Patterns in Java
Encapsulation

Kevlin Henney
kevlin@curbralan.com

It may seem odd to write about such a fundamental topic, but it is precisely because it is so fundamental that makes it worth writing about. I find that in writing about patterns — specifically those related to software design — encapsulation emerges as a recurring theme, either explicitly or implicitly. It is fundamental and recurring, but I would not necessarily call it a pattern: it is more of a principle, and one that forms a motivation for, and consequence of, many patterns.

Encapsulation is often thought and taught solely in terms of objects. A simple but common view is that encapsulation is concerned with making an object's data inaccessible to the outside world, and that a public method interface is provided to give a more service-oriented view of an object. Encapsulation is also commonly seen as a property of programming languages, typically expressed in terms of keywords like `public` and `private`.

This view is not so much wrong as narrow and not useful. It can lead to tidy coding guidelines — e.g. "always make data private" — but it doesn't offer the programmer a great deal in terms of design wisdom and motivation. Encapsulation is a property of a design rather than a language mechanism. It is perfectly possible to write encapsulated code in C without trying too hard or bending the language in any way, yet it clearly has neither objects (in the object-oriented sense of the word) nor type-level privacy.

Encapsulating

So what does encapsulation refer to?

- The self-containedness of a unit. The degree to which it is a cohesive and the degree to which it can reasonably reduce its dependencies on other units.
- A unit's ease of use, which relates to the surface area and simplicity of its interface. Importantly, simple is not the same as simplistic.
- The extent to which potentially changeable design decisions leak from a unit through its interface.
- The reinforcement of constraints, such as class invariants, thread safety, exception safety, etc.

Language features can help a great deal, but they are neither necessary nor sufficient. Likewise, the unit of encapsulation is not necessarily the class or the object, as is commonly believed. It can extend up to the package and the class hierarchy and down to the individual method. Object-oriented mechanisms make the realisation of encapsulation easier, but they do not necessarily cause it — you can write Fortran in any language.

Encapsulation implies both separation and unification. There is a separation of concerns, often expressed as the separation between a meaningful interface and a detailed implementation of some kind. There is also a unification: as the term *encapsulation* suggests, features are combined *in a capsule*. Dependent ideas are expressed dependently. For example, the simple act of combining data and function together in the same unit is part of the encapsulation concept. A change in one can lead to a change in the other; hence keeping them together ensures a stable dependency and a simple model for the programmer.

Encapsulating Methods

Taking a data-hiding-centric view of encapsulation often leads to the heavy use of Getters and Setters [Henney2003]. Instead of focusing on the capabilities that an object should have and the services it should provide, the focus is instead on the data representation first. Then, because public data is considered somewhat gauche in object circles, the next 'design' step is to make all the data private and provide corresponding Getters and Setters. Such classes are barely encapsulated. They expose too much of their internal structure to the world, but fail to capture the intended use of the objects of that class. Such record-based design requires class users to write and rewrite all of the manipulation code for objects themselves. In other words, the interface has not made the object easier to use and it has tied public users to the private implementation, so although it has used the language features it is not really encapsulated.

I even know of one company where one programming team labours under the tyranny of an even more extreme anti-guideline: each private field must have a public `get` and a public `set` method. It is almost as if they are willing the programmers to reduce the quality of their code and make the designs overly complex and their code verbose. By requiring that every field is both readable and writable the programmer must either write lots of additional code to enforce the correctness of the internal representation or they must just hope that class users will not use their class incorrectly (wish-based programming).

Constraint enforcement is an important consideration. In a multithreaded environment, it should be easy to use objects that are shared between threads correctly, and as hard as possible to use the incorrectly. We want these concerns expressed on the inside of a 'capsule', not the outside. Thread safety should not be the caller's full responsibility.

Consider an interface that has been designed using Command–Query Separation [Henney2000], a reasonable and desirable sequential programming practice:

```
interface EventSource
{
    Handler getHandler(Event event);
    void installHandler(Event event, Handler newHandler);
    ...
}
```

Let us assume that, in a multi-threaded system, each individual method will be implemented in a thread-safe fashion. However, it is not safe to assume that a sequence of calls to these methods will offer the same integrity guarantee. The user of an `EventSource` object must make the arrangements for such safety, which seems to normally involve the use of the `synchronized` keyword:

```
class Dispatcher
{
```

```
    ...
    public void changeHandler(Handler newHandler)
    {
        synchronized(eventSource)
        {
            oldHandler = eventSource.getHandler(event);
            eventSource.installHandler(event, newHandler);
        }
    }
    public void restoreHandler()
    {
        eventSource.installHandler(event, oldHandler);
    }
    private EventSource eventSource;
    private Event event;
    private Handler oldHandler;
}
```

This arrangement is inconvenient for the caller, and will be repeated all over the program wherever `EventSource` is used. But at least it's thread-safe... or is it? What if the `EventSource` implementation is a Proxy [Buschmann+1996, Gamma+1995]? All the `synchronized` block will have achieved is the locking and unlocking of a proxy object, not the real target object. Oops.

Such a design cannot be said to be encapsulated because, for the environment it was intended for, the interface is not self-contained or easy to use, and it does not preserve the constraints that matter to the programmer. Always be suspicious of `synchronized` when it used outside the object that is supposed to be made thread safe. A Combined Method [Henney2000] addresses the problem by combining all of the actions that must be taken together into a single method:

```
interface EventSource
{
    Handler getHandler(Event event);
    Handler installHandler(Event event, Handler newHandler);
    ...
}
```

The requirement for thread safety is now all with class author, who can arrange for thread safety inside their class rather than outside. The caller is now freed from this responsibility:

```
class Dispatcher
{
    ...
    public void changeHandler(Handler newHandler)
    {
        oldHandler = eventSource.installHandler(event, newHandler);
    }
    ...
}
```

Encapsulating Classes

Methods that encapsulate concerns make objects easier to use. The concern can sometimes affect the whole organisation of classes themselves, and not just their methods. Returning to the Getters and Setters problem, it makes sense for some strongly informational classes, such as for value or entity objects, to offer these methods as part (but not all) of their interface. However, what does not make sense is the requirement that all query methods should have corresponding modifier methods. It is a common habit for many programmers to provide these in pairs, for reasons of false symmetry, but this can have the effect of making their design harder not easier to use. The role that symmetry should play is that of simplification, not complication. In design aim first for transparency, followed by symmetry, followed by necessary and effective asymmetry.

Enforcing constraints, such as the correct and coherent modification of representation, becomes harder, and encapsulation can be weakened as a result. Returning to the context of threading, it is a mistake to design all classes to be thread-safe, but classes that are to be used in a threaded environment should be safe by simplicity and reduction not safe by complexity and addition. Sharing fine-grained value objects between threads should be transparent, requiring no work on the part of the user to ensure integrity. Using Immutable Value objects partnered with Mutable Companion objects [Henney2003] creates a more encapsulated design across two classes than by trying to contain all of the design in a single one.

Reducing the mutability of a class interface is generally a good thing. You can keep tight control over the state space of the object and, in the limit, potentially reduce it to immutability, which offers the greatest ease of use and simplicity in design. Ironically, real 'convenience' is often lost at the expense of 'convenience methods'. In the case of Immutable Value, better encapsulation is achieved through a narrower interface.

Encapsulating Hierarchies

An arrangement of many classes, such as the Immutable Value and Mutable Companion pairing, can sometimes offer more encapsulation than a single class, but this is not the only form of class community that supports encapsulation. More tightly nit than a community of colleagues is a family: when carefully designed a class hierarchy offers a strong form of encapsulation to its users.

Inheritance is the strongest form of coupling possible in an object-oriented system. It binds the public semantics and syntax of potentially many subclasses to a superclass, and the incautious use of non-private implementation features can further bind the hierarchy to internal details of the superclass. This coupling gives rise to the so-called *fragile base class problem* [Szyperski1998]: any change in the superclass will ripple out to both subclasses and class users, potentially breaking either their compilation or their correct execution. And the deeper the hierarchy and broader the user base, the deeper and broader the problem can become.

The irony of inheritance is that it can also be used to support loose coupling. Restricting its application and focusing on the contract published by an interface, rather than the implementation behind it, leads to the deliberate use of `interfaces`. Programmers are often taught about Java's `interface` feature in conjunction with abstract classes; programmers are sometimes told that the `interface` inheritance model is to make up for the restricted single inheritance model for classes. This is not a very useful view at all, and can even be considered harmful. It suggests that `interface` is a poor cousin of `class`. The truth is that by separating the public interface from a class into a separated interface, the resulting design is clearer to the user and easier to implement against for the class author. We might term

this practice Interface Decoupling, by analogy with Role Decoupling [D'Souza+1999]. Class users depend only on the published signatures. They can focus on method signatures and `javadoc` comments, rather than wade through a mass of method bodies, to find the functionality they are after. The class author can provide changes and alternative implementations without causing a rebuild in the class user code, assuming the presence of a suitable object factory [Henney2002]. The `EventSource` example presented earlier was an example of this practice.

So what are the ingredients of a successful class hierarchy? We can follow many of the consequences and principles of Interface Decoupling further:

- A class hierarchy should not be too deep. For example, assume no more than three levels until you can demonstrate a genuine need for more.
- A class hierarchy should be fully abstract at its root, expressed preferably in terms of `interfaces`.
- A class hierarchy should only be fully concrete at its leaves, which is why I often refer to this style as Concrete Leaves.
- Public methods should not be added as you descend the hierarchy, only defined.
- The public method interface should be as narrow as possible, omitting unnecessary optionality.
- Methods should be overridden only when they are `abstract` in the class or interface above. In other words, avoid providing or overriding default method implementations.

These recommendations represent an ideal and a discipline, but not something that can be followed at all times. Design is about negotiated compromise, but it is always worth knowing what terms and conditions you are compromising.

The class hierarchy criteria listed often find themselves in opposition. To arrive at an encapsulated hierarchy often means trading one feature off against another. A Composite [Gamma+1995] is a good example of this trade off. The aim is to make the distinction between single objects and multiple objects as transparent as possible. However, this often results in moving operations that are specific to the composite part up to the root of the hierarchy — a move that seems to contradict conventional wisdom on good practice. The overall effect of such a hierarchy is encapsulating, cohesive and loosely coupled, even thought the public interface at the root of the hierarchy has itself become less cohesive.

Encapsulating Packages

In the specific sense of the word, we can think of packages in terms of the Java language feature. The public–private separation that they support allows a package designer to expose only as much detail as the package clients need, and no more. The package designer is free to change the internal details of the package without affecting the package user's code.

Packages can also be split to represent better separation of concerns. A Separated Interface [Fowler2003] is encapsulated at two levels: first, there is Interface Decoupling, so that the usable interface of a class is represented in a separate named `interface`; second, the interface and implementing class are placed in separate packages to allow independent deployment and fully separated development.

In the more general sense of the word, we can think of packaging in terms of grouping, whether in terms of deployable components, whole subsystems, architectural layers, etc. Each packaging unit should be organised around a single and easily identifiable concept.

However, the concept should be cohesive rather than coincidental. For instance, although a package named `exceptions`, holding all the exception classes for a system, at first appears to be organised around a single concept, the concept is not a cohesive one. Such a package becomes a bucket. It is frequently changed — every new exception condition in the system leads to the package being modified — and is used everywhere, needlessly attracting the coupling of the whole system.

Likewise, be suspicious of any package named `util`, `utils` or `utility`. Such packages often represent a lack of design imagination, becoming dumping grounds for almost anything that doesn't fit into other pre-existing packages. Ultimately everything in a software system is a utility, so classifying utilities as distinct from other concepts is not particularly helpful. The `java.util` package is a case in point. It is a ragtag jumble of classes, some dealing with dates, some dealing with collections, some representing design dead ends (e.g. `Observer` and `Observable`), and so on. The cohesive groupings, such as date handling and collections warrant packages of their own, just as text handling and I/O have their own packages. It is important that intra-package cohesion is not undermined by inter-package coupling. In Java there is a cyclic dependency between `java.lang` and `java.io`, which means that in spite of the naming of `java.lang`, no single package can be considered the most fundamental of the Java language type system.

The need for acyclic package dependencies is a general principle for any form of packaging. In some ways it can be considered to be the defining principle in Layers [Buschmann+1996]. Layering is often considered with respect to abstraction, so that each layer in a system represents a progressively more abstract concept. However, there are many different reasons, and therefore competing forces, that can drive the separation of a system into layers: level of abstraction; different rates of change [Brand1994]; technology; development skills; organisational structure; geographical separation. Sometimes these are aligned, sometimes they are in conflict. The architecture must balance these so that each layer encapsulates a concern.

References

[Brand1994] Stewart Brand, *How Buildings Learn: What Happens After They're Built*, Phoenix, 1994.

[Buschmann+1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, 1996.

[D'Souza+1999] Desmond F D'Souza and Alan Cameron Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Wiley, 1999.

[Fowler2003] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Henney2000] Kevlin Henney, "A Tale of Two Patterns", *Java Report* 5(12), December 2000, available from `http://www.curbralan.com`.

[Henney2002] Kevlin Henney, "Symmetrie in Java", *JavaSPEKTRUM*, July 2002, available in English as "The Importance of Symmetry" from `http://www.curbralan.com`.

[Henney2003] Kevlin Henney, "Unvollendete Symmetrie", *JavaSPEKTRUM*, May 2003, available in English as "Unfinished Symmetry" from `http://www.curbralan.com`.

[Szyperski1998] Clemens Szyperski, *Component Software*, Addison-Wesley, 1998.