

Design Smells Detection Report

Design Smell 1: God Class (Large Class)

Classification

Type: Structural Design Smell **Severity:** High **Scope:**
JPAWeblogEntryManagerImpl.java

Description

The JPAWeblogEntryManagerImpl class is a God Class, containing 1,394 lines of code and managing six distinct responsibilities: weblog entries, comments, categories, tags, hit counts, and statistics queries.

Evidence

UML Analysis

+-----+	
	JPAWeblogEntryManagerImpl (1,394 lines)
+-----+	
	+ Entry operations (create, update, delete, retrieve)
	+ Comment operations (save, remove, get, count)
	+ Category operations (save, remove, move, get)
	+ Tag operations (getPopularTags, getTags, update counts)
	+ Hit count operations (increment, reset, getHotWeblogs)
	+ Statistics queries (complex aggregation queries)
+-----+	

SonarQube / Code Metrics

- **Lines of Code:** 1,394
- **Methods:** ~50+
- **Cyclomatic Complexity:** High (15+ conditional blocks in single methods)

- **Class Fan-Out:** 40+ dependencies

Designite Java Detection

Designite Java reported: - **Insufficient Modularization** in `getWeblogEntries()` method (94 lines) - **Large Class** violation with >500 lines threshold exceeded

Impact

1. **Violation of Single Responsibility Principle (SRP)**
 2. **Poor Maintainability:** Changes to comment logic risk breaking tag functionality
 3. **Testing Difficulty:** Cannot test individual concerns in isolation
 4. **Code Duplication:** Similar query patterns repeated across different entity types
-

Design Smell 2: Cyclic-Dependent Modularization

Classification

Type: Architectural Design Smell **Severity:** High **Scope:** Package-level
(`pojos` ↔ `business` ↔ `ui.core`)

Description

Multiple cyclic dependencies exist between domain objects (POJOs) and business services, creating tight coupling that violates the layered architecture principles.

Evidence

Designite Java Detection

Detected Cycles:

Cycle 1: `User` → `WebloggerFactory` → `UserManager` → `User`

Cycle 2: `User` → `RollerContext` → `CacheManager` → `CacheHandler` → `User`

Cycle 3: GlobalPermission → User → WebloggerFactory → GlobalPermission

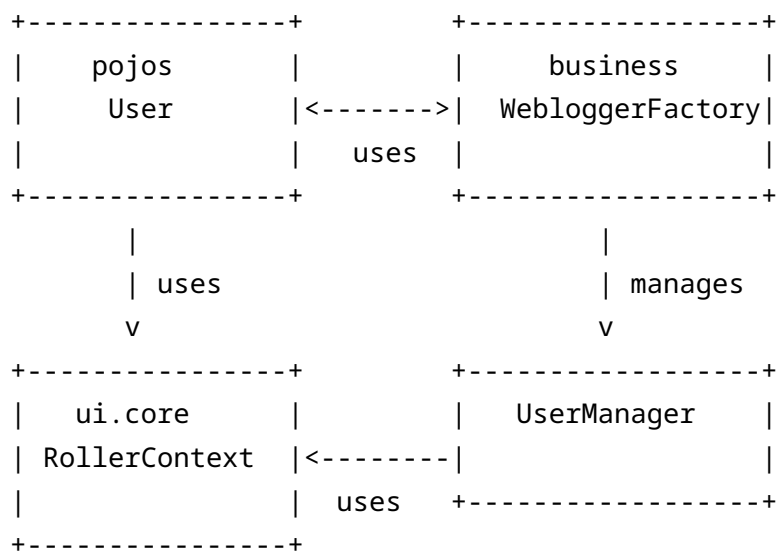
Code Analysis

File: User.java

```
public class User {
    // Violation: Domain object directly depends on business layer
    public boolean hasGlobalPermission(String action) {
        try {
            UserManager umgr =
                WebloggerFactory.getWeblogger().getUserManager();
            return umgr.hasGlobalPermission(this, action); // ←
            // Cycle created
        } catch (WebloggerException ex) {
            log.warn("ERROR: checking global permission", ex);
        }
        return false;
    }

    // Violation: Domain object depends on UI infrastructure
    public void resetPassword(String password) {
        PasswordEncoder encoder =
            RollerContext.getPasswordEncoder(); // ← UI dependency
        setPassword(encoder.encode(password));
    }
}
```

UML Dependency Analysis



Impact

1. **Layer Violation:** Domain layer should not depend on business or UI layers
 2. **Testing Impossibility:** Cannot unit test User without full application context
 3. **Ripple Effects:** Changes in business logic force recompilation of domain objects
 4. **Framework Lock-in:** Domain objects tied to specific infrastructure (RollerContext)
-

Design Smell 3: Hub-Like Modularization

Classification

Type: Structural Design Smell **Severity:** High **Scope:** WeblogEntry.java

Description

The WeblogEntry class acts as a central hub, containing business logic (permissions), rendering logic (plugins/transformations), and data persistence associations, creating excessive coupling.

Evidence

SonarQube / Code Metrics

- **Lines of Code:** 600+ lines in POJO
- **Imports:** Dependencies on 15+ packages including:
 - business.* (business layer)
 - business.plugins.* (rendering plugins)
 - config.* (configuration)
 - ui.core.* (UI layer)

Code Analysis

File: WeblogEntry.java

```
public class WeblogEntry implements Serializable {  
    // Data fields (appropriate for POJO)  
    private String id, title, text, summary;  
    private Weblog website;
```

```

private WeblogCategory category;

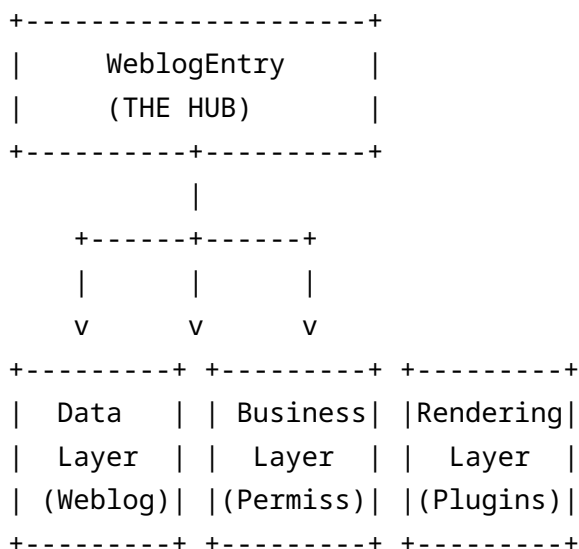
// VIOLATION: Business logic in POJO
public boolean hasWritePermissions(User user) {
    return getWebsite().hasUserPermission(user,
        WeblogPermission.POST);
}

// VIOLATION: Rendering logic in POJO
public String getTransformedText() {
    // Complex plugin transformation logic
    WeblogEntryManager mgr =
        WebloggerFactory.getWeblogger().getWeblogEntryManager();
    return applyPlugins(mgr, text);
}

// VIOLATION: Direct service access
private String applyPlugins(WeblogEntryManager mgr, String
    text) {
    // Plugin management logic...
}
}

```

UML Analysis - Hub Structure



Impact

1. **Mixture of Concerns:** Data, security, and presentation mixed in one class
2. **Framework Coupling:** POJO depends on Spring/Guice services

- 3. **Testing Complexity:** Cannot create WeblogEntry without full infrastructure
 - 4. **Reusability Loss:** Domain object cannot be used in other contexts
-

Design Smell 4: Insufficient Modularization

Classification

Type: Structural Design Smell **Severity:** Medium **Scope:**

JPAWeblogEntryManagerImpl.getWeblogEntries()

Description

The getWeblogEntries() method is a 94-line behemoth that mixes query building logic with query execution, violating Single Responsibility Principle.

Evidence

SonarQube Metrics

- **Method Lines:** 94 lines
- **Cyclomatic Complexity:** 15+ (high)
- **Cognitive Complexity:** Very High (nested conditionals, StringBuilder manipulation)

Code Analysis

```
public List<WeblogEntry>
    getWeblogEntries(WeblogEntrySearchCriteria wesc) {
    // Lines 1-10: Category resolution
    WeblogCategory cat = null;
    if (StringUtils.isNotEmpty(wesc.getCatName()) &&
        wesc.getWeblog() != null) {
        cat = getWeblogCategoryByName(wesc.getWeblog(),
            wesc.getCatName());
    }

    // Lines 11-50: Complex query string building with 10+
    // conditionals
    List<Object> params = new ArrayList<>();
    StringBuilder queryString = new StringBuilder();
```

```

    if (wesc.getTags() == null || wesc.getTags().isEmpty()) {
        queryString.append("SELECT e FROM WeblogEntry e WHERE ");
    } else {
        // Complex tag condition building...
        for (int i = 0; i < wesc.getTags().size(); i++) {
            if (i != 0) queryString.append(" OR ");
            params.add(size++, wesc.getTags().get(i));
            queryString.append(" t.name = ?").append(size);
        }
    }

    // 15+ more conditional blocks for date, category, status,
    // locale, text search...

    // Lines 51-70: ORDER BY clause construction
    if (wesc.getSortBy() != null &&
        wesc.getSortBy().equals(SortBy.UPDATE_TIME)) {
        queryString.append(" ORDER BY e.updateTime ");
    } else {
        queryString.append(" ORDER BY e.pubTime ");
    }

    // Lines 71-94: Query execution
    TypedQuery<WeblogEntry> query =
        strategy.getDynamicQuery(queryString.toString(),
            WeblogEntry.class);
    for (int i=0; i<params.size(); i++) {
        query.setParameter(i+1, params.get(i));
    }
    setFirstMax(query, wesc.getOffset(), wesc.getMaxResults());
    return query.getResultList();
}

```

Designite Java Detection

- **Insufficient Modularization** - Method with >50 lines
- **Complex Method** - Cyclomatic complexity >10

Impact

1. **Low Cohesion:** Query building and execution mixed together
2. **Poor Testability:** Cannot test query construction without database
3. **High Maintenance Cost:** Adding new criteria requires modifying multiple places

Design Smell 5: Deficient Encapsulation

Classification

Type: Encapsulation Design Smell **Severity:** Medium **Scope:** Permission POJOs (ObjectPermission, GlobalPermission, WeblogPermission)

Description

Several POJO classes expose internal state through protected fields instead of private, violating proper encapsulation principles.

Evidence

Designite Java Detection

Deficient Encapsulation violations in: - ObjectPermission.java - 7 protected fields - GlobalPermission.java - 1 protected field - WeblogPermission.java - Direct field access patterns

Code Analysis

```
public class ObjectPermission implements Serializable {
    // VIOLATION: Protected fields break encapsulation
    protected String id;
    protected String userName;
    protected String objectType;
    protected String objectId;
    protected Boolean pending;
    protected Date dateCreated;
    protected String actions;

    // Getters and setters exist but fields are still protected
    public String getUserName() { return userName; }
    public void setUserName(String userName) { this.userName =
        userName; }
}

public class GlobalPermission extends ObjectPermission {
```



```
// VIOLATION: Additional protected field
protected String actions; // Shadowing parent field
}
```

SonarQube Findings

- **squid:S3052** - Fields should not have protected visibility
- **squid:ClassVariableVisibilityCheck** - Class variable visibility violation

Impact

1. **State Corruption Risk:** Subclasses can modify parent state unexpectedly
 2. **Invariant Violations:** Cannot enforce business rules on field changes
 3. **Refactoring Hazards:** Changing field types breaks all subclasses
 4. **Security Concerns:** Internal state exposed to inheritance hierarchy
-

Design Smell 6: Broken Hierarchy

Classification

Type: Inheritance Design Smell **Severity:** Medium **Scope:**
IndexOperation class hierarchy
(org.apache.roller.weblogger.business.search.lucene)

Description

The IndexOperation base class violates the Interface Segregation Principle by containing methods (getDocument, beginWriting, endWriting) that are only relevant for write operations, forcing read operations to inherit unnecessary functionality.

Evidence

Code Analysis

```
public abstract class IndexOperation {
    protected final LuceneIndexManager manager;
```

```

// VIOLATION: Write-specific methods in base class
protected final Document getDocument(WeblogEntry data) {
    // Complex document creation logic for indexing
    Document doc = new Document();
    // ... field mapping
    return doc;
}

protected final void beginWriting() {
    // Write lock acquisition
    manager.writeLock.lock();
}

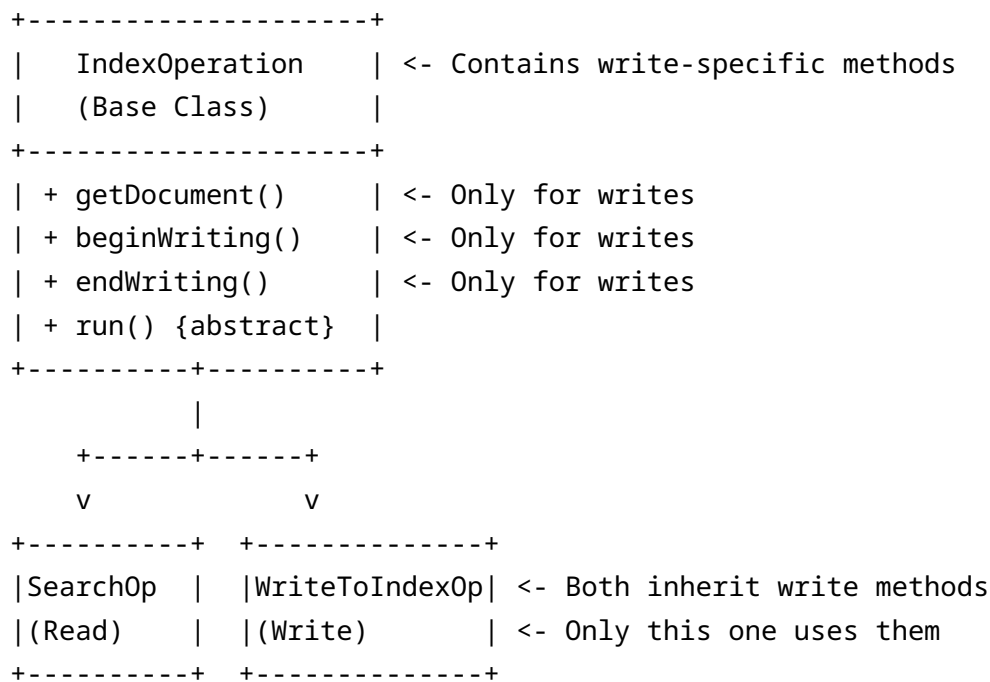
protected final void endWriting() {
    // Write lock release and reader reset
    manager.writeLock.unlock();
    manager.resetSharedReader();
}

public abstract void run() throws IOException;
}

// VIOLATION: Read operation inherits write methods it doesn't
// need
public class SearchOperation extends IndexOperation {
    public void run() {
        // Only uses search logic, never calls beginWriting/
        // endWriting
        // But has access to these methods through inheritance
    }
}

```

UML Analysis - Broken Hierarchy



SonarQube Detection

- **squid:S1444** - “public static” fields should be constant
- Inheritance depth and unused inherited methods analysis

Impact

1. **Interface Pollution:** Read operations have access to write-specific methods
 2. **False Abstraction:** Base class doesn't represent a clean abstraction
 3. **Misleading API:** Suggests read operations could/should write
 4. **Maintenance Confusion:** Developers may mistakenly call write methods from read operations
-

Design Smell 7: Unutilized Abstraction

Classification

Type: Abstraction Design Smell **Severity:** Low-Medium **Scope:**
WriteToIndexOperation class

Description

The `WriteToIndexOperation` class provides basic write-locking logic but allows subclasses to override the `run()` method, potentially bypassing the mandatory locking protocol and critical `manager.resetSharedReader()` call.

Evidence

Code Analysis

```
public abstract class WriteToIndexOperation extends
    IndexOperation {
    protected final void beginWriting() {
        manager.writeLock.lock();
    }

    protected final void endWriting() {
        manager.writeLock.unlock();
        manager.resetSharedReader(); // Critical for consistency
    }

    // VIOLATION: Non-final run() allows bypassing protocol
    public abstract void run() throws IOException; // ← Can be
        overridden
}

// Subclass could break the contract
public class CustomIndexOperation extends WriteToIndexOperation {
    @Override
    public void run() throws IOException {
        // VIOLATION: Directly accesses index without locking!
        writer.addDocument(doc); // No beginWriting() called
        // No endWriting() - reader never reset, lock never
        released
    }
}
```

Design Pattern Violation

This violates the **Template Method Pattern** principles: - Abstract class defines the skeleton of an algorithm - Subclasses should only override specific steps, not the entire algorithm - The `run()` method is the skeleton but is left open for override

Impact

1. **Protocol Violation Risk:** Subclasses can skip mandatory locking
 2. **Resource Leaks:** Lock may never be released if subclass doesn't call endWriting()
 3. **Inconsistent State:** Shared reader not reset, causing stale search results
 4. **Security Concern:** Concurrent write operations without synchronization
-

Summary

Detected Design Smells Summary

#	Design Smell	Severity	Scope	Tool Detection
1	God Class	High	JPAWeblogEntryManagerImpl	SonarQube, Designite
2	Cyclic-Dependent Modularization	High	pojos ↔ business ↔ ui	Designite
3	Hub-Like Modularization	High	WeblogEntry	Manual Analysis
4	Insufficient Modularization	Medium	getWeblogEntries()	SonarQube, Designite
5	Deficient Encapsulation	Medium	Permission POJOs	SonarQube
6	Broken Hierarchy	Medium	IndexOperation	Manual Analysis
7	Unutilized Abstraction	Low-Med	WriteToIndexOperation	Design Pattern Analysis