Kevlin Henney / kevlin@curbralan.com

A tale of two patterns

THERE IS A common myth that design can and should be independent of implementation detail, where *design* is commonly treated as an abstract concept and *implementation* as its concrete realization in code.* If we accept that "design is a creational and intentional act: conception of a structure on purpose for a purpose,"[1] then a structure cannot be fit for a purpose if it does not account for, and work with, its context. The context includes the target platform—language, tools, libraries, middleware, etc.—along with its functional and nonfunctional properties.

We would not think it reasonable to design a house without knowing the lay of the land, or a skyscraper without knowing the materials that could be used. The idea that we can treat concepts such as threading and distribution as mere coding detail is a sure fire way to waste a lot of energy (and time, money, etc.) in big, up-front design only to discover that the difference between theory and practice is bigger in practice than in theory. While there are certainly cases when parts of a high-level design can be invariant across different technologies, it is more often the case that we need to close the loop, allowing (even encouraging) knowledge of detail and practice to influence and inform the system's structure.

The role of patterns is to capture such architectural knowledge. They can map—prospectively or retrospectively—a design and its rationale, narrating from problem to solution, accounting for context, capturing the forces at work and the resulting consequences. Here, I focus on two patterns—Command-Query Separation and Combined Method—for allocating responsibility to methods in a class interface, examining how they interact and react to the context of concurrent and distributed vs. sequential and local execution.

Kevlin Henney is an independent consultant and trainer

Interface Design. In the general sense of the word, an *interface* represents a boundary between different systems or parts of a system. In software, an interface represents a partition that separates intent from realization, conceptual from concrete, and user from author. In Java, there are many interface concepts: the public section of a class represents an interface of methods and classes to all other potential users, its protected section an interface to its subclasses and its surrounding package, and so on. An interface represents the promise of functionality realized in a separate class; a package has a publicly usable section; and reflection is a different mechanism for offering and using an object's methods interface.

Constraints and Affordances. Taking the user-versus-author perspective, an interface also establishes and names an intended model of usage. The methods offered in a class interface are intended for use in a particular way. It is these constraints that the class author aims to present and enforce—at compile time through the type system and at runtime through exceptions or return values. The simplest example of this is the commonly understood meaning of encapsulation: Privacy of representation ensures that the class user manipulates information and behavior only through the public method interface of a class.

There is, however, more to encapsulation than just keeping data private. Encapsulation refers more generally to self-containment in design. A class that requires you to know a great deal about how and when to call which methods (e.g., "when calling this method followed by that method in a threaded environment you must synchronize the object explicitly") is not as encapsulated as one that fully contains and hides such issues (e.g., "this method is thread-safe"). The former design is weaker. It suffers design leakage, with many of its constraints implied but not strictly enforced. This puts the onus on the class user rather than the class supplier to do the work in fulfilling the design, and is, inevitably, tedious and error prone.

* Although in common currency, whether these definitions are actually valid and useful is another matter: It is the subject of much debate and probably another article!

In this light, *affordances* describe possible as opposed to intended uses[2]:

> The term *affordance* refers to the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used. A chair affords ("is for") support and, therefore, affords sitting. A chair can also be carried. Glass is for seeing through, and for breaking ... Affordances provide strong clues to the operations of things. Plates are for pushing. Knobs are for turning. Slots are for inserting things into. Balls are for throwing or bouncing. When affordances are taken advantage of, the user knows what to do just by looking: no picture, label, or instruction is required. Complex things may require explanation, but simple things should not. When simple things need pictures, labels, or instructions, the design has failed.

One of the class designer's responsibilities is to reduce the gap between constraints and affordances in the interface, matching intended and actual degrees of freedom, minimizing the scope for incorrect usage.

Context-Sensitive Design. Separating execution of a method in space or time—e.g., threads, remote method calls, or message-queued requests—can have a profound effect on the correctness and effectiveness of a design. The consequences of such separation cannot be papered over and ignored: concurrency introduces nondeterminism and context switching costs; distribution introduces concurrency and method calls that are subject to failure and escalated round-trip costs. These are design issues, not tweaks and bugs to be mopped up later.

In each case, one of the consequences is to discourage property-style programming—where an interface predominantly comprises get and set methods, each corresponding directly to a private field. Such classes are barely (meaning both *only just* and *naked*) encapsulated. Field accessors in an interface are often uninformative: They fail to communicate, simplify, or abstract the use of an object, leading to verbose and often error-prone user code. Property-style programming is not a great practice at the best times. Concurrency and distribution amplify these basic stylistic and practical problems by introducing subtle correctness issues and serious performance overheads.

Transparency and Bugblatter Beasts. Abstraction allows us to ignore details as necessary, so that our design-thinking balances contextual forces rather than is swamped by them. The challenge is in knowing what details are or are not necessary. The problem† arises when the details ignored are actually necessary.

Design often strives to make contextual forces as transparent as is reasonably and practically possible. Transparency can, however, become a seductive idea: Perhaps it is possible to make threading or remote object communication completely transparent, so that the user is unaware of them when communicating with an object? The Proxy pattern[3] supports a degree of transparency for remote communication. It underpins the programming model of both RMI and CORBA. The use of a local, surrogate object sporting the same interface as a remote object and responsible for the bit-bashing detail of marshaling and unmarshaling allows the caller to use a familiar method call model. However, such distribution transparency is not complete and it is not the same as the absence of distribution: The effects, such as call failure and latency, cannot be hidden fully and must still be considered.[4] Transparency is not a towel.‡

Command-Query Separation

Ensure that a method is clearly either a command or a query.

Problem. Methods can have query characteristics, where they return a value in response to a question, and command characteristics, where they carry out an imperative action to change the state of an object. A method may therefore be

† Note that some people treat the words *problem* and *challenge* as synonyms. They are not. Anyone who believes they are is probably trying to sell you something. History, if rewritten in this style, would sound quite different: "Houston, we have a challenge."

‡ "A towel ... is about the most massively useful thing an interstellar hitchhiker can have ... [You can] wrap it round your head to ... avoid the gaze of the Ravenous Bugblatter Beast of Traal (a mindbogglingly stupid animal, it assumes that you can't see it, it can't see you—daft as a brush, but very very ravenous)".[5]

classified as a pure query, a pure command (or modifier), or a hybrid of both.

For example, in java.util.Iterator, hasNext can be seen to be a query, remove a command, and next an awkward marriage of both query and command:

```
public interface Iterator
{
   boolean hasNext();
   Object next();
   void remove();
}
```

It is not possible to query an Iterator object as to its current value without causing it to advance to the next value. This leads to a mismatch with the structure of a for loop, where initialization, continuation, access, and advancement are separated for clarity, defining the signature of a loop:

```
for(initialization; continuation condition; advance)
{
   ... access for use ...
}
```

The result of combining query and command responsibility in the same method is often less clear. It can obstruct assertion-based programming[6,7] and typically requires additional variables for retaining query results:

```
for(Iterator iterator = collection.iterator();
   iterator.hasNext();)
{
   Object current = iterator.next();
   ... use current...
   ... again use current...
}
```

Solution. Ensure that each method has strictly either command behavior or query behavior, so that methods returning values are pure functions without side effects and methods with side effects do not have return values. "Another way to express this informally is to state that *asking a question should not change the answer*."[7]

The combined modifier–query style is common in C, C++, and Java, but this does not automatically make it desirable. In sequential programming, segregation of roles can make the effect of code clearer. Concentrating on the essential—rather than optional—characteristics of iteration, the standard Iterator interface could be refined by applying Command-Query Separation:

```
public interface Iterator
{
   boolean hasCurrent();
   Object getCurrent();
   void next();
}
```

Pure query methods return the same results under the same conditions. Assuming no modifications from other threads or command methods, a query can be called multiple times without requiring extra variables to hold snapshots of query results:

```
for(Iterator iterator = collection.iterator();
   iterator.hasCurrent();
   iterator.next())
{
   ... use iterator.getCurrent()...
   ... again use iterator.getCurrent()...
}
```

In this case, the divorcing of roles is a happy one. The role of each method is clearer and more cohesive. You can refactor a piece of code to separate query and command roles with the Separate Query From Modifier refactoring[8]: "You have a method that returns a value but also changes the state of an object. *Create two methods, one for the query and one for the modification.*"

Command-Query Separation better supports assertion-based programming. It can, however, encourage property-style programming if the convenient use of a class is not considered. This is a stylistic problem that becomes a practical problem in the presence of threads, requiring explicit external synchronization by the object user. Different queries that are commonly used together can be combined together to encapsulate the synchronization. Similarly, multiple commands can also be grouped as Combined Methods. In the presence of distribution, this also resolves the cost of accumulated round trips. However, if there are commands and queries that must be performed as a synchronized group, the user is once again faced with the problem of synchronization. This is a problem that cannot always be correctly resolved by a synchronized block—e.g., where the user is calling, knowingly or unknowingly, a Proxy object rather than the actual target object.

Combined Method

Combine methods that are commonly used together to guarantee correctness and improve efficiency in threaded and distributed environments.

Problem. Interfaces that offer predominantly fine-grained methods can, at first, appear minimal and cohesive—both desirable properties. However, during use it emerges that some of these interfaces are primitive without being cohesive. They are simplistic rather than simple, forcing the class user to work harder to achieve common tasks and navigate any subtle ordering dependencies (temporal coupling) that exist between methods. This is both tedious and error prone, leading to code duplication—the main *smell* to be avoided in code[8]—and offering new and exciting opportunities for bug breeding.

A sequence of methods that must be performed successfully together sees problems in execution when confronted by exceptions, threading, or distribution. If two actions, commonly performed together, must follow commit-or-rollback semantics—they must both be completed successfully or the effect of the first one must be rolled back if the second fails—and both are expressed as separate methods, clean up falls to (or on) the class user rather than the class supplier.

The introduction of threading makes things even less deterministic: A sequence of method calls on a mutable object

is not guaranteed to have the desired outcome if that object is shared across threads, even assuming that each individual method is thread-safe. Consider the following interface to an event source that allows handlers to be installed and queried for particular events:

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

Interleaved calls from other threads can lead to surprising behavior. Assuming that the source field refers to an object shared between threads, it is possible that a different handler will be installed by another thread between statements 1 and 2:

```
class EventSourceExample
{
   ...
   public void example(Event event, Handler newHandler)
   {
      oldHandler = eventSource.getHandler(event); // 1
      eventSource.installHandler(event, newHandler); // 2
   }
   private EventSource eventSource;
   private Handler oldHandler;
}
```

Again, it is the consumer rather than the supplier that pays the cost of constraint preservation:

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

If the target object is remote, the additional overhead of round-trip costs and the liability of method-call failure join concurrency as part of the context. In the previous example, we can assume that the time to execute the body of each method is miniscule, by orders of magnitude, when compared to the latency of the communication there and back. This cost is paid twice in the example, and many more times in other examples.

There is a further problem with required use of an external synchronized block that is more obvious in a distributed example but applies equally well in local threaded examples: the use of proxy objects between the caller and the target. In short, the use of synchronized blocks fails because the proxy rather than the target is synchronized. It would be an understatement to say that this can have a rather fundamental impact on the correctness of a system. Because the use of proxy is transparent behind an interface (given an interface to an object, how can you be sure that you are talking to the target directly, i.e., the use of custom or JDK 1.3 dynamic proxies?), there is little the caller can do to guarantee good behavior.

Solution. Combine methods that must be executed together coherently in the event of failure, threading, and distribution. The combination should reflect common use. Thus, a Combined Method may be clearer than a more primitive set of methods because it reflects directly the intended use. Recovery strategies and awkward usage can be encapsulated within a Combined Method, simplifying the interface from the class user's perspective. This improved encapsulation reduces unwanted affordances in the interface. The overall effect of Combined Method is to support a more transaction-like style of method design than is traditional.

It is often reasonable to provide query methods hand-in-hand with a combined command-query. However, this should be done on an as-needed basis rather than as a force of habit, otherwise the interface can grow without restraint or reason. Providing separated command methods is less often required because a combined command-query can play that role: The caller simply ignores the result. If returning an unused result incurs a cost—e.g., it is marshaled across the wire—it may then become reasonable to provide a separated command method.

Returning to the previous example, the design becomes simpler and more self-contained if the installHandler method returns the previous handler:

```
class EventSourceExample
{
   ...
   public void example(Event event, Handler newHandler)
   {
      oldHandler =
            eventSource.installHandler(event, newHandler);
   }
   private EventSource eventSource;
   private Handler oldHandler;
}
```

The caller has been offered a safer interface and no longer has to account for threading. This reduces the risk as well as the size of the code, placing responsibility for class design fully with the class designer rather than the class user. The presence of Proxy objects does nothing to affect the correctness of usage.

A Combined Method may be a combined set of queries, a combined set of commands, or a combination of both. Thus, it may complement or contradict Command-Query Separation. When there is conflict between the two practices, Combined Method should be applied in preference where it would make a difference in correctness and usability.

As another example, consider the case of acquiring a resource if it is available and continuing with something else if it is not. Assume, in the following interface, that the acquire method blocks until the resource becomes available:

```
interface Resource
```

```
{
    boolean isAcquired();
    void acquire();
    void release();
    ...
}
```

Something like the following usage code has been recommended as appropriate for a threaded system[6]:

```
class ResourceExample
{
    ...
    public void example()
    {
        boolean acquired = true;
        synchronized(resource)
        {
            if(!resource.isAcquired())
                resource.acquire();
            else
                acquired = false;
        }
        if(!acquired)
            ...
    }
    private Resource resource;
}
```

However, even leaving aside issues of readability and usability, this design is an unsuitable application of Command-Query Separation. It fails if we introduce a Proxy arrangement:

```
class ActualResource implements Resource {...}
class ResourceProxy implements Resource {...}
```

A Combined Method resolves the issues, making the presence of concurrency and indirection more transparent:

```
interface Resource
{
    ...
    boolean tryAcquire();
    ...
}
```

The following is clearly simpler as well as correct (and is arguably, for those reasons alone, the more tasteful design):

```
class ResourceExample
{
    ...
    public void example()
    {
        if(!resource.tryAcquire())
            ...
    }
    private Resource resource;
}
```

A consequence of Combined Method is that it can make some testing and assertion-based programming practices more awkward, where both command and query roles are combined. However, design by contract, in its original form, is not always an appropriate practice when reentrancy, threading, and distribution are also features of the design. Unit-testing approaches can provide the appropriate separation and level of confidence in these situations. Combined Method does not significantly complicate unit testing. If applied without consideration for usage, Combined Method can make a method interface less clear and the user's code longer and clumsier. In some cases, Execute Around Method[9] offers an alternative to Combined Method that can ensure atomicity and flexibility.

Conclusion

Whether or not a particular practice is appropriate can be heavily dependent on the context and if it is possible or practical to make the forces arising in that context transparent. In the case of Java's threading model, the oft-considered good habit of using Command-Query Separation can become a poor design practice. For this reason, although Command-Query Separation was articulated originally as a principle,[7] it is in truth a pattern: A principle is a general or universal truth; a pattern offers specific advice on how to build something, and its applicability is context-dependent.

The Combined Method pattern can reinforce Command-Query Separation, but in some cases it is clear that the two patterns are in tension. While both are focused on class interface design, the forces at work in each pattern are not the same, nor is their context of applicability. Command-Query Separation is driven by requirements of testability, comprehensibility, primitiveness, etc. and is applicable without risk in local, single-threaded contexts. Combined Method is driven by correctness in concurrent and distributed environments and the desire to encapsulate rather than leak nontransparent details.

Does the combination of command methods and query methods produce a less pure design for concurrent and distributed systems than for sequential ones? The answer in part hinges on what you mean by *purity*. Looked at another way, a design using method combination resolves more design forces than one not using them, resulting in a design that is fitter for purpose and more encapsulated than the alternative. If *pure* means *clean*, then this is more—rather than less—pure. Context determines practice. ■

References

1. Henney, K., "Design: Concepts and Practices," keynote at *JaCC Conference*, Sept. 1999, available via **www.curbralan.com** and **www.accu.org**.
2. Norman, D., *The Design of Everyday Things*, paperback edition, Basic Books, 1988.
3. Gamma, E. et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison–Wesley, 1995.
4. Waldo, J. et al., "A Note on Distributed Computing," Sun Microsystems Laboratories, Nov. 1994, **www.sun.com/research/techrep/1994/abstract-29.html**.
5. Adams, D., *The Hitchhiker's Guide to the Galaxy*, Pan, 1979.
6. Mannion, M., "Concurrent Contracts: Design by Contract™ and Concurrency in Java," *Java Report*, Vol. 4, No. 5, May 1999, pp. 51–66.
7. Meyer, B., *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.
8. Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison–Wesley, 1999.
9. Beck, K., *Smalltalk Best Practice Patterns*, Prentice Hall, 1997.