Using XML Configuration-Driven Development to Create a Customizable Ground Data System

Brent Nash, Martha DeMore
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
bnash@jpl.nasa.gov, mdemore@jpl.nasa.gov

Abstract—[1][2]The Mission data Processing and Control Subsystem (MPCS) is being developed as a multi-mission Ground Data System with the Mars Science Laboratory (MSL) as the first fully supported mission. MPCS is a fully featured, Java-based Ground Data System (GDS) for telecommand and telemetry processing based on Configuration-Driven Development (CDD). The eXtensible Markup Language (XML) is the ideal language for CDD because it is easily readable and editable by all levels of users and is also backed by a World Wide Web Consortium (W3C) standard and numerous powerful processing tools that make it uniquely flexible. The CDD approach adopted by MPCS minimizes changes to compiled code by using XML to create a series of configuration files that provide both coarse and fine grained control over all aspects of GDS operation.

The MPCS development team has implemented a generic, hierarchical architecture for specification of and access to system configuration information that allows configuration parameters to be specified at the system, mission, and user levels, transparent to the application that employs the information. The resulting implementation is an XML-based design, implemented in Java and Python, which is useful not only for MPCS or ground data systems in particular, but to any application.

While using XML-based CDD allows MPCS to have an exoteric functional interface that can be easily reconfigured (for different mission phases or even different missions) at runtime instead of compile time, the challenges of developing a sufficiently flexible configuration are significant. In order to create a reusable multi-mission GDS, it is necessary to balance the added complexity of developing configuration-driven code with the ability to create an overall configuration strategy that both developers and users can understand and utilize effectively.

This paper will discuss the configuration and development strategies employed by the MPCS development team and the associated lessons learned in developing an XML configuration-driven GDS to be used for the life of the MSL mission and future missions.

1
[1] Copyright 2008 California Institute of Technology. Government sponsorship acknowledged.
[2] IEEEAC paper#1256, Version 1, Updated Oct. 26, 2008

TABLE OF CONTENTS

1. INTRODUCTION 1
2. CONFIGURATION-DRIVEN DEVELOPMENT (CDD) 2
3. MPCS AND CDD 4
4. CONCLUSIONS 9
5. ACKNOWLEDGEMENTS 10
REFERENCES 10
BIOGRAPHY 10

1. INTRODUCTION

The Mission data Processing and Control Subsystem (MPCS) is the next-generation Ground Data System (GDS) under development for the Mars Science Laboratory (MSL). In addition to being a large component of the ground system for MSL, MPCS is also in varying degrees of use on the Mars Exploration Rovers (MER), the Diviner instrument on the Lunar Reconnaissance Orbiter (LRO), and the DAWN mission as well as other internal projects.

MPCS is being developed as a multi-mission ground system that will experience its first operational use on MSL, but will be easily adaptable to future missions. MPCS is responsible for the reliable interpretation, real-time distribution, and archival of spacecraft information for a wide range of science and operations customers. This information includes spacecraft telecommand data ("uplink") and spacecraft telemetry ("downlink"). For MSL, MPCS has two distinct targets for telecommanding and receiving telemetry:

- Flight Software (FSW) – The software interface into the flight hardware onboard the MSL vehicle itself.
- Simulation and Support Equipment (SSE) – The software interface into the Ground Support Equipment (GSE) that is part of the mission lifecycle up until the launch.

MPCS is unique in that it will be the first GDS to be used throughout the entire mission lifecycle of FSW development, test, and operations. MPCS is responsible for handling spacecraft telecommand and telemetry in a variety of different venues including:

- WorkStation TestSet (WSTS) – Individual FSW developers and other spacecraft team testers work on their individual development workstations sends commands to and receiving telemetry from a flight system emulator.

- Mission Testbeds – FSW Developers and spacecraft team testers do integration, verification, and validation procedures with various subcomponents of the actual flight system hardware.

- Assembly, Test and Launch Operations (ATLO) – The FSW and GDS systems are integrated with the assembled flight hardware in a flight-like test environment.

- Operations – The various phases of the operational mission from launch to the end of life of the mission.[3]

Each venue provides its own adaptation challenges as many of the Application Programming Interfaces (APIs) and transmitted data formats change based on the venue that is being used.

The major challenge for MPCS is providing a single software package that can be flexible enough to support all of the different venues in all different phases of a mission. Furthermore, MPCS not only has to handle the complex requirements of the various phases of the MSL mission, but it must be agile enough to be easily adaptable to future missions.

In tackling these significant software development challenges, MPCS has adopted an XML-based Configuration Driven Development (CDD) approach for writing object-oriented software, which has been extremely useful in providing a product flexible enough to be used in a variety of scenarios and agile enough to quickly adapt to new requirements and new missions. The following sections will first describe CDD as an abstract object-oriented software design methodology and will then delve into how MPCS has used CDD to meet its customers' needs.

2. Configuration-Driven Development (CDD)

Overview

The following section provides an overview of the benefits of adopting a CDD-based approach to software development and then discusses how to use CDD including how to choose a configuration representation, what information should be placed in the configuration and how code should be written to best leverage the configuration.

Why CDD?

As the name implies, CDD places the primary focus of software development on configuration written outside of the code. Just as models are built first in Model-Driven Development or tests are written first in Test-Driven Development, CDD involves writing basic configuration file structure and content before developing the business code that utilizes the configuration. By writing configuration first, it forces developers to focus on what information and business logic can be represented independent of the code that will operate on it. It also forces developers to determine if any pertinent information is already present in existing configuration and thereby cuts down on duplication of information and processing logic. Another benefit of CDD is that configuration files are easy to manage from a Configuration Management (CM) standpoint because they can be source-controlled and stored in a versioning system such as Subversion or CVS in the same way that source code is managed. CDD is not tied to one specific programming language either; because configuration is represented in a neutral format, the same configuration files can be used across a number of different programming languages. The biggest advantage of CDD is that the behavior of the software may be changed drastically without having to recompile or redeliver code.

Configuration Strategies

The first major choice to make when adopting a CDD approach is deciding how configuration files will be stored and represented. There is no single configuration representation used unilaterally across CDD-based projects, but the most popular choices are:

- Environment Variables – Configuration information is identified and given values at the Operating System (OS) level. Environment variables are useful in making the same information available to a number of separate processes, but can cause consistency issues between different users, shells, and environments.

- Homegrown Formats – Configuration information is specified in a format created by the developers who have complete control over the format and content restrictions of the configuration. The main disadvantage of a homegrown configuration format is that the parsing and validation business logic must generally be written from scratch, adding to development time. In addition, the format is difficult to share with other development teams and systems when integration becomes necessary.

2

[3] MPCS will not be used for telecomand in the MSL Operations phase.

- Keyword-Value Format – Configuration information is identified in a simple *name=value* format, generally one entry per line, in a configuration file. Keyword value formats are generally very easy to read, write, and interpret and most programming languages have existing parsers for keyword value formats. The main disadvantage is that keyword-value is a loosely defined format and it can be difficult to enforce organizational restrictions or validate the values of particular configuration entries.

- XML Format – Configuration information is identified in a World-Wide Web Consortium (W3C) standard language that has a well-understood format and wide user base. XML writing and parsing software is widely available and XML additionally provides the ability to do schema-based validation that can be further used to restrict configuration format and even restrict values on particular entries. The main disadvantage is that XML is a verbose language and can be confusing and intimidating to users who are not software developers or are unfamiliar with XML.

The choice of a configuration representation format is very important because it will affect the entire future of the CDD-based project. Once a configuration format is chosen, it can be very difficult and costly to change formats further down the road. Once the choice has been made on what configuration representation format to use, the next issue is deciding what should actually be placed in the configuration.

What To Configure

One of the most difficult parts of CDD is determining what information should be made configurable and what should be implemented in code. There are classes of information that should be made into configuration values:

- Common Information – Information that is repeated in multiple places in the code and has a nonzero chance of needing to change at some point in the software's lifetime. By pointing all code to a single location for a common piece of information, developers ensure that the software's behavior will remain consistent when the desired information needs to change.

- Default Information – Information that represents part of the standard out-of-the-box behavior of the software. By pulling default values from a configuration file, the out-of-the-box functionality of the software may be adjusted with little effort.

- Unstable Information – Information that is unknown or in flux at the time of development. This information that has a high likelihood of changing once the software has been delivered.

- Configuration-Dependent Information – Information that is dependent on another piece of configurable information.

In addition, there are classes of information that generally should not be configurable:

- Static Information – Information that has no likelihood of changing throughout the life of the software. Adding values that will never change to the configuration clutters the configuration while providing no tangible benefit.

- Compound Information – Information whose value can be completely determined based on other information already represented in the configuration. This information can easily be represented in software whereas representing it in configuration can lead to conflicting configuration values.

- Complex Logic – Configuration values are information that can be used in code to alter basic software flow and operation, but it is not the job of configuration to provide any type of logical assertions that would normally be written in code.

Unfortunately, not all information falls nicely into one of the described categories and in many cases CDD becomes a process of trial and error to determine what works well as configuration and what should remain written in code. As developers become more comfortable with CDD, the distinction between what should be configurable and what should not becomes more clear. In CDD, always err on the side of configurability rather than hard-coding information into code.

In conjunction with deciding what information should be made configurable, the development team must also make the decision of how configuration should be stored, represented and accessed from within the software itself. Though CDD can be used in any software development environment, the focus of the following discussion will be on object-oriented software design.

Software Design Patterns for CDD

As object-oriented programming languages have grown in popularity, a number of useful development paradigms, known as "design patterns", for solving recurring object-oriented design problems have arisen. In CDD, external configuration files will grow in size over the life of the project and the configuration information they contain will be needed in many different places in the code, so it is important to implement the configuration representation in

software in a way that is both efficient and scalable. There is an infinite number of ways to represent configuration in code, but some of the most popular strategies for configuration are described by the following design patterns:

- Singleton – Read the configuration into a single shared object that is globally accessible to all other objects.
- Dependency Injection – Read the configuration into one or more configuration objects and hand a configuration object to any other object that needs it.
- Monostate/Borg – Read the configuration into an internal state that is shared between a set of globally accessible configuration objects that behave as though they were a single object.

Using any of these design patterns will provide a uniform method of accessing configuration information in code. Representing the configuration in code is the core of CDD, but there are also other design patterns, such as the Abstract Factory (a.k.a. Kit) that harmonize well with the use of a CDD strategy.

An Abstract Factory pattern provides an interface for creating sets of objects without specifying their concrete classes. The power of the Abstract Factory design pattern in object-oriented languages such as Java is based on the use of reflective programming. Many popular object-oriented programming languages, such as C++, C#, Smalltalk, Java, Python, and Ruby, support a reflective-programming interface that allows objects to be dynamically instantiated at runtime by providing information to identify the object (e.g. the name of the object's class) rather than directly instantiating the object itself. By supplying reflection information in the configuration, reflective programming can dynamically determine what objects to instantiate at runtime based on configuration information rather than hard-coding the instantiation of one particular type of object. With an Abstract Factory pattern, a calling object can pass in the relevant information, the factory can look up the name of the class to instantiate in configuration, and then using reflection the requested object can be instantiated dynamically and handed back to the caller. CDD can be used without ever using design patterns, but by mixing CDD with creational design patterns like the Abstract Factory developers can achieve a new level of flexibility.

A CDD strategy was combined with an XML-based configuration solution and well-known object-oriented design patterns to create the MPCS GDS, the next generation GDS for MSL.

3. MPCS AND CDD

Overview

The following section provides an overview of the MPCS software system by explaining the MPCS data flow, providing an overview of the MPCS software architecture, describing the MPCS configuration file format and structure, and then discussing how the MPCS software architecture uses a configuration-oriented approach to achieve added flexibility. Though MPCS is used in both uplink and downlink scenarios, the following discussion will focus solely on the MPCS downlink telemetry flow.

MPCS Input Stream

Spacecraft produce a great deal of telemetry with varying characteristics according to the type of mission and stage in the mission lifecycle. Data is generally transmitted from the spacecraft via transfer frames that are encoded to prevent data corruption during transmission. Transfer frames contain data packets that in turn contain varying types of telemetry information. Each packet has an Application Identifier (APID) that specifies what type of telemetry it contains. There are three major types of telemetry read from packets and processed by MPCS:

- Engineering, Housekeeping, and Analysis (EHA) – Also known as "channels", EHA data describes measurements of hardware and software state (e.g. attitude pointing, temperature readings, etc.).
- Event Records (EVRs) – EVR data describes both expected and unexpected events that have occurred onboard.
- Data Products – Products are files containing science or engineering data such as images and instrument measurements.

MPCS receives either transfer frames or packets, in various forms, as input and it is the responsibility of MPCS to extract and interpret the telemetry items within the frames and packets and make them available to end-users. MPCS provides a realtime telemetry interface via a Java Messaging Service (JMS) bus and a historical telemetry interface via a relational database. In addition, constructed data products are stored on a Network File System (NFS) disk to make them available to users. MPCS performs many other functions during data processing, such as marking EHA channels with alarms to indicate potential anomalies, but these more in-depth functions are outside the scope of this discussion.

The interpretation of data flowing into MPCS happens in a pipelined fashion. First, frames go through synchronization to remove encoding and spurious data. Then, packets are

extracted from frames. Next, telemetry items, such as EVRs, EHA channels, and data products, are constructed from packet contents. Finally, telemetry items are made available to end-users via the realtime JMS bus, the historical relational database, and the networked filesystem.

MPCS Sessions

As a tool that is used throughout every stage of the mission lifecycle from low-level tests through operations, MPCS had to be built with data retention and archival in mind. All tests run with MPCS in the testbed and ATLO venues, as well as some tests in the WSTS venue, are stored in relational databases so that past tests may be re-analyzed and re-run in the future. The MPCS development team devised the notion of a "session" to store all of the specific information that would be needed to recreate a given test (e.g. telemetry dictionaries used, host where the test was run, etc.). In addition, an MPCS session also contains metadata such as a test name and unique test identifier that allow users to easily track down past sessions. At the beginning of every run of the MPCS software, users supply session information that MPCS uses for a variety of purposes including determining what venue (e.g. WSTS, testbed, ATLO) it is running in.

MPCS Architecture

The structure of the internal MPCS software architecture was designed to handle the pipelined processing flow of spacecraft telemetry. MPCS is built around a series of pipelined adapters that handle the various stages in telemetry processing and communicate with one another via a high-speed intra-process message bus that employs a publish/subscribe communication model between Java objects. The general telemetry processing architecture of MCPS is shown in Figure 1.

Figure 1 demonstrates the entire MPCS downlink flow from telemetry receipt to data availability via the JMS bus, relational database, and filesystem. The Raw Input Adapter is responsible for reliably receiving input telemetry and publishing it on the internal bus to be consumed. The Frame Sync Adapter performs frame synchronization on input transfer frames (if necessary). The Packet Extract adapter ingests transfer frames and extracts the packets contained in the frames (if necessary). The EVR Adapter, EHA Adapter, and Data Product Adapter all ingest packets and extract the type of information they're interested in (based on the packet APID) to construct EVRs, EHA channel values, and data products respectively. The data product adapter also writes data product files to the filesystem as they are constructed. The Database Adapter stores all relevant information in the relational database. Similarly, the JMS

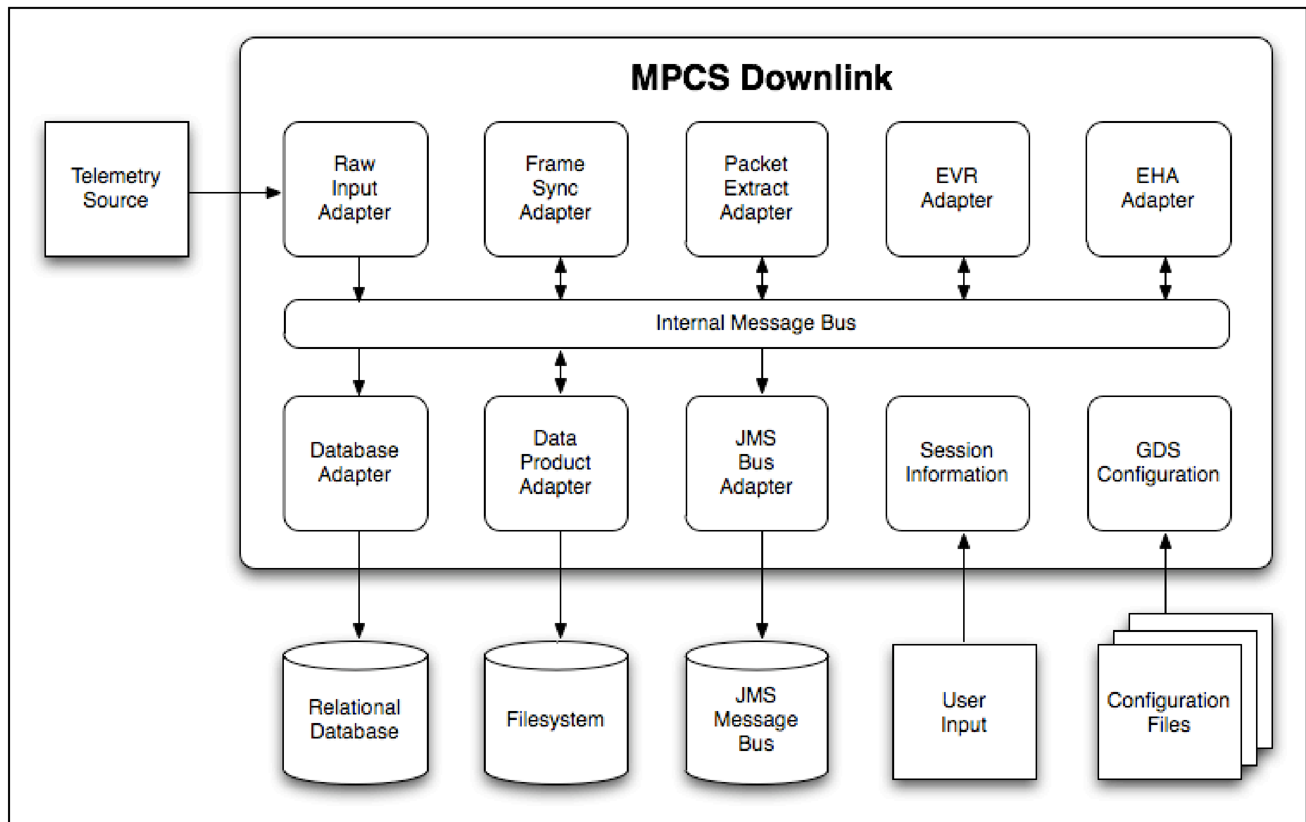

Figure 1 – MPCS Downlink Architecture

Bus Adapter publishes all relevant information to the JMS Message Bus[4]. The Session Information and GDS Configuration are read on startup and made globally available to all of the adapters in the downlink processing flow.

Adapters communicate with one another by publishing and subscribing to messages via the internal message bus. Adapters only communicate through the internal bus, they never communicate directly between one another. In this fashion, each adapter can act as an independent entity and function with no knowledge of what other adapters have been loaded as a part of the downlink processing stream. The set of adapters that are loaded each time MPCS is run is determined by three factors:

(1) The mission being run (e.g. MSL).

(2) The venue specified by the user in the session information (e.g. ATLO).

(3) Special override values in the MPCS configuration.

Using these three pieces of information, MPCS can dynamically determine what adapters are needed to process telemetry in the current test or operational scenario.

When adapting MPCS to do telemetry processing for a specific mission, the MPCS team must determine what adapters are needed for processing the new mission's telemetry. Some functionality, such as packet extraction, is based on a Consultative Committee for Space Data Systems (CCSDS) standard and will likely not change from one mission to the next. Processing of data such as EVRs and EHA, however, is very mission-dependent and it is highly likely that each mission will need its own custom adapters for processing these telemetry items. By using the notion of adapters to partition telemetry processing into independent pieces, MPCS can quickly adapt to new missions by simply building the necessary new adapters. If a new mission has very similar telemetry to an existing mission already supported by MPCS, the adaptation cost is minimal.

One of the major benefits of using a CDD approach in MPCS is that MPCS can use configuration to determine what adapters should be instantiated for each mission and each venue. In addition, each adapter has access to all of the MPCS configuration information, so the internal behavior of the adapters themselves can also be modified based on configuration entries. The following sections will describe the configuration strategy adopted by MPCS and demonstrate how the MPCS configuration is used at the object-oriented design level to dynamically affect MPCS telemetry processing behavior.

6

[4] Currently only telemetry items such as EVRs, EHA, and Data Products are published to the realtime bus. Frames and packets are not published.

MPCS Configuration Format

The MPCS software architecture is built around a set of XML configuration files. The MPCS team chose XML because of its well-structured format and its human-readability. It was the belief of the MPCS team that if an intuitive XML layout structure could be defined for the configuration, individual end-users would be able to easily edit parts of the configuration to customize their use of MPCS.

The actual structure of the MPCS configuration was designed so that related configuration values could be organized into different subsections in the configuration structure itself. The minimalist schema for the MPCS configuration defines only three types of configuration elements:

- Configuration Block – Used to define a subgroup in the configuration with no associated value.
- Configuration Value – Used to define a configuration entry with a single value that can be read from the configuration and used in code.
- Configuration List – Used to define a configuration entry with a set of values that can be read from the configuration and used in code.

Each type of configuration element is given a unique name to identify it in the context of the global configuration. A small sample MPCS configuration file is shown in Figure 2 below.

```
<GdsConfiguration>

  <config name=”propertyName”>value1</config>

  <configList name=”listPropertyName”>

    <listItem>value2</listItem>

    <listItem>value3</listItem>

  </configList>

  <configBlock name=”blockName”>

    <config name=”nestedPropertyName”>value4</config>

  </configBlock>

  <configBlock name=”outer”>

    <configBlock name=”inner”>

      <config name=”myProperty”>value5</config>
```

```
    </configBlock>

  </configBlock>

</GdsConfiguration>
```

Figure 2 – Sample MPCS Configuration File

The MPCS configuration file organization, shown in Figure 2, uses <configBlock> elements to group related configuration values together and uses <configList> and <config> elements to provide actual configurable values. To represent a simple configuration entry with a single value, a <config> element is used. In the example in Figure 2, the property named ”propertyName” has a value of “value1”. Similarly, to represent a simple list of values, a <configList> element can be used. In the example in Figure 2, the “listPropertyName” property provides a list of two values “value2” and “value3” that can be read as an array of values in code.

For more complex cases, configuration values can be placed into configuration block subgroups to provide a namespace mechanism that will prevent naming conflicts and allow hierarchical organization of properties. Configuration properties defined within configuration blocks have names that are prefaced with the configuration block name and a “.” character. In the example in Figure 2, the property “blockName.nestedPropertyName” would have a value of “value4”. Configuration blocks can be nested arbitrarily deep and properties in the nested blocks will follow the same name prefixing convention as described above. To access the value of “myProperty” in the example in Figure 2, the property name “outer.inner.myProperty” would be used.

MPCS Configuration File Hierarchy

As a multi-mission product used by many different users in a number of different environments, it was important for MPCS to provide a simple way for different users and different missions to have their own configuration values that would override the default values put in place by the basic MPCS system configuration. Rather than use a single central configuration file, the MPCS team chose to use a hierarchy of configuration files to increase flexibility. The MPCS system can be configured at the system level, the project/mission level, and the user level. The hierarchy of MPCS configuration files is shown in Figure 3.

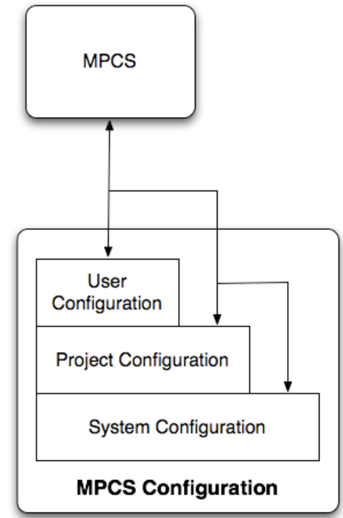

Figure 3 – MPCS Configuration Hierarchy

The system configuration is the complete set of all configuration properties and their values that are defined for MPCS. Everything that is made configurable should always be placed in the system configuration file. The project configuration file contains a subset of the contents of the system configuration file. If the same property is defined in the system configuration and project configuration, the value from the project configuration will be used. By providing a separate mechanism for project configuration, MPCS can use much of the same core code to do processing for different missions such as MER and MSL. Finally, MPCS provides the ability to specify a user configuration file that can override both the project and system configurations. This capability allows individual users to tweak MPCS behavior to suit their own testing and development needs. The system configuration is the only file that is absolutely necessary for MPCS to run; the project and user configurations are optional. Furthermore, the user configuration file may be disabled for controlled environments like flight operations where it is undesirable to allow users to override normal system behavior.

MPCS Configuration In Action

The overall structure of the MPCS code mirrors the structure of the configuration files at the system and project level. The majority of the MPCS code is written as a common, mission-independent core that uses dynamically instantiated adapters to handle venue-specific or mission-specific business logic.

MPCS uses a singleton design pattern to represent its configuration internally in code. The singleton configuration allows any piece of code to be able to easily pull information out of the MPCS configuration and because the majority of the MPCS design is single-threaded, the

scalability and synchronization issues of the singleton pattern are minimized.

The earlier discussion of the MPCS downlink architecture described how MPCS downlink uses a series of adapters to perform various sequential tasks in the telemetry processing chain. The diagram in Figure 1 represents a snapshot of the MPCS downlink processing as it is running, but what remains to be discussed is how MPCS determines what adapters to load before downlink processing begins. MPCS has the idea of a supervisory "Downlink Manager" object whose responsibility it is to instantiate all of the necessary adapters before telemetry processing begins. Refer back to Figure 1 throughout the following discussion to recall where each adapter fits in the telemetry processing pipeline.

The first and foremost adapter to be instantiated is the Raw Input Adapter that will read raw telemetry from an input source. The Raw Input Adapter to load is not known until runtime because it is dependent on the venue and data format information supplied by the user's MPCS session. MPCS has the ability to ingest a wide array of data including either frames or packets in various forms. In addition, MPCS can read its data from a file, a network socket, or any number of other APIs such as the interface to the Telemetry Delivery Subsystem (TDS) piece of Deep Space Network (DSN) where telemetry will be received operationally for MSL. The MPCS Downlink Manager uses an Abstract Factory design pattern that examines the session information and the configuration and then dynamically instantiates the proper input adapter for the given scenario. For example, in the MSL WSTS environment, MPCS must ingest transfer frames from a software socket whereas in the MSL testbed and ATLO environments MPCS ingests transfer frames from hardware that emulates the DSN. As part of the Abstract Factory pattern, all raw input adapters share a common interface and are therefore interchangeable in the MPCS telemetry flow shown in Figure 1.

The Frame Sync Adapter and Packet Extract Adapter work in a different paradigm than the Raw Input Adapter. Frame synchronization and packet extraction are based on CCSDS standards, so they are multi-mission adapters that will generally not change based on a particular mission. Based on the format of the input data, however, one or both of these adapters may be superfluous. The MPCS Downlink Manager looks at the input data format in the session information and then checks the configuration to determine if frame synchronization or packet extraction is necessary for that particular data format. For instance, if MPCS is configured to read a stream of data packets (recall that data packets are normally extracted from frames), then frame synchronization and packet extraction are both unnecessary since they have already been performed prior to the data reaching MPCS. Through configuration, MPCS is able to determine that based on a packet input format it should not instantiate these two adapters that serve no purpose.

The EVR, EHA, and data product adapters follow yet another configuration strategy to determine what should be loaded. As mentioned previously, extracting and constructing these particular telemetry items from packets is generally a mission-specific operation. In the MPCS system configuration, there is a property specifying a default adapter for processing each of these types of telemetry, but MPCS needs the ability to override what adapter is loaded based on the mission. Using the MPCS project configuration file, each mission can specify its own adapter to instantiate based on mission-specific needs. The MPCS Downlink Manager uses an Abstract Factory design pattern to examine the configuration properties that specify the names of the EVR, EHA, and data products adapters, and then instantiate the proper adapter for the given mission. Figure 4 demonstrates how the MPCS Downlink Manager instantiates the proper EVR adapter with the help of an Abstract Factory object.

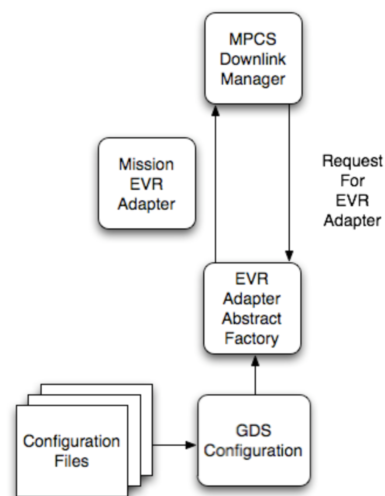

Figure 4 – MPCS EVR Adapter Instantiation

Another feature of the MPCS configuration is the control of whether EVR, EHA, or data product processing is done at all. At the system and project levels of the MPCS configuration, all of the EVR, EHA, and data product processing is always enabled[5]. At the user level, however, the configuration entries may be overridden to force MPCS not to process particular telemetry items. For instance, if a user is only interested in EVRs, then they can use the MPCS user configuration file to disable EHA and data product processing to reduce the MPCS processing load and improve throughput.

The remaining adapters, the Database Adapter and the JMS Bus Adapter, are a hybrid of the previously discussed scenarios. Like Frame Sync and Packet Extraction, Database and JMS operations are generally mission-

[5] The MPCS adaptations for Diviner and DAWN only process EHA telemetry, so no EVR or data product processing adapters exist.

independent so they are not overridden on a per-mission basis. In addition, like EVRs, EHA channels, and data products, database and JMS operation are generally desirable, so they are enabled by default at the system and project levels. In certain testing and debugging scenarios, however, it is desirable to disable one or both of these adapters. Enabling and disabling database and JMS usage can be done using the MPCS user configuration file in a similar manner to how the EVR, EHA, and data product adapters can be enabled and disabled.

In this manner, the MPCS Downlink Manager can use the MPCS configuration and session information combined with an Abstract Factory design patterns to dynamically alter every aspect of MPCS telemetry processing.

4. CONCLUSIONS

Over the course of its development, MPCS has come to find CDD an invaluable tool for maintaining flexibility in a constantly growing and evolving software architecture. MPCS is currently in use for testing through many different phases, including the WSTS, testbed and ATLO venues, of five different missions including the upcoming MSL mission. In working through the entire lifecycle of the MSL mission, MPCS has discovered that the MSL FSW and SSE systems are in flux and often the MPCS processing flow must change drastically from release to release.

CDD has proven very valuable in its flexibility in keeping up with changing needs and requirements across the different mission phases of all MPCS' customers. There have been numerous occasions where MPCS has been able to quickly adapt to feature requests and bug fixes solely based on editing entries in a configuration file. The MPCS team has been able to save a lot of time and effort by making fixes via configuration so that the MPCS software does not have to be redelivered to customers.

Although CDD has proven very useful in maintaining an agile software development process, the added flexibility and adaptability comes at a high initial cost. Developing a configuration-driven infrastructure involves spending up-front time and effort designing a configuration file format and developing parsing infrastructure. Once the infrastructure is in place, however, massive changes can be made to functionality in a much smaller timeframe. For inexperienced developers or developers new to MPCS, however, there is also a steep learning curve. Using CDD effectively involves developers adhering to particular design patterns and coding standards and it takes developers time to come up to speed with these restrictions.

CDD as a design strategy states that configuration structure and content should always be written prior to writing any business logic in code, but the MPCS team has discovered that CDD is better utilized as an iterative process. Developers will initially examine the configuration file and potentially make additions prior to doing code development, but it is impossible to know everything to put in the configuration before any development has been done. The MPCS team has found that the best practice is to write basic configuration, then write the business logic to leverage the configuration, and finally, iteratively refactor the code and configuration simultaneously to achieve as much flexibility as possible.

From a coding standpoint, CDD has also proven valuable in allowing the technology base of MPCS to expand. The MPCS core code is written solely in Java, but in the past year MPCS has also added a Python-based automated scripting tool as part of its delivery. Although MPCS is now composed of two distinct languages, Java and Python, both languages still easily share the same XML configuration base for modifying their functionality. Changes to the MPCS configuration affect both the Java and Python code uniformly. On a similar note, another advantage of CDD has been avoiding code duplication in the MPCS environment. By pulling common values out into the configuration, MPCS has been able to greatly cut down on the "ripple effect" of small changes propagating throughout all of the code.

One major concern that has arisen is that the MPCS configuration files have grown very large. The split of the MPCS hierarchy out into system, project, and user configuration files has helped contain some of the configuration bloat, but as MPCS continues to grow in size and scope, the configuration files continue to grow with it. For the future of MPCS as well as any other projects that plan on using a CDD strategy, it would be beneficial to split up configuration across a dynamically sizeable set of configuration files rather than trying to concentrate all configuration information into a fixed number of files. The MPCS team has also found that the configuration file(s) need occasional reexamination and pruning to prevent them from becoming bloated with redundant or vestigial information.

One of the more surprising limitations of the MPCS CDD approach has been the reluctance of users to take advantage of the user configuration file. By using XML, an inherently human-readable format in conjunction with the ability to adjust software behavior based on a user-supplied configure file, MPCS expected to have users easily modify the behavior of the software on their own with no necessary intervention from the MPCS team. In reality, however, MPCS has found that the average end user is still adverse to XML and either does not understand it or does not want to write it. A helpful strategy has been to provide a set of user configuration files for common override behaviors and then allow users to simply copy these files for their own use. Any needed user configuration overrides that do not fall into this category are rare and have been handled on a case-by-case basis without issue.

Overall, the benefits of the MPCS CDD strategy have greatly outweighed the issues that have been encountered. The MPCS CDD design has been flexible enough to adapt to a number of different missions and agile enough to keep up with the constantly shifting needs and requirements of the MSL mission.

5. ACKNOWLEDGEMENTS

The authors would like to thank the past and current MPCS team members Jesse Wright, Jim McKelvey, Mark Palm, Josh Choi, Clark Williams, Ashley Shamilian, Kyran Owen-Mankovich, Dan Allard, Lloyd DeForrest, Michael Tankenson, and Jordan Lei of the Jet Propulsion Laboratory for their contributions.

The work described in this paper was conducted at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

REFERENCES

[1] Aerospace Conference Web site http://www.aeroconf.org/

[2] Dan Allard, "Development of a Ground Data Messaging Infrastructure for the Mars Science Laboratory and Beyond," 2008 IEEE Aerospace Conference Proceedings, March 1-8, 2008.

[3] Martin Fowler, Inversion of Control Containers and the Dependency Injection Pattern, MartinFowler.com, http://martinfowler.com/articles/injection.html, January 23, 2044.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.

[5] Jacques Malenfant, Reflection in Logic, Functional, and Object-Oriented Programming: A Short Comparative Study, 1995 IJCAI Workshop on Reflection and Metalevel Architectures and Their Applications in AI, August 20-25, 1995.

[6] Robert C. Martin, The Principles, Patterns, and Practices of Agile Software Development, Prentice Hall, 2002.

[7] Steve McDuff, "Configuration Driven Development: A practical approach to code modification and duplication", IBM DeveloperWorks, http://www.ibm.com/developerworks/library/wa-configdev/, December 12, 2006.

BIOGRAPHY

***Brent Nash** has been a software engineer at the Jet Propulsion Laboratory (JPL) for three years. He currently works on the MPCS Ground Data System for the Mars Science Laboratory and other missions. He has also worked on a framework for agent-based systems and researched enterprise architecture best practices. He has a BS in Computer Science and Computer Engineering as well as an MS in Computer Science from the University of Southern California.*

***Marti DeMore** is a software engineer at JPL and is currently the cognizant engineer for the MPCS Ground Data System for the Mars Science Laboratory and other missions. She has 23 years of experience in software architecture, design, and development, primarily in the areas of enterprise messaging and distributed data and device management.*

She has a B.S. in Computer Science from California State Polytechnic University at Pomona.