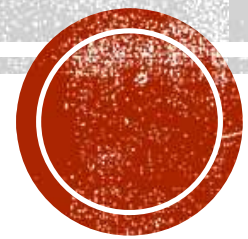
ACTOR OBJECT & MONITOR OBJECT PATTERNS



WHY PARALLELISM?

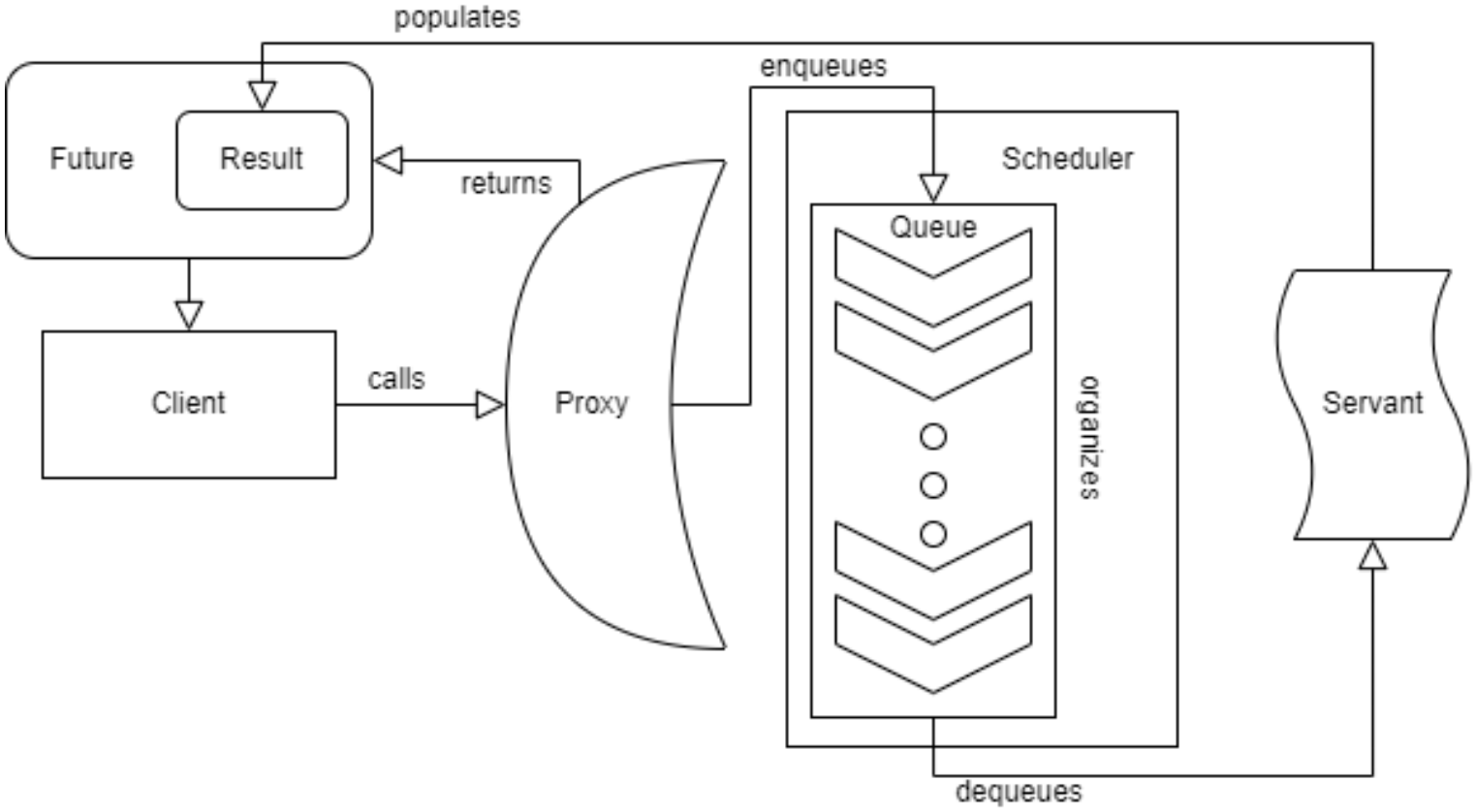
- **Improving performance transparently**
  - Taking advantage of today's hardware capabilities (utilizing multiple threads, hyperthreading…)
- **Improving performance explicitly**
  - Making independent computations overlap, as they don't need to run sequentially
- **Shortening perceived response time**
  - E.g., not blocking the GUI and letting user interact with other parts of the system while heavy computations take place in the background
- **Simplifying application design**
  - E.g., creating abstractions, turning worker threads into services, "packing" complex computations into self-contained threads…

WHY, PARALLELISM?!

- **Writing truly parallel software is hard**
  - Fully maximizing performance often results in incomprehensible code
- **There's lots of caveats one has to be aware of**
  - Deadlocks, livelocks, starvation, race conditions…
- **Bottlenecks can ruin it all**
  - Regardless how well the system runs, a single bottleneck can nullify any advantages of concurrency

ACTIVE OBJECT PATTERN

- Why have a thread when you can have a service?



+ LogLevel enum
+ Logger interface

THE PROBLEM – A LOGGER

```
public enum LogLevel {
    LOG,
    WARNING,
    ERROR
}
```

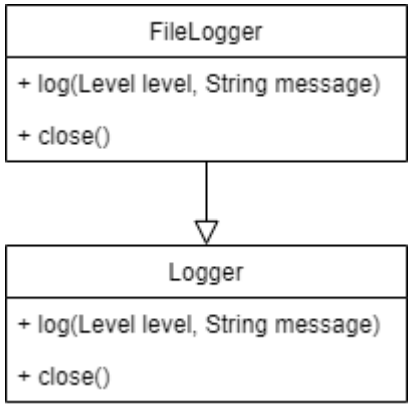
```
public interface Logger {
    void log(LogLevel level, String message) throws IOException;

    void close() throws IOException;
}
```

```
public static void main(String[] args) throws IOException {
    Logger logger = getDefaultLogger();

    for (int i = 0; i < 1000000; i++) {
        logger.log(LogLevel.LOG, String.format(" Sqrt of number %d is %f", i, Math.sqrt(i)));
    }

    logger.close();
}
```

LogLevel enum
Logger interface
\+ FileLogger class

CLASSIC LOGGER IN MAIN THREAD

- Logs clog the main thread
- Everything else needs to wait
- That's especially painful once we add a GUI



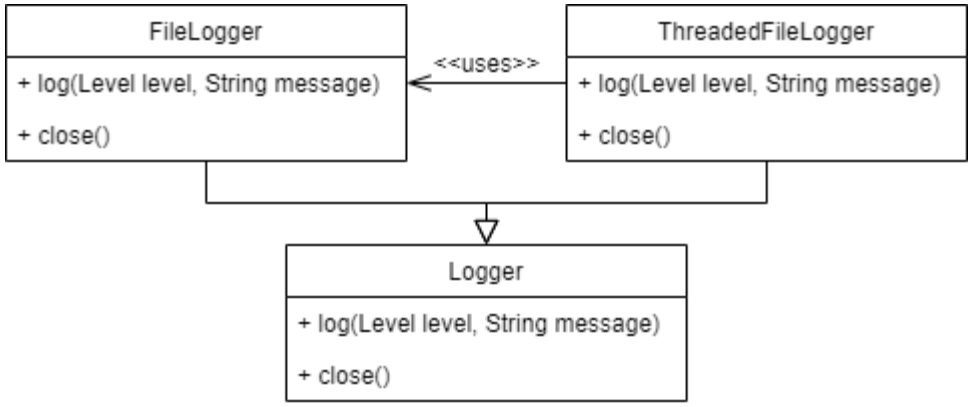

```
public class FileLogger implements Logger {

    public FileWriter writer;

    public FileLogger(String outFile) throws IOException {
        writer = new FileWriter(outFile, true);
    }

    @Override
    public void log(LogLevel level, String message) throws IOException {
        switch (level) {
            case LOG -> writer.write("LOG:" + message + "\n");
            case WARNING -> writer.write("WARNING:" + message + "\n");
            case ERROR -> writer.write("ERROR:" + message + "\n");
            default -> throw new IllegalArgumentException();
        }
    }

    @Override
    public void close() throws IOException {
        writer.close();
    }
}
```

Let's try putting it in another thread

LogLevel enum
Logger interface
FileLogger class
\+ ThreadedFileLogger class

LOGGER IN ANOTHER THREAD

- Pretty bad coordination (logs out of order)
- Messy
- Hard to extend
- If internal logger fails, we have no way of knowing



We need a scheduler to keep the correct order

```
public class ThreadedFileLogger implements Logger {

    private final Logger internalLogger;
    private Lock lock = new ReentrantLock();

    public ThreadedFileLogger(String outFile) throws IOException {
        internalLogger = new FileLogger(outFile);
    }

    @Override
    public void log(LogLevel level, String message) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    lock.lock();
                    internalLogger.log(level, message);
                } catch (IOException e) {
                    try {
                        internalLogger.log(LogLevel.ERROR, Arrays.toString(e.getStackTrace()));
                        internalLogger.close();
                    } catch (IOException ignored) {}
                }
                finally {
                    lock.unlock();
                }
            }
        }).start();
    }

    @Override
    public void close() throws IOException {
        internalLogger.close();
    }
}
```

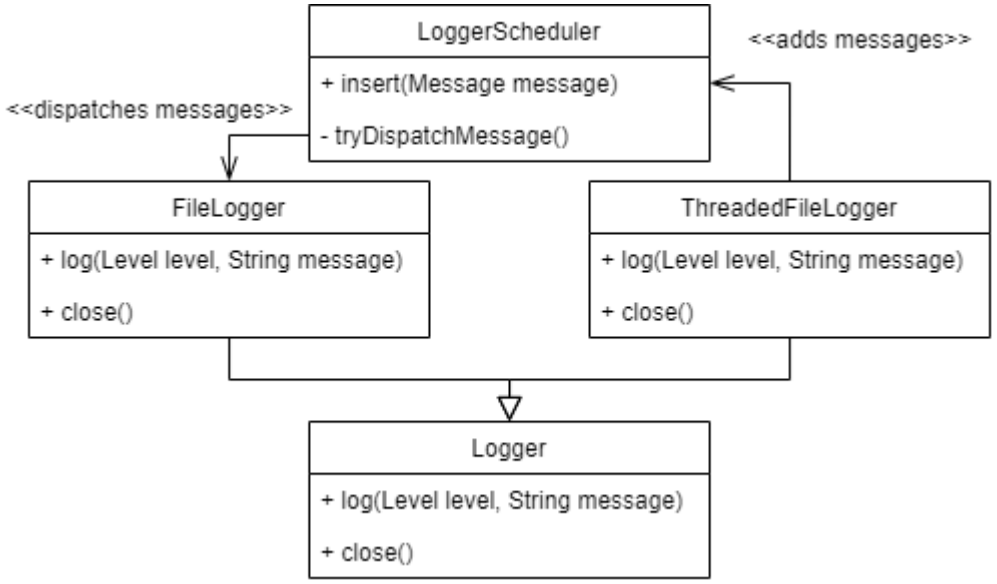
- Lock
- Log
- Unlock

```
LOG: Sqrt of number 3 is 1.732051
LOG: Sqrt of number 4 is 2.000000
LOG: Sqrt of number 53 is 7.280110
LOG: Sqrt of number 5 is 2.236068
LOG: Sqrt of number 6 is 2.449490
```

LogLevel enum
Logger interface
FileLogger class
ThreadedFileLogger class
\+ LoggerScheduler class

ADDING A SCHEDULER

- Scheduler runs on a separate thread
- Actively inspects messages, looking for the best one to dispatch based on heuristics



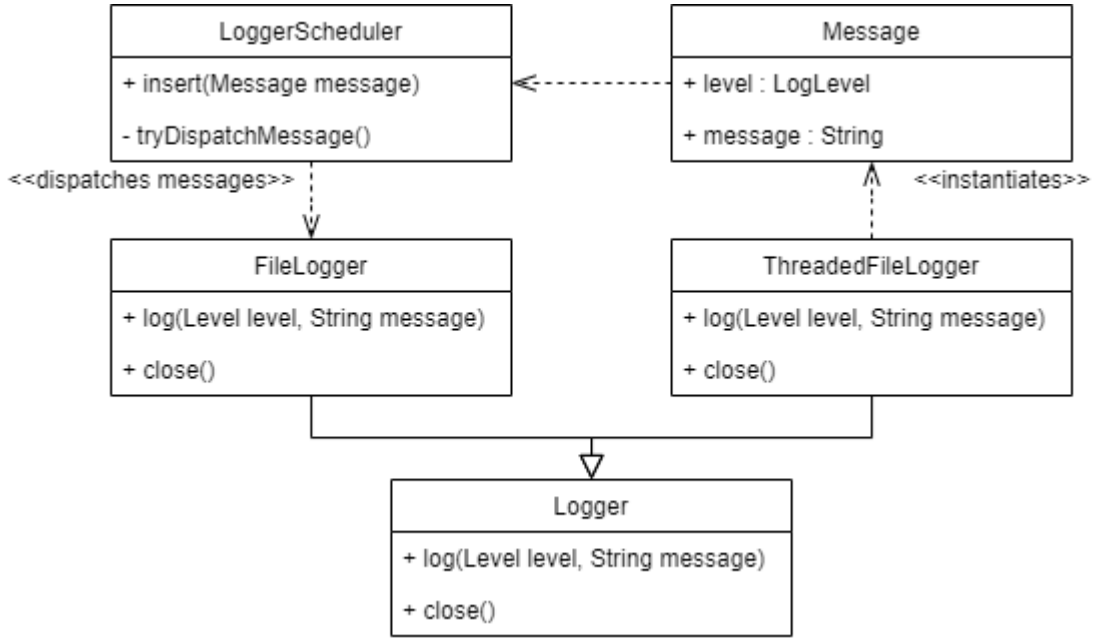

```
public class LoggerScheduler implements Runnable {

    private Queue<Message> queue = new ArrayDeque<>();
    private Lock queueLock = new ReentrantLock();
    private Logger servant;

    public LoggerScheduler(Logger servant) {
        this.servant = servant;
    }

    public void insertMessage(Message message) {
        queueLock.lock();
        queue.add(message);
        queueLock.unlock();
    }

    @Override
    public void run() {
        while(true) {
            queueLock.lock();
            tryDispatchMessage();
            queueLock.unlock();
        }
    }

    private void tryDispatchMessage() {
        Message bestMessage = null;
        for (var message : queue) {
            if (bestMessage == null || bestMessage.ordNum > message.ordNum) {
                bestMessage = message;
            }
        }
        if (bestMessage != null) {
            queue.remove(bestMessage);
            try {
                servant.log(bestMessage.level, bestMessage.message);
            } catch (IOException e) {
                try {
                    servant.log(LogLevel.ERROR, e.getStackTrace().toString());
                    servant.close();
                } catch (IOException ignored) {}
            }
        }
    }

}
```

Synchronized method to alter queue, called from non-scheduler threads

Scheduler's dispatch loop, needs to unlock to give space for proxy

ADDING A SCHEDULER

- Wrap logging parameters in a Message
  - Also contains info for Scheduler
- ThreadedFileLogger talks with Scheduler instead of Servant




```
  LogLevel enum
  Logger interface
        FileLogger class
~       ThreadedFileLogger class
+ LoggerScheduler class
+ Message class
```

```java
public class ThreadedFileLogger implements Logger {

    private final LoggerScheduler scheduler;

    public ThreadedFileLogger(String outFile) throws IOException {
        var internalLogger = new FileLogger(outFile);
        scheduler = new LoggerScheduler(internalLogger);
        new Thread(scheduler).start();
    }

    @Override
    public void log(LogLevel level, String message) {
        scheduler.insertMessage(new Message(level, message));
    }

    @Override
    public void close() throws IOException {
    }
}
```

```java
public class Message {
    private static int ordNumCounter = 0;
    int ordNum;
    LogLevel level;
    String message;

    public Message(LogLevel level, String message) {
        ordNum = ordNumCounter++;
        this.level = level;
        this.message = message;
    }
}
```

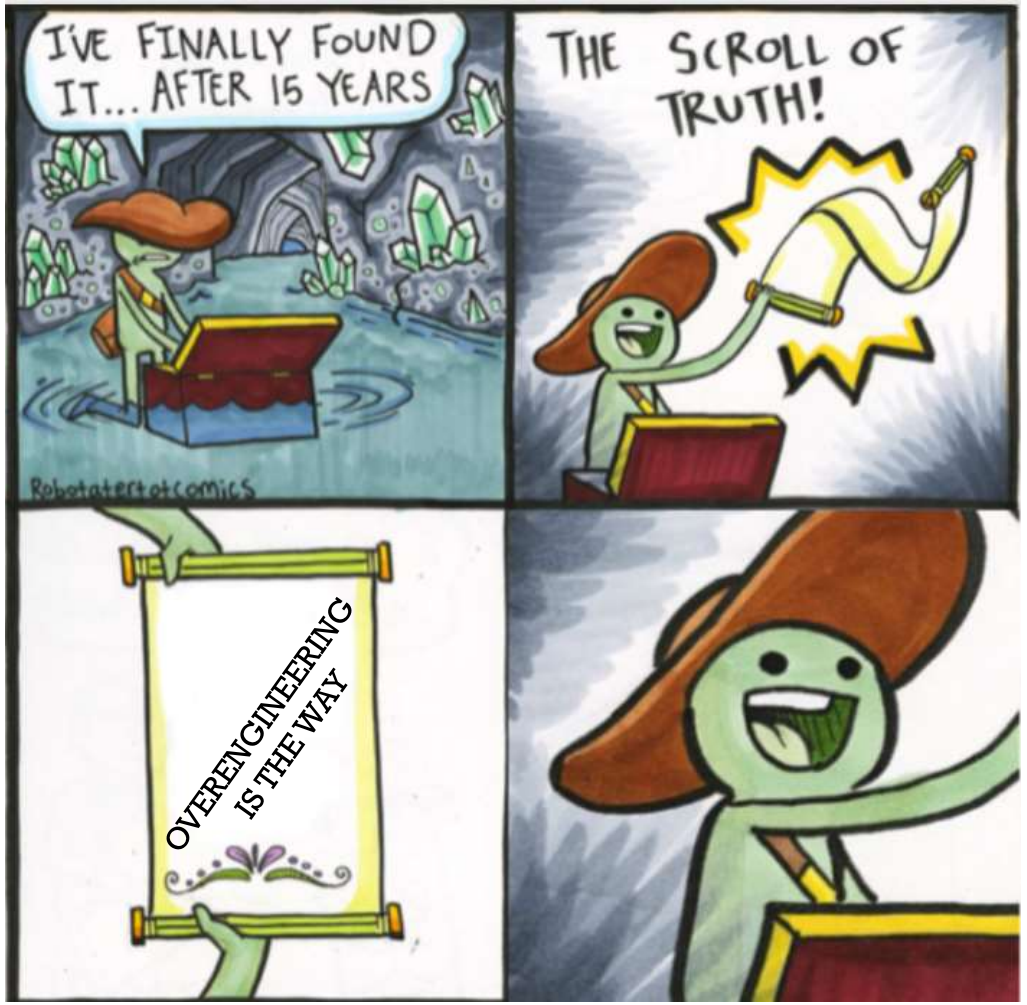
ADDING A SCHEDULER

- Logging should be in the correct order
- We leveraged logic from the proxy

- We're lacking 'close()' though
- Also what if we wanted something else than just passing messages?

We need ABSTRACTIONS

LogLevel enum
Logger interface
    FileLogger class
    ThreadedFileLogger class
LoggerScheduler class
Message class



MAKING MESSAGES INTO REQUESTS

LogLevel enum
Logger interface
    FileLogger class
    ThreadedFileLogger class
LoggerScheduler class
\- Message class
\+ LoggerMethodRequest class

```
public class Message {
    int ordNum;
    LogLevel level;
    String message;

    public Message(LogLevel level, String message) {
        ordNum = nextOrdNum();
        this.level = level;
        this.message = message;
    }
}
```

```
public abstract class LoggerMethodRequest {

    public final int priority = -1;
    public int ordNum = -1;

    public abstract void execute(Logger logger) throws IOException;

}
```

- Make messages more abstract
- Now they can pass complete logic, not just parameters

MAKING MESSAGES INTO REQUESTS

LogLevel enum
Logger interface
    FileLogger class
    ThreadedFileLogger class
LoggerScheduler class
LoggerMethodRequest class
\+ LoggerCloseRequest class
\+ LoggerLogRequest class

```
public abstract class LoggerMethodRequest {

    public final int priority = -1;
    public int ordNum = -1;

    public abstract void execute(Logger logger) throws IOException;

}
```

```
public class LoggerCloseRequest extends LoggerMethodRequest {

    public final int priority = 1;

    @Override
    public void execute(Logger logger) throws IOException {
        logger.close();
    }
}
```

```
public class LoggerLogRequest extends LoggerMethodRequest {

    public final int priority = 3;
    private static int ordNumCounter = 0;
    LogLevel level;
    String message;

    public LoggerLogRequest(LogLevel level, String message) {
        ordNum = ordNumCounter++;
        this.level = level;
        this.message = message;
    }

    @Override
    public void execute(Logger logger) throws IOException {
        logger.log(level, message);
    }
}
```

- Requests can now request actual methods of the servant
- Sky is the limit

```
public class Message {
    int ordNum;
    LogLevel level;
    String message;

    public Message(LogLevel level, String message) {
        ordNum = nextOrdNum();
        this.level = level;
        this.message = message;
    }
}
```

MAKING MESSAGES INTO REQUESTS

- LogLevel enum
- Logger interface
  - FileLogger class
  - ThreadedFileLogger class
- ~ LoggerScheduler class
- LoggerMethodRequest class
  - LoggerCloseRequest class
  - LoggerLogRequest class

- Make the scheduler compatible with method requests
- One last thing left to work out:
  - How to get data from the logger in main thread?

FUTURE / PROMISE

```
public class LoggerScheduler implements Runnable {

    private Queue<LoggerMethodRequest> queue = new ArrayDeque<>();

    // ...

    private void tryDispatchMessage() {
        LoggerMethodRequest nextRequest = null;
        for (var request : queue) {
            if (nextRequest == null) {
                nextRequest = request;
            }
            else if (nextRequest.priority < request.priority) {
                nextRequest = request;
            }
            else if (nextRequest.ordNum > request.ordNum) {
                nextRequest = request;
            }
        }

        if (nextRequest != null) {
            queue.remove(nextRequest);
            try {
                nextRequest.execute(servant);
            } catch (IOException e) {
                try {
                    servant.log(LogLevel.ERROR, e.getStackTrace().toString());
                    servant.close();
                } catch (IOException ignored) {}
            }
        }
    }

}
```

- Pick highest priority
- Pick lowest order

Execute the selected method

CREATING FUTURE

- As the result is not available at the time of calling, Future is returned instead
- Client can wait and pull the value out once it's done
- ThreadedLogger no longer conforms to original interface

```
  LogLevel enum
  Logger interface
        FileLogger class
+ ThreadedLogger interface
~   ThreadedFileLogger class
  LoggerScheduler class
  LoggerMethodRequest class
    LoggerCloseRequest class
    LoggerLogRequest class
+   LoggerCounterRequest class
```

```java
public class Main {

    public static void main(String[] args) {
        Logger logger = new ThreadedFileLogger("./out.txt");

        var count = 1000000;

        for (int i = 0; i < count; i++) {
            logger.log(LogLevel.LOG, String.format(" Sqrt of number %d is %f", i, Math.sqrt(i)));
        }
        System.out.println("Logging finished, now for the summary...");
        var counter = logger.getLogCounter();
        while(!counter.isDone()) {
            Thread.sleep(1);
        }
        System.out.println("Altogether logged " + counter.get() + " messages.");

        logger.close();
    }
}
```

Get future, wait for it to complete, get the result

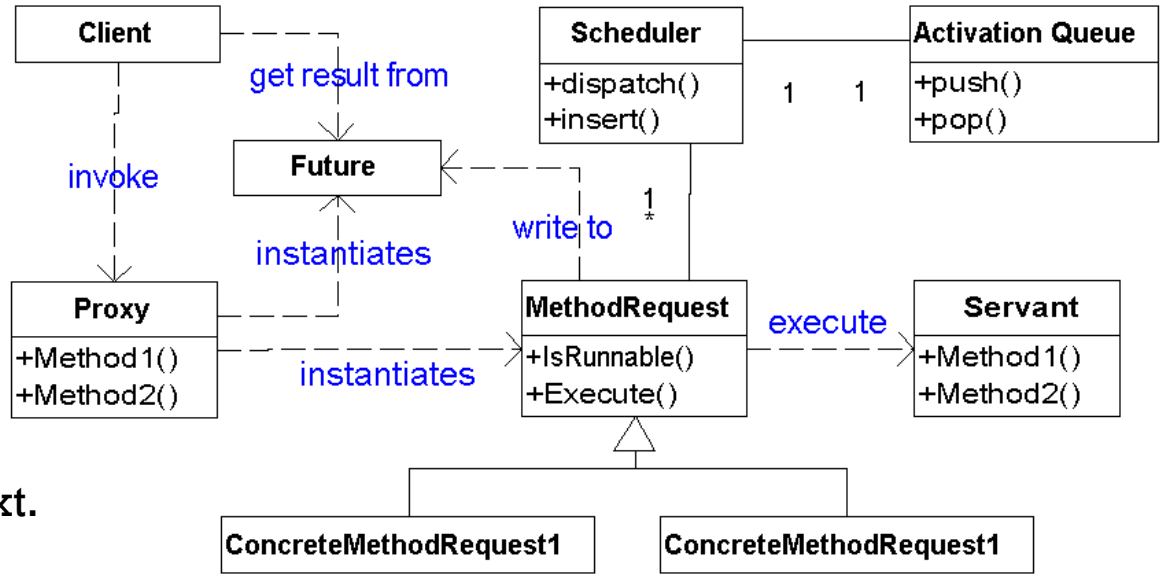
```java
public class LoggerCounterRequest extends LoggerMethodRequest {

    public final int priority = 2;

    CompletableFuture<Integer> response = new CompletableFuture<>();

    public Future<Integer> getResponse() { return response; }

    @Override
    public void execute(Logger logger) throws IOException {
        var counter = logger.getLogCounter();
        response.complete(counter.get());
    }
}
```

```java
public class ThreadedFileLogger implements Logger {

    // ...

    @Override
    public Future<Integer> getLogCounter() {
        var counterRequest = new LoggerCounterRequest();
        var counterResponse = counterRequest.getResponse();
        scheduler.insertRequest(counterRequest);
        return counterResponse;
    }
}
```

ACTIVE OBJECT PATTERN - FORMALLY

Consists of:

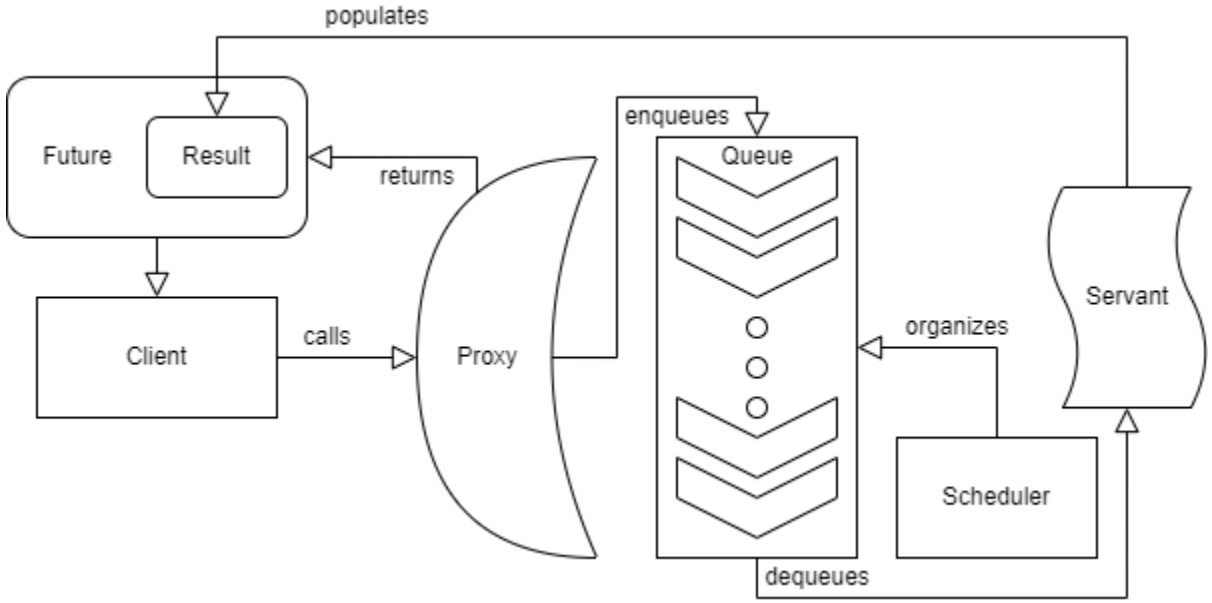
- A **proxy**
  - which provides an interface towards clients with publicly accessible methods.
- An **interface**
  - which defines the method request.
- A **list of pending requests** from clients.
- A **scheduler**
  - which decides which request to execute next.
- The **implementation** (servant)
  - of the active object method.
- A **callback** or variable
  - for the client to receive the result.



ACTIVE OBJECT PATTERN VS. OUR LOGGER

Consists of:

- A **proxy - ThreadedFileLogger**
- An **interface – Logger**
- A **list of pending requests** from clients.
- A **scheduler - LoggerScheduler**
- The **implementation - FileLogger**
- A **callback – Future<>**

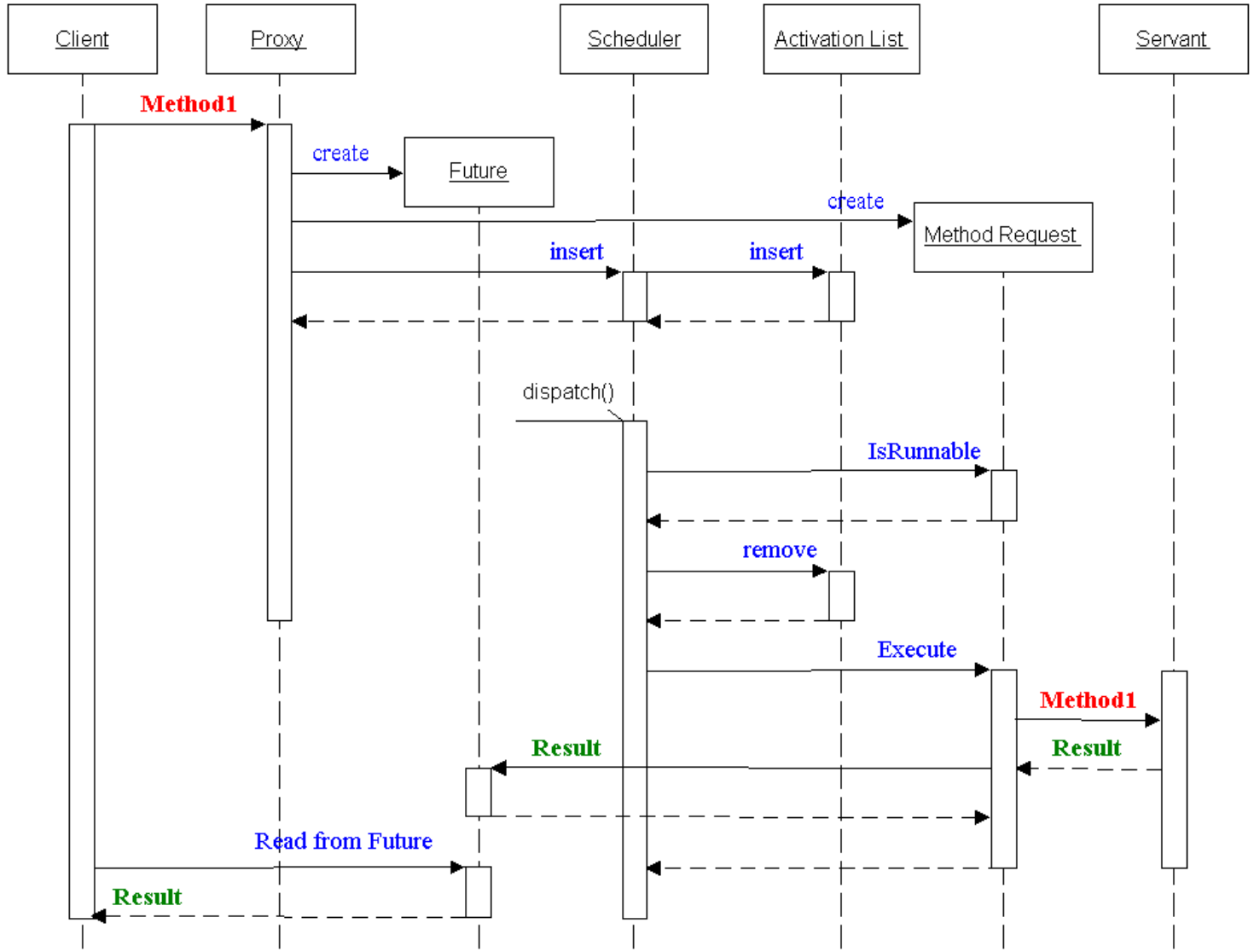


ACTIVE OBJECT – LAST ANALOGY: THE RESTAURANT

- A **proxy** is the waiter, taking orders and forwarding them to the kitchen
- A **list of pending requests** is that weird string you see in diners in movies where new orders are hung.
- A **scheduler** is the poor cook trying to pick which orders can be made sooner.
- The **implementation** (servant) is the chef, whom the client never sees and knows nothing about
- A **callback** is the table's number written in waiter's notebook, that'll make him bring the food once it's good to go.

THE FLOW

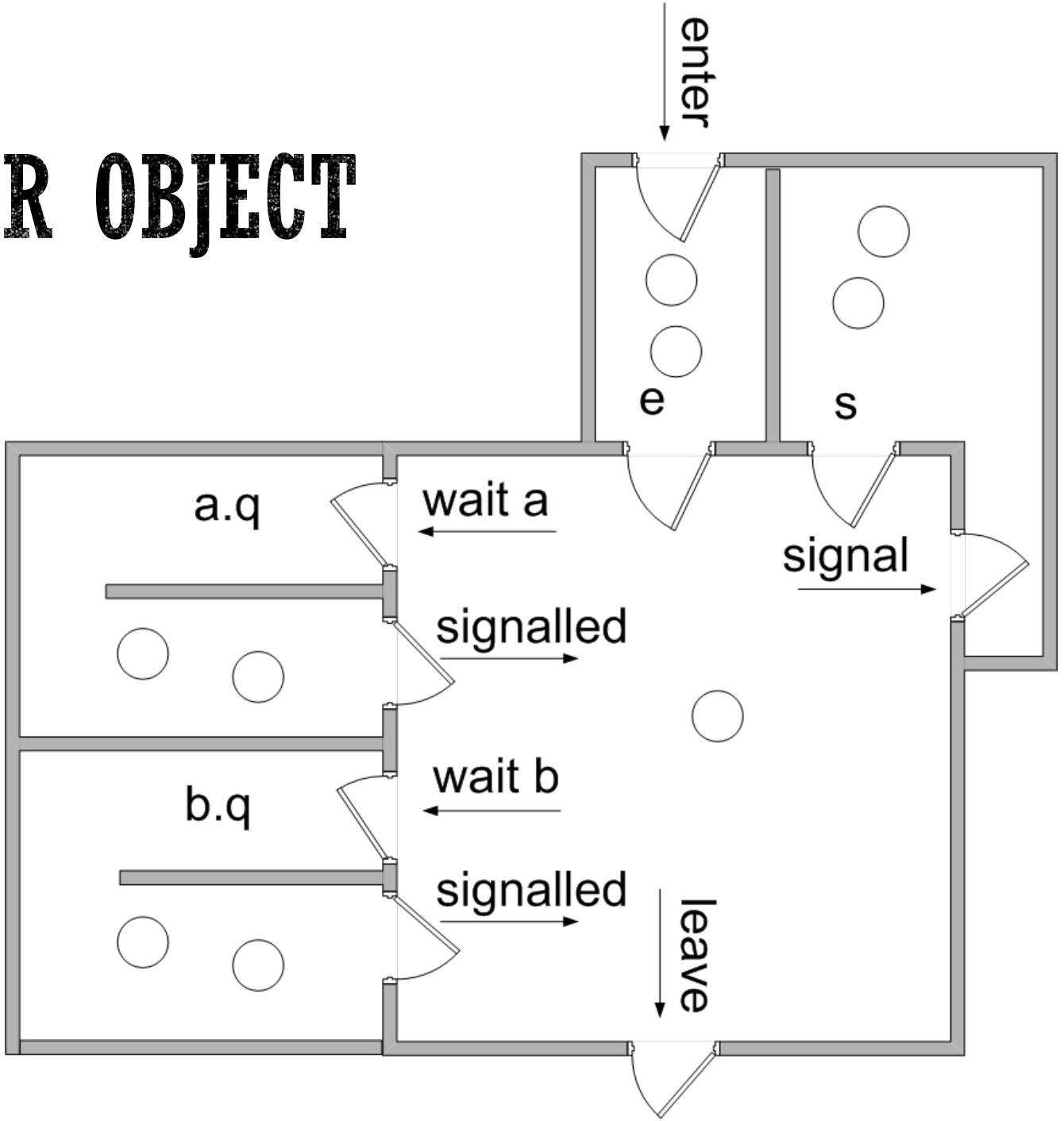
- Client calls the method
- Proxy creates a future and method request
- Scheduler adds it to activation list
- Once method is on top of the queue, it is executed by the servant
- Method request writes the return value (if any) to the future
- Finally client can read and use it



ACTIVE OBJECT – WHY AND WHEN

- Making a background process "more than just a process".
  - Full-fledged service
  - Can communicate to and fro using a fixed interface
- Can support arbitrarily complex scheduling heuristics
- Fair amount of added complexity – usually used on non-trivial worker threads.
- Good design (e.g., not passing MethodRequest mutable params by reference) removes pitfalls of multithreading.
- From client standpoint no added complexity (apart from futures).

MONITOR OBJECT



THE PROBLEM – ORDER MANAGER

- Single shared memory contains all the data
- Individual threads manipulate it
- Issues:
  - Lots of waiting
  - We know the suppliers only need to get access once a consumer takes something, and vice versa.

```
public class OrderSupplier {

    private OrderQueue queue;

    public OrderSupplier(OrderQueue queue) {
        this.queue = queue;
    }

    public void supply(Order order) {
        try {
            queue.addNext(order);
        } catch(Exception ignored) {}
    }
}
```

```
public class OrderConsumer {

    private OrderQueue queue;

    public OrderConsumer(OrderQueue queue) {
        this.queue = queue;
    }

    public void consume() {
        Order order = null;
        try {
            order = queue.getNext();
        } catch (Exception ignored) {}
    }
}
```

```
public class OrderQueue {

    public Queue<Order> orders;
    Lock lock = new ReentrantLock();

    public OrderQueue() {
        orders = new ArrayDeque<>(10);
    }

    public boolean isFull() {
        return orders.size() == 10;
    }

    public boolean isEmpty() {
        return orders.isEmpty();
    }

    public Order getNext() throws InterruptedException {
        while (true) {
            lock.lock();
            if (isEmpty()) {
                lock.unlock();
                Thread.sleep(1);
            }
            else {
                var next = orders.poll();
                lock.unlock();
                return next;
            }
        }
    }

    public void addNext(Order order) throws InterruptedException {
        while (true) {
            lock.lock();
            if (isFull()) {
                lock.unlock();
                Thread.sleep(1);
            }
            else {
                orders.add(order);
                lock.unlock();
                return;
            }
        }
    }
}
```

Nothing to take, sleep and check later

ADDING HEURISTIC

- Once queue is empty, deactivate all consumers
- Reactivate them once that changes
- Vice versa for suppliers
- We just invented **Condition Variables**
  - If condition is not met, put a thread on hold, and get back to it once condition holds again

```
public class OrderManager {

    // ...

    List<Thread> waitingForNonEmpty;
    List<Thread> waitingForNonFull;

    public OrderManager() {
        orders = new ArrayDeque<>(10);
        waitingForNonEmpty = new ArrayList<>();
        waitingForNonFull = new ArrayList<>();
    }

    // ...

    public void reactivateNonFullThreads() {
        for (var thread: waitingForNonFull) {
            thread.resume();
        }
    }

    public Order getNext() throws InterruptedException {
        while (true) {
            lock.lock();
            if (isEmpty()) {
                waitingForNonEmpty.add(Thread.currentThread());
                lock.unlock();
                Thread.currentThread().suspend();
            }
            else {
                var next = orders.poll();
                reactivateNonFullThreads();
                lock.unlock();
                return next;
            }
        }
    }

    // ...
}
```

Instead of polling, just take a nap and make it someone else's problem

Now things have changed, go tell the others

ADDING HEURISTIC

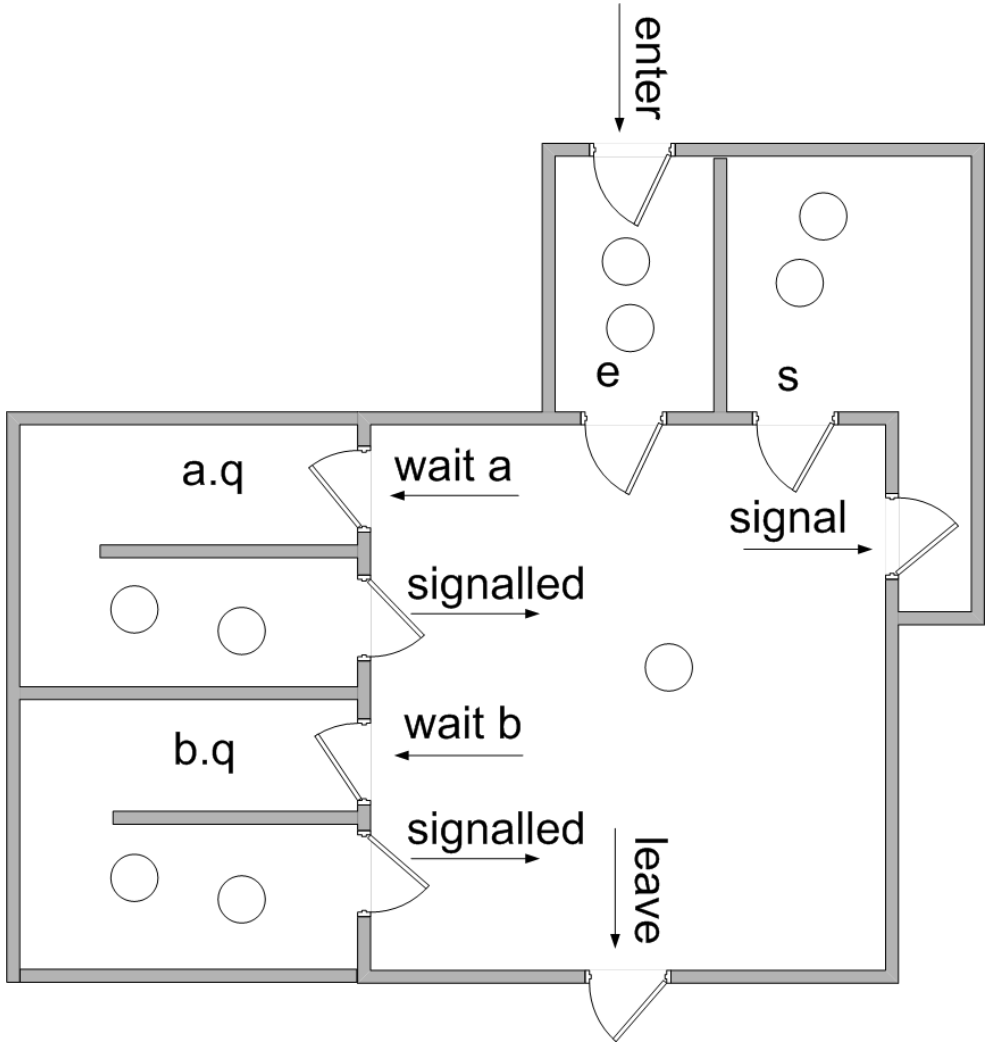
- Languages have fancy constructs for this:
  - Java has java.util.concurrent.locks.Condition
  - C# has System.Threading.Monitor
  - C++ has std::condition_variable
  - JS has… lots of questionable NPM packages

```
public class OrderManager {

    public Queue<Order> orders;

    Lock lock = new ReentrantLock();
    Condition notFull = lock.newCondition();
    Condition notEmpty = lock.newCondition();

    // ...

    public Order getNext() throws InterruptedException {
        Order next = null;
        lock.lock();

        while (isEmpty())
            notEmpty.await();

        next = orders.poll();
        notFull.signal();

        lock.unlock();
        return next;
    }

    public void addNext(Order order) throws InterruptedException {
        lock.lock();

        while (isFull())
            notFull.await();

        orders.offer(order);
        notEmpty.signal();

        lock.unlock();
    }
}
```

MONITOR OBJECT PATTERN - FORMALLY

Consists of:

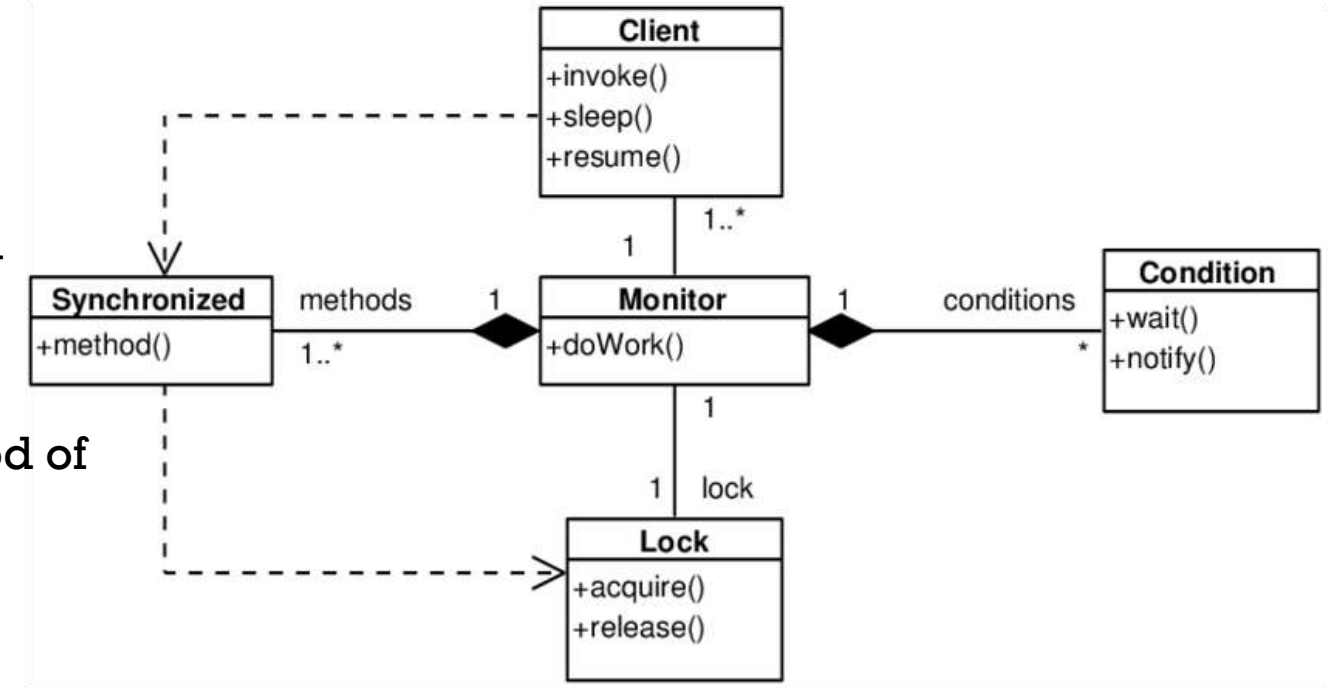
- Monitor object
  - Object which is accessed concurrently
- Synchronized method
  - Method within monitor object, publicly accessible
- Monitor lock
  - A lock object ensuring only one method of monitor is executed at the same time
- Monitor Condition
  - Helps determine which synchronized methods should be suspended and reactivated



MONITOR OBJECT PATTERN - FORMALLY

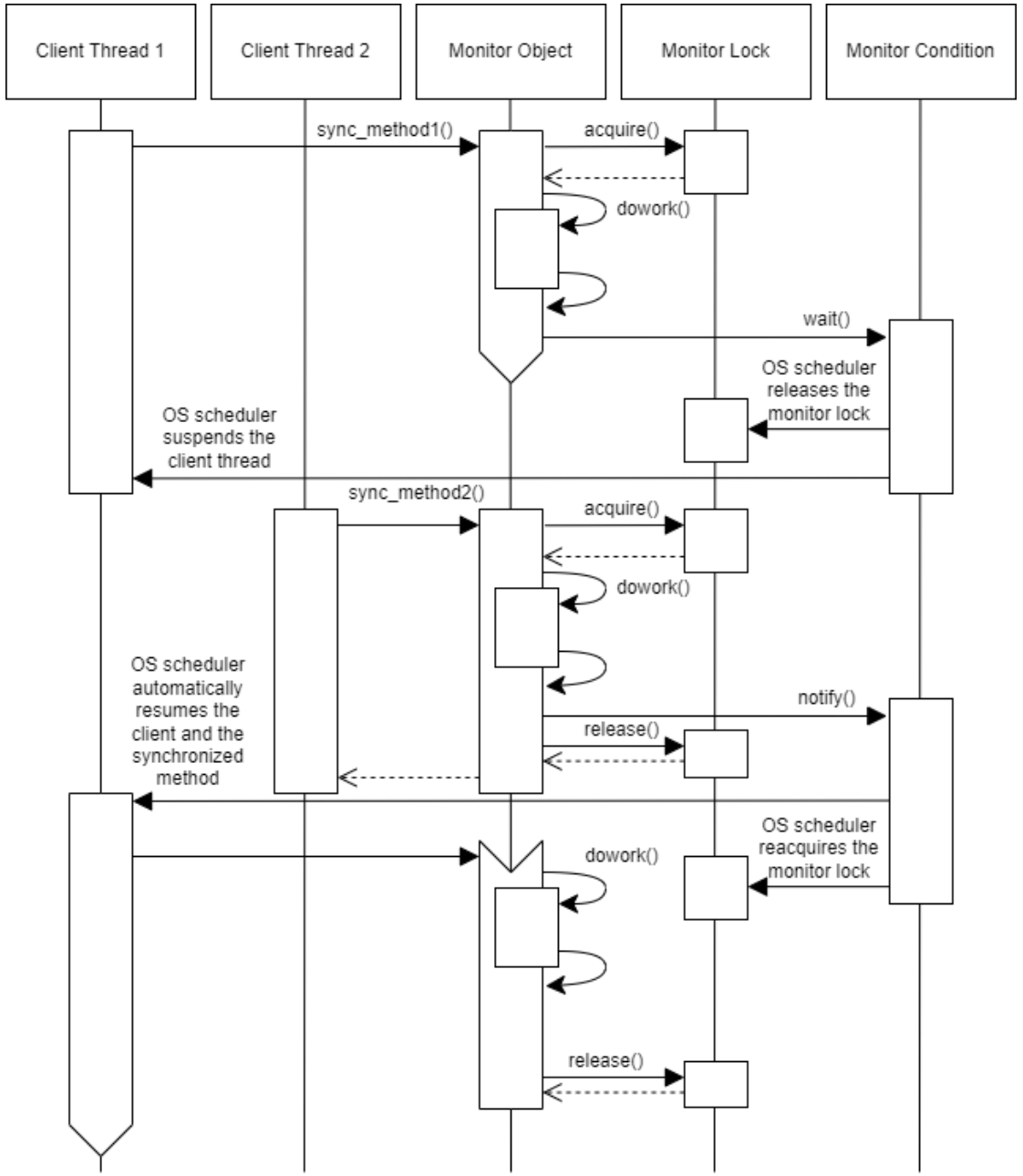
Consists of:

- Monitor object
  - Object which is accessed concurrently
- Synchronized method
  - Method within monitor object, publicly accessible
- Monitor lock
  - A lock object ensuring only one method of monitor is executed at the same time
- Monitor Condition
  - Helps determine which synchronized methods should be suspended and reactivated



THE FLOW

- Client 1 starts a method
- Acquires lock, does some work
- Encounters state where it cannot work (e.g., supplier finds full queue)
- Client 1 waits
- Scheduler releases lock
- Client 2 starts a method
- Does some work that changes the condition
- Notifies about the change and releases the lock
- Monitor condition re-activates Client 1, reacquires the lock
- Client 1 finishes its job, releases lock



MONITOR OBJECT – WHY AND WHEN

- Used when some shared memory has set of fixed contracts which affect each other, and are accessed by multiple processes
- Handles synchronization and enables processes to stay oblivious to it
- Lacks own thread of control and, therefore has no control over the order synchronized methods access it