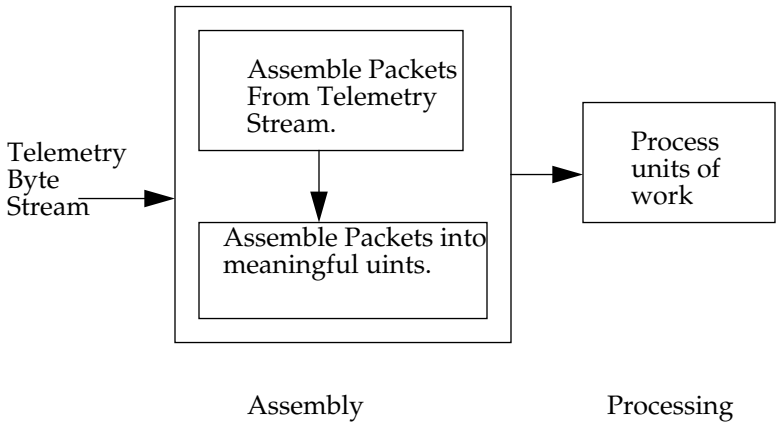
Handlers for Separating Assembly and Processing

Stephen P. Berczuk
MIT Center for Space Research
77 Massachusetts Avenue
Room NE80-6015
Cambridge, MA 02139
(617) 253-3840
berczuk@mit.edu

A science satellite is composed of different instruments developed by different institutions. In a science satellite telemetry processing application the requirements for different software components evolve at different rates. This is due in part to the fact that these applications are developed by different teams with often different operational and scientific requirements — one team is responsible for converting a "byte-stream" into telemetry packets and assembling the telemetry packets into units that represent the information in its original form[1], other teams are responsible for processing the data from the telemetry. The following diagram illustrates the system:



Processing is divided into assembly and processing phases.

The assembly subsystem has 2 roles:

- assemble the telemetry from the byte stream, this includes such tasks as reading a header field from the stream and then reading the number of bytes specified in the header.
- Assemble the packets into meaningful units of work. A pattern to isolate the main assembly subsystem from the details of what constitutes a unit of work can itself be the

[1] Telemetry is a transport medium. The satellite collects data into a buffer, and then divides the buffer into smaller pieces for transport. The ground systems must then re-assemble the packets into a representation of the buffer so that it can be interpreted.

subject of a pattern.

For the purposes of this pattern consider the two parts of the assembly system as one unit.

The operational requirements for the transmission and assembly of telemetry packets are well defined, since they are based on well established standards, such as the CCSDS specification, and on the specifications of the satellite hardware and controlling software. Telemetry assembly is also rather similar from mission to mission, thus requirements can be more easily defined.

Processing the assembled units of data is less well defined, since the operational requirements vary from mission to mission and may depend on preliminary analysis being done by scientists who designed the experiments on the satellite[2].

A telemetry processing application is a member of a class of applications in which one subsystem creates objects from external inputs in a well defined manner and another processes these collected data, in a less clearly defined manner.

A useful pattern would illustrate how one can isolate the "assembly" system from the "processing" system so that the creation system can be developed and tested without waiting for the processing system's requirements to firm up, thus liberating one system from the development delays caused by ill defined requirements for the downstream system.

This pattern should provide a clean interface so that subsystems written by different groups of engineers can be developed in a manner that avoids development bottlenecks. The assembly system should also be allowed to remain ignorant of the downstream processing.

Combining this pattern with a *Factory*-type pattern will decouple the interfaces between different development groups, by insulating the operations team from knowing the definition of units of work, and the details of which packets belong in which collections.

In a distributed development environment it is necessary to partition responsibilities in such a way that dependencies can be reduced while interoperability can be maintained. In particular, we should be able to complete the well defined portions of the system by isolating the development of these pieces from the "to be specified" pieces.

The prototypical system will be composed of two components:

- *Assembly* software that processes objects from an input stream, classifies them and assembles larger units, as necessary. Some of the units are composed of a number of packets; an example of this is a *partition*. Some of the units are simply individual packets, an example is packets containing *status* information about the spacecraft or the instrument. This software is typically well defined, and being key to any downstream processing, it is preferable that it be built and tested as soon as possible.

- *Processing* software, which analyses or reports on the data. This is typically more loosely defined since there are often many ways to process data (consider scientific data analysis applications). Because of the variety of ways in which data can be processed we wish to maintain a flexible system. The processing requirements may also take longer to come to closure since they are subject to the influence of more players.

Traditionally the systems are designed with end to end processing in mind, but if the end to end processing is not defined, then the entire system can be held back while the details of downstream

---

2. As an example, the hardware for the system I am currently working on was designed to transmit quite a bit of status information for verification, with the idea that some of it would be useful downstream. It is still being decided how the ground operation team will use all the information.

processing are resolved.

Each component is developed by distinct teams of developers, so building a tightly integrated system would create a great deal of uncertainty in the schedule.

We need a way to decouple the software that assembles the data from the software that analyzes the data, so that forward progress can be made and the assembly software can be completed. To isolate the assembler from the processor we need a way for the assembler to hand the completed unit off to the processor when the unit is available. We want a processing function to be added to a system dynamically, without changing the assembly software.

To summarize the forces at work:

- Requirements for one component of a system are available before downstream processing is defined.
- There is a need to test the assembly code independently.
- Assembly and downstream processing may be developed by separate, often non-collocated groups, with different communications dynamics.
- The assembly system should be developed assuming nothing of the downstream processing.

We can resolve the problem of decoupling the development of two systems, while maintaining operational coupling by the following:

By providing a facility to set handlers to be invoked when a complete "unit" is available, we are able to isolate the assembly software from the processing software. Virtual functions help make the interface consistent.

This is similar to a *callback* pattern, where a handler is specified to an object to be called when an event occurs, but the motivation differs. The emphasis in this pattern is on providing a way to allow teams of developers to decouple their work.

The classes representing the units of work that the assembly software processes have a facility to set a handler to be called when the unit is complete. The units of work can be individual data objects or objects created by assembling objects from a data stream. The interface to the assembly mechanism can be hidden by providing that each data object received by the assembly stream is classified to be an instance of a subclass of a processable unit, the packet, and a virtual apply() method is invoked. In the context of each apply() call, an object is inserted into the appropriate collection, and if the collection is complete, the handler is called.

Examples of collections include partitions (which represent a buffer of data), and status message packets.

The client creates a handler object (it can also be a pointer to a function) and sets the handler for objects of the same class. Only units which interest the client will be sent to it. The handler object can perform a variety of operations, such as interpreting the data stream and performing analysis, or simply reporting the information to a scientist or operations person.

Example syntax. (C++)

The following example shows a typical use of this pattern.

The collection (in this case, a *Partition*) class has methods to set a class-wide handler to be called when a unit is available, and a *dispose()* method to invoke the handler on the unit of work, The col-

lection can also have an *isComplete()* method.

The processing client sets the handlers to be called when a "unit of work" is available using the *setHandler()* method on the collection class. The unit of work handler is set for each class of units. The client then gets the next packet from the data stream (this makes use of a factory pattern, which classifies the bytes that represent a packet into the appropriate subclass). The virtual *process()* method is then invoked on the packet. The process method assembles the packet into a larger collection, if necessary. If the packet completes the unit of work (recall that a single packet can represent a unit of work) the handler is invoked.

```
PartitionHandler aCHandler;
StatusMsgHandler aSOMsgHandler;
class Partition
{
public:
// constructors and the like

setHandler(/*function taking a pointer to a Partition*/);
// set the handler to be called for all complete partitions

void dispose();
// apply the handler to the object

};
ingest_client(){
      StandAloneObject::setHandler(aSOMsgHandler);
      Collection::setHandler(aCHandler);

      Packet* foo = getNextPacket();
      // create a subclass of packet based on info in data stream.
      foo->process();
      // virtual function puts foo into the right collection (if //appro-
      priate) and the collection checks to see if it is full. if // foo
      is standalone, the handler is called.

      }

      Packet::process()
      {
      // takes the packet and inserts it into the collection, perhaps do-
      ing some processingto determine
      // which of a number of collections is appropriate
      theCollection->insert(this);
      if(theCollection->isComplete())
            theCollection->dispose();
      // dispatch the collection to the appropriate handler
      }
```

The assembly software needs to know only the following aspects of the downstream processing:

- What the units of work are that will be dispatched. The details of this that depend on the individual instruments can be hidden from the assembly client through the use of Factories and virtual functions.

- The methods used to set the handlers.

The downstream processing client needs to know only:

- The type of units of work to expect.
- The handling that each unit of work should be afforded.

Applications of this pattern include: telemetry processing where you must classify and assemble packets from a telemetry stream into partition, and certain packets represent status messages, which are not assembled into partitions.

This patterncan be applied in any system where there is a need for one system to perform an operation, and then hand results over to another system for further processing. It can be extended by allowing a list of handlers to be called whenver a unit of work is available, in a manner similar to the way the standard C library provides the *atexit*() function[3].

The use of a generic constructor to classify & build objects from a data stream, and the use of virtual functions on these objects to hide the details of assembly from the assembly application are patterns in their own right.

---

[3]. Though the need to register multiple handlers can be made unecessary by defining a handler object that peforms some arbitray list of functions on the complete unit of work.