A high-level description of the game architecture

What you created in *The zoBITRONics Inc* was not just a text adventure, but a **general adventure engine** into which one particular story was plugged. The game logic is not hard-wired into the narrative; instead, the world is described as structured data, and the player's actions are interpreted and applied to that data through a small, rule-based system.

At the heart of the system is a **parser pipeline**. When the player types a command, the input is first cleaned up (articles like *the* or *a* are ignored), then analysed into parts:
a **verb**, a **direct object**, possibly a **preposition**, and an **indirect object**.
Adjectives (*magnetic card*, *Persian rug*) and references (*it*, *that*) are handled explicitly, and the parser keeps track of what the player referred to previously.

Once the words are identified, the game performs **reference resolution**. This means the system tries to determine *which actual thing in the game world* the player is talking about. It does this using clear priorities: objects in the current location are preferred over objects in the player's inventory, attributes must match, and ambiguous references produce clarification messages ("Which one?"). Objects carried by other characters are treated differently, so social constraints are respected.

The **game world itself** is modelled as a network of entities:

- **Places** (rooms),
- **Things** (objects),
- **Obstacles** (doors, windows, passages),
- **Persons** (NPCs).

All of these share a common structure: each entity knows where it is, what it contains, and how it is linked to other entities. Containment is explicit and hierarchical: objects can be inside other objects, on top of them, or under them, and visibility depends on whether containers are open or closed. This allows realistic reasoning about what the player can see or interact with.

Movement is handled in a particularly elegant way. Exits from rooms do not necessarily lead directly to other rooms; they can lead to **obstacles** instead. An obstacle then decides what happens: whether passage is allowed, what message is shown, and where the player ends up. This makes doors, windows, holes, and similar features first-class elements of the world rather than special cases.

Actions are executed through a **rule-matching system**. Each verb has a table of rules that describe *when* a particular routine should apply — for example, depending on the preposition used, the objects involved, or the current location. These rules are checked in order, and the first matching rule determines what happens. This makes the behaviour of verbs largely data-driven, and allows story-specific exceptions to coexist with generic "physics" rules like moving objects, opening containers, or checking capacity and weight.

Time is a core part of the architecture. The game maintains an internal clock that advances not only when the player acts, but also while the player is thinking. On top of this clock runs a **scheduled event system**: events are queued for specific times and triggered automatically when their time is reached. This allows things to happen independently of the player's commands, such as phones ringing, characters reacting, or the game eventually reaching its end state.

Non-player characters are handled as active entities with their own behaviour routines. At each cycle of the main loop, characters can react to the passage of time or to changes in the world, making the environment feel alive rather than static.

All of this is tied together by a simple but powerful **main loop**:

1. Describe the current situation.
2. Wait for user input while time advances.
3. Parse and resolve the command.
4. Execute the appropriate action.
5. Advance time and trigger events.
6. Let objects and characters perform their own actions.
7. Repeat.

In essence

You built a **data-driven, rule-based adventure engine** with:

- a structured natural-language parser,
- explicit world modelling and containment,
- declarative action rules instead of hardcoded logic,
- a real in-game clock and event scheduler,
- and autonomous world behaviour.

Seen from today's perspective, it closely parallels the design philosophy of Infocom-style interactive fiction systems — but achieved *without* a dedicated language or virtual machine, directly in assembler, using carefully designed data structures and conventions.