The Proactor Pattern

António Sousa

UU

30 de Setembro de 2005

Outline

1. Intro
   - Proactor?
   - In detail...
2. Making things work
   - JAVA!
   - Some code
3. Verdict
   - Consequences
   - Finale

Dictionary definition

proactive

adjective
(of a person, policy or action) creating or controlling a situation by causing something to happen rather than responding to it after it has happened: be proactive in identifying and preventing potential problems

My definition

proactor

pattern
allows an efficient use of the asynchronous mechanisms provided by operating systems in event-driven applications, by integrating the demultiplexing of completion events and the dispatching of their event handlers;

A little background

- Asynchronous I/O
  - Operations run in the background and do not block user applications (libaio, linux kernel 2.6, Windows NT)
- Completion events
  - When an asynchronous I/O operation finishes a completion event is generated saying that the operation is completed (or not)
- Event handlers
  - The application-specific component that processes the results of the asynchronous operations

Why do we need this?

In multithreaded or reactive concurrency designs we have:

- Synchronization complexity
- Performance overhead (context switching)
- Complex programming

Proactor Pattern!

Performance in concurrency without its liabilities

Participants

Asynchronous operations

Potentially long-duration operations implementing some kind of service (invoked on a handle) that executes without blocking

Completion handlers

Interface defining the hook methods for processing the completion events generated by asynchronous operations

Asynchronous operation processor

Executes the asynchronous operations to completion generating and queuing the corresponding completion events

Participants

Completion event queue

Buffer where the asynchronous operation processor queues the completion events associated with a specific handle

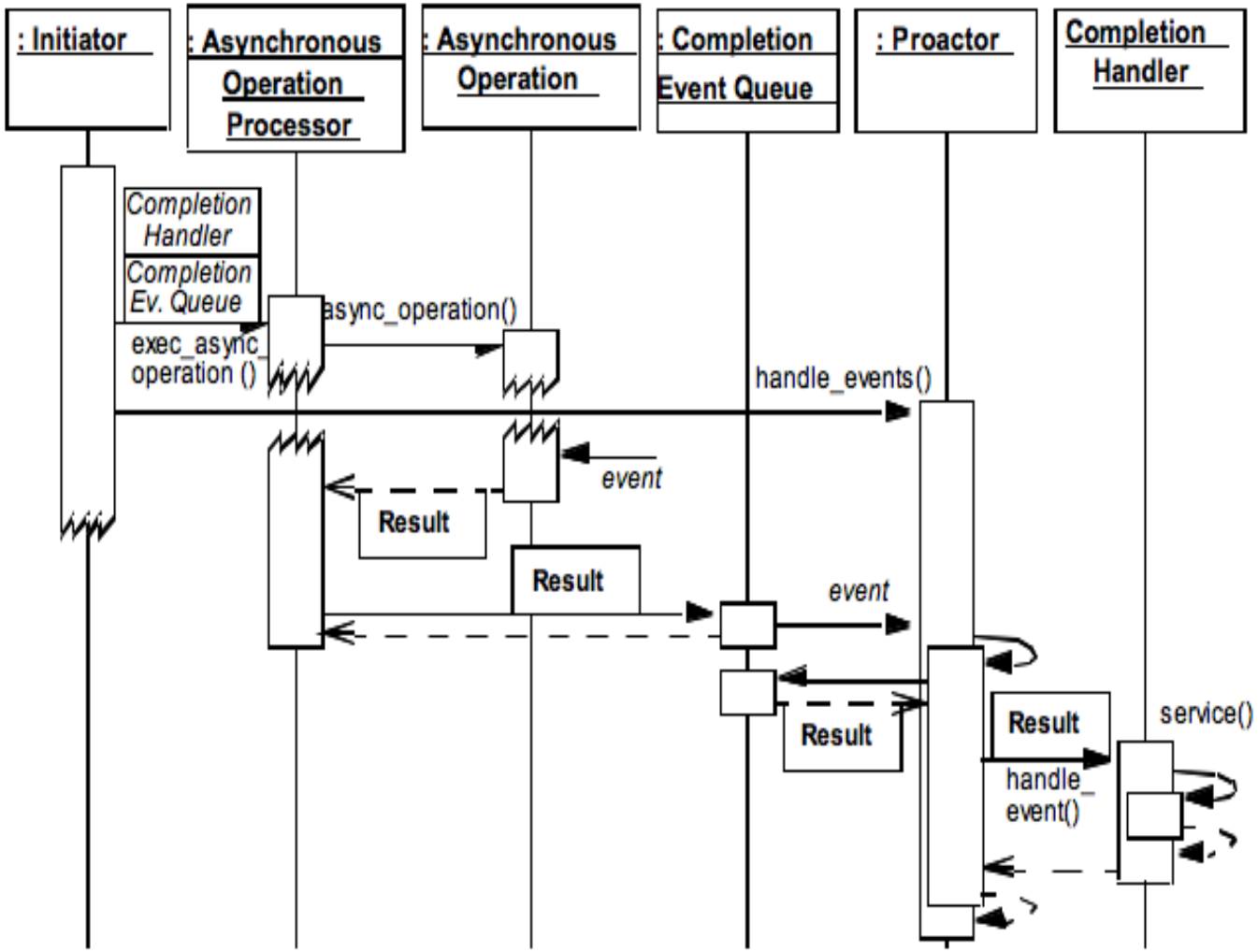
Completion Dispatcher (Proactor)

Makes use of the asynchronous event demultiplexer to dequeue completion event and then dispatches them to the correspondent hook method of the completion handler

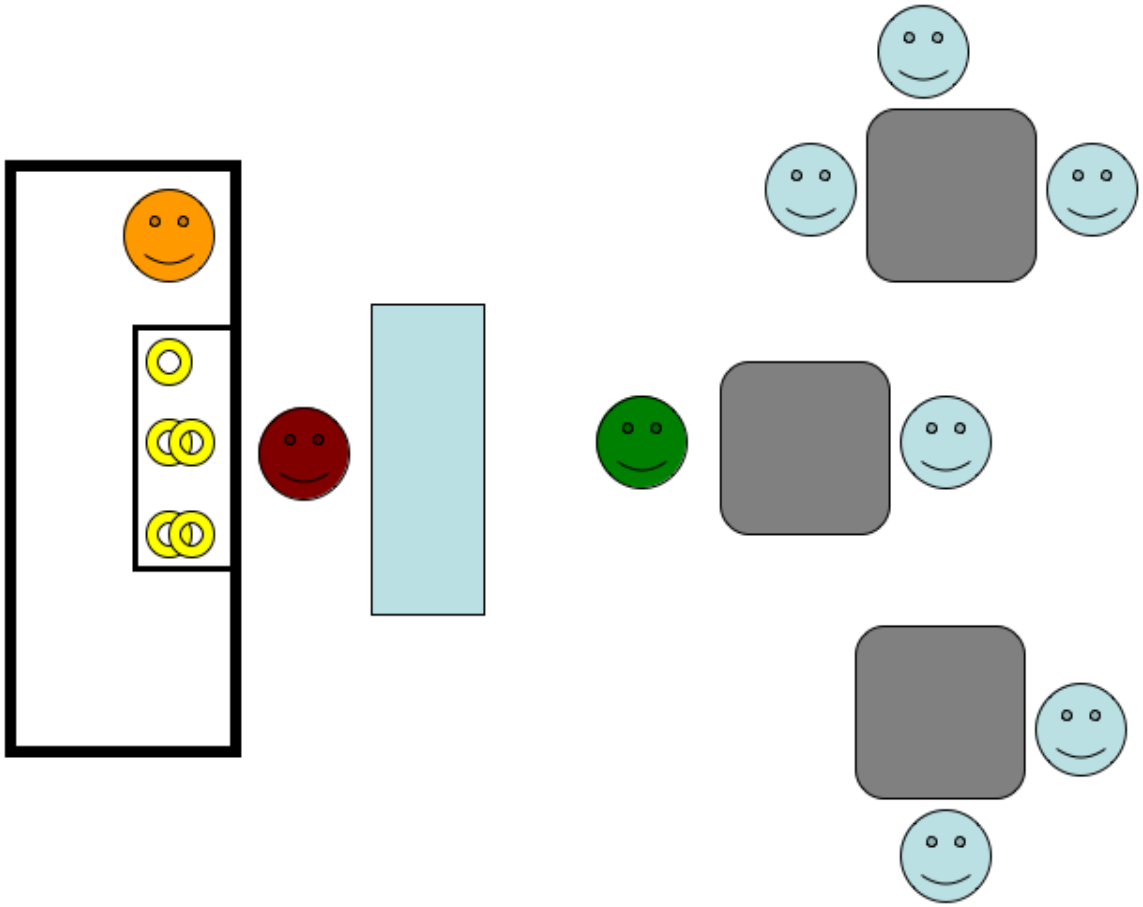
Initiator

Any entity in the application that invokes an asynchronous operation

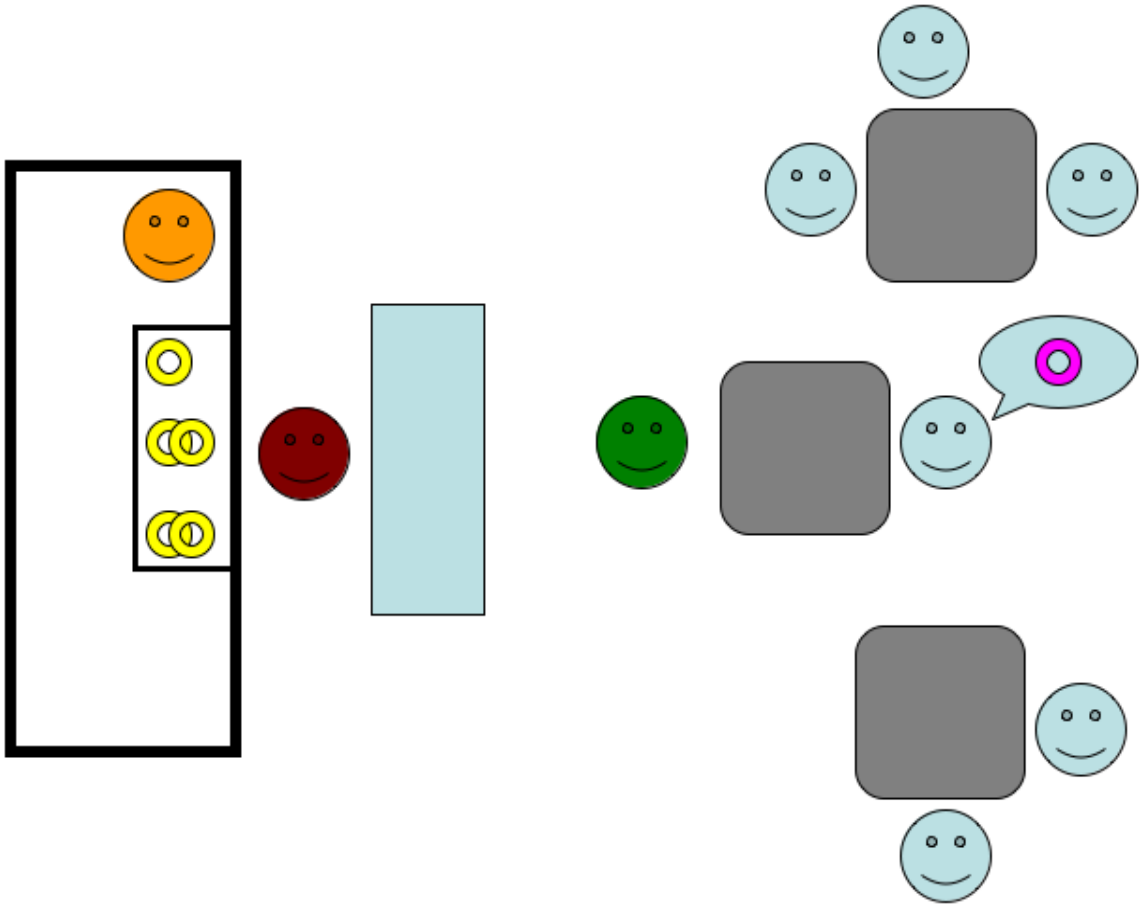
Collaborations



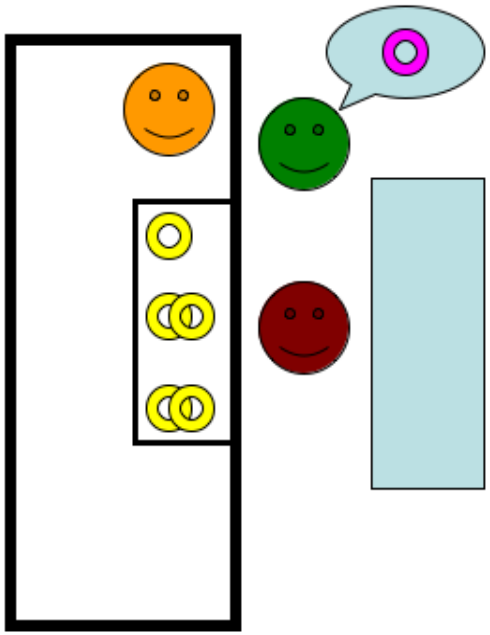
A little analogy

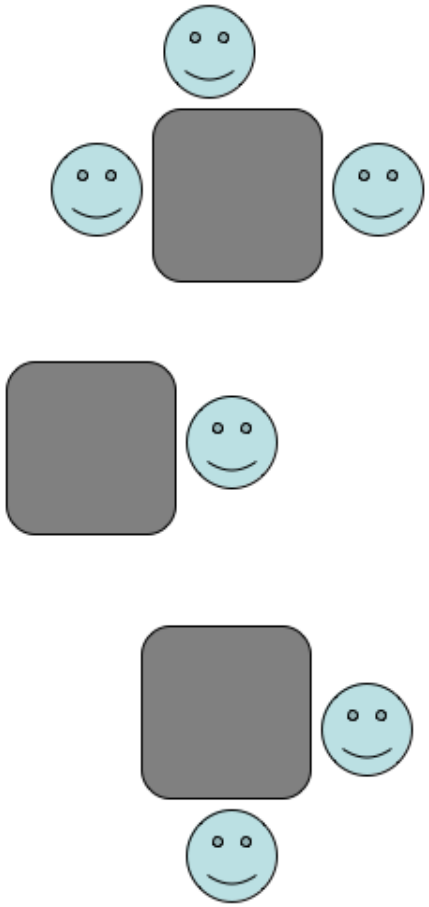


A little analogy

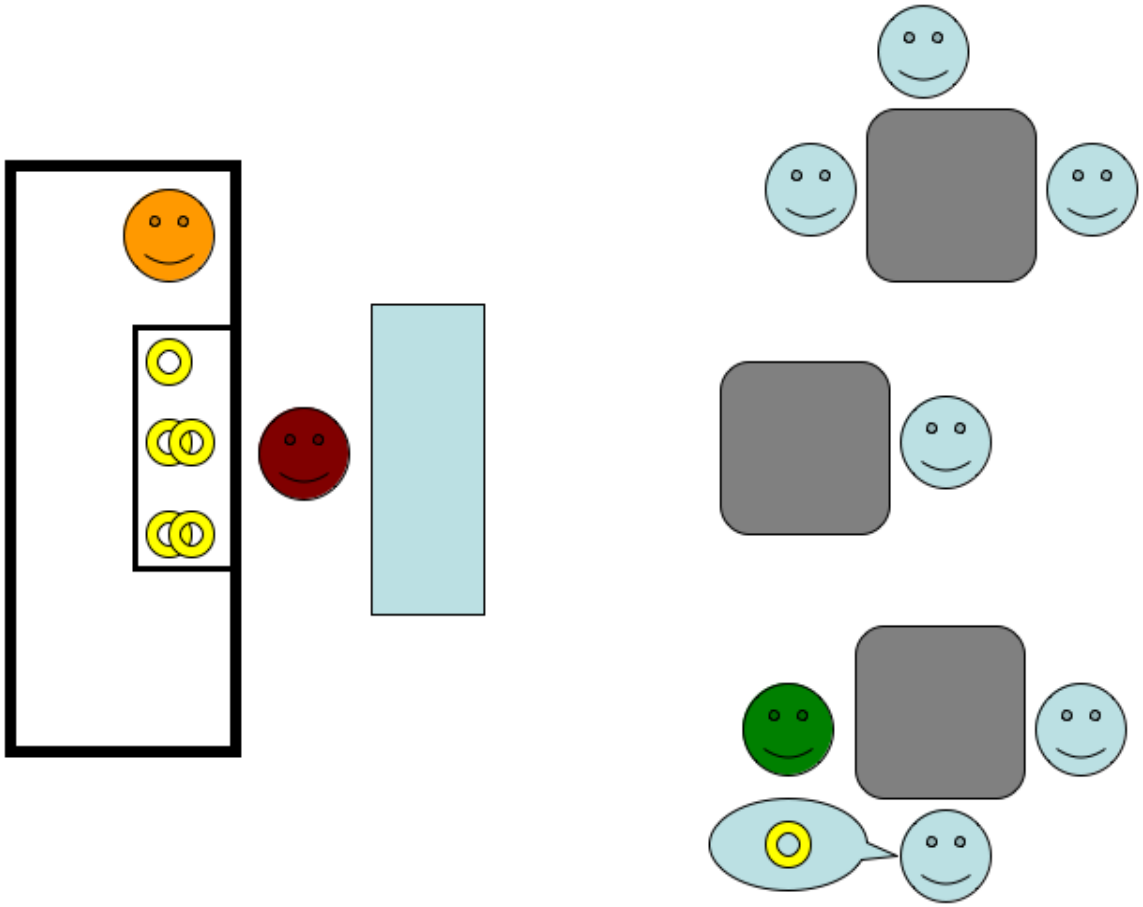


A little analogy

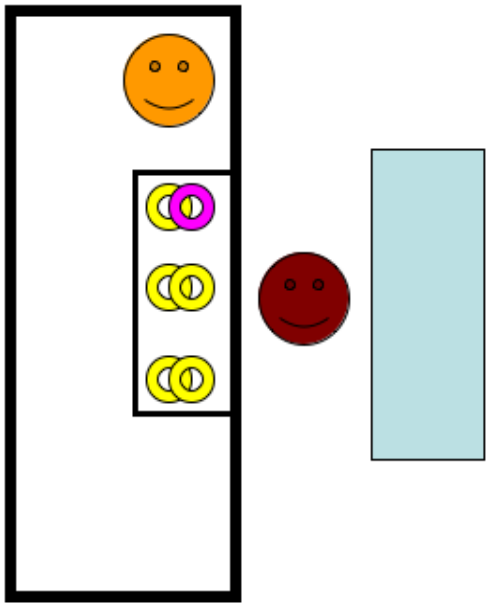


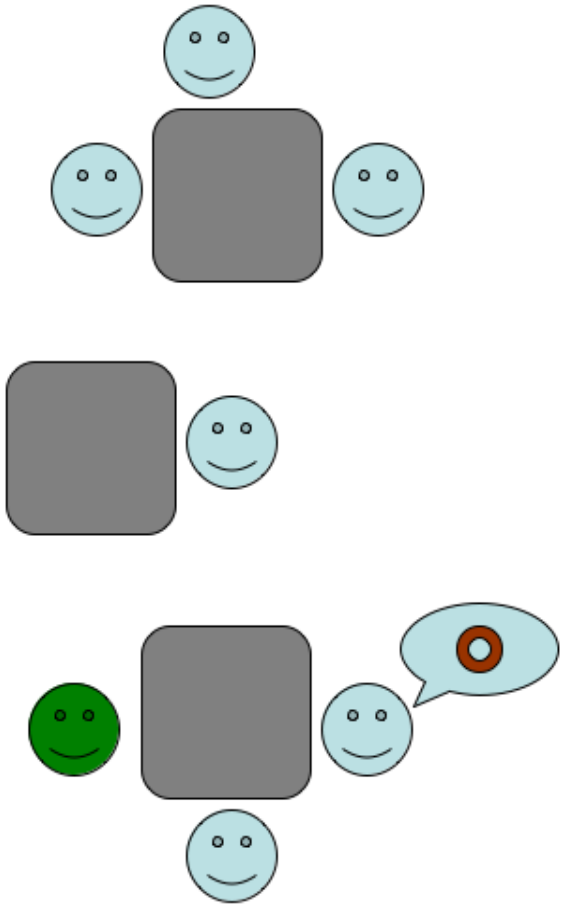


A little analogy

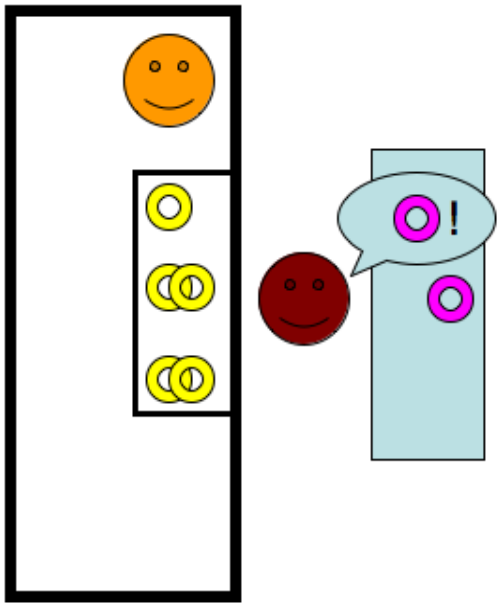


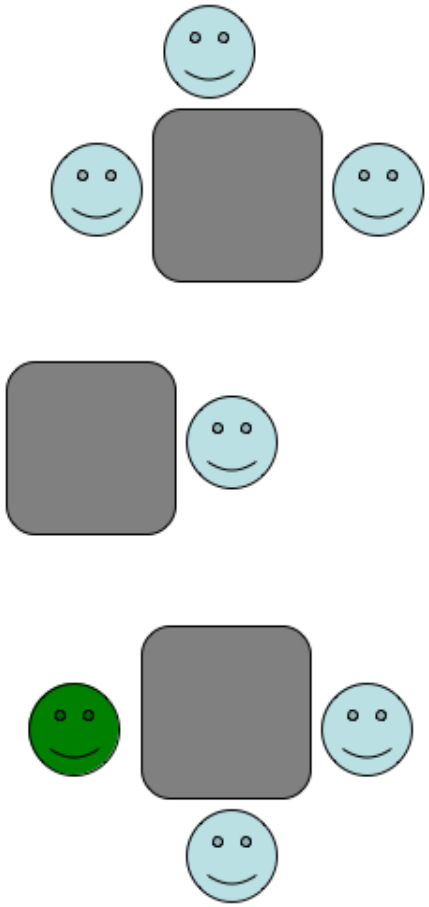
A little analogy



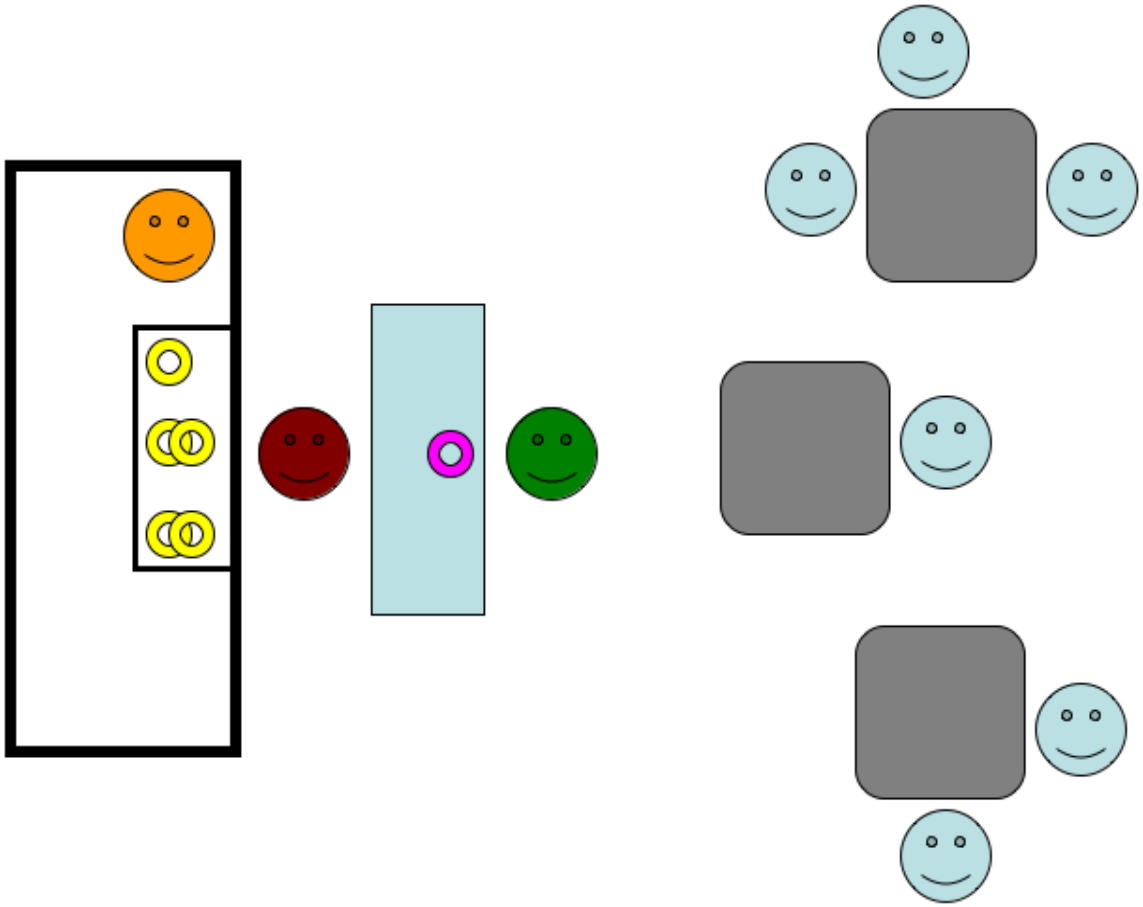


A little analogy

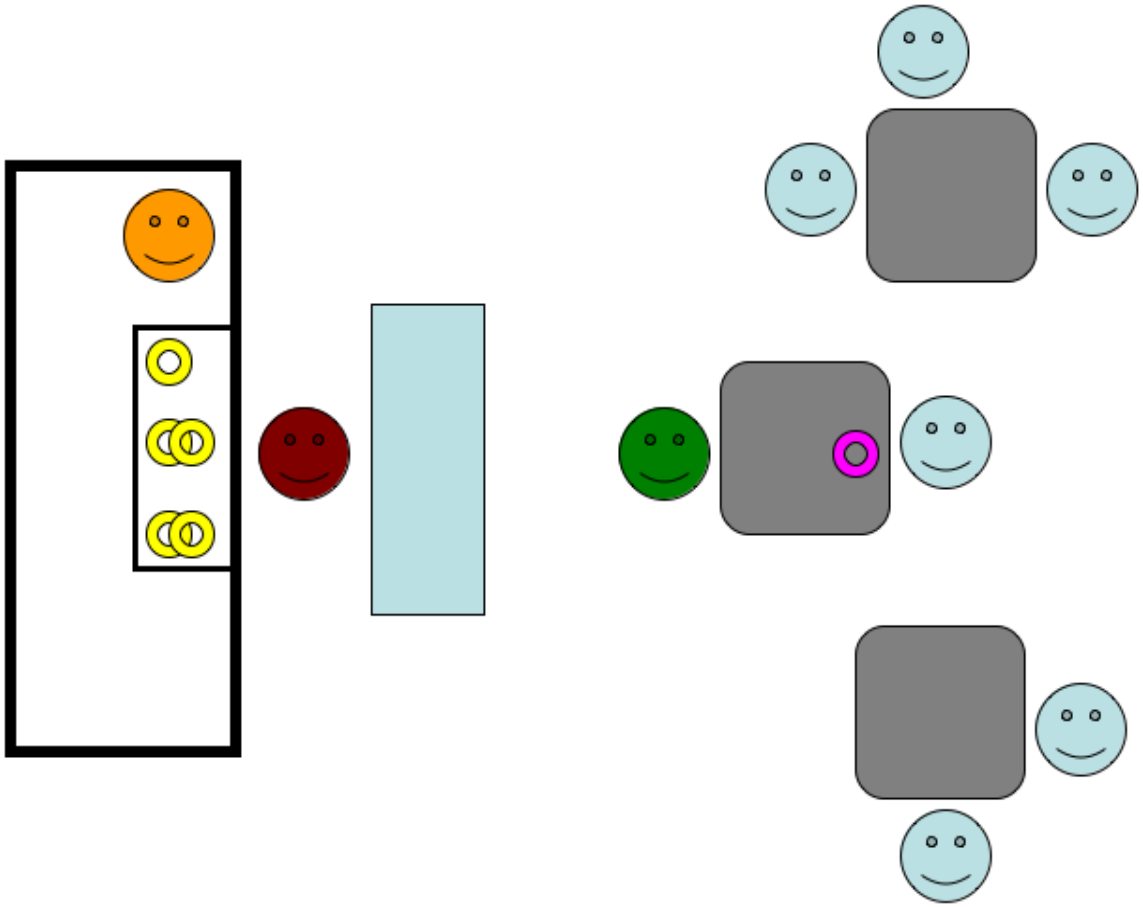




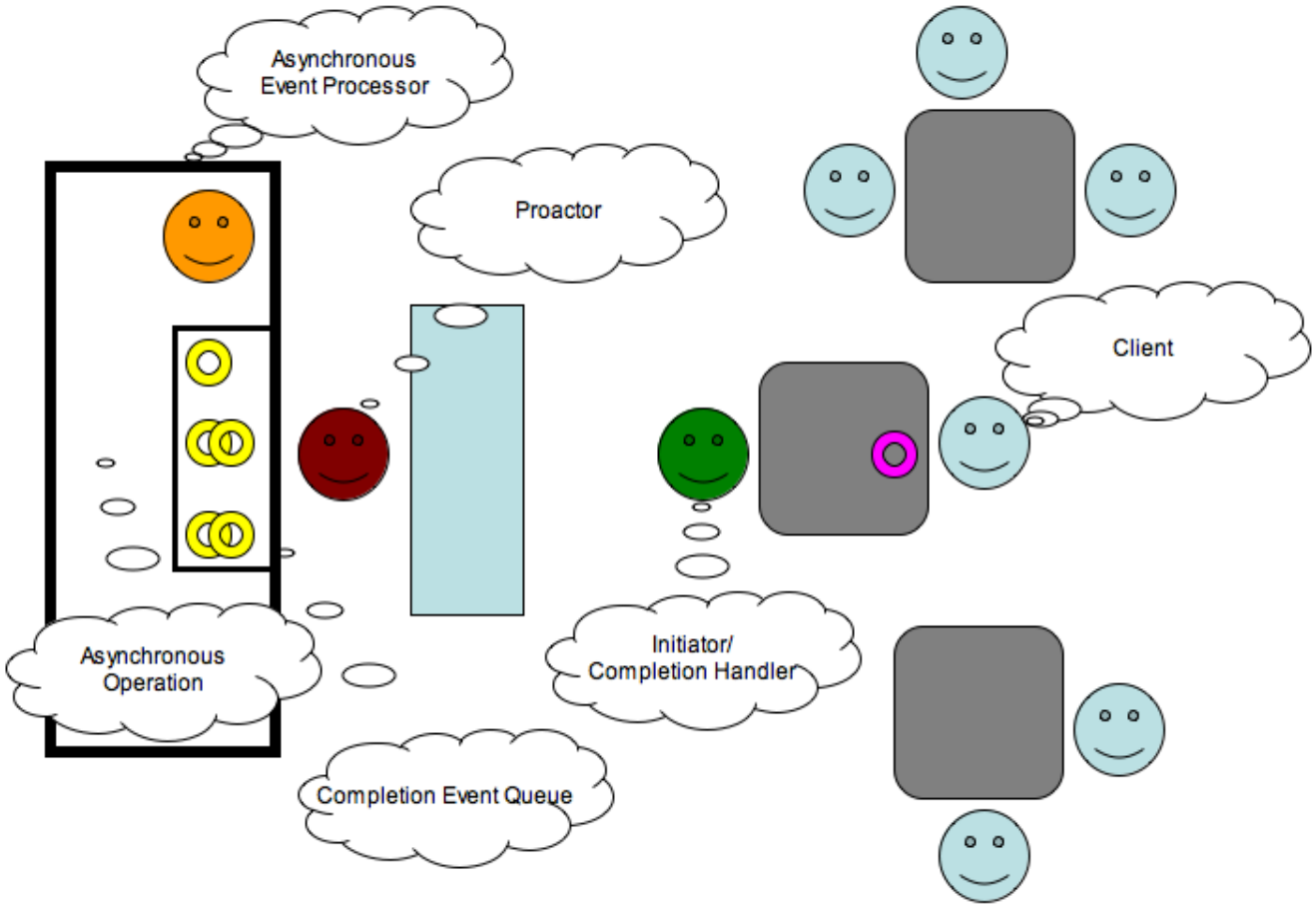
A little analogy



A little analogy



A little analogy



java.nio.*

Features

The NIO APIs include the following features:

- Buffers for data of primitive types
- Character-set encoders and decoders
- A pattern-matching facility based on Perl-style regular expressions
- Channels, a new primitive I/O abstraction
- A file interface that supports locks and memory mapping
- A multiplexed, non-blocking I/O facility for writing scalable servers

source: *http://java.sun.com/j2se/1.4.2/docs/guide/nio/*

java.nio.*

Features

The NIO APIs include the following features:

- Buffers for data of primitive types
- Character-set encoders and decoders
- A pattern-matching facility based on Perl-style regular expressions
- Channels, a new primitive I/O abstraction
- A file interface that supports locks and memory mapping
- A multiplexed, non-blocking I/O facility for writing scalable servers

source: *http://java.sun.com/j2se/1.4.2/docs/guide/nio/*

This sounds interesting!

`java.nio.channels`

Defines channels, which represent connections to entities that are capable of performing I/O operations, such as files and sockets; defines selectors, for multiplexed, non-blocking I/O operations.

SocketChannel

A selectable channel for stream-oriented connecting sockets.

Selector

A multiplexor of SelectableChannel objects.

SelectionKey

A token representing the registration of a SelectableChannel with a Selector.

Completion handler

- Type for the results

SelectionKey

Allows: defining operation type; obtaining the selector and the channel associated; include any `Object` as an attachment

- Type of dispatching (objects/function pointers)
- Dispatch strategy (single/multimethod)

CompletionHandler.java

```
public interface CompletionHandler {
  public void handleEvent(SelectionKey sk);
}
```

Asynchronous operation processor

- Asynchronous operation interface (Wrapper facade/ACT)
- Asynchronous operation processor mechanism

Proactor

- Hierarchy (multi-plattform proactors)
- Completion event queue and demultiplexing mechanisms

`Selector`

Queue: key set, selected-key set, cancelled-key set
Demultiplexing: `select()` method

- Dispatch strategy
- Concrete implementation

Proactor

Proactor.java

```
public class Proactor {
  private Selector selector;

  public Proactor() {
    selector =
      SelectorProvider.provider().openSelector();
  }

  public void register(SelectableChannel handle,
              int eventType, Object completionHandler) {
    handle.register(selector, eventType, completionHandler);
  }
  (...)
```

Proactor

Proactor.java

```
  (...)
  public void handleEvent() {
    while (selector.select() > 0) {
      Set readyKeys = selector.selectedKeys();
      Iterator i = readyKeys.iterator();
      while (i.hasNext()) {
        SelectionKey sk = (SelectionKey) i.next();
        i.remove();
        (CompletionHandler) sk.attachment()).handleEvent(sk);
      }
    }
  }
}
```

Concrete completion handlers

- State
- Handles
- Implementation

Acceptor.java

```
public class Acceptor implements CompletionHandler{
  private SocketAddress address;
  private ServerSocketChannel ssc;
  private Proactor proactor;
  (...)
```

Concrete completion handlers

Acceptor.java

```
(...)
public Acceptor(Proactor p, String host, int port) {
  address = new InetSocketAddress(host, port);
  proactor = p;
}

public void accept() {
  ssc = ServerSocketChannel.open();
  ssc.configureBlocking(false);
  ssc.socket().bind(this.address);
  proactor.register(ssc, SelectionKey.OP_ACCEPT, this);
}
(...)
```

Concrete completion handlers

Acceptor.java

```
  (...)
  public void handleEvent(SelectionKey sk) {
    ServerSocketChannel handle =
      (ServerSocketChannel) sk.channel();
    SocketChannel s = null;
    s = handle.accept();
    s.configureBlocking(false);
    Worker w = new Worker(proactor, s);
    w.work();
  }
}
```

Concrete completion handlers

Worker.java

```
public class Worker implements CompletionHandler {
  private Proactor proactor;
  private SocketChannel socket;
  private ByteBuffer b_read;
  private ByteBuffer b_write;

  public Worker(Proactor p, SocketChannel s) {
    proactor = p;
    socket = s;
    b_read = ByteBuffer.allocateDirect(1024);
  }
  (...)
```

Concrete completion handlers

Worker.java

```
  (...)
  public void work() {
    socket.read(b_read);
    proactor.register(socket, SelectionKey.OP_READ, this);
}

  public void handleEvent(SelectionKey sk) {
    System.out.println(new String(b_read.array()));
  }
}
```

Server

Server.java

```
public class Server {
  public static void main(String args[]) {
    Proactor proactor = new Proactor();
    Acceptor acceptor =
      new Acceptor(proactor, "localhost", 9999);
    acceptor.accept();
    proactor.handleEvent();
  }
}
```

Benefits

- Separation of concerns
- Portability
- Different concurrency mechanisms
- Simplification of application synchronization

Drawbacks

- Efficiency depends on the platform
- Complexity of programming, debugging and testing
- Scheduling and controlling asynchronous operations

Variants

Asynchronous Completion Handlers

All completion handlers are required to act as initiators

Concurrent Asynchronous Event Demultiplexer

The proactor demultiplexes and dispatches completion handlers concurrently

Shared Completion Handlers

Initiators can invoke multiple asynchronous operations at the same time sharing the same completion handler

Asynchronous Operation Processor Emulation

When asynchronous operations are not available in the OS, it's possible to emulate them (thread per operation,thread pool)

Known uses

Completion ports in Windows NT

Windows NT acts as an asynchronous operation processor: supports various types of asynchronous operations; generates completion events and queues them in completion ports

POSIX AIO

`aio*()` family APIs; similar to Windows NT

ACE Proactor Framework

ACE provides a portable Proactor framework (`AceProactor`)

Device driver interrupt-handling mechanisms

Hardware devices driven by asynchronous interrupts

Related patterns

Asynchronous completion token

Usually used in conjunction with the Proactor pattern. For every asynch op invoked, an ACT is created and returned to the caller when the operation finishes

Observer

All dependents are updated automatically when a change occurs; in the Proactor handlers are informed automatically when events occur

Reactor

Considered an asynchronous version of the Reactor Pattern

Related patterns

Active Object

Decouples method execution from invocation; in Proactor Pattern the asynchronous processor performs operation on behalf of the initiators

Chain of Responsability

Decouples event handlers from event sources; initiators and completion handlers in Proactor (although in CoR the source has no knowledge about which handler will be executed)

Leader/Followers and Half-Sync/Half-Async

Demultiplex and process various types of events synchronously

Questions?

?

Question

Considerer the following Proactor design decisions:

- Multiple-proactor implementation
- Spawned thread per event

Describe the different uses, advantages and disadvantages of both decisions