1 Pattern: Data flow architecture

Context

Sometimes the solutions for different problems share several identical steps, yet each problem requires its own solution.

Consider for example multimedia applications that process JPEG still images and MPEG videos. A JPEG viewer applies five transformations and each of them is performed in a separate step: JPEG syntax decoder, dequantizer, inverse DCT transformer, frame decoder and color conversion. Likewise, an MPEG player applies four transformations: MPEG syntax decoder (for simplicity, this also deals with temporal and spatial prediction), inverse DCT transformer, frame decoder and color conversion. For both applications, the transformations are applied sequentially, in the same order.

The inverse DCT transformer and color conversion are identical and operate on the same data types—images represented with one luminance component (Y) and two chrominance components (C_b and C_r). The interaction with the other transformations is limited. Control flow is static and the succession of steps rarely changes—usually, only for exceptional circumstances. Each time the inverse DCT transformer is presented a set of coefficients, it computes the corresponding spatial representation. Next this is passed to the frame decoder.

Problem

Processing within your domain is performed by applying a sequence of operations on similar data elements, in the same order. You want a reusable toolkit that can be used to build software solutions for a wide range of applications within the domain. Your applications are also required to dynamically change their functionality and/or adapt to changing environments, without compromising performance. What architecture can accommodate these requirements?

Forces

- A high-performance toolkit that is applicable for a wide range of problems from a specific domain is required;
- The application's behavior needs to be modified dynamically or adapt to changing environments at runtime;
- The loose coupling associated with the black-box paradigm has performance penalties;

- Scalability in several dimensions is required.

Solution

A data flow architecture organizes applications as a network of processing modules[1] that apply a series of transformations to one or several data streams. Each module takes its input from some upstream modules, performs a simple, generic transformation and passes the results to some other downstream modules. Enforcing strict, simple inter-module interfaces yields a large number of possible combinations that provide solutions to many problems within a particular domain. Therefore, in data flow architectures, processing modules are the *computational units*, while the network represents the *operational unit*.

Consequently, the functionality (transfer function) is determined by the:

- types of processing modules within the network; and
- interconnections between them.

Figure 1 shows a data flow JPEG decoder. The decoder reads a JPEG file and shows its contents on a display. Decoders for other formats are implemented by connecting different modules.

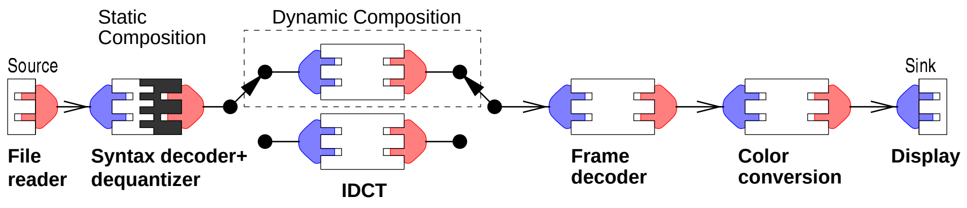


Figure 1: A data flow JPEG decoder.

This architecture is applicable only for data-driven applications, where the output is obtained by performing various sequential transformations to the input. It allows applications to scale with the number of data streams. The transfer function is determined by a combination of the individual transfer functions of each processing module. Control flow is distributed across the participating entities and is not explicitly represented at the architectural level.

Modules play a key role in data flow architectures. Applications that follow this pattern manifest an increased degree of modularity. This makes it easy to distribute the development effort among different groups. For example, visualization

[1]In this context, "module" is any processing unit within the application domain.

modules are developed by the visualization group, while MPEG modules are built by the multimedia group, etc. Therefore, the knowledge of domain experts is encapsulated within processing modules that are reusable in many different contexts.

The following design guidelines are essential for generic and reusable processing modules:

- The transfer function should not model the domain, depend on a particular solution or have side effects.
- Each module should be developed independently of its use.
- Processing modules should cooperate only by using the output of one (or several) as the input to another.

Decoupling the algorithmic properties of processing modules from a particular problem allows applications to use them as black-boxes. However, the black-box approach does not take into account context-specific factors and therefore has performance penalties. Good filters balance these two conflicting forces.

The fine granularity decomposition into simple transformations is an incarnation of the "divide and conquer" principle: instead of attempting to solve a complex problem all at once, split it into sub-parts and deal with each separately. This allows applications to scale with the number of transformations and processors—each processing module can run on a separate processor. However, because of the reduced communication overhead, specialized processing modules are more efficient than implementing their functionality with several generic filters. But specialization also limits the number of possible configurations that a filter can be part of. For example, a single specialized JPEG decoder module offers better performance than the decoder from Figure 1. However, in the latter case filters are reusable and can be employed for other applications (e.g., an MPEG decoder). Filter designers have to determine the right balance between granularity and performance.

The fine granularity decomposition has some other advantages. Consider how electrical engineers analyze or test a circuit. They use an oscilloscope to check the signals at different key points. This requires *access to each component*, and the first step is to remove the covers that enclose the circuit. Because of its modular structure, the data flow architecture provides easy access to its computational units, the processing modules. Like for the electrical circuit, this enables software engineers to analyze and test the application. For example, the decoder from Figure 1 provides direct access to the IDCT coefficients, should they be required for analysis or testing. Analyzing a functionally equivalent decoder which does not follow this pattern requires more effort to get to these coefficients. Therefore, from an electrical engineering perspective, data flow applications do not have "covers" that prevent access to their components.

Easy access to data at different processing stages also enables developers to use signal processing techniques (translation, interpolation, etc.). For fields like multimedia or scientific visualization, these techniques have been shown to improve performance [PLV97].

Inter-module communication is done by passing messages or tokens through unidirectional input and output ports—shown as blue (input) and red (output) plugs in Figure 1. Therefore, data flow systems replace inter-module direct calls with a message passing mechanism. The overhead associated with this mechanism is the main liability of the pattern.

Depending on their ports, modules are classified as follows (a comprehensive classification according to various criteria is available in [Lea96]):

Sources Interface with an input device and have one or several output ports.

Sinks Interface with an output device and have one or several input ports.

Filters Have both input and output ports (not necessarily only one in each direction) and perform processing on the information fed into the input port. When processing completes, the filter writes the results to the output port.

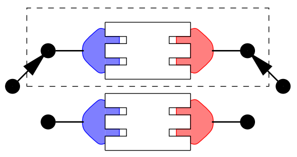
In Figure 1, the file reader is a source, the display adapter is a sink and the other processing modules are filters.

Unidirectional input and output ports are not a limitation. Rather, they increase a component's autonomy. Provided that there are no feed-back loops, processing is unaffected by the presence or absence of connections at the output port(s). Therefore, because any component depends only on the upstream modules, it is possible to change its output connections at runtime. This is called dynamic composition and determines some of the interesting properties of data flow architectures.

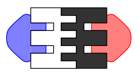
To allow interconectivity between two modules, the output port of the upstream module and the input port of the downstream module have to be *plug-compatible*. (For simplicity, in Figure 1 all modules have the same type of port.) However, having many port (plug) types limits interconectivity and requires adapters to connect incompatible ports.

Two priority levels are available for processing at the filter level. High priority processing can be done when the filter receives a message from the upstream module. Once acquired, the data can be processed at a later time, at lower priority. For instances where processing is time-comsuming—e.g., scientific visualization applications—the two priority levels are important. Otherwise, if the output values can be computed relatively fast, all processing is performed at high priority. However, scheduling (i.e., "when" the actual processing takes place) is nondeterministic in both cases.

Applications following the data flow architecture can adapt to heterogeneous and dynamically changing environments at runtime. Different implementations for stateless filters are exchanged to adjust the resource cost and quality characteristics within some user-defined limits. In Figure 1, two different IDCT modules are available and depending on their characteristics, the decoder uses one or the other. Therefore, *dynamic composition* allows an application to (1) adjust the resource consumption of the system at runtime, and (2) adapt to different computing and communications environments, as well as changes in resource availability.



Sometimes the overhead of inter-module communication has unacceptable performance penalties. An effective solution for this problem is to trade flexibility for performance [PLV96]. This is accomplished by replacing the adjacent performance-critical modules with a single optimized module. *Static composition* provides the underlying application (e.g., compiler) with enough information to collapse the sequence of filters into a functionally equivalent primitive filter, reducing the overhead of inter-module communication. However, this cannot be modified at runtime anylonger.



Static binding should be used to improve performance for filters that have low reconfigurability demands and are called with high frequency. Dynamic binding is best suited for filters that have high reconfigurability demands and low invocation frequency. A good balance between the use of static and dynamic composition ensures efficient implementations while maintaining a flexible, configurable architecture. Therefore, data flow applications are scalable with different communication and processing requirements. For the JPEG decoder from Figure 1, the syntax decoder is called many times for each image and therefore is statically bound to the dequantizer [PLV97]. The other components are called with lower frequencies and the message passing overhead is tolerable.

Posnak et al [PLV97] provide a quantitative evaluation of a data architecture in the context of their Presentation Processing Engine framework. The analysis considers the dimensions in which this pattern has the strongest impact: *reuse* and *performance*. (i) To estimate reuse, they consider the task of building a media player plug-in for Netscape. Using the PPE, this requires a few lines of Tcl code (see below) and little domain expertise. In contrast, reusing the Berkeley MPEG source code (therefore starting from a working product) requires a good understanding of the existing code and a significantly larger programming effort and domain expertise. (ii) For performance evaluation, they compare PPE applications with commercial products. The decoding time for a JPEG image (with the decoder from Figure 1) is within 5% of the time required by the Independent JPEG Group decoder. Similarly, the frame rate of an MPEG player is at most 10% lower than the Berkeley player. These results show that the data flow architecture can provide reusable, high-performance software solutions.

Users familiar only with the application domain (e.g., scientific visualization) can create new applications by simply connecting existing modules, without performing any other programming. This is accomplished with scripting languages [Lin94, JBR96] or visual programming tools [AEW96] that assist the creation of module networks [Foo88].

The network normally triggers recomputations whenever a filter's output value changes. While manipulated interactively, it should be possible to temporarily disable the network such that no computation triggers until the network is in a valid, stable state. Disabling computations is useful if several parameters need to be modified without having the filters recompute every time a change is made, or if processing takes a long time.

To summarize, the data flow architecture has the following **benefits** (✓) and **liabilities** (✗):

✓ It emphasizes reuse at the processing module level and facilitates the rise of end-user programming, automation and software components.

✓ Because the interaction mechanism between modules is simple, the architecture is suitable for domain-specific frameworks and often is the subject of visual programming tools.

✓ Data flow applications are scalable in several dimensions—number of transformations, processors, data streams; and communication and processing requirements.

✓ The fine granularity decomposition into simple filters provides access to the data stream at various processing stages.

✗ A data flow architecture is not a good choice for applications with dynamic control flow or feedback loops.

✗ For instances where the overhead of enforcing the inter-module interface is too high, a different architectural choice might be a better solution.

✗ It does not deal well with unanticipated conditions. Signaling errors that occur inside filter modules is cumbersome and difficult [BMR+96]. Modules do not make any assumptions about their context and communicate only

```
template<class DataType> class PushInput {
public:
  virtual ˜PushInput() { };
  virtual void Put(DataType &data) =0;
};
```

Figure 2: The `PushInput` class.

with other data-processing modules. Only when combined with additional knowledge about the network topology and the application's domain could an error message generated by a module be meaningful. The Processing and control partitions pattern provides one possible solution.

✗ The architecture works best for applications where modules exchange simple data and do not share state. It is not suitable when the exchanged data is complex or for database applications.

Resulting context

Applications following the data flow architecture can run on heterogeneous distributed systems. Because control flow is not explicitly represented, this change is transparent for sources, filters and sinks.

Implementation notes

This section follows the guidelines from [PLV96]. Processing modules inherit an input interface (port) from `PushInput` and an output interface from `PushOutput`. `PushInput`—Figure 2—is an abstract class that determines the input data type. Subclasses implement the `Put()` method to process the input data and pass it downstream. `PushOutput`—Figure 3—determines the output data type and defines methods for dynamic composition—`Attach()` and `Detach()`. Attachment between two modules is abstracted by the `PushPort` class—Figure 4. This facilitates establishing one-to-many connections, in which case `PushOutput` has to be modified accordingly.

C++ templates parameterize the classes presented in Figures 2–4 by the types of data that enters and leaves each processing module. The compiler's type checking system prevents connecting two modules which are not plug-compatible.

Based on these classes, implementing processing modules is straightforward. Figure 5 shows the code corresponding to a module that takes one `integer` input

```
template<class DataType> class PushOutput {
public:
  virtual ˜PushOutput() { };
  virtual void Attach(PushInput<DataType> *next)
   {
     _port=new PushPort<DataType>(next);
   };
  virtual void Detach()
   {
     delete _port;
   };
protected:
 inline void Output(DataType &data) { _port->Output(data); };
private:
 PushPort<DataType> *_port;
};
```

Figure 3: The `PushOutput` class.

```
template<class DataType> class PushPort {
public:
  PushPort(PushInput<DataType> *module)
    : _module(module) { };
  ˜PushPort() { };
  inline void Output(DataType &data) { _module->Put(data); };
protected:
  PushInput<DataType> *_module;
};
```

Figure 4: The `PushPort` class.

```
class Sqrt : public PushInput<int>, public PushOutput<double> {
public:
  Sqrt() { };
  ~Sqrt() { };
  void Put(int &i)
  {
    double d=sqrt(i);
    Output(d);
  };
};
```

Figure 5: An `Sqrt` processing module.

and outputs its square root. However, this simple implementation has several limitations with respect to how processing is initiated and how data is passed between modules. The following sections cover these aspects and provide alternatives.

In this example, type parameterization—templates—and function inlining are used to implement static composition. Figure 6 shows the code corresponding to a filter that increments its `integer` input. Processing—incrementing—is performed by the `Transform()` inline method. The `FastInc` class can parameterize the filter from Figure 7. Because `FastFilter` invokes only the inlined `Transform()` method, `FastInc`'s input and output ports are bypassed and the implementation is efficient. Figure 8 shows how to instantiate the static composition of `FastInc` with `FastFilter`. The resulting filter first increments the input and then outputs its square root. The idea behind implementing static composition is bypassing the input and output ports to eliminate the communication overhead. This can be accomplished in any language and does not require templates and inline methods.

Examples

1. **Scientific visualization** is one domain where the data flow architecture is used extensively. Systems like AVS [AVS93] or Iris Explorer [EXP93] allow users to create visualization applications and visualize various data types in many different ways. This is done by connecting modules into a network, with the help of a visual network editor. Because they provide a wide range of generic filters, these systems offer solutions to a large number of visualization problems. Figure 9 shows an example of an AVS network.

   Whenever required, the existing set of filters can also be expanded. New filters are created with a module builder tool. This allows end-users to extend

```
class FastInc : public PushInput<int>, public PushOutput<int> {
public:
  FastInc() { };
  ˜FastInc() { };
  virtual void Put(int &i)
    {
      int j=i;
      Transform(j);
      Output(j);
    };
  inline void Transform(int &i) { i++; };
};
```

Figure 6: A filter for static composition. Only the inlined method is used by the composite filter.

```
template<class T>
class FastFilter : public PushInput<int>, public PushOutput<double> {
public:
  FastFilter(T &component)
    : _component(component) { };
  ˜FastFilter() { };
  virtual void Put(int &i)
    {
      int j=i;
      _component.Transform(j);
      double d=sqrt(j);
      Output(d);
    };
private:
  T _component;
};
```

Figure 7: A composite filter.

```
FastInc fi;
FastFilter<FastInc> ff(fi);
```

Figure 8: Instantiating the composite filter.

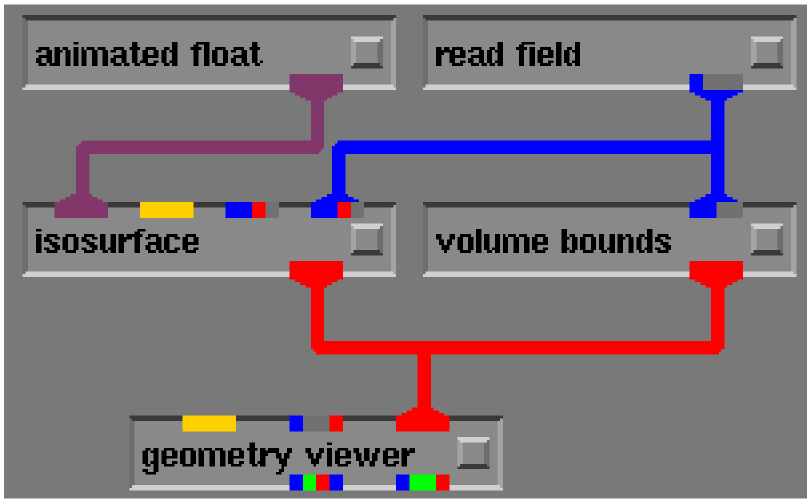


Figure 9: AVS Network. Top side ports are inputs, while bottom side ports are outputs. The port types are color-coded and the network editor does not permit connecting incompatible ports.

functionality with a minimal amount of programming.

2. Data flow architectures have also been investigated for **hardware** systems [Gur85, Pap91]. The execution model offers attractive properties for parallel processing—implicit synchronization of parallel activities and self schedulability. Unlike the von Neumann model which explicitly states the sequence of instructions, in the data flow model the execution of any instruction is driven by the operand availability. This emphasizes a high degree of parallelism at the instruction level. The Monsoon Project [Pap91] developed by MIT and Motorola produced a data-flow multiprocessor targeted to large-scale scientific and symbolic computation. Its success motivated much of the work on similar projects [NPA92] and contributed to spread the interest in data flow and parallel programming.

3. **Avionics Control Systems** (ACS) also employ this pattern [Lea94]. Because ACS are complex systems, constructing a set of components that merge all possible combinations is not feasible. The data flow architecture provides the means for combining together different types of existing components (filters) to serve a particular purpose.

4. Another domain where the data flow architecture is employed by applica-

tions (ActiveMovie [AMS], VuSystem [Lin94]), toolkits (Berkeley Continuous Media Toolkit [JBR96]) and frameworks (MET++ [Ack94], Presentation Processing Engine [PLV97], Java Media Framework [SUN97]) is **multimedia** [MN98].

VuSystem applications[2] illustrate very well the flexibility of this architecture. In the "room monitor," the output of a surveillance camera is analyzed such that just the frames that contain motion are recorded. The "joke browser" extracts only selected parts (e.g., jokes) from the complete recording of a late night show. Although these two applications are very different (the former processes real-time video while the latter provides content-based access), both of them have been built with the same tool just by connecting existing modules.

ActiveMovie allows users to play digital movies and sound encoded in various formats. It consists of sources, filters and renderers connected in a filter graph. All graph components are implemented as Component Object Model (COM) objects. Filters have pins which are connected with streams, but other communication channels for specialized communication (e.g., error notifications) are available.

The presence of the data flow architecture within the forthcoming Java Media Framework demonstrates its validity and confirms it as a recurring solution that has passed the test of time.

Variants[3]

1. Bill Walker's Ph.D. thesis [Wal94] presents a framework that uses this pattern in the context of **computer-assisted music**. Unlike typical data flow architectures, this system has *feedback loops* and *shared state*. The processing modules are called `Composers` and `Transformers`. Global information independent of the components is encapsulated within a `PolicyDictionary` object.

2. In [Rit84], Dennis Ritchie describes a variant of this pattern and how it is implemented in the **UNIX stream system**. Data flow is *bidirectional* and filters have queues for each direction. The queues are also employed for flow control—Section 3.

[2]Some of the VuSystem applications are available on the world-wide web at `http://www.tns.lcs.mit.edu/vs/vusystem.html`

[3]The differences between the data flow architecture and the variants discussed in this section are shown in italics.

Related Patterns

- Messages encapsulate all types of information exchanged by collaborating modules. The Payloads pattern allows modules to distinguish between messages and the data within them.

- Data flow architectures replace inter-module direct calls with a message passing mechanism. The Payload passing protocol patterns provide several alternatives for this mechanism.

- A distinct part of a data flow application is in charge with dynamic composition and handles error messages. Processing and control partitions describes how to organize such applications.

- Systems following the Layers pattern [BMR$^+$96] are organized as layers that exchange data between them. However, in this case each layer corresponds to a different abstraction level where data has different semantics.

- Static composition is an instance of the Composite pattern [GHJV95]. This ensures a consistent interface between primitive (e.g., statically composed filter) and container (e.g., sequence of filters) objects.

- Streams [Edw95], Pipes and filters [Meu95, BMR$^+$96] and Pipeline [Sha96] provide similar solutions. However, the data flow architecture is more general. For example, Pipes and filters takes a more static approach, where once the processing pipeline is set up, it is not allowed to change.

2 Pattern: Payloads

Context

In the landmark paper "The Early History of Smalltalk" [Kay93], Alan Kay remembers a few ideas from the sixties that have influenced his thinking about OOP. One of them was the method of transporting files on tape for the Burroughs 220, while he was working as a programmer in the Air Force (around 1961):

> There were no standard operating systems of file formats back then, so some designer decided to finesse this problem by taking each file and dividing it into three parts. The third part was all of the actual data records of arbitrary size and formats. The second part contained the B220 procedures that knew how to get at records and fields to copy and update the third part. And the first part was an array of relative pointers into entry points of the procedures in the second part (the initial pointers were in a standard order representing standard meanings).

Current computer systems use different means to exchange information. In most circumstances, this is done by passing messages through a fast communication channel. However, the above requirements are the same. (i) Messages encapsulate all types of information, including control and data of variable length. This corresponds to the third part of the B220 file. (ii) Applications need a mechanism to access the various types of information within messages. The mechanism depends on the message contents and corresponds to the B220 procedures. (iii) The standard pieces of information associated with each message are required by all message handlers—applications that handle messages. This corresponds to the array of pointers.

Problem

Communication within a software system is performed by exchanging different types of messages between communicating entities. How to distinguish between messages and the information within messages?

Forces

- Communication is restricted to message passing;
- The structure of each message depends on the information that it contains;

- The communicating entities exchange many different types of information;
- Applications receive a wide range of messages, including messages they do not process;
- Depending on the information they encapsulate, some messages need special processing.

Solution

Represent messages with payloads. A payload provides an abstract model that allows the communicating entities to identify different message types, as well as distinguish the information encapsulated within each payload.

Payloads have two components:

Descriptor component Descriptors contain *general information* about the message, such as type, asynchronous flag, priority level, etc. For each message type, the descriptor also contains *type-specific parameters*. For example, if the payload contains an image, one possibility is to encode its name, size and format.

Data component The data component holds the information transported by the message. Its size and format depend on the descriptor component.

The previous components offer solutions for two of the requirements identified in "Context." The descriptor provides a standard way to organize the information about each payload. It corresponds to the first part of the B220 file. Similarly, the data component contains the actual information. This corresponds to the third part of the file. However, so far none of this components offers behavior—the second part. This is covered later on in this section.

Not all payloads contain both components. Sometimes payloads that correspond to control messages do not need any additional data. Therefore, these payloads contain only the descriptor. Whenever both components are present, the size of the data component is usually much larger than the descriptor. This characteristic is used to optimize the payload copying mechanism—see below.

Figure 10 shows a control payload and a data payload corresponding to an image. The general information encoded in the descriptor contains the message type, an asynchronous flag and a priority level. In contrast, the descriptor of the data payload corresponds to an image and has three additional fields. These contain the image name, size and format.

An object-oriented approach is an elegant solution for payloads. *Polymorphism* provides behavior parameterized by the data type. This renders the explicit encoding of the payload type in the descriptor unnecessary. Removing the responsibility

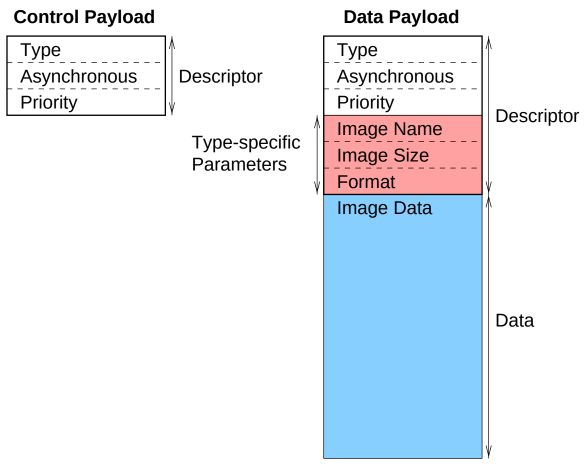


Figure 10: Different payloads and their components.

of message type checking from the programmer reduces complexity and improves efficiency. *Inheritance* groups the attributes and behavior common to all message types in a superclass, thus promoting reuse. Adding a new payload requires only to accommodate the new attribute(s)—if any—and to provide the appropriate behavior. Existing software continues to work with new payloads through the superclass interface. Therefore, unless it requires the new attributes or data, software can be reused without modification. *Encapsulation* allows attributes like an asynchronous flag or a priority level to be associated with each payload. Relocating the knowledge about this information inside payloads decreases the coupling between the communicating entities and the messages they exchange. Consequently, as long as the payload interface does not change, attribute representations can be modified without affecting existing payload handlers.

In concert with a message passing mechanism (Section 3), payloads replace direct calls when these are not feasible. One circumstance is whenever the communicating entities reside across logical boundaries. This is typical to distributed systems and multiprocessor computers. Applications (processes) running on different machines or processors exchange data only by sending messages to each other. Another circumstance is whenever the communicating entities exchange complex data. Inter-layer calls in a layered operating system typically require a large number of parameters. Several modern operating systems have replaced these calls with a message passing mechanism [TW97].

Although payloads offer several benefits over direct calls, these do not come for free. The pattern trades performance for flexibility. One of the main liabilities of this pattern is the overhead required for payload *assembling* and *disassembling*. However, the decreased coupling between the communicating entities and the messages they exchange facilitates the optimization of the message passing mechanism (Section 3). Transfer mechanisms that require additional information about messages are able to obtain it directly from the message, through a well-defined protocol. This can help reduce the message passing overhead.

Another operation that makes payloads (when used in conjunction with a message passing mechanism) less efficient than direct calls is payload *copying*. Sometimes it is possible to substitute payload copying with change of ownership. However, if the payload is sent to multiple recipients, it has to be cloned such that each receiver gets its own copy. But cloning wastes space and may not even be viable for payloads with large memory footprints. Under these circumstances, the receivers can share the same payload. Should a receiver modify the incoming payload, it has to obtain its own private copy. When copying is cannot be avoided, an optimized technique can be employed:

Shallow copy Copy just the descriptor and share data components. However, the shared data component cannot be changed.

Deep copy Copy the data components as well. Implementing this as copy-on-write can improve performance.

If the entities that exchange the data reside across hardware boundaries, changing of ownership without copying cannot be avoided and the entire payload has to be transferred.

One possible way to improve performance is to pack multiple payloads into a single container payload [SC95]. A typical instance is whenever payloads are transferred over a network. In this case, although the data transfer can be relatively fast, initiating individual connections is expensive. However, this approach is viable only for high message rates or bursty traffic, such that the sender does not have to hold a message too long before passing it further. Container messages are employed by operating systems which consist of several interacting components [TW97, CRJ87] and Information Systems [Fow97].

Passing payloads instead of using direct calls has additional consequences. (i) Payloads hide the details of the communication channel. Once communication is done only by passing payloads, changes of the communication channel are transparent. For example, the communicating entities can be relocated on multiple processing units that are linked by a high-speed network—e.g., ATM or Ethernet. Therefore, payloads facilitate the distribution of existing applications over multiple processing units, which can be heterogeneous. (ii) Payloads are standlone objects and can be persistent. Mission-critical applications use persistent messages to cope with failures. For example, IBM's Exotica/FMQM (FlowMark on Message Queue Manager) [AAA$^+$95] project is a fully-distributed workflow system that employs IBM's Message Queue Interface (MQI) for persistent messaging. MQI eliminates the need for a centralized database, which in many workflow systems constitutes a single point of failure and sometimes a bottleneck. Asynchronous communication between applications that run at different points in time becomes possible. Therefore, persistent payloads offer increased resilience to failures, greater scalability and flexibility of system configuration.

A large number of applications use payloads. Consequently, automated tools for payload management can be valuable during the software development process. One such tool is the Message Translation and Validation (MTV) Builder [Eng] from Accel Software Engineering. MTV automates software generation to facilitate communication between systems and devices of various types. The automatically-generated code provides message translation and validation capabilities that can be integrated into an application. Payloads are built and modified through a GUI. Starting from a visual representation, MTV Builder automatically generates the interface control documents, the translation and validation software, and the test software. Therefore, the code maintenance time is eliminated.

In summary, the payloads pattern has the following benefits (✓) and liabilities (✗):

- ✓ Payloads increase the overall flexibility and permit the addition of new features (e.g., priority levels, support for asynchronous events) with minimal changes.
- ✓ Adding new message types does not require changing the existing entities which are not interested in them. For example, in a data flow architecture (Section 1), filters pass downstream the messages they do not understand, without performing any processing—also known as "tunneling."
- ✓ Message contents can be modified at runtime. Modules insert or remove message components before passing them further. Consequently, the traffic can be reduced by piggybacking information on passing messages.
- ✗ The main liability of this pattern is its inefficiency when compared with direct calls. This is due to the high overhead associated with copying and payload assembling/disassembling.

Implementation notes

As mentioned before, the key properties of the object-oriented paradigm provide an elegant solution for the three requirements identified in "Context."

```
class Message {
public:
  Message();
  virtual ˜Message();
  virtual Serialize() =0;
  virtual Deserialize() =0;
  /*
  Other methods for messages
  */
};
```

Figure 11: The `Message` abstract base class.

A flexible implementation solution is to regard the payload as a composite message [SC95]. Concrete classes corresponding to various message types are derived from the abstract class `Message`—Figure 11—and provide implementations for its interface. The `Payload` class—Figure 12—is a composite `Message` that extends the interface with several descriptor-specific methods. In this example, each payload has associated with it a priority level and an asynchronous flag. Clients can subclass `Payload` to extend the descriptor-specific interface in a transparent way.

Examples

1. Many **multimedia** applications use the payloads pattern. VuSystem [Lin94] has `VideoFrame` for single uncompressed video frame; `AudioFragment` for a sequence of audio samples; and `Caption` for close-captioned text. The payloads provide marshalling/unmarshalling methods which allow VuSystem modules to reside on different computers. In ActiveMovie [AMS], payloads are either media samples or quality control data. Media data originates at the source and is passed only downstream. Quality control data provides a means to gracefully adapt to load differences in the media stream. It is used to send notification messages from a renderer either upstream, or directly to a designated location. All the elements of the architecture architecture recognize an asynchronous event which requires graceful flushing of old data, followed by global resynchronization.

2. Abstract Syntax Notation One (ASN.1) [X2088a] is a method of specifying abstract objects in the **Open Systems Interconnection (OSI) architecture**. The Basic Encoding Rules [X2088b] describe how to encode values of each ASN.1 type as a string of bytes. BER encodes each part of a value as a

```
class Payload : public Message {
public:
  Payload();
  virtual ˜Payload();
  // Message interface
  virtual Serialize() { /* */ };
  virtual Deserialize() { /* */ };
  /*
  Other methods for messages
  */
  // Payload-specific interface
  int PriorityLevel() const { return _priorityLevel};
  bool IsAsynchronous() const { return _isAsynchronous};
  /*
    Other descriptor-specific methods
  */
  // Composite interface
  virtual void AddMessage(Message *) { /* */ };
  virtual void RemoveMessage(Message *) { /* */ };
private:
  int _priorityLevel;
  bool _isAsynchronous;
  /*
    Other descriptor-specific data
  */
  vector<Message *> _components;
};
```

Figure 12: The `Payload` class.

(identifier,length,contents) tuple. The identifier and length correspond to the descriptor component, while the contents corresponds to the data component. Although ASN.1 and BER have been designed for the application-presentation interface within the OSI architecture, they have also been applied for distributed systems.

3. As mentioned before, this pattern is also employed by some **operating systems**. In the case of UNIX streams [Rit84], payloads are called message blocks. A header specifies their type (data or control), as well as other attributes (e.g., asynchronous event).

4. **Parallel computing** is another domain where payloads are used extensively, particularly for architectures that do not have shared memory. The Message Passing Interface (MPI) [SOHL$^+$96] is a standardized and portable message passing mechanism which runs on a wide variety of parallel computers. MPI messages call the descriptor component "message envelope."

Related Patterns

- Applications that use the data flow architecture typically employ payloads to encapsulate the data that flows through the network.

  What happens when a source module (for example, a file reader) reaches the end of the input stream? This condition is signaled by sending downstream a payload that corresponds to the "end of stream" control message. The mechanism is propagated down the filter network until it reaches the sink. Therefore, payloads enable modules to exchange control information in a transparent way.

- Marshalling and unmarshalling methods require payloads to reconstruct themselves from a byte stream. The receiver end can employ a factory method [GHJV95] to reconstruct the appropriate type of payload.

- Composite message [SC95] facilitates the packaging of several messages into a composite. Its objective is to improve performance by reducing the message passing overhead.

- The proxy [GHJV95, BMR$^+$96] pattern can help reduce the overhead of payload copying. A proxy payload has the same interface but postpones the data transfer until its data or descriptor components are needed.

References

[AAA+95] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, R. Günthör, and M. Kamath. EXOTICA/FMQM: A persistent message-based architecture for distributed workflow management. In *Proc. IFIP WG8.1 Working Conference on Information Systems Development for Decentralized Organizations*, Trondheim, Norway, August 1995.

[Ack94] Philipp Ackermann. Design and implementation of on object-oriented media composition framework. In *Proc. International Computer Music Conference*, Aarhus, September 1994.

[AEW96] Philipp Ackermann, Dominik Eichelberg, and Bernhard Wagner. Visual programming in an object-oriented framework. In *Proc. Swiss Computer Science Conference*, Zurich, Switzerland, October 1996.

[AMS] Microsoft Corporation, Seattle, WA. *ActiveMovie Software Development Kit version 1.0*. `http://www.microsoft.com/devonly/tech/amov1doc/`.

[AVS93] CONVEX Computer Corporation, Richardson, TX. *ConvexAVS Developer's Guide*, first edition, December 1993.

[BMR+96] Frank Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. John Wiley & Sons, July 1996. ISBN 0-47195-869-7.

[CRJ87] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proc. USENIX C++ Workshop*, pages 109–123, Santa Fe, NM, November 1987.

[Den80] J. B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11), 1980.

[Edw95] Stephen Edwards. *Streams: A Pattern for "Pull-Driven" Processing*, volume 1 of *"Pattern Languages of Program Design"*, chapter 21. Addison-Wesley, 1995. Edited by James O. Coplien and Douglas C. Schmidt. ISBN 0-201-60734-4.

[Eng] Accel Software Engineering. Message translation and validation builder. `http://www.accelse.com/`.

[EXP93] Silicon Graphics, Inc., Mountain View, CA. *IRIS Explorer User's Guide*, 1993.

[Foo88] Brian Foote. Designing to facilitate change with object-oriented frameworks. Master's thesis, University of Illinois at Urbana-Champaign, 1988.

[Fow97] Martin Fowler. *Analysis Patterns—Reusable Object Models*. Addison-Wesley Object-Oriented Software Engineering Series. Addison-Wesley, 1997. ISBN 0-201-89542-0.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[Gur85] J. R. Gurd. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1), January 1985.

[JBR96] MacDonald H. Jackson, J. Eric Baldeschwieler, and Lawrence A. Rowe. Berkeley Continuous Media Toolkit API. Submitted for publication, September 1996.

[Kay93] Alan C. Kay. The early history of Smalltalk. *ACM SIGPLAN notices*, 28(3):69–92, March 1993.

[Lea94] Doug Lea. Design patterns for avionics control systems, November 1994. DSSA Adage Project ADAGE-OSW-94-01.

[Lea96] Doug Lea. *Concurrent Programming in Java*. The Java Series. Addison-Wesley, 1996. ISBN 0-201-69581-2.

[Lin94] Christopher J. Lindblad. A programming system for the dynamic manipulation of temporally sensitive data. Technical Report 637, Massachutes Institute of Technology, August 1994. Laboratory for Computer Science.

[Meu95] Regine Meunier. *The Pipes and Filters Architecture*, volume 1 of *"Pattern Languages of Program Design"*, chapter 22. Addison-Wesley, 1995. Edited by James O. Coplien and Douglas C. Schmidt. ISBN 0-201-60734-4.

[MN98] Dragoş-Anton Manolescu and Klara Nahrstedt. A scalable approach to continuous-media processing. In *The 8th International Workshop on Research Issues in Data Engineering: Continuous-Media Databases and Applications*, Orlando, FL, February 1998.

[Moh97] C. Mohan. Recent trends in workflow management products, standards and research. In *Proc. NATO Advanced Study Institute (ASI) on Workflow Management Systems and Interoperability*, Istanbul, Turkey, August 1997. Springer-Verlag.

[MS96] David Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996. ISBN 0-201-63398-1.

[NPA92] R. S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proc. 19th Annual International Symposium on Computer Architecture*, pages 156–167. ACM, 1992.

[Pap91] Gregory M. Papadopoulos. *Implementation of a general-purpose dataflow multiprocessor*. MIT Press, 1991.

[PLV96] Edward J. Posnak, R. Greg Lavender, and Harrick M. Vin. Adaptive pipeline: an object structural pattern for adaptive applications. In *The 3rd Pattern Languages of Programming conference*, Monticello, IL, September 1996.

[PLV97] Edward J. Posnak, R. Greg Lavender, and Harrick M. Vin. An adaptive framework for developing multimedia software components. *Communications of the ACM*, 40(10), October 1997.

[Rit84] Dennis M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, March 1994.

[SC95] Aamod Sane and Roy Campbell. Composite Messages: A Structural Pattern for Communication Between Processes. In *Proc. OOPSLA Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems*, October 1995.

[Sha96] Mary Shaw. *Some Patterns for Software Architecture*, volume 2 of *"Pattern Languages of Program Design"*, chapter 16. Addison-Wesley, 1996. Edited by John M. Vlissides, James O. Coplien and Norman L. Kerth. ISBN 0-201-89527-7.

[SOHL$^+$96] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI—The Complete Reference*. The MIT Press, 1996. ISBN 0-262-69184-1.

[SPG91] Abraham Silberschatz, James L. Peterson, and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, third edition, 1991. ISBN 0-201-51379-X.

[SUN97] SUN Microsystems, Inc. Java media players. `http://java.sun.com/products/java-media/jmf`, November 1997.

[TW97] Andrew S. Tanenbaum and Albert S. Woodhul. *Operating Systems—Design and Implementation*. Prentice-Hall, second edition, 1997. ISBN 0-13-638677-6.

[Wal94] William F. Walker. *A Conversation-Based Framework For Musical Improvisation*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[X2088a] CCITT Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1), 1988.

[X2088b] CCITT Recommendation X.209: Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), 1988.