Optimizing Flight Control Software With an Application Platform

Irene Skupniewicz Smith
SGT, Inc.
NASA Ames Research Center
P.O. Box 1, MS 269-3
Moffett Field, CA 94035
650-604-4414
irene.s.smith@nasa.gov

Nija Shi
SGT, Inc.
NASA Ames Research Center
P.O. Box 1, MS 269-3
Moffett Field, CA 94035
650-604-1376
nija.shi@nasa.gov

Christopher Webster
Carnegie Mellon University
NASA Ames Research Center
P.O. Box 1, MS 269-3
Moffett Field, CA 94035
650-604-5192
chris.webster@nasa.gov

Abstract—**Flight controllers in NASA's mission control centers work day and night to ensure that missions succeed and crews are safe. The IT goals of NASA mission control centers are similar to those of most businesses: to evolve IT infrastructure from basic to dynamic. This paper describes Mission Control Technologies (MCT), an application platform that is powering mission control today and is designed to meet the needs of future NASA control centers. MCT is an extensible platform that provides GUI components and a runtime environment. The platform enables NASA's IT goals through its use of lightweight interfaces and configurable components, which promote standardization and incorporate useful solution patterns. The MCT architecture positions mission control centers to reach the goal of dynamic IT, leading to lower cost of ownership, and treating software as a strategic investment.**

TABLE OF CONTENTS

1. INTRODUCTION

NASA's Mission Control Center (MCC) at Johnson Space Center in Houston is operated day and night, ensuring that scientists and specialists aboard the International Space Station (ISS) remain safe. ISS flight controllers currently rely on over a thousand separate software applications developed over the years to perform specific monitoring and control functions. At other mission control centers at NASA's Ames Research Center and the Jet Propulsion Laboratory in California, flight controllers are similarly controlling small satellites and autonomous rovers.

Mission Control Technologies (MCT) is software designed to power mission control centers in NASA today and into the future. MCT must support high system availability, composition, and certification of flight controller designed components; and rapid reconfigurability of equipment in the control center. The MCT platform architecture enables these goals through its use of lightweight interfaces and loosely coupled components.

Our customers' goals are similar to those of most businesses: to evolve their infrastructure from basic to dynamic. Dynamic infrastructure is a goal of a mature IT organization model put forth by Microsoft and others [1]. In the context of space operations, a dynamic infrastructure can support missions and simulations concurrently, will accommodate data migrations such as changes to telemetry streams, and can readily support the next generation of vehicles. Applications written via the MCT platform help move the MCC toward this goal by providing software standardization, software reconfigurability, rapid certification, and by capturing operation patterns (best practices, business processes). Reaching the goal of dynamic infrastructure means lower cost of ownership, and software as a strategic investment.

While these are the MCT platform's goals, it also has clearly defined "non-goals." MCT is designed to be an extensible application platform, but not to provide IT infrastructure services such as identity management, data protection and data recoverability.

2. ARCHITECTURE OVERVIEW

MCT [5, 6, 7] is organized around "user objects" that are designed to mirror the data and behavior of their real-world counterparts. An MCT user object is referred to as an "MCT component." Some examples of components are telemetry elements, procedures, commands, physical parts such as pumps or batteries, and collections of other MCT components. An MCT collection component contains other MCT components; MCT displays are compositions of views of these components.

An MCT component is implemented as an OSGi (Open Service Gateway Initiative) [4] bundle. OSGi provides Java classpath isolation, version support, and a service model. Additionally, the OSGi service programming model isolates faults in specific code bundles that can be handled generically in the platform. Finally, OSGi provides the ability to dynamically deploy software bundles, allowing a patch to be applied to a running system.

The MCT platform uses several persistent stores, as shown in Figure 1. A MySQL relational database serves as the internal storage mechanism. The database stores the component models and the relationships between

components, as well as the persistent view information. Database replication provides read availability during failure of the master (write) instance. The database contains information about the user objects but not telemetry values. Real-time telemetry values are buffered in a local embedded Oracle Sleepycat Berkeley database and are used to populate short-term historical values on plots. MCT serves to combine multiple disparate data sources. For example, during spacecraft operations MCT combines telemetry metadata, real-time data and archive data stores for the telemetry values.

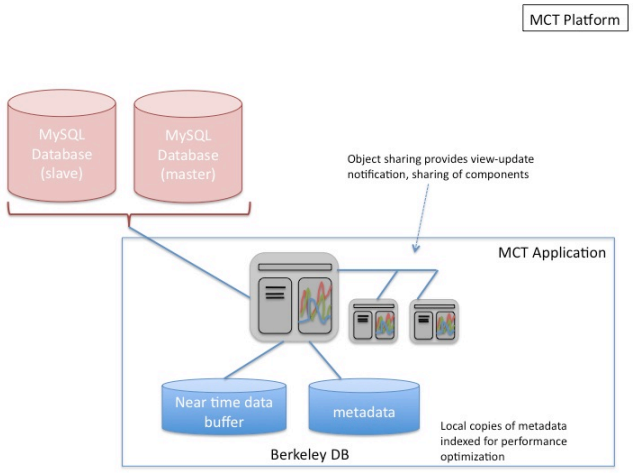


Figure 1 – MCT Application Platform

MCT provides a metadata service. Metadata is information about the data symbols, such as units and calibration. Metadata data stores can be quite large. Furthermore, metadata may be frequently managed and revised; for example, when a new instrument is added to a vehicle, a new set of sensors may be added. Telemetry components can be written for future telemetry types that allow defining and creating data components and their metadata.

The MCT platform provides a policy system that allows control over the CRUD (create, read, update, and delete) aspects of the site policy. The policy system controls which views are available on which component types, view ordering, and composition (the embedding of components into other components).

The DataProvider service gets streams of data ("data feeds") identified by feed IDs. Typically, feeds are streams relayed from a monitored vehicle. However, MCT's component architecture can support various feed types, such as for scaling to large amount of data or to accommodate for new data formats and heterogeneous types, by plugging in a new DataProvider component. Components have been written to support both near-real-time data and archived data (preserved from previous missions), simulation, and predictive data. Predictive data is typically generated using archived samples and scientific models and serves to anticipate a flight failure event.

DataProvider interfaces handle data asynchronously and use a standard interprocess communication mechanism such as TCP Sockets or RESTful Web Services. Each data provider component is identified by a "feed ID," and an MCT core service aggregates the feeds for display in MCT components. Thus in the preceding discussion, feed provider components can simultaneously provide near-real-time, simulated, archived, and predictive data.

The MCT object sharing mechanism allows a component to be shared with any other MCT user. Direct sharing involves dragging a set of components into a drop box. The drop gesture makes the component available to the user (or set of users, depending on the drop target) for viewing or modifying based on policy, and also changes the object's state "shared." Once a component becomes shared, modifications to the state of the component (including the persistent state of its views) require coordination to prevent issues such as lost updates. MCT currently supports the ability to unlock an object for exclusive writing (pessimistic locking) wherein the object is unavailable for editing until the lock is released. The common work practice is to do most development in a user's private sandbox prior to sharing objects (specifically for building displays that are destined for certification for shared use). The object sharing mechanism is implemented by polling the database for updates, maintaining "dirty" markers, sending component updates over the network, and remotely invoking a refresh on the GUI element for that component.

MCT also offers tagging and labeling. The tagging mechanism allows attaching attributes to components. This is employed in control center workflows such as a certification workflow.

The labeling mechanism allows flight controllers to abbreviate and shorten labels while ensuring consistency in the computation of the abbreviation. MCT has a user-customizable labeling algorithm for plot legends and tabular alpha display views. MCT's labeling is automated, eliminating manual verification of each label resulting in predictable abbreviations.

3. Designing for Flexibility

The pioneers of patterns in software development defined patterns for user interfaces such as patterns Window per Task and Standard Panes [2]. The software community adopted these patterns as best practices because they promoted consistent user navigation experience. As the set of patterns grew, they were classified and defined in ways to make it easy for software developers to find and use.

On such classification is where they are used. Patterns may be used in the *application* domain—in our case, flight operations—or in the *solution* domain. When we look at the domain of the solution we are looking at the internal structure of the software, and are reflecting primarily software-centric concerns. Patterns in the solution domain include idioms that are related to specific programming

languages, architectural designs, and design patterns (these are smaller in scope than architectural designs) [2]. In our world, the application domain is that of the flight controller. Their daily processes include user-centric patterns that can be captured, implemented and enforced via the application platform. For example, a flight controller who needs to certify a display uses peer reviews and group lead verifications, and passes along the display to users in "drop boxes". This workflow and its drop boxes were captured in our software.

Software-centric Patterns

MCT is no different than other product development efforts, in that software patterns are adapted based on what is appropriate for the product. In order to achieve flexibility, design pattern definitions usually introduce additional levels of indirection, which in many cases complicate the resulting design pattern. In real-world applications, using all elements of a pattern can unnecessarily complicate an implementation and hurt performance. Invariably MCT's implementations of design patterns are modified so they are the simplest, most testable solutions.

Dynamic Capabilities Ensure Extensibility—MCT employs a modified form of the Extension Interface pattern to define capabilities for a component type. The focus of Extension Interface [3] is to engineer a class that supports additional methods or services. Clients query the object first to determine if it supports an extension before attempting to use that extension. Normally, a class is extended by subclassing and adding methods to the derived class. Extension Interface provides extensibility without compile-time subclassing. It allows multiple interfaces to be implemented by a component, while preventing bloating of interfaces and breaking of client code when developers extend or modify the functionality of the component.

A full implementation of the Extension Interface pattern would have added unnecessary levels of indirection, and unneeded objects. Thus MCT's Capability implementation uses a simple Capability extended interface that gets the job done. Figure 2 shows an MCT Component with its Capability API. The client may query the component before using an extended interface.

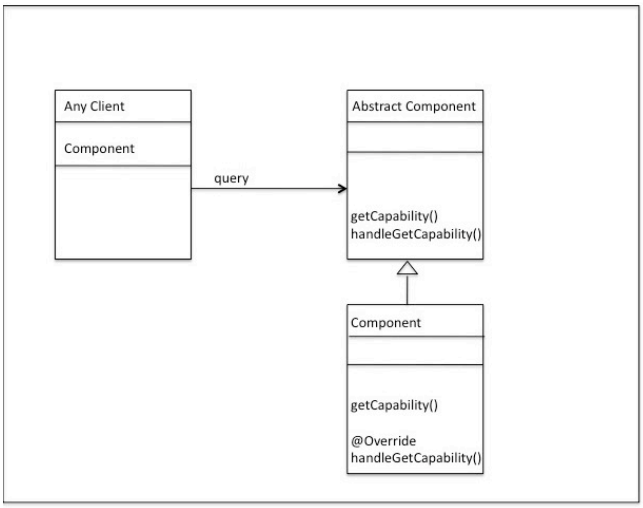


Figure 2 – Dynamic Capabilities

MCT Capabilities are not always known at compile time, either because they are dynamic, or because the component has not yet been defined. For example, an MCT component has an Initializer capability at the beginning of its lifecycle and then no longer. Also an MCT component may be updatable—or not—at various times during its lifecycle. The MCT Capability interface has the getCapability() method that takes a generalized capability object as input and returns a specific capability class. Referring to the above example, if a component is presently updatable, it will return the Updatable class.

MCT provides an extensible component architecture, therefore the APIs and capabilities of a component will be defined both in the platform and also as part of the unanticipated consequences of extending the platform. Requiring the component implementation class to implement a specific interface results in component code that contains multiple concerns and quickly becomes difficult to maintain. Instead, the MCT component API defines a getCapability() API where components can return instances of interface implementations if the capability is supported. This allows component authors to cleanly separate concerns.

Context-aware Objects Implement Flexible Policies and Menus—To ensure that MCT meets the security requirements of all our customers, we built an extensible context-aware policy management system to control access for user actions—in particular, the user actions that are access-controlled include object inspection, modification, and creation. We have implemented policy injection points before execution of these user actions. The policy management system maintains a map of policies. A policy must implement our policy interface, which requires a policy to take a context and then returns a Boolean. A context is an object that contains information for executing the policy. Such information includes items such as the objects to be changed and the purpose for the change. A policy is stateless and must be reusable in a different

context. On top of the policies, the policy manager maintains a list of policy categories. A policy category corresponds to a user action, and it contains a list of policies to execute for this user action. The execution is implemented using a variant of the Chain of Responsibility pattern [10]. For a given context, a user action is granted when all policies return true. Figure 3 illustrates the policy execution flow. Each circle represents a policy, and the policy manager executes a list of policies registered for a policy category. A solid green circle means the execution of that policy returns true; red means otherwise. The gray circle illustrates a policy that was not executed. The first scenario illustrates a failed case, while the second illustrates a successful case. Policy execution fails when the first policy returns fails. The implementation of a policy must be context-free and must not depend on the execution results and sequence of other policies.

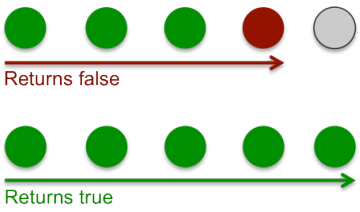


Figure 3 – Policy Execution Flow

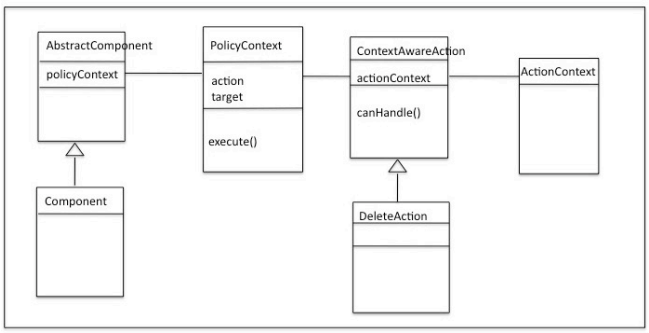


Figure 4 – Context-Aware Policy

Figure 4 shows a PolicyManager providing the execute() API. The execute() call returns false unless the policy decision is supported by all the context elements, including the action, the target, and the policy context. For example, a flight controller may want to edit the display name of a symbol, and site policy dictates that only group lead controllers have edit capabilities. The policy context defines the role of the flight controller, the type of component to be edited (this is the target), and the action which in this case is to lock the component. If the policy returns true, the lead controller locks the component, makes the edit and then unlocks the component. Note that in the locked state, we make an editable clone of the component, and upon completion we merge the changes back to the component.

We use context-aware objects in a similar manner to implement dynamic menus. Each MCT menu item is defined with its action list. Whether a menu item is enabled and displayed depends on the action and the menu context. For example, if a component is displayed in a directory tree, its "delete" menu item is disabled, whereas if the same component is displayed in a composition canvas, the delete menu item and its action are enabled. Policy decisions can be injected into the dynamic menu design; for example, a menu item may be enabled only for users in the group lead role. The menu facility can operate on individual menu items, submenus, or entire menus.

Using Aggregation Ensures Scalability—The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. As described in the overview, MCT implements object sharing by invoking a refresh on displays whose components have been identified as dirty and needing refresh. This uses the basic Observer pattern. Observer is also used by OSGi Services for service discovery and resolution [9]. In this design, the client registers its need for services into the framework, and the available services monitor the framework for all clients interested in that service. In the context of MCT, components register themselves as listeners with the OSGi framework, and other services discover these requests. The benefit of this software design is extensibility: MCT components can communicate, while they are loosely coupled.

However, since Observer defines a one-to-many dependency between objects whereby dependents are notified and updated automatically, it is not used for MCT's data feed displays (feed views). If MCT's feed updates were implemented using Observer, MCT would not scale because feed states change constantly and the number of MCT feed views is unlimited. In other words, Observer is an anti-pattern for MCT feed updates. Instead, MCT uses a feed aggregator service to provide data to MCT views. In our implementation shown in Figure 5, MCT implements a single class FeedAggregatorService which services a set of FeedProvider clients. FeedAggregatorService also prepares aggregated data suitable for painting to MCT visual components. A SwingWorker thread gets the data from FeedAggregatorService and dispatches the data to each view. Because it can federate feed providers, FeedAggregatorService supports regulated painting performance; MCT's GUI components can incrementally slow the paint calls or speed them up to adjust for loads.

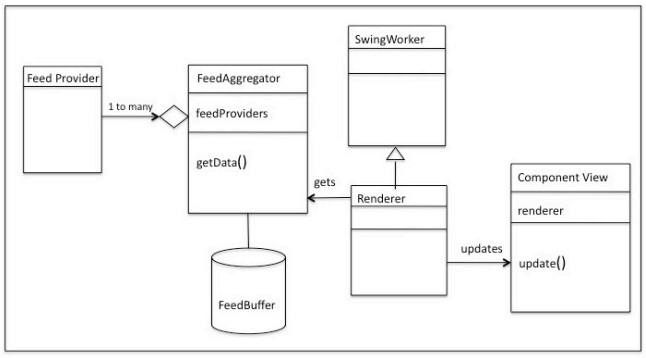


Figure 5 – Feed Aggregator Ensures Scalability

FeedAggregatorService employs Chain of Responsibility pattern to determine which feed provider can handle a request for data. Each FeedProvider registers itself with a service level indicating its speed of retrieval, for example whether the speed is fast (real time) or slow (indicating the data is archival) and FeedAggregatorService examines the service level request and compares it to its current list of FeedProvider handlers. If the service level request can be satisfied, this provider is used. Otherwise FeedAggregatorService inspects the service levels of other providers to determine if they can be used. A chained pattern simplifies the overall design, because a provider doesn't have to keep provider references and only needs to be aware of its own data service capability. Another benefit of chained responsibility is that data services can be added or removed dynamically simply by adding or removing feed providers; the order of chaining can also be altered to affect the aggregated data service.

User-Centric Patterns and Processes

The mission control workplace has dozens of people processes or workflows. Capturing these processes into software adds value. In fact, the process of formally defining and capturing the process for the first time adds value. Invariably, capturing a people process into software means that the software is enforcing site workflow steps, ensuring higher quality and consistent workflows with more predicable results.

As the MCT team worked with our customers, we implemented site workflows into the software in several areas. Some work practices we captured include introducing a full symbol taxonomy, loading viewable symbol metadata, tagging metadata to mission control activities, and certifying components. This paper describes one of these processes, certification, in detail.

The mission control's Certification Workflow is needed to verify that displays and their data can support mission critical needs. Typically a new display is introduced by a lead flight controller, then another lead flight controller certifies the display by a number of workflow steps such as using it in a flight simulation.

The goal of certification involves verifying that the right labels are applied, that data formatting is correct, and that visual composition and layout meet the needs of the flight activity. For example, a verifier checks that a Battery Power group contains relevant battery symbols with valid unit definitions. The verifier ensures that data values are displayed in an intuitive layout, with data formatting appropriate to the data type (for example, a date or real number with a precision of 3). The verifier also ensures that data elements are labeled in a human-interpretable way.

In a basic mission control IT infrastructure, display layouts and data formats are defined using text files. Each application has its own display-formatting language. The lead flight controller designs these displays and layouts by editing text files, and copying the file into place. The displays are sometimes customized for a particular vehicle or flight, so the text files may be renamed and tweaked. The result is a proliferation of displays, each of which needs to be certified. Once a display is verified, it enters the control center's Certification Workflow for peer review and approval.

With MCT, the layouts are done within the application using graphical drag-and drop-and in-place edits. The data types are defined along with the telemetry, reducing errors in data formatting, and the labels are stored with each piece of telemetry. MCT's labeling algorithm improves safety, because it guarantees that all the pieces of the telemetry's official name are visible somewhere near that telemetry's value—adjacent to the value, or as part of it in a column or row header or panel or window title. The legacy applications rely on users to manually type labels that meet that criterion. Nevertheless, NASA missions require certification of all displays, therefore we captured the control center's certification workflow and implemented it with MCT components.

MCT's Certification Workflow provides tools that use MCT "dropbox" and "tagging" features to implement certification workflow. A dropbox is a collection component that is shared. When a component is dropped into it, the dropped component becomes visible to users with privileges to see inside that dropbox. The MCT tagging feature (tagging can be a GUI gesture or programmatic) attaches properties to MCT components. When a flight controller creates a display in anticipation of a new mission and needs to have it certified, he drops it into a collection named "Ready for Peer Review." The action of dropping that component tags the component with the property "Ready for Peer Review," and makes that object read-only. When a peer user decides that object is verified, he tags it as "Ready for Approval." This GUI gesture automatically removes that object from the "Ready for Peer Review" collection, and adds it to the "Ready for Approval" collection. The workflow continues until the object is in a "Certified" collection.

With MCT, a piece of telemetry can be viewed as an object, and a display can be viewed as a collection of references to those objects. Because the same telemetry object can be

referenced in many display collections, a change in one telemetry object is reflected in all display collections referencing that object. MCT's component model supports this inherently; therefore when a component is certified once, in one place, it is certified throughout. This not only speeds up the process of certification, but also provides additional safety as the mapping is only required in a single place rather than for each display.

Because tags are persisted with the component, MCT now can differentiate between certified and uncertified components. Uncertified components might be accessible in a training activity or a simulation, but mission-critical activities will accept only certified displays. The software enforces this in a way that was not possible with the legacy applications. The architectural elements of tagging and object sharing are integrated into the MCT platform, and are implemented with persistence and concurrency protection such as locks when an object is edited by one user. Objects are shared across the cluster of MCT instances, so a change to a component in one MCT instance will be available to update that component in all MCT instances.

4. DESIGNING FOR STANDARDIZATION

An important goal of MCT is to standardize both the user experience and the underlying code. The following sections describe standardization of MCT's visual displays, component design, and computations.

Standardization of Visual Components

Flight controllers monitor thousands of signals, and are trained in pattern recognition to differentiate between normal data and data that contain aberrations. Without standardized displays, their ability to see patterns is reduced. Flight controllers arrange data tables and plots into logical groupings to aid in visual monitoring. Consistency across views ensures that alert icons are the same across components, and that menu item locations are consistent.

Mission control center activities change often, with new flights and equipment, training and simulation activities, and flight following test runs. To support this, existing displays need to be modified and new displays need to be introduced. Thus the IT infrastructure needs to support display composition and reconfiguration, such that the resulting displays are consistent, easy to read, and have a comfortable, pleasant look and feel.

Applications developed with MCT platform can provide rapid user composition of displays that are unified and consistent. User composition empowers flight controllers to make rapid changes to visualizations, within the constraints of organizational specified polices, without the need for code changes.

We ensure consistency between MCT visual components in a number of ways. Each component has consistent default views; for example, an Alpha View is the same across diverse components. Component-specific menu items are added using MCT's dynamic menu feature, and access to components is enforced using our policy manager.

MCT employs the Composite design pattern [8], which allows building structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes. Using Composite, the same operations can be applied over both collections and individual objects. In other words, in most cases we can ignore the differences between compositions-of-objects and individual objects; the MCT designers say that in MCT "a thing is a thing."

Standardization of Component Development

When a need is not met by the core components shipped with MCT, an MCT developer designs and writes a new component. MCT employs the well-known Model View Controller (MVC) pattern within each component, thus our component design can be considered an MVC micro-architecture. Because MCT component developers are familiar with the pattern, they have a shorter learning curve.

To create a new component, an MCT component developer may choose to modify an existing MCT component. When this approach is used, the developer may need to redefine only the model part—the part that is unique to his application needs. The developer may redefine the default views for an existing component, or write a new view and attach it to an existing MCT component. An MCT component developer defines a new component type by implementing the ComponentProvider interface. ComponentProvider allows a component type to be defined and a new component instance to be registered in an application instance. The MCT platform ships with other visual controls in addition to default component views, such as status indicators and menus. When an MCT developer defines a new component type, he also defines its menu items and the actions associated with each menu item. The new menu items and their actions automatically are added to MCT's menu definitions.

Importantly, the component developer can use MCT policies to provide access control to the new component. For example, site policy may dictate that view edits can only be performed by a lead flight controller. Thus if the current user is not in the lead flight controller role, the policy is enforced by disabling and graying out the menu item. The policy manager, described elsewhere in this document, is capable of controlling many platform concerns, not just menus. Examples include definitions of workflows and support of "locked down" mode.

As mentioned in the introduction, an MCT component is packaged as an OSGi bundle. An OSGi bundle may implement a set of services, consume services, or do both. The OSGi Service architecture prescribes using OSGi Services as the preferred method to communicate between bundles. Accordingly, MCT uses OSGi services extensively to communicate between MCT components. This facilitates

loose coupling, as MCT components do not hold a hard reference to other MCT components; rather, they hold a service reference. OSGi Services are used to export functionality from one MCT Component to another MCT component, and to import functionality from other components. MCT's core set of components communicate using dozens of services; some of these are DataProvider, EventProvider, FeedAggregator, and Evaluator (described later in this paper).

A component packaged as an OSGi bundle enjoys all the benefits of OSGi deployment. Classpath isolation is the ability to have a unique classpath for each bundle to prevent conflicting class versions. Classpath isolation is particularly useful in solving the "transient dependency" problem. Suppose your application depends on dozens of libraries, which in turn depend on dozens of other libraries. The secondary dependencies, called transient dependencies, may include different versions of the same library. Without OSGi and therefore with a single class loader, a single version of the other may be loaded for the entire application, creating a situation in which some primary dependencies resolve with the correct transient dependency but others resolve with the incorrect version. Typically this version incompatibility manifests in runtime problems (which can be tough to debug!). With OSGi, MCT component developers are free to introduce their own bundles independent of other MCT bundles.

Standardization of Data Computations

Mission-critical calls are made with computed signals, called computationals. An original vehicle signal may be computed into a new signal, or signals may be interpreted for display on the control monitor. Because mission-critical calls are based on computationals, it is critical to control how they are defined. However, control centers may not always do an optimal job at standardizing computationals, since these computations have disparate origins and implementations.

At the vehicle or equipment manufacturer, the hardware back-plane originates the signal. These definitions are shipped to NASA and are version controlled in a database. These signal definitions may be associated with equipment-originated "enumerations" that translate a signal value into a semantic meaning; for example, a particular symbol's value of 7 may be associated with the string "STANDBY VALID." Another type of ground computation is mathematical, whereby one or more signal values are put through a function. Additionally, some calculations require multiple signal inputs; for example, two discrete signals could be used in combination to determine the output enumeration such as "ON" or "OFF." With MCT we created a singled component type, EvaluatorComponent, to handle all these computational types.

At our customer's site, computationals were handled with various legacy software implementations, at various layers in the infrastructure. Although the equipment-originated enumerations were stored in a database, they were not rolled out into the mission control center. Also, since each user application provides its own implementation of enumerations, enumeration definitions are stored in local files. Furthermore, execution of mathematical calculations is done by user applications (rather than a centralized server), and these applications are maintained independently by various flight control disciplines. As the IT infrastructure matures, it is desirable to move computation execution to a centralized server, to roll equipment-originated enumerations into the control center, and to define evaluations in a central database. The MCT framework is not being used to implement a centralized computational server, instead our customer plans to add a computational server with their next network upgrade. MCT is being used to roll out equipment-originated enumerations and to consolidate evaluation definitions for all components.

The EvaluatorComponent is able to implement equipment-originated computations and to replace legacy display computations. An evaluator component provides persistent storage and a flexible creation wizard to allow incorporation of arbitrary execution engines. The evaluator component provides a common interface for integration, and provides a Service Provider Interface (SPI) for execution engine developers to add language bindings and even enhanced editors. This is conceptually similar to the mechanism used to support dynamic execution languages in Java SE.

EvaluatorComponent is an MCT component that has an Evaluator capability (see Figure 6). An Evaluator has an Executor object associated with it. Executor requires the evaluate() API which takes a set of input signals from data feeds. Executor also defines an API to accommodate computations that require multiple signal inputs. MCT evaluators use language contexts. Employing elements of the Interpreter Pattern [10], the evaluators interpret context-sensitive language elements into code solutions. Languages are defined for each type of enumeration, such as an equipment-originated enumeration or a user-defined enumeration.

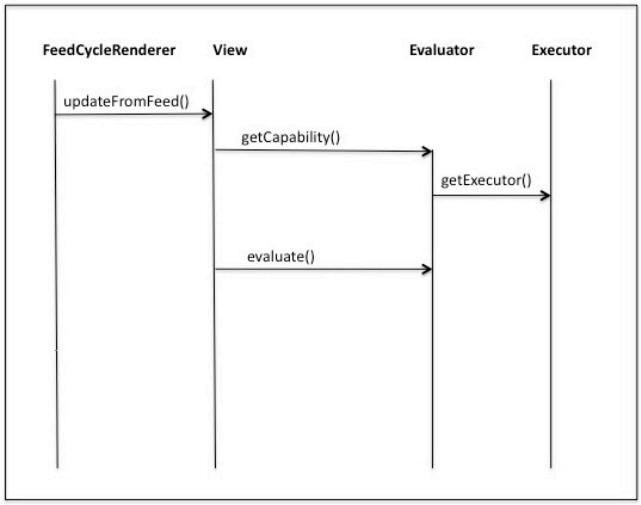


Figure 6 – Evaluator parameterizes execution

The executor interface is implemented by classes that can execute evaluations, so any class that wishes to assume the role of an executor can implement this interface. This design employs an element of the Command Pattern [4], in that we are encapsulating a request as an object, thereby letting us parameterize clients with different requests. In other words, we are allowing MCT Components with an executor role to assume responsibility for certain executions. Similarly because MCT defines an evaluator interface, a customer can "swap in" a new evaluator by writing an MCT component that implements the interface, and associating this new component with certain telemetry components. The end result of this flexible design is lower cost of introducing new evaluative elements into the control center. And with OSGi these elements can be introduced simply by adding a .jar file.

5. SUMMARY

Beginning in 2010, MCT is running in on our customer's control center at JSC. Flight controllers are using MCT in simulations and flight following. The time to create large displays ("mega-displays") is significantly lower, and because labels are standardized, display certification costs are lower. At other customer sites, the MCT developer platform is being used to create operational software for small satellite operations.

Optimized applications become strategic investments. In mission operations where space vehicles and their hardware get central focus, an IT infrastructure may be considered an afterthought. However, a rational approach considers IT infrastructure and software applications to be strategic enablers to the business of space ops, and integral with business productivity investments. At their fullest, mission operation centers with a dynamic infrastructure are fully aware of the strategic value their infrastructure provides in helping them run the business efficiently and staying ahead of competitors. Costs are controlled because the inventory of applications is decreased—instead of hundreds of applications in a flight control center, the replacement is a small inventory of applications that are managed with mature policies. The center purchases only the software and licenses that support this inventory.

When processes are fully automated, operational costs can be reduced because error-prone manual processes are captured in the software. Results can be audited and version controlled. Similarly, when software changes are needed to accommodate mission control reconfigurations, costs of software ownership can be controlled because fewer manual steps are involved. Also, the benefits of implementing new features needed to take on new business opportunities begin to outweigh the incremental costs of making those software changes.

REFERENCES

[1] D. Barney. "Infrastructure Optimization for IT." *Redmondmag.com*, January 2008. http://redmondmag.com/features/article.asp?editorialsid=2394.

[2] F. Buschmann et al. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley & Sons, 1996.

[3] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects.* Wiley & Sons, 2000.

[4] E. Freeman, E. Robson, B. Bates, and K. Sierra (2004). *Head First Design Patterns*. O'Reilly Media, 2004.

[5] J. Trimble and J. Walton. "Mission Control Technologies: A New Way of Designing and Evolving Mission Systems." In *SpaceOps 2006 Conference*. AIAA 2006.

[6] J. Trimble and A. Crocker. "A Flexible Evolvable Architecture for Constellation Mission Systems User Applications." In *SpaceOps 2008 Conference*. AIAA 2008.

[7] J. Trimble and A. Crocker. "Reinventing User Applications for Mission Control." In *SpaceOps 2010 Conference*. AIAA 2010.

[8] C. Lasater. *Design Patterns*. Wordware Publishing, 2006.

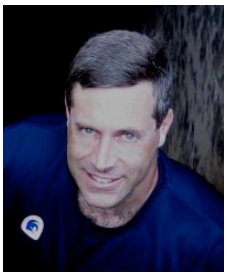
[9] Knopflerfish OSGi Tutorials. http://www.knopflerfish.org/tutorials.html

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

BIOGRAPHIES

Irene Skupniewicz Smith's role on the NASA MCT project includes integration with telemetry, the mission control data service, and deployment related aspects of object sharing and database. When she first came to NASA she added an SSL security layer to the Mars Exploration Rover situational awareness product. Before NASA she was at Hewlett-Packard as a software engineer, where she coded on teams for a just-in-time inventory management database system, a financial security policy management product, and HP's Java Lab. Prior to HP, Irene worked for Westinghouse in factory automation, programming automation tools. She has a masters in EE from Carnegie Mellon University.

Nija Shi has been a software developer in the Mission Control Technologies project at NASA's Ames Research Center since 2008. Her primary focus is on policy management and the architecture of MCT. Before joining NASA, she was a principal member of technical staff at the Server Technologies division of Oracle. While at Oracle she worked on JDeveloper 11g, a free integrated development environment that simplifies the development of Java-based SOA applications and user interfaces with support for the full development life cycle. Prior to Oracle, she interned at the Lawrence Livermore National Lab during 2004-2005, where she participated in the development of Babel, a tool for mixing C, C++, Fortran77, Fortran90, Python, and Java in a single application, and CALE, a 2D ALE hydrodynamics computer simulation program. She received her Ph.D. in Computer Science from the University of California, Davis in 2007. Her dissertation was on the reverse engineering of design patterns from Java source code.



Christopher Webster is a Computer Scientist working for the Mission Control Technologies project at NASA's Ames Research Center. He focuses on MCT architecture, performance, and developer APIs. Prior to NASA, Chris was a senior staff engineer at Sun Microsystems serving as lead engineer for Project zembly, the code in the cloud development and transparently scalable deployment environment (for web-based widgets including Facebook applications). He has also been the technical lead for the NetBeans XML tools project, a core member of the SOA development team, and was fundamental in getting Java EE support into NetBeans. Chris is an author of the NetBeans Field Guide and Assemble the Social Web With Zembly. Chris also worked as a computer scientist for the Lawrence Livermore National Laboratory, leading an effort to bring atmospheric dispersion modeling and visualization outside the data center as part of the National Atmospheric Release Advisory Center. He is a founder of jexamples.com, a search site dedicated to semantically correct searching for Java code examples mined from open source projects.

Chris currently holds three patents and has more than seven patents pending. He has been a speaker at Java One, Community One, Sun Tech Days, and the Server Side Symposium. He has a Master's degree in Computer Science from Baylor University and a Bachelors of Science in Computer Science from the University of Hawaii, Hilo.