Organizational Multiplexing: Patterns for Processing Satellite Telemetry with Distributed Teams

Stephen P. Berczuk
MIT Center for Space Research

Organizational issues play a significant role in many software architectures, though often more as a side effect than by design. This paper presents a pattern language for developing ground software for satellite telemetry systems that illustrates how to take organization into account in the architecture. In particular, this paper addresses:

- extending a pattern from the 1995 PLoP conference [Berczuk95] into a pattern language.
- assembling patterns from other pattern languages into a domain specific pattern language, and
- including social context as a motivating context for a pattern.

Since scientific research satellite systems are often developed at academic institutions, where similarities between ongoing projects exist only at the architectural level (code typically cannot be reused), documenting architectural insights such as these in the pattern form can be a very practical way to achieve reuse. Since assembly of telemetry involves (re)creation of objects from a serial stream, these patterns are relevant to designers of any systems which creates objects from a serial stream

Introduction

Organizational issues have an impact on the development of software systems. It is important to design an architecture in a way that fits well with existing social and organizational constraints.

As an example of how these issues can be addressed, this paper presents a pattern language to guide the development of a ground based system that will process telemetry data from an earth-orbiting astronomical observatory. Systems of this type involve many diverse, and often geographically distributed groups of people; such a system is challenging to design from a social as well as a technical point of view.[1] The patterns present in this system are applicable to other systems that have similar organizational constraints. Since telemetry can be considered a form of persistence, where objects are serialized into a stream and transmitted, some of these patterns will also be of interest to builders of systems that use parsing and reconstruction of objects from a serial data stream.

---

[1] The patterns in this paper might also be of use to designers in other situations where development teams are distributed.

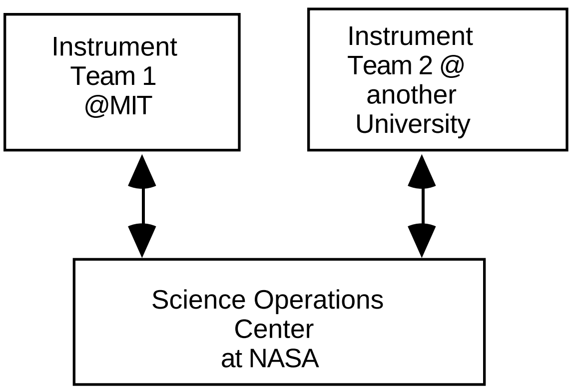
This pattern language makes use of Coplien's patterns of organization [Coplien95][2] as context for the other patterns, and also shows how to effect some of these patterns in the context of a ground based scientific telemetry processing system.

After a description of the relevant elements of the organizatonal structure of a typical project, patterns which address some of the issues raised by this structure will be described.

Project Organization

Typical Project Organization

Most of the projects being pursued at the Center For Space Research at MIT share an organization similar to the following:[3]



- A number of *instrument teams* each with primary responsibility for deployment and analysis of the data from a single scientific instrument that will be on the satellite. Often these teams will be at geographically separate locations.
- A central organization (the *operations center*) which performs a coordination function. The operations center may not have the expertise to understand the scientific objectives of each of the instrument teams,

---

[2]To facilitate understanding of paper copies of this document, the appendix describes summarizes the patterns from [Coplien95] [Gamma+94] and [Beck] referenced in this paper.

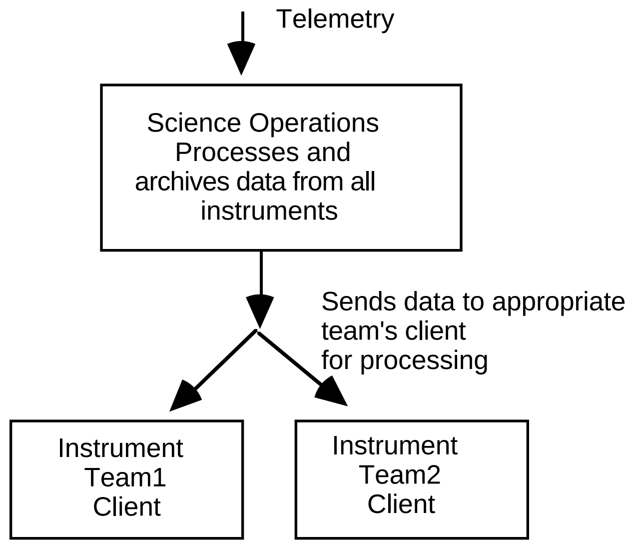
[3]This particular structure is based on that of the X-ray Timing Explorer project (XTE), currently in progress at the MIT Center for Space Research.

and is focused on the operational aspects of the system (processing so many bytes of telemetry per second, and archiving data, for example).

- A small base of shared knowledge between instrument teams and operations center.
- Small work groups, particularly at the instrument team locations. At each team site, many of the subsystems fulfill the context of *Solo Virtuoso* [Coplien95].

The architecture of the data analysis system takes the following general form:

Typical Architecture



The operations center depends on the instrument teams for providing details about telemetry specific to each instrument.

Each of the teams has very different motivations; the instrument teams are focused on getting the best science from "their" instruments, and the operations team is focused on getting the system assembled (with science often a secondary consideration). The teams have a large degree of autonomy, but they must agree on certain interfaces. As a result, a minimal amount of coupling is best[4]. A major challenge in designing an architecture of such a system is to handle the

---

[4]It can be argued that minimizing coupling is an important consideration in building any system, but because of the degree of decentralization and non locality in the projects being discussed here, it is especially important.

organizational and political issues that arise, in such a way that individual organizations are not overly affected by external forces.

The general organizational forces involved in scientific satellite telemetry processing systems can be summarized as follows:

- Distributed experience
- Small teams, distributed geographically.
- Little carryover of personnel from one project to the next.

This paper focuses on the process of classification and interpretation of the telemetry packets as they are received from the spacecraft, and dispatching the resulting data objects for further processing. Other patterns can be written to guide the development of other aspects of the system such as commanding and data analysis.

While this paper discusses patterns found in a scientific satellite application, the patterns have applications to other domains as well.

The Patterns

The architectural goals and the patterns that complete them are as follows:

- Facilitate autonomous development: *Loose Interfaces*
- Interpret a Data Stream: *Parser/Builder*
- Divide responsibilities for interpretation: *Hierarchy of Factories*
- Connect Systems: *Handlers*

1. Loose Interfaces

Context: To help development of a system with many teams proceeding at a reasonable pace it is important to keep interfaces between systems loose. This is particularly important in a situation where there are teams of developers that are geographically distributed and a situation where rapid turnaround time for design and development is important.

Problem: Communication is difficult. If requirements are changing and the teams are located in a variety of places then the poor communication can stall a project. This can be particularly problematic when an organization does not have an architectural center, such as specified by *Architect Controls Product* [Coplien95].

This is particularly applicable in a research satellite application where teams are small, requirements are changing, and the potential for gridlock is great if dependencies are too high. The operations center is the organizational center of the architecture, but does not always have the capability to design a complete system. To avoid development bottlenecks, we need to be able to limit the effect one team's work will have on another.

Therefore: Limit the number of explicit, static, interfaces. Use loose interfaces like *Callback* and *Parser/Builder* and *Hierarchy of Factories* to achieve this.

Decoupling interfaces in this way will also simplify the development of *Early Program* [Beck], since it provides a mechanism for building incremental systems. It can also facilitate implementation of *Developer Controls Process* [Coplien95], by making it easy to define features that can be controlled by a developer or group.

2. Parser/Builder

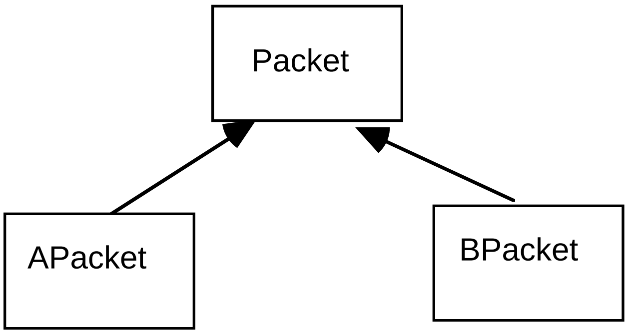
Context: Many systems need to read data from a stream and classify elements on the steam as objects. A way is needed to create arbitrary objects based on tokens in the data stream.

Problem: Given a data stream, we want to interpret it, classifying the elements into the appropriate class of object. The data stream contains tags that can be used to identify the raw data, and we want to convert the stream into object form, so we can process the data.

As a non-telemetry related example consider the problem of reading in raw UNIX files and classifying them into types of files based on their "magic number" —as in the tags in the /etc/magic file. You could create the appropriate subclass of *File* and then invoke its virtual edit() method, bringing up the appropriate editor.

In a telemetry processing system each telemetry packet has identifying information in its header. The telemetry processing system design requires that an object, once created, knows how to process itself (i.e., we will not use a dispatch table, or a switch on type— this is to satisfy the *Organization Follows Location*[Coplien95] pattern). At the lowest level objects will be created using a *Factory Method* [Gamma+94]. Each class of packets will be processed differently; some will assemble themselves into larger units, others will issue messages. Consider the following hierarchy, as shown in figure 1, for a spacecraft that there are two subclasses of *Packet*:*APackets* and *BPackets*.

Sample Packet Hierarchy



We want each *Packet*, once created, to process itself by using a virtual method, *process()*. If we pass a data stream into a factory, we want to return a pointer to a *Packet* that has the appropriate type. To summarize the forces:

- There is a need to interpret a raw data stream.
- There is a generic way to process the packets once they are returned from the factory.
- The raw data contain tags which can be used for classification.

Therefore: to resolve these forces use a *Parser/Builder* which reads the identifying information from the header of the packet, and creates an object of the appropriate type, removing only one object's worth of data from the stream.

An example of a client interface is:

```
while (!dataStream.empty()) {
      PacketFactory f;
      Packet* p = f.make(dataStream);
      if(p) p->process()
}
```

This is a variant of *Abstract Factory*[Gamma+94] but the object to be created is defined in the data stream, rather than by the client. *Factories* and *Parser/Builder*s can be used to in part implement *Loose Interfaces* by providing a means of separating clients from producers of data (assuming that data producers also define the factories).

Other uses:

In some object persistence mechanisms, objects are assigned class Id's which are placed in the storage stream. These Ids are restored first to allow the system decide what class object to make from the restored stream.

Parser/Builder is used in in the pattern *Query Objects* [5][Chasms] to convert SQL statements to QUERY objects. [Riehle96] discusses similar issues, building objects on a desktop using specifications.

The distinction between this pattern and *Builder* [Gamma+94] and *Factory Method* [Gamma+94] is that in this pattern the factory reads from a stream and the client does not know which type of object will be returned. For text interpretation, *Parser/Builder* can be a front end to the *Interpreter* [Gamma+94] pattern.

3.Hierarchy of Factories

Also Known As: *Composite Factory*

Context: Once we decide that the *Parser/Builder* is the right way to create objects, we need to partition the details of how to construct objects of various classes into the various groups responsible for this construction, in other words we need to have *Loose Interfaces*. We want to complete *Form Follows Function* [Coplien95] or *Organization Follows Location* [Coplien95]. On a lower level we want to implement *Developer Controls Process* [Coplien95] for a system which creates objects of various types.

Problem: In a distributed work group it is important to divide responsibilities as cleanly as possible and reduce coupling. There should be a way to do this in a creational system.

Sometimes the secrets of classifying elements in a data stream are divided between various groups. The reasons for this partitioning can involve company politics, or simply that the knowledge of the telemetry formats is distributed and there is a strong desire to reduce coupling. We need a way to partition the

[5]*Query Objects* addresses the problem of handling the generation and executation of SQL statements in an object-oriented way, when you are trying to use a relational database to store objects.

responsibilities for classifying the telemetry packets, while maintaining a centralized client interface.

In a telemetry application, various instruments can generate telemetry which is then fed into one stream. The instruments are developed by different teams (at different institutions, for example), and these teams have control over the format of the telemetry that they generate (after taking some standard headers into account).

We want a way to isolate the details for identifying each team's objects, while at the same time allowing the objects to be identified and created in a single application. The scheme that we develop should be layered so that the main factory needs to know only of the existence of a class of objects, but need not know how deep the hierarchy below that class is. Packets created from the hierarchy are processed in a generic way, perhaps by using virtual functions.

One way to address the classification problem is to put all the classification/dispatch logic into a single *Parser/Builder* (combining the *Interpreter* [Gamma+94] pattern with a *Builder* [Gamma+94]) — perhaps by using a big switch statement— and rely on communications between groups to ensure that the details make it into the master code through some communications method. This is error prone, and subject to delays. We could also divide the processing into a number of factories and have the client call each in turn. This violates our requirement of transparency, and the client needs to know when a new class of object is added.

It would be useful to have a way to have the client interface emulate a single Factory, but hide the details of the construction hierarchy.

To summarize the forces:

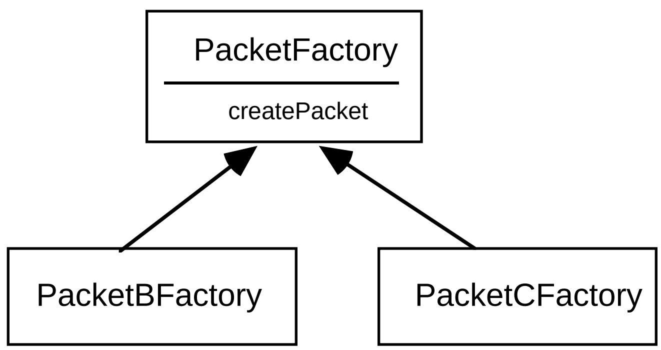
- Division of responsibilities (*Organization Follows Location*).
- A need for a central interface for parsing data streams and building objects.
- A need to add objects to the construction hierarchy in a manner transparent to clients.
- The ability (or requirement) to process entities by virtual functions.
- Each class of object can know about its immediate derived classes.

Therefore: Use a hierarchy of factories, each of which understands the criteria for making a packet of its type, and knows about the immediate subtypes. The client invokes the make method with the base class factory instance. That factory checks to see that there is indeed an object of class packet in the stream, based on some attributes. The factory then passes the data stream to the factories of each of its immediate subclasses, which check the appropriate data fields in the manner of the *Parser/Builder* pattern.

The *Singleton* pattern [Gamma+94] can be used to access the factories for the derived classes, or the members of the hierarchy can be registered with the master factory at run time.

While this pattern violates encapsulation to some extent by requiring that a base class know about its immediate subclasses, it can be made acceptable by agreeing on generic interface classes (say, one per team) and allow each team free reign to subclass these interface classes. Also in this application this requirement is not terribly limiting, since the top level operations team knows abou the basic instrument team interfaces and the number of instrument teams is fixed by contract when the project begins.

Hierarchy of Factories



An example implementation is:

```
//Base class factory method
      Packet* PacketFactory::make(Stream* dataStream){
       Packet* pkt=0;
      if(isAPacket(dataStream) {
      if(! pkt = APacket::factory()->make(dataStream))
             if(!pkt    =    BPacket::factory()-
>make(dataStream)){
        pkt = new Packet(dataStream);
      }
      return pkt;
      }
```

The result of applying this pattern is that each class needs to know only:

- The criteria for what constitutes a member of that class in terms of elements in the data stream.
- The immediate subclasses.

It is possible to use a *Registration* [6] mechanism to inform the base class of what the subclasses are rather than hard coding the relationship. It is also possible to implement this pattern using containment rather than inheritance.

Other uses:

This pattern is also useful for isolating the definition of packets for which a single team is responsible, so the information can be encapsulated, making it easier to work on a project with large or widely distributed teams.

Related Patterns

This is similar to the *Builder* [Gamma+94] pattern in that it has a hierarchy of "factories." It is different in that the data stream defines what is made rather than the application explicitly specifying what objects to construct by argumenst to the factory.

This pattern helps us realize *Organization Follows Location* and *Code Ownership* [Coplien95]

4. Handlers [7]

Context: This pattern provides for decoupling needed to implement *Organization Follows Location* [Coplien95], for the products of a creational system. After we assemble packets from the telemetry stream we process them, generating data products. We need a way to direct the processing of these new data products.

---

[6] This pattern is not yet written, but would specify a mechanism for notifying a base class factory that a derived class factory has been created. The basic idea would be similar to the View/Model connection in a Model/View/Controller mechanism, but would also address issues of uniqueness (only one instance of each derived class can notify a base class) and guaranteed notification (the construction of any object/factory of the derived classes would generate a registration event automatically).

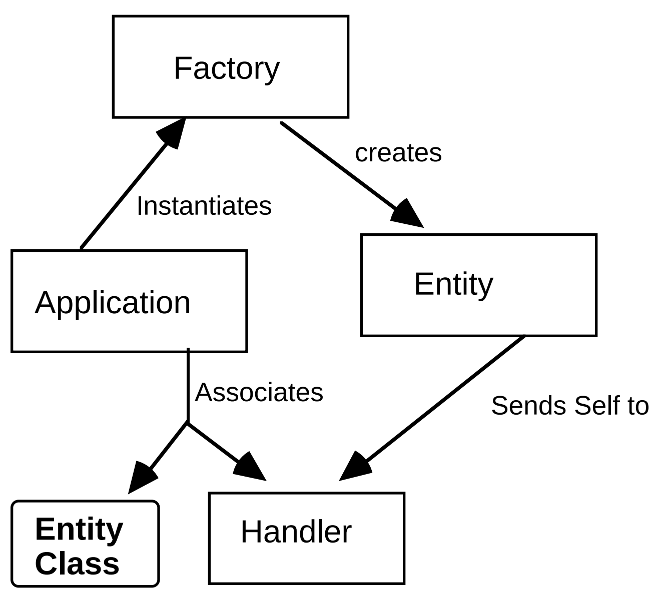
[7] Adapted from PLoP 94 paper. See [Berczuk95] for details.

Problem: In an environment where components developed by separate teams with different specific goals must inter-operate, it is necessary to partition responsibilities in such a way that dependencies can be reduced while interoperability is maintained. In particular well defined portions of the system should be isolated fron the to be specified pieces. This may be particularly important if the teams are geographically distributed.

To summarize the forces at work:

- The requirements for the end to end system are not complely specified.
- Requirements for one component of a system need to be available before downstream processing is defined.
- Upstream and downstream components will be demonstrated/tested at different times.
- Upstream components should know nothing of the downstream processing.

Callback Pattern



Solution: Use a callback mechanism to define connections between assembly process and the processing process. Provide a mechanism (*Registration*) to assign a *Handler* object for which a completed entity will be forwarded.

One implementation uses a static *Handler* data member for class *Unit*. Whenever a *Unit* is completed it hands itself to the *Handler* for processing.

The issue of when to subclass and when to differentiate objects by an attribute can be a confusing one, and can only be resolved by examing the specific details such as the number of classes, and the type of behaviour.

This pattern is similar to *Observer* [Gamma+94] but differs in that the "observer" is the class of object being created, and the event that triggers the notification is the creation of an object of a class.

Conclusions

Organizational issues play a significant role in determining the direction a software system can take. They affect the context in which a system is developed. It is also necessary to architect a system to minimize the negative effects of organizational issues. When the organizational patterns such as those in [Coplien95] exist, there should be ways to architect a system to aid the realization of good organizational patterns. This paper illustrates some of the ways to reflect organizational issues in the context of other patterns as well as showing some ways to implement good organizational patterns in a software architecture.

There is a common structure to scientific satellite applications, and there also tends to be little carryover of personnel between projects, so many of the lessons learned must be rediscovered. Since there are different hardware and software platforms from one project to the next and the details of each project vary greatly, *code reuse* is not really feasible. *Design elements* can be reused, however. This application domain, and others which share these factors, could benefit greatly from documentation of architectural principles as patterns.

While the patterns explicitly address scientific spacecraft satellite telemetry, the ideas in this paper can be applicable to any system being built with a number of distinct teams. Some of the patterns such as *Parser/Builder* also have applicability in other creational systems.

References

[Berczuk95] Berczuk, Steve, "Handlers for Separating Assembly and Processing." In *Pattern Languages of Program Design*. James O. Coplien & Douglas Schmidt Eds. Reading, MA: Addison Wesley, 1995.

[Coplien95] Coplien, James. "A Development Process Generative Pattern Language." In *Pattern Languages of Program Design*. James O. Coplien & Douglas Schmidt Eds. Reading, MA: Addison Wesley, 1995.

[Gamma+94] Eric Gamma, Ralph Johnson, Richard Helm & John Vlissides, *Design Patterns: Elements of Object-Oriented Software Architecture*. Addison Wesley, 1994.

[Beck] Beck, Kent. *Early Development Patterns.* Portland Patterns Repository. URL: http://c2.com/ppr/early.html.

[Chasms] Brown, Kyle and Whitenack, Bruce. "Crossing Chasms– A Pattern Language for Object-RDBMS Integration." In *Pattern Languages of Program Design*., Volume II John Vlissides, Norm Kerth, and James O. Coplien Eds. Reading, MA: Addison Wesley, 1996.

[Riehle96] Riehle, Dirk. "Patterns for Encapsulating Class Trees." In *Pattern Languages of Program Design*., Volume II John Vlissides, Norm Kerth, and James O. Coplien Eds. Reading, MA: Addison Wesley, 1996.

Acknowledgements

Thanks to Dirk Riehle, Doug Lea and Lena Davis for reviewing early and final drafts, and the members of the PLoP working group for making many useful suggestions. The IM Pei home group at PLoP 95 made helpful suggestions on how to reference other patterns, and the Appendix is based on their ideas. This work was supported in part by NASA/GSFC contract number NAS5-30612.

Authors address: Room 37-561; 77 Massachusetts Ave, Cambridge MA 02139. current email address; berczuk@acm.org.

Appendix: Brief Descriptions of Referenced Patterns

This appendix briefly describes the intent of the referenced patterns. These summaries are quite brief and the orginal pattern should be consulted to fully understand it.

Patterns from "A Development Process Generative Pattern Language" [Coplien95].

These summaries are an adaptation of the work of the IM Pei group at PLoP 95.

Architect Controls Product. Describes how a central architect in a controlling role can help build a cohesive product.

Code Ownership. Modules are assigned to developers who control then and typically make all changes. This simplifies the task of keeping up with the details of the entire system.

Developer Controls Process. . Use a developer as a focal point for the project communication and process for the development of a given feature.

Form Follows Function. . In a project lacking well-defined roles, group closely related activities (those coupled by either implementation, manipulation of the same artifacts, or domain related) together.

Organization Follows Location. When assigning tasks arcoss a geographically distributed workforce, assign architectural responsibilities in such a way that decisions can be made locally.

Solo Virtuoso. Design and implementation of a small software product which is required in a short time should be done by one or two people.

Patters from *Design Patterns* [Gamma+94].

These summaries are from the intent summaries on the inside cover of [Gamma+94], and are repeated here for reference.

Abstract Factory. Provide an interface for creating families of related or deoendent objects without specifying their concrete classes.

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

Factory Method. Define an interface for creating an object but let classes decide which classes to instantiate.

Interpreter. Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Singleton. Ensure a class only has one instance, and provide a global point of access to it.

Patterns from Early Development Patterns [Beck]

Early Program. Build concrete software early on that shows how the system works and fulfills specified scenarios.