When using generic operations with explicit dispatch on type, if new types are added to the system, every operation must be updated to handle the new types. However, if a new operation is added, nothing need be changed in the existing code. One must simply construct a new procedure for the new operation such that it dispatches on all the existing types.

With a data-directed style, either new types or new operations could be added in a standalone installation. None of the existing code would need to be updated. However, depending on how the packages are organized, doing so could almost certainly require some knowledge of the implementation details of preexisting packages. It might be easier to update preexisting packages to handle new operations if the packages were organized by type, or to handle new types if they were organized by operation. Regardless of how new entries are added to the table, if new operations are added to the system, new generic operations would need to be registered (a minor consideration, most likely).

In a system that uses a message-passing style, if new types are added to the system, they can be added as standalone procedures; nothing need change in the existing code. On the other hand, if new operations are added to the system, every data-object would need to be updated to handle the new operations.

In a system in which new types are often added (and new operations are infrequently or never added) message-passing is a clear winner. In the opposite situation, when new operations are often added but the types are essentially constant, generic operations with explicit dispatch are most appropriate.

In a situation in which either new types or new operations (or both) could be added at any time, the data-directed style allows for potentially simple additions. Even if the packages are organized by type, there is nothing preventing the addition of a package to handle a new operation. There is a compromise, however, in maintainability since such a fluid organization pattern would make it increasingly hard to predict or remember which packages contain which procedures. There is also potentially a penalty in performance as packages added later in the system's lifecycle may call out to generics which call lookups which return procedures which call out to more generics and so on.

As an example of this last point, let's say we add a `complex-conjugate` package to the complex number system descibed in the previous section. Since this package would not have access to the internal procedures of either the `rectangular` or `polar` packages, once lookup has retrieved the procedure to compute the complex-conjugate (however it is implemented), in order to get access to the selectors it needs, it would need to call out to the generic selectors, which would then call lookup a second time, returning the appropriate procedure to generate the values complex-conjugate needs to operate on.

If a subsequently added operation needed the complex-conjugate of a complex number as part of its computation, that would mean three lookups, and the chain could potentially grow indefinitely.