Monitores

Programación Concurrente 2016
UNQ

Recap

- Los programas concurrentes usan eficientemente los recursos computacionales

- Sin embargo, si hay memoria compartida *interleavings* indeseados pueden llevar a resultados incorrectos

- Los Semáforos son una herramienta eficiente para resolver problemas de sincronización

Semáforos - Desventajas

- Son de muy bajo nivel:
  - Como un "*Goto*" en programación
    - Es fácil cometer errores (ausencia de releases, acquires)
- No están vinculados a datos
  - Pueden aparecer en cualquier parte del código

Monitores

- Combinan Tipos Abstractos de Datos (TAD) y exclusión mutua
  - Propuesto por Tony Hoare [1974]
- Incorporado en lenguajes de programación modernos
  - C#
  - Java

Elementos Principales

- Conjunto de operaciones encapsuladas en módulos (como una clase)

- Un único lock (mutex) — que asegura exclusión mutua en todas las operaciones del monitor

- Variables especiales (llamadas “variables de condición”) — son usadas para programar sincronización condicional

Ejemplo Counter

- Construir un contador con dos acciones: `inc()` y `dec()`.
- Ningún thread debe poder modificar simultáneamente el valor del contador
  - Pensar una solución usando Semáforos
  - Pensar una solución usando Monitores

Contador - Semáforos

```
class Counter {

  private int c = 0;
  private Semaphore mutex = new Semaphore(1);

  public void inc() {
    mutex.acquire();
    c++;
    mutex.release();
  }

  public void dec() {
    mutex.acquire();
    c--;
    mutex.release();
  }
}
```

Contador - Monitores

```
monitor Counter {

   private int c = 0;

   public void inc() {
      c++;
   }

   public void dec() {
      c--;
   }
}
```

Condition Variables

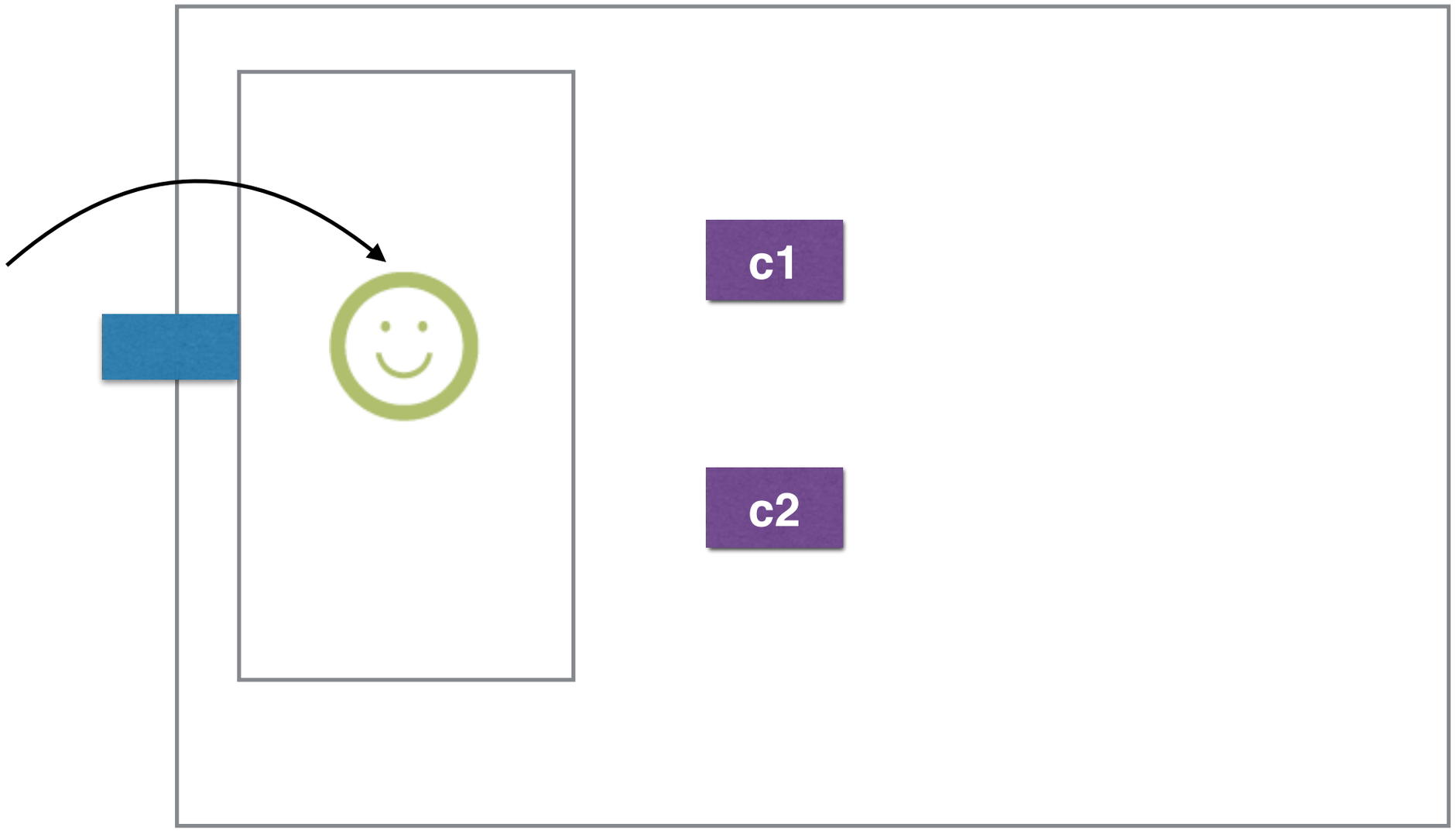
- Están ligadas al monitor
- Poseen dos operaciones
  - c.wait()
  - c.notify()
- Como los semáforos, tienen asociadas una cola de procesos bloqueados

wait/notify

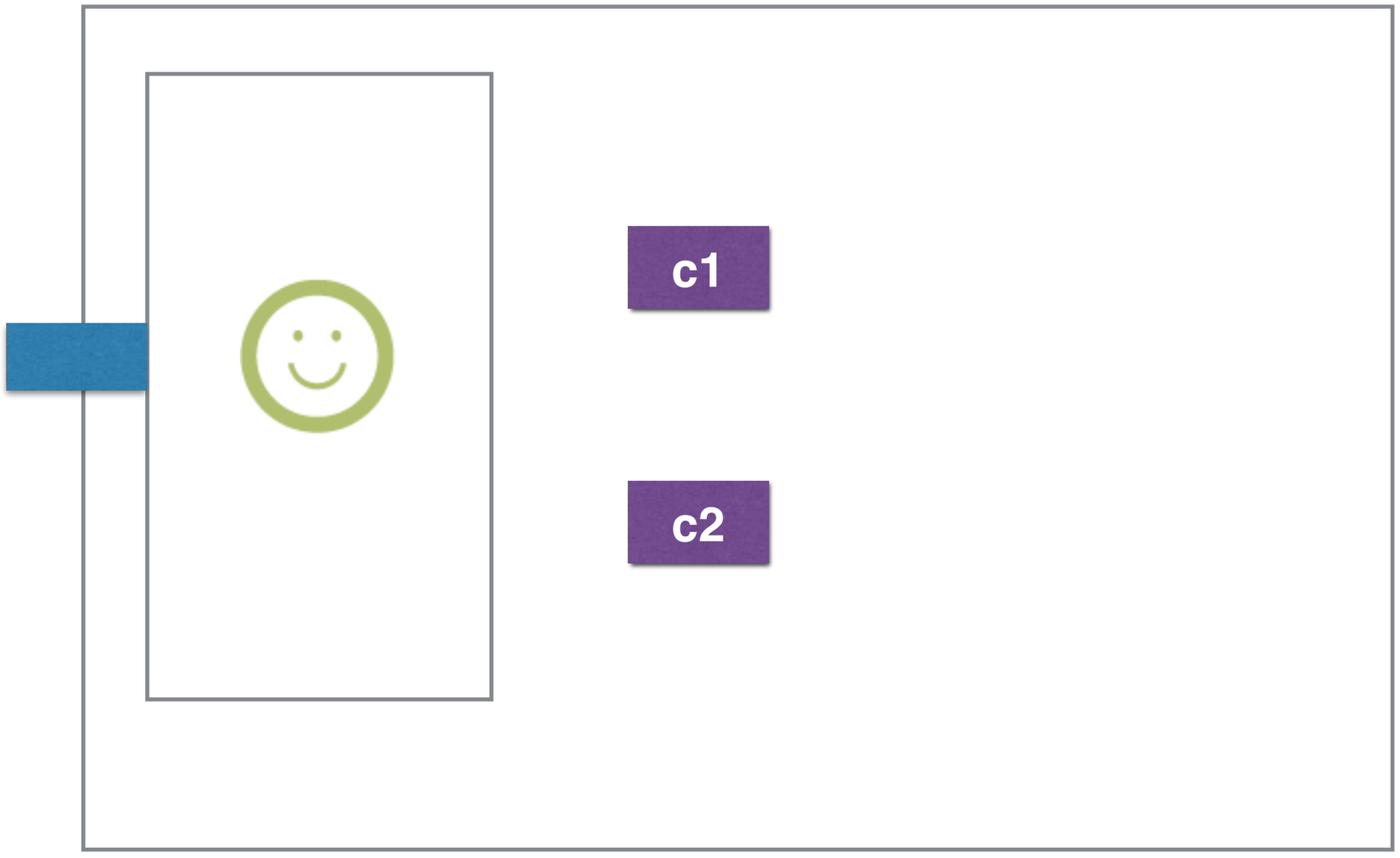
- Dado una variable de condición C:
  - `C.wait()`
    - bloquea **siempre** al proceso y lo coloca en la cola de espera de la variable C.
    - Al bloquearse, suelta el mutex del monitor.
  - `C.notify()`
    - desbloquea al primer proceso en la cola de espera de la variable C, y continua la ejecución.
    - Si no hay procesos en espera no tiene ningún efecto.

Gráficamente

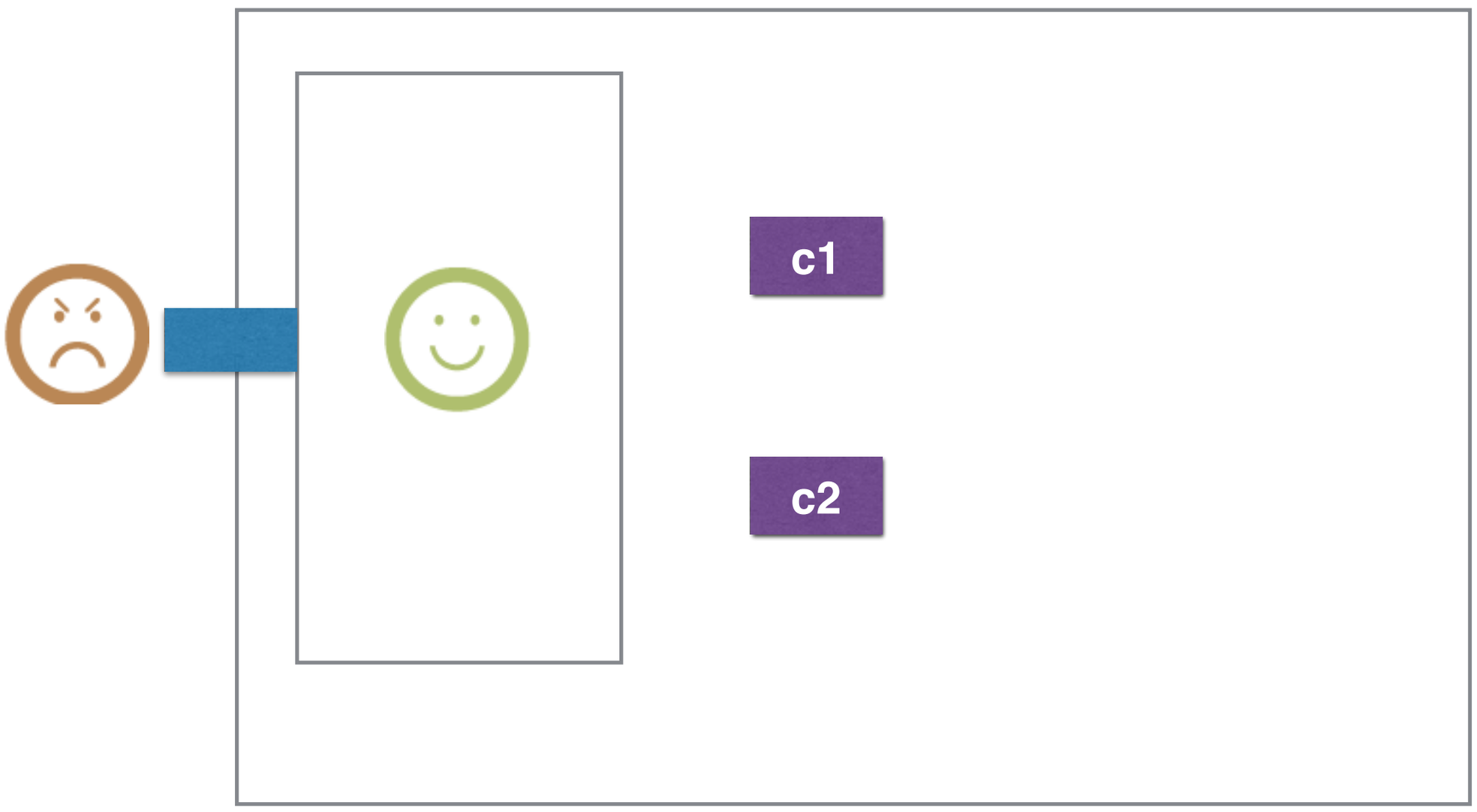
Gráficamente



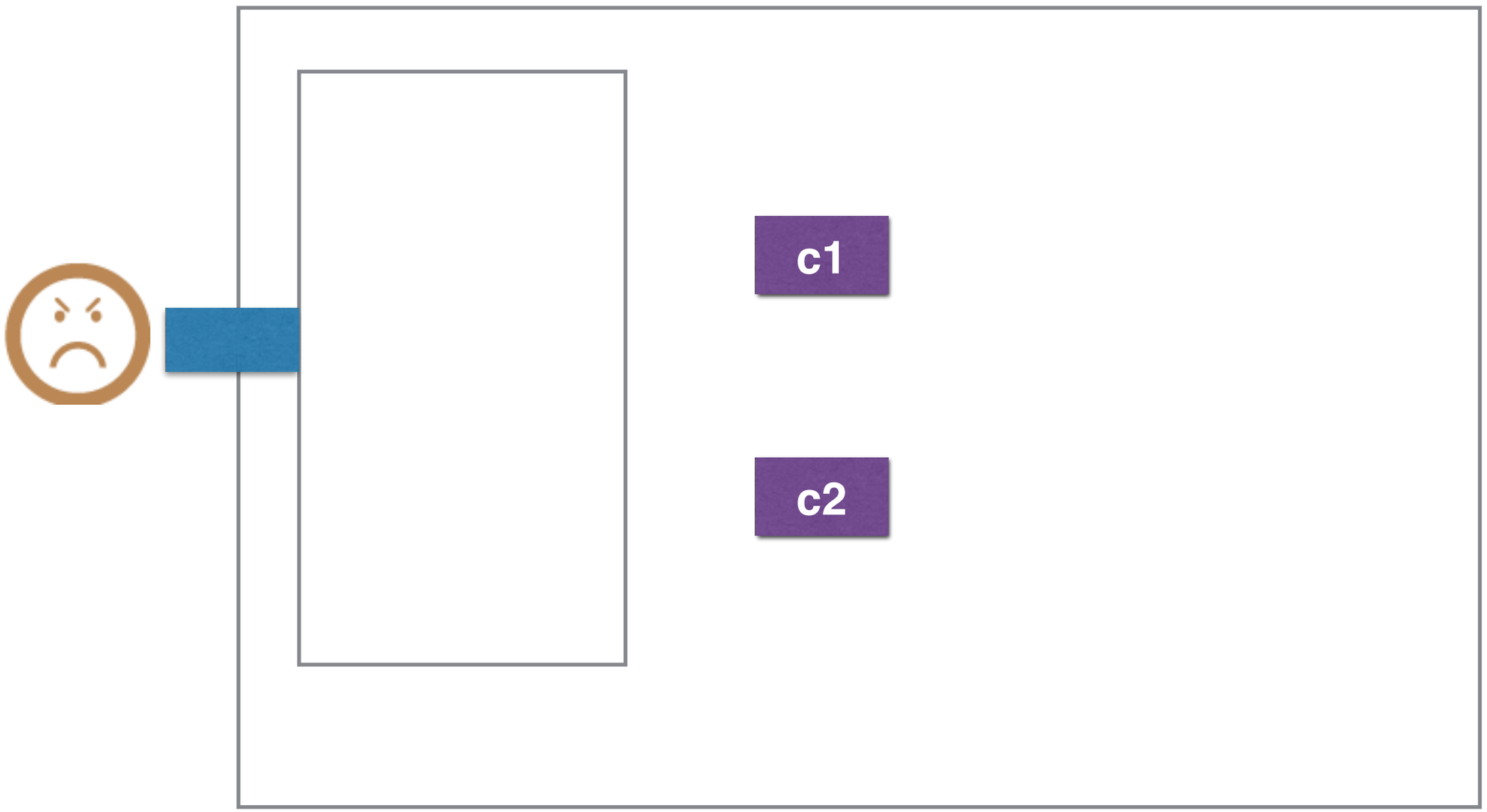
Gráficamente



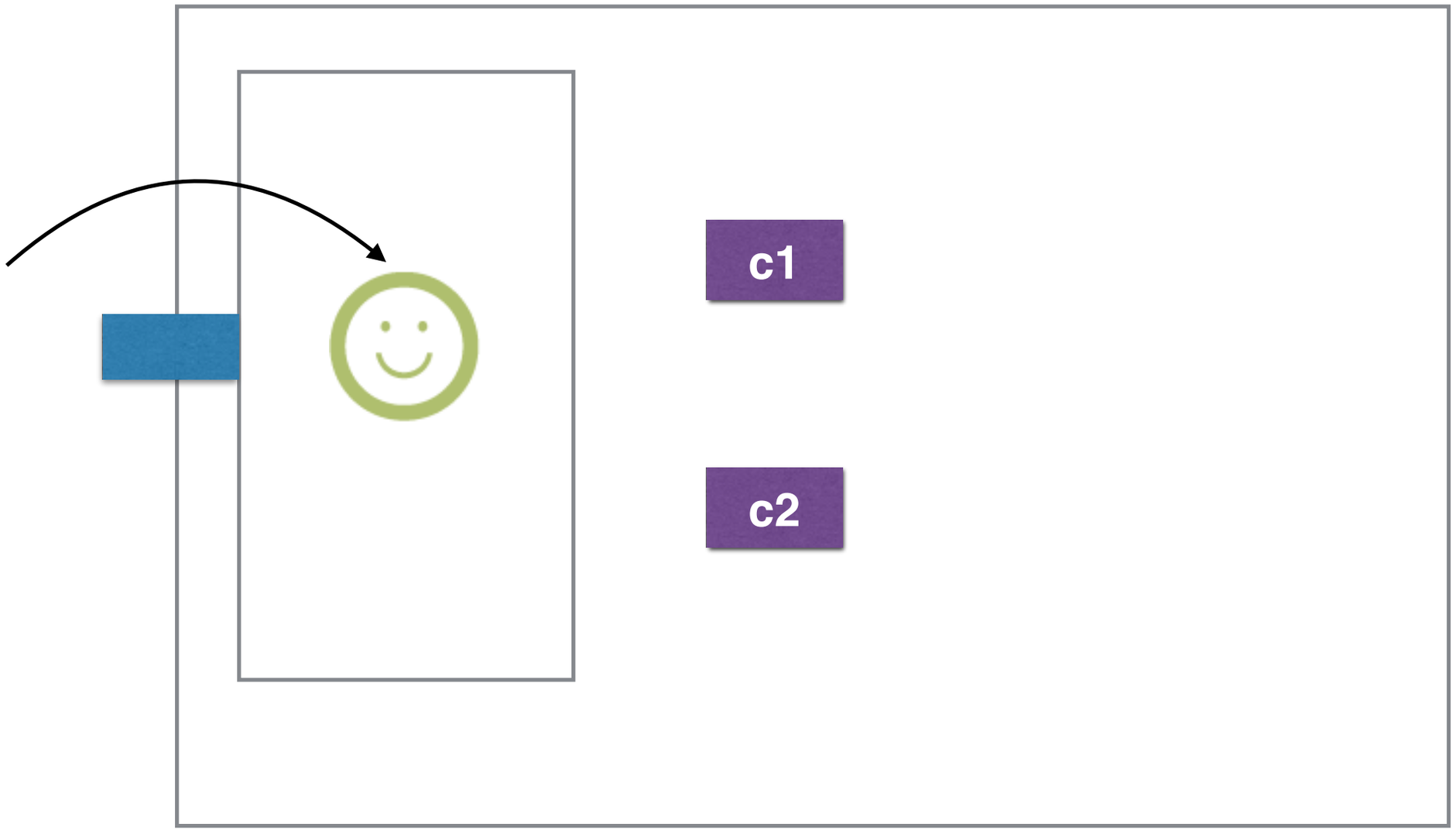
Gráficamente



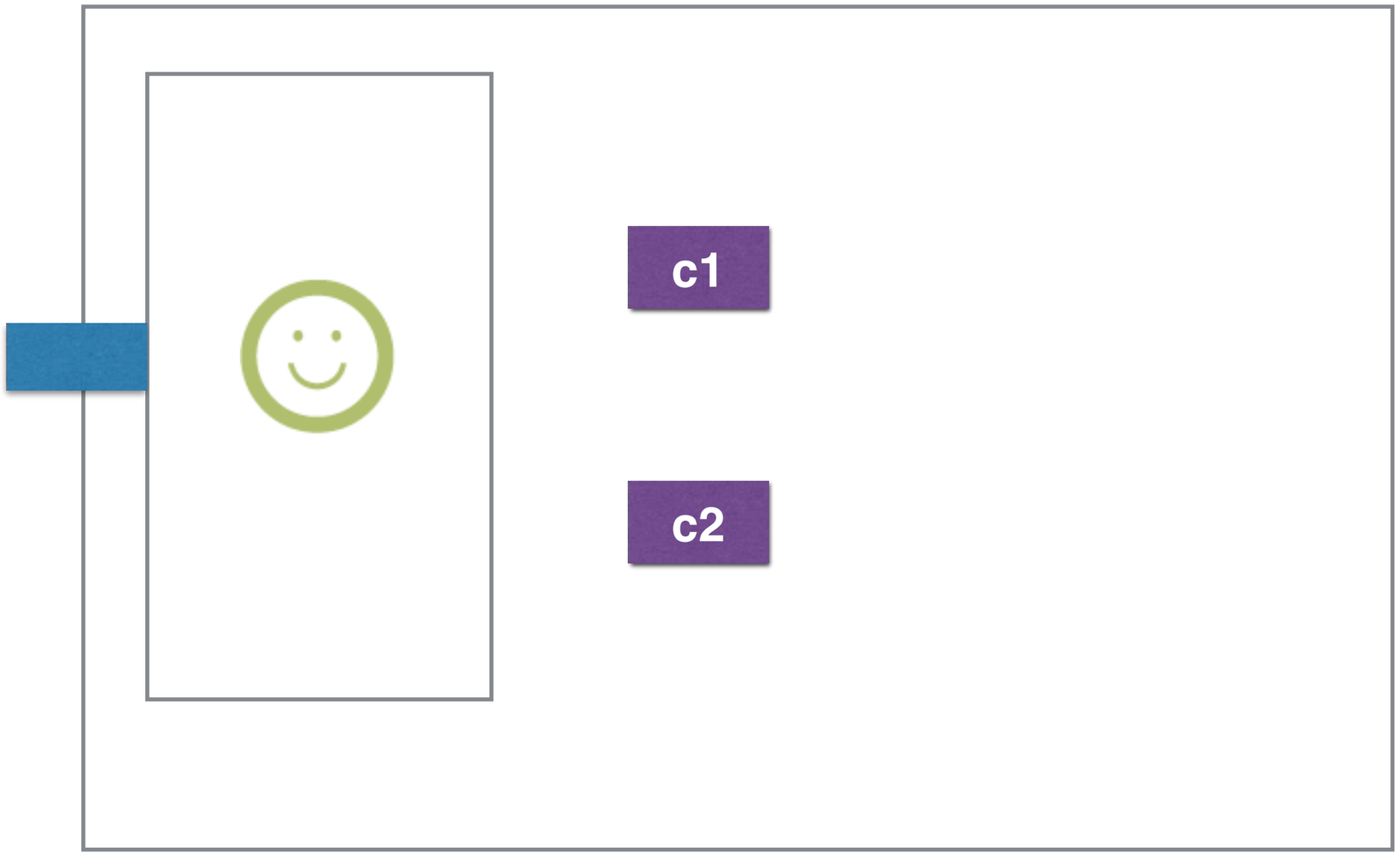
Gráficamente



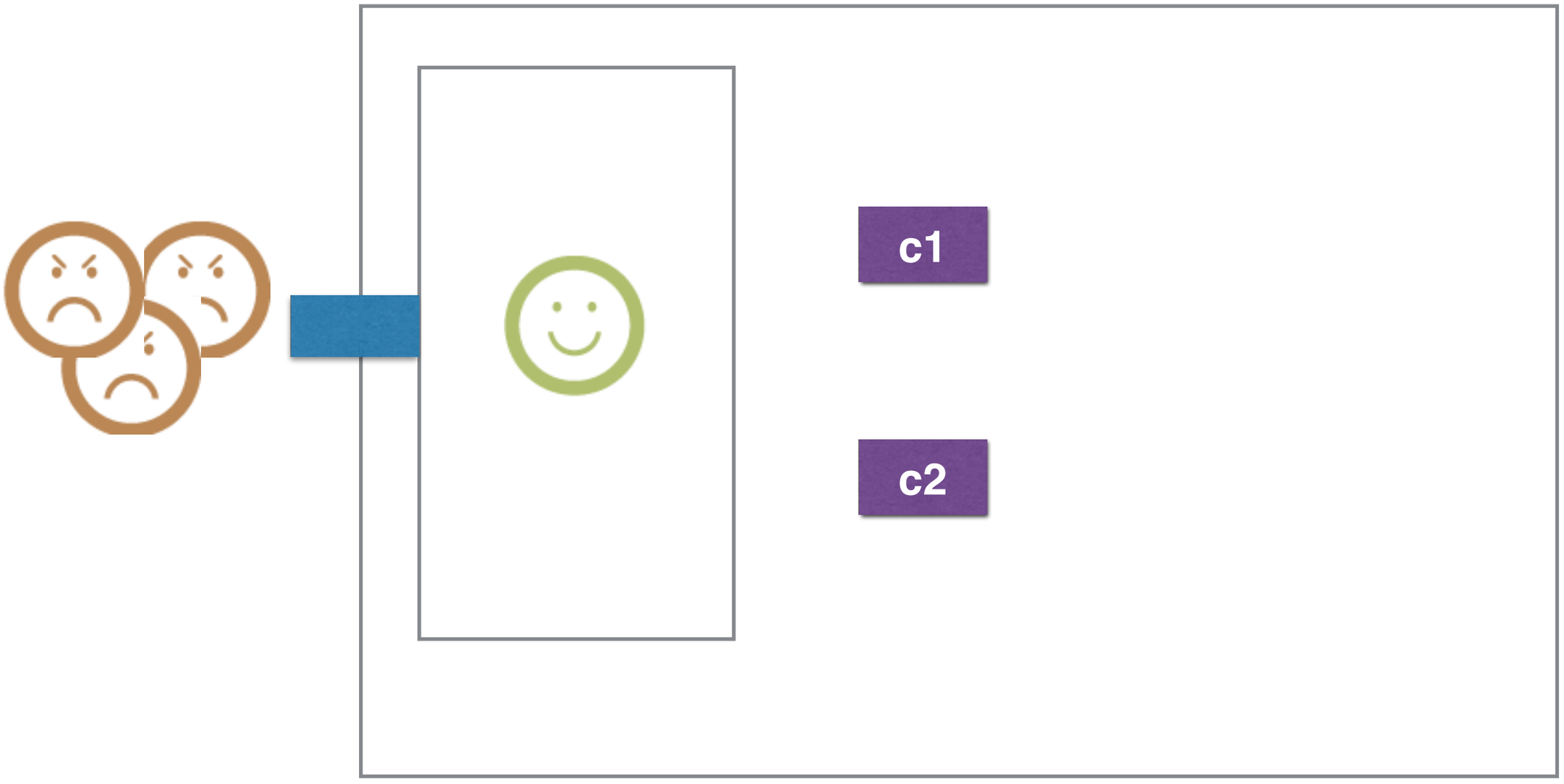
Gráficamente



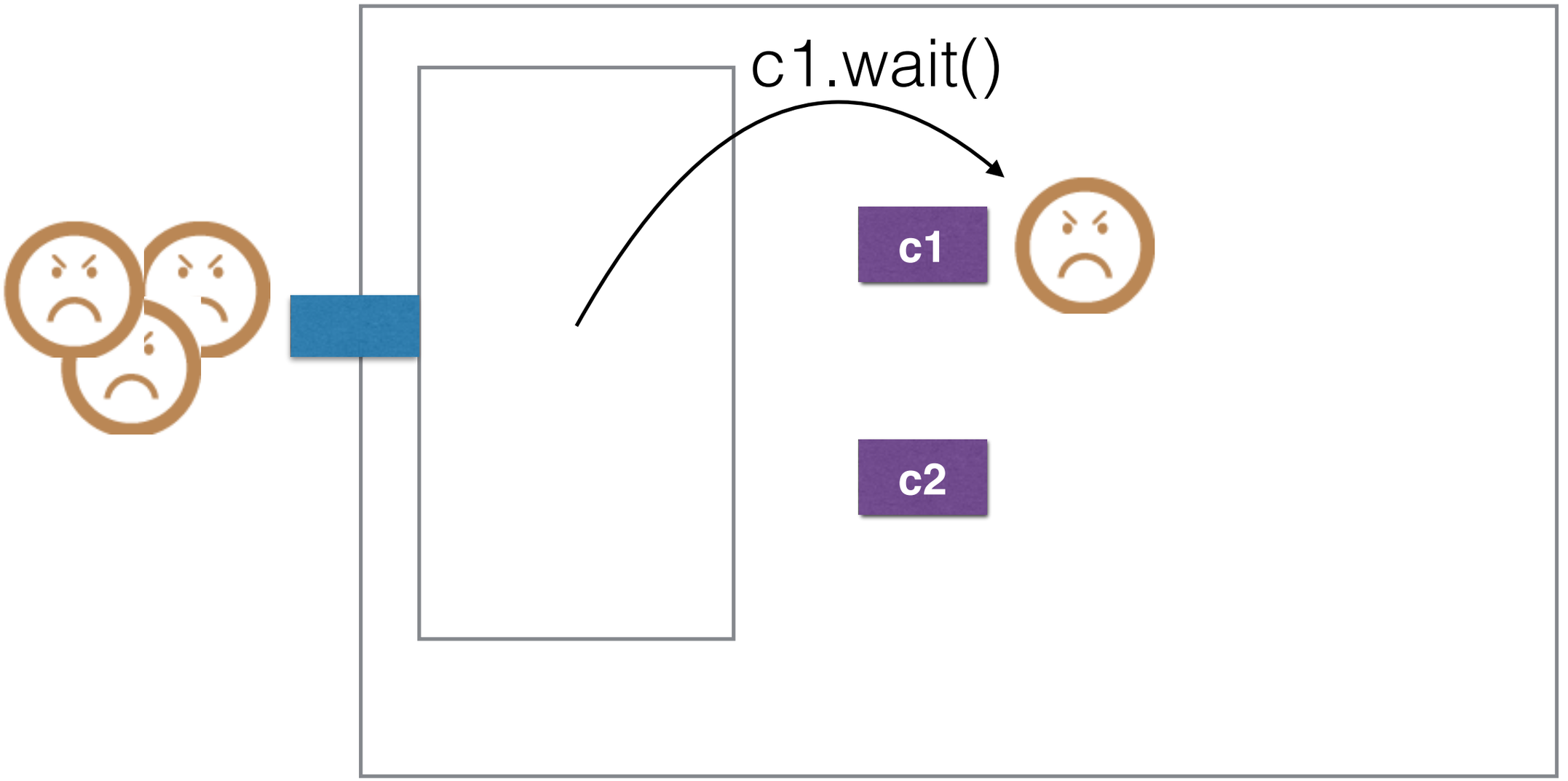
Gráficamente



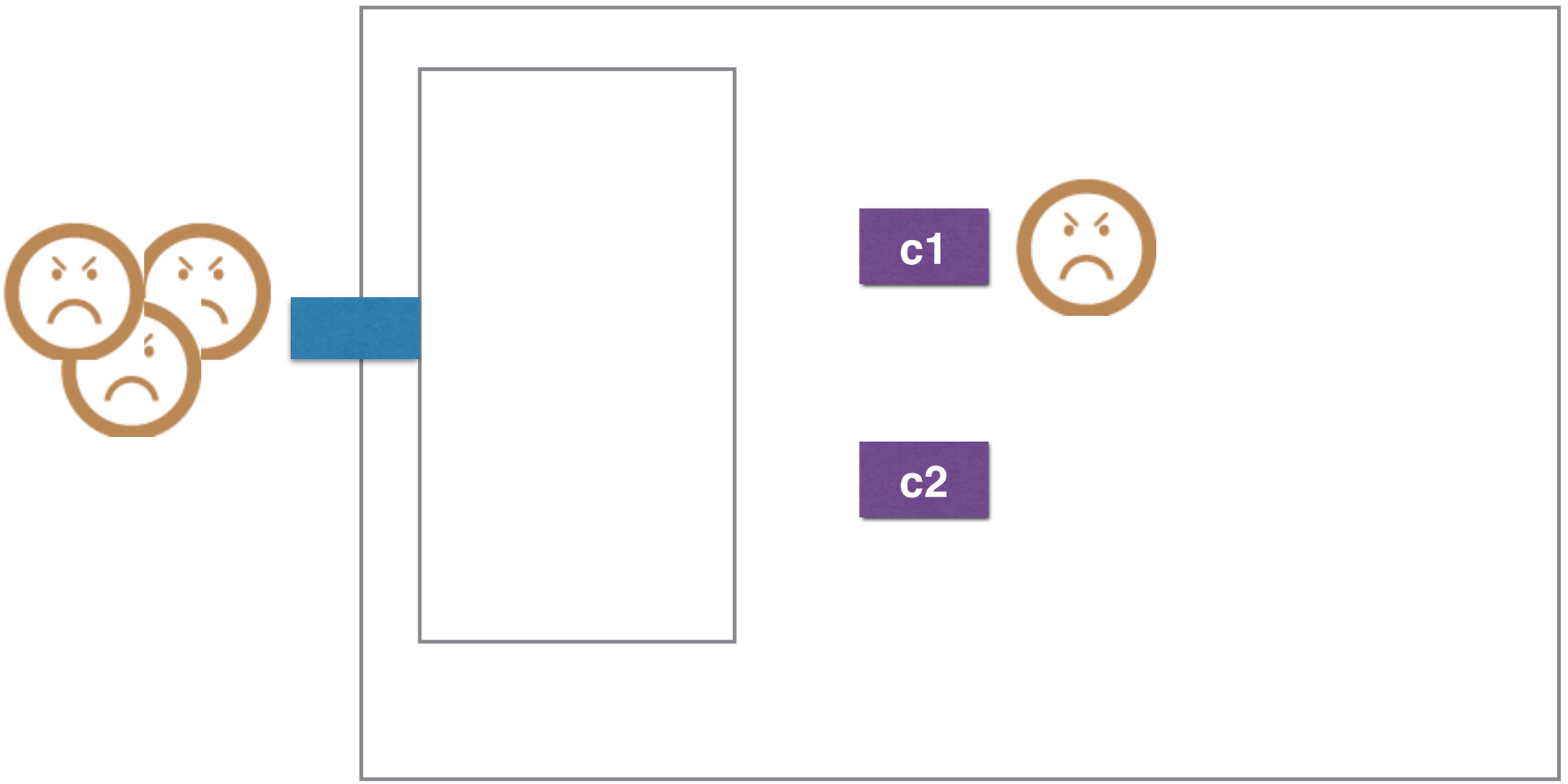
Gráficamente



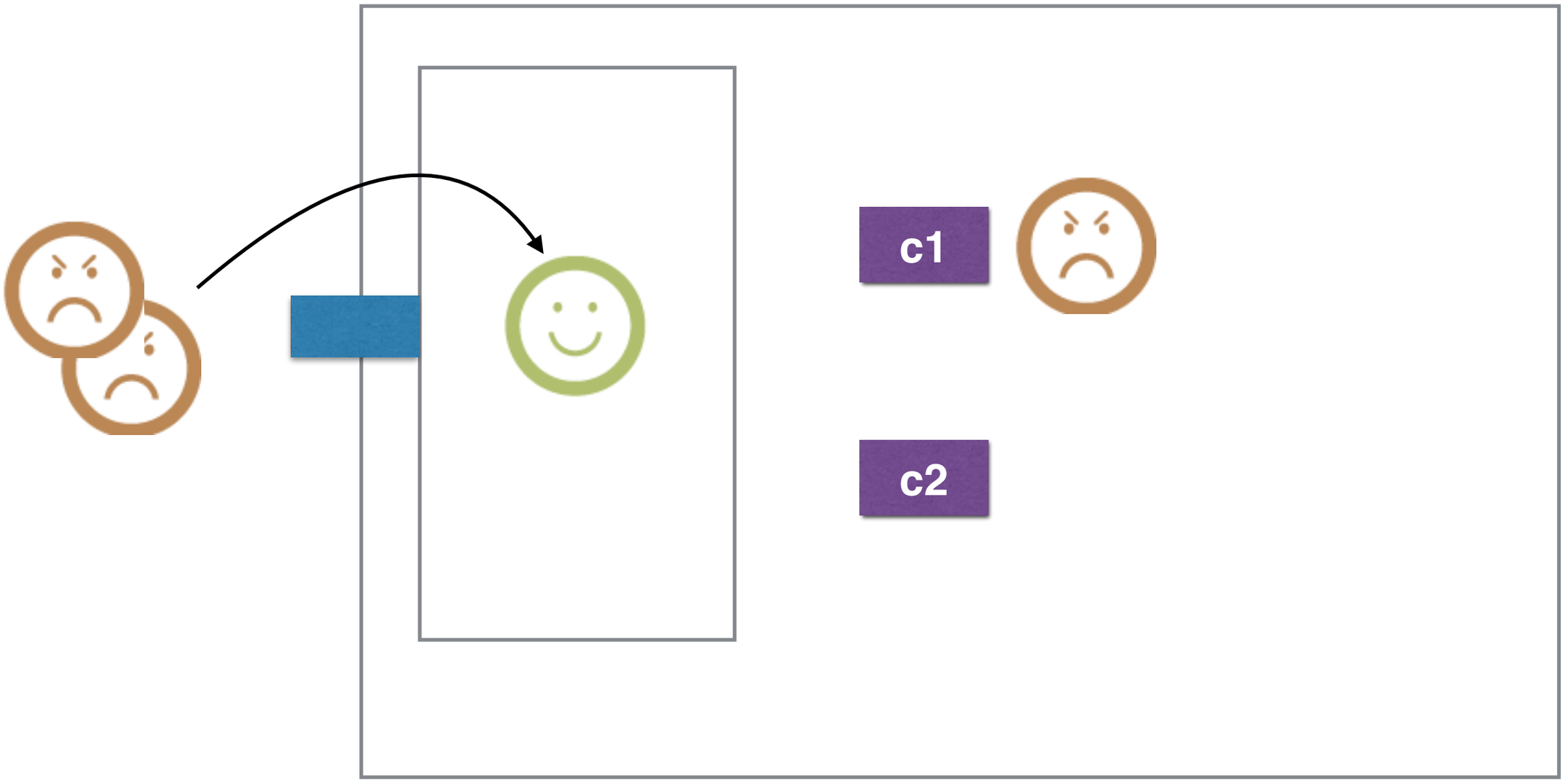
Gráficamente



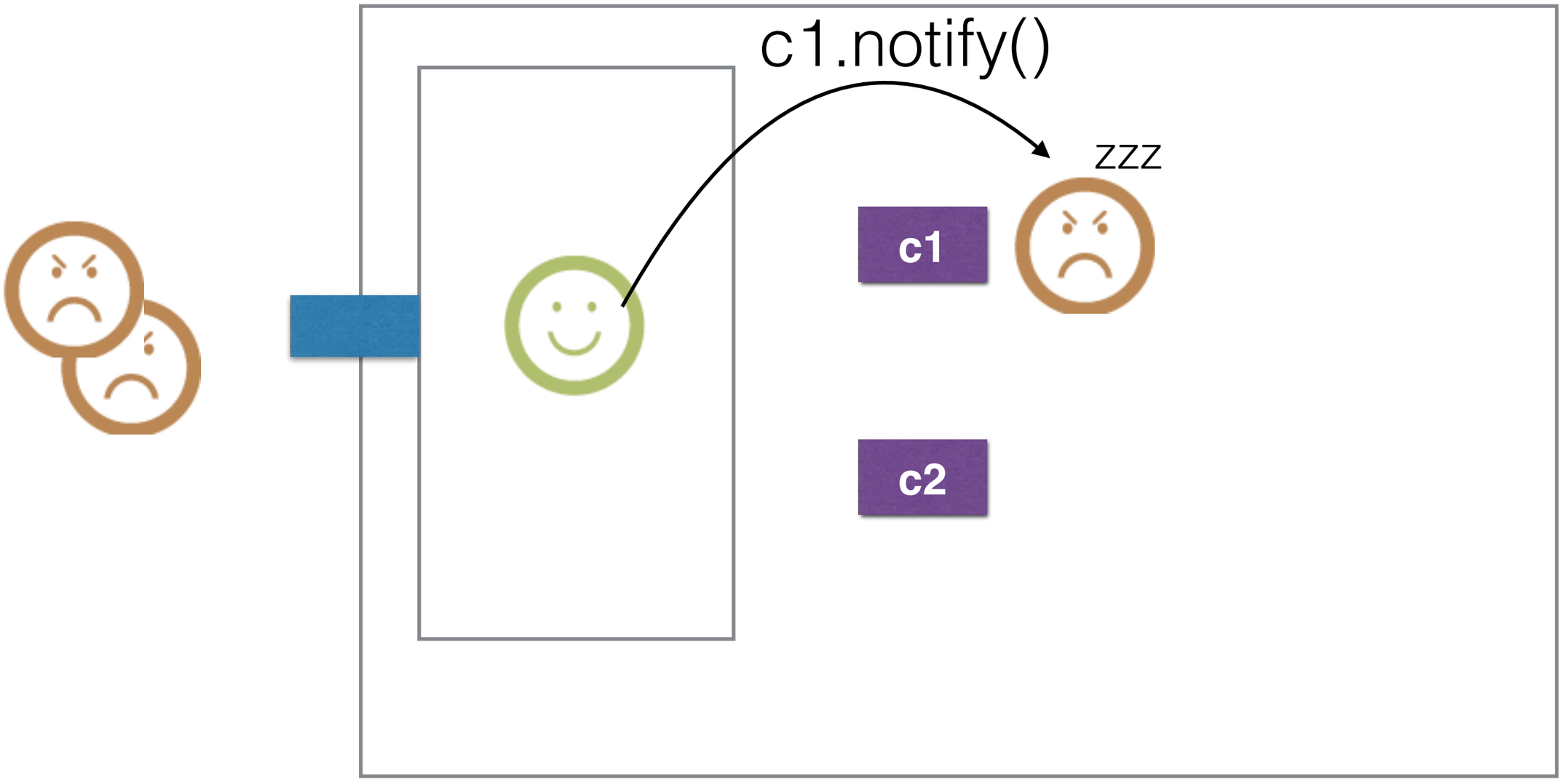
Gráficamente



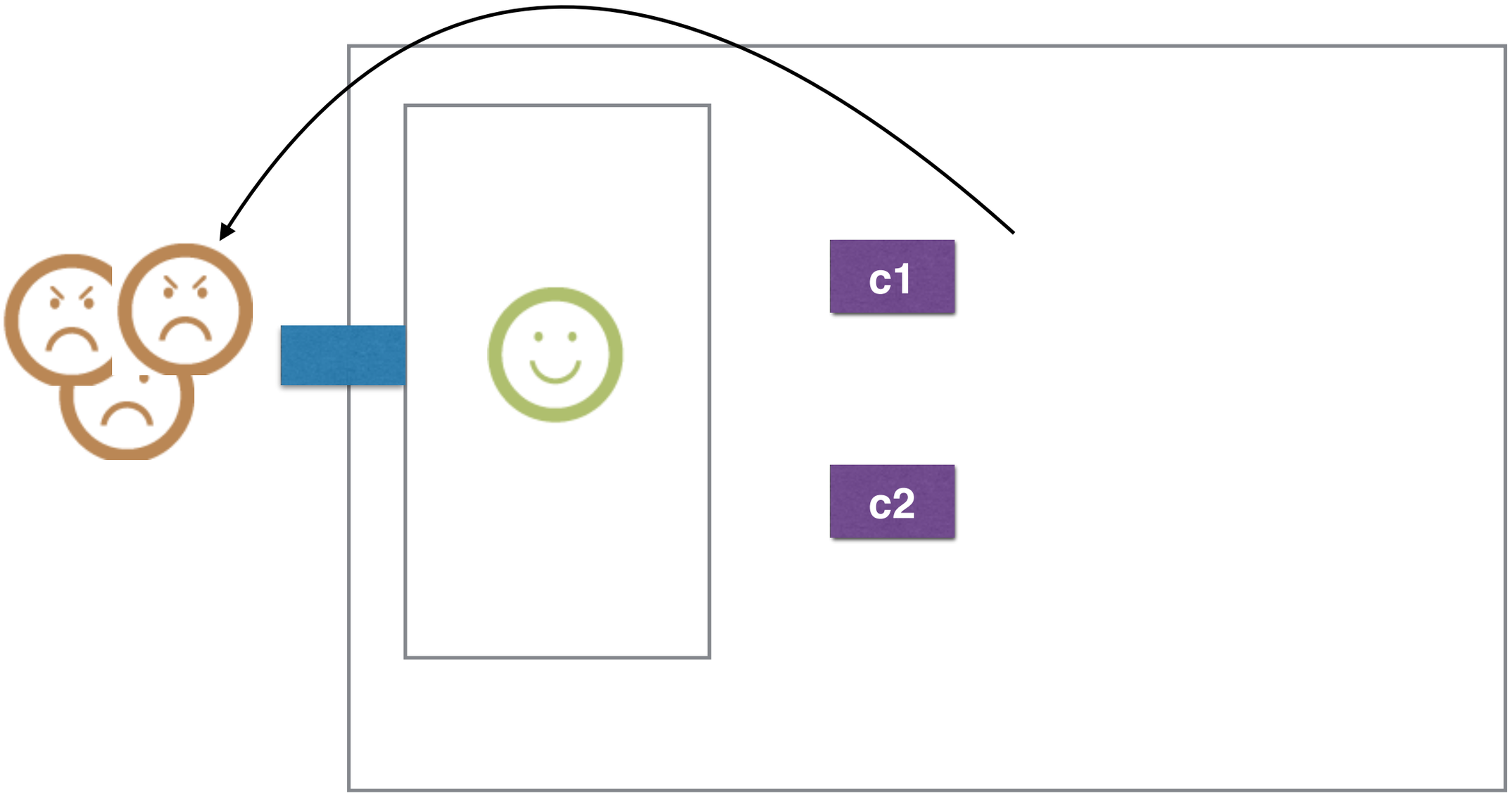
Gráficamente



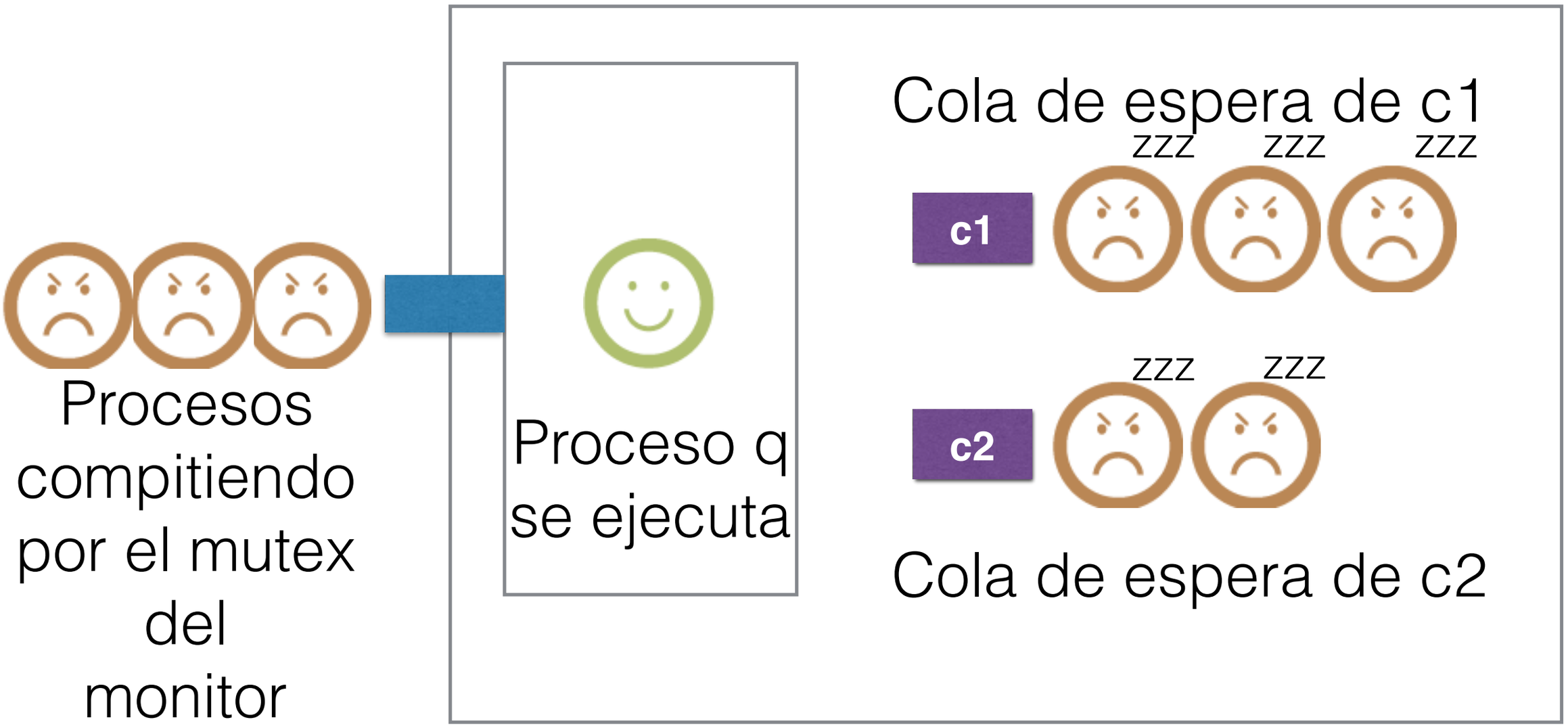
Gráficamente



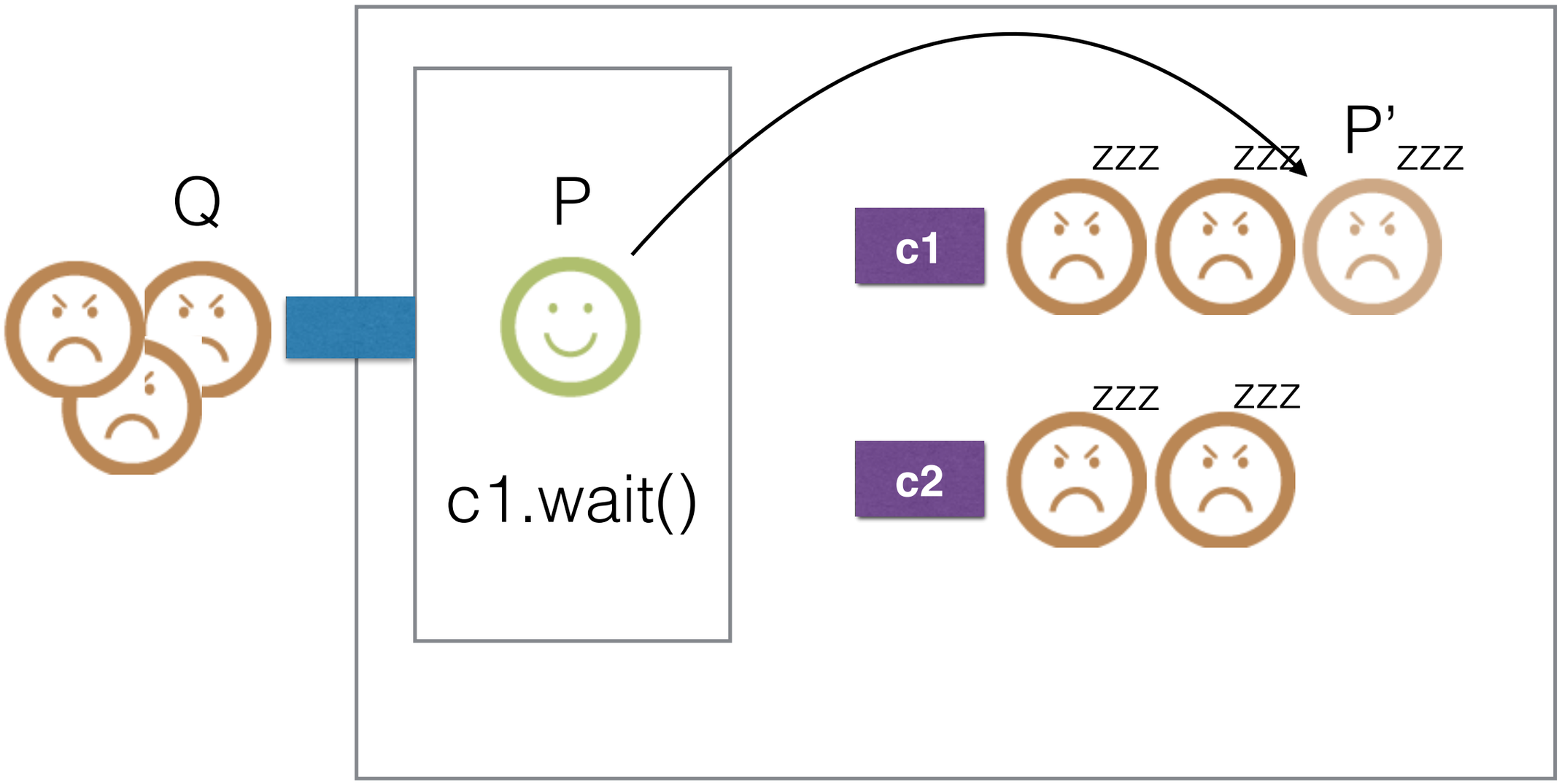
Gráficamente



Gráficamente

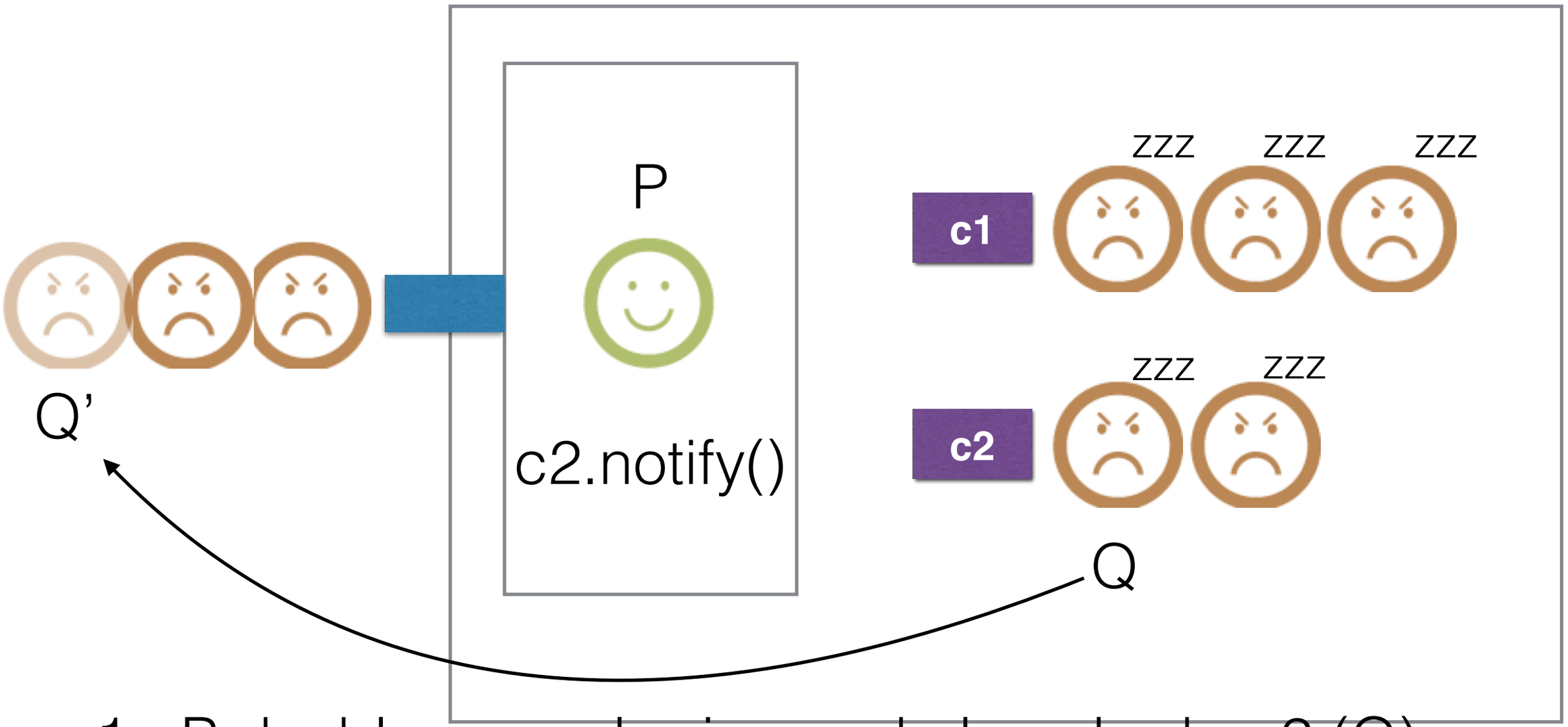


Gráficamente



1. P se coloca al final de la cola de espera de c1
2. Q comienza a ejecutar

Gráficamente



1. P desbloquea al primero de la cola de c2 (Q)
2. Q se coloca para adquirir el mutex para ejecutar
3. Al finalizar su ejecución, P suelta el mutex

Wait()

- El proceso que invoca al wait() sobre la variable c:
  - Se bloquea y se coloca en la cola de espera de la variable c.
  - Libera el lock (mutex) para que otro proceso pueda entrar a la sección crítica.

Notify()

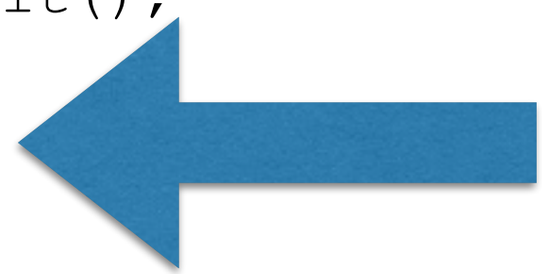
- Si el proceso P invoca notify() sobre la variable c:
  - Si hay procesos en la cola de espera de cond:
    - El proceso Q (el primero de la cola de espera de c) se desbloquea
    - Cuando el proceso P suelta el mutex del monitor, el proceso Q **compite** para tomarlo.
- Si no hay procesos en espera para cond, entonces el proceso P sigue ejecutando

Ejemplo Counter

- Modificar el contador para evitar que el valor se decremente cuando el contador es igual a 0.

Counter - Monitores

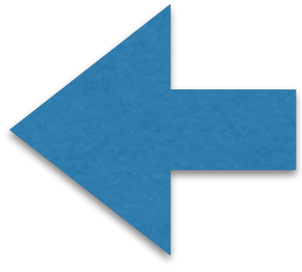
```
monitor Counter {

  private int c = 0;
  private condition notZero;

  public void inc() {
    c++;
    notZero.notify();
  }

  public void dec() {
    if (c==0) {
      notZero.wait();
    }
    c--;
  }
}
```



Otro proceso puede modificar la variable de condición cuando el proceso W obtiene el lock

Counter - Monitores

```
monitor Counter {

  private int c = 0;
  private condition notZero;

  public void inc() {
    c++;
    notZero.notify();
  }

  public void dec() {
    while (c==0) {
      notZero.wait();
    }
    c--;
  }
}
```



Siempre Chequeamos la condición antes de continuar

Ejemplo Buffer

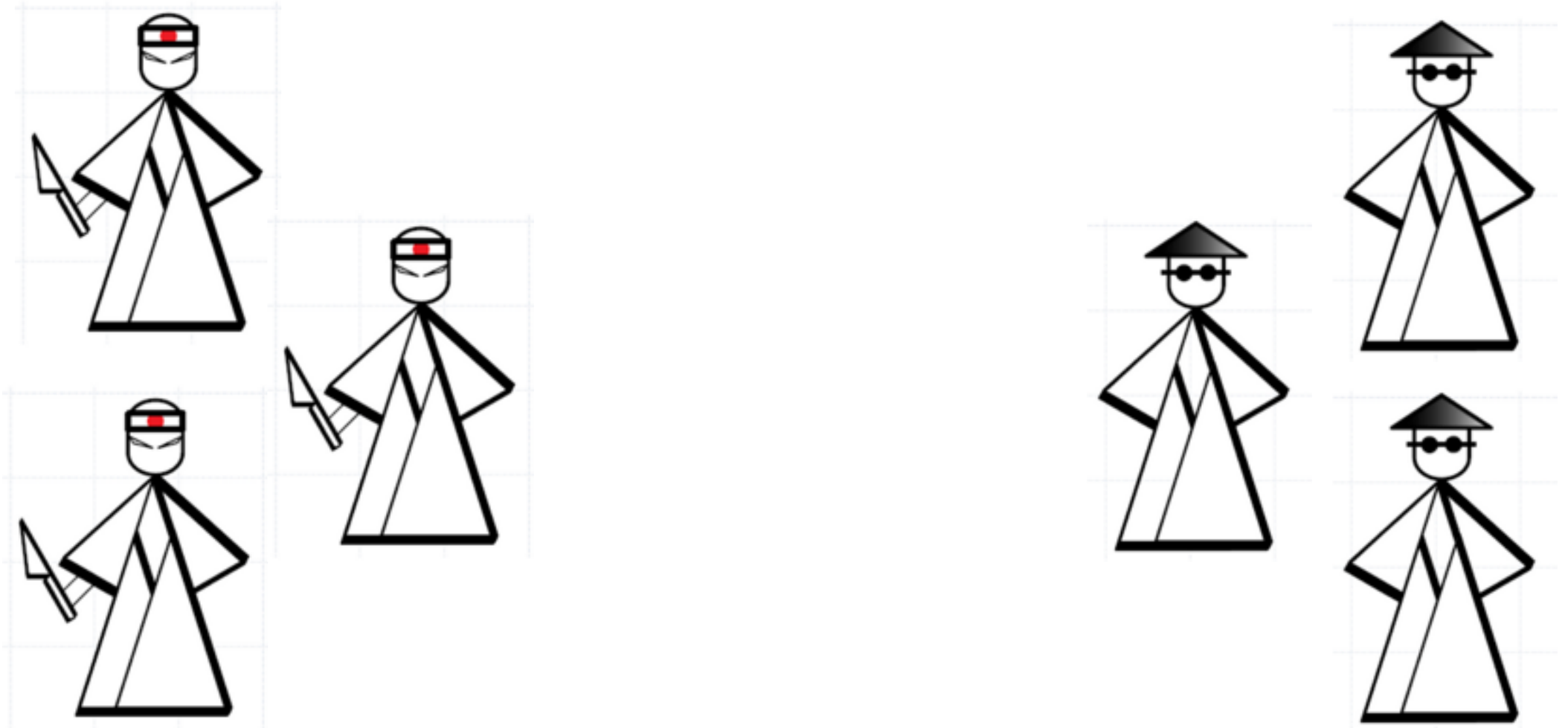
- **Crear** un monitor que sea un Buffer de capacidad 1 para con operaciones read() y write()

- El write() escribe el dato (sobre-escribe lo anterior)

- El read() lee el dato (y libera el dato que puede ser sobre-escrito)

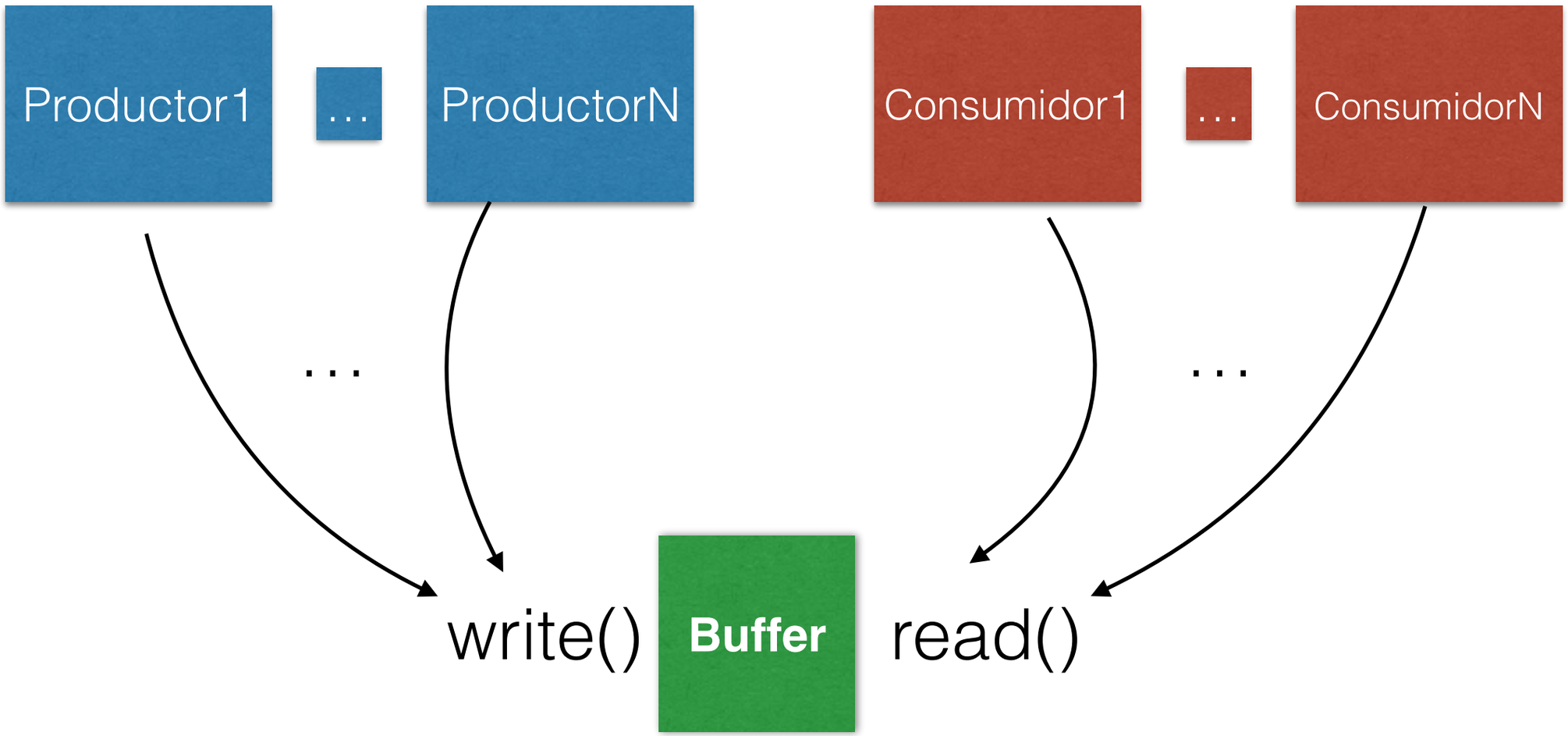
Buffer de Capacidad 1

```
monitor Buffer {

   private Object dato = null;

   public Object read() {
      Object aux = dato;
      dato = null;
      return aux;
   }

   public void write(Object o) {
      dato=o;
   }
}
```



Productor-Consumidor

Productor-Consumidor



Productor-Consumidor

- **Extender** el monitor para que el Buffer pueda ser usado en un esquema de Productor/Consumidor
  - El Productor no puede escribir si hay un dato
  - El Consumidor no puede leer si no hay dato

Buffer de Capacidad 1

```
monitor Buffer {
  private Object dato = null;
  private condition hayDato;
  private condition hayEspacio;

  public Object read() {
    while (dato==null) {
       hayDato.wait();
    }
    Object aux = dato;
    dato = null;
    hayEspacio.notify();
    return aux;
  }

  public void write(Object o)
    while (dato!=null) {
      hayEspacio.wait();
    }
    dato = o;
    hayDato.notify();
  }
```

Ejemplo - Buffer

- **Extender** el monitor para que a un Buffer de capacidad N que pueda ser usado en un esquema de Productor/Consumidor

  - El Productor no puede escribir si hay un dato

  - El Consumidor no puede leer si no hay dato

Buffer de Capacidad N

```
monitor Buffer {
  private Object[] datos = new Object[N];
  private int begin = 0;
  private int end = 0;
  private condition hayDato;
  private condition hayEspacio;

 public Object read() {
    while (isEmpty()) {
       hayDato.wait();
    }
    Object result = data[end];
    end = end+1%N;
    hayEspacio.notify();
    return result;
}
```

```
public void write(Object o) {
  while (isFull()) {
    hayEspacio.wait();
  }
  data[begin]=o;
  begin = begin+1 % N;
  hayDato.notify();
}
```

Monitores en Java

Monitores en Java

- Todo objeto tiene un lock y **una única variable** de condición.

- Los métodos *wait*, *notify* y *notifyAll* pertenecen a la interfaz de la clase Object

- La exclusión mutua del monitor se garantiza con el uso del keyword "*synchronize*" en cada método

IllegalMonitorStateException

- Sólo se puede invocar wait, notify y notifyAll desde métodos synchronize de la clase
- Sino, se emite una IllegalMonitorStateException

```
public Object m1() {
  this.wait();//IllegalMonitorStateException
}
public Object m2() {
  this.notify();//IllegalMonitorStateException
}
```

- Escribir un buffer de longitud N usando Monitores de Java (wait, notify, notifyAll)

```
class Buffer {
 private Object[] data = new Object[N];
 private int begin=0;
 private int end=0;
public void write(Object o) {
   data[begin]=o;
   begin = begin+1 % N;
 }
public Object read() {
   Object result = data[end];
   end = end+1%N
   return result;
 }
```

```
private boolean isEmpty() {
   return begin==end;
 }

private boolean isFull() {
   return (begin+1%N)==end;
 }
```

```
public synchronized void write(Object o) {
    while (isFull()) {
      this.wait();
    }
    data[begin]=o;
    begin = begin+1 % N;
    this.notifyAll();
}
```

```
public synchronized Object read() {
    while (isEmpty()) {
     this.wait();
    }
    Object result = data[end];
    end = end+1%N;
    this.notifyAll();
    return result;
}
```

Threads en Java

- Son procesos “livianos” (comparten espacio de memoria)

```
public class Productor extends Thread {

  private final Buffer buff;

  public Productor(Buffer b) {
    this.buff = b;
  }

  public void run() {
    int i=0;
    while (true) {
      buff.write(new Integer(i));
      i++;
    }
  }
}
```

Threads en Java

- Son procesos “livianos” (comparten espacio de memoria)

```
public class Consumidor extends Thread {

  private final Buffer buff;

  public Consumidor(Buffer b) {
    this.buff = b;
  }

  public void run() {
    int i=0;
    while (true) {
      Object obj = buff.read();
      System.out.println(“Objeto leido” + obj.toString());
    }
  }
}
```

Threads en Java

- Extender la clase `java.lang.Thread`
  - Sobre-escribir el método `run()`
- Crear un objeto de la subclase de Thread
  - Ejecutar el método `start()`
  - Opcionalmente, ejecutar el método `interrupt()`

Threads en Java

```
public class Main {

  public static void main(String[] args) {
    Buffer buff = new Buffer();
    Productor p = new Productor(buff);
    p.start();
    Consumidor c = new Consumidor(buff);
    c.start();
}
```

start() crea un nuevo thread de ejecución
e invoca al método “run”

InterruptedException

- El método wait() puede tirar la excepción `InterruptedException`

- Esto ocurre cuando a un thread en espera se lo interrumpe (`thread.interrupt()`)

- Aunque no ocurra en nuestro programa, debemos considerar el caso.

Lectores-Escritores

Lectores-Escritores

- Existen recursos compartidos por 2 clases de procesos:
    - **Lectores**: acceden al recurso sin modificarlo
        - pueden acceden al mismo tiempo para leer el recurso
    - **Escritores**: acceden al recurso y pueden modificarlo
        - como máximo puede haber un único proceso accediendo el recurso

Lectores-Escritores

- Dar una solución para acceso de lectores y escritores usando monitores

- La solución debe dar prioridad a la escritura

- Las operaciones a implementar son `beginRead()/endRead()` y `beginWrite()/endWrite()`

Lectores-Escritores

```
class Database {

 public void beginWrite() { // Escritores
 …
 }
public void endWrite() {
…
}
public void beginRead() { // Lectores
…
}
public void endRead() {
…
}
```

Solución Lectores-Escritores

```
class Database {
 private int writers=0;
 private int readers=0;
 private int waitingWriters=0;
private int waitingReaders=0;

public boolean canWrite() {
   return writers==0 && readers==0;
}
public boolean canRead() {
   return writers==0 && waitingWriters==0;
}
```

Solución Lectores-Escritores (cont.)

```
public void beginWrite() {
    while (!canWrite()) {
       waitingWriters++;
       this.wait();
       waitingWriters--;
     }
   writers = 1;
}
public boolean endWrite() {
    writers = 0;
    if (waitingReaders > 0 || waitingWriters > 0) {
       this.notifyAll();
    }
}
```

Solución Lectores-Escritores (cont.)

```
public void beginRead() {
    while (!canRead()) {
       waitingReaders++;
       this.wait();
       waitingReaders--;
     }
   readers++;
}
public boolean endRead() {
    readers--;
    if (waitingReaders > 0 || waitingWriters > 0) {
       this.notifyAll();
    }
}
```

Solución Lectores-Escritores (cont.)

```
public class Reader extends Thread {
    public Reader(Database db) {
      this.db = db;
    }
    public void run() {
      while (true) {
        db.beginRead();
        // read
        db.endRead();
      }
}
```

Solución Lectores-Escritores (cont.)

```
public class Writer extends Thread {
    public Writer(Database db) {
      this.db = db;
    }
    public void run() {
      while (true) {
        db.beginWrite();
        // write
        db.endWrite();
      }
}
```

Solución Lectores-Escritores (2 Writers+100 Readers)

```
public class Main {
    public static void main(String[] arg) {
        Database db = new Database();
        Writer w0 = new Writer(db);
        w0.start();
        Writer w1 = new Writer(db);
        w1.start();
        for (int i =0 ; i<100; i++) {
          Reader r = new Reader();
          r.start();
        }
}
```

InterruptedException

```
public void beginWrite() {
    while (!canWrite()) {
      waitingWriters++;
      try {
        this.wait();
      } catch (InterruptedException ex) {
        waitingWriters--;
        return;
      }
      waitingWriters--;
    }
  writers = 1;
}
```

InterruptedException (cont.)

```
public void beginRead() {
    while (!canRead()) {
      waitingReaders++;
      this.wait();
    } catch (InterruptedException ex) {
      waitingReaders--;
      return;
    }
    waitingReaders--;
  }
  readers++;
}
```

Monitores

- Puede haber deadlock?
- Puede haber starvation?

Recap

- Los Monitores son un mecanismo de más alto nivel que Semáforos para sincronizar programas concurrentes

- Vimos dos sabores de Monitores:
    - Monitores Conceptuales (teóricos)
    - Monitores Java
- Productor/Consumidor
- Lectores/Escritores (con prioridad Escritores)