



the
POWER
of
JAVA™

M A R K
LOGIC



JavaOne
Sun and Network Associates

How to Build a Scalable Multiplexed Server With NIO

Ron Hitchens

Senior Engineer
Mark Logic Corporation
<http://marklogic.com>

President
Ronsoft Technologies
<http://ronsoft.com>



TS-1315

Think Big

Sockets, Sockets and More Sockets

Learn to use New I/O (NIO) effectively to build a scalable, multiplexed server with Java™ technology

Building an NIO Server

Introduction

Understanding the problem

Defining a solution

NIO implementation

Summary

Pig Footed Bandicoot

Ron Hitchens

Years spent hacking UNIX® internals

Device drivers, I/O streams, etc.

Wrote a small performance tuning book in 1995

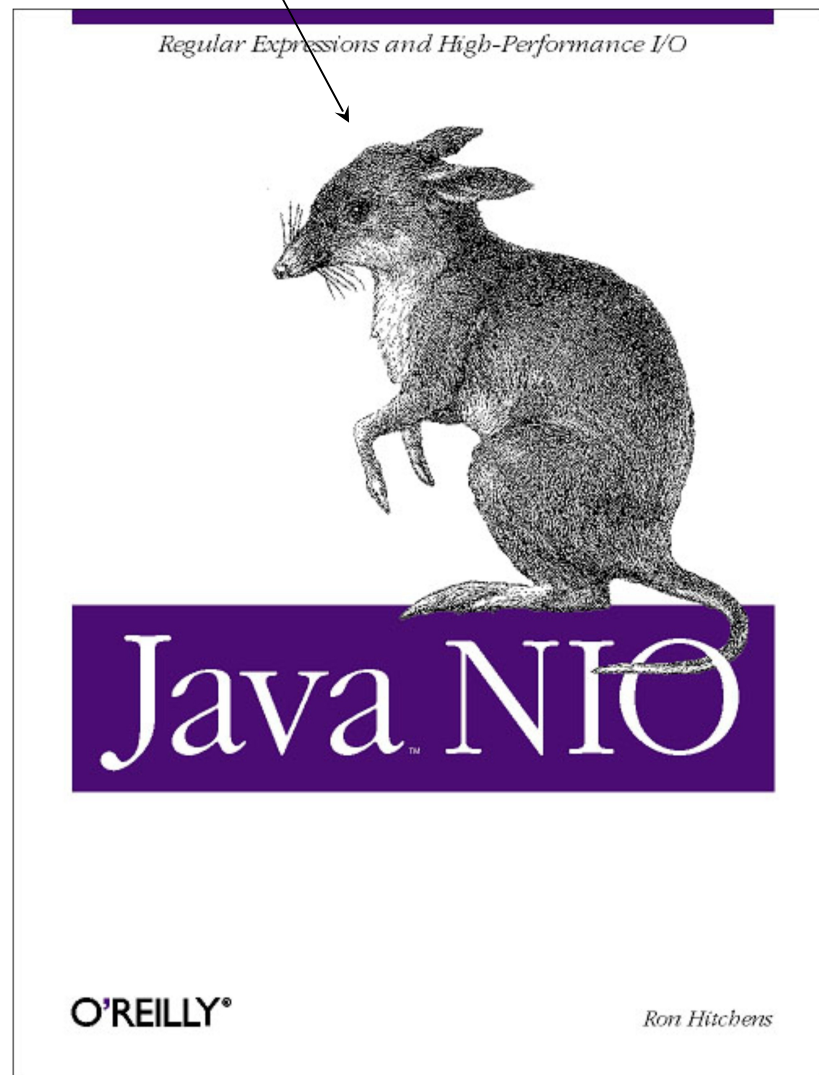
Got into Java technology in a big way in 1997

Java NIO published August 2002

Wrote an NIO-based chat server that manages 1000s of connections 24x7

Been at Mark Logic since 2004

Lots of XML, XQuery and Java technology, not so much NIO lately



Building an NIO Server

Introduction

Understanding the problem

Defining a solution

NIO implementation

Summary

What Does a Server Do?

- A server processes requests:
 - Receive a client request
 - Perform some request-specific task
 - Return a response
- Multiple requests run concurrently
- Client requests correspond to connections
 - Sequential requests may be sent on a single socket
- Requests may contend for resources
- Tolerates slow, misbehaving or unresponsive clients

Multiplexing Strategies

Approaches to Managing Many Sockets Concurrently

- Poll each socket in turn
 - Impractical without non-blocking sockets
 - Inefficient, not very fair and scales poorly
- Thread-per-socket
 - Only practical solution for blocking sockets
 - Stresses the thread scheduler, which limits scalability
 - Thread scheduler does readiness selection—inefficiently
- Readiness selection
 - Efficient, but requires OS and Java VM support
 - Scales well, especially for many slow clients

Other Considerations

Things to Think About When Thinking Big

- Multi-threading issues are magnified
 - Concurrent access controls may become a bottleneck
 - Non-obvious example: formatting text messages for logging
 - Potential for deadlocks
 - Per-thread overhead
 - Diminishing returns as threads/CPU ratio increases
- Quality-of-service policy under load
 - Define acceptable performance criteria
 - Define what happens when threshold(s) are reached
 - Do nothing different, prioritize requests, queue new requests, reject new requests, redirect to another server, and so on and so on...
- Client profile
 - Ratio of connected clients to running requests
 - Can (or must) you tolerate malfunctioning or malicious clients?

Building an NIO Server

Introduction

Understanding the problem

Defining a solution

NIO implementation

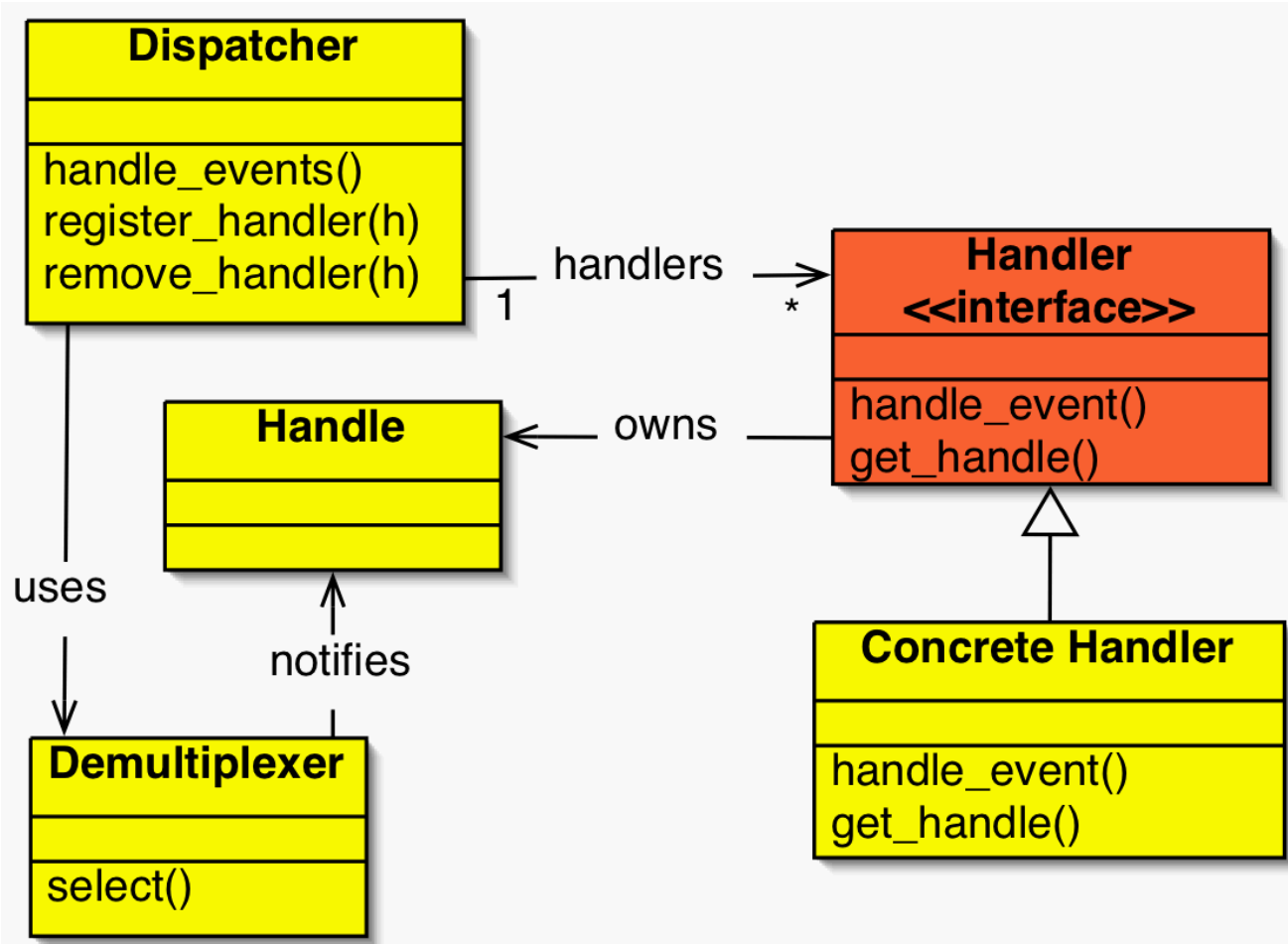
Summary

The Reactor Pattern

AKA: Dispatcher, Notifier

- Published in *Pattern Languages of Program Design*, 1995, ISBN 0-201-6073-4
- Paper by Prof. Douglas C. Schmidt
 - <http://www.cs.wustl.edu/~schmidt/patterns-ace.html>
 - Google for: Reactor Pattern
- Describes the basic elements of a server
- Gives us a vocabulary for this discussion

Reactor Pattern UML



Reactor Pattern Participants

The Moving Parts

- **Handle**
 - A reference to an event source, such as a socket
- **Event**
 - A state change that can occur on a Handle
- **Demultiplexer**
 - Reacts to and interprets Events on Handles
- **Dispatcher**
 - Invokes Handlers for Events on Handles
- **Handler**
 - Invoked to process an Event on a Handle

What Does a Server Do?

- A server processes requests:
 - Receive a client request
 - Perform some request-specific task
 - Return a response
- Multiple requests run concurrently
- Client requests correspond to connections
 - Multiple requests may be sent on a single socket
- Requests may contend for resources
- Tolerates slow or unresponsive clients

Dispatcher

Handler

Demultiplexer

Handle

Event

Scalability

Scalability and Robustness [1]

In Order to Scale Well...

- The Demultiplexer must:
 - React quickly and reliably to new Events
 - Promptly notify the Dispatcher of new Events
 - Efficiently manage large numbers of Handles (sockets)
 - Never delay or block while doing any of the above
- The Dispatcher must:
 - Efficiently map an Event on a Handle to a Handler
 - Tell Demultiplexer how to treat Handle while Handler is running
 - Implement Quality of Service policies
 - Schedule Handlers for execution, usually in different threads
 - Gracefully cope with all possible Handler errors
 - Efficiently manage large numbers of running Handler threads
 - Never delay or block while doing any of the above

Scalability and Robustness [2]

In Order to Scale Well...

- Handlers must:
 - Be well-behaved
 - Run concurrently with other handlers
 - Stick to their defined responsibilities
 - Avoid or minimize resource contention
 - Not impede other handlers unnecessarily
 - Finish promptly
 - Never block indefinitely while doing any of the above

Dispatcher Flow (Single Threaded)

```
Register handler(s) for event(s)
```

```
...
```

```
Do forever
```

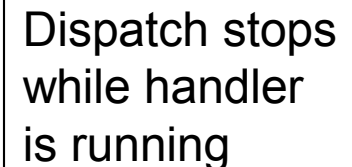
```
  Ask demultiplexer for current events on registered  
  handles (may block indefinitely)
```

```
  For each current event*
```

```
    Invoke handler for event
```

```
    Clear the event for the handle
```

Dispatch stops
while handler
is running



*Events should "latch on" until handled

Dispatcher Flow (Multi-Threaded)

Do forever

Ask demultiplexer for current events on registered handles
(may block indefinitely)

For each current event

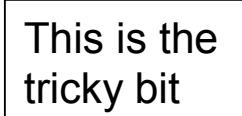
Ask demultiplexer to stop notification of the event

Note that the handler for this event is running

Schedule handler for execution in a worker thread

Clear the event for the handle

This is the
tricky bit



<some time later, in some other (handler) thread>

Tell dispatcher the handler has finished running

Tell demultiplexer to resume notification of the event

Synchronize perfectly, don't clobber or miss anything



A Quick Diversion...

Don't Forget—The Channels Are Non-Blocking

- Network connections are **streams**
 - If your code assumes structured reads, it's broken
- When reading:
 - You may only get some (or none) of the data
 - Structured messages **will be** fragmented
 - You must buffer bytes and reconstitute the structure
- When writing:
 - The channel may not accept all you want to send
 - You must queue output data
 - Don't spin in a handler waiting for output to drain

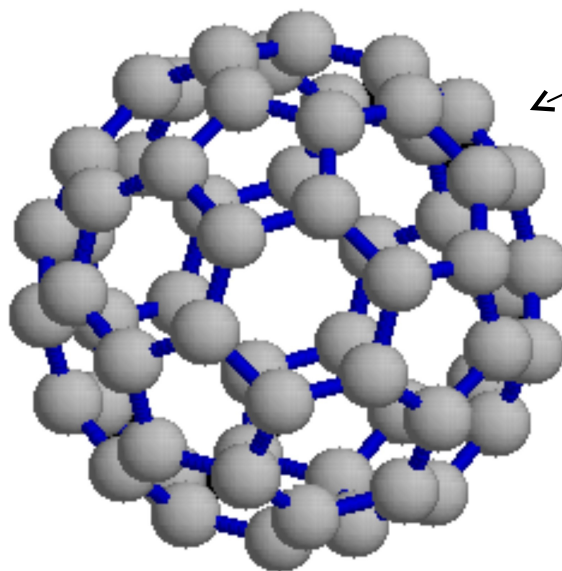


Observations

- Handling ready-to-write is just buffer draining
 - Handlers shouldn't wait for their output to be sent
 - Generic code should handle output queue draining
- Reads are non-blocking and may fragment
 - Generic code should do input queue handling
- Client handlers process complete “messages”
 - Generic code should queue data for reassembly
- Handler threads must interact with Dispatcher
 - Dispatcher needs to know when threads finish
 - Handler may want to disable reads, unregister, etc.

Hey, That Sounds Like a Framework

- Yep. Inversion of Control—it's all the rage
- Make it once, make it solid, make it reusable



Carbon₆₀ BuckyBall
(Buckminsterfullerene)

Building an NIO Server

Introduction

Understanding the problem

Defining a solution

NIO implementation

Summary

Assumptions



- The server is to be multi-threaded
 - Using the `java.util.concurrent` package (Java SE 5)
- One select loop (Dispatcher)
 - Accepting new sockets is done elsewhere
 - We only care about configurable input handlers
- One input Handler per channel
 - No handler chains
 - Input and output processing are not directly coupled
- Queuing is done by the framework
 - Input handlers do not enforce queuing policies

Let's Quickly Review

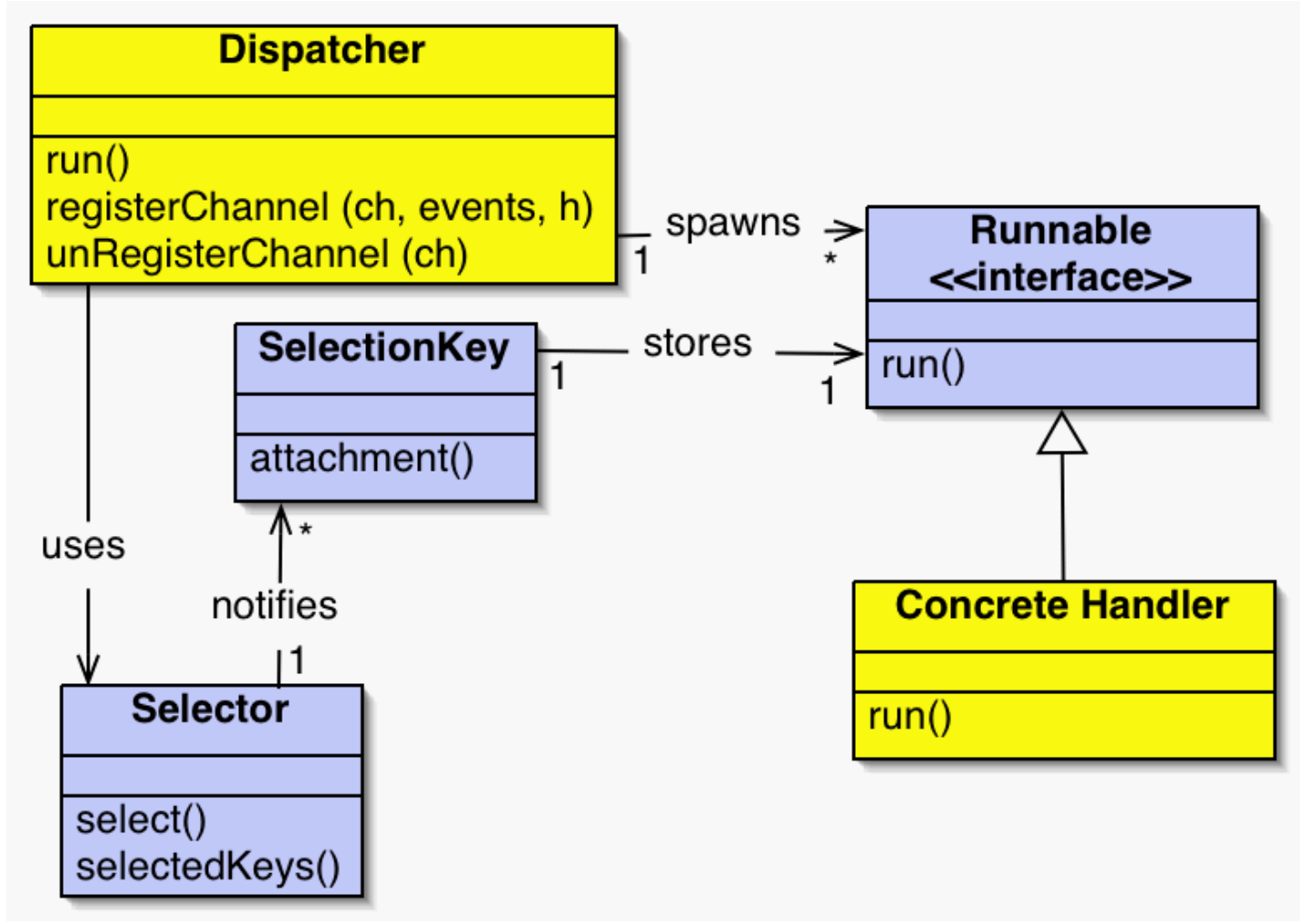
Readiness Selection with NIO

- **Selector (Demultiplexer)**
 - Determines which registered channels are ready
 - Holds a set of keys representing ready channels
 - A ready channel had at least one event occur in the past
 - Events are added but never removed from a key in this set
- **SelectionKey (Handle)**
 - Associates one Selector with one SelectableChannel
 - Tells Selector which events to monitor for the channel
 - Events not in the interest set are ignored
 - Contains the set of events as-of last select() call
 - Ready ops persist until key is removed from the selected set
 - May hold an opaque Object reference for your use

Reactor Pattern Mapped to NIO

- Handle
 - SelectionKey
- Event
 - SelectionKey.OP_READ, etc
- Demultiplexer
 - Selector
- Dispatcher
 - Selector.select() + iterate Selector.selectedKeys()
- Handler
 - In a multi-threaded world, an instance of Runnable

NIO Reactor UML

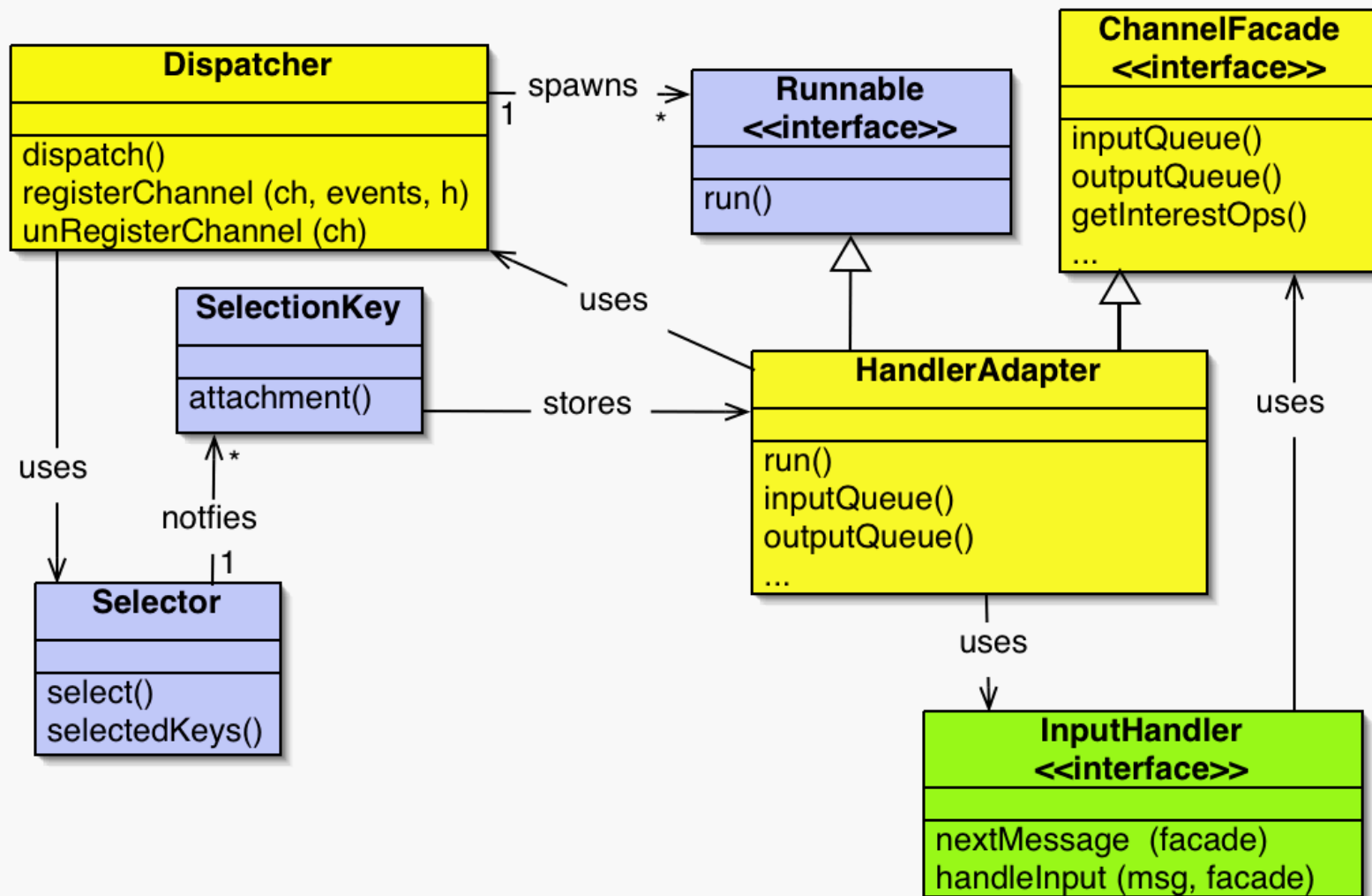


Dispatcher Framework Architecture

Decouple I/O Grunt Work From the Client Handler Logic

- The core Dispatcher invokes an internal adapter class
- Client code registers an InputHandler for a channel
- Adapter instances manage the channel and its queues
 - When new data arrives, adapter asks InputHandler to determine if a full message has arrived yet
 - If so, the message is dequeued and passed to the handler
 - The client handler is passed a Façade interface through which it may interact with the channel and/or queues
- The client InputHandler is decoupled from NIO and Dispatcher implementation details
- The Dispatcher framework is decoupled from any semantics of the data it processes

NIO Reactor as a Framework



Dispatcher Interface

```
public interface Dispatcher
{
    void dispatch() throws IOException;

    Object registerChannel (SelectableChannel channel,
        InputHandler handler)
        throws ClosedChannelException;

    void unregisterChannel (Object key);
}
```



Wrangling Threads

Don't Even **Think** About Writing Your Own Thread Pool

- `java.util.concurrent.Executor`
 - Backport to 1.4 is available
 - Based on Doug Lea's `EDU.oswego.cs.dl.util.concurrent` library
- Executor takes a `Runnable`
 - `Runnable` takes no arguments
- The framework's `HandlerAdapter` class will:
 - Serve as the `Runnable` instance submitted to `Executor`
 - Encapsulate Event state for the worker thread
 - Coordinate hand-off and rendezvous with the `Dispatcher`
 - Contain the input and output queues
 - Present a *Façade* through which the `InputHandler` may interact with the framework (`Dispatcher` and queues)

Core Framework Dispatch Loop

```
public void dispatch()
{
    while (true) {
        synchronized (guard) { /* empty */ }
        selector.select();
        Set<SelectionKey> keys = selector.selectedKeys();

        for (SelectionKey key : keys) {
            HandlerAdapter adapter = (HandlerAdapter)key.attachment();

            invokeHandler (adapter, key);
        }
    }
}
```



Another Quick Diversion...

The Selector Class is Kind of Cranky About Threads

- While a thread is sleeping in `select()`, many `Selector` and `SelectionKey` methods can **block** indefinitely if invoked from another thread
- The trick is to use a **guard object** to handshake
- Selection thread locks then releases the guard
- Other threads wishing to change `Selector` state
 - Obtain a lock on the guard object
 - Wakeup the selector
 - Do whatever (ex: `key.interestOps()`)
 - Release the lock on the guard object

Registering an InputHandler

```
public Object registerChannel (SelectableChannel channel,
    InputHandler handler)
{
    HandlerAdapter adapter =
        new HandlerAdapter (this, queueFactory, handler);
    synchronized (guard) {
        selector.wakeup();
        return channel.register (selector,
            SelectionKey.OP_READ, adapter);
    }
}

class HandlerAdapter implements Runnable, ChannelFacade
{ . . . }
```


Unregistering an InputHandler

```
public void unregisterChannel (Object key)
{
    if ( ! (key instanceof SelectionKey)) {
        throw new IllegalArgumentException ("bad key...");
    }

    SelectionKey selectionKey = (SelectionKey) key;

    synchronized (guard) {
        selector.wakeup ();
        selectionKey.cancel();
    }
}
```

Invoking a Handler in Another Thread

```
private Map<SelectionKey,HandlerAdapter> runningHandlers;  
  
private void invokeHandler (HandlerAdapter adapter,  
    SelectionKey key)  
{  
    synchronized (runningHandlers) {  
        adapter.prepareToRun (key);  
        key.interestOps (0);    // stop selection on channel  
  
        runningHandlers.put (key, adapter);  
    }  
  
    executor.execute (adapter);    // returns immediately  
}
```

While a Thread Is Running

- Channel's interest ops are all disabled
 - Handler cannot be allowed to re-enable them
 - Selector would fire and spawn another thread
 - HandlerAdapter class mediates and buffers changes
- Other threads also must not change interest ops
 - Dispatcher tracks which handlers are running
 - Changes delegated to adapter if handler is running
- Handler could block if it accesses channel or key
 - Relevant Event information is buffered in adapter
 - Handler is never passed a real channel or key

Preparing a Thread to Run

```
class HandlerAdapter implements Runnable, ChannelFacade
```

```
{
```

```
    . . .
```

```
    public void prepareToRun (SelectionKey key)
```

```
    {
```

```
        this.key = key;
```

```
        this.channel = key.channel();
```

```
        this.interestOps = key.interestOps();
```

```
        this.readyOps = key.readyOps();
```

```
    }
```

```
}
```

This runs in the Dispatcher thread before the Handler thread starts, safe to access key object.

Handler Thread Life-Cycle

```
public void run()    // entered by Executor-managed thread
{
    try {
        drainOutput (readyOps);
        fillInput();
        ByteBuffer message = clientHandler.nextMessage (this);
        if (message != null) {
            clientHandler.handleInput (message, this);
        }
        dispatcher.resumeSelection (key, interestOps);
    } catch (Throwable e) {
        dispatcher.unregisterChannel (key);
    }
}
```

First: Drain Any Queued Output

```
private void drainOutput (int readyOps)
{
    if (((readyOps & SelectionKey.OP_WRITE) == 0)
        || outputQueue.isEmpty())
    {
        return;    // cannot write now, or queue is empty
    }

    outputQueue.drainTo (this.channel);

    if (outputQueue.isEmpty()) { // turn off write selection
        interestOps &= ~SelectionKey.OP_WRITE;
    }
}
```

Second: Invoke Client InputHandler

```
public void run()
{
    try {
        drainOutput (readyOps);
        fillInput();
        ByteBuffer message = clientHandler.nextMessage (this);
        if (message != null) {
            clientHandler.handleInput (message, this);
        }
        dispatcher.resumeSelection (key, interestOps);
    } catch (Throwable e) {
        key.cancel();
    }
}
```

A Handler's View of the World

```
interface InputHandler
{
    ByteBuffer nextMessage (ChannelFacade channelFacade);
    void handleInput (ByteBuffer message, ChannelFacade
        channelFacade);
}
interface ChannelFacade
{
    InputQueue inputQueue();
    OutputQueue outputQueue();
    void setHandler (InputHandler handler);
    int getInterestOps();
    void modifyInterestOps (int opsToSet, int opsToReset);
}
```


Last: Cleanup and Resume

```
public void resumeSelection (SelectionKey key, int interestOps)
{
    synchronized (guard) {
        selector.wakeup();
        if (key.isValid()) {
            key.interestOps (interestOps);
        }

        synchronized (runningHandlers) {
            runningHandlers.remove (key);
        }
    }
}
```

This method lives in the Dispatcher object but runs in the Handler thread

A Trivial Echo-Back Handler Example

```
public ByteBuffer nextMessage (ChannelFacade channelFacade)
{
    InputQueue inputQueue = channelFacade.inputQueue();
    int nlPos = inputQueue.indexOf ((byte) '\n');

    if (nlPos == -1) return (null);

    return (inputQueue.dequeueBytes (nlPos));
}

public void handleInput (ByteBuffer message, ChannelFacade channelFacade)
{
    channelFacade.outputQueue().enqueue (message);
}
```

A Few Words About Queues

- The framework need only see trivial interfaces
- Handlers will need more, and perhaps specialized, API methods
- Use Abstract Factory or Builder pattern to decouple queue creation (dependency injection)
- Output queues (usually) must be thread-safe
 - A handler for one channel may need to enqueue data to a different channel
- Input queues usually don't need to be
- Buffer factories need to be thread-safe

Basic Queue Interfaces

```
interface InputQueue
{
    int fillFrom (ReadableByteChannel channel);
}

interface OutputQueue
{
    boolean isEmpty();
    int drainTo (WritableByteChannel channel);
    // enqueue() is not referenced by the framework, but
    // it must enable write selection when data is added
    // write selection is disabled when the queue becomes empty
}
```

Danger! Danger, Will Robinson!



- Never let the Dispatcher thread die
 - Everything will go very quiet
 - Be sure to catch and handle **all** possible throwables
- Beware Executor thread pool policies
 - If “caller runs” is enabled, the Dispatcher thread can execute the handler code—beware alien code
- Put sensible limits on queue sizes
 - Not too small, especially for output
 - Don’t statically allocate per-channel, use factories
 - Don’t over-use direct buffers

Tuning



- Too big a subject to cover here
- Use the knobs in `java.util.concurrent`
 - Optimal thread count is dependent on CPU count
 - Backlog vs. caller runs vs. discard, etc.
- Use buffer factories to obtain space for queues
 - Pool direct buffers, if used, they're expensive to create
 - Heap buffers probably shouldn't be pooled
- Limit policies may be different for input vs. output queues
 - Output limits are typically higher
 - Input limits can be small, network layer queues too

Building an NIO Server

Introduction

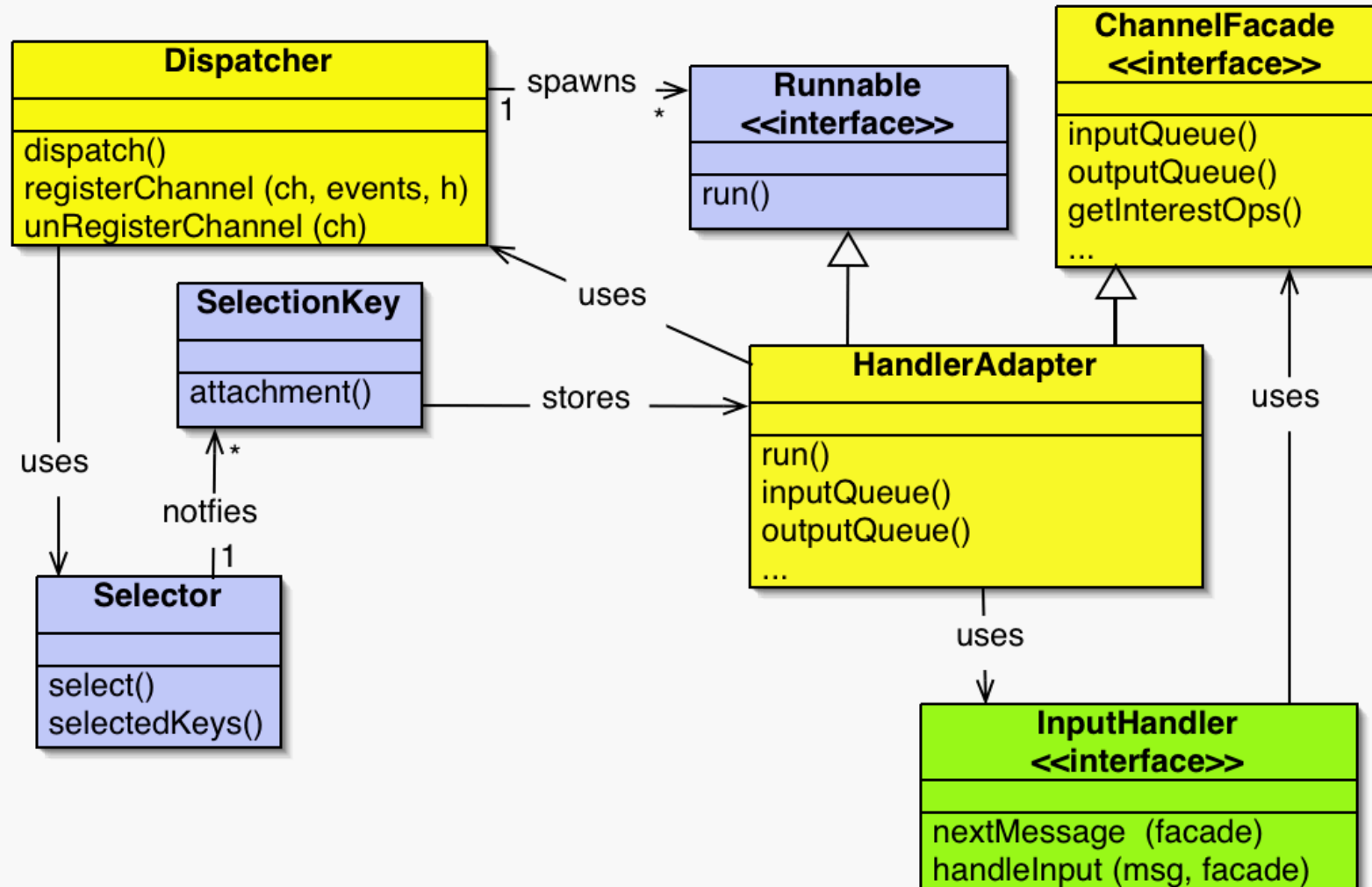
Understanding the problem

Defining a solution

NIO implementation

Summary

A Picture Is Worth...



Summary

- The core of the problem is generic boilerplate
- Decouple application-specific code from generic
- Keep the critical parts lean, efficient and robust
 - Lock appropriately, but sparingly
 - Delegate work to handler threads
 - Protect the framework from alien handler code
- Use good object design and leverage patterns
- Keep it simple

For More Information

Sample Code and Information

- <http://javanio.info>

Books

- *Java NIO*, Ron Hitchens (O'Reilly)
- *Java Concurrency In Practice*, Brian Goetz, et al (AW)
- *Concurrent Programming in Java*, Doug Lea (AW)



Q&A

Ron Hitchens



the
POWER
of
JAVA™

M A R K
LOGIC



JavaOne
Sun and Network on Demand Summit

How to Build a Scalable Multiplexed Server With NIO

Ron Hitchens

Senior Engineer
Mark Logic Corporation
<http://marklogic.com>

President
Ronsoft Technologies
<http://ronsoft.com>



TS-1315