JPE XML Fork Language – High-Level Outline & PRD (v1.0)

1. High-Level Outline: JPE XML Fork Language

1.1 What JPE XML Is

Working name: JPE XML Fork (JPE-XML)
Tagline: “Write mods in plain English, export pure XML.”

Core idea

- A human-friendly XML dialect where:
  - Tag names and attributes are plain English.
  - Structure is still 100% valid XML, so tools and games can parse it.
  - Every JPE-XML file can be compiled / transformed into standard game XML (e.g., Sims 4 tuning XML).
  - 1:1 mapping between “Just Plain English” concepts and underlying XML structures.

1.2 Core Goals

- Beginner-first design
  - Assume user barely knows what “XML” is.
  - No required prior coding knowledge.
- Zero-loss translation
  - Anything expressible in raw XML must be expressible in JPE-XML.
  - Round-trip: Raw XML → JPE-XML → Raw XML without breaking the game.
- Readable & teachable
  - Files should read like structured instructions:

```
<mod name="Free Starting Money">
  <when event="game_starts">
    <action give_money="10000" to="active_household" />
  </when>
</mod>
```

- Tool-friendly
  - Strict schema, good validation, auto-completion support.
  - Easy to integrate into IDEs, GUIs, and CLI tools.

1.3 Key Components

1) Language Specification
- Tag set (e.g., <mod>, <when>, <condition>, <action>, <loot>, etc.).
- Attributes and allowed values.
- Types (strings, numbers, enums, lists).
- Mapping rules: JPE-XML → engine XML.

2) Transformation Engine
- Parser & validator for JPE-XML.
- Back-end that outputs standard game XML.
- Clear error messages that say “what you did” and “how to fix it”.

3) Tooling
- CLI: jpexml validate, jpexml build, jpexml explain.
- Editor support: snippets, templates, hover docs.
- Optional GUI “wizard builder” that generates JPE-XML.

4) Docs & Manuals (Beginner-Friendly)
- “What is XML?” explained in a few pages with pictures.
- Step-by-step “Your First Mod in JPE-XML”.

- Cookbook of copy-paste recipes.
- Error handbook: “What did I break and how do I unbreak it?”

1.4 Example High-Level Syntax (Conceptual)

You’re not coding yet, just showing the vibe:

```
<mod name="Welcome Gift" version="1.0">
  <description>Give new households a welcome bonus.</description>

  <when event="household_created">
    <if condition="is_active_household">
      <action type="give_money" amount="5000" to="household" />
      <notify player="true">Welcome to the neighborhood!</notify>
    </if>
  </when>
</mod>
```

2. PRD: JPE XML Fork Language

2.1 Background & Motivation

Problem:

- Modding is locked behind hostile XML and non-obvious schemas.
- Beginners struggle with:
  - Cryptic tags
  - Required attributes that aren’t obvious
  - Tiny typos that break the entire mod

Solution:

- Create a plain English XML fork (JPE-XML) that:
  - Hides low-level complexity behind human-readable tags.
  - Enforces valid structure using a strict schema.
  - Provides tools that teach as they validate.

2.2 Objectives

1) Language
- Define JPE-XML spec v1.0 that covers:
  - Core mod metadata
  - Events & triggers
  - Conditions
  - Actions
  - Simple data definitions (loot, traits, buffs, tuning hooks, etc.)
- Ensure 1:1 translation to engine XML for supported features.

2) Tooling
- Provide a CLI tool:
  - jpexml validate file.jpe.xml
  - jpexml build file.jpe.xml -o output_folder
  - jpexml explain error.log
- Provide basic editor integration (snippets + schema).

3) Documentation

- All manuals must be beginner-proof:
  - Simple language.
  - Tons of examples.
  - Step-by-step guides with screenshots (or at least placeholders for them).
- Documentation must cover:
  - “What is XML?”
  - “What is JPE-XML?”
  - “From zero to first mod in under 1 hour.”

2.3 Target Users

Primary
- Complete beginners to coding.
- Modders who currently only download mods and want to start editing/making their own.

Secondary
- Intermediate modders who want faster authoring and clearer structure.
- Tool developers integrating JPE-XML into bigger pipelines.

Assumption: Many users will only know “I open Notepad, I paste stuff, it works.” The system must respect that.

2.4 Scope (v1.0)

In scope
- Language for:
  - Basic mods: tuning swaps, small gameplay changes, buffs, traits, simple events.
  - Event/condition/action logic like:
    - when X happens
    - if conditions
    - do Y
- Transformation to a single target engine XML dialect (e.g., Sims 4 tuning).
- Beginner documentation & manuals.
- CLI + schema for editor assistance.

Out of scope (v1.0)
- Super complex AI/state machines.
- Full custom GUIs.
- Multi-game support (can be v2+).
- Advanced performance tuning or game-specific micro-optimizations.

2.5 Functional Requirements

2.5.1 Language Features

1) Mod Metadata
- <mod> root element (exactly one per file).
- Required attributes:
  - name
  - version
- Optional:
  - author
  - id (unique key)
  - game_version_min, game_version_max

2) Human-Readable Description
- <description> tag: free-form text.
- Recommended for every mod.

3) Events & Triggers
- <when> elements to define triggers.
  - Attribute event should be an enumerated value:
    - game_starts, lot_loaded, household_created, sim_added, etc.
- Optional target attribute for context.

4) Conditions
- <if> elements nested under <when> or <action_block>.
- Attributes or nested tags:
  - condition="is_active_household"
  - Or child elements:
    <if>
      <check type="sim_age" is="teen" />
      <check type="household_funds" greater_or_equal="5000" />
    </if>

5) Actions
- <action> tags with a type attribute.
  - give_money, add_buff, remove_buff, change_skill_level, show_notification, etc.
- Required and optional attributes defined per action type.

6) Simple Data Blocks
- <loot>, <buff>, <trait>, etc. as higher-level constructs that map to underlying tuning.
- Each with ID, name, description, and relevant fields.

7) Comments & Hints
- XML comments supported:
  <!-- This action gives newbies a starter bonus -->
- Requirement: Documenters must explain what comments are and how to use them for clarity and sanity.

2.5.2 Transformation Engine Requirements

1) Validation
- Validate against JPE-XML schema:
  - Missing required attributes → clear error.
  - Disallowed attributes → clear error.
  - Wrong types (string vs number vs enum) → clear error.

2) Compilation
- Convert JPE-XML → game XML:
  - Preserve IDs, references.
  - Generate correct namespaces and low-level tags.
- On success:
  - Output clearly:
    - Where the XML was written.
    - What game resources it defines.

3) Error Messages
- Must be plain English:
  - Bad: AttributeError: amount not found
  - Good: You used <action type="give_money"> without an "amount". Add amount="1000" (or another number) inside the <action> tag.

4) Explain Mode
- jpexml explain file.jpe.xml prints:
  - Summary in human language: “This mod gives new households 5000 funds when they are created.”
  - Per-section breakdown:
    - What <when> does
    - What each <if> and <action> does

2.5.3 Tooling & UX Requirements

1) CLI Tool (jpexml)
- Commands:
  - jpexml init – create barebones example mod.
  - jpexml validate path – validate and print friendly errors.
  - jpexml build path -o output – compile to game-ready XML.
  - jpexml explain path – describe the mod in plain English.

2) Editor + IDE Support
- XML schema file for JPE-XML.
- Snippets (e.g., type jpe-when and get a stubbed <when> block).
- Hover docs for common tags and attributes.

3) GUI Wizard (optional early prototype)
- Simple forms for:
  - “When should this happen?”
  - “What should happen?”
- GUI writes JPE-XML under the hood.

2.6 Non-Functional Requirements

- Beginner UX
  - No jargon in error messages without explanation.
  - Provide links or references like: “See Beginner Manual, Chapter 3: Events.”

- Stability
  - No partial builds. On any hard error, abort and say so clearly.

- Performance
  - Simple mods should build in under a second.

- Portability
  - Tools should run on Windows, macOS, and Linux (Python or Node baseline).

2.7 Documentation & Manuals (Dummy-Proof Set)

Each document should assume zero experience and use friendly examples.

1) JPE-XML 101: Beginner’s Manual
- What XML is (with diagrams).
- How tags work.
- What a JPE-XML file looks like vs raw game XML.
- First mod walkthrough:
  - Create a “Welcome Gift” mod step by step.
- Troubleshooting “why doesn’t my mod show up?”

2) Language Reference Manual
- Full tag list:
  - <mod>, <when>, <if>, <action>, <loot>, <buff>, etc.
- Per tag:
  - Description in English.
  - List of attributes with types and examples.
  - Example snippet.
- “If you’re not sure, copy this example and tweak numbers.”

3) Cookbook: Copy-Paste Recipes
- Chapters like:
  - “Give starting money”
  - “Apply a buff at 6 AM”
  - “Increase skill gain for teens”
  - “Show a funny notification on event X”
- Each recipe:
  - Short concept explanation.
  - JPE-XML snippet.
  - Notes: what to change.

4) Error Handbook
- Common error messages with:
  - What it means in plain language.
  - How to fix it.
- Sections:
  - “XML won’t load”
  - “Game doesn’t see my mod”
  - “My mod doesn’t do anything”

5) Tooling & Workflow Guide
- Install CLI.
- Basic commands with examples.
- Using an editor (VS Code / Notepad++ / similar) with JPE-XML.
- Simple workflow:
  1) Edit file
  2) Run validate
  3) Run build
  4) Drop into Mods folder
  5) Test in game

6) Developer Integration Guide (for tool devs)
- How to embed the parser / transformer in other tools.
- Command-line integration.
- File formats, options, and exit codes.

2.8 Roadmap (Example)

Phase 1 – Language Skeleton & Basic Docs
- Define minimal tag set.
- Implement parser + schema.
- Implement validate + build.
- Draft JPE-XML 101 Beginner Manual.

Phase 2 – Cookbook & Error UX
- Add more actions and conditions.
- Expand Cookbook with 10–20 recipes.

- Build Error Handbook.
- Improve error messages and explain.

Phase 3 – Tooling & Integration
- Editor schema + snippets.
- Simple GUI wizard prototype.
- Developer integration guide.

Phase 4 – Polish & v1.0 Release
- Full Language Reference Manual.
- Final testing with sample mods.
- Versioned spec and changelog.

2.9 Risks & Mitigations

- Risk: Beginners still get overwhelmed.
  Mitigation: Templates, examples, and “delete what you don’t need” patterns in docs.

- Risk: Underlying engine XML changes.
  Mitigation: Keep JPE-XML stable; update the compiler mappings version by version.

- Risk: People edit the compiled XML and break round-trip.
  Mitigation: Strong docs: “Edit your JPE-XML, not compiled XML.” Clear warnings in generated files.