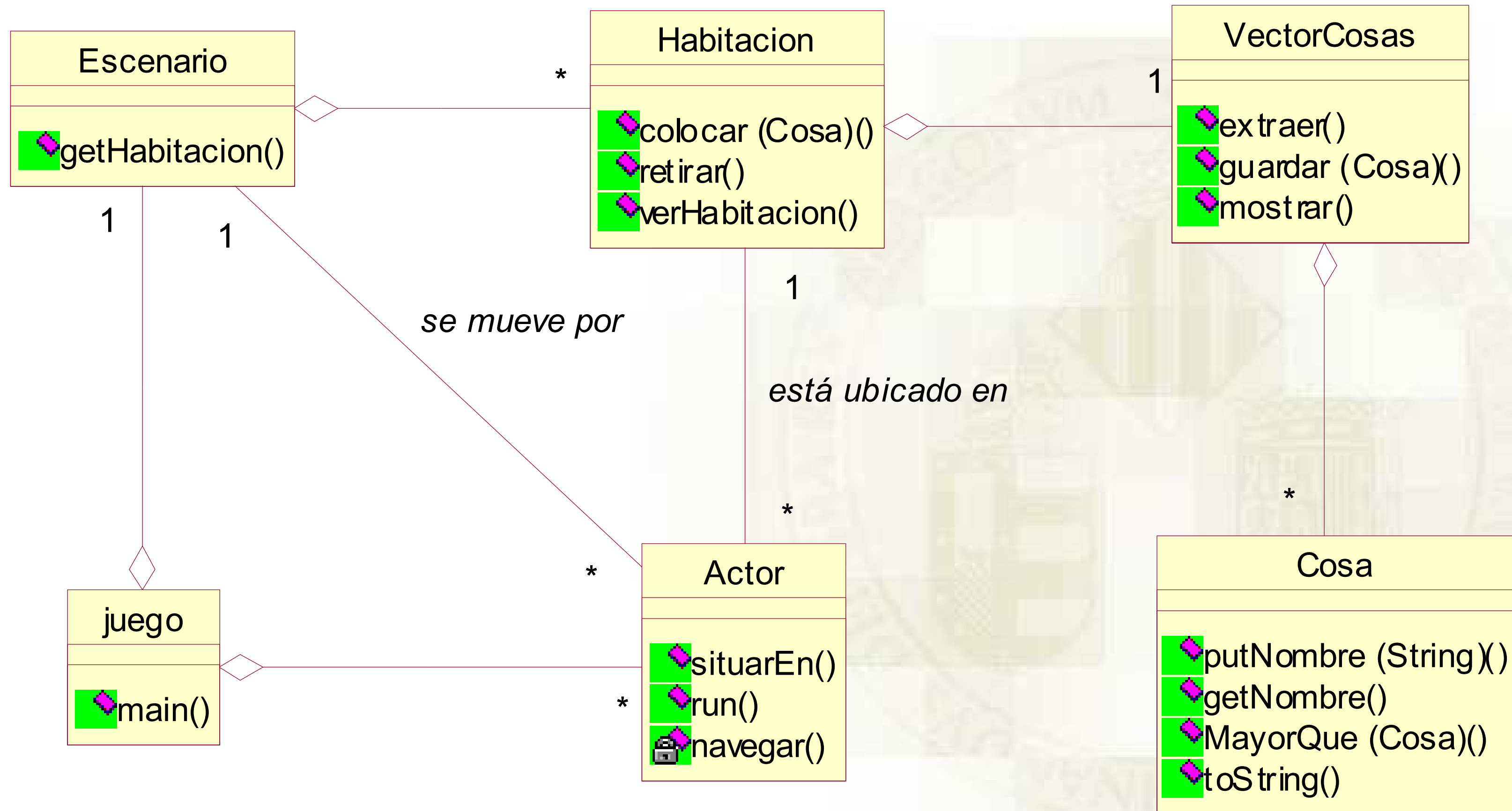
Ejemplo: Un mundo virtual

- **Múltiples humanoides virtuales se mueven en un escenario.**
- **Un escenario consta de varias habitaciones.**
- **En las habitaciones hay objetos (cosas) que los humanoides pueden ver y coger.**
- **Los humanoides se desplazan por la habitación observando, cogiendo cosas y dejando cosas.**
- **Los humanoides pueden tener un hilo de ejecución independiente.**
- **Los humanoides dentro de la misma habitación interaccionan cogiendo y dejando objetos en la habitación.**

Ejemplo: Un mundo virtual (2)



Ejemplo: Diagrama UML del problema



Clase juego

```
public class juego {

   public static void main (String[] args) {

      final int N_HAB = 1; //habitaciones del escenario
      final int N_ACT = 2; //actores en el escenario

      Escenario e = new Escenario(N_HAB);
      Actor[] agente = new Actor[N_ACT];

      for (int i=0; i<N_ACT; i++) {
         agente[i] = new Actor(e,"Agente " + i);
         agente[i].situarEn((int)( Math.random() * N_HAB ));
         agente[i].start();
      }

   }

}
```

Clase Escenario

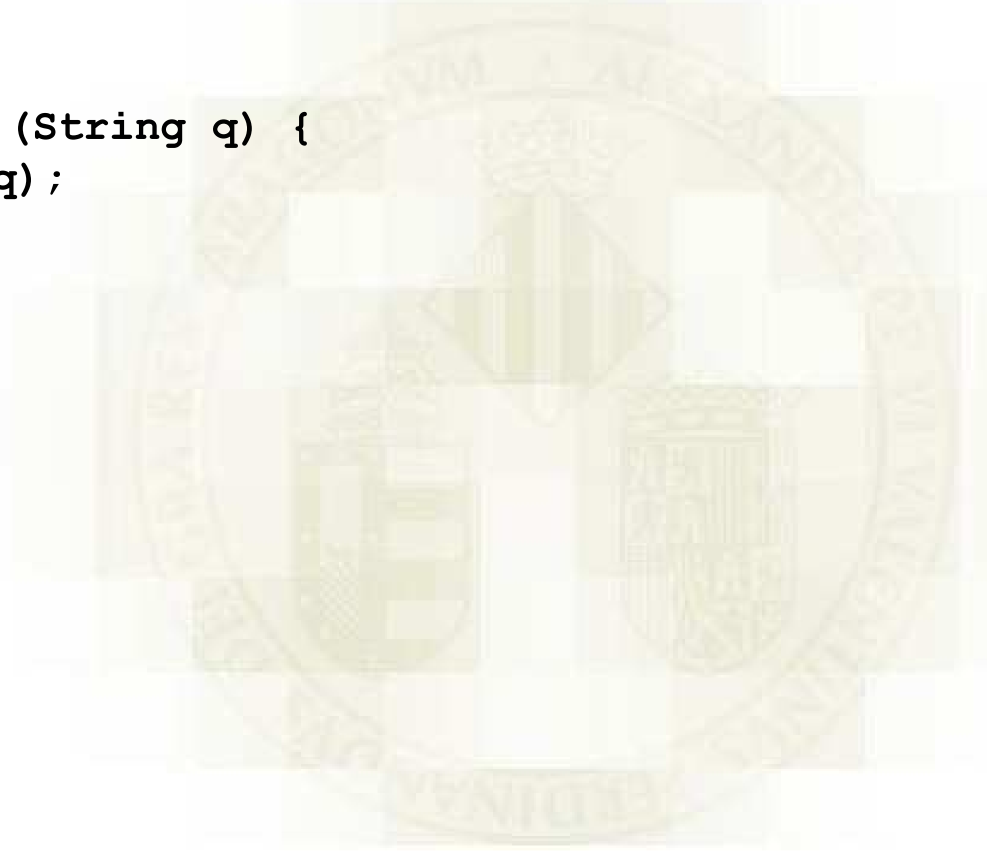
```
class Escenario {

  private Habitacion [] elementos; //habitaciones del escenario
  private static final int n_cosas = 10; //num. cosas en la habitacion

  public Escenario (int num) {
    elementos = new Habitacion[num];
    for (int i=0; i< num; i++)
      elementos[i] = new Habitacion(n_cosas);
  }

  public Habitacion getHabitacion (int ind) {
    return elementos[ind];
  }

}
```

Clase Habitacion

```
class Habitacion {

   private VectorCosas contenido; //contenedor de cosas
   private int numCosa; //para identificar las cosas

   public Habitacion (int n_cosas) {
      int pos;

      this.numCosa = 0;
      this.contenido = new VectorCosas();
      for (int i=0; i<n_cosas; i++) {
         pos = (int)( Math.random() * 100.0 );
         this.colocar ( new Cosa(pos, pos, pos), "" );
      }
   }

   public void colocar (Cosa c, String q){
      this.numCosa ++;
      c.putNombre (this.numCosa);
      this.contenido.guardar (c, q);
   }
```

Clase Habitacion (2)

```
public void retirar (String q){
   this.contenido.extraer(q);
}

public void verHabitacion (String q) {
   this.contenido.mostrar(q);
}


}//fin Habitacion
```



Clase VectorCosas

```
class VectorCosas {

   private static final int MAX = 100;
   private int tam;
   private Cosa[] lista;

   public VectorCosas () {
      lista = new Cosa [MAX];
      tam = 0;
   }

   public synchronized void mostrar (String quien) {
      int p = 0;

      System.out.println (quien + " ve (" + tam + "):" );
      for (int i=0; i<tam; i++)
        System.out.println("    " + quien + ", " + lista[i] );
   }
```

Clase VectorCosas (2)

```
public synchronized void guardar (Cosa c, String quien) {
   boolean encontrado = false;
   int p = 0;

   try {
      while (tam == MAX ) {//puedo almacenar?
         System.out.println (quien + " espera poder colocar Cosa.");
         wait();
      }
      System.out.println(quien + " coloca C" + c.getNombre());
      while ( (p < tam) && (! encontrado) ) {//almaceno ordenado
         if ( lista[p].mayorQue(c) ) encontrado = true;
         else p++;
      }
      if ( ! encontrado )
         lista[tam] = c;
      else {
         for (int i=tam-1; i>=p; i--)
            lista[i+1] = lista [i];
         lista[p] = c;
      }
      tam++;
      notifyAll(); //ya esta, notifico.
   }
   catch (InterruptedException e) {}
}
```

Clase VectorCosas (3)

```
  public synchronized void extraer (String quien) {
     int eleccion = (int) ( Math.random() * tam );

     try {
        //puedo extraer?
        while ( tam == 0 ) {
           System.out.println (quien + " espera poder eliminar Cosa.");
           wait();
        }
        //elimino "eleccion"
        System.out.println(quien + " elimina C" + lista[eleccion].getNombre());
        for (int i=eleccion + 1; i<tam; i++)
           lista[i-1] = lista[i];
        tam--;
        //ya esta, notifico.
        notifyAll();
     }
     catch (InterruptedException e) {}
  }

}//fin ListaCosas
```

Clase Cosa

```
class Cosa {

   private int x, y, z; //posicion 3D
   private int nom; //identificador de cada objeto

   public Cosa () {
      this (0, 0, 0);
   }

   public Cosa (int a, int b, int c) {
      this.x = a;
      this.y = b;
      this.z = c;
   }

   public void putNombre (int n) {
      this.nom = n;
   }
```

Clase Cosa (2)

```
public boolean mayorQue ( Cosa c ){
   boolean mayor = false;

   if ( this.x > c.x ) mayor = true;
   else if ( this.y > c.y ) mayor = true;
      else if ( this.z > c.z ) mayor = true;
   return mayor;
}

public String toString () {
   String str;

   str = "C" + this.nom + " (" + this.x + ", " + this.y + ", "
            + this.z + ")";
   return str;
}

}//fin Cosa
```

Clase Actor

```
class Actor extends Thread {
  private Escenario mundo; //escenario en el que se mueve
  private Habitacion donde; //habitacion en la que esta
  private String nombre; //identificador

  public Actor (Escenario e, String nom){
    this.mundo = e;
    this.nombre = nom;
    this.situarEn(0);
  }

  public void situarEn (int h){
    this.donde = this.mundo.getHabitacion(h);
  }

  private void navegar() {
    try {
      sleep ((long) (Math.random() * 100.0));
    }
    catch (InterruptedException e) {}
  }
```

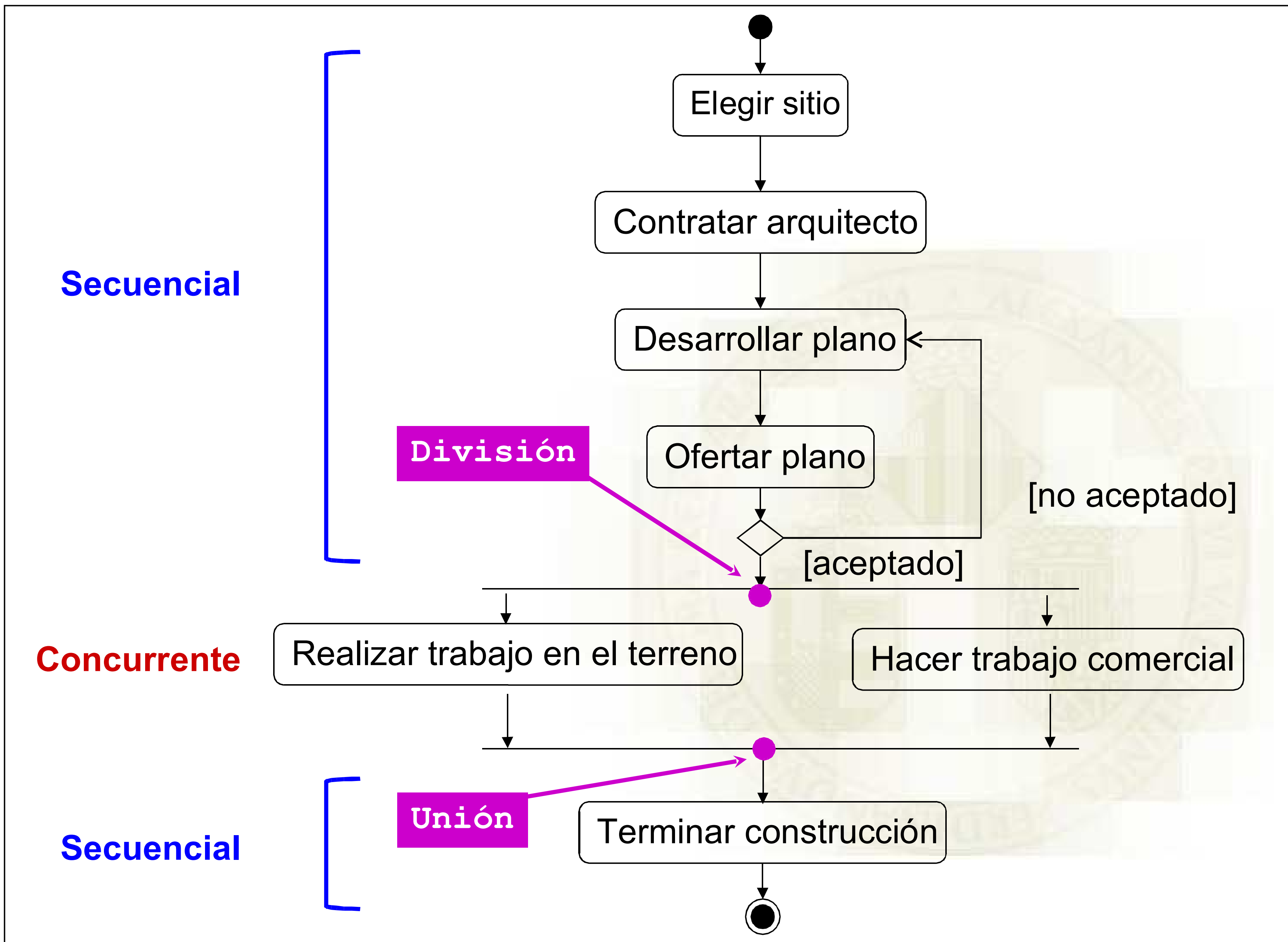
Clase Actor (2)

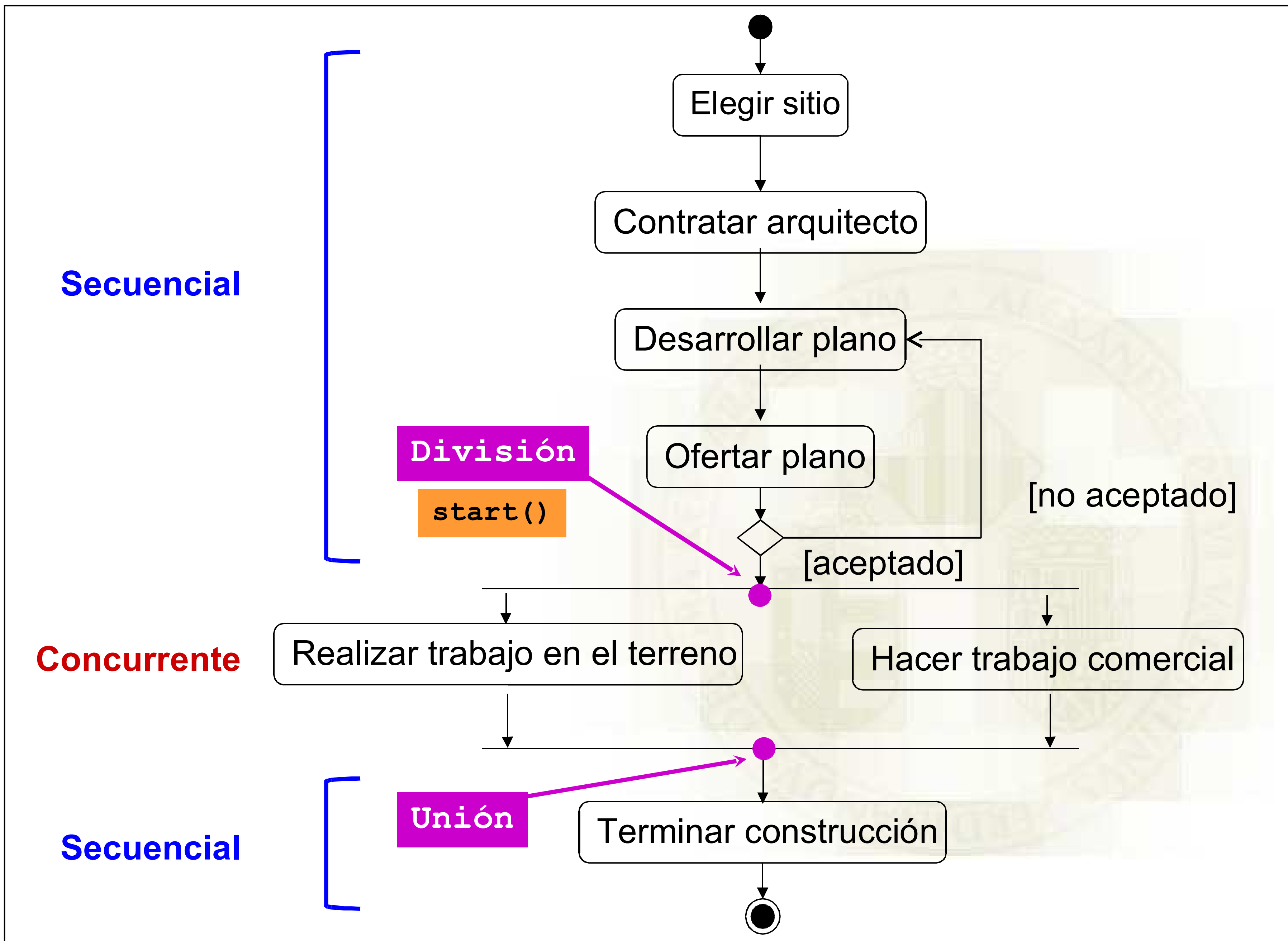
```
  public void run () {
    int pos;
    int veces = 0, cuantos;
    final int LIM = 3;

    while ( veces < LIM ) {
      this.donde.verHabitacion (nombre); //observar
      this.navegar(); //moverse
      cuantos = (int) (Math.random() * 3);
      for (int i=0; i<cuantos; i++)
        this.donde.retirar(nombre);
      this.navegar();
      cuantos = (int) (Math.random() * 3);
      for (int i=0; i<cuantos; i++) {
        pos = (int) (Math.random()*100.0);
        this.donde.colocar( new Cosa (pos, pos, pos), nombre );
      }
      System.out.println("fin ciclo de " + nombre);
      veces++;
    }
      }
}//fin Actor
```

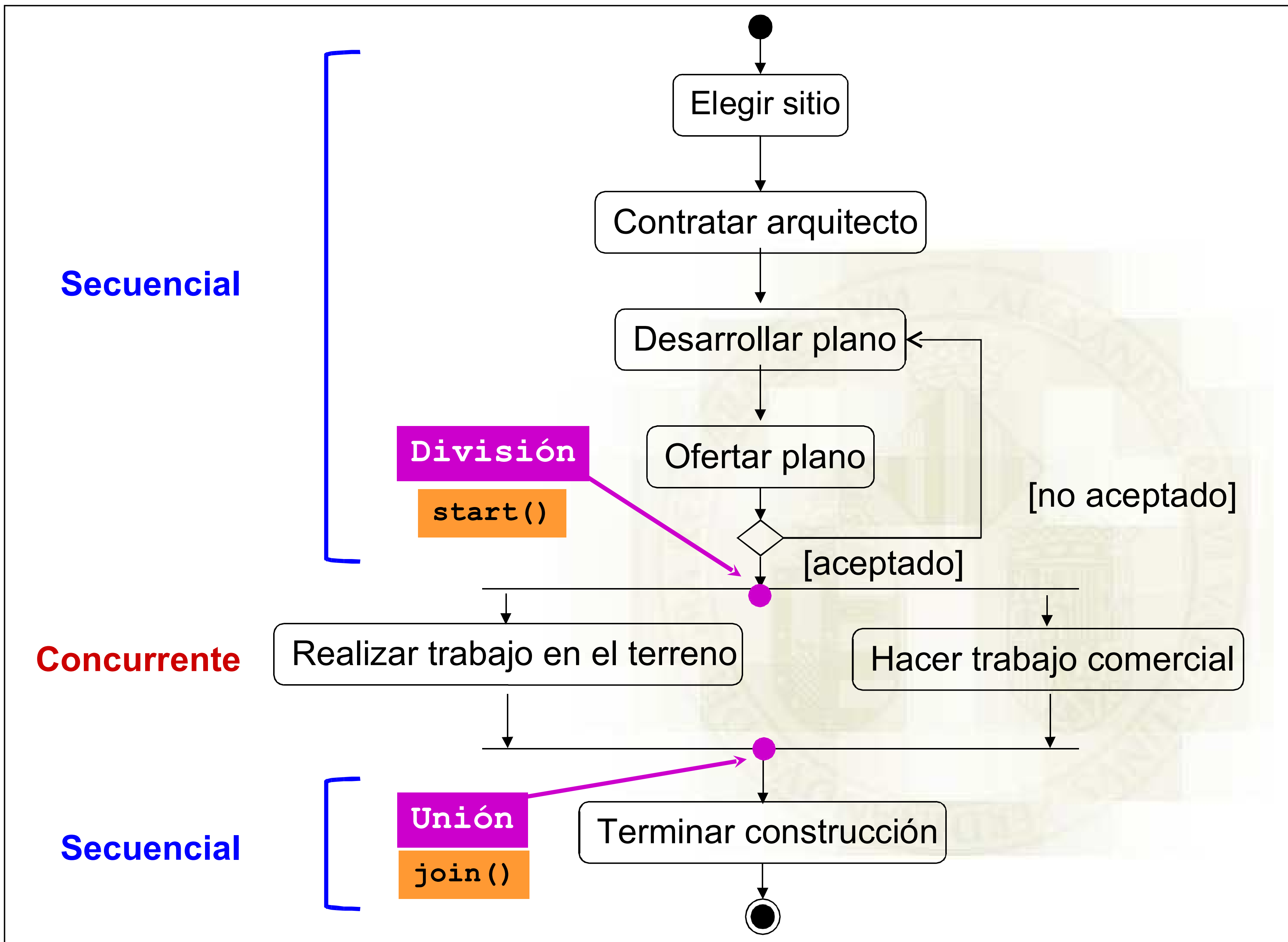
Reunificación de tareas

- **Un programa concurrente, en general, tiene bloques secuenciales:**
  - ✓ Existe concurrencia en partes concretas del programa, no necesariamente en todo él.

- **Después de bloques concurrentes puede ser preciso reunificar el flujo de control del programa para continuar la secuencia.**


Secuencial
Elegir sitio
Contratar arquitecto
Desarrollar plano
División
Ofertar plano
[no aceptado]
[aceptado]
Concurrente
Realizar trabajo en el terreno
Hacer trabajo comercial
Unión
Secuencial
Terminar construcción


Secuencial
Elegir sitio
Contratar arquitecto
Desarrollar plano
División
start()
Ofertar plano
[no aceptado]
[aceptado]
Concurrente
Realizar trabajo en el terreno
Hacer trabajo comercial
Unión
Secuencial
Terminar construcción


Secuencial
Elegir sitio
Contratar arquitecto
Desarrollar plano
División
start()
Ofertar plano
[no aceptado]
[aceptado]
Concurrente
Realizar trabajo en el terreno
Hacer trabajo comercial
Unión
join()
Secuencial
Terminar construcción

Método `join()`

- **Permite la reunificación:**

  `h.join ();`

  ✓ El hilo activo espera a que "muera" (finalice `run()`) el hilo `h`.

Ejemplo:

```
elegirSitio();
contratarArquitecto();
desarrollarPlano();
do {
        ofertarPlano();
} while ( ! aceptado );
trabajoTerreno.start();
trabajoComercial.start();
trabajoTerreno.join();
trabajoComercial.join();
terminarConstruccion;
```

Concurrente (trabajoTerreno.start() … trabajoComercial.join())

División: trabajoTerreno.start(); trabajoComercial.start();

Unión: trabajoTerreno.join(); trabajoComercial.join();

Ejemplo (join) (test)

```
public class jointest {
   public static void main (String[] args) {
      Vector a, b;

      a = new Vector (50, "a");
      b = new Vector (100, "b");
      a.start();
      b.start();
      try {
         a.join();
         b.join();
      }
      catch (InterruptedException e) {}

      System.out.println("Suma (x) a: " + a.suma());
      System.out.println("Suma (x) b: " + b.suma());
      System.out.println("Suma (x^2) a: " + a.sumaSqr());
      System.out.println("Suma (x^2) b: " + b.sumaSqr());
      System.out.println("Media a: " + a.media());
      System.out.println("Media b: " + b.media());
   }
}
```

Ejemplo (join)

```
class Vector extends Thread {

   private int[] datos;
   private String nombre;
   private int tam;
   private int suma;
   private long sumaSqr;
   private double media;

   Vector (int t, String n) {
      this.nombre = n;
      this.tam = t;
      this.suma = 0;
      this.sumaSqr = 0;
      this.media = 0.0;
      this.datos = new int [ this.tam ];
      for (int i=0; i < this.tam; i++) {
         datos[i] = (int) (Math.random () * 100.0);
      }
   }
```

Ejemplo (join) (2)

```
public int suma () {
  return (this.suma);
}

public long sumaSqr () {
  return (this.sumaSqr);
}

public double media () {
  return (this.media);
}

public void run () {
  for (int i=0; i< this.tam; i++) {
    System.out.println ( this.nombre + ": " + i );
    this.suma += datos[i];
    this.sumaSqr += datos[i] * datos[i];
  }
  this.media = (double) this.suma / (double) this.tam;
}

} //fin clase Vector
```

Ejemplo (indicador de presión)

```
//Indicador de presion
public class indicador {
   static int presion = 0;
   static final int limiteSeguridad = 20;

   public static void main ( String[] args ) {
      trabajador[] t = new trabajador[10];

      for (int i=0; i<10; i++) {
         t[i] = new trabajador();
         t[i].start();
      }

      try {
         for (int i=0; i<10; i++)
            t[i].join();
      }
      catch (InterruptedException e){}

      System.out.println ("Lectura de indicador: " + presion + "(limite 20)");
   }
```

Ejemplo (indicador de presión)

```
static class trabajador extends Thread {
  public void run () {
    if ( indicador.presion < (indicador.limiteSeguridad - 15) ) {
      try {
        sleep (100); //simula retraso en activar el control
      }
      catch (InterruptedException e){}
      indicador.presion += 15;
    }
    else ; //no hacer nada presión demasiado elevada
  }
}
```

Ejemplo (indicador de presión, v.2)

```
//Indicador de presion
public class indicador2 {
   static int presion = 0;
   static final int limiteSeguridad = 20;
   static Object sincObj;

   public static void main ( String[] args ) {
      sincObj = new Object();
      trabajador[] t = new trabajador[10];

      for (int i=0; i<10; i++) {
         t[i] = new trabajador();
         t[i].start();
      }

      try {
         for (int i=0; i<10; i++)
            t[i].join();
      }
      catch (InterruptedException e){}

      System.out.println ("Lectura de indicador: " + presion + "(limite 20)");
   }
```

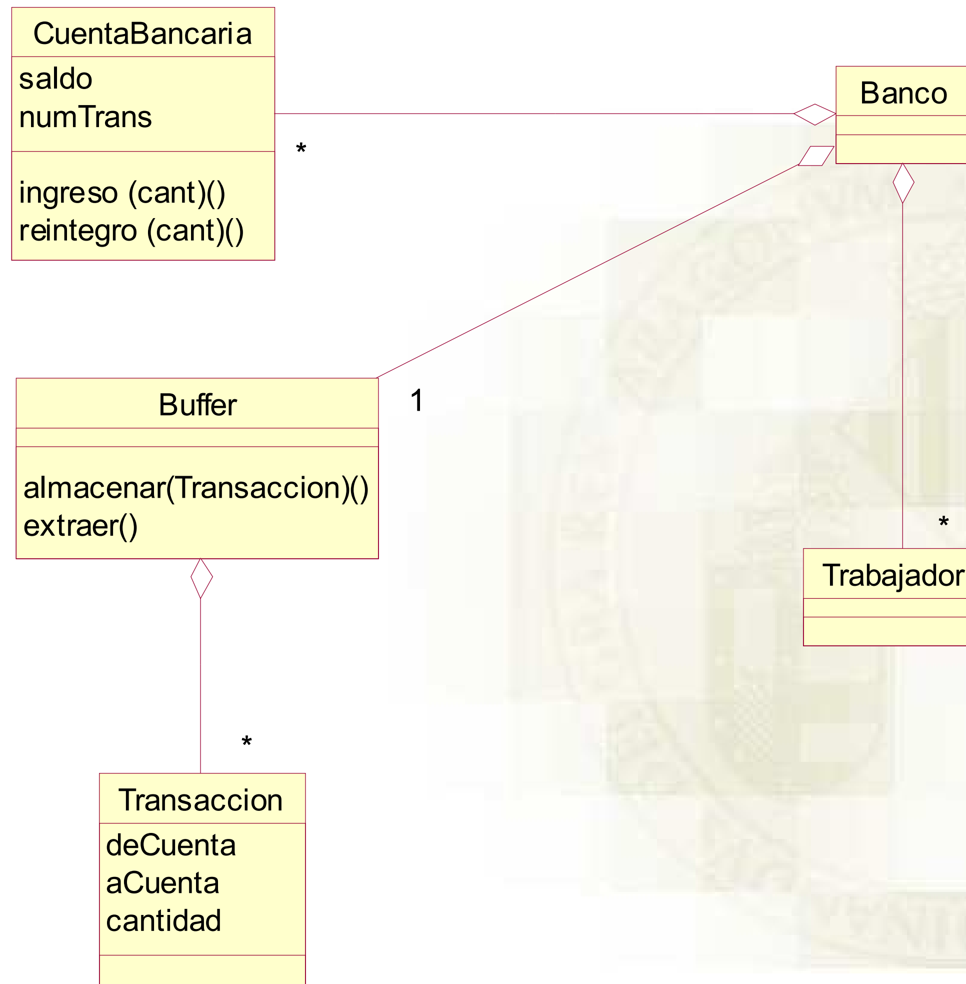
Ejemplo (indicador de presión, v.2)

```
static class trabajador extends Thread {
   public void run () {
      synchronized (indicador2.sincObj) {
         if ( indicador2.presion < (indicador2.limiteSeguridad - 15) ) {
            try {
               sleep (100); //simula retraso en activar el control
            }
            catch (InterruptedException e){}
            indicador2.presion += 15;
         }
         else ; //no hacer nada presión demasiado elevada
      }
   }
}
```

Problema: Transacciones bancarias

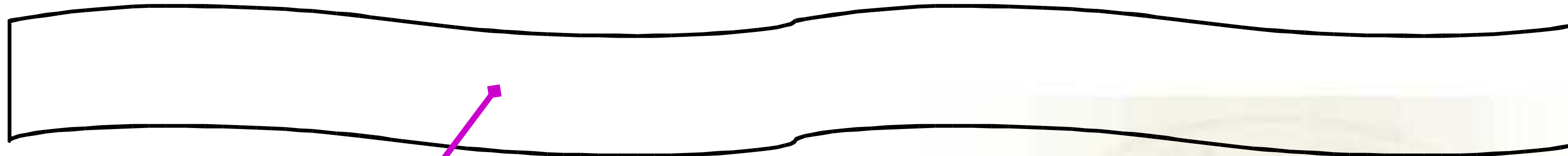
- **Registrar transferencias de dinero entre diferentes cuentas bancarias.**
- **La información sobre las transacciones están grabadas en un archivo texto.**
- **Existe una tarea que se encarga de leer la información del archivo y almacenarla en un Buffer en memoria.**
- **Múltiples trabajadores pueden recoger del Buffer información sobre transferencias y hacerlas efectivas.**

Problema: Transacciones bancarias



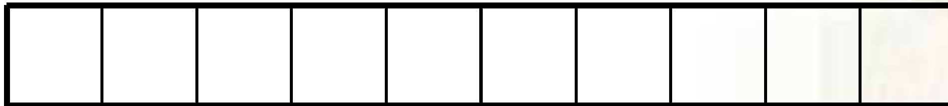
Problema: Transacciones bancarias

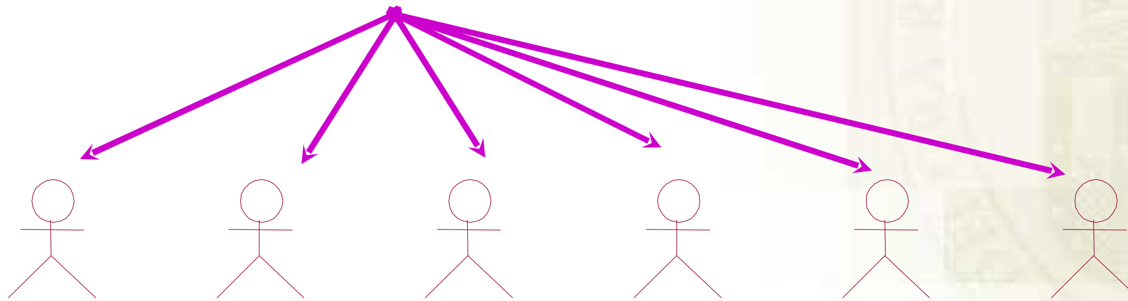
Archivo con transferencias



Cuentas de Clientes

Buffer





Cajeros