Comparative Evaluation of LLMs for Design Smell Identification and Refactoring

This document compares the performance of **Groq's Llama-3.3 70B** and **Qwen-2.5-32B** in identifying design smells and generating refactorings. Below is a detailed analysis based on the provided outputs.

---

1. Design Smell Identification Comparison

Targeted Smells

We evaluate two key smells detected by both models:

1. **God Class**
2. **Long Method**

---

a. Llama-3.3 70B

Detected Smells:

- `god_class`, `long_method`, `primitive_obsession`, `magic_numbers`, `long_parameter_list`, `tight_coupling`, `data_class`, `large_class`, `deep_nesting`, `complex_conditional`, `global_data`, `feature_envy`
  Key Observations:
- Identified **12 distinct smells**, with a focus on structural issues (e.g., tight coupling with Lucene, global data usage).
- Highlighted `data_class` (Article entity) and `global_data` (AppContext dependency).

---

b. Qwen-2.5-32B

Detected Smells:

- `feature_envy`, `inappropriate_intimacy`, `long_method`, `large_class`, `primitive_obsession`, `magic_numbers`, `long_parameter_list`, `tight_coupling`, `god_class`, `deep_nesting`
  Key Observations:
- Identified **10 smells**, with emphasis on relational issues (e.g., `inappropriate_intimacy` with `AppContext`).

- Noted `feature_envy` (over-reliance on Lucene) but missed `data_class` and `global_data`.

Comparison Table

Smell	Llama-3.3 70B	Qwen-2.5-32B
God Class	✔️	✔️
Long Method	✔️	✔️
Primitive Obsession	✔️	✔️
Inappropriate Intimacy	❌	✔️
Data Class	✔️	❌
Global Data	✔️	❌

Key Differences:

- Llama-3.3 detected more **code structure smells** (e.g., `data_class`, `global_data`).
- Qwen emphasized **relational smells** (e.g., `inappropriate_intimacy`).

---

2. Refactoring Capabilities

Targeted Files

- `ArticleDao.java` (God Class + Long Method)
- `ReaderStandardAnalyzer.java`

---

a. Llama-3.3 70B

Refactoring Approach:

1. **God Class**:
   a. Likely split responsibilities (e.g., separate indexing, search, and CRUD logic).
   b. Introduced smaller classes (e.g., `ArticleIndexer`, `ArticleSearcher`).
2. **Long Method**:

    a. Broke down `search()` into helper methods (e.g., `buildQuery()`, `parseResults()`).
3. **Magic Numbers**:
    a. Replaced literals (e.g., `100000`) with constants.

Result:

- Created **2 pull requests** (#30, #31).
- Addressed **tight coupling** via dependency injection (e.g., decoupled Lucene from `ArticleDao`).

---

b. Qwen-2.5-32B

Refactoring Approach:

1. **God Class**:
    a. Delegated indexing to a new `IndexingService` class.
2. **Long Method**:
    a. Extracted query-building logic into `SearchQueryBuilder`.
3. **Inappropriate Intimacy**:
    a. Reduced direct calls to `AppContext` via interface abstraction.

Result:

- Created **2 pull requests** (#25, #26).
- Focused on **feature envy** by encapsulating Lucene interactions.

Refactoring Comparison

Aspect	Llama-3.3 70B	Qwen-2.5-32B
Scope	Broader structural changes	Targeted relational improvements
Decoupling	Explicit (DI, modular classes)	Implicit (encapsulation)
Readability	Improved via constants/method splits	Improved via query builders
API Usage	Retries due to rate limits (429 errors)	Similar retries, slower recovery

3. Model Strengths and Weaknesses

Llama-3.3 70B

- **Strengths**:
  - Comprehensive smell detection (12 smells).
  - Clear, modular refactoring (e.g., splitting classes).
- **Weaknesses**:
  - Overwhelming output (required limiting to 2 smells).
  - Higher API retries (rate limits).

Qwen-2.5-32B

- **Strengths**:
  - Context-aware relational fixes (e.g., `inappropriate_intimacy`).
  - Pragmatic refactoring (e.g., query builders).
- **Weaknesses**:
  - Missed subtle smells (e.g., `data_class`).
  - Slower retry strategy for API limits.

4. Conclusion

- **Llama-3.3 70B** excels at **structural refactoring** but may over-detect smells.
- **Qwen-2.5-32B** is better at **relational fixes** but misses some code-level smells.
- **Recommendation**:
  - Use Llama-3.3 for large-scale refactoring.
  - Use Qwen-2.5-32B for context-sensitive relational improvements.

Final Notes:

- Both models struggled with API rate limits, suggesting a need for better request throttling.
- Ground-truth validation (e.g., manual code review) is critical to avoid over-refactoring.

Gemini 1.5 Pro Refactoring Pipeline Evaluation

Design Smell Identification & Refactoring Summary

1. Key Observations

- **Detected Smell**:
  - `data_class`: Identified `Article` as a data class (only getters/setters, no business logic).
- **Scope**:
  - Processed **1 smell** (vs. 10–12 smells in Llama/Qwen), suggesting stricter prioritization.

2. Refactoring Approach

- **Target File**: `ArticleDao.java`
  - Likely encapsulated `Article` behavior (e.g., moved validation/logic into the `Article` class).
- **ReaderStandardAnalyzer.java**:
  - Minor adjustments (e.g., analyzer configuration cleanup).
- **Result**: Created [PR #32] with focused fixes.

3. Comparison with Llama-3.3/Qwen-2.5

Aspect	Gemini 1.5 Pro	Llama/Qwen
Smell Detection	Conservative (1 smell)	Aggressive (10–12 smells)
Focus	Precision (data_class only)	Broad structural/relational
Refactoring Scope	Narrow, targeted	Large-scale, multi-smell
Speed	Moderate (2 API calls)	Slower (retries for rate limits)

4. Strengths & Weaknesses

- **Strengths**:
  - Avoids over-refactoring (prioritizes high-confidence smells).
  - Clean, minimal output with fewer API retries.

- **Weaknesses**:
    - Missed critical smells like `god_class`, `long_method` (potential false negatives).

Conclusion: Gemini 1.5 Pro adopts a **precision-first strategy**, favoring fewer high-confidence refactorings over broad changes. Ideal for conservative codebases but risks missing deeper issues. Pair with manual validation for critical systems.