Sorting Hat

Java Swing Final Project
Write-up

Author: Mauricio Gidi
Date: 14 Dec 2025

Course: CSCI E-10b — Introduction to Computer Science Using Java, II
Term: Fall 2025
Student: Mauricio Gidi
Teaching Assistant: David Habermehl

1. Project summary

Sorting Hat is a desktop Java application that presents a short personality-style questionnaire and assigns the user to a Hogwarts house. The application uses Swing/AWT for an event-driven GUI and Gson to load its item banks and scoring weights from JSON resources.

Many online "sorting quizzes" map each answer straight to one house. That is simple to implement, but it does not match the idea that the Sorting Hat evaluates a student as a whole person. This project models a multi-trait profile and then produces calibrated house probabilities instead of hardwiring a single answer → a single house.

The model tracks 10 underlying traits (COURAGE, AMBITION, LOYALTY, INTELLECT, CURIOSITY, CONFORMITY, RISK_TAKING, COMPASSION, ASSERTIVENESS, DILIGENCE). Each response nudges one or more traits, and the current trait pattern is converted into four house scores and a softmax probability distribution.

2. User experience and screen flow

The UI is implemented as a single main window that swaps between screens using a CardLayout. User input is constrained to on-screen controls (buttons/slider), which helps keep the application user-proof and prevents invalid input formats.

- Welcome: animated entrance, then Continue.
- Mode selection: choose Quick/Standard/Thorough.
- Likert phase: for each item, the user selects a position on a 1.0–5.0 slider (0.5 increments) and clicks Next; the app measures response time.
- Decision point: compute house probabilities; if a tie is detected, run tie-breaker; otherwise show results.
- Tie-breaker: short forced-choice scenarios between the tied houses; responses also use response-time weighting.
- Results: show winning house, probability chart, and "View Trait Profile" dialog.

2.1 Examples of operation (sample interaction)

Example A: typical run (no tie-breaker)

1. Choose mode: Standard
2. Answer Likert items (5-point scale). Example responses:
    1) "I would bravely stand my ground against a dangerous foe to protect my friends." → Agree
    2) "I prefer coming up with a clever plan over rushing in wand-blazing when facing a challenge." → Neutral
    3) "I love riddles and puzzles, and I often spend free time trying to solve challenging magical enigmas." → Agree
3. (...remaining items omitted)
4. View Results: the app displays the winning house and a probability chart. Example (illustrative): Gryffindor 0.48, Hufflepuff 0.22, Slytherin 0.19, Ravenclaw 0.11.

Example B: tie-breaker path (demonstration run)

1. Choose mode: Quick
2. Answer every Likert item as Neutral (center value) and click Next quickly.
3. Because the answers keep trait scores near the baseline, the house probabilities remain close and a tie group is detected.
4. The app shows 1 forced-choice round (multiple short A/B scenarios between the tied houses).
5. After the forced-choice questions, the Results screen is shown with updated probabilities.
6. Note: question order is randomized, so exact numbers vary between runs. Probabilities always sum to 1.0; a larger top probability can be interpreted as higher confidence, even though the UI intentionally only displays the numeric distribution.

2.2 Flow and architecture diagrams

Welcome (Continue)
→ Mode Selection (Quick / Standard / Thorough)
→ Likert Phase (answers + response time)
→ Compute house probabilities (softmax over house scores)
→ Decision: **Tie detected?** (top houses within threshold)
  Yes → Forced-choice tie-breaker (targeted comparisons) → Results (House + probability chart; Trait Profile optional)
  No (or tie resolved) → Results (House + probability chart; Trait Profile optional)

3. Data and resources

All quiz content and weights are loaded from JSON resources on the runtime classpath. Images are also loaded from the classpath and used to theme the UI.

Important runtime requirement: the program loads all JSON and image assets using absolute classpath paths such as "/resources/LikertItems.json" and "/resources/ParchmentBackground.png". If the /resources/... files are missing, renamed, or not on the classpath, the application cannot initialize its item bank and will fail during startup. In addition, the JSON files must remain syntactically valid and internally consistent.

JSON resources

- TraitToHouseWeights.json: lists trait names, house names, and a trait→house weight matrix used to aggregate trait scores into house scores.
- LikertItems.json: Likert items with an id, display text, per-trait weights, IRT parameters (discrimination a and thresholds b1..b4), and timing defaults; also includes “quick” and “standard” forms (lists of item ids).
- ForcedChoiceItems.json: forced-choice items keyed by a pair of houses; each item has a stem and two options, each option mapping to a house; also includes timing defaults.

Image resources

- HatIcon.png: window icon and welcome imagery.
- ParchmentBackground.png: full-window parchment background with vignette and tint animation.
- QuickIcon.png / StandardIcon.png / ThoroughIcon.png: mode selection icons.

4. Architecture

The project follows a simple Model–View–Controller structure. The controller owns the quiz state machine; the model owns item selection and scoring; the view renders screens and reports user actions back through a listener interface.

- Controller: MainController drives the flow (welcome → mode → Likert → optional tie-breaker → results).
- Model facade: SortingHatModel exposes a small API for selecting items, recording responses, and reading probabilities.
- Views: MainWindow (JFrame) hosts card screens; screen panels implement the actual layouts; SortingHatTheme centralizes styling.

Note on code size: the view/ package is verbose because it contains the Swing layouts, styling helpers, and image-based theme (aesthetics). The core program logic that affects correctness and scoring is concentrated in model/ (SortingHatModel, ItemBank, ScoringEngine, ResponseTimeWeight).

5. Template classes

The following classes are the primary “template classes” used to model the problem domain and support the application. For each class, the write-up lists what it models, its state (instance variables), and what it does (key methods).

5.1 model.SortingHatModel

Models: The domain facade for a single quiz run, coordinating item selection and scoring.

State: ItemBank itemBank; ScoringEngine scoringEngine.

Key methods:

- selectLikertItemsForMode(mode): returns a randomized list for Quick/Standard/Thorough.
- recordLikertResponse(item, value, seconds): forwards a response to the scoring engine.
- recomputeAndGetHouseProbabilities(): recomputes house scores and returns probabilities.
- getTopHouse(), getTieGroup(threshold[, subset]): tie detection helpers.
- selectNextForcedChoiceItemForPair(a, b), applyForcedChoiceAnswer(item, optionKey, seconds): forced-choice tie-breaker support.

5.2 model.ItemBank

Models: The quiz item repository loaded from JSON resources.

State: Lists of Likert items and forced-choice items; lookups by id; form id lists; trait/house lists; weight matrix.

Key methods:

- Constructor loads JSON resources using Gson (classpath InputStreams).
- selectLikertItemsForMode(mode): returns shuffled items; Quick/Standard use form id lists.
- selectNextForcedChoiceItemForPair(a, b): rotates and reshuffles forced-choice items per house pair.
- applyForcedChoiceAnswer(item, optionKey, seconds): resolves option → house and calls ScoringEngine.applyForcedChoiceResult().

5.3 model.ScoringEngine

Models: Ongoing scoring state: trait scores, house scores, and house probabilities.

State: traitScoreByTrait (map), houseScoreByHouse (map), houseProbabilityByHouse (map).

Key methods:

- recordLikertResponse(item, value, seconds): updates trait scores using response-time weighting and IRT-based latent estimate.
- recomputeHouseScores(): aggregates trait scores into house scores and recomputes probabilities via softmax.
- applyForcedChoiceResult(chosen, other, timeWeight): logistic predicted-win update scaled by learning rate and time weight.
- getTopHouse(), getTieGroup(threshold[, subset]): supports tie-breaker logic.

5.4 model.ResponseTimeWeight

Models: A utility for converting response time (seconds) into a [0,1] multiplier.

State: None (static utility).

Key method: compute(timing, seconds) applies a piecewise-linear ramp based on JSON timing parameters.

5.5 controller.MainController

Models: The quiz flow state machine and event handler (implements MainWindow.UserActionListener).

State: Active LikertSession, optional TieBreakerSession, and a Swing Timer for the tie-breaker interlude.

Key methods:

- start(): shows the welcome screen and initializes the window configuration.
- onQuizModeSelected(...): starts a LikertSession for the chosen mode.
- finishLikertPhaseAndDecideNextStep(): computes probabilities and decides results vs tie-breaker.
- TieBreakerSession.advanceFlow(): runs forced-choice rounds until tie resolves or rounds end.

5.6 view.MainWindow and screens

Models: The Swing UI shell and its screens; measures response time between showing a question and submission.

State: CardLayout + screen panels; QuestionTimer (nanosecond-based); background panel with tint animation.

Key methods:

- showWelcomeScreen(), showModeSelectionScreen(), showLikertQuestionScreen(), showForcedChoiceQuestionScreen(), showResultsScreen().
- showTraitProfileDialog(): modal dialog showing per-trait scores.

6. Scoring and algorithms

The application combines three ideas: (1) Likert responses update trait scores, (2) trait scores aggregate into house scores, and (3) house scores are normalized into a probability distribution. A tie-breaker path then applies additional forced-choice updates.

6.1 Response-time weighting

Each item includes timing parameters: expected_time_sec, rapid_threshold_sec, and down_weight_factor. The ResponseTimeWeight.compute() method returns a multiplier in [0,1]: very rapid responses are down-weighted, and slower responses ramp up to full weight at the expected time.

```
If t <= rapid_threshold:   weight = d * (t / rapid_threshold)

If t >= expected_time:     weight = 1.0

Otherwise:                 weight = d + (1-d) * ((t-rapid_threshold)/(expected_time-rapid_threshold))
```

6.2 Likert scoring with IRT-inspired latent estimate

Each Likert item contains IRT parameters: discrimination a and thresholds b1..b4 for five response categories. The scoring engine maps the selected response (1.0–5.0, including half steps) to a latent level θ by using midpoints between thresholds (bounded by global θ min/max), and linearly interpolates when the response is fractional.

```
basePoints = θ * a

traitContribution = basePoints * responseTimeWeight * trait_weight
```

6.3 Trait-to-house aggregation and probabilities

House scores are computed as weighted sums of trait scores using the matrix in TraitToHouseWeights.json. House probabilities are then computed via a softmax over house scores (with max-subtraction for numerical stability).

```
houseScore[H] = Σ_trait ( traitScore[trait] * weight[H][trait] )

prob[H] = exp(houseScore[H] - maxScore) / Σ exp(houseScore[*] - maxScore)
```

6.4 Tie detection and forced-choice tie-breaker

After the Likert phase, a tie group is defined as all houses whose probability is within a fixed threshold of the maximum. If the tie group has more than one house, the controller runs forced-choice comparisons between houses in the tie group.

- Forced-choice items are indexed by house pairs and rotated (with reshuffling) to avoid repeating the same question order.
- Each forced-choice answer updates two house scores in opposite directions using a logistic predicted-win probability and a learning rate.
- The update is scaled by response-time weight so rushed choices contribute less.

```
predictedChosenWins = 1 / (1 + exp(-(scoreChosen - scoreOther)))

error = 1 - predictedChosenWins

delta = learning_rate * timeWeight * error

scoreChosen += delta;  scoreOther -= delta
```

7. Build and run instructions (Gson on classpath/modulepath)

The project loads JSON and images using Class.getResourceAsStream("/resources/..."), so the resources directory must be available on the runtime classpath. Gson must be present at both compile time and runtime.

Recommended terminal approach:

Windows

```
mkdir build
```

```
javac -cp "libs/gson-2.10.1.jar" -d build SortingHatApp.java controller\*.java model\*.java view\*.java config\*.java

java -cp "build;libs/gson-2.10.1.jar;." SortingHatApp
```

macOS/Linux

```
mkdir -p build

javac -cp "libs/gson-2.10.1.jar" -d build SortingHatApp.java controller/*.java model/*.java view/*.java config/*.java

java -cp "build:libs/gson-2.10.1.jar:." SortingHatApp
```

8. Testing and robustness notes

The application is designed to avoid invalid user input by using GUI controls instead of free-form text fields. Parsing and null-handling are defensive in several places (for example, parsing ActionEvent commands and validating resource ids). Likert items are validated for blank/duplicate ids when loading JSON.

- Manual test passes should include: each mode length, tie-breaker path, and results → trait profile dialog.
- Resource loading failures surface as runtime exceptions; packaging should ensure resources are on the classpath.

8.1 Demonstration instructions

A) Each mode length (Quick / Standard / Thorough)
• Run the program and select each mode once to confirm the expected number of Likert questions appears (Quick=12, Standard=30, Thorough=60).

B) Forcing the forced-choice tie-breaker path
• Select Quick.
• For every Likert question, choose Neutral (center value) and click Next quickly (under the rapid-threshold time).
• This keeps the trait scores near the baseline, so the four house probabilities remain close; the tie group should contain multiple houses and the tie-breaker will appear.
• Complete the forced-choice questions and confirm the app returns to the Results screen.

C) Trait profile view
• On the Results screen, click “View Trait Profile” and confirm a dialog appears with the current trait scores.

D) Window close + quit behavior
• Close the window using the OS close button (X) or use any provided Quit/Exit control; confirm the application terminates cleanly.

9. Academic integrity: AI use and references

All files in the resources directory (JSON item banks and images) were created with assistance from ChatGPT. This use was permitted by the instructor and is documented here.

References

OpenAI (2025) ChatGPT (Version 5.1) [Large language model]. Available at: https://chat.openai.com/ (Accessed: 5 December 2025).

International Personality Item Pool (n.d.) IPIP: International Personality Item Pool. Available at: https://ipip.ori.org/index.htm (Accessed: 5 December 2025). (This source was cited as part of ChatGPT's internal reference usage and was independently accessed and verified by the student.)

AI Use Statement

ChatGPT (OpenAI, 2025) was used to generate initial questionnaire content and JSON structures for LikertItems.json, ForcedChoiceItems.json, and TraitToHouseWeights.json, as well as image assets in resources/.