Assignment 2 — Reactor pattern

T-106.5600 Concurrent programming

Jan Lönnberg

Laboratory of Software Technology
Helsinki University of Technology

14th October 2008

Slides by Jan Lönnberg and Teemu Kiviniemi

Introduction

The Story

- As a Hangman fan, you want to create a networked version of the game.
- Multiple players co-operate in guessing a word.
- Also being a fan of design patterns, you decide to use the Reactor design pattern for the game server.

Introduction

You get:

- Reactor API declarations.
- Example `Handle`s and a test program for the Reactor.
- Some test code.

Your task is to:

- Implement the Reactor pattern.
- Write a multiplayer Hangman server based on the Reactor pattern.

Reactor pattern

What is the Reactor pattern?

- Reactor is a design pattern for handling concurrently arriving events.
  - A design pattern is a general solution to a common programming problem.
- Reactor can be used to hide complexity caused by concurrency from the application logic.
- Reactor is commonly used in a wide range of concurrent programs including servers and graphics libraries.

Reactor pattern

Structure of Reactor pattern

- Handles
  - Receive events needed by the application (e.g. timers, files, packets, synchronisation constructs, user actions).
- Synchronous event demultiplexer (demux)
  - Serialises the concurrently occurring events.
- Initiation dispatcher
  - Dispatches the serialised events to the correct event handlers.
  - Provides methods to (de)register handles and event handlers.
- Event handlers
  - Process events one at a time as needed by the application.

Reactor pattern

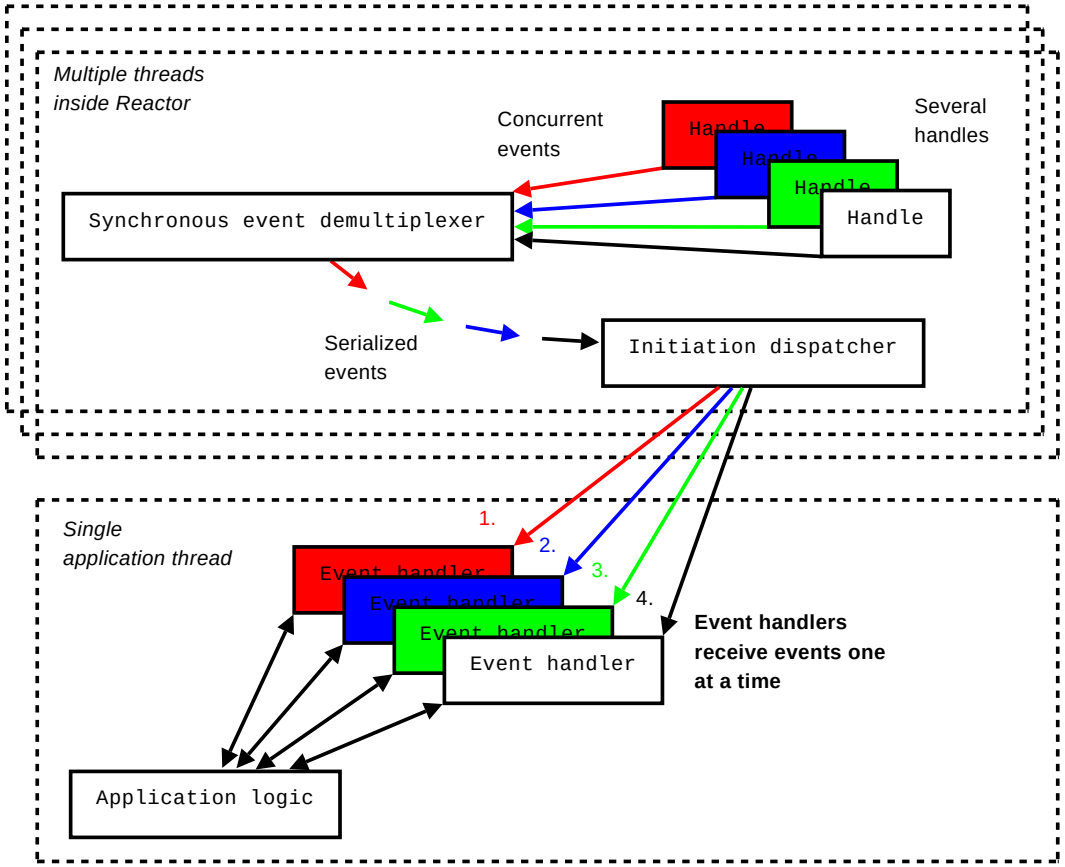

Figure: How concurrently occurring events can be serialised with the Reactor pattern

Reactor API

Java interface (1/3)

- `public class Dispatcher`
  - Implements the Dispatcher part of the Reactor pattern.
  - `public Dispatcher()`
    - Create a new `Dispatcher` with no `EventHandler`s.
  - `public void handleEvents() throws InterruptedException`
    - Waits for events for all registered `EventHandler`s and dispatches them until no more `EventHandler`s are registered.
  - `public void addHandler(EventHandler<?> h)`
    - Register a new `EventHandler h`.
  - `public void removeHandler(EventHandler<?> h)`
    - Deregister a previously registered `EventHandler h`.

Reactor API

Java interface (2/3)

- `public interface EventHandler<T>`
  - Handles events from its `Handle`.
  - `public Handle<T> getHandle()`
    - Get the `Handle` from which the `EventHandler` receives events.
  - `public void handleEvent(T s)`
    - Handle an incoming message represented by an object `s`, as received from the `Handle`.

Reactor API

Java interface (3/3)

- `public interface Handle<T>`
  - Represents one end of a (possibly bidirectional) communications channel.
  - `public T read()`
    - Wait for a message from the channel and return it (or an object representing an error condition).
    - May return `null` indicating that the `Handle` has been closed or has a fatal error and may no longer be `read()`.

Reactor API

Implementing the API

- Use the basic Java synchronisation primitives (those built into the language and `java.lang`, not `java.util.concurrent`).
- Inefficient solutions, such as polling and busy-waiting, will be rejected.
- Use of unsafe methods such as `java.lang.Thread.destroy()` will also lead to rejection.

Hangman

Gameplay

- The players guess one letter at a time.
- Each time a player guesses a letter, the server sends each player a message describing the state of the guessing so far.
- Players share guesses. Execution is completed when the word is guessed or when the amount of remaining attempts reaches zero.
- After the message describing the last guess is sent and processed, the server must disconnect all clients and terminate.

Hangman

Players

- Zero or more players connect to the Hangman server through TCP/IP.
- The players try to guess the word co-operatively.
- Players may (dis)connect at any time during the game.
- The first line of input sent to the server must be the name of the player.

Hangman

The multiplayer Hangman server

- Communicates with its clients through a TCP/IP socket, using plain text.
- Handles all incoming communication using the Reactor pattern.
- May use only one thread directly. Any other threads must be created and managed by the Reactor.

Hangman

Starting the server

- The Hangman server must start by the command:
  - `java hangman.HangmanServer <word> <guesses>` where:
    - `<word>` = the word to guess
    - `<guesses>` = the number of failed attempts allowed before termination
- If the server startup was successful, the server must print only the number of the TCP port it is listening on to standard output. The example `AcceptHandle` does this for you.

Hangman

Connecting to the server

- *telnet* and *netcat* can be used as clients for the Hangman server.
- To connect, you can use one of the following commands:
  - `telnet <hostname> <port>`
    or
  - `netcat <hostname> <port>`
    where:
    - `<hostname>` = the host the Hangman server runs on (which is `localhost` if the server runs on the same host as the client)
    - `<port>` = the port on which the Hangman server is listening

Assignment

Important notice

- Implementing the Reactor pattern and the Hangman server is two separate tasks.
- Your Hangman implementation must work with any compliant Reactor implementation and vice versa.

A tip

- Test your implementations with the tester available on the assignment web page.
- If the tester works correctly with your code, it is probably good. If not, you know there is something you can improve.

Assignment

Doing the assignment

- Group size: 1–2 students.
- Grading does not depend on group size.
- Grade: 0 (fail), 1 (pass) – 5 (pass with honours).

Submission

- Deadline is 2008-11-10 23:59.
- Submit a ZIP archive containing:
  - Your Reactor implementation.
  - Your Hangman server implementation.
  - Your report.
- There will be no extra round for re-submissions or late submissions.

Assignment

Coding style

- The program should be written in clear, reasonably object-oriented Java.
- Explanatory comments are required for code that is not self-explanatory to a programmer fluent in Java.
- English for variable names, method names, comments and reports is recommended.
- Cryptic method and variable names may not be used.

Assignment

Instructions and requirements

- Full instructions are on the course home page.
- Include a report explaining:
  - The reasoning behind your solution.
  - Measures you have taken to ensure your solution is correct
  - Any unexpected behaviour you may have encountered during testing and how you have investigated and resolved it.
- The instructions on the course web page are the authoritative description of the assignment.

Assignment

Questions and clarifications

- Technical questions and clarification requests should be sent to the course newsgroup:
  `opinnot.tik.rinnakkaisohjelmointi`
- Clarifications (if any) will be posted to the newsgroup.
- Hands-on sessions will be held on Wednesdays 2008-10-15, 2008-10-22 and 2008-11-05.

Assignment

Tips

- Read the instructions thoroughly.
- Test your code well. The supplied test package is useful.
- Follow the newsgroup.
- Start working now.

Conclusion

Conclusion

- Assignment is intended to help learn monitors and condition variables, Java threads and client-server programming (using the Reactor pattern).
- These slides and the full assignment instructions are available on the course web page.