might reasonably choose to copy values across nonlocal regions of memory even when move operation are available.[28]

On the other hand, making a *noncopyable* type movable introduces a distinct new semantic. If we know what it means to *copy* an object, then we have a complete specification for what it means to *move* it. Without that specification, we're charting new waters. Some of the reasons why a given type is noncopyable might well apply to moves too. Again, some noncopyable types might not have a natural uninitialized or "empty" state, so an appropriate **moved-from state** will need to be designed, documented, and tested — irrespective of whether the details of that state are made available to clients.

In practice, we find that most types of objects fall into two broad categories: (1) those that are used to represent a platonic value (VST) and (2) those that perform some sort of service (**mechanism**). Well-factored software components typically serve one or the other of these roles, but not both. For example, `std::complex<double>` is a VST, whereas `std::thread` is a mechanism. A standard container, such as `std::vector`, carries with it a fair amount of machinery, but most of that is in support of its value, which is typically a sequence of VSTs, whereas a scoped guard has no value whatsoever and serves only as a manager of the lifetime of some externally created resource.

Objects that have been made noncopyable are typically so because there's no platonic value to copy. If all that's needed is to share access to such mechanisms, then raw pointers might be sufficient. If, however, the need is to pass around unique ownership of such a noncopyable object, then that's another story; see *Use Cases — Passing around resource-owning objects by value* on page 771. In most cases, `std::unique_ptr` provides a standard and well-understood idiom for passing around unique ownership of noncopyable objects without the risks and development costs associated with crafting our own move-only type; see *Implementing a move-only type without employing `std::unique_ptr`* on page 791.

For example, suppose we have a preexisting (possibly C++03) mechanism that currently meets our needs. As this type doesn't try to represent a value, it doesn't implement any copy operations, equality comparison operations, etc. Suppose also that this type allocates dynamic memory. Although our mechanism cannot meaningfully be copied, it does have a reasonable default constructed state and, with the advent of move semantics in C++11, we could — in theory — implement arguably plausible move operations. Should we? What would be the return on our investment?

Adding move operations to such an inherently noncopyable type is unlikely to yield any meaningful utility that could not otherwise be achieved, far more safely and affordably, by an external application of `std::unique_ptr`. In addition, any attempt to retrofit an existing mechanism with move operations will invariably involve substantial development effort. Moreover, great care would be needed to ensure that (1) the documentation reflects the modified behavior, (2) appropriate new unit tests are added, and (3) the moved-from state of the object behaves sensibly. What's more, given that this mechanism might already be in wide use, such an invasive enhancement might break preexisting client code, designed

[28] See **halpern21c**.