MINING SOURCE CODE FOR DESIGN REGULARITIES

Kim Mens
Département d'Ingénierie Informatique
Univérsite catholique de Louvain
Kim.Mens@uclouvain.be

Andy Kellens
Programming Technology Lab
Vrije Universiteit Brussel
akellens@vub.ac.be

Motivation

The aim of this working session on Industrial Realities of Program Comprehension is to exchange and discuss experiences, opportunities, challenges and strategies for the application of program comprehension techniques in industry. In this position paper we focus on a potentially interesting *opportunity and challenge* for adopting program comprehension techniques, and *source code mining* techniques in particular, in an industrial setting: mining source code for design regularities.

Design regularities are an important aspect of current-day software implementations. Coding conventions, design patterns, programming idioms and architectural constraints are only a few examples of design regularities that govern the implementation of large and complex software systems. Maintaining these regularities in the source code of an evolving software system requires adequate documentation that is continuously monitored and verified for consistency with the source code of that system.

The formalism of *intensional views* [1,2] and their supporting tool-suite IntensiVE[1] [3], for example, permit to document and verify design regularities by means of program queries that group source code entities into views, and that impose constraints over these views. By checking the validity of the views and constraints with respect to the source code, the tool-suite provides fine-grained feedback concerning inconsistencies between the design regularities and the source code. The intensional view approach, however, currently provides *no* support for automatically identifying these regularities (i.e., views and constraints) from the source code of a program. Instead, the approach assumes that the views are created "by need" by the software developers or architects that document or analyze the program. This assumption is unrealistic in an industrial setting where developers have very little time to document their programs adequately, where programs have several tens to hundreds of thousands of lines of code, and where legacy software is the norm rather than the exception.

If we want to apply techniques that support the documentation and verification of design regularities in such a setting, there is a real need for program comprehension techniques that help in identifying the design regularities adhered to by a program.

Context: documenting and verifying design regularities in source code

In a nutshell, intensional views [1,2,3] are a technique for describing a conceptual model of a program's design regularities and verifying consistency of that model with respect to the source code of that program. Views describe concepts of interest to a programmer by grouping program entities (classes, methods, fields, ...) that share some structural property. These sets of program entities are

[1] IntensiVE, which stands for Intensional View Environment, is a pre-commercial set of tools that supports applying the intensional view approach to programs written in programming languages like Smalltalk and Java.

specified intensionally, i.e., by means of an executable description that collects the set of entities satisfying that description. This intension[2] is expressed in the logic meta-programming language SOUL [4], which allows us to write queries that collect program entities in either Smalltalk or Java programs.

For example, in order to model the concept of "all getter methods" in a program, we specify an intensional view using the following query that groups all methods that contain a single statement returning an instance variable (defined in the same class or inherited from one of its ancestors):

```
getterMethod(?class, ?method, ?field) if
    isMethodInClass(?method, ?class),
    fieldInClassChain(?field, ?class),
    fieldName(?field, ?fname),
    methodStatements(?method, ?slist),
    ?slist = <return(variable(?vtype, ?fname))>
```

Without explaining all details of the SOUL syntax and semantics, upon evaluation the above query accumulates all solutions for the logic variables `?class`, `?method` and `?field`, such that the class `?class` implements a `?method` which contains a return statement returning the `?field`. This query is the intension of the view.

In addition to such views, design regularities can be encoded by declaring n-ary constraints over these views
For example:
- a unary constraint that states that a view should not be empty
- or that all of its entities are of a certain kind,
- or a binary constraint that declares that two views are extensionally equivalent (i.e., they provide an alternative way of describing the same set of entities)
- or how all entities in one view are related to those of a second view (e.g., all methods in the first view must call at least one method in the second view).

This combination of logic program queries, intensional views and relations that impose constraints over those views, provides a flexible means of documenting and verifying structural and design regularities in programs.

Challenge: extracting design regularities from source code

The current approach of documenting and verifying design regularities with intensional views still has an important limitation if we want to apply it to industrial-scale software, because it requires the intensional views and relations to be defined by a software developer or architect. Evidently, this is not an easy task, especially not for very large legacy software where documentation often is scarce and out-dated.

To aid a developer or architect in uncovering the design and structural regularities that govern a system, we therefore think there is a need for (semi-)automated techniques that mine the source code of a program for regularities. This process of discovering design regularities is related to aspect mining [5], a novel research direction in the domain of aspect-oriented software development that aims at uncovering crosscutting concerns in existing code bases.

---

[2] The *intension* – with an 's' of a set is its description or defining properties, i.e., what is true about the members of the set. The *extension* of a set is its members or contents.

Such aspect mining techniques are based on the assumption that crosscutting concerns are characterized by a number of symptoms such as the rigorous use of naming and stylistic conventions [6,7], high fan-in values [8] or code duplication [9]. By using source code analysis approaches such as clone detection, or techniques from data mining such as cluster analysis [10] and formal concept analysis [11], these aspect mining techniques mine the source code for groups of source-code entities that all exhibit a similar symptom of crosscutting.

The task of mining for design regularities is based on the same assumptions as aspect mining, namely that certain concepts (crosscutting or not) in the source code of a system are characterized by a recurring pattern in their implementation. We are confident that techniques similar to those for mining aspects can be used to discover structural source-code regularities.

Although some aspect mining techniques are particularly devised to detect instances of crosscutting concerns only, techniques such as the work of Tourwé et al. [7] and He et al. [12] propose a strategy in which the source code is mined for recurring patterns in general. Using extensive post-filtering, the set of results is reduced to those sets of entities that represent a crosscutting concern.

Our intuition that these techniques are suitable for identifying regularities is strengthened by our own findings. In [7] we reported on an experiment in which we applied formal concept analysis [11] to group source-code entities which contained similar substrings. In addition to identifying a number of potential crosscutting concerns, we were also able to identify in this experiment certain implementation idioms, design patterns and domain-specific concepts that were characterized by a similar naming scheme.

In general, the output of such a mining algorithm is a collection of sets of related source-code entities. As such, a developer still needs to transform this extensional set of entities into an intensional view. In other words, a developer needs to specify a program query that describes the source-code entities returned by the mining technique. To automate this process, we suggest using techniques such as inductive logic programming [13]. This machine learning technique takes as input a set of facts and returns a set of logic rules that provide a description for those facts. We have performed some initial experiments [14] using inductive logic programming as a means to induce an intension of a view automatically from a set of source-code entities.

Risks

Evidently, semi-automatically mining the source code of a program for useful design regularities is a non-trivial task with many pitfalls. Just like for aspect mining research we expect, amongst others, the following kinds of problems:

- the code of the program being analyzed may not be well-structured enough to extract relevant regularities;
- it may be difficult to extract the relevant from the irrelevant regularities (poor precision);
- mining techniques may exhibit poor recall (i.e., important regularities may be missed);
- too much user involvement may be required (for example to filter out irrelevant results, to refine the discovered regularities or to guide the search);
- turning sets of program entities that adhere to a similar design regularity into an actual program query may be harder than expected.

Nevertheless, in spite of these expected problems we are hopeful that it is possible to come up with a pragmatic solution that solves or avoids most of these problems and is able to identify, with a

minimum of user-involvement, a set of relevant verifiable design regularities from the source code of a program.

Conclusion

We have a background both in documenting and verifying structural and design regularities in the source code of large software systems, and in the emerging field of aspect mining. Whereas a variety of tools exist to document and check design regularities at various levels of details, most approaches either rely on a fixed set of predefined rules or assume that the software designers and architects themselves will codify the rules. Such an approach does not scale-up to industrial-scale legacy software. We therefore think there is a need and opportunity for program comprehension techniques, and source code mining techniques in particular, for suggesting potential rules or at least sets of software artifacts that exhibit a similar structural regularity. Inspiration may be taken from the field of aspect mining research, even though there are still many pitfalls to be overcome.

Acknowledgements

Andy Kellens is funded by a scholarship provided by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen). This research is partially funded by the Interuniversity Attraction Poles Programme - Belgian State Belgian Science Policy.

References

[1] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views: A case study. Elsevier Journal on Computer Languages, Systems & Structures, 32(2-3):140–156, 2006.
[2] A. Kellens. Maintaining causality between design regularities and source code. PhD thesis, Vrije Universiteit Brussel, June 2007.
[3] http://www.intensional.be
[4] http://prog.vub.ac.be/SOUL
[5] A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. Transactions on Aspect-oriented Development (TAOSD), 2007.
[6] D. Shepherd, T. Tourwé, and L. Pollock. Using language clues to discover crosscutting concerns. In Workshop on the Modeling and Analysis of Concerns, 2005.
[7] K. Mens and T. Tourwe. Delving source-code with formal concept analysis. Elsevier Journal on Computer Languages, Systems & Structures, 31(3-4):183–197, 2005.
[8] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In Working Conference on Reverse Engineering (WCRE), pages 132–141. IEEE Computer Society, 2004.
[9] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. On the use of clone detection for identifying crosscutting concern code. IEEE Transactions on Software Engineering, 31(10):804–818, 2005.
[10] A. Jain and R. Dubes. Algorithms for Clustering Data. Prentice Hall, 1988.
[11] B. Ganter and R. Wille. Formal Concept Analysis: Mathematical Foundations. Springer-Verlag, 1999.
[12] L. He, H. Bai, J. Zhang, and C. Hu. Amuca algorithm for aspect mining. In Software Engineering and Knowledge Engineering (SEKE), 2005.
[13] Bergadano and D. Gunetti. Inductive Logic Programming: From machine learning to software engineering. MIT Press, 1995.
[14] T. Tourwé, J. Brichau, A. Kellens, and K. Gybels. Induced intentional software views. Elsevier Journal on Computer Languages, Systems & Structures, 30(1-2):35–47, 2004.