JOIN THE REVOLUTION

How to Build a Scalable Multiplexed Server With NIO Mark II

Ron Hitchens

Senior Engineer

Mark Logic Corporation

San Carlos, CA

ron.hitchens@marklogic.com

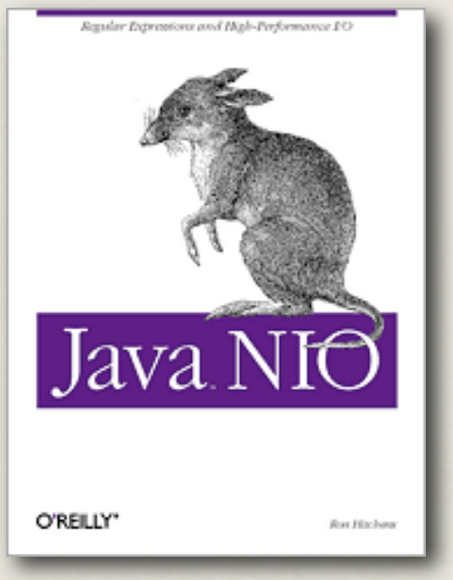
ron@ronsoft.com

SD WEST 2008

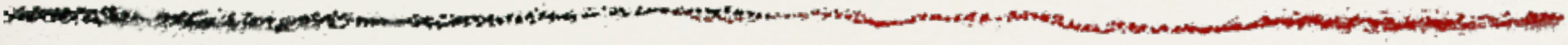
MARCH 3-7, 2008, SANTA CLARA, CA

How to Build a Scalable Multiplexed Server With NIO Mark II

Ron Hitchens
Mark Logic Corporation
March 6, 2008
ron.hitchens@marklogic.com
ron@ronsoft.com



http://javanio.info/ ← *Check for updates to these slides*



Architect a scalable, multiplexed Java Server using the New I/O (NIO) and Concurrency APIs

Ron Hitchens

Years spent hacking UNIX® internals

- Device drivers, I/O streams, etc.

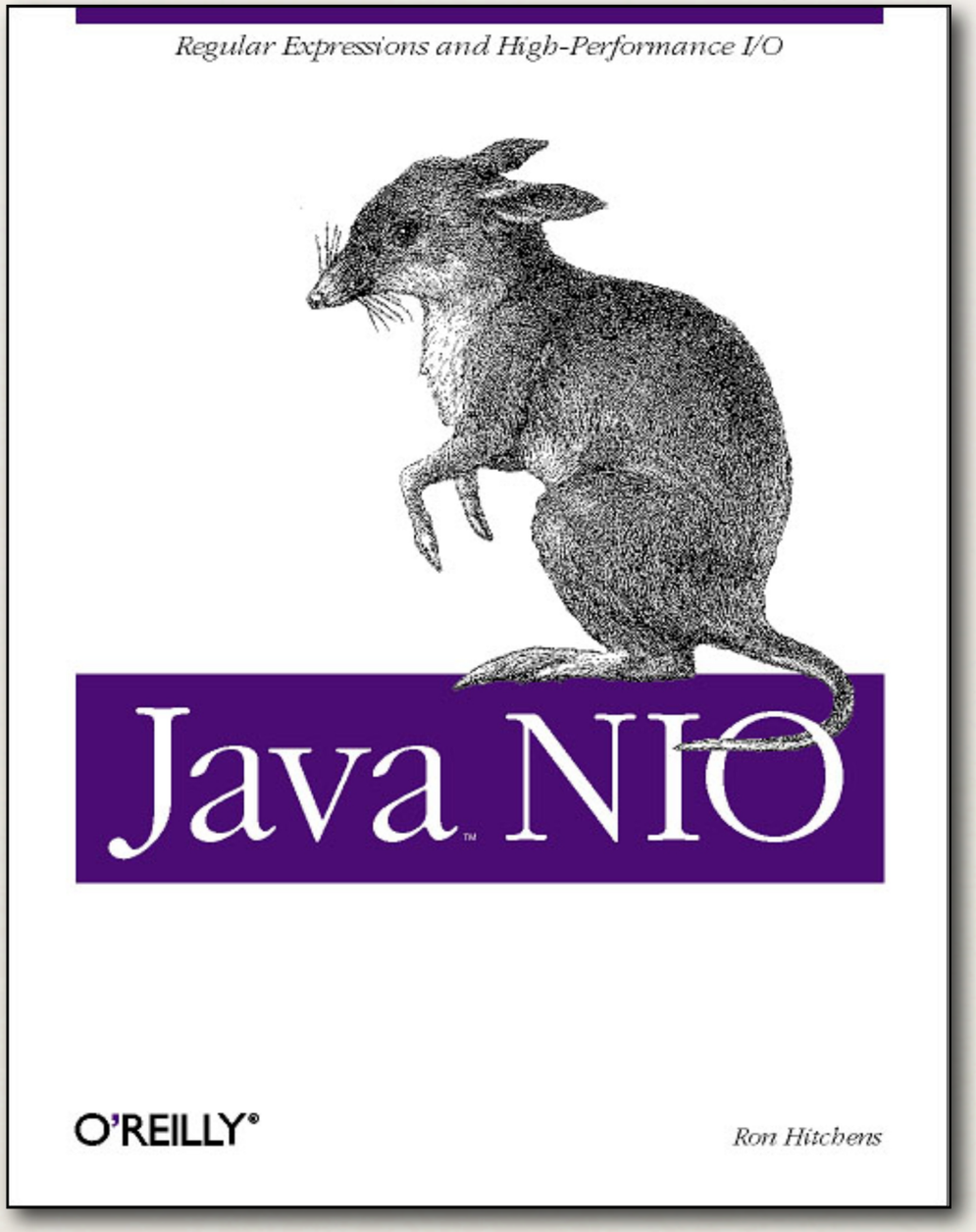
Java NIO published August 2002

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

Getting Started With XQuery (Pragmatic)



Building an NIO Server

Understanding the problem

Defining a solution

An NIO implementation

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
- Must tolerate slow, misbehaving or unresponsive clients

Multiplexing Strategies

- Poll each socket in turn
  - Impractical without non-blocking sockets
  - Inefficient, not fair and scales poorly
- Thread-per-socket
  - Only practical solution with 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

- Multi-threading issues are magnified
  - 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?

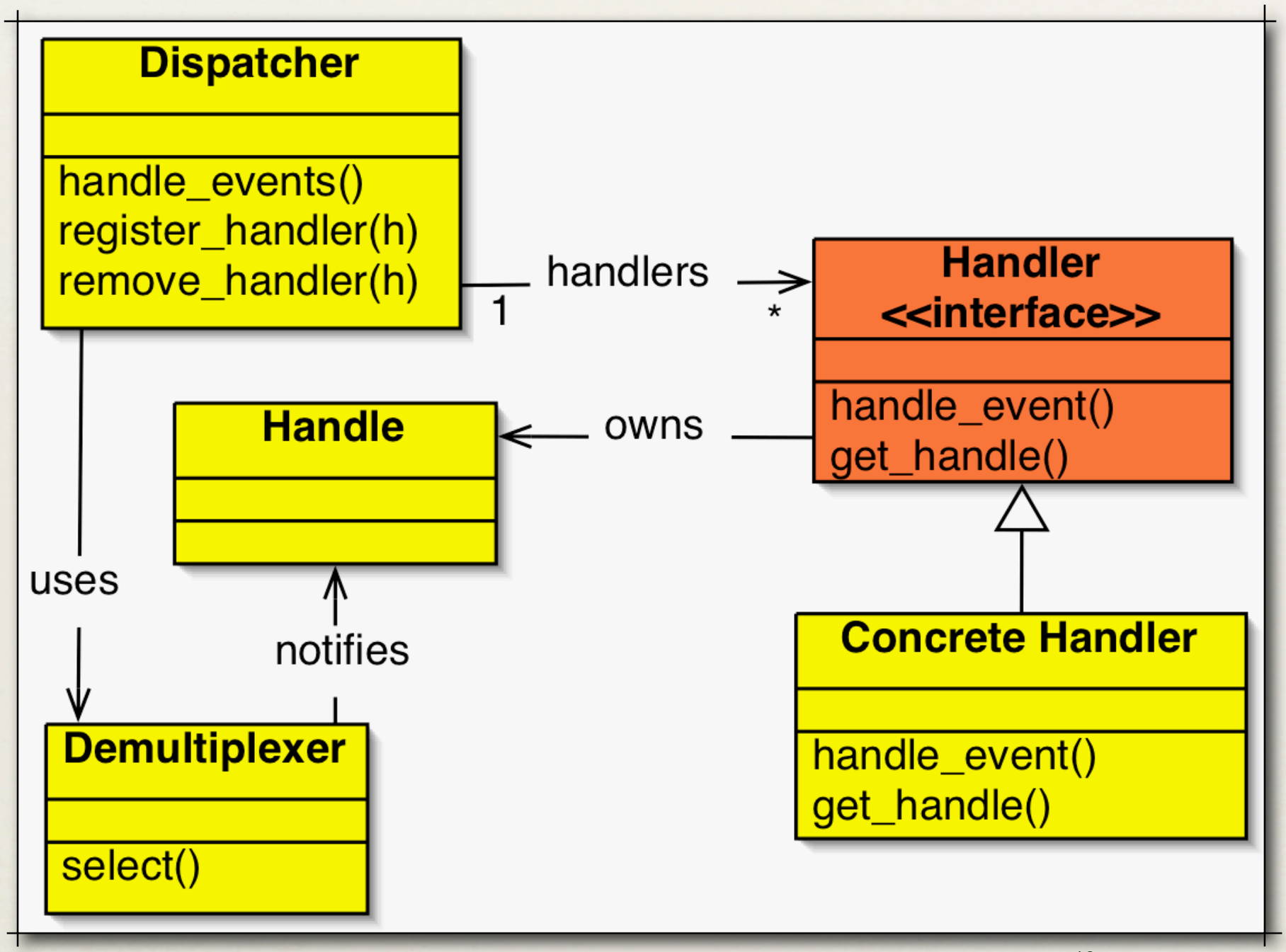
The Reactor Pattern

AKA: Dispatcher, Notifier

- Published in *Pattern Languages of Program Design*, 1995, ISBN 0-201-6073-4
- Paper by Prof. Douglas C. Schmidt *
  - Google for: Reactor Pattern
- Describes the basic elements of a server
- Gives us a vocabulary for this discussion

* http://www.cs.wustl.edu/~schmidt/patterns-ace.html

Reactor Pattern UML



Reactor Pattern Participants

- 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

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
            Schedule handler for execution
            Clear the event for the handle

<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
```

This is the tricky bit

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
    - Perhaps it never will

Observations

- Handling ready-to-write is just buffer draining
  - Handlers should enqueue their output
  - Generic code can drain it
- Reads are non-blocking and may fragment
  - Generic code can fill input queues
- Client handlers process “messages”
  - Handler decides what constitutes a message
- Handler threads interact with the Dispatcher
  - Dispatcher needs to know when handlers finish
  - Handler may want to disable reads, unregister, etc.

Yet Another Framework?

- Yep. Inversion of Control—it's all the rage
- It's mostly generic, common to any server
- Make it once, make it solid, make it reusable

Assumptions

- Our server will be multi-threaded
  - Use the java.util.concurrent package (Java SE 5)
- One select loop (Dispatcher)
  - Accepting new sockets is done elsewhere
  - We only care about 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

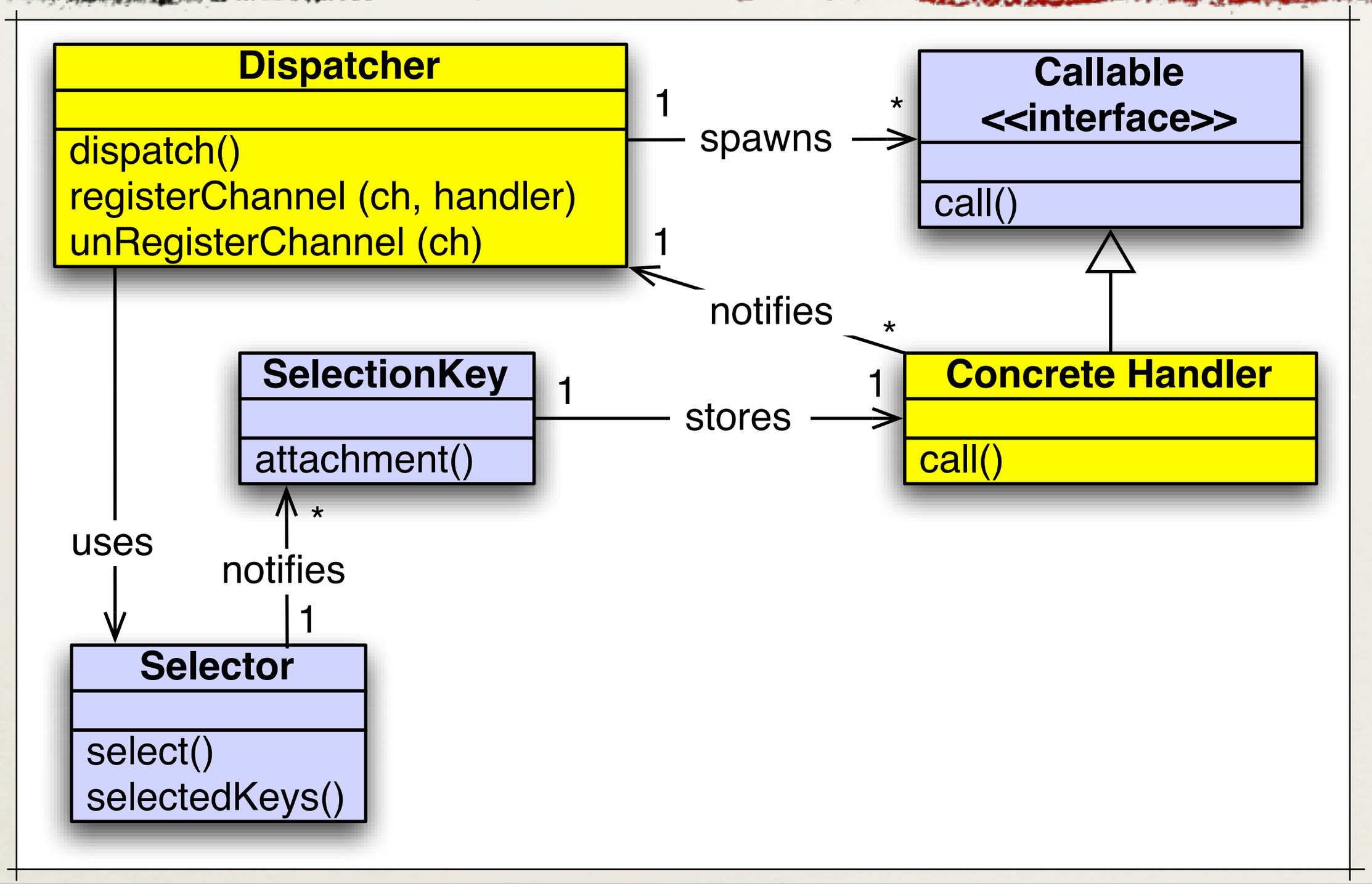
Readiness Selection with NIO

- Selector (Demultiplexer)
  - Holds a Set of keys representing ready channels
    - This is the "selected set" of keys
    - Events are added to but never removed from a key in this set
- SelectionKey (Handle)
  - Associates a Selector with a SelectableChannel
  - Holds set of events of interest for the channel
    - Events not in the interest set are ignored
  - Holds a set of triggered events as-of last select() call
    - Events 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
  - An instance of Runnable or Callable

NIO Reactor UML

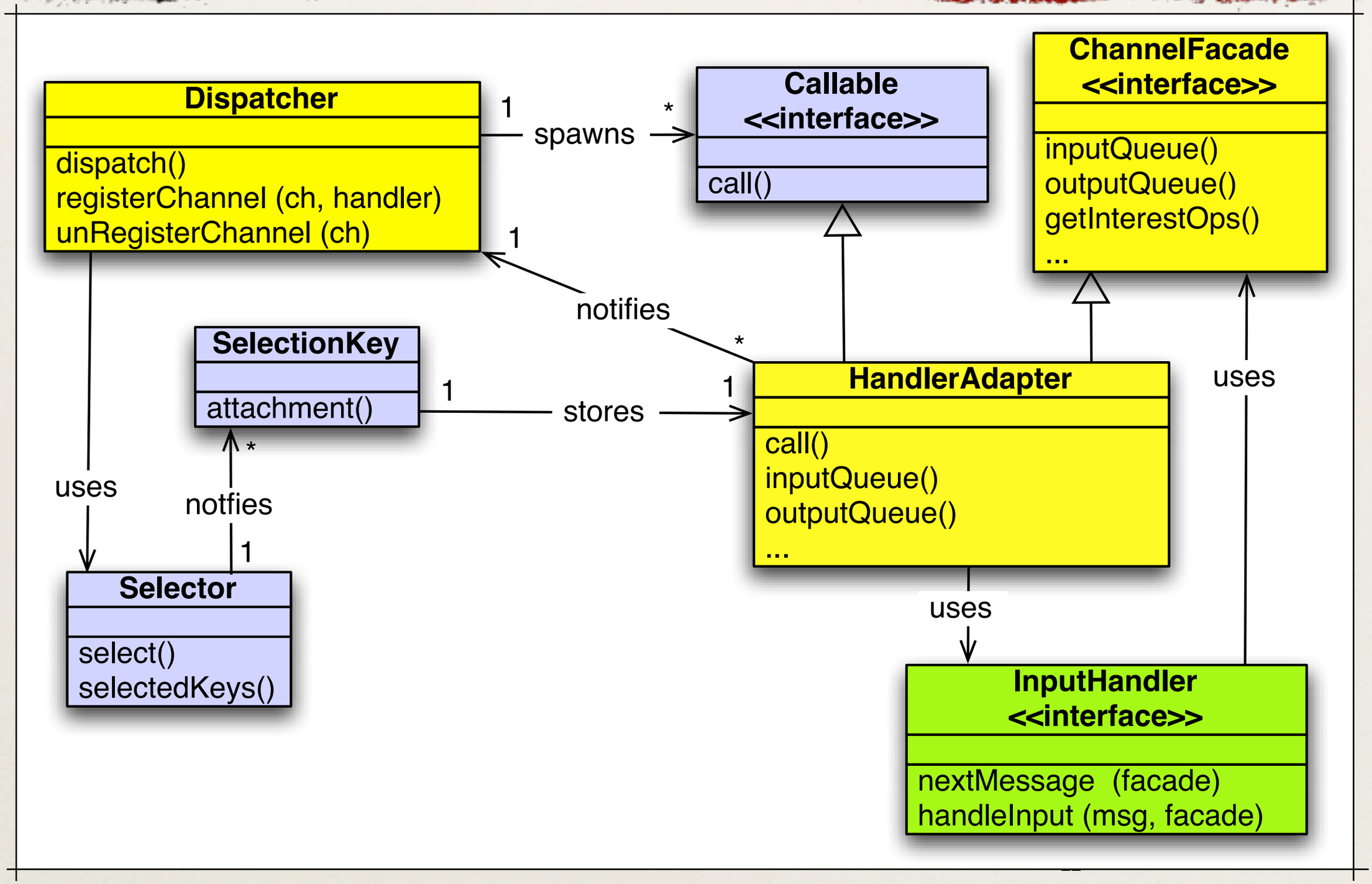


Dispatcher Framework Architecture

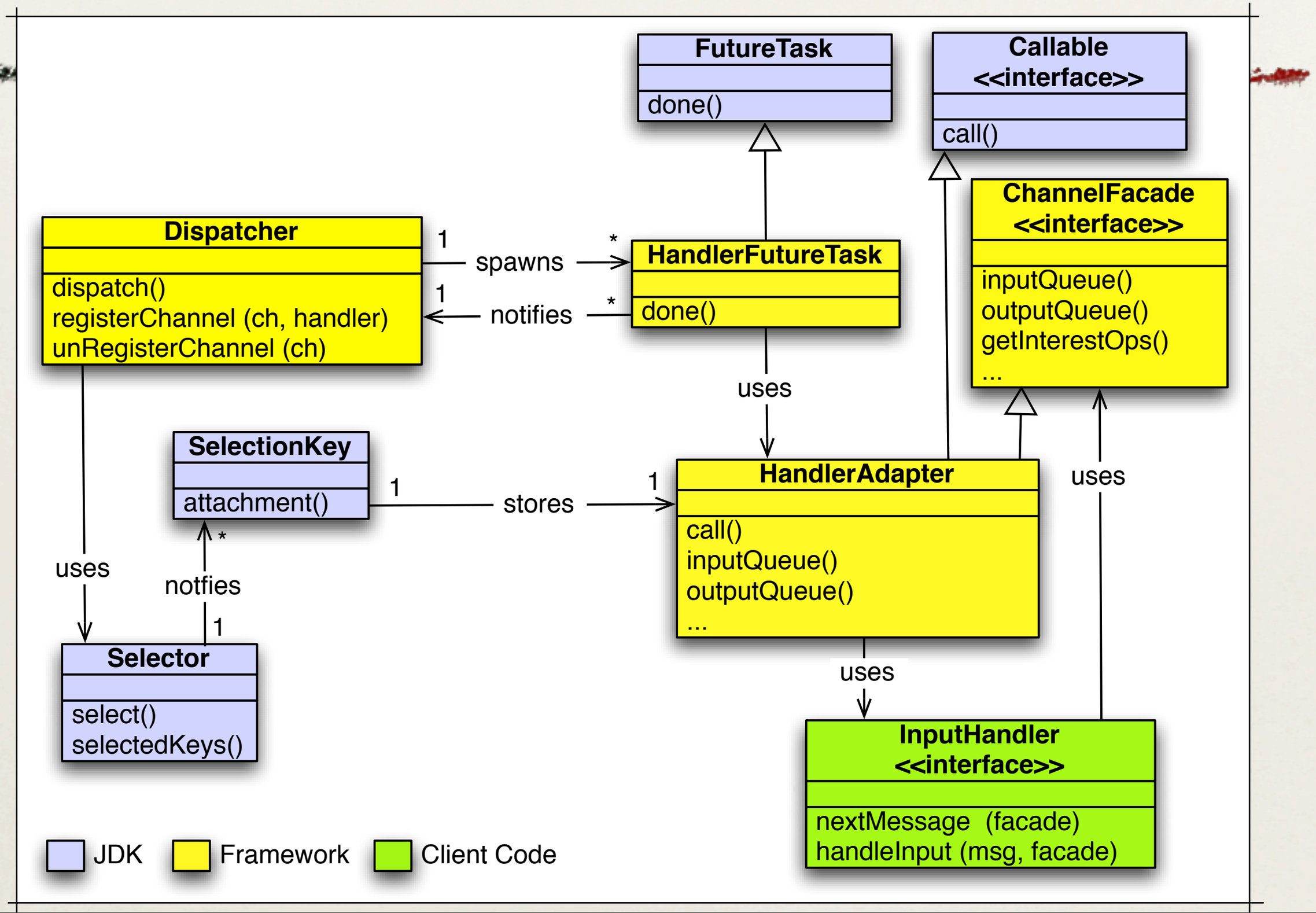
Decouple I/O Grunt Work From the Client Handler Logic

- Client code registers an InputHandler for a channel
- Dispatcher wraps handler in an internal adapter class
- Adapter instances manage the channel and its queues
  - When new data arrives, adapter asks InputHandler to determine if a full message has arrived
  - If so, the message is dequeued and passed to the handler
  - The client handler is passed a ChannelFacade 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



An Even Better NIO Framework



Dispatcher Interface

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

    ChannelFacade registerChannel (
       SelectableChannel channel,
       InputHandler handler)throws IOException;


    void unregisterChannel (ChannelFacade key);
}
```

Wrangling Threads

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

- java.util.concurrent.Executor
  - Backport to 1.4 is available
- Executor takes a Callable
  - Callable takes no arguments but has a return type and may throw exceptions
- The framework’s HandlerAdapter class will:
  - Serve as the Callable that Executor will run
  - 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

Core Dispatcher Loop

```
public void dispatch()
{
    while (true) {
        selectorGuardBarrier();
        selector.select();
        checkStatusChangeQueue (statusChangeQueue);
        Set<SelectionKey> keys = selector.selectedKeys();
        for (SelectionKey key : keys) {
            HandlerAdapter adapter = (HandlerAdapter)key.attachment();
            invokeHandler (adapter);
        }
        keys.clear();
    }
}
```

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 a different thread
- Use a guard object to handshake
- Selection thread grabs then releases the guard
- Other threads wishing to change Selector state
  - Lock the guard object
  - Wakeup the selector
  - Do whatever (eg: `key.interestOps()`)
  - Release the guard lock

Reader/Writer Lock Barrier

- ReadWriteLock: Improvement over synchronized
- Worker threads acquire read locks
  - Multiple may be granted at once
- Selection thread acquires write lock
  - Must wait for all read locks to be released
- Lets multiple handler threads complete in one Selector wakeup cycle
  - Handler threads hold their lock for a very short time
  - Re-pooling threads quickly pool improves efficiency
- Need to take greater care managing lock state
  - Locks must be explicitly released

Selection Guard Implementation

From NioDispatcher.java

```
import java.util.concurrent.locks.ReadWriteLock;

private final ReadWriteLock selectorGuard =
     new ReentrantReadWriteLock();

private void selectorGuardBarrier()
{
     selectorGuard.writeLock().lock();     // may wait here for readers
     selectorGuard.writeLock().unlock();   // allow readers
}

private void acquireSelectorGuard()
{
     selectorGuard.readLock().lock();     // close Selector barrier
     selector.wakeup();                   // wake Selector if sleeping
}

private void releaseSelectorGuard()
{
     selectorGuard.readLock().unlock();  // release my hold on barrier
}
```

Registering an InputHandler

```
public ChannelFacade registerChannel (
   SelectableChannel channel, InputHandler handler)
{
   HandlerAdapter adapter = new HandlerAdapter (handler, this,
    bufferFactory);
   acquireSelectorGuard();

   try {
       SelectionKey key = channel.register (selector,
           SelectionKey.OP_READ, adapter);
       adapter.setKey (key);
       return adapter;
   } finally {
       releaseSelectorGuard();
   }
}
```

```
class HandlerAdapter implements Callable<HandlerAdapter>,
   ChannelFacade
{ . . . }
```

Unregistering an InputHandler

```
public void unregisterChannel (ChannelFacade token)
{
   if ( ! (token instanceof HandlerAdapter)) {
        throw new IllegalArgumentException ("bad registration…");
   }

   HandlerAdapter adapter = (HandlerAdapter) token;
   SelectionKey selectionKey = adapter.key();

   acquireSelectorGuard();

   try {
        selectionKey.cancel();
   } finally {
        releaseSelectorGuard();
   }
}
```

While a Worker Thread Is Running

- Channel's interest ops are all disabled
  - Handler cannot be allowed to re-enable them
    - Selector would fire and spawn another handler thread
  - HandlerAdapter class mediates and buffers changes
- Other threads must not change key's interest ops
  - Always use ChannelFacade, it buffers if needed
- Handler could block if it accesses channel or key
  - Handler is never passed a real channel or key
  - Event information is buffered in adapter
  - Interest op changes are buffered for later

Invoking a Handler in Another Thread

```
private void invokeHandler (HandlerAdapter adapter)
{
    adapter.prepareToRun (key);
    adapter.key().interestOps (0); // stop selection on channel
    executor.execute (new HandlerFutureTask (adapter));
}
```

More about this shortly

Preparing a Handler to Run

```
class HandlerAdapter implements Callable<HandlerAdapter>,
   ChannelFacade
{
    private volatile boolean running = false;
    private final Object stateChangeLock = new Object();
. . .

void prepareToRun()      // package local
{
    synchronized (stateChangeLock) {
        interestOps = key.interestOps();
        readyOps = key.readyOps();
        running = true;
    }
}
```

Handler Thread Life-Cycle

```
public HandlerAdapter call() throws IOException
{
   try {
        drainOutput();
        fillInput();

        ByteBuffer msg;
        while ((msg = clientHandler.nextMessage (this)) != null) {
            clientHandler.handleInput (msg, this);
        }
   } finally {
        synchronized (stateChangeLock) { running = false; }
   }
   return this;
}
```

First: Manage The Queues

```
private void drainOutput() throws IOException
{
   if (((readyOps & SelectionKey.OP_WRITE) != 0)
        && ( ! outputQueue.isEmpty()))
   {
        outputQueue.drainTo ((ByteChannel) channel);
   }

   if (outputQueue.isEmpty()) {
        disableWriteSelection();
        if (shuttingDown) {          // set by fillInput on EOS
            channel.close();
        }
   }
}
```

Similar logic for `fillInput()`, see example code

Second: Invoke Client InputHandler

```
public HandlerAdapter call() throws IOException
{
   try {
        drainOutput();
        fillInput();

        ByteBuffer msg;
        while ((msg = clientHandler.nextMessage (this)) != null) {
            clientHandler.handleInput (msg, this);
        }
   } finally {
        synchronized (stateChangeLock) { running = false; }
   }
   return this;
}
```

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);
}
```

Handler Is Wrapped In FutureTask

java.util.concurrent.FutureTask

- Overrides done() method
  - Called after return from call() in HandlerAdapter
  - Appends itself to a BlockingQueue
  - Wakes the selection thread
  - Worker thread returns to the Executor pool
- Selection thread drains the queue each time around
- For each queued HandlerAdapter
  - If the connection has terminated, unregister handler
  - Otherwise, the interest set is updated with the new value buffered in the adapter object

HandlerFutureTask Class

```
private class HandlerFutureTask extends FutureTask<HandlerAdapter>
{
   private final HandlerAdapter adapter;  // Stored by constructor

   protected void done()
   {
        enqueueStatusChange (adapter);   // notify selection thread

        try {
            get();           // Get result or throw deferred exception
        } catch (ExecutionException e) {
            adapter.die();           // selection thread will drop it
        }
   }
}
```

Finally: Reap Completed Handlers

```
public void dispatch()
{
    while (true) {
        selectorGuardBarrier();
        selector.select();
        checkStatusChangeQueue();
        Set<SelectionKey> keys = selector.selectedKeys();
        for (SelectionKey key : keys) {
            HandlerAdapter adapter = (HandlerAdapter)key.attachment();
            invokeHandler (adapter);
        }
        keys.clear();
    }
}
```

Cleanup Completed Handlers

```
private void checkStatusChangeQueue()
{
   HandlerAdapter adapter;
   while ((adapter = statusChangeQueue.poll()) != null) {
        if (adapter.isDead())
            unregisterChannel (adapter);
        else
            resumeSelection (adapter);
   }
}

private void resumeSelection (HandlerAdapter adapter)
{
   SelectionKey key = adapter.key();
   if (key.isValid()) key.interestOps (adapter.getInterestOps());
}
```

Running in the selection thread, no need for the guard lock

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 want to add data to a different channels queue
- Input queues usually don’t need to be
- Buffer factories need to be thread-safe

Basic Queue Interfaces

```
interface InputQueue
{
    int fillFrom (ReadableByteChannel channel);
    boolean isEmpty();
    int indexOf (byte b);
    ByteBuffer dequeueBytes (int count);
    void discardBytes (int count);
}

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

Only methods in blue are used by the framework

Trivial Echo-Back Handler Example

Implementation of InputHandler

```
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);
}
```

Less Trivial Chat Server Example

```
public ByteBuffer nextMessage (ChannelFacade channelFacade)
{
   InputQueue inputQueue = channelFacade.inputQueue();
   int nlPos = inputQueue.indexOf ((byte) '\n');
   if (nlPos == -1) return null;
   if ((nlPos == 1) && (inputQueue.indexOf ((byte) '\r') == 0)) {
        inputQueue.discardBytes (2);   // eat CR/NL by itself
        return null;
   }
   return (inputQueue.dequeueBytes (nlPos + 1));
}

public void handleInput (ByteBuffer message, ChannelFacade facade)
{
   protocol.handleMessage (channelFacade, message);
}
```

Chat Server Continued

```
public void handleMessage (ChannelFacade facade, ByteBuffer message)
{
   broadcast (users.get (facade), message);
}

private void broadcast (NadaUser sender, ByteBuffer message)
{
   synchronized (users) {
        for (NadaUser user : users.values()) {
            if (user != sender) {
                sender.sendTo (user, message);
            }
        }
   }
}
```

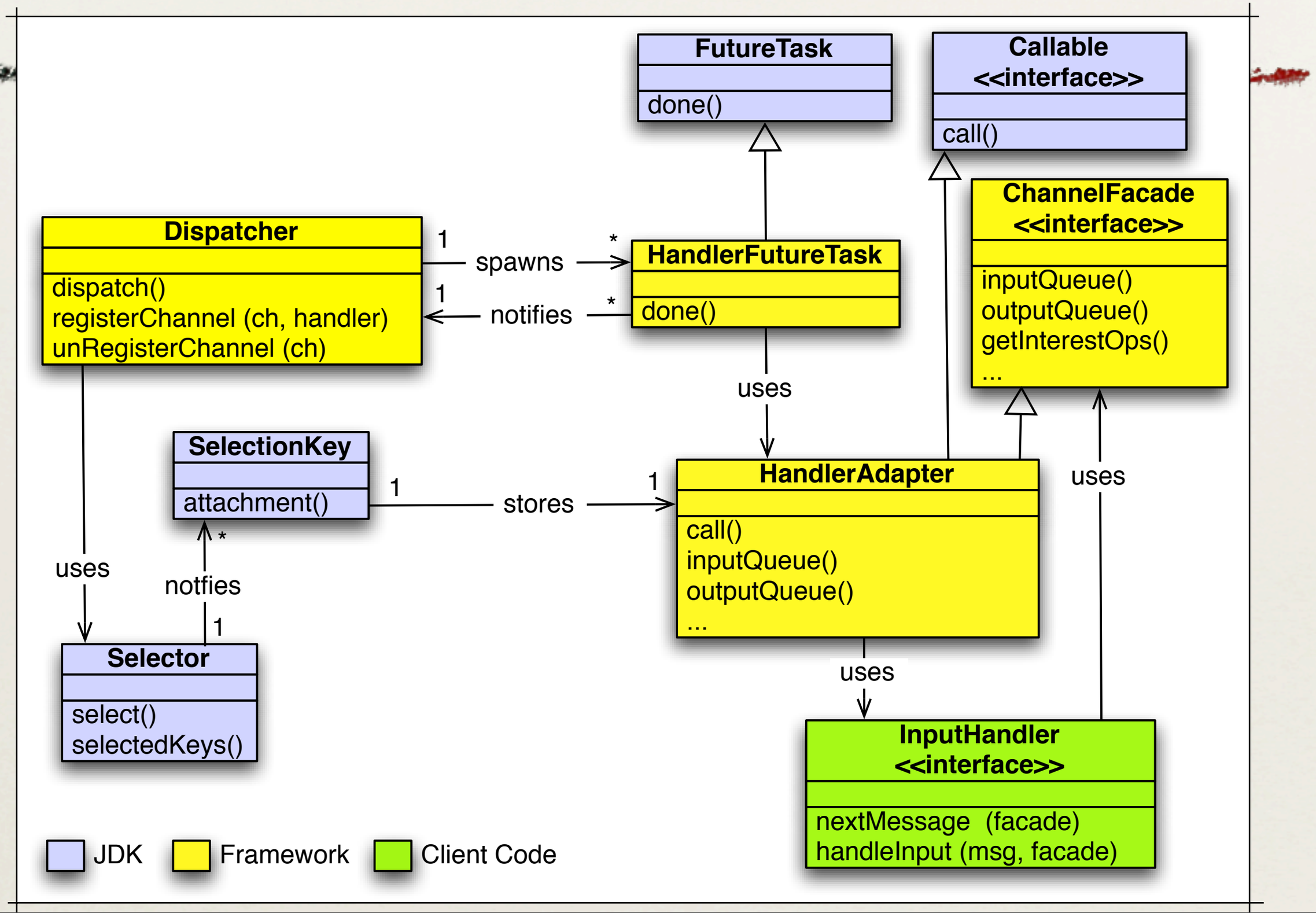
Danger! Danger, Will Robinson!

- Never let the Dispatcher thread die
  - Everything will go very quiet
  - Be sure to catch and handle **all** possible throwables
    - This is done by the HandlerFutureTask class
- Beware Executor thread pool policies
  - If “caller runs” is enabled, the Dispatcher thread can execute the handler code—that’s not good
- 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 and levers in java.util.concurrent
  - Optimal thread count is dependent on CPU/core 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, the network layer queues too

A Picture Is Worth…



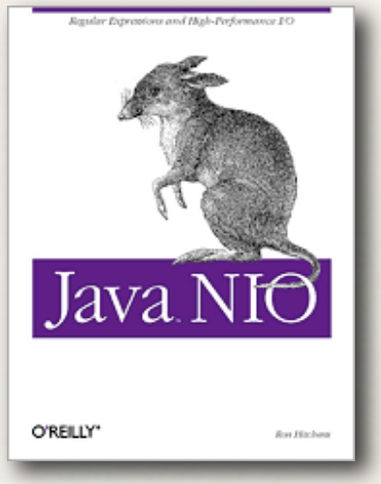
Summary

- 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

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
ron@ronsoft.com
http://javanio.info/

JOIN THE REVOLUTION

How to Build a Scalable Multiplexed Server With NIO Mark II

Ron Hitchens

Senior Engineer

Mark Logic Corporation

San Carlos, CA

ron.hitchens@marklogic.com

ron@ronsoft.com

SD WEST 2008

MARCH 3-7, 2008, SANTA CLARA, CA