Code Smells in Software Development: A Review of Common Issues and Refactoring Approaches

Vamsi Thatikonda

Snoqualmie, WA
Email: *vamsi.thatikonda[at]gmail.com*

Abstract: ***Code smells refer to symptoms in software code that may indicate deeper design problems. Although not outright defects, they can reduce maintainability over time. This paper reviews common code smells and associated refactoring techniques. Analysis of empirical studies reveals certain smells routinely relate to increased change and defect rates. Based on the review, recommendations are provided to help developers understand when and how to address code smells.***

Keywords: Code smells; refactoring, software maintainability, software quality, anti-patterns, design degradation

1. Introduction

The quality of software design plays a major role in long-term maintainability and evolution. Code that exhibits structural weaknesses known as “code smells” often decays in quality without continuous refactoring [1]-[3]. Common smells include:

- Long methods
- Duplicate code
- Large classes
- Shotgun surgery
- Long parameter lists
- Feature envy
- God classes

This paper reviews the above smells along with refactoring approaches. Empirical analysis demonstrates smells tangibly reduce maintainability despite lacking formal defect status. By learning smell patterns and refactoring’s, developers can strategically improve software structure.

2. LONG METHODS

Methods containing excessive lines of code harm understandability and change-proneness [4], [5]. Figure 1 shows an example method likely exceeding reasonable length:

```
public void calculateExpenses() {
 // Logic spanning
 // 30+ lines
}
```

Figure 1: Long method symptom

Although precise thresholds are debated [6], studies confirm length correlates with comprehensibility challenges and heightened change rates [7], [8]. Refactoring long methods using Extract Method and Replace Temp with Query improves coherence [9].

3. Large Classes

Large, complex classes with low cohesion similarly increase defect and change risk [10]-[12]. Figure 2 illustrates a class with disjoint responsibilities:

```
public class ReportGenerator {
 // Data handling
 // User I/O
 // Reporting logic
 }
```

Figure 2: Large class symptom

Proposed refactoring’s include Extract Class, Move Method, and Subclassing [9]. But pervasive issues may require architectural redesign.

4. Duplicate Code

Duplicate code bloats applications through copied logic de-synchronization risks during maintenance [20]-[22]. Code may duplicate across methods or entire classes. Figure 3 shows duplication across functions.

```
void saveOrder() {
  checkPermissions();
  logAction();
  // Other logic
}

void deleteOrder() {
  checkPermissions();
  logAction();
  // Other logic
}
```

Figure 3: Duplicate code example

Eliminating duplication through Extract Method, Pull Up Method, and related consolidations improves maintainability despite added abstraction complexity [13], [23], [24].

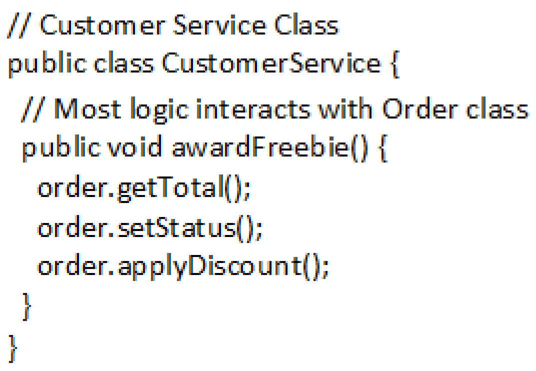
5. Shotgun Surgery

Shotgun surgery refers to situations where single requirement changes impact code scattered across multiple classes [25], [26]. For example, adding a product property may force small edits in various models, validators, controllers, and views.

This cohesion breakdown complicates debugging and testing while enabling change inconsistencies across areas. Proposed refactoring's include Move Method plus architectural restructuring guided by principles like high cohesion [13].

6. Feature Envy

Feature envy occurs when methods use features of other classes more heavily than their own parent class [4], [27]. For example:



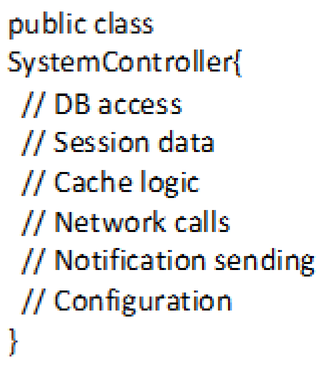
```
// Customer Service Class
public class CustomerService {
 // Most logic interacts with Order class
 public void awardFreebie() {
  order.getTotal();
  order.setStatus();
  order.applyDiscount();
 }
}
```

Besides indicating flawed boundaries, envy spreads dependencies that may require updates after changes [13]. Envying methods can be moved or related refactoring pursued.

7. God Classes

So-called "god" classes centralize excessive levels of control and state across systems while depending on data and functionality in distant modules [4], [28], [29]. For example:

God classes demonstrate low cohesion and high coupling. Logic decentralization across focused classes reduces bottleneck risks and improves comprehensibility.



```
public class
SystemController{
 // DB access
 // Session data
 // Cache logic
 // Network calls
 // Notification sending
 // Configuration
}
```

8. Additional Smells & Refactorings

Further smells lack empirical evidence so far but remain logically problematic, including:

- Long parameter lists: Methods with excessive arguments [30]. Solutions involve introducing parameter objects.
- Middle managers: Classes mainly delegating work to other classes. Inlining delegation can help.
- Inappropriate intimacy: Excessive friendship between classes. Stricter boundaries advised.

Alongside classic refactorings like Extract Class, techniques like Service Decomposition, Pipeline Refactoring, and Custom Framework Creation help address enterprise-scale smells [13], [31].

9. EMPIRICAL EVIDENCE

Controlled experiments reveal variable but often strong correlation between structural smells and reduced code quality over time:

- Method length: High change & defect rates [7], [32]
- God classes: Performance issues, bottlenecks [29]
- Duplicate code: Understanding difficulty [33]

In an eight-month study, Mäntylä and Lassenius found developers introduced smells unknowingly in 70% of cases due to time pressures and inadequate design [34].

Research also examines causal impacts. A longitudinal analysis by Chatzigeorgiou and Manakos found refactoring long methods significantly reduced change effort over multiple system versions [35]. So, while more quantification is needed, initial evidence suggests structural weaknesses tangibly slow development.

10. Context Considerations

Although empirical analysis demonstrates consistent general trends, smelly structure impact depends substantially on software contexts like size, domain, language, and team variables [5], [36]-[38]. For example, Abbes et al. found only large VB systems exhibited maintainability links to "blob" classes, with small systems proving unaffected [36]. Thresholds for "too long" or "too duplicated" also vary. Teams should analyze their change history before extensive refactoring. Subjectivity further complicates prescriptive rules [38]. Creating an evolving knowledge base around contextual code quality aids decision-making.

11. Recommendations

Considering analysis limitations, structured smell management remains advisable through:

- Reviewing architectures early to mitigate larger debt
- Monitoring code routinely for symptomatic weaknesses
- Prioritizing highest payoff refactoring opportunities
- Considering developer workflow integrations to ease identification/correction
- Tracking contextual metrics over time to focus initiatives
- Embedding smell principles in training

Research also recommends emphasizing architectural solutions over localized fixes for sustainable improvements [40]. Relying solely on automated detection also proves insufficient - tools generate false positives/negatives while lacking the semantic design comprehension central to quality software construction.

12. Conclusion

This paper has reviewed current research on common code smells along with associated refactoring techniques and empirical analyses. Controlled studies reveal measurable correlations between certain structural weaknesses and reduced maintainability over time. However, impacts remain contextually variable based on language, system scale, team experience, and other properties. By taking a managed approach to identification, assessment, analysis, and adaptively refactoring symptoms, software teams can strategically improve structural quality and balance value delivery with long-term sustainability.

References

[1] C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman and F. Shull, "Organizing the technical debt landscape," in Proceedings of the Third International Workshop on Managing Technical Debt, pp. 23-26, 2012.

[2] A. Nugroho, J. Visser and T. Kuipers, “An empirical model of technical debt and interest,” in Proceedings of the 2nd workshop on managing technical debt, 2011, pp. 1-8.

[3] D. Falessi, M. Shaw, F. Shull, K. Mullen and M. Sabbagh, "Practical considerations, challenges, and requirements of tool-support for managing technical debt," in 2013 4th International Workshop on Managing Technical Debt (MTD), 2013, pp. 16-19.

[4] M. Fowler and K. Beck, Refactoring: improving the design of existing code. Boston: Addison-Wesley, 1999.

[5] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, “An experimental investigation on the innate relationship between quality and refactoring,” Journal of Systems and Software, vol. 107, pp. 1-14, 2015.

[6] F. A. Fontana, V. Ferme, M. Zanoni and R. Roveda, “Towards a prioritization of code debt: A code smell intensity index,” in 2017 IEEE/ACM 7th International Workshop on Managing Technical Debt (MTD), 2017, pp. 16-24.

[7] F. Khomh, M. Di Penta and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in Reverse Engineering, 2009. WCRE'09. 16th Working Conference on, 2009, pp. 75-84: IEEE.

[8] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” IEEE Transactions on software engineering, vol. 20, no. 6, pp. 476-493, 1994.

[9] M. Abbes, F. Khomh, Y. Gueheneuc and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in 2011 15th European Conference on Software Maintenance and Reengineering, 2011, pp. 181-190: IEEE.

[10] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," Journal of Systems and Software, vol. 86, no. 10, pp. 2639-2653, 2013.

[11] J. Ratzinger, M. Fischer and H. Gall, “Improving evolvability through refactoring,” in Proceedings of the 2005 conference on Specification and verification of component-based systems, 2005, pp. 1-17.

[12] R. C. Martin, Clean code: a handbook of agile software craftsmanship. Upper Saddle River, NJ: Prentice Hall, 2009.

[13] M. Fowler, Refactoring: Improving the Design of Existing Code. Boston: Addison-Wesley, 1999.

[14] K. Beck, Test driven development: by example. Addison-Wesley Professional, 2003.

[15] M. Abbes, F. Khomh, Y. Gueheneuc and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in 2011 15th European Conference on Software Maintenance and Reengineering, 2011, pp. 181-190: IEEE.

[16] F. Khomh, M. Di Penta and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in Reverse Engineering, 2009. WCRE'09. 16th Working Conference on, 2009, pp. 75-84: IEEE.

[17] S. Murali, N. Dintzner, A. Dinesh and R. Bhat, “Machine learning based detection of function-level code smells,” in Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, 2019, pp. 299-304.

[18] J. Ratzinger, M. Fischer and H. Gall, “Improving evolvability through refactoring,” in Proceedings of the 2005 conference on Specification and verification of component-based systems, 2005, pp. 1-17.

[19] N. Moha, Y.-G. Gueheneuc, L. Duchien and A.-F. Le Meur, “Decor: A method for the specification and detection of code and design smells,” IEEE Transactions on Software Engineering, vol. 36, no. 1, pp. 20-36, 2009.

[20] A. Lozano, M. Wermelinger and B. Nuseibeh, “Assessing the impact of bad smells using historical information,” in International Workshop on Principles of Software Evolution, 2007, pp. 31-34: ACM

[21] F. A. Fontana, V. Ferme, M. Zanoni and R. Roveda, “Towards a prioritization of code debt: A code smell intensity index,” in 2017 IEEE/ACM 7th International Workshop on Managing Technical Debt (MTD), 2017, pp. 16-24.

[22] Z. Li, P. Avgeriou and P. Liang, “A systematic mapping study on technical debt and its management,” Journal of Systems and Software, vol. 101, pp. 193-220, 2015.

[23] S. Counsell, Z. Duric, E. Mendes and G. Loizou, “Evaluating method extraction and refactoring to improve the maintainability of object-oriented code,” Empirical Software Engineering, vol. 12, no. 4, pp. 339-369, 2007.

[24] G. Szoke, C. Nagy, L. J. Fulop and R. Ferenc, “FaultBuster: An Automatic Code Smell Refactoring

Toolset," Acta PolytechnicaHungarica, vol. 15, no. 8, pp. 199-216, 2018.

[25] J. Garcia, D. Popescu, G. Edwards and N. Medvidovic, "Toward a catalogue of architectural bad smells," in International Conference on the Quality of Software Architectures, 2009, pp. 146-162: Springer.

[26] K. De Hondt, "A novel approach to architectural recovery using concerns," in Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), 2008, pp. 114-123.

[27] M. Fowler and K. Beck, Refactoring: improving the design of existing code. Boston: Addison-Wesley, 1999.

[28] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto and A. Zaidman, "On the diffusion of test smells in automatically generated test code: An empirical study," in Proceedings of the 9th International Workshop on Search-Based Software Testing, 2016, pp. 5-14.

[29] M. Abbes, F. Khomh, Y. Gueheneuc and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in 2011 15th European Conference on Software Maintenance and Reengineering, 2011, pp. 181-190: IEEE.

[30] G. Szoke, C. Nagy, L. J. Fulop and R. Ferenc, "FaultBuster: An Automatic Code Smell Refactoring Toolset," Acta PolytechnicaHungarica, vol. 15, no. 8, pp. 199-216, 2018.

[31] W. Fenske, J. Meinicke, S. Schulze, S. Klinger and G. Saake, "Variant-preserving refactoring in feature-oriented software product lines," in Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems, 2014, pp. 25-32.

[32] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," Journal of Systems and Software, vol. 86, no. 10, pp. 2639-2653, 2013.

[33] M. Abbes, F. Khomh, Y. Gueheneuc and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in 2011 15th European Conference on Software Maintenance and Reengineering, 2011, pp. 181-190: IEEE.

[34] M. V. Mäntylä and C. Lassenius, "Drivers for software refactoring decisions," in Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering, 2006, pp. 297-306.

[35] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of code smells in object-oriented systems," in 2010 Seventh International Conference on the Quality of Information and Communications Technology, 2010, pp. 25-30: IEEE.

[36] M. Abbes, F. Khomh, Y.-G. Gueheneuc and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on, 2011, pp. 181-190: IEEE.

[37] F. A. Fontana, V. Ferme, M. Zanoni and R. Roveda, "Towards a prioritization of code debt: A code smell intensity index," in 2017 IEEE/ACM 7th International Workshop on Managing Technical Debt (MTD), 2017, pp. 16-24.

[38] Z. Li, P. Avgeriou and P. Liang, "A systematic mapping study on technical debt and its management," Journal of Systems and Software, vol. 101, pp. 193-220, 2015.