Towards Transitory Encapsulation

Sebastian Fleissner

Research School of Computer Science
The Australian National University
sebastian.fleissner@anu.edu.au

Abstract

Encapsulation and information hiding are essential and fundamental to object-oriented and aspect-oriented programming languages. These principles ensure that one part of a program does not depend on assumptions on the internal structure and logic of other parts of the program. While this assumption allows for clearly defined modules, interfaces and interaction protocols when software is initially developed, it is possible that rigid encapsulation causes problems, such as brittleness, as software changes and evolves over time. We suggest that, just as the strength of type systems have relaxed over time, perhaps structural boundaries could, too be relaxed. Perhaps there could be a new kind of flexible encapsulation: one that allows non-permanent and flexible boundaries between program parts.

Categories and Subject Descriptors D.3.3 [*Software / Programming Languages*]: Language Constructs and Features

Keywords Flexible Encapsulation

1. Introduction

Type systems started out non-existent, then gradually moved to being strongly enforced, and then (in some languages) gradually eased back again, into an arrangement where types are inferred by the interpreter. Work flows followed a similar forth and back arrangement, with the provision of strict directory structures, aligned along structural boundaries and defined by the user, and then with the introduction of Mylyn relaxing back into a dynamically derived collection of related elements.

At least to date there has been no major easing of encapsulation, such that it, like types in dynamic languages, can be inferred. There are times when rather than having a manually defined structure, the programmer should be able to rely on the interpreter for delineation of an "object". What if a programmer needs a dynamically defined entity that is more flexible, fluctuating in its encapsulation over time, or from different perspectives. Or perhaps, the encapsulation of some entities are so complex that it cannot be straightforwardly manually defined but should instead be trusted only to the interpreter to construct?

2. A Case for Something Different

The relaxation of type systems is limited. While types are now inferred, they rarely change (during runtime) once defined. It is commonly accepted that large modular or object-oriented programs become more brittle as they evolve, because of the static nature of interfaces. As pointed out in [1] and [3], evolution of software systems can lead to complex and inflexible designs, which in turn lead to a huge amount of effort for enhancements and maintenance. Rigid interfaces, which facilitate interaction between objects, can not easily be changed once the implementation of an object-oriented program has commenced, because a change in one object's interface can result in a ripple effect that leads to adjustments to many other objects.

Lets consider an object-oriented program in which two or more objects have a subject-observer relationship, also known as the observer pattern [2]. Figure 1 illustrates the conceptual design of this pattern.

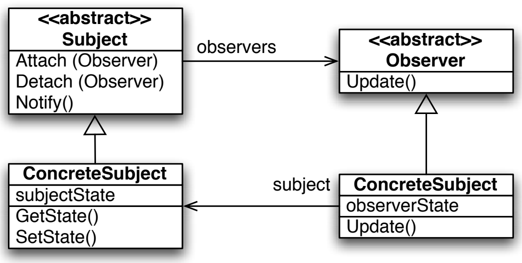

Figure 1. The Observer Pattern

Although the implementation of the Observer pattern is not very complex, its existence alone underlines the lack of mechanisms for defining relationships, other than static in-

SPLASH Companion'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3722-9/15/10
http://dx.doi.org/10.1145/2814189.2817274

heritance relationships, between objects in object-oriented programming. Furthermore, even though the Observer pattern facilitates a small degree of flexibility, the protocols for registering, unregistering, and notifying observers have to be fixed during the design phase and later even small changes can result in a snowball effect that forces modification of many other objects. Another issue is that the Observer pattern requires the subject to be aware of its observers, to maintain a list, and to explicitly dispatch information to each of them whenever its state changes. However, the spirit of observation is that an observer should be able to observe a subject that is oblivious of being observed.

3. Transitory Encapsulation

Here is what transitory encapsulation of program entities could look like:

- The scope of an entity's encapsulation is defined by a boundary and the scope of encapsulation can change over time (expand or shrink) as the software is running.
- Scope boundaries can be changed from rigid to flexible and vice versa at any time.
- Existing scope boundaries can be removed and new boundaries can be added during runtime.

Relaxing the boundaries would have a profound impact on how information is shared or passed from one entity to another. Recall the example of the clunky object-oriented subject-observer relationship. Lets imagine a programming language and associated virtual machine supporting the flexible boundaries of transitory encapsulation. In this case a subject-observer relationship could be established by simply changing the scope of the subjects boundary or temporarily removing the boundary completely. Then the observer could observe a certain piece of information in the subject actively, without requiring the subject to know each observer send out notifications whenever a change occurs. Figure 2 provides a visual representation of what the subject-observer relationship using flexible boundaries might look like.

In Figure 2 the subject and observer are shown as program entities that are located at certain positions in a virtual two-dimensional space. The scope of the boundary of the subject is extended and other and any other entity located within the circle, in this case the observer, can read the data associated with the subject. As a result, instead of having the subject notifying and passing data to the observer, the observer simply reads data from its current position in the virtual space.

It is important to note that we consider transitory encapsulation as a new principle at the programming language and virtual machine level and not as a feature provided by a software architecture or middle-ware. Most of today's popular programming languages are textual that tend to enforce rigid encapsulation to some degree. Generally, the scope of encapsulation is fixed: if a piece of information is outside an entity, it is going to stay out for as long as the program is running unless it is somehow passed into the entity through a well defined interface.

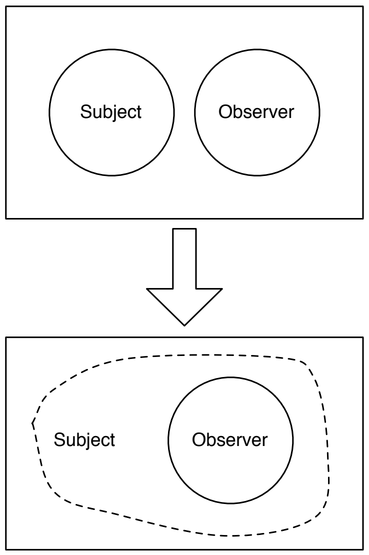

Figure 2. Subject-Observer with Flexible Boundaries

4. Conclusions

Rigid encapsulation is not a bad thing. If fact, we believe it is a great principle for creating modules that - at least in theory - can be easily separated from one another and then recombined. The internal structure of a module can change without affecting other modules as long as the public interface remains the same. However, rigid encapsulation can become a problem when the public interface changes - and they do change sooner or later. While refactoring tools can help to some extend, the more complex the system grows, the more damage changes can cause.

The benefits of having transitory encapsulation - especially in combination with traditional scoping systems - allows programs to reorganize themselves when new entities are introduced or existing entities are adjusted. It would allow developers to add functionality without having to change existing interfaces.

References

[1] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *TOOLS '99*, page 18, Washington, 1999. IEEE Computer Society. ISBN 0-7695-0278-4.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison Wesley, 1995. ISBN 0 201 63361 2. http://www.aw.com.

[3] C. L. Nehaniv, J. Hewitt, B. Christianson, and P. Wernick. What software evolution and biological evolution don't have in common. In *Second International IEEE Workshop on Software Evolvability*. IEEE Computer Society, 2006.