2A)Design Smells Analysis Report for RSS-Reader

Project: RSS-Reader
Analysis Tools: SonarQube, DesigniteJava
Prepared by: C Swaroop (Team 27)

---

1. Introduction

This report summarizes the design smells identified in the RSS-Reader project by combining insights from SonarQube—which provided quantitative metrics (Lines of Code, Security, Reliability, Maintainability, Security Hotspots, Coverage, and Duplications)—and DesigniteJava—which flagged specific design issues. The goal is to detect seven design smells that indicate potential violations of fundamental design principles, thereby hindering maintainability, testability, and scalability. Each identified design smell is documented with supporting evidence from both quantitative metrics and design flags, along with targeted refactoring recommendations.

---

2. Project Metrics Overview

The following tables summarize the current quantitative metrics for the key production modules and their submodules:

Overall Module Metrics

Module	Lines of Code	Security	Reliability	Maintainability	Security Hotspots	Coverage
Reader Parent	16,610	5	50	752	21	0.0%
reader-core	8,161	5	13	353	12	0.0%
reader-web	7,360	0	36	376	8	0.0%
reader-web-common	648	0	1	23	1	0.0%

Detailed Metrics for reader-core Submodules

Submodule	Lines of Code	Security	Reliability	Maintainability	Security Hotspots	Cover
reader/core	6,252	2	11	204	10	0.0%
constant	21	0	0	1	1	–

dao	3,238	2	1	110	5	0.0%
event	140	0	0	0	0	0.0%
listener	511	0	3	43	4	0.0%
model	1,230	0	1	3	0	0.0%
service	521	0	1	23	0	0.0%
util	591	0	5	24	0	0.0%

Detailed Metrics for reader-rest Packages

Package	Lines of Code	Security	Reliability	Maintainability
java/com/sismics/reader/rest	1,970	0	2	58
assembler	31	0	0	1
constant	6	0	0	0
dao	38	0	0	4
resource	1,851	0	2	51
util	44	0	0	2

Package	Lines of Code	Security	Reliability	Maintainability	Securit Hotspot
main/java/com/sismics	544	0	1	18	1
rest	181	0	0	10	1
security	112	0	0	0	0
util/filter	251	0	1	8	0

---

3. Identified Design Smells

3.1 Cyclic-Dependent Modularization

Definition:
Cyclic-Dependent Modularization occurs when classes or modules depend on each other in a circular fashion, which increases coupling and reduces overall system robustness.

Supporting Evidence:

- **DesigniteJava:**
  - Classes such as `ReaderAgent` and `ReaderDeployer` (from the agent package) and related UI classes (e.g., `SettingPanel`, `StatusPanel`, `TrayController`, `AgentFrame`) are flagged for cyclic dependencies.
- **Module Metrics:**

    - *reader-core* shows 8,161 LOC with high maintainability (353), and *reader-web* shows 7,360 LOC with 376. These high values are indicative of the intertwined dependencies that can lead to cycles.

Impact:

- Tight coupling across modules leads to ripple effects when changes are made.
- Increases complexity and makes maintenance and testing more difficult.

Refactoring Recommendations:

- Introduce abstraction layers such as interfaces or façade classes to decouple modules.
- Refactor classes to minimize unnecessary interconnections.

3.2 Deficient Encapsulation

Definition:
Deficient Encapsulation arises when a class exposes its internal state too broadly, enabling unintended modifications and compromising data integrity.

Supporting Evidence:

- **DesigniteJava:**
    - Classes like `Setting` (in the agent model) and others in the security and filter packages are flagged for exposing internal state.
- **Module Metrics:**
    - The overall complexity in *reader-core* and *reader-web* (as shown by their metrics) suggests that data is not sufficiently protected within these modules.

Impact:

- Exposing internal state can lead to side effects and bugs.
- It increases the maintenance overhead as any internal change might have cascading effects.

Refactoring Recommendations:

- Change field visibility to `private` and use getter/setter methods to control access.
- Where possible, refactor classes to be immutable.

3.3 Insufficient Modularization

Definition:
Insufficient Modularization occurs when a module handles multiple unrelated responsibilities, resulting in a monolithic design.

Supporting Evidence:

- **DesigniteJava:**
    - The `Setting` class has been flagged for handling diverse responsibilities.
- **Module Metrics:**

- The *reader-core* (8,161 LOC) and *reader-web* (7,360 LOC) modules contain a high number of maintainability. These figures suggest that too many concerns are being mixed within single modules.

Impact:

- Difficult to understand and maintain a module that serves too many purposes.
- Increases the likelihood of unintended side effects when modifying one part of the module.

Refactoring Recommendations:

- Break large modules into smaller, focused submodules.
- Separate unrelated concerns into dedicated classes or services.

3.4 Unutilized Abstraction

Definition:
Unutilized Abstraction is present when abstract classes or interfaces are defined but not effectively used by their implementations, resulting in unnecessary complexity.

Supporting Evidence:

- **DesigniteJava:**
  - Several REST-related classes (e.g., `BaseJerseyTest`, `TestStarredResource`, `TestCategoryResource`, etc.) have been flagged for unutilized abstraction.
- **Module Metrics:**
  - The REST package in `java/com/sismics/reader/rest` has 1,970 LOC, but its deep inheritance hierarchies do not translate into significant reuse, suggesting that the abstract layers are underutilized.

Impact:

- Introduces extra layers without yielding benefits, increasing cognitive overhead.
- Makes the inheritance structure more complicated than necessary.

Refactoring Recommendations:

- Remove or consolidate abstract classes/interfaces that do not add clear value.
- Favor composition over inheritance for achieving code reuse and flexibility.

3.5 Imperative Abstraction

Definition:
Imperative Abstraction forces subclasses to follow a rigid, predetermined implementation as defined by their abstract superclass or interface, thereby limiting flexibility.

Supporting Evidence:

- **DesigniteJava:**
  - Certain REST resource classes (e.g., `TestStarredResource` and `TestCategoryResource`) are flagged for imposing overly rigid abstraction.
- **Module Metrics:**

    - The complexity in REST modules (with 1,970 LOC and related packages showing additional complexity) indicates that such rigid abstractions add extra boilerplate and limit customization.

Impact:

- Restricts the ability of subclasses to modify behavior.
- Leads to higher maintenance costs due to inflexible design.

Refactoring Recommendations:

- Simplify abstraction layers and consider using composition instead of deep inheritance to allow for more flexible implementations.

3.6 Broken Hierarchy

Definition:
Broken Hierarchy is evident when the inheritance structure fails to enforce a consistent and predictable contract across classes, resulting in divergent behavior among subclasses.

Supporting Evidence:

- **DesigniteJava:**
    - Several REST resource classes (such as `TestStarredResource`, `TestCategoryResource`, `TestThemeResource`, etc.) are flagged for broken hierarchy.
- **Module Metrics:**
    - The deep inheritance trees in REST modules, with sub-packages like assembler, dao, and resource, indicate inconsistencies that undermine the intended polymorphic behavior.

Impact:

- Produces unpredictable behavior across the codebase.
- Increases the maintenance burden due to inconsistent class contracts.

Refactoring Recommendations:

- Flatten the inheritance hierarchy by eliminating unnecessary levels.
- Reevaluate base classes to ensure that all subclasses conform to a consistent contract.
- Replace rigid hierarchies with interfaces and composition where appropriate.

3.7 Wide Hierarchy

Definition:
Wide Hierarchy refers to an excessively broad or deep inheritance structure that complicates the understanding and maintenance of class relationships.

Supporting Evidence:

- **DesigniteJava:**
    - Base classes in the REST packages (e.g., `BaseJerseyTest` and `BaseResource`) have been flagged for wide hierarchy.
- **Module Metrics:**

    - The REST package (1,970 LOC) and its associated submodules (e.g., resource package with 1,851 LOC and 9.4% duplications) highlight an overly extensive inheritance structure.

Impact:

- Increases cognitive load on developers.
- Changes to base classes may have far-reaching unintended effects.

Refactoring Recommendations:

- Simplify the inheritance chain by reducing the number of levels.
- Consolidate common functionality into fewer, more coherent base classes.
- Consider using composition as a more flexible alternative to deep inheritance.

---

4. Conclusion

Based on the combined insights from SonarQube and DesigniteJava, the following seven design smells have been identified in the production code of the RSS-Reader project:

1. **Cyclic-Dependent Modularization**
2. **Deficient Encapsulation**
3. **Insufficient Modularization**
4. **Unutilized Abstraction**
5. **Imperative Abstraction**
6. **Broken Hierarchy**
7. **Wide Hierarchy**

These issues are substantiated by both quantitative metrics (for example, *reader-core* has 8,161 LOC with 338 code smells and 4.6% duplications, while *reader-web* has 7,360 LOC with 347 code smells and 1.9% duplications) and qualitative flags from DesigniteJava. Addressing these design smells through targeted refactoring will lead to improved maintainability, testability, and scalability of the system.

Prepared by: C Swaroop (Team 27)