New clients with old servers:

A pattern language for client/server frameworks

Kirk Wolf[1] and Chamond Liu[2]

Introduction

While developing object-oriented frameworks for Smalltalk workstations communicating with legacy host business systems, we find that several patterns recur in mutually supportive arrangements. Many of the low-level patterns have already been described by other authors [Coad 1992] [Gamma et al. 1994]; our purpose is to describe how they fit and cooperate with higher-level patterns to produce complete frameworks. Our attempt to form such a description is motivated by Christopher Alexander's pattern language [Alexander 1977] [Alexander 1979], with which whole metropolitan regions are designed or improved by applying successively more detailed patterns, culminating with patterns such as "half-inch trim". While Alexander's exposition proceeds top-down, ours begins with low-level patterns and works up to a whole framework; this approach shows how higher-level patterns build directly on lower-level ones.

Most of the matters we discuss are of particular relevance to framework *developers* — the individuals who must solve the infrastructural problems of converting host data to and from client objects, coordinating persistent with non-persistent versions of objects, using communications paths economically, and windowing. Framework *users*, while having to be conversant with the framework's externals — available services, abstract classes from which to inherit, and methods that must be overridden — must concentrate largely on business specific matters. For example, in an order processing application, they must focus on the data and behavior of an order: its information content, the states through which it transitions, plus the behavior that supports the business rules of the legacy system. As a rough indication of relative scale, the project we are now working on has about 80 framework classes and 400 domain classes. We distinguish framework developers (who program the former) from users (who program the latter) and aim this paper mostly at developers. Nevertheless, we believe framework developers can only succeed if they have experience as framework users, too.

Conventional descriptions of frameworks emphasize classes and their collaborations; they connote a single, fixed architecture, providing no guidance on how one might construct alternative frameworks to solve the same problem. Raising the emphasis from detailed descriptions of classes to descriptions of *patterns* suggests a more flexible approach to framework construction. Recent work by [Eggenschwiler and Gamma 1992] and [Johnson 1992] describes specific frameworks with the help of patterns, the latter with the express purpose of assisting the framework's users. We use patterns as a family of principles for framework developers; their interrelationships form an overall scheme — or pattern language — for building client/server frameworks to support business applications with traditional mainframe systems.

Although we focus on interrelationships between patterns, we cannot overlook the classes that participate within each pattern. Unlike the elements (nouns) of Alexander's patterns, the elements (classes) of object-oriented patterns are generally not part of the vernacular. Thus, software pattern descriptions must

---

[1] Wolf Associates Inc., 305 Willowpointe Drive, St. Charles, MO 63304, 72073.3571@compuserve.com

[2] Clarity Computing, 4 Eldorado Drive, Poughkeepsie, NY 12603, chamond@aol.com

usually elaborate on participating classes. To attempt otherwise would be like trying to describe Alexander's "light on two sides of a room" to someone who has never heard of a room.

A schematic summarizing our results appears in the concluding section. We invite readers who prefer a top-down exposition to begin there and work backwards.

Preamble: a note on reification

Reification is a pervasive and beneficial theme of object-oriented design. It is the act of forming objects from entities that seem very unlike objects. Reification, then, "objectifies" that which may not at first appear to be an object. (This is analogous, in linguistics, to turning parts of speech that are not nouns into nouns.) The classic example of reification is the *command* pattern: designers reify user interface commands into objects in their own right so that they may be stored, retrieved, executed, and most significantly, unexecuted or undone [Schmucker 1986] [Meyer 1988] [Gamma et al. 1994]. Other candidates for reification are services like searching (see below), constraints, exceptions (see below), strategies [Gamma et al. 1994], and many-to-many relationships between objects.

Reification is not so much a pattern as a broad class of patterns; specific patterns like command, search, ... may be thought of as instances of reification. Despite this lack of specificity, reification is a useful approach to any problem that manifests unanticipated complications; reification often clarifies a design.

Exception

Problem Abnormal or unusual conditions often occur far in the execution stack from the point at which a program should most naturally respond to the condition. The customary solution is to return information needed to handle the condition back through the calling chain to the handling site. Alternatively, global variables are sometimes used to exchange such information. The first solution results in considerable code clutter, especially when the information is complex, and the second exposes too much to unconcerned portions of the system.

Solution Defining these conditions as specializations of class Exception categories them into hierarchies of conditions. Further, the *exception* pattern decouples detection of conditions from their handling and widens the bandwidth from lower level services to their upper level clients without the clutter that the conventional call-return protocol causes. This pattern is built into C++ and some Smalltalk implementations, and is well-documented elsewhere. See [Hinkle and Johnson 1992 & 1993] and [Dony 1990].

Solitaire

Problem Some objects occur at most once in an application. This phenomenon should appear explicitly in a framework, for the sake of both developers and users.

Solution A class for which at most one instance may occur is called a *Solitaire* (or *Singleton*) [Gamma et al. 1994]. Two roles for solitaires are prominent here: *identifiers*, that are responsible for recognizing whether two objects actually have the same identity; and *keepers*, that are responsible for maintaining a cache of valuable resources. Identifiers mediate integrity and keepers improve performance.

Objects from Records

Problem Exchanging data between object-oriented clients and legacy host database systems is complicated by the "impedance mismatch" between rich, deeply nested objects and flat relational rows. Records (strings of bytes) are inherent in legacy host systems, and therefore must participate in any architecture for exchange. The client must transform these records to and from full-fledged Business Objects — the objects that are at the core of the problem domain, objects such as Order in an order processing system. The Record's distinct fields must be transformed to and from the BusinessObject's instance data.

Solution *Objects from Records* provides for flexible, efficient exchange of data between a client and host server. This pattern moves the data access responsibility from the BusinessObject to a Record object. Class Record is responsible for getting, setting, and committing field objects to and from a string of bytes. Each instance of a concrete subclass (like Order) of BusinessObject has a corresponding instance of Record. We like to think of each BusinessObject as wrapping behavior around a Record, which represents the object's instance data.

Records collaborate with RecordMaps; a RecordMap defines the layout for all Records of the same kind. This layout is in turn specified by FieldMaps; a FieldMap defines the structure of a type of record field and is responsible for converting objects to and from a string of bytes. Thus a RecordMap consists of an array of FieldMaps and an array of corresponding positions within the record structure. Concrete subclasses of FieldMap handle conversions for dates, timestamps, numeric types (generally built with the *Pas de deux* pattern [Liu 1993]), character strings (by codepage), or more domain-specific objects such as PhoneNumber, Address, etc. FieldMaps raise *exceptions* when encountering conversion errors.

Since all Records for a single kind of Business Object share a single RecordMap, the storage overhead for Records themselves is small. All RecordMaps are instances of a single RecordMap class, so there is no need to proliferate subclasses of RecordMap or Record. RecordMap instances may be generated directly from Cobol record definitions or host data dictionaries and are kept in a RecordMapRegistry (a *solitaire).*

For high-performance access to field objects, Records cache the first access to a field so that subsequent accesses bypass conversion. Changes to Record fields only change the cache until the changes are committed.

Request

Problem Client/server frameworks use different communications interfaces — APPC, TCP/IP sockets, Remote Procedure Calls, ... — to exchange information with host applications. The API models and requirements for maintaining connection states vary between these interfaces. These variations should be invisible to client applications.

Solution The *request* pattern removes dependencies on the particular communications interface. A client instantiates a Request object which atomically exchanges a collection of input records for a set of output records. A Request thus encapsulates a unit of work, or transaction, which is executed by the host/server. Requests mediate access to the protocol of a particular communications service, maintain connections to the communications layer, mediate user authentication with the host, and raise *exceptions* to their clients.

Note: Because our units-of-work are limited to the duration of a Request, we use an optimistic transaction strategy to detect and recover from integrity errors from concurrent updates to data. That is,

each updatable record contains a timestamp and the host rejects update requests unless this timestamp matches the timestamp in the corresponding database row.

Materialization

Problem A fundamental property of an object-oriented client/server framework is to render objects at the client from the server, whether the server supports objects or, as happens with the conventional host transaction-based systems considered here, not. Moreover, application code on the client should be unaware of how or when this rendering occurs.

Solution The *materialization* pattern uses the *request* and *objects from records* patterns to transform host data to and from objects. To make this transformation invisible to application code further requires the *proxy* [Gamma et al. 1994] or *ghost* patterns. A proxy is an object inserted between an object and its clients; to clients, the proxy appears to be the underlying object. The proxy establishes an indirection that can mask the extent to which the underlying object has been materialized. A ghost has none of the behavior of the underlying object. Messages intended for the underlying object are "not understood" by the ghost; the ghost intercepts the first such message, whereupon it materializes the object and then "becomes" it, finally redispatching the message. In other words, a ghost metamorphoses into the materialized object upon first use. By contrast, a proxy embeds the materialized object within it upon first use and survives as the only external interface to the object. Ghosts are impractical to implement in C++.

Both these forms are "on-demand" or "lazy" — they defer materialization until absolutely needed. Proxies may alternatively be "eager", which means that they embed the materialized object prior to first use. When materializing complex networks of business objects, eager and lazy materialization may be mixed to balance fast initial materialization against fast subsequent use, while remaining transparent to the rest of the system. Proxies also go by the name "handle-body" [Coplien 1992]. The handle is the proxy itself, and the body is the underlying object.

In situations where communications bandwidth must be conserved, especially when retrieving lists of objects instead of single objects, a framework may partially materialize an object, producing a *SkinnyObject* with a subset of the complete object's protocol. Should a skinny object receive a message intended for the complete object, the skinny object behaves as a ghost, transparently morphing to its full form.

Finalization

Problem Applications often need an opportunity to take some explicit action at the moment of an object's demise, that is, at the moment that the last reference to an object is removed. The classical example occurs when the last reference to a handle of an operating system resource is removed; the application needs the opportunity to free the resource at that time. As we will see in the next pattern, there are other occasions when the application or framework must be informed when an object is no longer used by any other objects.

Solution The *finalization* pattern provides the facility for taking action at an object's demise. Smalltalk-80 provides direct support for finalization through class WeakArray: a WeakArray references its elements just as an ordinary Array would, except that it doesn't impede their garbage collection. When the last reference to an element disappears, i.e. when the element is garbage collected, the WeakArray is informed, giving it an opportunity to act. In other Smalltalk implementations or C++, finalization may be implemented by using techniques that resemble reference counting. (Our own

implementation uses "dependency": finalization of an object occurs when it loses its last "dependent". The full treatment of dependency appears later, in its more customary user interface context.)

Identity management

Problem Because objects *materialize* from identity-less records, the application is prone to materializing multiple copies of the same object. It is therefore conceivable for multiple order objects to represent the same order, with the same order number. Such multiplicity needlessly clutters the workplace, distracts the user, and introduces the risk of inconsistent versions of the object.

Solution To prevent multiplicity of business objects, we use the *identity management* pattern. The responsibility resides in an abstract class we call File. A File is an *identifier solitaire* that manages the identity of business objects to ensure that if, for instance, two lookups produce the same business object, the second lookup references the existing object instead of producing another copy of it. We use a separate File for each kind of business object. Each File maintains a set or dictionary of the objects whose identity it manages. Business objects register themselves with their File when they materialize; when they are no longer referenced, they must be unregistered. In other words, the File relies on *finalization* to know when to unregister one of its objects.

Files also provide users with a conceptual representation of a repository for business objects; users imagine that orders are housed in an order file, or order repository. The fact that the file also implements identity management for its business objects is transparent to users.

Mega-scrolling

Problem A request may return a collection of objects that is too large to *materialize* practically on the client.

Solution *Mega-scrolling* (which we also call *Virtual lists*) limits the size of the returned collection. This limited subcollection may satisfy the user, but if not, mega-scrolling issues another request to retrieve an adjoining subcollection. In practice, the business objects materialized by mega-scrolling are often skinny objects that contain only enough state to render summary views of the objects.

Mega-scrolling may be designed so that the user explicitly requests the next subcollection, or so that the user interface anticipates the need for the next subcollection on behalf of the user. Either way, each mega-scroll requires the client and host to exchange context information that defines the next sub-collection. Our current implementation provides sufficient information to drive a brand new host transaction for each succeeding mega-scroll; alternatively, succeeding mega-scrolls could refer to a persistent host resource such as an SQL cursor.

Searching for business objects

Problem The act of locating an object of interest is an elemental activity, so mundane that it risks being taken for granted. Screening a repository by certain criteria, or merely extracting from it a single element identified by a unique key, are both examples of this act. Note that searching is something that may need to be retried, only somewhat differently, because the original results aren't what the user hoped for, or because she would like to filter or reorder the results.

Solution *Searching for business objects* can be reified by defining an abstract class called Search. Search objects define the criteria that govern a search or lookup. Since they can persist after a search

operation has completed, they are available for modification and reuse, and they are a basis for subsequent filtration, reordering, or *mega-scrolling*. The search pattern unifies lookup and searching. All searches for business objects, no matter how trivial, begin by constructing a Search object defining criteria for the search. A File receives a message to search, with the Search object as a parameter. The result is a List (or a single object if the intent of the search is really just a lookup). Note that the result may be null, if the search fails to locate anything. Searching relies on *identity management* to avoid rematerializing an object that is already present at the client.

Dependency (model-view, broadcast, observer, MVC)

Problem Coupling business objects to the views that render and control them is bad engineering and ultimately also bad for users. Such coupling complicates code and design maintenance, and also inevitably clouds the underlying conceptual model: the behavior of the model gets inextricably tangled with the behavior of the user interface, for both developers and users.

Solution The loose coupling between model objects and their views, pioneered in the late 1970's at Xerox PARC as the Model-View-Controller (*MVC*), has been one of the most durable patterns in the history of object-oriented programming. Nowadays the original MVC triad more often coalesces into an MV dyad. Nevertheless, the essential benefit remains: the model objects are oblivious to the specific expectations of their views, thereby uncoupling the task of developing model objects from the task of developing the user interface. A full exposition of MVC appears in [Collins 1995]. The pattern goes by various names: *observer* [Gamma et al. 1994], which implies the obliviousness of the "observed" model object, and *broadcast* [Coad 1992], which refers to the mechanism by which models inform views that changes have occurred. Because we also apply this pattern to non-UI situations (see the next section), we often refer to it as *dependency*. As suggested earlier, dependency can also be exploited for reference counting purposes.

Creating and updating business objects

Problem Business objects must come into existence and change over time. These actions usually initiate at a client and must eventually reflect onto the host. They become more complicated when objects depend on other objects. For example, an order is likely to depend on its customer, so that changes to the latter must affect the former.

Solution A File uses *materialization* (more precisely, "de-materialization") to apply changes to its business objects. Updates must propagate to dependent objects so that the dependents have an opportunity to update as well. We propagate updates with the *dependency* pattern; the objects in question are not views, but they behave exactly as views do. That is, they are dependents of another object, just as views are, and they receive broadcasts of updates, just as views do. Dependency is the inverse of the *composite* pattern [Gamma et al. 1994]; either pattern solves the problem of cascading updates to other objects. However, we prefer dependency because it supports two-way communication between the objects: the dependent can gather whatever information it chooses to from the broadcaster.

Broadly interpreted, updating includes practically any public behavior of a business object. An order, for example, may experience several state transitions (enter, bill, suspend, validate, cancel, ...) during its life cycle. The transitions and states are specific to the business and may entail extensive analysis and design, well beyond the concerns of a framework's mechanisms. (The pattern *objects from states* [Beck 1994] can help organize state transition problems like this.)

Factory method

Problem When creating a view object for a model, it is better for client code to assume as little as possible. The client generally shouldn't even know the class of the view.

Solution A *factory method* [Gamma et al. 1994] creates an object to be associated with an existing object, like a window or view object to be associated with a model object. Different classes of objects implement the factory method polymorphically, so that for instance, each different class of model object may create a different kind of associated view. A factory method frees its client of the need to know the kind of object to be associated with its receiver.

Window keeping

Problem Building complex windows from scratch in a GUI can be expensive. For example, moderately complex windows can take many seconds to appear, even on high performance desktop systems. Users cannot tolerate such performance characteristics. Another usability problem occurs if, when the application identifies a business object on which a window already exists, the application opens a new, redundant view on the object instead of resurfacing the existing view.

Solution Class WindowKeeper is a *solitaire* that keeps track of the application's windows. When the user dismisses a window, the window keeper simulates closing it, then caches it. Should an object subsequently require a window of this class, the window keeper disinters the cached window and opens it on the object. This all but eliminates the penalty for building a window for the object. Moreover, since the window keeper tracks all of the application's windows, it can know when to resurface an existing window to avoid redundant windows on the same object.

Viewing

Problem Besides finding and creating or updating business objects, the only remaining function of an application is to provide visibility to them. Most projects use the machinery of a platform-specific UI framework or toolkit to build this function. Regardless of the quality of the framework or toolkit, however, we frequently see several design shortcomings of applications built around them: burdening client code with knowledge of the view(s) that apply to a particular kind of model object; complicating design (and ultimately clouding the users' conceptual model for the application) by intertwining models and views, slow window creation, and proliferation of redundant windows on the desktop.

Solution *Viewing* generally uses the rich variety of function provided by some UI toolkit. Beyond this function, *viewing* also requires the *factory method*, *dependency*, and *window keeping* patterns to overcome the problems outlined above. These patterns together help maintain clear separation in user-interface design and provide practical desktop management.

A client/server framework

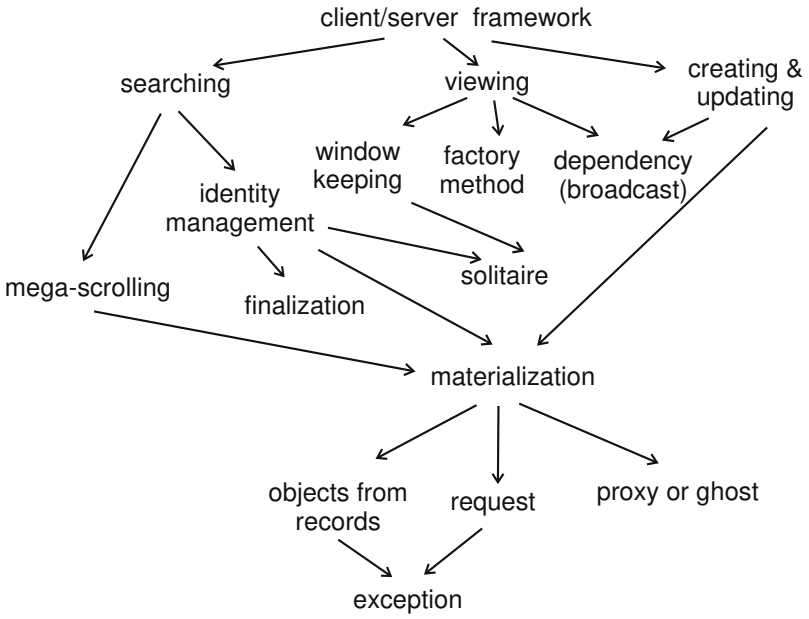
Problem An object-oriented client/server framework for communicating with a legacy host system, reduced to its most elemental terms, must let its user locate objects of interest, examine them, and change (or create) them.

Solution The *client/server framework* pattern, then, is an aggregation of the *searching*, *viewing*, and *updating/creating* patterns. A successful framework depends on the quality with which these three

patterns are implemented, which in turn depend recursively on the quality with which their constituent patterns are implemented.

Conclusion

In the spirit of Christopher Alexander's pattern language for designing a human community, we have outlined the patterns and their interrelationships for designing a coherent client/server framework. The following figure summarizes the key relationships. An arrow indicates that a higher-level pattern relies on a lower-level pattern.



This schematic amounts to an informal pattern language that anchors our thinking about framework designs and limits the space of potential framework development paths to a subspace of possibilities that can reasonably be grappled with. We hope that this subspace — or language — proves helpful to other developers, but we also would like to know what other framework languages for client/server computing might look like.

Acknowledgments

We are grateful to the referees and workshop participants of PLoP'94 for their many constructive suggestions to earlier iterations of this paper.

References

[Alexander 1977] Christopher Alexander, Sara Ishikawa, Murray Silverstein, with Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel. *A Pattern Language.* New York, NY: Oxford University Press, 1977.

[Alexander 1979] Christopher Alexander. *The Timeless Way of Building.* New York, NY: Oxford University Press, 1979.

[Beck 1994] Kent Beck. "Death to case statements (part 2)," *The Smalltalk Report* v3 n4 (January 1994), pp. 8-9.

[Coad 1992] Peter Coad. "Object-oriented patterns," *Communications of the ACM* v35 n9 (September 1992), pp. 152-159.

[Collins 1995] Dave Collins. *Designing Object-Oriented User Interfaces.* Redwood City, CA: Benjamin/Cummings, 1995.

[Coplien 1992] James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Reading, MA: Addison-Wesley, 1992.

[Dony 1990] Christophe Dony. "Exception Handling and Object-Oriented Programming: Towards a Synthesis," *ACM SIGPLAN Notices (OOPSLA ECOOP '90 Proceedings)* v25 n10 (October 1990), pp. 322-330.

[Gamma et al. 1994] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture.* Reading, MA: Addison-Wesley, 1994.

[Hinkle and Johnson 1992 & 93] Bob Hinkle, Ralph E. Johnson . "Taking Exception to Smalltalk, Parts 1 & 2," *The Smalltalk Report* v2 n3 & v2 n4 (November/December 1992 & January 1993).

[Johnson 1992] Ralph E. Johnson. "Documenting Frameworks using Patterns," *ACM SIGPLAN Notices (OOPSLA'92 Conference Proceedings)* v27 n10 (October 1992), pp. 63-76.

[Liu 1993] Chamond Liu. *Duets, pas de deux, and double dispatch.* Position paper, 1993 OOPSLA Patterns Workshop.

[Meyer 1988] Bertrand Meyer. *Object-Oriented Software Construction.* Hertfordshire, UK: Prentice Hall International, 1988.

[Schmucker 1986] Kurt J. Schmucker. *Object-Oriented Programming for the Macintosh.* Hasbrouck Heights, NJ: Hayden, 1986