Designite: A Customizable Tool for Smell Mining in C# Repositories

Tushar Sharma

Dept of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
tushar@aueb.gr

Abstract

Code smells indicate the presence of quality issues in a software system. For a thorough large scale smell mining study, researchers require tools that not only allow them to detect a wide range of smells in a large number of repositories automatically but also offer mechanisms to customize the analysis. In this paper, we present a tool *Designite* that detects 19 design and 11 implementation smells for source code written in C# programming language. Designite provides a command line tool, in addition to an interactive user interface, to support automation required for a large scale mining study. Furthermore, the tool allows customization of quality analysis parameters, such as metric thresholds, to serve a wider range of users.

1 Introduction

Code smells [Fow99, SSS14] indicate the presence of quality issues in a software system. A high number of smells in a software system makes the system difficult to maintain and evolve. Therefore, identifying smells in code and refactor them help us improve and maintain the quality of the software.

In the last two decades, many smell detection tools have been developed to aid researchers and practitioners. These tools use various detection techniques such as metrics [Mar05, VMDP14, FM13, SLT06], machine-learning algorithms [KVGS09, MKMD16, CMC15, MAB+12], historical information [PBDP+15], and rules/heuristics [MGDM10, PDMG14]. For a quantitative analysis, researchers require tools that allow them to detect smells in a large number of repositories automatically. Although some of the available tools enable researchers to mine smells, we perceive the following gaps in the present set of smell detection tools from smell mining perspective.

- The present set of tools detects only a subset of known smells. God class (or blob), shotgun surgery, feature envy, refused bequest, and long method (or god method) smells are the most common smells detected by the present set of tools. Measuring and determining code quality based on only a handful of smells raises a serious threat to validity of the code quality measurements.

- The target programming language of the subject systems is Java for almost all of the academic tools. The lack of tools that support other programming languages makes it difficult to conceive a mining study that targets subject systems belonging to other programming languages.

- Smell detection methods depend on many parameters that influence the outcome of an analysis. These parameters include thresholds used to detect specific smells and the way source code projects are included or excluded for source code analysis. The present set of tools lacks the support to customize the parameters of an analysis.

We present a tool *Designite* [SMT16, Sha17] that we developed to overcome the above-mentioned shortcomings. We explain briefly the functionality and features offered by the tool in the next section.

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017 (sattose.org).
07-09 June 2017, Madrid, Spain.

2 Smell Detection using Designite

In this section, we first briefly present architecture of the tool. We then elaborate on the features of Designite that are useful in the context of smell mining. Further, we list other key features of the tool.

2.1 Architecture of the tool

Figure 1 shows the major components of the tool. Designite uses NRefactory [NRe16] to parse C# code and prepares Abstract Syntax Tree (AST). The source model layer accesses the AST and prepares a simple hierarchical meta-model. The meta-model contains objects of projects containing information about analyzed projects. Each object of an analyzed project contains the objects of namespaces implemented in the project. Similarly, each object of a namespace contains objects of types that are part of the namespace and so on. The meta-model captures the required source code information which is used by the back-end of Designite to infer smells and compute metrics.

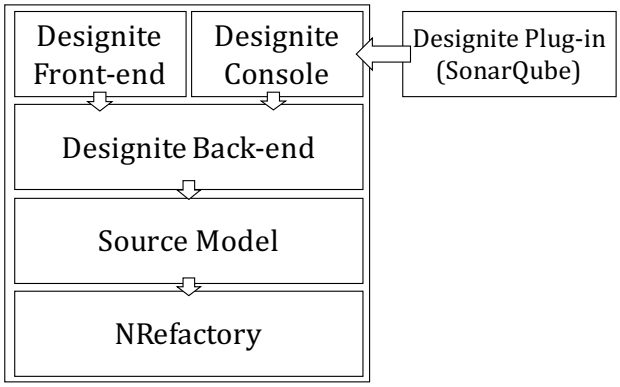

Figure 1: Architecture of the tool

2.2 Smell mining using Designite

Designite is a software design quality assessment tool. The tool supports identification of 30 code smells (19 design smells [SSS14] and 11 implementation smells [Fow99]). Table 1 and table 2 list all the supported design [SSS14] and implementation smells [Fow99] respectively with a brief description.

Designite analyzes source code written in C#. Users can initiate an analysis using a user interface and observe results of the analysis in a visually pleasing yet easy to understand manner. Apart from the user interface, Designite also provides a console application which is particularly useful for analyzing a large number of repositories automatically.

Customization is one of the key features of the tool. A user can customize an analysis within Designite in numerous ways:

- A user can specify the projects to analyze using Designite console application in following ways.
  - A user can initiate source code analysis by specifying the path of a C# solution along with other required parameters.
  - A C# solution may have multiple projects. A user can include or exclude specific projects for the analysis in an input batch file. Excluding specific projects from an analysis is very useful in certain situations; for instance, a user can exclude all test projects (containing "test" in the project name) just by specifying "test" against a specific configuration setting in the input batch file.
  - The tool can also analyze a git repository. Furthermore, a user can provide specific versions (by specifying commit hashes) to be analyzed while performing smell trend analysis. Alternatively, a user can specify a total number of versions to be analyzed from the git history. The tool selects specified number of versions by choosing a version after each m commits starting from the first commit. Here, m is defined by the total number of versions in the repository divided by the required number of versions to be analyzed.
- Similarly, the tool allows users to change the analysis parameters. Specifically, users can choose which of the supported smells they wish to detect using the tool. Figure 2 shows the preference window to customize parameters of an analysis; for the console application, the same effect can be achieved by modifying a configuration file. Additionally, users can change metric thresholds that are used to detect smells.
- The format of the produced output can also be specified. Currently, the tool supports CSV, Microsoft Excel, and XML formats.

2.3 Other features

We list other key features of the tool below:

- The tool detects 30 object-oriented metrics applicable to different source code entities (*i.e.,* project, namespace, class, and method).
- The tool can analyze multiple versions of a software and perform trend analysis of smells. As shown in figure 3, trend analysis reveals how many smells got introduced, refactored, or remained in a version from the previous version.

Table 1: Supported Design Smells with Their Brief Description

Design smell	**Brief description**
Imperative Abstraction	an operation is turned into a class
Unnecessary Abstraction	an abstraction that is actually not needed
Multifaceted Abstraction	an abstraction has more than one responsibility assigned to it
Unutilized Abstraction	an abstraction is left unused
Duplicate Abstraction	two or more abstractions have identical names or identical implementation
Deficient Encapsulation	the declared accessibility of one or more members of an abstraction is more permissive than actually required
Unexploited Encapsulation	client code uses explicit type checks
Broken Modularization	data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions
Insufficient Modularization	an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, or implementation complexity
Hub-like Modularization	an abstraction has high incoming and outgoing dependencies
Cyclically-dependent Modularization	two or more abstractions depend on each other directly or indirectly
Wide Hierarchy	an inheritance hierarchy is "too" wide
Deep Hierarchy	an inheritance hierarchy is "excessively" deep
Multipath Hierarchy	a subtype inherits both directly as well as indirectly from a supertype
Cyclic Hierarchy	a supertype in a hierarchy depends on any of its subtypes
Rebellious Hierarchy	a subtype rejects the methods provided by its supertype(s)
Unfactored Hierarchy	there is unnecessary duplication among types in a hierarchy
Missing Hierarchy	a code segment uses conditional logic to explicitly manage variation in behaviour
Broken Hierarchy	a supertype and its subtype conceptually do not share an "IS-A" relationship

Table 2: Supported Implementation Smells with Their Brief Distribution

Implementation smell	**Brief description**
Long Method	a method is excessively long
Complex Method	a method with high cyclomatic complexity
Long Parameter List	a method has long parameter list
Long Identifier	an identifier with excessive length
Long Statement	an excessive long statement
Complex Conditional	a complex conditional statement
Virtual Method Call from Constructor	a constructor calls a virtual method
Empty Catch Block	a catch block of an exception is empty
Magic Number	an unexplained number is used in an expression
Duplicate Code	a code clone within a method
Missing Default	a switch statement does not contain a default case

- Visualization is one of the vital strengths of the tool. The tool presents detected smells using a sunburst diagram. As shown in figure 4, a sunburst representation not only presents the detected smells in a visually appealing manner but also allows user to navigate and filter the detected smells interactively.

  Additionally, users can visualize the distribution of smells in a software system using treemap. As shown in figure 5, a smell treemap shows the hotspots of the system which could be chosen for refactoring.

  Further, a pie-chart of the metrics could help a user to holistically visualize the state (green, yellow, orange, or red) of the code from a specific metrics point of view.

- The tool can be used to observe and analyze dependencies between types, namespaces, and projects using Dependency Structure Matrix (DSM).

- For refactoring prioritization, the tool computes hotspots (*i.e.,* classes with high number of smells).

- The tool also detects code clones.

A free fully-featured academic license of the tool could be acquired for all academic purposes.

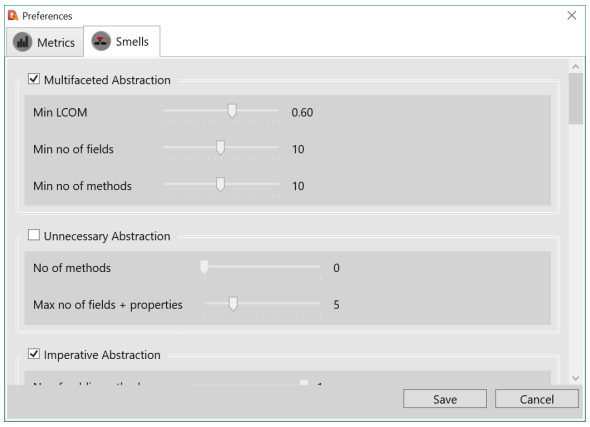

Figure 2: Preferences Window for Customizing a Source Code Analysis

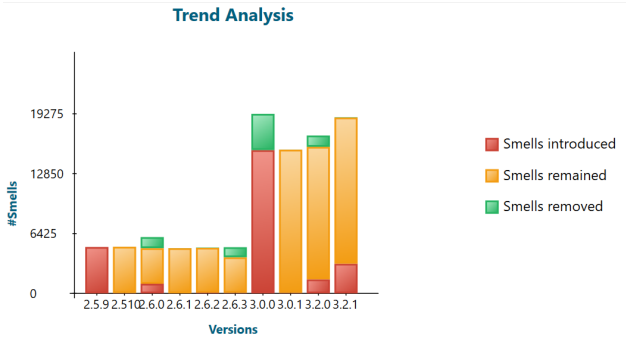

Figure 3: Trend Analysis of Smells

3 Conclusions

In this paper, we revealed a need of a tool for software engineering researchers to perform large scale smell mining studies. It is desired from a smell detection tool to support detection of a wide range of code smells and allow customization of the analysis. We present Designite that detects 19 design smells and 11 implementation smells in C# code. The tool offers mechanisms to customize the analysis based on the context and the requirements. We hope that the research community will exploit the features of the tool for their smell mining studies.

In the future, we would like to add support of architecture smells in the tool. Also, we plan to create an IDE plug-in of the tool for Microsoft Visual Studio.

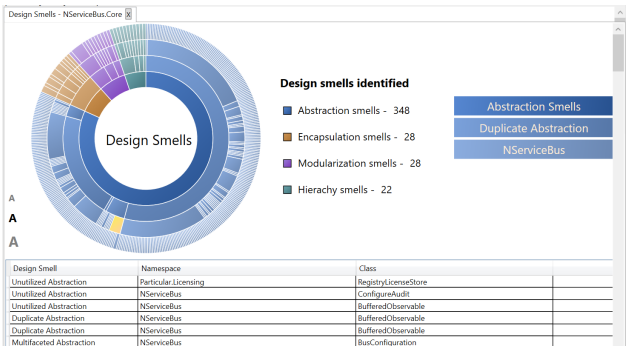

Figure 4: Sunburst Representation of Smells for Effective Navigation and Filtering

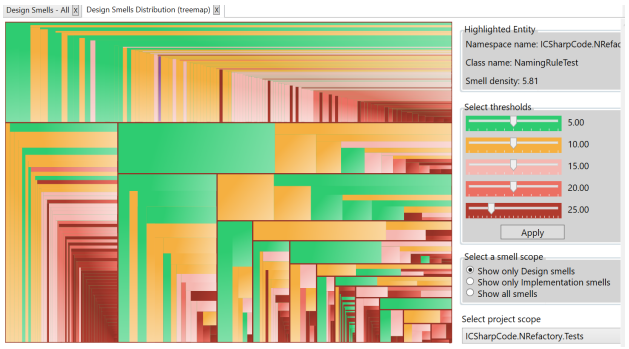

Figure 5: Visualizing Distribution of Smells Using Treemap

References

[CMC15] Gabriela Czibula, Zsuzsanna Marian, and Istvan Gergely Czibula. Detecting software design defects using relational association rule mining. *Knowledge and Information Systems*, 42(3):545–577, March 2015.

[FM13] Amin Milani Fard and Ali Mesbah. JSNOSE: Detecting javascript code smells. In *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013*, pages 116–125. The University of British Columbia, Vancouver, Canada, IEEE, January 2013.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley Professional, 1 edition, 1999.

[KVGS09] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A Bayesian Approach for the Detection of Code and Design Smells. In *QSIC '09: Proceedings of the 2009 Ninth International Conference on Quality Software*, pages 305–314. IEEE Computer Society, August 2009.

[MAB+12] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. Support vector machines for anti-pattern detection. In *ASE 2012: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 278–281. Polytechnic School of Montreal, ACM, September 2012.

[Mar05] R Marinescu. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 701–704. Universitatea Politehnica din Timisoara, Timisoara, Romania, IEEE, December 2005.

[MGDM10] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.*, 36(1):20–36, 2010.

[MKMD16] Usman Mansoor, Marouane Kessentini, Bruce R Maxim, and Kalyanmoy Deb. Multi-objective code-smells detection using good and bad design examples. *Software Quality Journal*, pages 1–24, February 2016.

[NRe16] NRefactory. `https://github.com/icsharpcode/NRefactory`, 2016. [Online; accessed 16-May-2017].

[PBDP+15] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, May 2015.

[PDMG14] Francis Palma, Johann Dubois, Naouel Moha, and Yann-Gaël Guéhéneuc. Detection of REST patterns and antipatterns: A heuristics-based approach. In Xavier Franch, Aditya K Ghose, Grace A Lewis, and Sami Bhiri, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 230–244. Springer Berlin Heidelberg, January 2014.

[Sha17] Tushar Sharma. Designite - A Software Design Quality Assessment Tool. `http://www.designite-tools.com`, 2017. [Online; accessed 16-May-2017].

[SLT06] Mazeiar Salehie, Shimin Li, and Ladan Tahvildari. A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 159–168. University of Waterloo, IEEE Computer Society, June 2006.

[SMT16] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. Designite — A Software Design Quality Assessment Tool. In *Proceedings of the First International Workshop on Bringing Architecture Design Thinking into Developers' Daily Activities*, BRIDGE '16. ACM, 2016.

[SSS14] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt.* Morgan Kaufmann, 1 edition, 2014.

[VMDP14] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23(3):501–532, 2014.