Packet Handler[†]

An Object-Oriented Framework for Satellite Telemetry Processing

David Van Camp
Software Architects
P.O. Box 3104, Collegedale, TN 37315
70323.3510@compuserve.com

Wednesday, February 19, 1997

Submitted to the Using Patterns Conference, March 1997, at the Mohonk Mountain House in New Paltz, NY, USA.

Introduction

The Packet Handler framework implements a reusable abstract architectural design [Johnson91] for use in the development of distributed data processing systems which follow the general model: 1) Read a formatted block of data, 2) Process the data in some well-defined manner, 3) Write some optional number of data blocks, then, 4) Go do it again. Packet Handler is currently under development for use in the creation of distributed real-time telemetry processing modules in the *NASA-Goddard Hubble Space Telescope Vision2000 Control Center System.* Vision2000 is an ambitious effort to revamp the ground-based Hubble support systems to reduce maintenance costs and provide a flexible, stable platform for improved processing capabilities and future enhancements.

Packet Handler meets the goals of Vision2000 by providing a highly standardized and consistent *pipes and filters* [GS94] architectural foundation which speeds and simplifies the development of many of the individual executable modules which collectively form Vision2000. Individual systems are built as *specializations* of the Packet Handler framework by inheriting from specific classes in the framework class library and extending the functionality to meet the specialized demands the particular system requires. Packet Handler provides many *hotspots* of flexibility [Pree95] where developers may extend or modify the default processing defined by the architecture. Object-oriented design patterns [GOF95] were extensively applied to various aspects of the design to insure that Packet Handler meets the highest standards of quality possible as well as to simplify understanding of the resulting design [Beck96].

System Configuration

The top-level object model (Figure 1, next page), shows the primary classes which are used to create a specific packet handling system. Each packet handler system must run in separate process or thread. Multiple processes or threads may be chained together for staged packet handling, each stage performing a specialized part of the work and then passing the results onto the next stage. Certain stages may be designated as producers or consumers of packets. For a producer, packets would typically be sent to other modules as requests for processing, possibly containing information about where to send the resulting information. For consumers, handling of received packets would result in data written to a file or database, or perhaps, information may be displayed in a window.

The Packet Handler framework is written in C++ for Silicon Graphics (SGi) and Hewlet-Packard (HP) Unix. A variety of external input and output interprocess communications (IPC) technologies are supported. Currently these include: a proprietary commercial off-the-shelf (COTS) hardware system called "Veda", the standard Sockets library, shared memory queues and specially formatted file streams. Future plans include adding support for Oracle RDBMS and Rouge-Wave's Net.h++. Other sources such as Named-Pipes, Internet FTP and other protocols, can be easily added if and when they are needed.

[†] *This work was performed under contract NAS5-50000 for the NASA-Goddard Hubble Space Telescope Vision2000 Control Center System in Lanham, Maryland.* `http://ccs.gsfc.nasa.gov/`

The Architecture of Packet Handler

A minimal packet handler-derived system consists of a set of the following objects (Figure 1):

- A packet handler which forms the core of the system. This may be a direct instance of G_PacketHandler or a subclass of it.
- An input buffer which retrieves blocks of data from an external input source and formats them into packet objects. The input buffer object may be a direct instance of G_PacketInputBuffer or a subclass of it.
- Some number of output buffers which send packets to specific output destination(s) as structured blocks of data. Each output buffer object may be a direct instance of G_PacketOutputBuffer or a subclass of it.
- Some number of packet objects, each of which encapsulates a particular kind of data block which the system can receive or process. One specific subclass of G_Packet is required for each kind of data block the system needs to support.

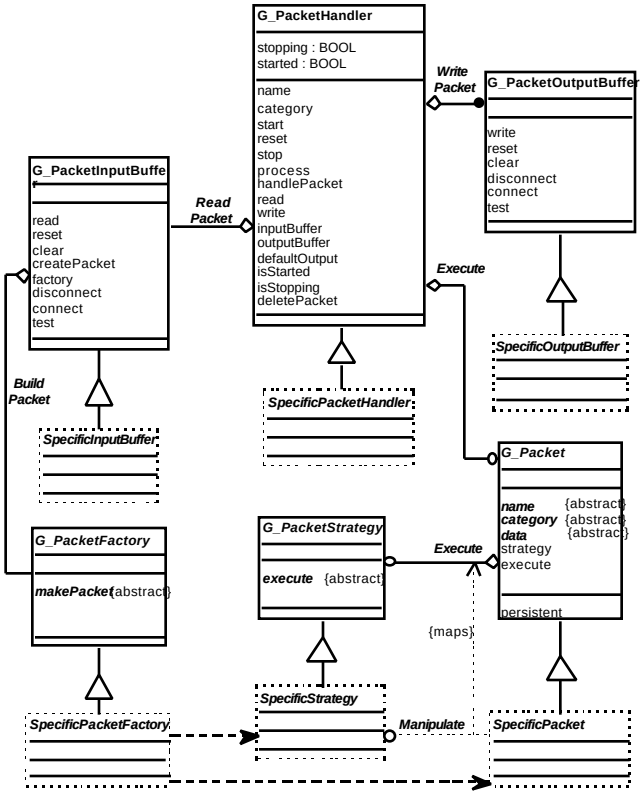

Figure 1 - High-Level Architecture of Packet Handler

- Some optional number of packet handling strategies, each of which encapsulates the processing required to handle a particular packet or category of packets within the specific system. Each packet strategy must be a subclass of G_PacketStrategy.
- One or more packet factories, each of which takes a block of data read by the input buffer and creates, and initializes one or more of the specific packet objects the system supports. Each factory object must be a subclass of G_PacketFactory.
- Initialization code to create and initialize the packet handler, input buffer, output buffer(s) and packet factory(s) used by the system, and then call the start method in the packet handler. The start method will return only when system processing has completed. Most of this initialization is typically performed in the constructor of a specific packet handler subclass.

Design Patterns in the Packet Handler Framework

The Packet Handler framework architecture was designed using a technique known as "pattern-generated architecture" [BJ94]. Pattern-generated architecture is a fairly new technique which employs design patterns to speed design while increasing quality, flexibility and consistency. Additionally, the purpose, benefits and reasons of a specific design generated from patterns are typically easier to explain and understand, *if the reader understands the design patterns used.* [VanCamp96]

Consequently, the reader of this document is expected to have a good understanding of design patterns and the specific patterns employed in this design to fully comprehend this document. The remainder of this section presents a short introduction to design patterns and brief descriptions of the particular patterns used herein. This should provide enough background for the reader to follow the design discussion, however, for full comprehension the reader will be required to obtain a copy of the design pattern catalog and study the particular patterns cited below.

Overview of Design Patterns

The pattern catalog which contains the patterns used in this design is: *Design Patterns: Elements of Reusable Object-Oriented Software*, written by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, published by Addison-Wesley Publishing Company, New York, NY, 1995, ISBN 0-201-63361-2. The authors of this book, having reached a level of acclaim seldom seen in the software design field, have been affectionately dubbed the *Gang-of-Four* (GOF). Consequently, this book is abbreviated: [GOF95]. [GOF95] is highly recommended reading for anyone interested in improving their object-oriented design skills.

Design patterns were proposed as a mechanism for expressing design structures by the GOF in a landmark paper delivered at the *European Conference for Object-Oriented Programming* in 1993 [GOF93], which included the following description:

> *"Design patterns identify, name, and abstract common themes in object-oriented design. They preserve design information by capturing the intent behind a design. They identify classes, instances, their roles, collaborations, and the distribution of responsibilities. Design patterns have many uses in the object-oriented development process:*
>
> - *"Design patterns provide a common vocabulary for designers to communicate, document, and explore design alternatives. They reduce system complexity by naming and defining abstractions that are above classes and instances. A good set of design patterns effectively raises the level at which one programs.*
> - *"Design patterns constitute a reusable base of experience for building reusable software. They distill and provide a means to reuse the design knowledge gained by experienced practitioners. Design patterns act as building blocks for constructing more complex designs; they can be considered micro-architectures that contribute to overall system architecture.*
> - *"Design patterns help reduce the learning time for a class library. Once a library consumer has learned the design patterns in one library, he can reuse this experience when learning a new class library. Design patterns help a novice perform more like an expert.*
> - *"Design patterns provide a target for the reorganization or refactoring of class hierarchies [23]. Moreover, by using design patterns early in the lifecycle, one can avert refactoring at later stages of design."*

Many authors have contributed to the ever growing volume of literature on design patterns. *The Patterns Home Page*, an Internet web site maintained by the University of Illinois at Urbana-Champagne contains a wealth of information on patterns. Published and unpublished papers are available including additional design patterns, discussions on their use and history, experience reports, case studies of pattern-generated architectures and more. You can visit the *Patterns Home Page* using any web browser at:

http://st-www.cs.uiuc.edu/users/patterns/patterns.html

Patterns Used in this Design

The following is a list of the key design patterns employed in the design of the Packet Handler framework. For each pattern below, the name, page number and intent is provided from [GOF95]. For detailed information on the these patterns, see [GOF95].

Builder (97)

Separate the construction of a complex object from its representation so that the same construction process can create different representations

Factory Method (107)

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Adapter (139)

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Composite (163)

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Facade (185)

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Command (233)

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Strategy (315)

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

Template Method (325)

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Packet Handler System Operation

The specific operating structure and behavior of a system developed using the Packet Handler framework is defined by the specific classes declared, created and initialized by a particular system. Each system must create the specific packet handler, input buffer, output buffer(s), factory(s) and some other helper classes that the system will use to operate (see Figure 2). Packet Handler provides a number of default classes which may be used directly or a specific implementation may define specialized classes by inheriting from the framework classes.

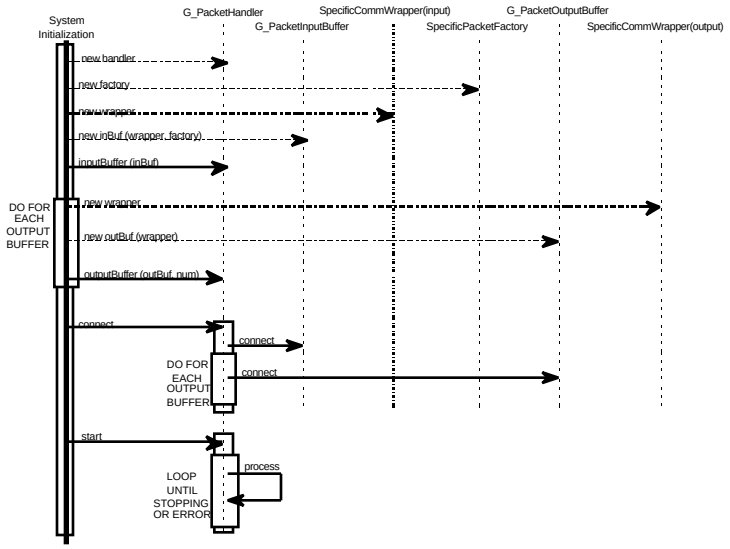

Figure 2 - Typical Packet Handler Initialization Event Trace

Object construction and initialization is typically performed in the constructor of a specific packet handler. The specific packet handler will also provide various methods required by the system to perform specialized processing.

Once the system is fully initialized, the *connect* method in G_PacketHandler is called to enable the input and output buffers for communications. Packet handling begins when the *start* method in G_PacketHandler is invoked. This starts the central loop which simply calls the *process* method until the system is halted (by calling the *stop* method) or an unrecoverable error occurs.

Packet Processing

The overall philosophy of packet handler based systems can be summed up by this simple metaphor: *The packet handler is the system's heart. The specific packet strategies form the brains of the system. Packets are the blood which flows through the system. All other classes are the organs which provide a decentralized support network for the handler and the strategies.*

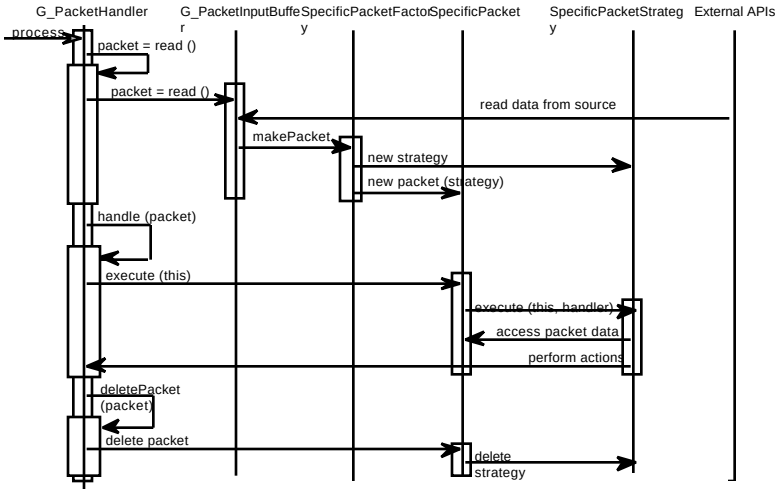

Figure 3 - Default Packet Processing Event Trace

The specifics of packet processing and the classes used to implement this process are detailed in later sections of this document. This section provides a general overview of the default process as provided by G_PacketHandler.

The *process* method in G_PacketHandler is a *Template Method [GOF95]* which simply calls three *hook* methods to read a packet, handle the packet and then delete the packet (see Figure 3)

Generally, specific packet handler subclasses will not need to override or extend the default packet processing hook methods unless specialized error handling or a fundamental change in the way packets are read, handled or deleted is required. A possible common exception may be overriding the *deletePacket* method to implement packet object caching in systems having severe real-time processing constraints.

The *read* method in G_PacketHandler simply calls the *read* method in G_PacketInputBuffer and returns its result. G_PacketInputBuffer simply retrieves a block of data from the input source using a communications wrapper class (described later) and then passes that data to the specific packet factory defined for the system by calling the *makePacket* method defined in G_PacketFactory and overridden in the subclass. The factory inspects the data and builds the proper packet object, it's associated strategy and binds them together. When control returns to G_PacketHandler's *read* method, the system has the information it needs to process *and* the algorithm needed to perform this processing.

The *handlePacket* method in G_PacketHandler simply calls the *execute* method in the just received packet object, including a reference to itself as a parameter. Since packets may be shared between systems but the handling often is not shared, packets usually do not implement any processing in the *execute* method. The default execution defined in G_Packet is to simply call the *execute* method in the packet's associated strategy, passing it the packet handler's reference and a reference to the packet itself as parameters. The strategy will use these references to access any public methods provided by the packet or the handler to perform whatever specific operations are required to process the specific packet. When processing has completed, the *execute* method will return one of three possible values: *Success (1)*, *Canceled* (0) or *Error (-1).* If *Error* is returned, the packet handler will shutdown operations. If either *Success* or *Canceled* are returned, processing continues normally. (Note: packet handler subclasses may extend the *handlePacket* method if they wish to handle canceled packets differently then successfully processed packets.)

Finally, the *deletePacket* method simply destroys the packet object. The destructor of G_Packet will then destroy the strategy object, insuring the system has no memory leaks. If packet object caching is desired, specialized packet handler subclasses must override this method.

The remaining functions in the packet handler object form a *Facade* [GOF95] which the strategies in a system may use to perform the various tasks it needs to implement. G_PacketHandler provides a small number of functions by default, including *write* (write a packet to an output buffer), *stop* (halt operations without error), etc. However, each specific subclass of G_PacketHandler will usually provide additional functions as required for its packet processing strategies. For example, one particular system requires that a sequence number be checked and validated for each data message sent. The subclass of G_PacketHandler in this system provides as *checkSeqNum* function which provides this service. If an invalid sequence number

is passed to *checkSeqNum*, the specialized packet handler will record an error and return FALSE to the calling strategy. The strategy responds by halting processing and returning *Canceled* from its execute method. Other specific systems will require alternate strategies for handling packets along with different specialized functions in their packet handler object.

Packet Reading and Writing

G_PacketHandler uses an instance of either G_PacketInputBuffer or a specialized subclass to read packets from the external IPC source. Additionally, G_PacketHandler maintains an ordered collection of zero or more instances of G_PacketOutputBuffer or specialized subclasses. These input and output classes encapsulate the work required to perform packet reading and writing and they determine the source and destination addresses.

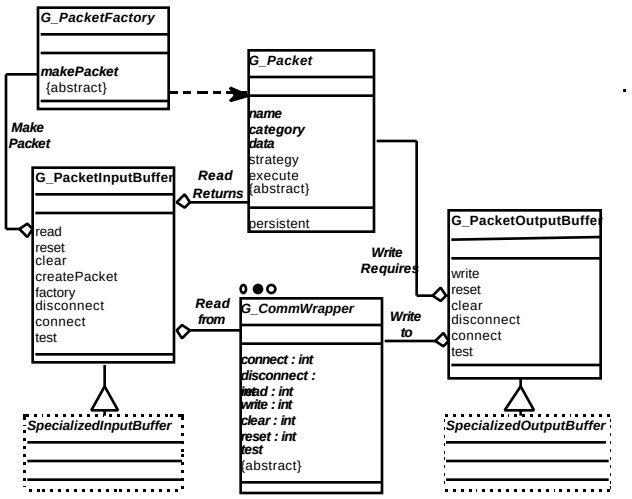

Figure 4 - Packet Input and Output Buffer Associations

The packet input and output buffers maintain no references back to the packet handler. Thus they function as independent units which may be incorporated in other systems that must either send or receive packets to or from a packet handler derived system. This is typically required by systems which act as either the initial producer or the final consumer of packets in a linked chain of packet handler processes.

On initialization, each instance of G_PacketInputBuffer and G_PacketOutputBuffer require separate references to specific communications wrapper subclasses of G_CommWrapper. These must be different objects – input and output buffers cannot share a single communications reference. The communications wrapper provides a generic interface which the input and output buffers can use to communicate with any of the specific IPC technologies that Packet Handler supports.

In addition to a communications wrapper, G_PacketInputBuffer initialization also requires a reference to a specific packet factory subclass of G_PacketFactory. The packet factory implements a *Builder* [GOF95] object which creates one of the specific packets required by a specific system by inspecting the raw data.

Prior to reading or writing packets, the *connect* methods must be called for the input and output buffers to enable the IPC communications channel. The buffers simply respond by calling the *connect* methods in their respective communications wrappers specifying *Client* mode for output buffers or *Server* mode for input buffers.

G_PacketInputBuffer's *read* method calls the *read* method in its communications wrapper to get a block of data, waiting if no data is currently available. When a data block is returned, *read* then passes that block as a parameter to the *makePacket* method in the packet factory. The packet factory inspects the data to determine what kind of packet is needed to encapsulate it and creates the proper subclass of G_Packet (and, usually, the associated subclass of G_PacketStrategy for that packet) and returns it to G_PacketInputBuffer, which in turn returns it to the caller of its *read* method.

G_PacketOutputBuffer's *write* method takes a packet as a parameter and simply calls the packet's *data* method to retrieve the data structure it encapsulates and then passes that data structure to the communications wrapper's *write* method. The communications wrapper will output the data block to whatever IPC interface it uses to send it to the destination it is connected to.

The input and output buffers also provide some simple methods for controlling communications. These simply call the associated functions in the communications wrapper. G_PacketInputBuffer also provides the *factory* method which may be used to change the packet factory used for packet creation after initialization. Systems which require specialized processing, may override or extend most of these methods as needed. Generally, however, the default processing should be sufficient for most packet handler systems unless specialized error handling is required.

Wrapping Interprocess Communications

The G_CommWrapper class hierarchy implements an object-style *Adapter [GOF95]* which provides a simple, exchangeable abstract interface which the input and output buffers use to allow the packet handler to work transparently with a variety of IPC technologies. Each of the specialized wrapper subclasses adapt a particular technology by providing a mapping from the simple target interface of G_CommWrapper to an API-level adaptee class which performs the actual operations as required.

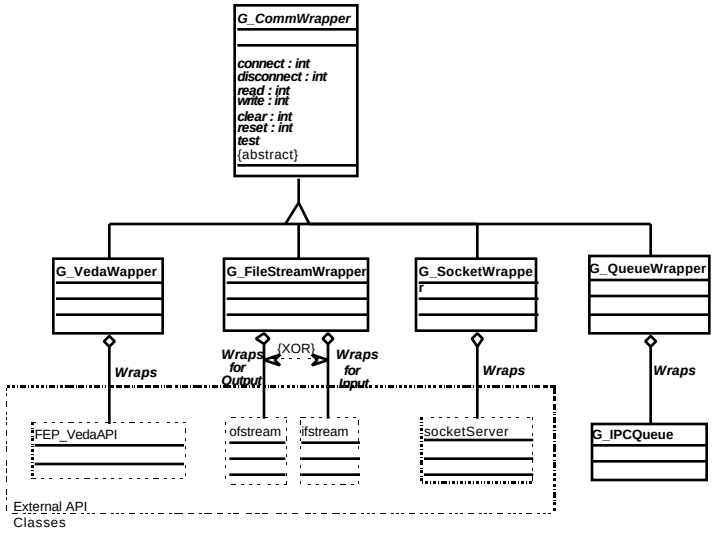

Figure 5 - The Communications Wrapper Class Hierarchy

The communications wrapper classes maintain no references to any other objects in the Packet Handler framework. Consequently, they may be freely used in any design which requires adaptable IPC. Currently, four specific wrappers are defined (more may be added in the future):

- G_VedaWrapper which provides a mapping for communications using a proprietary commercial off-the-shelf (COTS) hardware system called “Veda”;
- G_SocketWrapper which provides a mapping for using the standard Sockets API library;
- G_FileStreamWrapper which allows test data to be written to or read from a file using an internally defined file format;
- G_QueueWrapper which provides a mapping to a shared memory queue for interprocess communications within a single executable.

Each of the specific wrapper subclasses override all methods defined in G_CommWrapper and implement the code needed to perform each method in the logical manner for type of technology the wrapper is adapting. For example, the *connect* methods in G_VedaWrapper and G_SocketWrapper simply call the equivalent function(s) to connect as either a *client* (to send messages) or a *server* (to receive messages) in their respective adaptee classes and return a success or failure status. G_FileStreamWrapper, however, will perform *connect* by creating either an *ofstream* (for client mode) or *ifstream* (for server mode) object and have it create a new file for writing or open an existing file for reading, respectively. Finally, *connect* for the G_QueueWrapper class only needs to test to insure that the SharedMemoryQueue class specified in the G_QueueWrapper constructor is operational.

Similarly, the remaining methods for each specific wrapper are each defined in an equally appropriate manner for the specific technology they adapt. Each of these methods return a consistent set of return values which the caller may check to determine that the operation worked properly, or, if it failed, the general reason for the failure. (Note: methods may be added in the future to the wrapper class interface to allow callers to retrieve device-specific error information.)

Creating Packets and Strategies using Factories

The input buffer uses an object factory to create the specific packets used by a system along with the specific strategies used to handle those packets. Figure 6 illustrates the design structure provided by the packet handler framework to support this task.

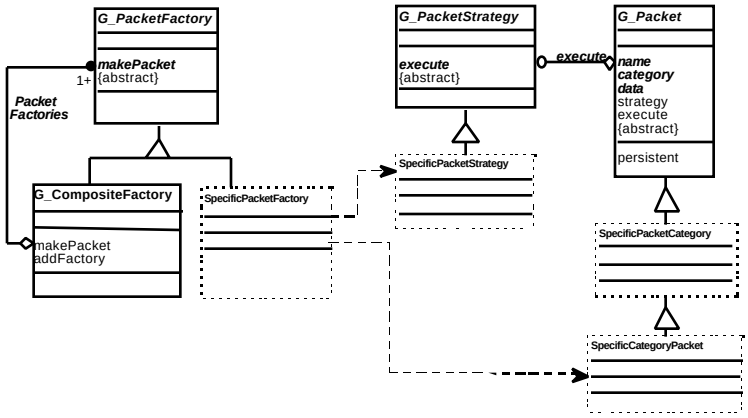

Figure 6 - Packet Factories Create Packets and Strategies

Packet factories do not receive any reference to the input buffer which calls it. Consequently, packet factories may be used for other purposes if needed. G_PacketFactory only provides a single pure virtual function declaration for *makePacket.* Each subclass of G_PacketFactory must provide the code to determine what packet to make from a block of data in *makePacket* or return NULL if it does not have the ability to make a packet from the data provided. Additionally, each packet usually requires a strategy, so the factory must determine the proper strategy to create as well.

Once *makePacket* has created the packet, it then calls the packet's *data* method, passing the data block as a parameter. The packet then initializes is internal state using that data block. The design of specific packet factories is typically based on the *Builder* [GOF95] pattern.

Sharing Packet Factories Between Systems

We can reduce code redundancy between systems by sharing factories between systems. However, if the factory must create packets which require different strategies in different systems, special techniques are required to allow factory sharing. Packet handler supports two techniques which you can use to facilitate factory sharing: composite factories and factory methods.

A composite factory is a degenerate form of the *Composite* [GOF95] pattern. Packet Handler provides the G_CompositeFactory class so that a system may include any number of factories grouped in a tree structured hierarchy, the root of which may be accessed as if it were a single factory. G_CompositeFactory allows individual factories or even groups (sub-trees) of factories to be shared while still allowing a system to add more factories. Thus packets may be grouped via these factories so that only those that are needed in each particular system are included.

The *makePacket* method in G_CompositeFactory simply calls the *makePacket* method in each of its child factories, one after the other, until one is found which returns a packet constructed from the data block. Then, the composite factory simply returns that packet to its caller, or, if all child factories returned NULL, then the composite factory returns NULL.

Composite factory solves the problem of grouping packet construction for sharing between systems, but not the problem of using different strategies for those packets in each system. This problem is solved by applying the *Factory Method* [GOF95] pattern.

Figure 7 illustrates an example of how the factory method would be applied to solve the problem of sharing a factory which makes three packets (Xpacket, Ypacket and Zpacket) that will be shared between two systems (SystemA and SystemB), but the associated strategies will not. Two strategies are provided for each packet, one for each system.

In this example, the framework user has created a specialized factory (SharedPacketFactory) which implements the code required in *makePacket* to determine the type of packet to construct and then constructs that packet. However, it does not construct the associated strategy. Instead, SharedPacketFactory declares and calls a separate *Factory Method* (*makeXStrategy, makeYStrategy* and *makeZStrategy)* to construct each individual strategy.

Next, the user created two subclasses of SharedPacketFactory, each of which implements the three methods to construct the proper strategies for a single system. SystemAPacketFactory constructs the strategies required by SystemA (X, Y and ZPacketSystem**A**Strategy), and SystemBPacketFactory constructs the strategies required for SystemB (X, Y and ZPacketSystem**B**Strategy.)

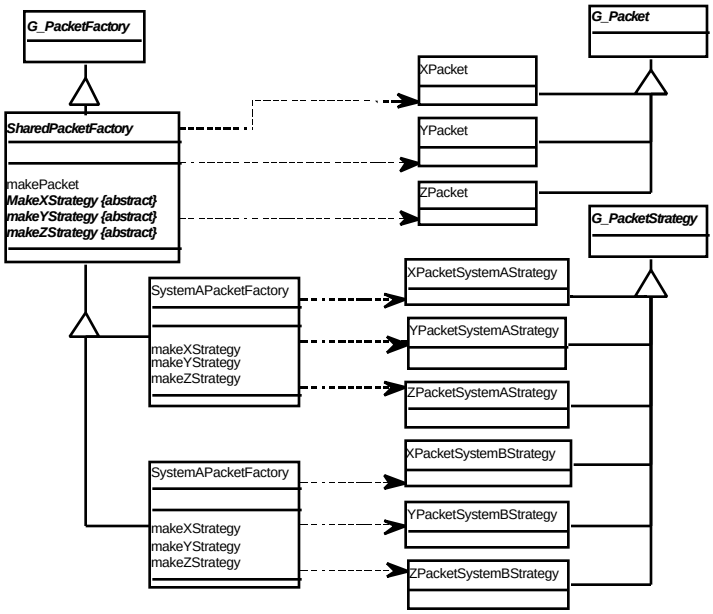

Figure 7 - Using Factory Method for Sharing Packets without Sharing Strategies

Consequently, the code implemented by the user for SharedPacketFactory's *makePacket* method would look something like this:

```
strategy = NULL;

// Check the data format then create the proper packet object and
// strategy for that format. Defer strategy creation to the
// specific subclass so that the proper strategy is created for
// which ever system is currently executing.

switch ( dataformat ) {
case XPACKET_FORMAT:
      packet = new XPacket ( data, size );
      If packet <> NULL Then {
            strategy = makeXPacketStrategy ();
      }
      break;
case YPACKET_FORMAT:
      packet = new YPacket ( data, size );
      If packet <> NULL Then {
            strategy = makeYPacketStrategy ();
      }
      break;
case ZPACKET_FORMAT:
      packet = new ZPacket ( data, size );
      If packet <> NULL Then {
            strategy = makeZPacketStrategy ();
      }
      break;
default:
      packet = NULL;
}
```

```
// If a packet was created, then check to make sure that
// a strategy was returned by the subclass. If it was, bind the
// strategy to the packet. Destroy the packet if no strategy
// was returned.

if (packet <> NULL)
      if (strategy <> NULL)
      }
            // bind the strategy to the packet
            packet.strategy (strategy)
      }
      else {
            delete packet;
            packet = NULL;

      }
}
return packet;
```

Next the make strategy methods in the SystemAPacketFactory and SystemBPacketFactory subclasses are overridden to construct and return the proper strategy object for each packet. For example, in the SystemAPacketFactory, the *makeXStrategy* method would be implemented like this:

```
G_PacketStrategy * SystemAPacketFactory::makeXStrategy ( void ) {
      return new XPacketSystemAStrategy ();
}
```

Alternately, in SystemBPacketFactory, the *makeXStrategy* would look like this:

```
G_PacketStrategy * SystemBPacketFactory::makeXStrategy ( void ) {
      return new XPacketSystemBStrategy ();
}
```

Naming and Categorizing Packets

Another aspect of sharing packets which must be considered is that certain systems may need to process each type of packet individually, but others may work on groups of packets based on a category. Packet Handler supports this paradigm.

Figure 8 shows the default three-layer inheritance packet hierarchy supported by the Packet Handler framework. G_Packet defines the high-level abstract interface which other classes in the framework use to interact with the packets in a generic manner.

Each packet class directly inherited from G_Packet must override the pure virtual *category* method and return its class name as a string, like this:

```
char * MyPacketCategoryClass::category ( void ) {
      return "MyPacketCategoryClass";
}
```

Categories allow the framework user to write code which needs to call methods defined for a specific packet category to insure that it is working with the proper specific subclass. Packet Handler-based systems require the *category* method since the C++ compilers currently used do not yet support *run-time type information* (RTTI). Consequently, code which must call methods defined in a specific packet category subclass use the *category* method to insure that they have are referencing the expected subclass prior to down-casting, like this:

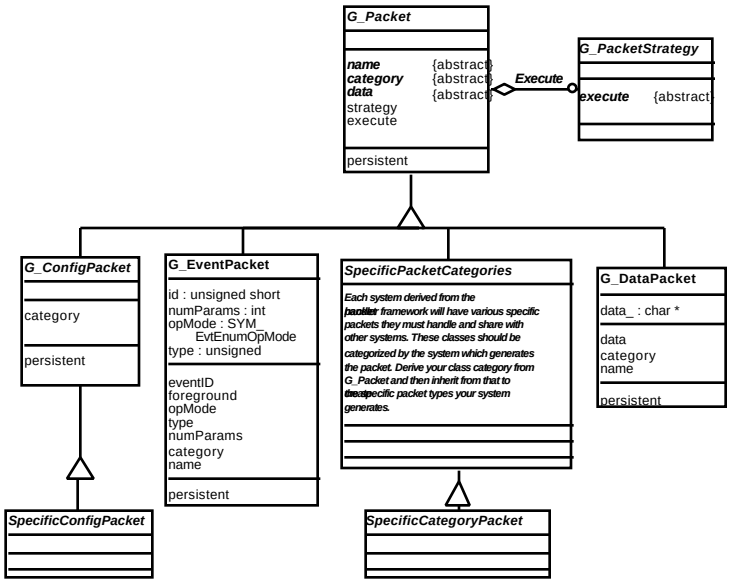

Figure 8 - Packets and Packet Categories

```
if (strcmp
(packet-
>category(), "MyPacketCategoryClass") == 0) {
      MyPacketCategoryClass * category;
      category = (MyPacketCategoryClass *) packet;
      category->someSpecificMethod ();
} else {
      // ... error! ...
}
```

Similarly, G_Packet also defines the pure virtual *name* method. Specific packet classes, which form the leaf nodes of the packet inheritance hierarchy, must override the *name* method in the same manner as shown for the *category* method, above. Code written to process a specific packet subclass should use the *name* method to insure that it has a proper reference prior to down-casting.

The Default Packet Categories

The default packet hierarchy in Packet Handler defines three packet categories: G_ConfigPacket, G_EventPacket and G_DataPacket. Each of these overrides the *category* method to return its particular class name, as described above. Additionally, G_EventPacket and G_DataPacket double as specific packet classes. Consequently, these two classes also override the *name* method.

G_DataPacket is designed as a generic encapsulation of an unspecified block of data. This packet class is used in systems which do need to perform any specialized processing on a specific data structure, but, instead simply passes the data on to other packet handling modules. A specific example of this is described in the discussion of the *Stage 1 Pass-Through Handler* later in this document.

G_EventPacket is used by packet handler systems which need to format special "event" messages which are later processed by the *System Monitor* -- a specialized module in the Vision2000 architecture which is not derived from the packet handler framework. Further discussion of the system monitor module is outside the scope of this document. See the appropriate Vision2000 reference documents for more specific information on event registration and handling.

G_ConfigPacket is the only true category class currently provided by the Packet Handler framework. This class provides a convenient grouping mechanism for conceptually dividing packets into commands which may alter the system's state and packets of data that are processed. The reader should note, though, that

Packet Handler, by default, does not process configuration requests any differently than any other kind of packet. However, Packet Handler does provide a small number of standard configuration class packets and an associated set of strategies which each system will normally include. The configuration packets currently defined are:

- G_ConfigShutdown: Stop all packet handling activities. Control returns to the caller of the *start* method in G_PacketHandler with a 0 return value, indicating normal shutdown. If necessary, packet handling may be restarted without losing information.
- G_ConfigFlushBuffer: The *Stage 1 Pass-Through Handler* (described later) responds to this request by clearing the shared memory queue, discarding any unprocessed packets.
- G_ConfigSetRetries: This request specifies the maximum number of times an output buffer may attempt to send a packet to its destination if it receives a *Link Busy* error. The default is zero.
- G_ConfigSetDestination: This request specifies the output buffer to use by default when sending a packet to its destination using the *write* method in G_PacketHandler.

Additional configuration requests will be added in the future. Packet Handler-derived systems may automatically include the default configuration requests by creating an instance of the G_ConfigPacketFactory class and including it in the hierarchy of packet factories used by G_PacketInputBuffer as previously described in "*Sharing Packet Factories Between Systems*" on page 8.

Defining Strategies of Packet Handling

Strategies were previously described as *"the brains of the system"* (page 5). The design and implementation of the specific strategies used by each particular system is crucial to insuring that the system performs its intended operations correctly and efficiently. The behavior of each strategy is defined by the code implemented in its *execute* method. The *handlePacket* method in G_PacketHandler indirectly invokes the Strategy's *execute* method by calling the *execute* method in the particular packet it has been asked to handle. By default, all packets simply forward this request on to the associated strategy, including a pointer to both the specific packet and the specific packet handler as parameters (Figure 9).

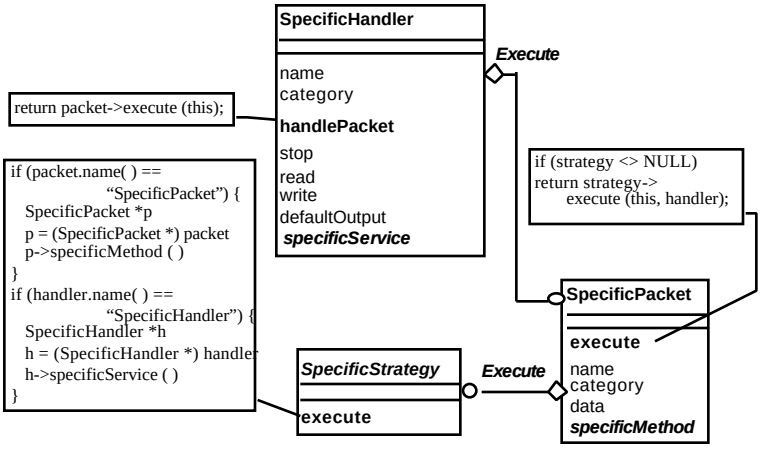

Figure 9 - Strategy Execution with Specific Packets and Handlers

The *execute* method defined in G_PacketStrategy takes pointers to G_PacketHandler and G_Packet as parameters. However, since strategies often must access methods defined in the specific subclasses actually passed to it, the code implemented in *execute* will typically need to use the *name* or *category* methods provided by G_Packet to insure that it can safely downcast the pointer. For the same reason, G_PacketHandler also provides equivalent *name* and *category* methods. The default implementation for these methods in G_PacketHandler is to return "G_PacketHandler". Subclasses must override these methods to insure that the proper name and category is returned.

Designing a Specialized Packet Handler System

The classes, their responsibilities and the way they collaborate as defined by the Packet Handler framework form a well organized and consistent architecture for deriving specific packet processing systems. This

architecture dictates the standard manner in which specific Packet Handler-derived systems are usually designed. Framework users will typically follow these simple steps:

1. Determine the specific kinds of data packets the system can receive, transmit and process. Create a specific G_Packet subclass for each, optionally grouping them into categories.
2. Determine the specific algorithms required to process each specific kind of packet or category. Create a specific G_PacketStrategy subclass for each, implementing each algorithm in the *execute* methods.
3. Create one or more subclasses of G_PacketFactory which create the proper specific packets for each individual kind of data block the system will receive and binds each to the associated specific strategy.
4. Determine any special services that the strategies require to perform their packet handling operations. Create a subclass of G_PacketHandler and add an additional method for each service.
5. In the constructor of the G_PacketHandler subclass, write the code needed to create and initialize the input buffer, output buffer(s), factory(s), communications wrappers and any additional objects that may be required.
6. Write a main routine which creates the G_PacketHandler subclass and then calls its *connect* and *start* methods. After the *start* method returns, destroy the packet handler and exit.

Two Process Staged Packet Handling

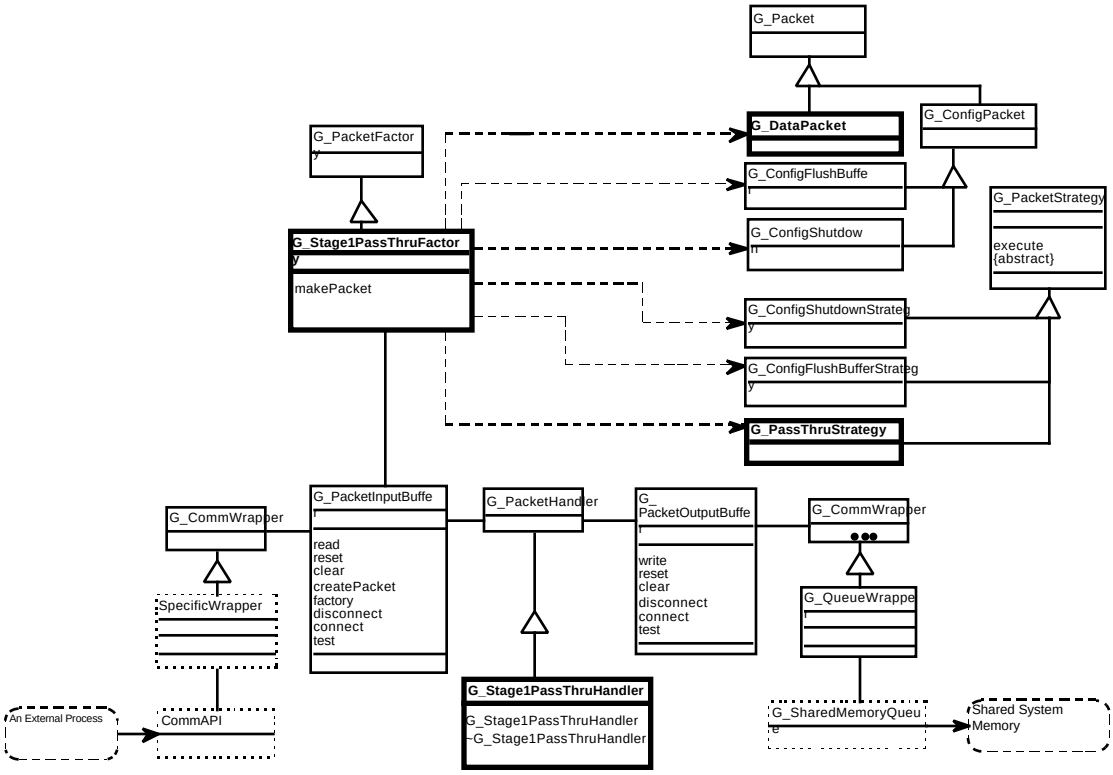

Figure 10 - Architecture of the Generic Stage 1 Pass-Through Process.

A typical configuration used by the Vision2000 modules is to create two or more processes in each executable module for staged packet handing. Each stage implements a distinct Packet Handler-derived system. The stages communicate via shared memory queues. The first stage simply retrieves packets from the input source and transfers them to a shared memory queue. Later stages remove the packets from the shared memory queue and perform the actual processing needed for that module.

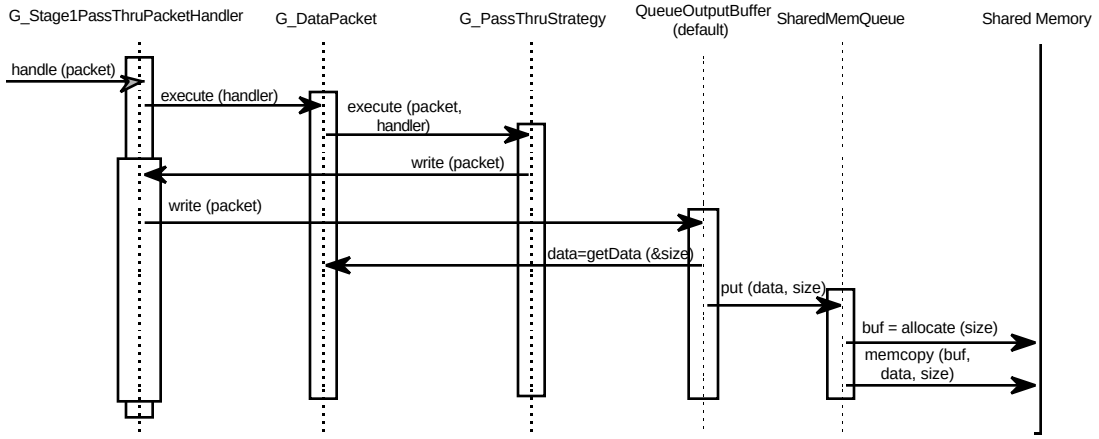

Figure 11 - Generic Stage 1 Pass-Through Packet Handling

The primary reason for this multi-stage configuration is to insure that the modules in the Vision2000 system that generate data packets are not blocked waiting for packet handling to complete. This configuration is typically required since Vision2000 IPC modules normally use blocking IO.

Since this configuration is so common, a generic set of specific classes are provided to fully implement the first stage. For the first stage process, the application need only create an instance of G_Stage1PassThruPacketHandler (a subclass of G_PacketHandler) and provide it with the communications wrapper to use for data input and a shared memory queue object (an instance of G_SharedMemoryQueue) for output.

The shared memory queue object must be created prior to forking the child process. We also want to insure that the first stage is initialized prior to starting the later processes. The following code illustrates the typical initialization of a two stage process:

```
int main () {

      G_SharedMemoryQueue *queue;
      queue = new G_SharedMemoryQueue (maxItems, itemSize);

      G_SocketWrapper * socket;
      socket = new G_SocketWrapper (destAddress, portID);

      // initialize stage 1 -- exit if it fails!
      G_Stage1PassThruPacketHandler stage1 (socket, queue);

      if (stage1.connect() < 0) {

            //error! Connect failed!
            exit ( -1 );

      } else if ( fork () == 0 ) {

            // Initialize second stage packet handler here,
            // passing the queue pointer to its constructor:
            // Note: if we also pass the socket wrapper, then
            // stage2 can use it to send messages back to stage1.
            MyPacketHandler stage2 ( queue, socket );

            if (stage2.connect() > 0) {

                  stage2.start ()

                  // after returning, packet handling is done.
                  // Note, the handlers will delete the queue

            } else {
```

```
                    // Couldn't connect stage2! So shutdown stage1
                    // This must be done by creating a Shutdown
                    // Config request and putting it on the queue...
                    G_ConfigShutdown shutdown ();
                    int size;
                    char data [] = shutdown.data ( &size )
                    queue->put (data, size);
                    delete data [];
              }
        } else {
              //start stage 1...
              stage1.start ();
              //when we return, all is done...
        }
  }
```

Conclusion

The packet handler framework provides a robust, highly configurable class library for use in the creation of packet processing systems. It drastically reduces the work required to design and implement these systems while insuring that each system shares as consistent, well tested architectural foundation. Systems built using this framework can easily share common data structures and algorithms while still allowing the developers a large amount of freedom to determine the special processing needs of each system individually.

The design and implementation of highly flexible, yet robust abstract architectures has historically been an extremely difficult skill to learn and master. Communicating the internal structures and purposes of such systems has also been historically difficult. My experience with the design, implementation and developer training for Packet Handler indicates that the usage of design patterns has significantly reduced the difficulty of these tasks. Personally, I've found that design patterns aid in increasing my own understanding of the consequences of each design decision in terms of future flexibility and conformance to state-of-the-art design practices.

The packet handler framework forms an important first step for the Vision2000 development effort towards significantly reducing development costs while increasing overall system flexibility and consistency. Further analysis is needed to determine what other reusable class libraries may be advantageous for the development of modules which do not fit into the Packet Handler architecture. Additional work is needed to find ways to seamlessly link Packet Handler with these alternate architectures. Additional documentation, developer training and a centralized reuse repository will also be needed. Packet Handler is only the beginning.

Acknowledgements

The Packet Handler framework and a number of Packet Handler derived applications are currently under development by the Front-End Processing Team of the Vision2000 Control Center System in Lanham, MD under the management direction of Mike Garvis. The development team members are: Sheng Chen, Bob Crane, James Jao, Ken Kang, Mike Lin and Wu Liu.

References

[Beck96] Kent Beck, James O. Coplien, Ron Crocker, Lutz Dominick, Gerard Meszaros, Frances Paulisch and John Vlissides. “Industrial Experience with Design Patterns”, In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, March 1996, pp. 103-114.

[BJ94] Kent Beck and Ralph Johnson. Patterns Generate Architectures. In *European Conference on Object-Oriented Programming (ECOOP)*, 1994.

[GOF93] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, “Design Patterns: Abstraction and Reuse of Object-Oriented Design”. In *European Conference on Object-Oriented Programming*, Kaiserlauten, Germany, July 1993. Published as Lecture notes in Computer Science #707, pp. 406-431, Springer-Verlag.

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns -- Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.

[GS94] David Garlan and Mary Shaw, “An Introduction to Software Architecture”, School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-94-166, Pittsburgh, PA, January 1994. Also published as “An Introduction to Software Architecture,” *Advances in Software Engineering and Knowledge Engineering, Volume I,* edited by V. Ambriola and G. Tortora, World Scientific Publishing Company, New Jesery, 1993. Also appears as CMU Software Engineering Technical Report CMU/SEI-94-TR-21, ESC-TR-94-21.

[Johnson88] Ralph E. Johnson and Brian Foote, “Designing Reusable Classes” *Journal of Object-Oriented Programming,* 1 (2):22-25, 1988.

[Johnson91] Ralph E. Johnson and Vincent F. Russo, “Reusing Object-Oriented Designs”. Department of Computer Science, University of Illinois Technical Report UIUCDCS 91-1696, Urbana, IL, 1996.

[Pree95] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley Publishing Company, 1995.

[VanCamp96] David Van Camp, “Object-Observations: Reusing the Wheel”. KeaneTechnotes, 1 (2) 8--14, Keane Inc., Albany, NY, 1996.