The Conclave

Programming 2

Final report for group project

Sapienza, Università di Roma
Applied Computer Science and Artificial Intelligence
Academic Year 2024/2025

Instructor: Cenciarelli Pietro

Abstract

This paper describes the design and implementation of a desktop application that simulates the election of a pope inside a virtual Sistine Chapel. The simulation, written in Java 17, models each cardinal as an independent thread that roams a two-dimensional board, exchanges opinions, and eventually casts a vote. A dedicated *Conclave* thread orchestrates multiple scrutinies until a candidate reaches a two-thirds quorum, after which the graphical interface proclaims the *Habemus Papam*. We provide a class-level overview, detail the core algorithms—movement, persuasion, voting—and discuss performance characteristics and avenues for further work.

Table of contents

1. Introduction

The papal conclave is a centuries-old procedure characterised by secrecy, debate and successive ballots. Modelling such a social process in software offers a sandbox for experimenting with collective decision making, graph diffusion and multithreaded coordination. The project analysed in this paper was produced as an academic exercise and is composed of roughly 1 500 lines of Java code.
A front-end built with *Swing* presents the Latin mass, a start menu, live vote counts and finally the elected pope; the back-end runs the stochastic simulation.

The goal of this report is:

1. Present the overall architecture and execution flow.
2. Detail the algorithms that drive movement, persuasion and voting.
3. Illustrate key implementation choices, highlighting strengths and areas for improvement.

2. Architectural Overview

The code is organised as follows

Layer	Package(s)	Principal classes	Responsibility
Simulation core	`project.conclave`	*Conclave*, *Cardinal*, *Board*, *Square*, *Opinion*	Domain logic, multithreading
Utilities	`project.util`	*Heap*, *Reader*, *Triplet*, *ConclaveSetupException*	Generic data structures & helpers
Presentation	`project.gui`	*Window* + components (*StartButton*, *Title*, *LeaderboardLabel*, *PopeLabel*, *ScrollingText*)	Swing UI

The entry point is *Main* in the root package, while a CSV file (passed at runtime) contains the roster.

2.1 Data model

Cardinal

Attributes capture **identity** (`name`, `surname`, `id`), **influence** (an integer weight), **position** (inner class `Position` storing *x*, *y*) and **state** (heap of opinions, availability flags, counters).

Opinion is a simple pair `(id, value)` representing the support a cardinal assigns to a candidate.

Heap implements a fixed-size **max-heap** of *Opinion* objects plus an auxiliary *positions* array that offers *O(1)* access to arbitrary nodes—crucial for quickly updating scores during debates.

Triplet records `(name, surname, votes)` and is comparable for leaderboard sorting.

The conclave unfolds on a *Board*—a square lattice of `size × size` *Square*s, each holding a list of resident cardinals. No walls exist; movement wraps inside bounds.

2.2 Thread Topology

```
Main Thread
 ├─ UI event-dispatch thread (Swing)
 └─ Conclave (extends Thread)
      ├─ Cardinal[0] (Runnable)
      ├─ Cardinal[1]
      └─ … Cardinal[n-1]
```

The *Conclave* thread spawns one worker per cardinal at each scrutiny and blocks until all have finished debating and voting. Synchronisation relies on `Thread.interrupt()` and polling plus explicit `wait()/notify()` on shared locks (`Main.data`, individual *Cardinal* monitors).

3. Core Algorithms

3.1 Movement in the Sistine Grid

Each cardinal starts at a random square (`Position.setRandom()`) and, while the *Conclave* marks the *debate* phase, executes `moveRandomly()`:

1. Build `availableDirections` based on current coordinates (edges forbid crossing the perimeter).
2. Choose one direction uniformly at random.
3. Atomically remove `this` from the old square, update `(x,y)`, and add to the new square.

The algorithm's critical sections are protected by synchronising on the target *Square* object, preventing inconsistent occupancy lists.

3.2 Informal Debates and Opinion Dynamics

When two cardinals share a square, a handshake determines who becomes *caller* vs *listener*. Both parties execute `exchangeInformation(Cardinal caller)`, whose heart is a **probabilistic influence contest**:

```
threshold = myInfluence / (myInfluence +
callerInfluence) × 100
if random(0..99) <= threshold
        caller wins → listener updates
else   listener wins → caller updates
```

After a duel, the loser adopts (partially) the top opinion from the winner's heap, weighted by the latter's influence. Each participant carries an array `encounteredOpinions[cardinalId]` to avoid counting the same persuasion twice.

3.3 Voting and Scrutiny Cycle

At the end of the 3-second debate window:

1. *Conclave* interrupts every *Cardinal* thread.
2. Each cardinal's heap head (most valued candidate) is read and tallied into `votes[int id]`.
3. Results are pushed into `Main.data` as *Triplets*, the UI refreshes the leaderboard, and the majority check runs.

A cardinal becomes **pope** when their vote count satisfies

votes[i] ≥ 2 × floor(cardinals.size / 3)

—an implementation of the canonical *2/3* super-majority. If unmet, all opinion heaps persist, new *Cardinal* threads are spawned, and the process reiterates.

3.4 Quorum Calculation

Canon law prescribes that a new pontiff must be elected by at least two thirds of the cardinal electors present. The implementation translates this rule into integer arithmetic:

```
int target = (cardinals.size() / 3) * 2;   // 2/3
without floating point
if (votes[pope] >= target) {
    /* success */
}
```

Because `int` division truncates toward zero, the expression is correct for every cardinal count that is a multiple of three and errs on the safe side (requires one extra vote) otherwise.

3.5 Termination and Resource Cleanup

Once the condition is met the `Conclave` thread breaks out of its loop, interrupts any lingering worker threads (belt & braces) and publishes the pope's full name via the static field `Main.pope`. The UI reads that field in `Window.displayPope()` and replaces the whole content pane with a bold "Habemus Papam" title and the elected name. In production one would also:

- shut down the Swing timer responsible for scrolling text;
- close the CSV scanner;
- interrupt the event-dispatch thread on a clean exit.

3.6 Complexity Analysis

Movement and *opinion updates* are `O(1)` amortised; *vote counting* is `O(n)` per scrutiny. The heap operations remain logarithmic, but with only one entry per cardinal, they behave effectively as constant time. The whole simulation therefore scales linearly with the number of participants.

4. Key Classes in Detail

4.1 Conclave

Conclave holds static references to the *Board* and `ArrayList<Cardinal>`. Its `run()` method:

```
initializeBoard(csvPath, boardSize);
while (true) {
    spawnCardinals();
    Arrays.fill(votes, 0);
    sleep(3000);
    interruptWorkers();
    waitWorkers();
    collectVotes();
    if (hasTwoThirdsMajority()) break;
}
announcePope();
```

Robustness: the constructor validates the CSV path and throws *ConclaveSetupException* on error.

4.2 Cardinal

Implements *Runnable;* the loop terminates on `Thread.interrupt()`. Concurrency primitives:

- A cardinal waits (`wait(random(250—750))`) when it has spoken to all neighbours to mitigate busy-waiting.
- Method-level `synchronized` blocks ensure that opinion exchanges are transactional.

The twin counters `cardinalsToTalkTo` and `cardinalsToListenTo.size()` implement a simple rendezvous protocol: a cardinal finishes only after delivering and receiving every scheduled message.

4.3 Heap

Functions: `add(id, value)`, `remove(id)`, `getTop()`. The heap stores one node per candidate and updates values in logarithmic time via `heapifyUp`/`heapifyDown`. The auxiliary `positions[]` solves *decrease-key* without a hash map.

4.4 Utility & GUI Components

Reader parses semicolon-separated values and skips the CSV header. *Window* encapsulates a `JFrame` and exposes high-level views: mass scroll, main menu, votes leaderboard and pope screen. Child components adopt monospace fonts for an *ad hoc* aesthetic. Synchronisation with the simulation uses the shared `Main.data` and `Main.menu` locks.

4.5 GUI Workflow and Styling

Window anchors the Swing hierarchy and opts for absolute positioning rather than layout managers—a valid choice given the fixed dimensions of 600×400 px. Each view is composed like a slide:

- **Menu View** – shows a mono-spaced title and a `StartButton`. When the user clicks, the listener performs `synchronized (Main.menu) { menu.notify(); }`, unblocking the main thread.
- **Mass View** – creates a `ScrollingText` pane with Latin verses of the *Exit omnes* mass, centers it horizontally, and starts a `javax.swing.Timer` that raises its *y* coordinate by 2 px every 30 ms, achieving a slow upward crawl.

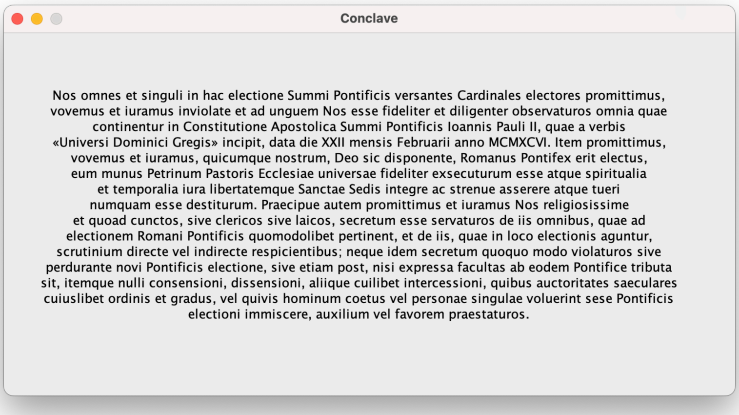
- **Leaderboard View** – accepts `ArrayList<Triplet<String,Integer>>` and, after sorting, instantiates one `LeaderboardLabel` per entry. Labels maintain consistent alignment by padding vote counts to three digits (`String.format("%03d", votes)`).
- **Pope View** – emphasises the finale with a 32 pt bold font. Finally, *Habemus Papam*!

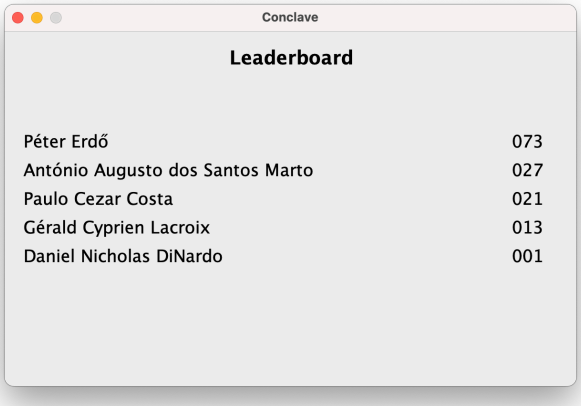
All custom components share the `DejaVu Sans Mono` typeface to impose a retro terminal vibe. Colours are OS defaults, achieving sufficient contrast without explicit RGB codes.

Screenshots

<< The Conclave is set to start, it just needs the Camerlengo (you) to start it

>> The Exit Omnes Mass is conducted by the Camerlengo. No worries, we've got you covered!





<< Cardinals are influencing each other, the leaderboard updates!

>> White smoke comes from the Sistine Chapel! We got the new Pope! *Habemus Papam!*