But visual detecting results need manual intervention. Some bad smells relevant to cohesion can be detected using distance theory. Simon et al. [12] defined a distance based metric to measure the cohesion between attributes and methods. The inspiration about the approach in this paper is drawn from the work [12] in the sense that it also employs the Jacquard distance. However, the approach has proposed several new definitions and processes to get improvements. The conception of distance metrics is defined not only among entities (attributes and methods) but also between classes. In [13], the distances between entities and classes are defined to measure the cohesion among them.

There is less research about bad smell detection of Large Class. Liu et al [14] proposed a detection and resolution sequence for different kinds of bad smells to simplify their detection and resolution, including Large Class bad smell. But Liu paid more attention to the schedule of detection rather than Large Class detection itself, and the specific detecting process was not provided in the paper. In Large Class bad smell detection, class size measures have been introduced

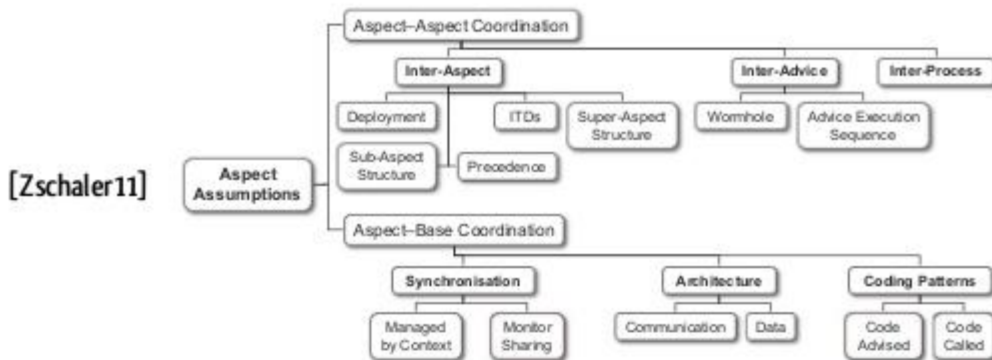
When class size is large, it is seen as Large Class. In bad smell detection tools, the main way [15] of measuring class size is to measure the number of lines of code i.e. NLOC, or the number of attributes and methods. PMD[16] and Check style[17] both use NLOC as detection strategy . The former uses athreshold of 1000 and the second a threshold of 2000. The fixed threshold value is not fastidious for Large Class bad smell detection, and easy to cause false detection. And in these tools, there is no function about refactoring of Large Class bad smell.These researches above show that, the detection of Large Class bad smell is based on fixed threshold comparison. Since the fixed threshold is selected manually, the objectivity is low. Moreover, the refactoring method is decided manually, and there is no suggestion or scheme about that.

How to determine the smells?

The change in the number of smells found usually reflects some significant change in the source code that hinders its degradation. We therefore ask ourselves whether,assuming that the tools may be imprecise, or may have a poor recall, they still can be used by managers to observe, on a broader scale, the evolution of software and assess the general trend of its internal quality. Our experiment will be based on the information on the density ratio of the smells reported by the tools for each version of the project, and on the overall history of the project as deduced from a manual differential analysis of the source code across versions. We will attempt to informally correlate changes, by manual review, in smell density across versions, and the prevalent position of smells in the code, with some basic facts on project development that can be deduced from source code analysis (introduction of new functionalities, refactoring,etc.).

Assumption-based code smells

... aspect/aspect aspect/base assumptions



assumption:

Examples. The Glassbox aspect `JxtaSocketMonitor` assumes to have precedence over its super-aspect `AbstractMonitor`. This is the default semantics of AspectJ. However, an assumption remains that this is not changed by any `declare precedence` clauses anywhere in the code.

tool support:

warn when advice precedence DAG is not completely connected

warn about aspects that override implicit precedence

continuously check assumptions made explicit in code

Figure 1: Assumption of code smells

Licensed Under Creative Commons Attribution CC BY

The quarantine programs are open source programs which contain a large number of classes. In the detection method, the inputs are the codes, and the outputs are the bad smell classes. As the bad smell group location above, the bad smell groups may not be the largest groups. Similarly, the identifying method is not to simply select the x largest classes. So it is the key of Large Class bad smell detection: the detecting basis is not from the metrics of destination class itself (length or others), but from metrics of all the classes. In this paper, [24] the bad smell location in class is identified with the inner cohesion of classes. The cohesion metric is defined with the entity distance theory. In entity distance theory, these concepts should be defined.Enity,Proper set, Cohesion Metric and Distance.

How are we going to present the results?
Code Bad Smells are structures which cause detrimental effects on software. However, little empirical evidence has been provided. Most existing Code Bad Smell detection tools are Metric-based. We argue about their accuracy. Programmers that use detected smells during development or maintenance of a system to improve the code. Code inspectors (or reviewers) that use detected smells to assess the quality of the code.

The classes which are sure to have Large Class bad smell is refactored. And the refactoring process is Extract Class, which means the destination class should be divided into two or more new classes. In practice, the destination class would be divided into two parts, and the bad smell detection would be executed again. The basic idea of refactoring scheme is to divide the entities in the destination class based on the cohesion degree among them. So the key ideas are how to represent cohesion degree between entities in classes and how to cluster entities in classes.

Sometimes you will see a class with four subclasses, each of which only implements three simple methods. Often you will get a vague feeling that the class doesn't deserve subclasses, but you won't immediately be able to see how to eliminate them. This feeling can last for months or even years. Don't worry. If you keep nibbling away at the problems you can see how to solve, eventually you will find yourself looking at the subclasses again, and all the difficult issues to resolve have disappeared. Once you've done this, look for new opportunities to use inheritance now that you are no longer wasting it.

Primitives, which include integers, Strings, doubles, arrays and other low-level language elements, are generic because many people use them. Classes, on the other hand, may be as specific as you need them to be, since you create them for specific purposes. In many cases, classes provide a simpler and more natural way to model things than primitives. In addition, once you create a class, you'll often discover how other code in a system belongs in that class. Fowler and Beck explain how primitive obsession manifests itself when code relies too much on primitives. This typically occurs when you haven't yet seen how a higher-level abstraction can clarify or simplify your code

3. Implementation of Proposed Algorithm

The critical incident technique (CIT) consists of two major phases: data collection and data analysis. The rest of this section describes how we adapted each of these phases for identifying the usability problems of IPT tools for each and every subject Evaluators can collect the critical incidents through surveys, interviews, observing the participants, or asking the participants to report the incidents during the task. These data collection techniques are not scalable to many users, are based on arti_cial tasks, or interfere with users' work. So, we made our data collection automatic to collect a large set of data that covers many usage scenarios of the refactoring tool in a form that is amenable to automatic data analysis. We made the data collection unobtrusive to avoid altering pro-grammars' behavior. Finally, instead of collecting the data from preened tasks performed at the lab, we decided to collect the data from real tasks that are more representative of how the refactoring tool is used in practice.

3.1 Entities Algorithm

Algorithm: Agglomerative Clustering Algorithm
Input: ench entities and their distance
Output： two new clusters
Begin
each entity is assigned to be a single cluster;
While(clustering number is more than 2)
merge two clusters A, B with the lowest distance value as cluster C;
Foreach（any other cluster X in the class）
Dist（C, X）=Avg（Dist（A, X）, Dist（B, X））;
EndFor
EndWhile

A refactoring precondition is a property that the refactoring tool checks at various stages, e.g., selection, invocation, conjuration, and commit, to guarantee that the change will preserve the behavior of the program. If a precondition fails, the refactoring reports a message whose type depends on the severity of the problem and the stage of refactoring. We refer to such a message as a refactoring message or just a message in this paper. The Eclipse refactoring tool may report any of about 640 messages of four types to its user [16]:We made Coding Spectator capture this information be-cause the selection onsets captured by Eclipse do not always reect exactly the ones used by the programmer due to some normalization that Eclipse applies on the selections.

We developed Coding Spectator [12], an unobtrusive tool for collecting the usage data of the Eclipse refactoring tool. The only interaction that the participants had with Coding Spectator was to install it like any Eclipse plug-in, and enter their username and password when prompted to submit their data to our central repository. We chose to make the data collection process unobtrusive to study software evolution practices in the wild Coding Spectator captures more data about the usage of the refactoring tool than what Eclipse already does.

Licensed Under Creative Commons Attribution CC BY

3.3 Class Number Algorithm

Algorithm：Class number statistics
Input： *i* G
Output： *i* P ,
Begin
Foreach （*i* G）
Foreach （*j* =1,2,… *N*)
If （min min [(1) ,] i A Î A + j - ×m A + j ×m）
i P
++;
EndIf
EndFor
EndFor
End

The classes with bad smell should be refactored by Extract Class according to the entities distance and agglomerative clustering algorithm. After refactoring the programs should be test again.

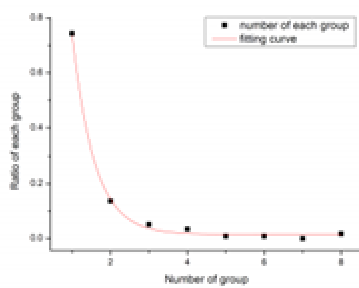


Figure 1: Graph curve for using CIT

The first step is to identify the source page u and destination pages each v € V' where V'€V.

Table 1: Cohesion metrics of group 8 class members of Tyrant0.80 program

Class name	*Number of lines*	*Cohesion metric*
Creature	898	5.763
GameScreen	625	3.125
Map	788	12.061

4. Conclusion

In this paper the approach of Large Class bad smell detection and refactoring scheme has been proposed. Fixed-threshold-based detection method is analyzed to be rigid and error-prone. In this model, the class groups that are far away from the distribution curve are treated as containing bad smells potentially. And combining with cohesion metric computing, the bad smell classes are confirmed in the class groups. After using Agglomerative Clustering Technique, the scheme of Extract Class is proposed for refactoring An alternate refactoring path contains events such as cancellations, repeated invocations, and error messages. We mined alternate refactoring paths in a large, real-world refactoring usage data set and analyzed a subset of it to identify usability problems. As a result, we found 15 usable-ity problems, all of which have been acknowledged by the Eclipse developers and four have already been _xed. This result shows that alternative factoring paths reveal usability problems.

References

[1] M. Fowler, (1999) "Refactoring: Improving the design of existing code", Addison-Wesley, pp89-92.
[2] B.F. Webster, (1995) "Pitfalls of Object Oriented Development", first M&T Books, Feb.
[3] A.J. Riel, (1996) "Object-Oriented Design Heuristics", Addison-Wesley.
[4] G. Travassos, F. Shull, M. Fredericks, & V.R. Basili., (1999) "Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Increase Software Quality", Proceeding of 14th Conference in Object-Oriented Programming, Systems, Languages, and Applications, pp47-56.
[5] R. Marinescu, (2004) "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws", Proceeding of 20th International Conference in Software Maintenance, pp350-359.
[6] Ladan Tahvildari & Kostas Kontogiannis, (2003) "A Metric-Based Approach to Enhance Design Quality through Meta-Pattern Transformations", 7th European Conference Software Maintenance and Reengineering, pp183-192.
[7] M. O'Keeffe & M. O'Cinneide, (2008) "Search-based refactoring: an empirical study", Journal of software maintenance and evolution: research and practice,pp345-364.
[8] K. Dhambri, H. Sahraoui & P. Poulin, (2008) "Visual Detection of Design Anomalies", Proceeding of 12th European Conference in Software Maintenance and Reeng, pp279-283.
[9] G. Langelier, H.A. Sahraoui & P. Poulin, (2005) "Visualization-Based Analysis of Quality for Large-Scale Software Systems", Proceeding of 20th International Conference in Automated Software Engineering , pp214-223.
[10] M. Lanza & R. Marinescu, (2006) "Object-Oriented Metrics in Practice", Springer-Verlag. pp125- 128.
[11] E. van Emden & L. Moonen, (2002) "Java Quality Assurance by Detecting Code Smells", Proceeding of 9th Working Conference in Reverse Engineering, pp120-128
[12] E. M. del Galdo, R. C. Williges, B. H. Williges, and D. R. Wixon. An Evaluation of Critical Incidents for Documentation Design. In Proc. Human Factors and Ergonomics Society, pages 19{23, 1986.}
[13] J.W. Han & M. Kamber, (2005) "Data Mining Concepts and Techniques", Morgan Kaufmann Publishers
[14] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverseengineering approach combining metrics and program visualization. In Proc. 6th Working Conference on Reverse Engineering (WCRE'99), pages 175 186, 1999.
[15] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. efactoring test code. In Proc. 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), pages 92 95, May 2001.
[16] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence

Licensed Under Creative Commons Attribution CC BY

fromchange management data.IEEE Transactions on Software Engineering, 27(1):1 12, 2001.
[17] M. E. Fagan. Design and code inspections to reduce errors in program development. IBM Systems Journal, 15(3):182 211, 1976.
[18] G. Florijn. RevJava Design critiques and architectural conformance checking for Java software. White Paper. SERC, the Netherlands, 2002. See alsohttp://www.serc.nl/people/florijn/work/designchecking/RevJaa.htm.
[19] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.
[20] T. Gilb and D. Graham. Software Inspection. AddisonWesley, 1993..
[21] http://home.engineering.iastate.edu/~hungnv/Personal/papers/PhpSync.pdf
[22] http://www.cs.uiuc.edu/~hanj/pdf/cikm10_tweninger.pdf
[23] International Journal of Software Engineering & Applications (IJSEA), Vol.4, No.5, September 2013

Licensed Under Creative Commons Attribution CC BY