Reverse Engineering Object-Oriented Applications Into High-Level Domain Models With Reoom

Tuan Anh Nguyen, Christoph Csallner
Computer Science and Engineering Department
The University of Texas at Arlington
Arlington, TX 76019, USA
Email: tanguyen@mavs.uta.edu, csallner@uta.edu

***Abstract*—Automatically pinpointing those classes in an object-oriented program that implement interesting domain concepts would be valuable for industrial software maintainers. We encode two observations of programmer behavior in Reoom, a novel light-weight static analysis. In a comparison with its most closely related competitor, Womble, on third-party open source applications, Reoom scaled to larger applications and achieved better overall precision and recall.**

I. Introduction

Having an up-to-date design document such as a UML class diagram of an application's domain model is rare in practice. Engineers often do not like documenting their designs [1] and do not keep documents in sync with the code [2].

But during maintenance, engineers want to distinguish *domain model classes* (which contain important business data) from the remaining classes (the *plumbing classes*). Manually pinpointing these domain model classes is notoriously hard and time-intensive. But having such up-to-date designs could make maintainers more productive [3]–[5].

Pinpointing domain model classes is hard, as it may require reasoning about a complex code base to extract high-level information that is not obvious from the code. Further, each analyzed program implements its own unique business logic in its own ways (with varying naming and coding conventions, domain terms, design pattern mix, etc.).

Parts of this task have been automated [6]–[9], the most closely related approach is Womble [6]. But these techniques ignore the distinction between core and plumbing classes [7], [8] or require seeding with a true domain model class [6].

This paper describes two observations of how developers likely express domain model properties in code, encode these observations in Reoom, and compare Reoom with Womble on third-party code. In these experiments, Reoom scaled to larger applications and achieved better overall precision and recall.

II. Observations (O), Assumptions, And Design

Reoom builds on two observations or heuristics, which share the idea that a programmer tends to pick an implementation style that documents the important domain concepts in the code. First, **if an intermediate result is a domain model object then the program is more likely to refer to it explicitly (O1)**, by assigning it to a local variable, instance field, or method parameter. Such an explicit reference may

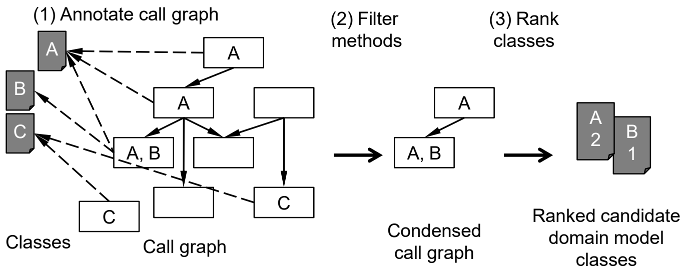

Fig. 1: Reoom's main processing steps, illustrated on a toy program, whose methods explicitly refer to three classes.

aid debugging. Another possible motivation is that a domain model object may be seen as more stable and therefore better suited for an explicit reference.

Second, **a domain model class is likely used together with other domain model classes (O2)**, to navigate and update the domain relations the class participates in and to provide business functions. In contrast, a plumbing class tends to be more specialized and focused.

Reoom looks for these observations in the call graph (i.e., its call chains[1]) and the graph that maps each method to the classes the method refers to explicitly as the static type of local variables, method parameters, and instance fields. In short, the more often a class A is referenced explicitly with another class in a call chain, the more likely A is a domain model class.

Figure 1 shows Reoom's main processing steps. Step (1) checks observation O1, annotating each method with the classes the method text refers to explicitly. Figure 1 shows a simple example call graph with 8 methods that explicitly refer to classes A–C.

Step (2) checks observation O2. Specifically, Reoom removes a method m from the call graph if m does not appear in any call chain that explicitly refers to at least two different classes. Reoom further removes methods that are not connected via the call graph, as they likely represent low-level utility methods. Reoom also removes methods that are not called, only have a single successor, and reference

[1]Recall that a call chain is a (partial) path through an application's static call graph [10]. If a call chain is feasible it can be thought of as a snapshot of the top N invocation frames of the application's runtime call stack.

Subject				Womble							Reoom Light							Reoom		
					∩		∪		∅			Locals		Params		Return				
	kLOC	c	d	t[s]	p	r	p	r	p	r	t[s]	p	r	p	r	p	r	t[s]	p	r
jMusic 1.6.2	23.2	272	5	timeout							29	7	100	16	100	25	100	397	19	100
pdf-sam 2.2.1	8.8	156	9	270	100	11	16	100	45	14	15	13	67	23	56	17	56	158	38	89
pizza_wo	1.1	18	4	1	50	25	57	100	52	31	6	36	100	100	75	40	50	30	100	100
SH	28.3	161	27	timeout							28	20	85	28	85	30	74	662	37	56

TABLE I: Reoom vs. Womble: Higher precision (p) and recall (r) values are better; SH = SweetHome3D 1.5; c = classes and interfaces; d = domain model classes in c; t = runtime; ∅ = average precision and recall of Womble's seeded runs.

explicitly the same classes as their successor, since such a pattern is a simple delegation that does not add interesting domain information. In the example, Step (2) removes six methods from the call graph.

Reoom further removes a class from the call graph if it does not have at least one field with corresponding methods that read and update this field. This step checks for the common pattern that domain model information is both read and updated during execution.

In Step (3) Reoom extracts the classes that annotate the remaining call graph methods as domain model classes. Reoom ranks these classes by how often they are referenced explicitly in the call graph. In our example, classes A and B are most likely domain model classes, in this order.

III. Evaluation

The Reoom prototype is built on the static inter-procedural Java analysis framework MoDisco [11]. Using MoDisco, Reoom includes in its call graph all explicit method and constructor calls that occur in the analyzed source code of public methods and constructors.

Reoom over-approximates calls to virtual (non-private non-static) methods by considering all overriding non-abstract methods. Not captured are calls made via Java reflection or calls from bytecode or native code.

Most closely related to Reoom, Womble [6] also performs a lightweight static analysis to infer from Java code a high-level domain model. Both Reoom and Womble aim to suppress from the inferred model certain plumbing classes. Womble focuses on suppressing container classes. Womble's input is the bytecode of at least one "root" class, which Womble uses as a starting point to find additional domain model classes.

We ask the following three **research questions**. How do Reoom and Womble compare in **runtime performance (RQ1)** and **precision and recall (RQ2)**? What is the relative **benefit of the relatively expensive second step (RQ3)**, which requires inter-procedural analysis?

Table I summarizes the results on four third-party Java open-source subject applications, which range from 1.1 to 28.3 kLOC[2]. All experiments ran on a 16 GB RAM 2.6 GHz Core i7 MacBook Pro running OS X 10.10.2. For pdf-sam we relied on our domain knowledge to identify the domain model package. The other subjects had detailed design documentation and we took care to analyze the software version that matched the documentation most closely.

Pizza_wo is a plain Java version of the well-documented Pizza Shop tutorial. The jMusic documentation says "[t]he jm.music.data package [is] the only one that you really need to know."[3] While this package contains nine classes, the documentation describes five of them as "[t]hese classes form the backbone of the jMusic data structure".

SweetHome3D's documentation contains a UML class diagram with 27 of the 35 classes in the model package. The developers describe this as: "This UML diagram should help you understand which classes are available in the Sweet Home 3D model".[4]

The runtime listed for Womble is the sum of d runs. As Womble requires at least one seed class, we ran Womble on each real domain model class (handing Womble one "free" correct result). We then computed precision and recall for the intersection (∩) and union (∪) of all classes Womble identified. To compare Womble with Reoom, we average the precision and recall Womble achieved over d runs (assuming the user correctly guessed one domain model class for the seed).

In the **experimental results** (Table I), Reoom's runtime was one order of magnitude higher than Womble's. But Womble timed out (24h) for the two larger applications.

Overall Reoom achieved better precision and recall than Womble. For example, while Womble timed out for SweetHome3D, Reoom selected from 161 classes 41 as candidate domain model classes. The 10 classes Reoom ranked highest are all included in the 27 classes listed by the developer's domain model documentation.

To compare the impact of its two main steps, we also ran Reoom without step (2). To explore the impact of the three kinds of referenced types on step (1), we ran the first step three times, each time annotating the call graph with only one kind of directly referenced types. Table I shows these three runs' combined runtime. The results indicate that *Reoom Light* (i.e., Reoom without step (2)) provides a good trade-off between Womble and full Reoom. Further, locals and parameters provided better recall than return types.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grants No. 1117369 and 1527398.

[2]As counted by JavaNCSS (https://github.com/codehaus/javancss)

[3]http://explodingart.com/jmusic/jmtutorial/x41.html

[4]Figure 13 on http://www.sweethome3d.com/pluginDeveloperGuide.jsp

References

[1] R. Farenhorst, J. F. Hoorn, P. Lago, and H. van Vliet, "The lonesome architect," in *Proc. Joint Working IEEE/IFIP Conference on Software Architecture 2009 and European Conference on Software Architecture (WICSA/ECSA)*. IEEE, Sep. 2009, pp. 61–70.

[2] A. Forward and T. C. Lethbridge, "Problems and opportunities for model-centric versus code-centric software development: A survey of software professionals," in *Proc. International Workshop on Models in Software Engineering (MiSE)*. ACM, May 2008, pp. 27–32.

[3] B. Anda and K. Hansen, "A case study on the application of UML in legacy development," in *Proc. 5th ACM/IEEE International Symposium on Empirical Software Engineering (ISESE)*. ACM, Sep. 2006, pp. 124–133.

[4] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche, "The impact of UML documentation on software maintenance: An experimental evaluation," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 6, pp. 365–381, Jun. 2006.

[5] W. J. Dzidek, E. Arisholm, and L. C. Briand, "A realistic empirical evaluation of the costs and benefits of UML in software maintenance," *IEEE Transactions on Software Engineering (TSE)*, vol. 34, no. 3, pp. 407–432, May 2008.

[6] D. Jackson and A. Waingold, "Lightweight extraction of object models from bytecode," in *Proc. 21st ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, May 1999, pp. 194–202.

[7] M. Keschenau, "Reverse engineering of UML specifications from Java programs," in *Companion to the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Oct. 2004, pp. 326–327.

[8] A. Sutton and J. I. Maletic, "Recovering UML class models from C++: A detailed explanation," *Information and Software Technology*, vol. 49, no. 3, pp. 212–229, Mar. 2007.

[9] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering (TSE)*, vol. 35, no. 4, pp. 573–591, Jul. 2009.

[10] A. Rountev, S. Kagan, and M. Gibas, "Static and dynamic analysis of call chains in Java," in *Proc. ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, Jul. 2004, pp. 1–11.

[11] H. Brunelière, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: A generic and extensible framework for model driven reverse engineering," in *Proc. 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, Sep. 2010, pp. 173–174.