Monitor Object

Concurrency Pattern

Huib van den Brink

Center for Software Technology, Utrecht University
http://www.cs.uu.nl/groups/ST/

October 14, 2005

Outline

1. Concurrency problems
   - Problems involved
   - Active Object
2. Monitor Object
   - The pattern
   - Metaphor
   - Synchronized implementation example
   - Lock on parts of method / among several objects
   - Introduced problems
   - Performance issues
   - Pros & Cons
3. Finally
   - Questions
   - Excercise

Computers aren't woman

Concurrency problems

1. Handle multiple requests simultaneously
2. Modify state of objects
3. Need for control of atomic actions
4. Regulate and schedule access to objects

Active Object pattern

The Active Object pattern could do the trick, but...

Not always suitable because

1. More complicated then necessary
2. Request and execution both in separate thread
3. The ability of decoupling synchronization is not always needed
4. Scheduling and registration (of Activation List) only slows it down
5. Doesn't locate the synchronization closely to functionality

Monitor Object pattern

Monitor Object pattern, Thread-safe Passive Object or Code Locking pattern

Design

1. Create an object containing the unsafe implementation (Synchronized Method)

Monitor Object pattern

Monitor Object pattern, Thread-safe Passive Object or Code Locking pattern

Design

1. Create an object containing the unsafe implementation (Synchronized Method)
2. Create an object responsible for synchronization (Monitor Object)

Monitor Object pattern

Monitor Object pattern, Thread-safe Passive Object or Code Locking pattern

Design

1. Create an object containing the unsafe implementation (Synchronized Method)
2. Create an object responsible for synchronization (Monitor Object)
3. Assign a Lock object to the Monitor Object (Monitor Lock)

Monitor Object pattern

Monitor Object pattern, Thread-safe Passive Object or Code Locking pattern

Design

1. Create an object containing the unsafe implementation (Synchronized Method)
2. Create an object responsible for synchronization (Monitor Object)
3. Assign a Lock object to the Monitor Object (Monitor Lock)
4. Provide the implementation means to suspend and resume (Monitor Condition)

Monitor Object pattern

Monitor Object pattern, Thread-safe Passive Object or Code Locking pattern

Design

1. Create an object containing the unsafe implementation (Synchronized Method)
2. Create an object responsible for synchronization (Monitor Object)
3. Assign a Lock object to the Monitor Object (Monitor Lock)
4. Provide the implementation means to suspend and resume (Monitor Condition)
5. N clients can invoke now simultaneously

Means provided by the pattern

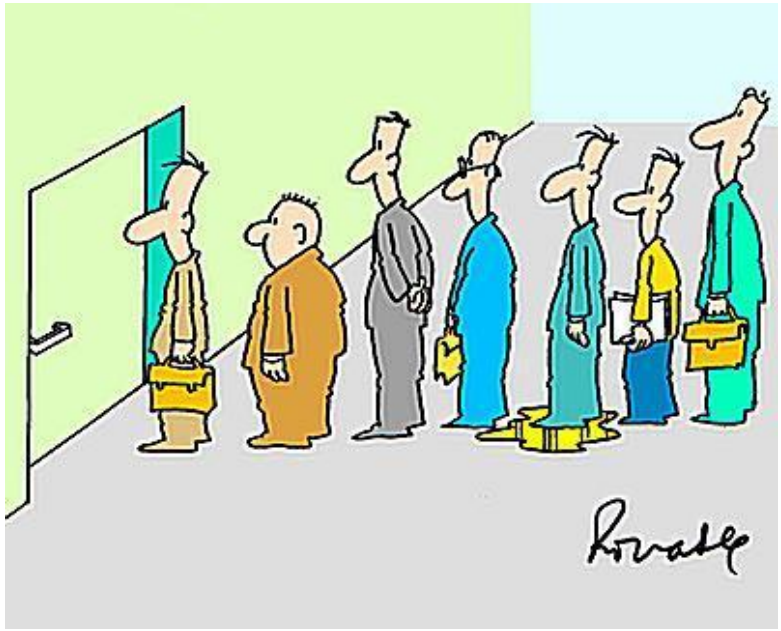
1. Only one synchronized method at a time running within an object
2. Separation of the low level lock acquirement and the synchronized implementation
3. Ability to suspend and resume execution within an method
4. Tightly coupled locking mechanism for increased performance

1: Free toilet

2: Toilet taken and locked

Separation of toilet and lock

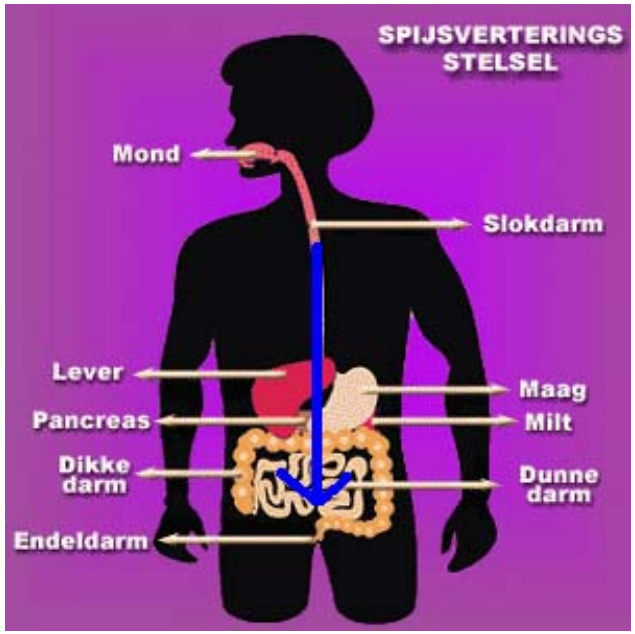
3: Next people waiting in order of arival



4: People unlock and leave

Enabling next waiting to go

5: People who can't deliver yet

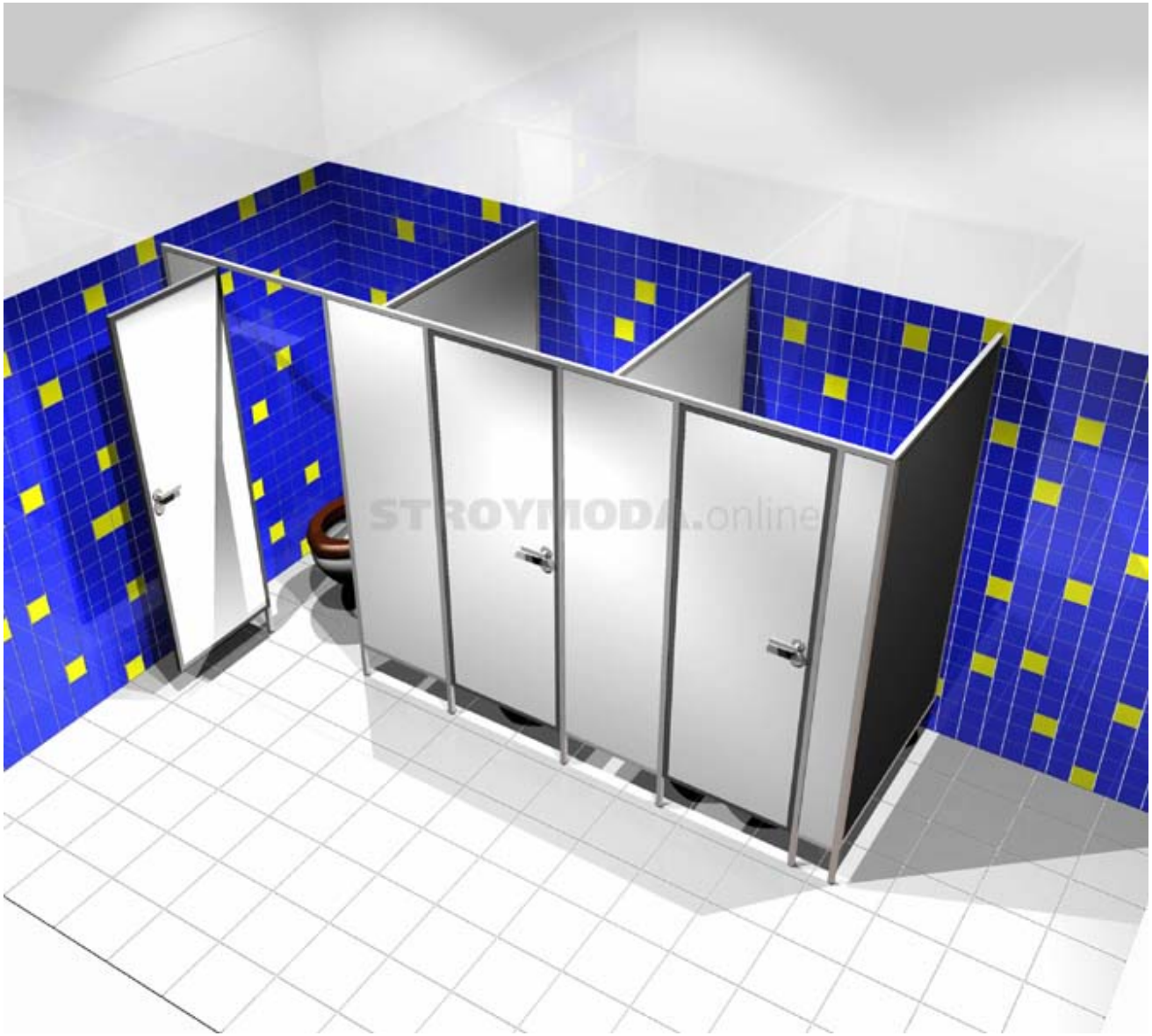


Don't hold the lock on constipation

6: People want to continue

Just continue when you are feeling ready (notified by digestion)

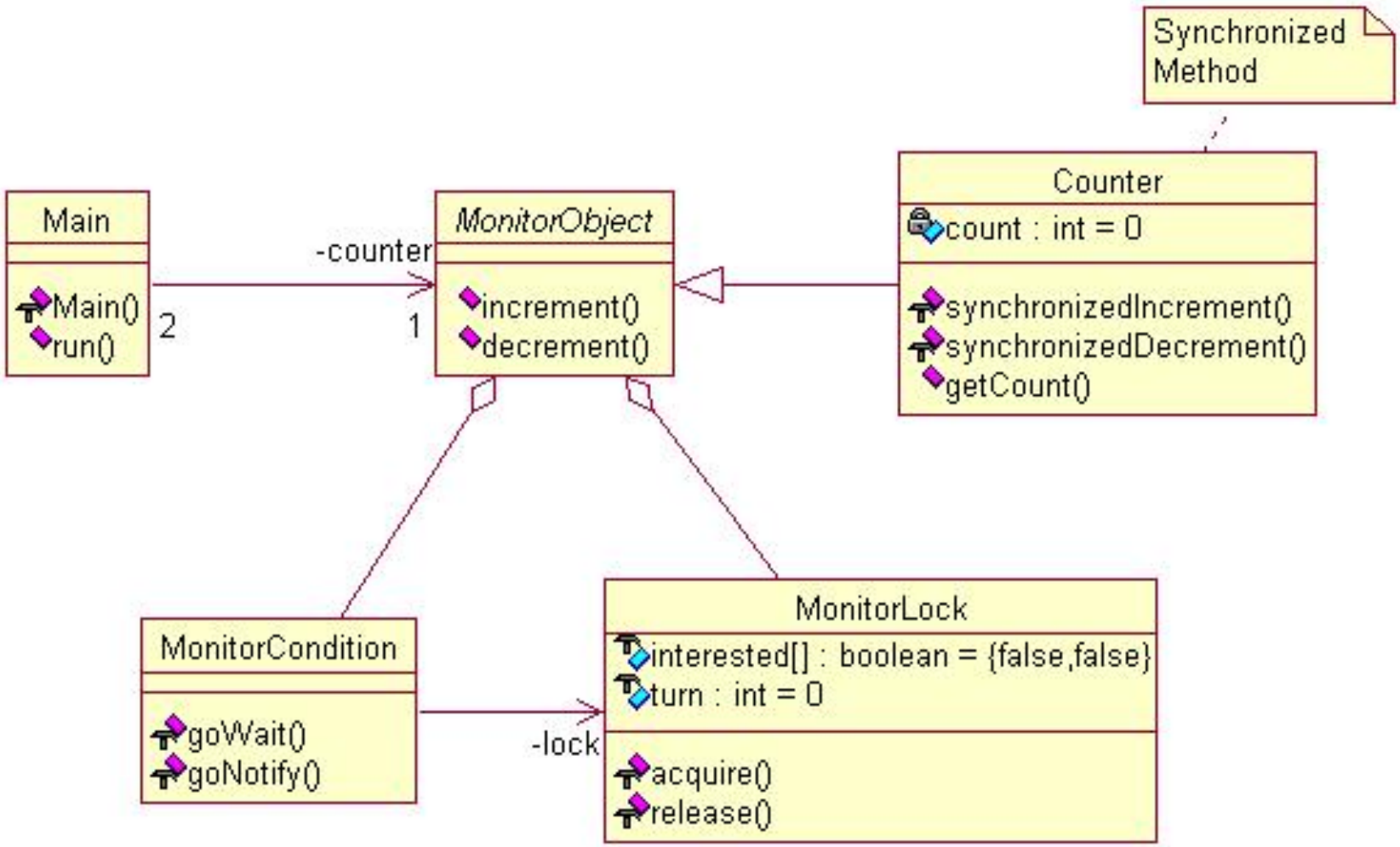
7: Multiple objects with each their own lock



Pitfalls

1. Not calling wait() in stable state
   Pull your pants up first !
2. Forget to release lock when an exception occurs
   Children panic when they can't open the door
3. Not making the method synchronized when needed
   Don't forget to lock your toilet door !

Example: synchronized implementation (1)



Example: synchronized implementation (2)

```
// Synchronized Method
class Counter extends MonitorObject
{
    private int count = 0;

    void synchronizedIncrement(int threadId)
    {
        //wait(threadId);     // will hold after a while, because

        count++;
        System.out.println("Thread " + threadId + ": " + count);
    }

    void synchronizedDecrement(int threadId)
    {                           // not every notify has an increment
        //notify(threadId);   // waiting

        int current = count;
        current--;              // buffer with latency
        try{ Thread.currentThread().sleep(10); } catch(Exception
        count = current;
        System.out.println("Thread " + threadId + ": " + count);
    }

    public int getCount(){ return count; }
}
```

Example: synchronized implementation (3)

```
public abstract class MonitorObject
{
    private MonitorLock      lock      = new MonitorLock();
    private MonitorCondition condition = new MonitorCondition(lock);

    public void increment(int threadId)
    {
        lock.acquire(threadId);

        synchronizedIncrement(threadId);

        lock.release(threadId);
    }

    public void decrement(int threadId)
    {
        lock.acquire(threadId);
        synchronizedDecrement(threadId);
        lock.release(threadId);
    }

    void wait  (int threadId) { condition.goWait(threadId);   }
    void notify(int threadId) { condition.goNotify(threadId); }

    abstract void synchronizedIncrement(int threadId);
    abstract void synchronizedDecrement(int threadId);
    public abstract int getCount();
}
```

Example: synchronized implementation (4)

```
class MonitorCondition implements Runnable
{
    private MonitorLock lock;
    private volatile int threadId;

    MonitorCondition(MonitorLock lock)
    {
        this.lock = lock;
    }

    void goWait(int threadId)
    {
        lock.release(threadId);

        synchronized(this)
        {
            try{ wait(); } catch(Exception e){}
        }
    }

    void goNotify(int threadId)
    {                                   // not thread safe, so
        this.threadId = threadId;       // only suited for 2 threads
        new Thread(this).start();       // non-blocking request of lock
    }                                   // (dead-lock otherwise)

    public void run()
    {
        lock.acquire(1 - threadId);
        synchronized(this){ notify(); }
    }
}
```

Example: synchronized implementation (5)

```
// Dekker (1965)
class MonitorLock
{                                       // both not yet interested in the lock
    volatile boolean[] interested = { false, false };
    volatile int turn = 0;              // may also be 1

    void acquire(int id)                // thread id = 0 or 1
    {
        int other = 1 - id;
        interested[id] = true;          // show interest of taking the lock

        while(interested[other]) // while the other one also is interested
        {
            // if the other thread has got the turn
            if(turn == other)
            {
                // give up interest as long as the other has its turn:
                interested[id] = false;
                while(turn == other)
                    continue; // busy waiting

                // now he has lost the turn, say you're interested
                interested[id] = true;
            }
        }
    }

    void release(int id)
    {
        interested[id] = false;         // finished, so not interested in the lock
        turn = 1 - id;                  // give the other thread the turn
    }
}
```

Sharing a lock among several objects

1. Lock only when executing a part within a certain method
2. Expand the locking to affect several objects instead of just one

Locking less than a whole method

In the counter example

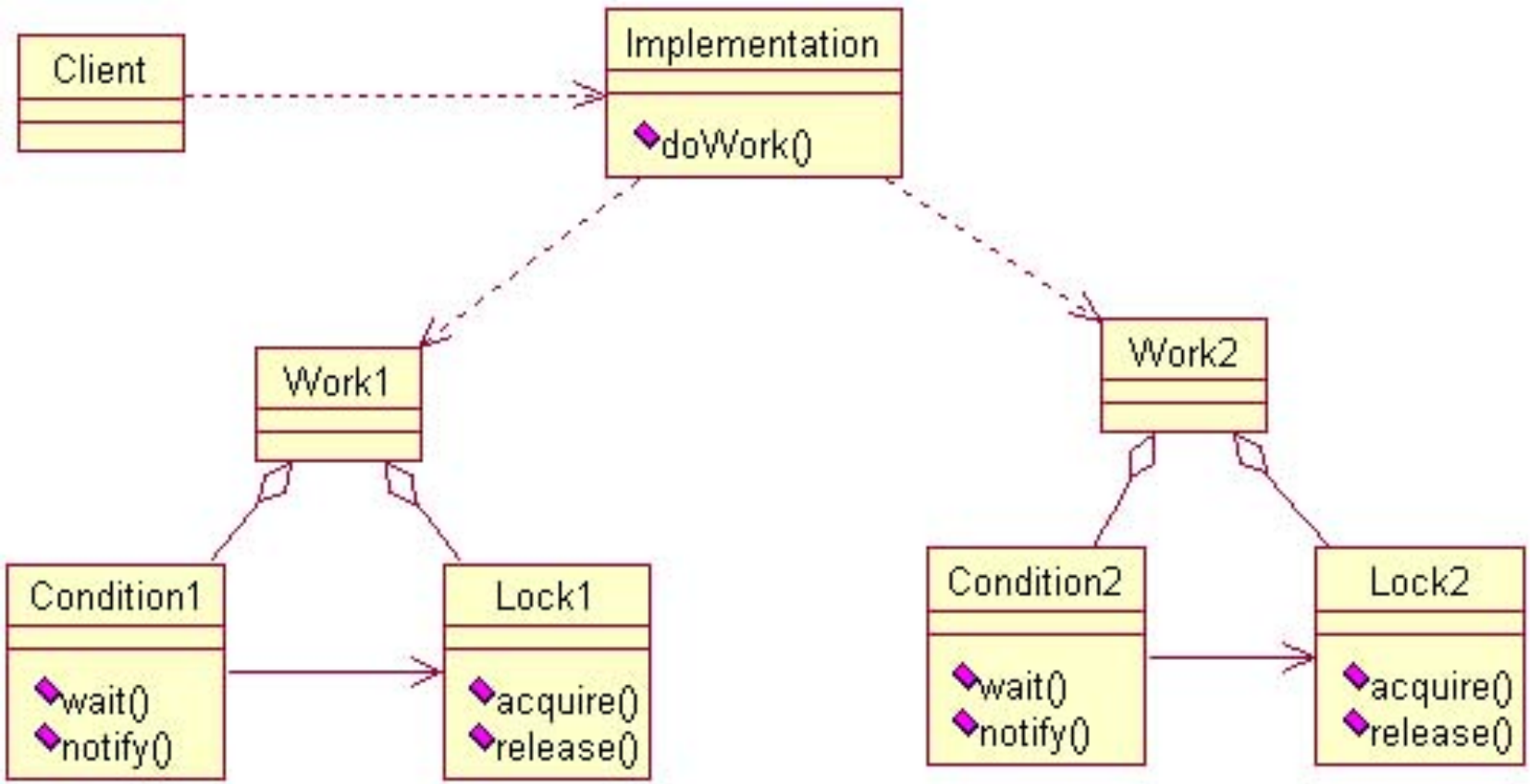
```
void synchronizedDecrement(int threadId)
{
    //...

    lock.acquire(threadId);

    int current = count;
    current--;
    try{ Thread.currentThread().sleep(10); } catch(Exception e){}
    count = current;

    lock.release(threadId);

    // ...
}
```

Ugly and losing abstraction

The Java way:

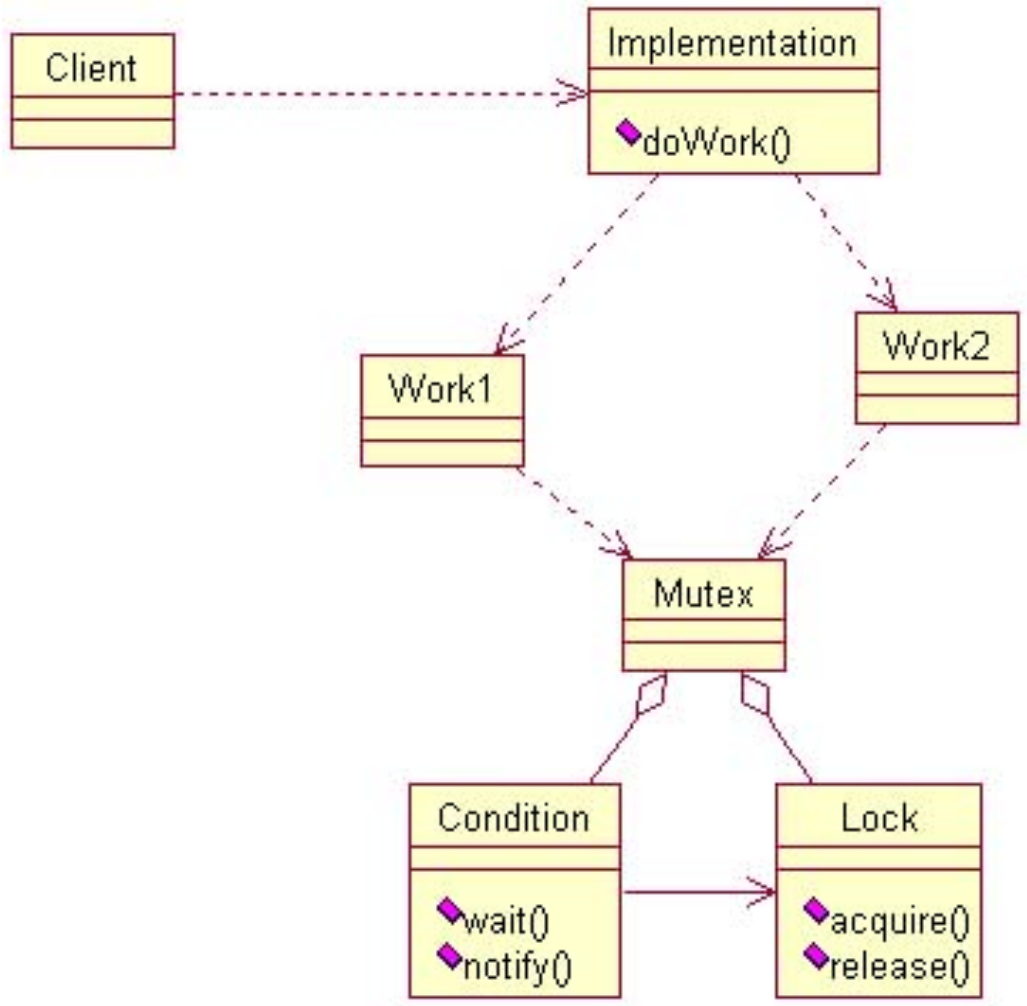
```
void partly()
{
    synchronized(this)
    {
        //...
    }
}
```

Each object it's own lock



Worries of deadlock

All objects sharing lock



Mutex provides extra influences

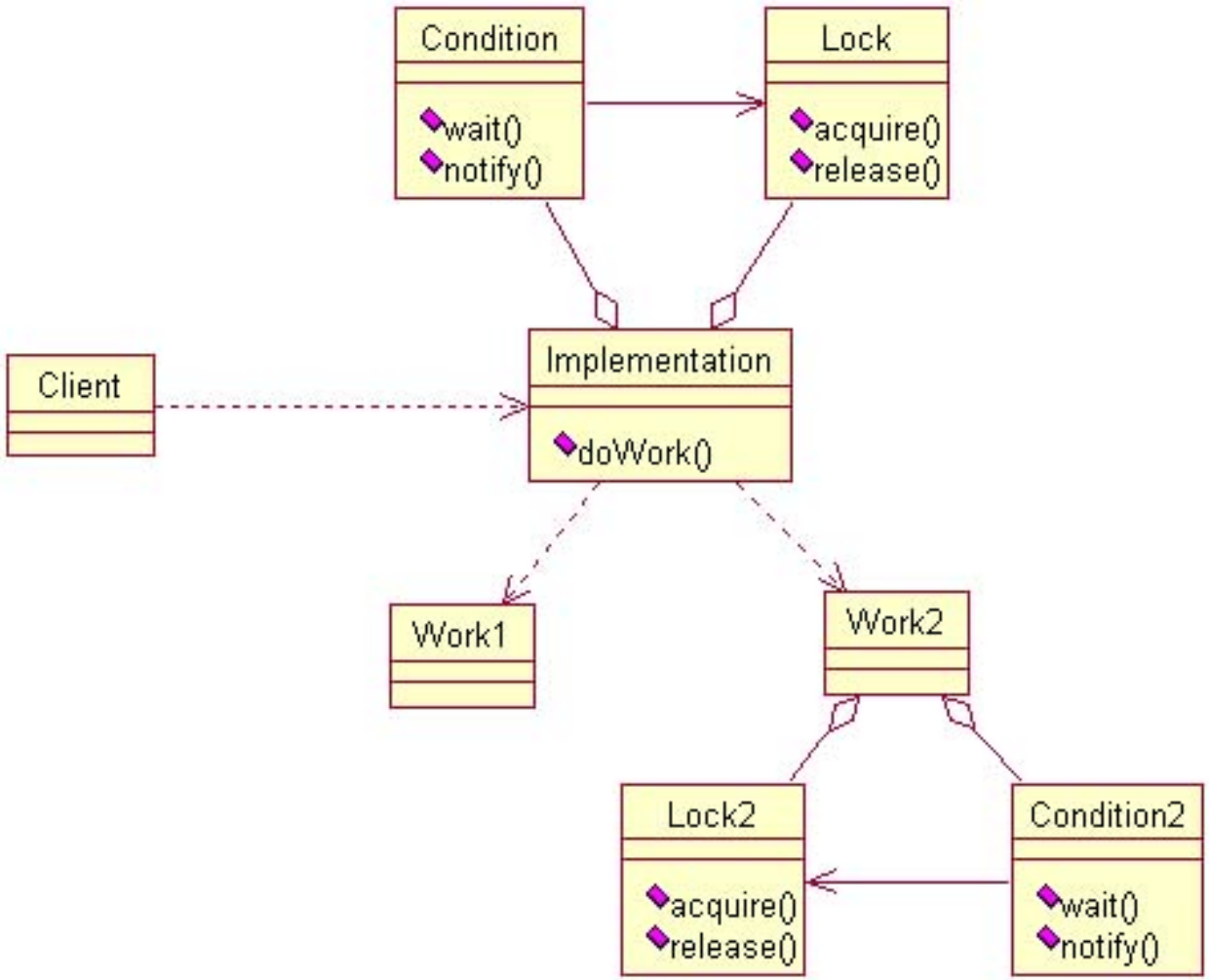
A mutex using the Monitor Object pattern

```
public class Mutex
{
    //acquired == true when this Mutex is 'given away' to one thread
    volatile boolean acquired = false;
    Thread thread = null;

    public synchronized void acquire()
    {
        // while a thread has to wait
        while(acquired)
        {
            try
            {
                // if the thread doens't already have the access
                if(this.thread != Thread.currentThread())
                    wait(); // let him wait
                else
                    break;
            }
            catch(Exception e){}
        }

        // let the other threads wait
        this.thread = Thread.currentThread();
        acquired = true;
    }

    //...
```

A mutex using the Monitor Object pattern cont'd

```
    //...

    public synchronized void release()
    {
        // only the thread that called the acquire can
        // release the lock
        if (acquired && this.thread == Thread.currentThread())
        {
            // wake other waiting threads up
            thread = null;
            acquired = false;
            notify();
        }
    }
}
```

The pattern doesn't always simplify it



New problems created

1. Nested monitor acquirements
2. Inherited methods not automatically synchronized
3. Starvation while it shouldn't
4. Deadlocks

Deadlock avoidance

1. Don't call synchronized methods in synchronized bodies when not strictly necessary

Deadlock avoidance

1. Don't call synchronized methods in synchronized bodies when not strictly necessary
2. Using timout mechanisms releasing held locks combined with exceptions
   (dangerous for timeouts also can occur due to e.g. network failure)

Deadlock avoidance

1. Don't call synchronized methods in synchronized bodies when not strictly necessary
2. Using timout mechanisms releasing held locks combined with exceptions
   (dangerous for timeouts also can occur due to e.g. network failure)
3. Only use one lock for entire system
   (only usefull if performance really isn't a matter)

Deadlock avoidance

1. Don't call synchronized methods in synchronized bodies when not strictly necessary
2. Using timout mechanisms releasing held locks combined with exceptions
   (dangerous for timeouts also can occur due to e.g. network failure)
3. Only use one lock for entire system
   (only usefull if performance really isn't a matter)
4. Always acquire locks in the same order
   (not very reliable though !)

Deadlock avoidance

1. Don't call synchronized methods in synchronized bodies when not strictly necessary
2. Using timout mechanisms releasing held locks combined with exceptions
   (dangerous for timeouts also can occur due to e.g. network failure)
3. Only use one lock for entire system
   (only usefull if performance really isn't a matter)
4. Always acquire locks in the same order
   (not very reliable though !)
5. Keep registration of:
   Which locks currently acquired by which method
   Which locks needed for the method to execute
   (and then implement canThreadWaitOnLock())

High level use of mutexes and the Monitor Object pattern

java.util.concurrent.locks

Provides extended capabilities (introduced in Java 1.5)

1. Non-blocking attempt to acquire a lock using tryLock()
2. 2 acquires from same thread needs 2 releases
3. Provides information like 'isLocked' and 'getLockQueueLength'
4. Provides means to extend the amount and types of conditions to wait on
5. Ease use of timeout mechanisms

Performance penalty

Performance in Java

Synchronized vs Non-synchronized method

Incrementing and decrementing value 200 million times in a loop with only one thread

JDK version	Synchronized	Not Synchronized	δ in ms
1.1.8	1032 ms	1016 ms	16
1.4.2	1859 ms	1421 ms	438
1.5.0	2141 ms	1719 ms	422

Performance penalty

Monitor object principle used by:
Hashtable, Vector, StringBuffer, util.Properties

Hashtable vs HashMap

Tested on Maps containing 4.000 entries
and average based on 500 runs

	Hashtable	HashMap
fill	2 ms	3 ms
iterate	154 ms	111 ms
remove	6 ms	5 ms

Performance penalty

Performance in Java

Synchronized vs Non-synchronized method

Calling System.getProperties(), hard for the VM to optimize, 500 times using 10 threads

	void method()	synchronized(this){...}	synchronized method()
avarage	62 ms	80 ms	94 ms
maximum	186 ms	240 ms	282 ms

The Pros and Cons of the pattern

Benefits

1. Simplification of concurrency control implementation
2. Simplification of scheduling method execution
3. Implementation 'separated' from concurrency control
4. Locking mechanism and implementation closely coupled

The Pros and Cons of the pattern

Drawbacks

1. Locking mechanism and implementation closely coupled
2. Limited amount of control (e.g. no reordering of calls)
3. Limited scalable
4. Complicated extensibility semantics
5. Inheritance anomaly
6. Nested monitor lockout

Unavoidable drawbacks

1. Concurrency remains complicated
2. Caching at different levels
3. Almost impossible to test
4. Big responsability on the programmer just implementing methods

Questions ?

Excercise

In a situation where there are several layers of caching, can you think up of an structure how you would design the Monitor Object pattern within that situation ? Which object(s) would have the lock, what objects/methods could get called simultaneously, who would invoke who, how many locks would you have and why ?
Draw some UML diagram
Briefly explain your intention
Some help:

```
class A
int value = b.get();
b.set(value-1);
```

```
class B
int get()
{
    int value = c.get();
    c.set(value-1);
    return value;
}

void set(int value){
    c.set(value); }
```