prd08 diagnostics and exception translation

PRD 08 — Diagnostics and Exception Translation System

1. Product Vision
The Diagnostics and Exception Translation System turns raw parser, generator, validation, and plugin errors into clear, actionable feedback for users. It also produces machine-readable error reports for later inspection and integration and powers intelligent highlighting of problematic files.

2. Goals
- Translate low-level exceptions into human-readable explanations.
- Persist errors and warnings as structured output files per build.
- Intelligently highlight problematic files and resources in desktop and mobile UIs.

3. Inputs and Outputs
Inputs:
- Exceptions from XML parsing and generation.
- Exceptions from JPE and JPE-XML parsing.
- Validation failures from structural and semantic checks.
- Plugin execution errors.
- Cloud build failures.

Outputs:
- In-memory EngineError objects.
- Human-readable messages (short and long).
- Error report files (JSON and optional text) stored in project-level reports folders.
- Aggregated views for UIs.

4. Error Taxonomy
- Categories:
  PARSER_JPE
  PARSER_JPE_XML
  PARSER_XML
  VALIDATION_SCHEMA
  VALIDATION_SEMANTIC
  IO_FILE
  PLUGIN
  SYNC_CLOUD
- Severity:
  INFO, WARNING, ERROR, FATAL.

Each error must have a stable code, a category, and a severity.

5. EngineError Schema (Conceptual)
An EngineError object contains:
- code: a stable error code (e.g., E_JPE_UNEXPECTED_TOKEN).
- category: the category of error (e.g., PARSER_JPE).
- severity: INFO, WARNING, ERROR, or FATAL.
- message_short: a brief explanation.
- message_long: a detailed explanation with potential causes and remedies.
- file_path: the file that triggered the error, if applicable.
- resource_id: an optional logical resource identifier (interaction, buff, etc.).
- language_layer: which layer experienced the failure (JPE, JPE-XML, XML, IR, PLUGIN).
- position: optional line/column information.
- snippet: an excerpt of the offending text, if available.

- suggested_fix: optional recommendations on how to fix the issue.
- stack_trace_sanitized: an internal or user-friendly stack trace.
- plugin_id: optional ID of a plugin if the error originated from one.
- extra: additional structured metadata.

6. Error Reports
- For each build, the system may generate a JSON error report containing:
  build_id, project_id, status, and arrays of errors and warnings.
- Optionally generate a plain-text summary outlining the most important issues.
- Provide stable file naming and location conventions so UIs can discover reports easily.

7. Exception Translation Logic
- Normalize raw exceptions: capture type, message, and minimal stack trace.
- Classify errors by source and context.
- Map error to a template based on code, using human-friendly language.
- Enrich error with file path, resource information, source snippet, and potential fixes.

8. Intelligent File Highlighting
- Determine severity per file and folder based on contained errors and warnings.
- Expose APIs to:
  Get errors by file.
  Get errors by resource.
  Get summaries for a build.
- Support project tree highlighting in desktop and health indicators in mobile.

9. Integration with Desktop, Mobile, Cloud, and Plugins
- Desktop: problems pane, inline highlights, file tree indicators, and error report viewers.
- Mobile: project health markers, error lists, and drilldown into specific JPE lines.
- Cloud: build endpoints return serialized errors and warnings; history is available per project.
- Plugins: must return structured failures; plugin errors are tagged with plugin identifiers and surfaced like core errors.

10. Non-Functional Requirements
- Robustness: the diagnostics layer should not crash on malformed or unexpected error states.
- Clarity: error messages must be understandable even to users without deep technical background.
- Extensibility: new error codes and categories can be added without breaking existing clients.
- Observability: capture diagnostics metrics to inform quality improvements.