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

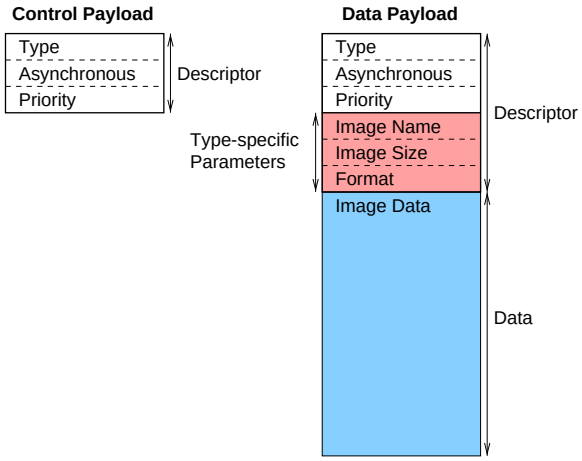

Figure 10: Different payloads and their components.

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 **??**), 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 **??**). 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 one container payload such that all of them are transferred in a single step [SC95]. This approach is well suited to high message rates or bursty traffic, such that the sender does not have to hold a message too long before passing it further. It has been successfully applied in operating systems which consist of several interacting components [TW97, CRJ87].

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 MQI API for persistent messaging. MQI eliminates the need for a centralized database, which for many workflow systems constitutes a single point of failure and sometimes a bottleneck. 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."

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

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

Figure 11: The `Message` abstract base class.

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 *(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-

```
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.

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.

[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.

[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.

[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.

[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.

[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.