Encapsulation in Object-Oriented Programs

Jiun-Liang Chen Feng-Jian Wang

Institute of Computer Science and Information Engineering
National Chiao Tung University
HsinChu, 30050, Taiwan, R.O.C.
TEL:+886-35-712121~54718
FAX:+886-35-724176
e-mail:{jlchen, fjwang}@csie.nctu.edu.tw

1. Issue of Encapsulation

In stead of decomposing data and procedures, object-oriented (OO) programming encapsulates the both through an object. An object is instantiated from a class which defines attributes (related data) and methods (operational procedures). The definition of a class may be recursive since the class can encapsulate the instance(s) of another classes as its attributes. An object containing other objects is called a *complex* object. The form of a most primitive object, an object containing no other object, is nothing but a data entity and is called a *simple* object. Encapsulation solidates an object by hiding the details; it may also obstruct dependency analysis for complex objects by information hiding.

Program Dependency Graph (PDG) [OtOt84] represents the structure of a program. It is a set of graphic representations, vertices for entities in a program and edges for dependencies between the entities. PDG has been used in various dependency analysis techniques, such as program slicing [HoRe+91] and program optimization [FeOt+87]. Recently, many researches (such as [MaMc+94, LaHa95, ZhCh+96]) extended PDG, by augmenting class and object relationships (e.g., inheritance, membership, ..., etc.) in a PDG, to represent OO programs for dependency analysis. In extended PDG's, a vertex denotes an object while an edge denotes a dependent relation. However, a complex object can not guarantee the cohesion of and coupling among its components to be strong. Extended PDG's have some defects for dependency analysis, especially to represent a complex object as a vertex. For example, the extended PDG for objects *A*, *B*, and *C* with dependencies between them is shown in Figure 1. In the graph, the dependencies, "*B* depends on *A*" and "*C* depends on *B*," imply "*C* depends on *A*." On the other hand, *B* itself is a complex object containing objects b_1 and b_2. The dependencies defined in more details are "b_1 depends on *A*," and "*C* depends on b_2," but no dependency between b_1 and b_2. Consequently, *C* does not depends on *A* with transitive property of *B*'s dependency; This conflicts with the previous result, "*C* depends on *A*." In other words, a transitive relation can not be directly applied to such extended PDG's.

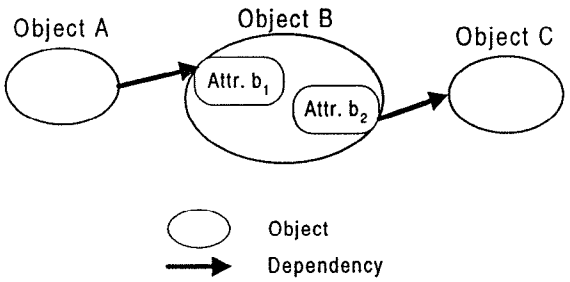

Figure 1. The dependency between objects

2. Dependency between Objects

To investigate the dependency analysis obstructed by encapsulation, we take a data-dependent relation into discussion. An object is said to be *data-dependent* on another if the change of former's value affects the latter's. Formally, a data dependency between two arbitrary objects can be defined as following:

Definition 1 Let O_s be a set of simple objects, and O_c be a set of complex objects. Given two objects, α and β, α is said to be *data-dependent* on β, denoted as $\alpha \Re \beta$, if one of the following cases holds:
(1) $\alpha, \beta \in O_s$, the change of β's value might make α's change,
(2) $\alpha \in O_s$, $\beta \in O_c$, $\exists\, b$, b is an attribute of β, such that $\alpha \Re b$,
(3) $\alpha \in O_c$, $\beta \in O_s$, $\exists\, a$, a is an attribute of α, such that $a \Re \beta$, or
(4) $\alpha, \beta \in O_c$, $\exists\, a$, a is an attribute of α, and $\exists\, b$, b is an attribute of β, such that $a \Re b$.

This definition is recursive and terminates at case (1). In this definition, the first case states that a simple object is data-dependent on anther. The rest cases state that there exists a component object in a complex object to form the data dependency. Besides, the last case indicates that when a complex object depends on another, only part of component objects of the former, not the whole, depends on that of the latter.

Definition 1 describes a direct data dependency between two objects. However, data dependency may be caused not only directly but also indirectly. Indirect dependencies may be computed according to transitive property of dependencies. According to Definition 1, the transitive property of data dependencies is inherent since an object may be data dependent on the portion of another. Therefore, applying transitive property to compute indirect dependency has to consider the components of a complex object on which another depends. The following property can be applied to compute an indirect data dependency.

Property 1 Given arbitrary three objects α, β, and γ, then $\alpha \Re \beta$ and $\beta \Re \gamma$ imply $\alpha \Re \gamma$, denoted as $\alpha \Re \beta \wedge \beta \Re \gamma \rightarrow \alpha \Re \gamma$, when one of following cases holds:
(1) $\beta \in O_s$,
(2) $\beta \in O_c$, $\exists\, b$, b is an attribute of β, $\alpha \Re b$, and $b \Re \gamma$, such that $\alpha \Re b \wedge b \Re \gamma \rightarrow b \Re \gamma$.

Property 1 describes the condition for an indirect data dependency between two objects via a third object, a simple or complex object. If the third is simple, the dependency can be computed directly with transitive property. Otherwise, there exists one component of the third object via which an indirect dependency between original two holds. With this property, one can find indirect data dependencies precisely without the problem described in prelim.

3. Remark

According to the previous definition and property, an indirect data dependency can be computed precisely with transitive property. When performing the computation, one has to decompose encapsulated components of complex objects. Undoubtedly, the algorithm for identifying such dependency is not cost-effective because each complex object needs to be decomposed into bottom simple objects eventually. To avoid this, one can pursue an approximate solution by restricting the level of encapsulation at which a complex object is decomposed. The algorithm for the solution is more cost-effect, even though the solution may contain some spurious dependent relations (like the example in Figure 1). In spite of this, an approximate solution is still useful because it is more precise than that from extended PDG's directly. Furthermore, OO paradigm brings building block programming style. For example, C++ standard template library [StLe95] provides basic data structure components, container classes. These classes encapsulate a number of objects that usually encapsulate their own attributes and methods. Therefore, we suggest that the level of encapsulation at which a complex object is decomposed should not be less than

two.

Acknowledgment

This research was partly sponsored by National Science Council, Taiwan, ROC, under contract No. NSC85-2213-E009-030.

References

[FeOt+87] J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9(3), pp.319~349, July 1987.

[HoRe+91] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems,* vol 12(1), pp.26~60, 1990.

[LaHa95] L.D. Larsen and M.J. Harrold, "Slicing Object-Oriented Software," Technical Report No.95-103, Department of Computer Science, Clemson University, 1995.

[MaMc+94] B.A. Malloy, J.D. McGregor, A. Krishnaswamy, and M. Medikonda, "An Extensible Program Representation for Object-Oriented Software," *ACM SIGPLAN Notices*, vol. 29(12), pp.38~47, 1994.

[OtOt84] K.J. Ottenstein and L.M. Ottenstein, "The Program Dependence Graph in a Software Development Environment," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp.177~184, 1984.

[StLe95] A. Stepanov and M. Lee, "The Standard Template Library", Technical Report, Hewlett Packard Laboratories, 1995.

[ZhCh+96] J. Zhao, J. Cheng, and K. Ushijima, "Static Slicing of Concurrent Object-Oriented Programs," submitted to COMSAC'96, IEEE, 1996.