Monty Hall: Finite State Machines, Simulation, and Evidence

Jeremy Evert

December 1, 2025

Contents

List of Figures

Chapter 1

Why Games Work (and Why Monty Hall Still Haunts Our Brains)

1.1 Games as a learning engine

Games are fun for a simple reason: they turn decision-making into a tight loop. You learn the rules, take an action, observe the outcome, and adjust. That loop is fast, emotional, and memorable.

For learning programming and math, games are especially useful because they quietly force us to think in the same structures we use in computing:

- **State:** what is true right now?
- **Transitions:** what event causes the next step?
- **Rules:** what actions are allowed at each step?
- **Evidence:** how do we know our strategy is good?

This project uses a game (the Monty Hall problem) as a friendly doorway into three serious ideas: finite state machines, simulation-driven evidence, and testable software design.

1.2 Why game shows became a television superpower

Game shows fit television extremely well. A viewer can drop in mid-episode and still understand the stakes quickly. Compared to many scripted programs, game shows can also be produced efficiently, and their structure naturally leaves room for advertising.

Television history also includes an important cautionary tale: in the 1950s, several quiz shows were revealed to have been manipulated, leading to public backlash and investigations. That period helped shape later expectations around televised contests.[**?**]

Over time, successful daytime game shows leaned into formats that were simple, repeatable, and brand-safe. The result was programming that could attract steady audiences and therefore steady advertising dollars.

1.3 Where the money comes from (prizes, sponsors, and the network)

The financial model behind many game shows is straightforward in *structure* even when the exact numbers are opaque in *detail*:

- Networks sell advertising time against the audience the show brings in.
- Production budgets pay for staff, sets, crew, post-production, and distribution.
- Prizes may be paid directly by the show/network or supplied/offset by sponsors as part of promotional arrangements.[**?**]

A practical research note for students: you will often see people ask questions like *"How much did this show gross?"* or *"How much profit did the station make?"* Those figures are rarely reported cleanly at the single-show level. Networks and studios tend to report finances at the company, division, or season level. So in this chapter we focus on what we can support well: the business logic, the longevity of the format, and the way audience attention translates into revenue.

1.4 Monty Hall: the man behind the doors

Monty Hall was a Canadian-American broadcaster and producer best known as the host and co-creator of *Let's Make a Deal*. The Television Academy notes that he appeared in more than 4,700 episodes of the show.[**?**]

This matters for us because it explains why the Monty Hall puzzle became culturally sticky: the show was not just a one-off novelty. It was a long-running, high-visibility format that normalized suspenseful choice as entertainment.

1.5 *Let's Make a Deal*: a marketplace of suspense

Let's Make a Deal is built around bargaining and uncertainty. Contestants (often called "traders") make choices between known rewards and unknown possibilities hidden behind doors, curtains, or boxes. Sometimes the unknown is a great prize; sometimes it is a humorous "zonk."

The show began in the 1960s and has had multiple runs and revivals across networks and syndication.[**?**] One reason the format has lasted is that it is a reliable machine for emotional moments: anticipation, risk, regret, surprise, and celebration.

Modern museum commentary emphasizes the importance of the show's daytime audience and how that audience connects to advertising value.[**?**]

1.6 The modern revival

A modern version of *Let's Make a Deal* has aired on CBS since 2009.[**?**] Even as sets and hosts evolve, the core ingredient remains the same: forced choices under uncertainty.

1.7 The three-door game (and the puzzle it inspired)

The classroom-friendly Monty Hall problem usually appears in a simplified, clean form:

1. There are three doors: behind one is a prize, behind the other two are goats.
2. You pick a door.
3. The host (who knows where the prize is) opens one of the other doors to reveal a goat.
4. You are offered a final choice: **stay** with your original door or **switch** to the remaining closed door.

This becomes the famous Monty Hall problem. Under the usual assumptions, switching wins with probability 2/3, while staying wins with probability 1/3.[**?**]

Why does this puzzle matter in a computing course? Because it is a perfect example of how humans can feel confident and still be wrong. Our intuition tends to treat the final choice as "50/50," but the process that produced the final two doors is not symmetrical. The host's action carries information, and the best strategy depends on modeling that action correctly.

1.8 What we are building in this project

We'll treat the Monty Hall game as an engineered system:

- First, we specify the game precisely using a **finite state machine** (Chapter 2).
- Next, we design a **data-collection FSM** that formalizes how we will simulate and record outcomes (Chapter 3).
- Then we implement the simulation as clean **object-oriented code** with strong **unit test coverage**, and we present results with tables and plots (Chapter 4).
- Finally, we summarize what we learned (Chapter 5) and include the full ChatGPT-assisted workflow as a transcript with analysis (Chapter 6).

1.9 Where we go next

Chapter 2 introduces finite state machines and uses an FSM diagram of Monty Hall as our blueprint. Once the game is written as states and transitions, it becomes much easier to implement, test, and measure.

Chapter 2

Finite State Machines (FSMs)

2.1 What is a finite state machine?

A **finite state machine** is a simple but powerful way to model a process that moves through a limited number of situations (called **states**) based on events or inputs (called **symbols**).

At any moment:

- the machine is in **exactly one state**,
- it receives an **input event** (or a timer tick, or a button press, etc.),
- it follows a **transition** to the next state,
- and (optionally) it produces an **output/action**.

A common formal definition (for a deterministic finite automaton, DFA) is the 5-tuple $(Q, \Sigma, \delta, q_0, F)$:

- Q = set of states
- Σ = input alphabet (the set of possible events)
- δ = transition function $(Q \times \Sigma \to Q)$
- q_0 = start state
- F = set of accepting/terminal states (optional for many programming-style FSMs)

In software engineering, we often use FSMs without emphasizing "accepting states"—instead we care about: **valid transitions**, **illegal transitions**, and how to implement the state logic cleanly.

2.2 Warm-up example 1: traffic light

Before we touch Monty Hall, here is a classic FSM: a traffic light. It has a tiny state set, clear transitions, and it's easy to test.

2.3 Warm-up example 2: turnstile (coin / push)

Another famous teaching FSM is a subway turnstile. Important lesson: **the same input can have different effects depending on the current state**. That's the entire point of state.

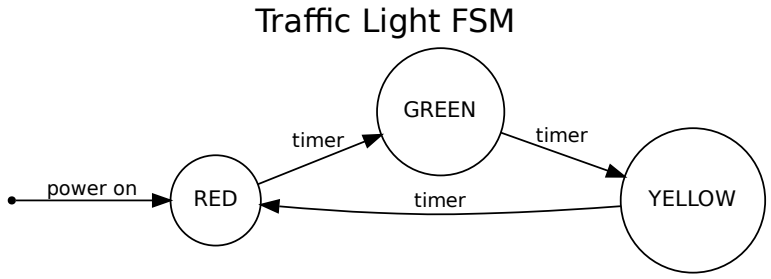

Figure 2.1: A simple traffic light FSM.

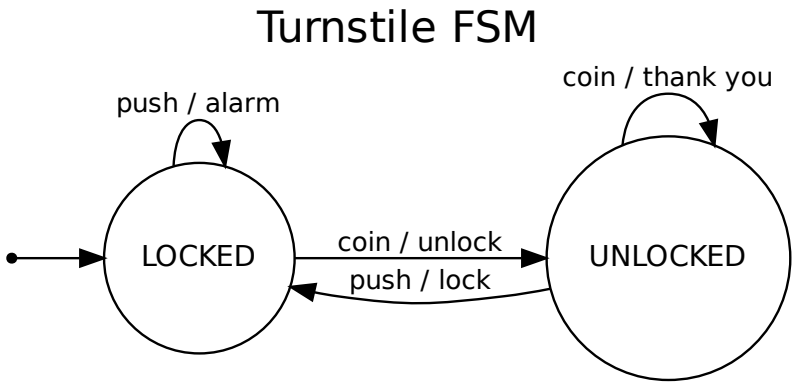

Figure 2.2: Turnstile FSM: locked/unlocked behavior.

2.4 Warm-up example 3: login session / lockout

FSMs also show up everywhere in security and application UX. A login flow is a state machine: logged out, authenticating, logged in, locked out, etc.

2.5 How FSMs map to code (the practical part)

In code, an FSM usually becomes:

- an `enum` (or strings) representing states,
- an "event" type (another enum or strings),
- a transition function: `(state, event) -> new_state`,
- optional actions performed on transitions.

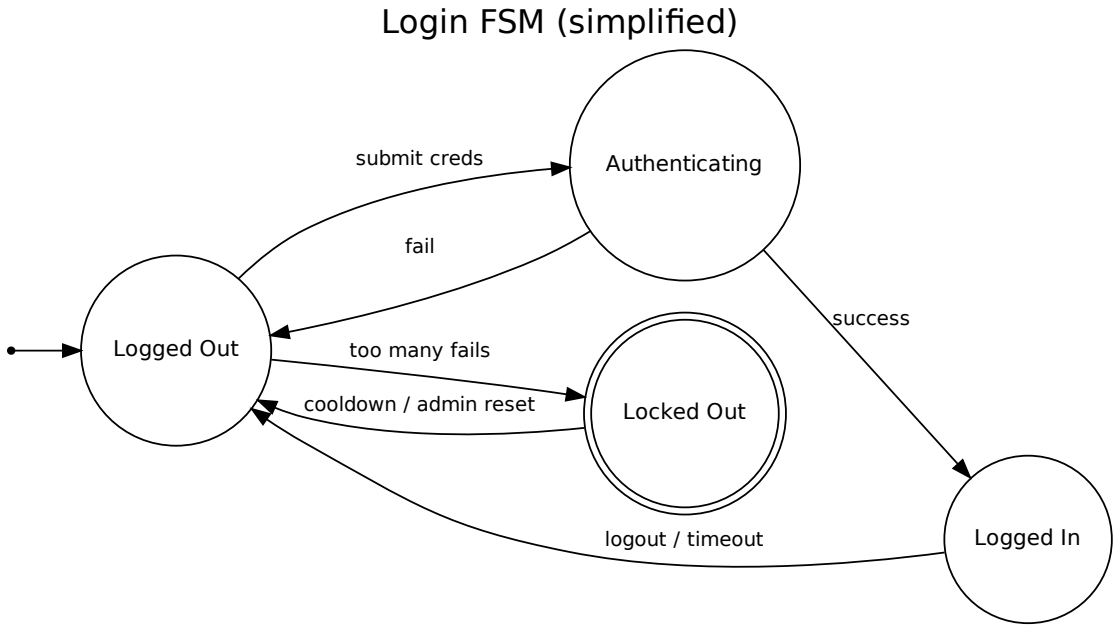

Figure 2.3: A simplified login FSM with lockout.

This approach gives you three superpowers:

1. **Clarity:** you can explain behaviour with a diagram.
2. **Testability:** you can unit-test every transition and every illegal move.
3. **Instrumentation:** you can log events and states to produce clean datasets.

2.6 Deterministic vs. nondeterministic (and why we mostly use deterministic in software)

A **deterministic** FSM has at most one outgoing transition per input symbol from each state. A **nondeterministic** FSM can have multiple possible next states for the same input. In programming projects like ours, we typically build deterministic FSMs, then model randomness as:

- probabilistic choices (e.g., random door placement),
- or hidden variables (car door, chosen door),
- or both.

When an FSM includes variables (like "car_door" or "player_choice"), many people call it an **extended finite state machine (EFSM)**: the state diagram is still the backbone, but we track a few additional values to make the model realistic.

2.7 The Monty Hall FSM (our blueprint)

Now we apply the same idea to the Monty Hall game. We will refine and expand this FSM as needed, but the key stages are: **setup** → **player picks** → **host reveals** → **player decides** → **resolve** → **reset**.

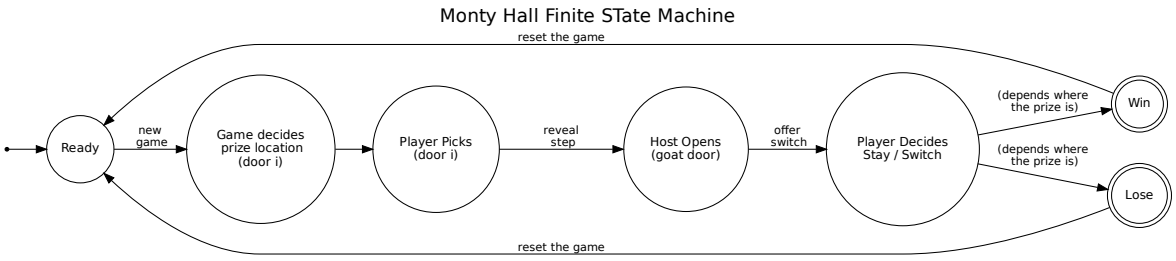

Figure 2.4: Monty Hall FSM (Graphviz-generated).

2.8 Where we go next

In Chapter 3, we design a data-collection FSM that formalizes *exactly* how simulation trials run, what gets logged, and how results flow into tables and plots. That design becomes the contract for the code and the unit tests.

Chapter 3

Designing the Data-Collection FSM (The Dataset We Can Trust)

3.1 Why a "data-collection FSM" at all?

Chapter 2 argued that FSMs give us clarity, testability, and instrumentation—especially the power to log clean datasets from a process that involves randomness. [**?**] That last one (instrumentation) is our focus now: we are going to define *exactly* what we log, when we log it, and how those logs roll up into results.

This chapter produces two deliverables:

1. A precise list of **fields** that define one Monty Hall trial (our row in a CSV).
2. A **Data-Collection FSM** that tells us what happens in what order, including logging and stopping rules.

3.2 The per-trial log (one row per game)

We will record one row per simulated trial. These are the minimum fields that let us verify randomness, debug mistakes, and compute meaningful statistics.

Table 3.1: Trial log fields (one row per simulated game).

Field	Meaning / why we care
trial_id	Unique integer for traceability and reproducibility
seed	RNG seed (optional) to reproduce a suspicious run
prize_door	Where the prize was placed (should be uniform over {1,2,3})
player_door	The player's initial choice (should be uniform over {1,2,3})
reveal_door	Host's revealed goat door (must not equal prize_door or player_door)
decision	`stay` or `switch` (coin toss in our baseline simulation)
final_door	The door the player ends on after decision
win	`1` if final_door == prize_door, else `0`
strategy	`random`, `always_stay`, `always_switch` (for experiments)

3.3 What we aggregate as we go (running totals)

In addition to the per-trial log, we maintain running counters so we can: (1) print progress, (2) build plots efficiently, and (3) decide when we have enough trials.

At minimum, keep these four "outcome buckets":

- stay_win
- stay_lose
- switch_win
- switch_lose

Also keep distribution checks:

- counts of prize_door = 1,2,3
- counts of player_door = 1,2,3
- counts of decision stay vs switch

These sanity checks catch mistakes like "door 3 never gets the prize" or "my random choice is biased."

3.4 When do we stop? (confidence, not vibes)

We want students to see the difference between:

"We ran a bunch of trials." vs *"We ran enough trials to support a claim."*

We will use a confidence interval on a proportion (win-rate). Let $\hat{p}$ be the observed win-rate for a strategy, based on n trials. A simple (approximate) margin of error is:

$$\text{halfwidth} = z\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$$

where z matches the confidence level (e.g., 1.96 for 95%, 2.576 for 99%). [**?**]

3.4.1 A practical stopping rule

Pick:

- confidence level (90%, 95%, or 99%),
- a tolerance ϵ (example: $\epsilon = 0.01$ means $\pm 1\%$).

Stop when **both** strategy win-rates have halfwidth $\leq \epsilon$:

- halfwidth(switch win-rate) $\leq \epsilon$
- halfwidth(stay win-rate) $\leq \epsilon$

This makes the simulation self-aware: it runs until the estimates are sharp enough to be worth graphing.

3.4.2 Worst-case "how many trials might we need?"

Because $\hat{p}(1-\hat{p})$ is maximized at $\hat{p} = 0.5$, a conservative bound is:

$$n \geq \frac{z^2}{4\epsilon^2}$$

So for 99% confidence ($z \approx 2.576$) and $\epsilon = 0.01$:

$$n \gtrsim \frac{(2.576)^2}{4(0.01)^2} \approx 16588$$

That number is large on purpose: it teaches why "a few hundred trials" can still be noisy.

3.5 The Data-Collection FSM (EFSM style)

In Chapter 2 we noted that when we track extra variables like "prize door" and "player choice," we are really building an *extended* finite state machine (EFSM). :contentReference[oaicite:3]index=3 Our Data-Collection FSM keeps the same clean state backbone, but annotates each step with:

- which variables are set (prize_door, player_door, etc.)
- what gets logged at that step
- what gets aggregated

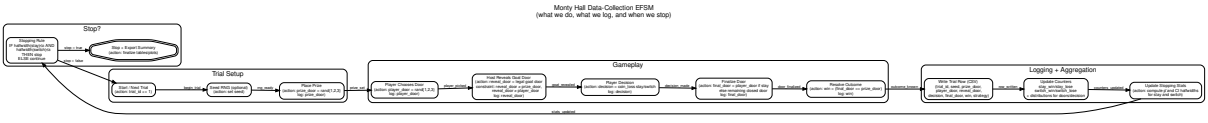

Figure 3.1: Data-Collection FSM for Monty Hall simulation (Graphviz-generated).

3.6 How this maps to code (preview of Chapter 4)

Your Chapter 2 "FSM-to-code" recipe still applies: state enum, events, transition function, and actions on transitions. :contentReference[oaicite:4]index=4 The only difference is that our actions now include:

- writing a trial row to CSV,
- updating running counters,
- computing confidence halfwidths and deciding whether to stop.

That is why we designed this chapter before coding: the FSM becomes the contract for both the implementation and the unit tests.

Chapter 4

Full ChatGPT Transcript and Reflection

4.1 Transcript

4.2 What worked well

Bullet points: prompt clarity, iteration loop, testing-as-design, diagram-first thinking, etc.

4.3 What we would do differently

How you'd tighten prompts, add constraints, verify claims, and improve reproducibility.

Bibliography

[1] Fsm notes in this project: clarity, testability, instrumentation. See Chapter 2 of this report. Internal project reference.

[2] Nist/sematech e-handbook of statistical methods: Confidence intervals. https://www.itl.nist.gov/div898/handbook/. Accessed: 2025-12-01.

[3] CBS. Let's make a deal. https://www.cbs.com/shows/lets_make_a_deal/. Accessed: 2025-12-01.

[4] Encyclopaedia Britannica. Monty hall problem. https://www.britannica.com/topic/Monty-Hall-problem. Accessed: 2025-12-01.

[5] PBS American Experience. The aftermath of the quiz show scandal. https://www.pbs.org/wgbh/americanexperience/features/quizshow-aftermath-quiz-show-scandal/. Accessed: 2025-12-01.

[6] Television Academy. Let's make a deal. https://interviews.televisionacademy.com/shows/lets-make-a-deal. Accessed: 2025-12-01.

[7] Television Academy. Monty hall. https://www.televisionacademy.com/bios/monty-hall. Accessed: 2025-12-01.

[8] The Strong National Museum of Play. Let's make a deal still a big deal. https://www.museumofplay.org/blog/lets-make-a-deal-still-a-big-deal/. Accessed: 2025-12-01.

[9] Wrapbook. How to account for prize money in your unscripted game show. https://www.wrapbook.com/blog/unscripted-game-show-prize-money. Accessed: 2025-12-01.