Fraunhofer Institut
Experimentelles
Software Engineering

The Accessor Classification Approach to Detect Abstract Data Types

Authors:
Jean-François Girard
Martin Würthner

IESE-Report No. 073.98/E
Version 1.0
December 21, 1998

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competetive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach
Sauerwiesen 6
D-67661 Kaiserslautern

The Accessor Classification Approach to Detect Abstract Data Types

Jean-François Girard and Martin Würthner

Fraunhofer Institute for
Experimental Software Engineering
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
{girard,wuerthne}@iese.fhg.de

Abstract

*This report presents a new approach to identify abstract data types (**ADT**) in source code. For each type defined in a system, this approach assigns a role to functions related to this type. Then, using various heuristics, it associates these functions with types to form ADTs.*

A prototype tool has been implemented to support this approach. It has been applied to two C systems (31 and 38 Kloc respectively). The ADTs identified by the approach are compared to those identified by software engineers who did not know the proposed approach. In a case study, this approach has been shown to identify, in most cases, more ADTs than four published techniques applied on the same systems. This new approach produces a small number of false positives.

1. Introduction

The Bauhaus project[1] aims at recovering the software architecture of a system "as it is implemented". That is, recovering multiple views which would indicate the main **components** of the system, their **connectors** (how they communicate), and the **constraints** on these connectors and components.

This report presents an approach which can be used as a first step in components identification. A case study [Gira97a] suggests that abstract data types are instances of the smallest components which are significant at the architectural level. The authors have named such building blocks **atomic components**, because they can be used to build larger components.

1. Research collaboration between the Fraunhofer IESE and the University of Stuttgart.

According to Sommerville [Somm92] an **abstract data type** (**ADT**) is an abstraction of a type which encapsulates all the type's valid operations and hides the details of the implementation of those operations by providing access to the types exclusively through a well-defined set of operations.

These atomic components are important to the authors' research on architecture recovery, but they are also of general importance. Their role has been recognized to provide information hiding and thus to support maintainability [Ghez91] and reuse [Somm92].

However, these atomic components are not always explicitly captured in the source code, because programming languages like C do not provide good support to express them or software development does not always exploit them. Consequently, reverse engineering techniques are required to identify them in these circumstances.

This report presents a new reverse engineering approach to identify abstract data types (**ADT**) in source code. This approach, called **accessor classification**, assigns a role (client, constructor, member, violator) to each function related to a type. When a function related to more than one type is encountered, the approach either assigns the function to the type it is mostly related to (according to its role) or groups the types to form a single ADT.

In order to evaluate the proposed accessor classification approach, it was applied to two C systems (31 and 38 Kloc respectively) and the results were compared to the atomic components identified by software engineers. four other published atomic component identification techniques were applied to these systems. On this benchmark, the accessor classifcation approach identifies, in most cases, more correct ADTs than the other techniques.

Paper Overview

The remainder of the report is organized as follows: Section 2 presents related research. Section 3 presents the approach. Section 4 describes the experiment setup used to evaluate atomic components identified by automatic techniques. Section 5 discusses results. Section 6 concludes and proposes further research.

2. Related Research

This section presents a brief overview of the related research on ADT identification. There is a rich literature on the detection of ADTs from systems written in procedural languages (usually C). Many techniques [Ogan94, Gira97a, Yeh95, Canf93a, Canf94] that aim at ADT recovery focus on the relations between types and the parameters and the return type of a function (called function signature). These relations are usually represented by edges in a graph where the nodes correspond to functions and user-defined data types of a system. The basic idea of these techniques is that the set of connected components of this graph form the set of ADT candidates. However, these candidates are often too large, because of functions which examine or modify structures of many types. As a result, these ADT candidates often combine and thereby hide multiple correct reference ADTs.

Each of the following techniques proposes a different heuristic to avoid these large candidates by imposing additional conditions on the connected components:

- *Same Module heuristic* [Gira97a]: It breaks these large candidates at the module boundary. Thus groups only those routines, global variables, and types together that are declared in the same module (i.e. file_name.c & file_name.h).
- *Part Type heuristic* [Ogan94]. It filters out types in a parameter list that are part of another type in the list.
- *Internal Access heuristic* [Yeh95]: It filters out routines which do not access to the fields of record types that are part of an ADT.
- *Enumeration & Dominance heuristic* [Canf94]: It filters out enumerations and subranges from the types considered to form an ADT when the ADT is too large. It also uses the dominance relation on the call graph to identify functions which are called only from an ADT in order to package them with the ADT.

Another approach, *Similarity Clustering* [Gira99], does not rely on these connected components, but exploits similarity of context. It groups entities (functions and user-defined types) according to the proportion of features (entities they access, their name, the file where they are defined, etc.) they have in common. The intuition is that if these features reflect the correct direct and indirect relationships between these entities, then entities which have the most similar relationships should belong to the same ADT. In a case study [Gira99] comparing it with the other approaches, it identified more ADTs than any of the other approaches. However, this result comes with a high number of false positives.

In practice, in languages like C, atomic components are seldom captured explicitly and software development does not always exploit them. As a result, their encapsulation is often violated by direct accesses which bypass the accessor functions of the atomic components. None of these approaches attempt to identify these violators and to exclude them from their ADTs. This often leads to ADT candidates which are larger than what a software engineer would describe as an ADT.

3. Accessor Classification Approach

In contrast, the accessor classification approach, identifies violators and clients of ADTs, and excludes them from the ADT candidates it produces. This approach also yields a small number of false positives due to an early filtering of invalid ADTs. This translates in shorter review time for human analysis or better inputs for tools employing ADTs as an intermediate step toward other abstractions.

3.1. Outline

The accessor classification approach proceeds according to the steps depicted in Figure 1.

1. It classifies each function with respect to every type to which it is related, according to the role this function plays with respect to this type.
2. It rejects types which do not lead to a valid ADT. That is, if the type does not have at least one constructor, two members and one client.

   The desired effect is to reduce the number of functions that are related to more than one type.
3. Hide functions that are violators or clients from their respective type
4. For each function *F* related to type *X* &*Y*

   apply multi-type heuristics to either
   - assigns the function to the type it is mostly related to
   - or group type *X* and *Y* to form a single ADT.

Figure 1. Approach's outline

3.2. Roles of Functions

For each type, the functions related to this type are classified according to their role with respect to the ADT containing the type. The classification uses the following rules:

- A function F is a **constructor** of the ADT containing type T if
  - F has T as a return type
  - F accesses at least one field of T
  - F has no parameter of type T
- F is an **member** of the ADT containing type T if
  - F accesses at least one field of T
  - F has a parameter of type T, *T or &T
- F is a **client** of the ADT containing type T if
  - F does not access any field of T
  - F calls at least one routine R with parameter of type T, *T or &T and R accesses at least one field of T
- F is a **violator** of the ADT containing type T if
  - F accesses at least one field of T
  - F does not have a parameter of type T, *T or &T

3.3. Multi-type Heuristics

The approach can be extended by adding heuristics to deal with functions that are related to more than one type. These heuristics decide if such a function should be assigned exclusively to one type or if the types should be grouped to form a single ADT. The following two heuristics have been used to produce the reported results. Given a function F, which is related to types X and Y:

- F is constructor(X) vs. member(Y)

  if X is the type of a field of Y then
  remove F as the constructor of X
  else
  group X and Y in the same ADT
- F is member(X) and member(Y)

  group X and Y in the same ADT

4. Experiment Setup

In order to evaluate our accessor classification approach, it was applied to two medium-size programs, and the results were then compared to the atomic components identified by software engineers. The components identified by the approach are called the *candidate components*. The atomic components identified by software engineers will be called *reference components*. This section summarizes the experimental setup and the analysis method used.

4.1. Systems Studied

The analyses described above were applied to two medium size C programs (see Table 1 for their characteristics). Aero is an X window system-based simulator for rigid body systems [Kell95] and bash is a Unix shell.

Table 1. Systems Studied

System Name	Version	Lines of Code	# User Defined Types	# Global Variables	# User Defined Routines
aero	1.7	31 Kloc	57	480	488
bash	1.14.4	38 Kloc	60	487	1002

4.2. Human Analysts

Four software engineers were given the task of identifying atomic components in each system. These systems were unknown to them. There was no overlap of their work. They needed about 20 hours for each system to gather the atomic components of the respective systems.

Table 2. Human Analysts.

software engineer	Programming Experience	System Analyzed
se1	2 years research	bash
se2	2 years research	bash
se3	5 years research	bash
se4	> 5 years industry; 1.5 years research	aero

The software engineers were provided with the source code from each system, a summary of connections between global variables, types, and functions, and guidelines defining what is an ADT (and other type of atomic components they should look for), mentionning that programmers break encapsulations and that they should follow their understanding of the code rather than structural rules.

Table 3 shows the numbers abstract data types that were identified by the group of software engineers for each studied system.

Table 3. Reference Atomic Components.

System	#ADT
Aero	10
Bash	22

The variation in experience and the number of people working on each system prevent comparison of different techniques across systems. However, a study [Gira99],

where these software engineers analyzed the same subsystems sugests that the agreement among the ADT identified is suffient to be used as a reference point. Furthermore, the fact that none of the software engineers knew the automatic analyses to be applied to the systems, prevented a bias toward a specific technique.

4.3. Comparison of Candidate and Reference Components

This subsection explains how imperfect matches between candidate and reference components are compared and classified.

Candidate components Cs and reference components Rs are compared using an approximate matching to accommodate the fact that the distribution of functions, global variables, and types into atomic components is sometimes subjective. We treat one component S as part of another component T (denoted by S << T) if at least 70 percent of the elements of S are also in T.

Based on this approximation, the generated candidates are classified into 3 categories according to their usefulness to a software engineer looking for atomic components:

- **Good** when the match between a candidate C and a reference R is close (i.e., C << R and R << C).

  Matches of this type require a quick verification in order to identify the few elements which should be removed or added to the atomic component.
- **Ok** when the relationship holds only in one direction for candidates C_i and references R_i:
  - C_i << R, but not R << C_i (i > 0)
  - R_i << C, but not C <<R_i (i > 0)

  Partial matches of this type require more attention to split, combine, or refine a component.
- **Bad** candidate components are not close enough to the reference components to guide the software engineer's work.

4.4. Accuracy

In order to indicate the quality of imperfect matches of candidate and reference components, an accuracy factor has been associated with each match. The accuracy between a candidate C and a reference R is computed by the following formula:

$$accuracy(C, R) = \frac{|C \cap R|}{|C \cup R|}$$

For matches between more than two components (n~1 and 1~n) the union of all elements of the n components is used to compute the accuracy. The accuracy is not defined for n~m matches, because the m references are not always unique.

5. Results

This section compares the atomic components recovered by similarity clustering with those recovered by other techniques. Then it discusses the impact of false positives.

Benchmark Results.

The following techniques to recover atomic components were applied on the two systems described :

- naive connected components
- same module [Gira97a]
- part type [Ogan94]
- internal access [Yeh95]
- accessor classification

The number of recovered ADTs and their accuracy are reported in Table 4.

Table 4. Detected ADT.

Method	System	ADT			
		Good		OK	
		#	acc.	#	acc
Naive Connected components	Aero	1	1	1	0.31
	Bash	7	1	4	0.41
Same Module	Aero	1	0.91	2	0.30
	Bash	2	0.86	3	0.43
Part Type	Aero	1	1	2	0.44
	Bash	7	1	4	0.44
Internal Access	Aero	2	0.98	2	0.47
	Bash	10	0.93	4	0.40
Accessor classification	Aero	3	0.93	3	0.41
	Bash	15	0.88	2	0.25

The number of atomic components and their accuracy are not the only aspects to consider, one has to look a the number of false positives (bad category) and the number of reference components which were not recognized by a technique are also important. Both of them are depicted in Table 5, where one can note that the number of false positives and missed reference components are similar for each approach with the exception of same module with bash and internal access for aero.

Table 5: False Positives and Missed References

Technique	aero		bash	
	F.P	M.	F.P.	M.
Naive Connected Components	4	1	6	2
Same Module	5	7	5	15
Part Type	4	1	7	2
Internal Access	1	1	4	4
Accessor Classification	4	3	4	5

6. Conclusions

In this report, we presented a new approach to extract abstract data types from source code. This approach, called accessor classification, assigns a role (client, constructor, accessor, member, violator) to each function related to a type. When a function related to more than one type is encountered, the approach either assigns the function to the type it is mostly related to (according to its role) or groups the two types to form a single ADT.

We compared the components identified by a group of software engineers on two C systems (31 and 38 Kloc respectively) to the components identified by similarity clustering and four other published approaches. Our accessor classification approach identifies more ADTs than the other approaches. Still it performs this task producing a low number of false positive and missed references similar to the other approaches.

Future work

This experiment will be extended to a larger number of systems to obtain more significant results and to be able to evaluate if these conclusions can be generalized.

New heuristics should be developed to decide if the functions related to many types should be assigned exclusively to one type or if the types should be grouped to form a single ADT. Similarly more precise filtering heuristics should be constructed.

Notes

The results for the internal access approach differ from those previously published, because the approach has been modified to focus only on fields of records and unions.

Acknowledgment

We would like to express special thanks to Joachim Bayer, Hiltrud Betz, Stephan Kurpjuweit, Minna Mäkäräinen, and Klaus Schmid for their manual analysis of the subject systems used in our comparison.

References

[Bigg89] T. J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22:36–49, July 1989.

[Canf93a] G. Canfora, A. Cimitile, and M. Munro. A reverse engineering method for identifying reusable abstract data type. In *Working Conference on Reverse Engineering*, pages 73–82. May 1993.

[Canf93b] G. Canfora, A. Cimitile, M. Munro, and C. J. Taylor. Extracting abstract data type from C programs: A case study. In *International Conference on Software Maintenance*, pages 200–9. September 1993.

[Canf94] G. Canfora, A. Cimitile, M.Tortorella, and M. Munro. A precise method for identifying reusable abstract data types in code. In *International Conference on Software Maintenance*, pages 404–413. September 1994.

[Ghez91] C. Ghezzi, M. Jazayeri, and D. Madrioli. *Fundamental Software Engineering*. Prentice Hall International, 1991.

[Gira97a] J.F Girard and R. Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *International Conference on Software Maintenance*, 1997.

[Gira97b] J.F Girard, R. Koschke, and G. Schied. Comparison of abstract data type and abstract state encapsulation detection techniques for architectural understanding. In *Fourth Working Conference on Reverse Engineering*, October 1997.

[Gira99] J.-F. Girard, R. Koschke, and G. Schied. A Metric-based Approach to Detect Abstract Data Types and State Encapsulations. To be published in *Journal of Automated Software Engineering*, vol 6, 1999.

[Kell95] H. Keller, H. Stolz, A. Ziegler, and T. Bräunl. Virtual mechanics simulation and animation of rigid body systems with aero. *Simulation for Understanding*, 65(1):74–79, July 1995.

[Ogan94] R.M. Ogando, S.S. Yau, and N. Wilde. An object finder for program structure understanding in software maintenance. *Journal of Software Maintenance*, 6(5):261–83, September-October 1994.

[Rich92] Richter. Classification and learning of similarity measures. In *Annual Conference of the German Society for Classification*, number 16th. Springer Verlag, 1992.

[Schw91] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *International Conference on Software Engineering*, pages 83–92, May 1991.

[Shan72] C. E. Shannon. *The mathematical theory of communication*. Urbana: Univ. of Ill. Press, 1972. ISBN 0-252-72548-4.

[Somm92] I. Sommerville. *Software Engineering*. Addison Wesley, fourth edition, 1992.

[Yeh95] A.S. Yeh, D.Harris, and H. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. In *Second Working Conference on Reverse Engineering*, pages 227–236, Los Alamitos, California, July 1995.

Document Information

Title:	The Accessor Classification Approach to Detect Abstract Data Types
Date:	December 21, 1998
Report:	IESE-073.98/E
Status:	Final
Distribution:	Public