Another tale of two patterns

PROPER TOMATO KETCHUP is thixotropic. Its notorious reluctance to leave its bottled home is matched only by its enthusiasm to keep on pouring when it finally starts to flow—"You shake and shake and shake the bottle, first none comes, then the lot'll."

Ejection solutions range from the therapeutic direct application of brute force to the more counterintuitive. Directly hitting the bottom and shaking the bottle with the top off eventually pays off but lacks predictability, elegance, and economy. The intervention of a third party, such as a knife, often helps to coax the reluctant liquid from its bottle. As a virtue, patience is often its own reward: It also eventually yields ketchup if the bottle is up-ended and held stationary. You can also meet the non-Newtonian phenomenon of thixotropy head on, using the properties that make it so awkward to your advantage. For instance, to convince the ketchup that it really would rather behave as a liquid than a solid, some big up-front shaking of the bottle (with the top on ...) can get it in the mood.

Perhaps the most elegant percussive solution is to work with it in its solid state. Hitting the base of the bottle serves only to move the bottle in the direction of motion: The ketchup is otherwise unmoved by your sense of urgency. If you move the bottle sharply in the opposite direction, however, the same physics apply except that now the relative motion of the bottle, with respect to the inertial ketchup, means the ketchup progresses toward the open end. Hit the side of the wrist holding the bottle with your other fist for best effect.

Patterns capture design practices that address a particular recurring problem. In some cases, the solution is a reinforcement of something already known and sometimes observed—a guideline that clarifies understanding and encourages a more consistent application of the practice. In other cases, the solution is something counterintuitive, almost turning the problem—and many common solutions—on its head to arrive at the result.

Moving from sauce to source, but staying with the themes of competing solutions and controlled flow, I will look at the forces affecting idiomatic programming in Java and two approaches—FINALLY FOR EACH RELEASE and EXECUTE-AROUND METHOD—that deal with exception safety.

Habits and Idioms

Convincing a compiler that it wants to accept your code is a matter of correctness. You soon acquire the habits and scars that allow you to write code that is correct as far as the compile-time rules of the language are concerned. You are then left with two outstanding issues: writing the right code, and writing it right. A mix of conscious design choice (part novel and part experience) and unconscious habit (derived from experience) resolves these issues.

Patterns. Patterns capture practice, but two pattern features are commonly forgotten in the rush to apply them: context and consequences. The context of a pattern describes when it can be applied and what forces give rise to the design problem you are confronting. A pattern has consequences (also known as its resulting context), some that are beneficial, and some that are potential liabilities. Patterns are not cookie-cutter solutions that can be applied independent of system: Design is about making decisions, exploring options, and balancing tradeoffs, which is where an understanding of context and consequences comes in. Without these, you might as well go back to shaking the ketchup bottle and hoping that luck and brute force will eventually win out—brute force eventually gets you ketchup, but only luck determines whether or not the application is where you intended.

When applied to programming, idiom refers to both patterns and syntax-level conventions. In terms of patterns, "idioms are low-level patterns that depend on a specific implementation technology such as a programming language," according to *Software Patterns.*[1] In terms of syntax conventions, consider the set and get prefixes for property accessors. While this usage is definitely idiomatic—codified originally in JavaBeans—it is unrelated to pat-

Kevlin Henney is an independent consultant and trainer based in the UK.

terns; prefixing with get and set does not solve any design issues in general Java code. It is perhaps a sign of over-eagerness to be buzzword-compliant—albeit at the cost of accuracy—that led the JavaBeans documentation to refer to this naming convention as design patterns.

Java as Context. Java idioms are patterns that have the Java language and libraries as part of their context. There are particular features that influence the expression and formulation of a design. Idioms vary between languages for this reason.

For instance, because of a unified class hierarchy, reflection, and garbage collection, some Smalltalk idioms can be carried over to Java. However, because of the fundamental differences between dynamically and statically checked type systems, others cannot. Some can be carried over with a little re-interpretation. For example, in Smalltalk, blocks of code are treated as objects and can be passed around as independent objects for execution by other methods. This feature simplifies the use of many COMMAND-based designs.[2] Java, however, does not integrate blocks directly into its runtime object model. In my article, "Java Patterns and Implementations,"[3] the BLOCK idiom emulates the Smalltalk concept by taking advantage of Java's anonymous inner-class feature. With a little syntactic overhead, multiple statements can be grouped together and passed around for subsequent execution. The important feature is that this code can refer to final local variables, and fields and methods in the surrounding object (see Listing 1).

Without inner classes, the alternative would be to define a separate named class. The local variables and a reference to the surrounding would have to be passed in explicitly at construction, assuming the methods and fields had appropriate visibility. Some uses of anonymous inner classes leave a lot to be desired. A good way to obscure code is to introduce a large inner class with multiple methods halfway through an expression. The stylistic difference with the BLOCK idiom is its constraint of the method count to one, i.e., the execution of the block.

Constraints and Affordances. Design can be seen formulated in terms of constraints, the intended degrees of freedom in a model. According to D. Norman's "The Design of Everyday Things,"[4] the flip side of constraints is affordances, the actual degrees of freedom present, i.e., the usage that a design affords. This includes the ways in which the design can be misused. How easy is it to misuse an interface? To forget that whenever you call method a you should follow it with a call to method b? In other words, temporal coupling between methods. In my last column[5] I explored this issue with respect to thread safety and fine-grained property methods. Object interfaces that are not designed to prevent thread-unsafe usage will—inevitably—be used incorrectly, through no fault of the user, simply because they sometimes have more to remember than they can commit to habit.

Paired Actions

Paired actions are a common enough turn of phrase in any programming environment: open a file, use it, close it; acquire a lock, use the resource, release the lock; set a wait cursor, perform a task, restore the previous cursor. The most common form of this sequence is related to resource acquisition and release—in sketch form:

```
resource.acquire();
resource.use();
resource.release();
```

LISTING 1

Example of the BLOCK idiom.

```java
interface Command
{
    void execute();
}

class Client
{
    public void register(Service service, final ArgType arg)
    {
        final LocalVarType local;
        ...
        service.setAction(
            new Command()
            {
                public void execute()
                {
                    arg.method();
                    local.method();
                    field.method();
                    method();
                }
            });
        ...
    }
    private void method() ...
    private FieldType field;
}

class Service
{
    public void setAction(Command newAction)
    {
        action = newAction;
    }
    public void onEvent()
    {
        ... action.execute(); ...
    }
    private Command action;
}
```

Catch as Catch Can. Apart from the tedious and error-prone nature of this repetition in code ("every time I call acquire I must remember to also call release"), something else upsets this simple picture: exceptions. Forgetting the second call is a problem that is independent of programming language (although some developers are better at committing it to habit than others); but exceptions, and the features available for handling them, are particular to Java's context. Garbage collection and synchronized take care of any memory and monitor releases, but if resource.use() throws an exception, resource.release() will be missed. Ketchup everywhere.

The brute-force solution would be to recruit an ungainly assemblage of trys and catches:

```
resource.acquire();
try
{
    resource.use();
}
catch(Exception caught)
{
    resource.release();
    throw caught;
}
resource.release();
```

The repetition and intricacy compounds the scope for getting things wrong.

And Finally. Java supports the finally block for addressing this problem in a direct and simple fashion. This avoids repetition of the release code and the clumsiness of the catching flow. As far as many programmers are concerned, this is as much as they need to know: The presence of the mechanism is all that is needed to deal with programming with exceptions in Java. However, it addresses neither the (necessary) repetition of the release for each acquisition nor the scope for coding errors in arranging the try finally flow. There is more to this mechanism than the simple reactive advice "use try finally to guarantee release when you get an exception."

The FINALLY FOR EACH RELEASE idiom details the practice for safe and consistent use of try finally, whereas EXECUTE-AROUND METHOD tackles the root of the problem rather than its symptoms from a different angle. The former pattern is the one most familiar to Java programmers, and many almost take it for granted, whereas the latter is a pattern found in Smalltalk[6] but, because of awareness (or lack of it), is less common in Java.

Finally for Each Release

Always follow the first of a required pair of actions with a try finally *block.*

Problem. How can you ensure that the second action of a pair of actions is always executed correctly? Given a

pair of actions that must always be executed together, and that straddle a piece of usage code, it is easy to forget that an exception in the usage code will bypass the second action, e.g., the code resource.use() could generate an exception, rendering the code unsafe:

```
resource.acquire();
resource.use();
resource.release();
```

The compiler can remind the programmer of any forgotten checked exceptions in the signature of a method, and hence the presence of exceptions in a particular piece of code, and therefore the need to make it exception safe. However, runtime exceptions arrive unannounced and exception lists that are copied automatically—tool-generated or copy-and-pasted from another method's signature—may not act as a reliable aide-mémoire. The only certain, but unclear, reminder is the occasional resource-based bug in the event of an exception. Unsafe code can appear to work correctly for much of the time, making it all the more mysterious when things go wrong. The purpose of exceptions is to communicate existing problems, not to create new ones.

Where programmers recognize the code must be made exception safe, try finally provides the mechanism available for shoring up their code. However, it is not always a simple matter of division of statements between the try and finally blocks. For instance, here is code that incorrectly attempts to release an unacquired resource in the event of an acquisition exception:

```
try
{
    resource.acquire();
    resource.use();
}
finally
{
    resource.release();
}
```

Here is a solution that is verbose and makes the assumption that an already-acquired resource won't throw:

```
try
{
    resource.acquire();
    resource.use();
}
finally
{
    if(resource.isAcquired())
        resource.release();
}
```

Solution. Identify all paired actions in a piece of code

and, regardless of assurances to the contrary or silence in the documentation on the matter, assume that exceptions are always a possibility, and therefore there is always a need for exception safety with paired actions. The expression of the try finally blocks should include only the usage and release code:

```
resource.acquire();
try
{
    resource.use();
}
finally
{
    resource.release();
}
```

There are essentially two aspects to both the problem and the solution given by this pattern: the consistent application of try finally for all paired actions and the correct formulation of the try finally code used. This takes the basic try finally mechanism and associates it with a practice, instilling a habit for consistent ("don't wait for a reminder, always assume the worst") and correct ("only return what you have borrowed") application.

The benefit of this idiom is that it ensures consistency, safety, and correctness in all code that is bracketed by paired actions. Consistent application of try finally after the first action keeps the whole sequence simple.

The obvious liability is that it requires either continuous awareness or successful commitment to habit. A programmer may use a class and, because of other detail, forget the bracketing nature of the actions. Although try finally provides a mechanism for resolving code duplication that would arise if only try catch were used, it does not eliminate the duplication of the release code or indeed the need for try finally all over a system. Duplicate code is always something to be frowned upon—it smells bad (see M. Fowler's comments in *Refactoring: Improving the Design of Existing Code*[7])—and the failure to abstract the repetition is not something that FINALLY FOR EACH RELEASE addresses.

A release action that is potentially more involved can further increase the detail the programmer deals with and the duplication of code throughout a system, e.g., rollback depending on whether or not an exception is thrown:

```
boolean succeeded = false;
resource.acquire();
try
{
    resource.use();
    succeeded = true;
}
finally
{
    if(succeeded)
```

```
        resource.releaseWithCommit();
    else
        resource.releaseWithRollback();
}
```

Execute-Around Method
Encapsulate pairs of actions in the object that requires them, not code that uses the object, and pass usage code to the object as another object.

Problem. How can you ensure the second action of a pair of actions is always executed? Given a pair of actions that must always be executed together, and that straddle a piece of usage code, it is easy to forget either the second action or that an exception in the usage code will bypass the second action.

FINALLY FOR EACH RELEASE addresses dealing with paired-action exception safety through the recognition of paired actions and the adoption of appropriate code. However, it is not just the issue that it is not always apparent code may not be exception safe, but that the pairing is missing—trying to notice something by its absence is often harder than noticing something by its presence. The onus is very much on the class user to resolve these issues, whereas the use of paired actions is very much a consequence of the class design.

In principle, then, the responsibility for correct usage should lie with the class author. However, documentation and example code are the only recourse that class authors have to communicate to their class users that a paired action must be executed around their usage code. Even assuming this information is read, the user may need a reminder to apply it: Runtime bugs are the only reminders available. The class author can neither write a method interface that enforces the constraint at compile time nor a class implementation that guarantees detection of any error in use at runtime as soon as it occurs.

There are two related issues here that need to be addressed in a more complete design: temporal coupling that must either be enforced or encapsulated by the class designer, and code duplicated by the class user for more than a single use of the class.

Solution. Provide a method on the resource class that receives a COMMAND object—typically a BLOCK—that encapsulates the user's particular usage of the resource. The method performs the necessary pair of actions around execution of the COMMAND object.

```
resource.apply(
    new Command()
    {
        void execute()
        {
            resource.use();
        }
    });
```

The COMMAND object can access the resource object either as part of its own context, as shown, or be passed the resource object as an argument by the resource object itself, i.e., this, as follows:

```
resource.apply(
    new Command()
    {
        public void execute(Resource resource)
        {
            resource.use();
        }
    });
```

The resource class is responsible for the pairing of actions and should use FINALLY FOR EACH RELEASE:

```
class Resource
{
    public void apply(Command usage)
    {
        acquire();
        try
        {
            usage.execute();
        }
        finally
        {
            release();
        }
    }
    private void acquire() ...
    private void release() ...
    ...
}
```

This solution approaches the problem from a different angle, focusing on the usage code rather than the acquisition–release pairing. After all, from the perspective of the class user the motivation to have such an object is to use it, not to perform its bookkeeping operations for it. By focusing on this, the result is a better-encapsulated design: Encapsulation is about self-containedness in design, not just about privatizing data. The class designer now hides the policy within the class rather than leaving it as an outstanding task for the user.

The policy is expressed and enforced in a single place, reducing code duplication and accommodating change and complexity more easily. For instance, where rollback of the resource is required in the event of an exception, it becomes the responsibility of the class designer not the class user to ensure correctness:

```
public void apply(Command usage)
{
    boolean succeeded = false;
    acquire();
```

```
    try
    {
        usage.execute();
        succeeded = true;
    }
    finally
    {
        if(succeeded)
            releaseWithCommit();
        else
            releaseWithRollback();
    }
}
```

In appearance, the obvious liability of this idiom is that, in using BLOCK, there is a certain amount of non-functional syntactic overhead—whitespace, brackets, method name, etc. At runtime, there is the creation of an extra object to contain the usage, and object creation is often something that must be watched. At compile time, there is also the creation of an additional class. The extra class is effectively transparent in the code, but the inner class will appear as an associated .class file.

Conclusion

Different views of the same problem can expose quite different solutions, whether it is the need for ketchup or exception safety. In the case of Java, FINALLY FOR EACH RELEASE can be considered a common logistical idiom, whereas EXECUTE-AROUND METHOD is a tactical idiom that better encapsulates intended use. It builds on FINALLY FOR EACH RELEASE, but encapsulates it as internal detail. EXECUTE-AROUND METHOD is a more sophisticated solution, but in many respects simpler, higher level, and more elegant.

Design and usage problems should be solved by the designer not the user, so the designer needs to determine how best to minimize the scope, enforcing constraints and reducing unwanted affordances. ∎

References

1. Coplien, J., *Software Patterns*, SIGS Books, 1996, currently out of print.
2. Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995.
3. Henney, K., "Java Patterns and Implementations," *BCS OOPS Patterns Day*, October 1997, available from **http://www.curbralan.com** under the "plop & plunk" heading on the Papers page.
4. Norman, D., *The Design of Everyday Things,* paperback edition, Basic Books, 1988.
5. Henney, K., "A tale of two patterns," *Java Report,* Vol. 5, No. 12, December 2000, pp. 84-88, also available from **http://www.curbralan.com**.
6. Beck, K., *Smalltalk Best Practice Patterns,* Prentice Hall, 1997.
7. Fowler, M., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999.