Putting the Users in Charge:  
A Collaborative, User Composable Interface for NASA’s Mission Control

Jay P. Trimble  
NASA Ames Research Center  
jay.p.trimble@nasa.gov

Tom Dayton  
UARC  
U. of California at Santa Cruz  
tdayton@acm.org

Madelyn Quinol  
UARC  
U. of California at Santa Cruz  
madelyn.quinol@nasa.gov

Abstract
The software used by flight controllers in NASA’s Mission Control is made up of monolithic applications—a paradigm that comes with limitations. Each application is its own self contained world, limiting the granularity of collaboration to the application or screen level, and creating functional stovepipes and information redundancies. Applications are difficult to modify; they require recoding and compilation. Change control processes necessary for safe spaceflight limit changes after the start of mission training, but the start of training is the first time that users can use the software in a mission context.

Mission Control Technologies (MCT) has been developed to put the users in control by allowing users to compose software (policy controlled), and allow fine-grained sharing of information at the level of “user object” information entities. By replacing applications with shareable compositions, users of diverse needs get flexibility to interact and collaborate.

1. Background: Mission Control Overview

NASA’s iconic Mission Control Center (MCC) at Johnson Space Center (JSC), the flight operations hub of America’s human spaceflight program since the days of Gemini, remains operationally consistent with its fundamental form from the glory days of Apollo. Mission Control work is organized into classifications such as systems, planning, trajectory, and analysis, each of which is subdivided into “disciplines” such as biomedical engineer and surgeon. We will focus on the systems disciplines, which monitor vehicle health and status, and which control vehicle subsystems. Those systems are the software, hardware, people, and processes that are used to plan and control a mission. This is true for robotic as well as human spaceflight.

Current mission operations systems are built as a collection of monolithic “stovepiped” applications. Each application serves the needs of a narrow user base associated with a specific discipline or functional role. To accomplish specific tasks, each application embodies specialized functional knowledge and has its own data storage, data models, programmatic interfaces, user interfaces, and customized business logic. In effect, each application creates its own walled-off environment. While individual applications are sometimes reused across multiple missions, it is expensive and time-consuming to maintain these systems, and both costly and risky to upgrade them in the light of new requirements or to modify them for new purposes. It is even more expensive to create activities that are integrated across a set of monolithic, disparate, applications.

These problems affect the lifecycle cost (especially design, development, testing, training, maintenance, and integration) of each new mission operations system. They inhibit system innovation and evolution. Those problems hinder NASA in adopting new operations paradigms, including increasingly automated space systems such as autonomous rovers, autonomous onboard crew systems, and integrated control of human and robotic missions.

Hence to achieve NASA’s vision affordably, reliably, and effectively, we need to consider and mature new ways to build mission control systems that overcome the problems inherent in systems of monolithic applications. The keys to the solution are modularity and interoperability (see figures 1 and 2). Modularity will increase extensibility (evolution), reusability, and maintainability. Interoperability will enable composition of larger systems out of smaller parts, and enable the construction of new integrated activities that tie together, at a deep level, the capabilities of many of the components. Modularity and interoperability together contribute to flexibility.

Examples of what can be made modular include front-end items (e.g., user interface elements and tools), middleware services, backend capabilities (e.g., storage, databases), models, and core algorithms (e.g., constraint propagation, resource projection). Within
this framework, key dimensions include granularity (fine or course modules, and thin or thick interfaces) and content standardization level (communications protocol, syntax, and semantics). The component model reduces or eliminates current issues in sequential processes, stovepipes, and error-prone file transfers. This approach will enable future mission operations systems software to be flexible, reusable, seamless, and to have consistent user interactions (to minimize training requirements and operational error risk). This approach will ease incorporation of new technologies, and distribution of mission control systems across NASA and remote institutions as required by missions.

2. Software Flexibility and Space Mission Operations

NASA mission hardware, software, and the missions themselves are designed to meet requirements that necessarily are created years before the mission flies. The spacecraft is built to meet those mission requirements. Robotic missions are operated by ground teams, but crewed missions are operated collaboratively by the ground and the onboard crew.

The mantra for crewed missions is plan-train-fly. For software, the time gap between specification and use presents a fundamental issue. Software for missions is designed well ahead of training based on the requirements as they are known at the time the software spec was created. However, even the best up-front effort to design software will turn out to be lacking once simulation and training start. By the time a mission flies, mission controllers will find many capabilities they need that the software does not have.

The best of modern software practice involves many iterations of design and development with tight collaboration between user experience designers, developers, and users. For a space mission, this may be impossible before the start of mission training. In contrast, if you are developing software for most conventional activities such as finding the nearest restaurant, the software designers have many opportunities to test their software in the real world. If you're designing software for a Mars rover mission or a Saturn orbiter, it is nearly impossible to observe or interact with users in an actual usage scenario until the start of simulations and training for the mission, because probably you can't find groups of users already operating similar Saturn orbiters or Mars rovers. The exception to this is the International Space Station (ISS), which is in continuous operation.

One other important factor to consider is that, once simulations start for a space mission, all mission software and processes are under strict change control.

This makes changing software to adapt to user needs a long, rigorous process. In most cases, timely changes to user software in response to changing requirements is not possible due to mission safety and change control process issues.

Given the design-to-use time gap for most space missions, and the change control process, how can we meet user needs by being able to modify software quickly, without putting missions at risk? Our answer is user-composable software. By putting the power in the hands of the users, with appropriate policy controls and safety mechanisms, we can enable users to make changes to their software as they need to, without the support of developers. Mission Control Technologies (MCT) is our platform for creating user-composable software. All of the users’ software objects are contained in a single user environment, from which users may create functional software compositions. A policy manager allows the user organizations to enable and restrict permissions per organizational policy.

3. Collaboration for Mission Control

Mission control activities are highly collaborative. Flight controllers must share information in many forms, including spacecraft health data, procedures, plans, and real-time voice communications. From the days of Apollo through the first generation of Space Shuttle missions, information sharing in Mission Control has changed little. Historically, to share information on a computer display, mission controllers used “channel attach.” A digital television equipment (DTE) guide showed a list of available channels. Anyone in Mission Control could bring up a display from that channel on one of their screens by entering a number into a hardware keyset. The granularity of this sharing was at the display level. If you needed any piece of information, you had to take the whole display. The alternative was to build your own display with only the few parameters you needed—a process that took months.

Other ways to share information included a camera channel, which allowed an operator to focus a camera on a table and show whatever information could fit into the space on a TV channel. Last but not least, pneumatic tubes allowed operators to send physical objects between consoles; typically those were hard copies of displays.

The current version of Mission Control has modernized the technology to client-server systems running on X-Windows and Linux, but sharing is still done at the granularity of a complete display. Also available in Mission Control are PC-based systems that are part of the non-critical flight network and provide other options such as file exchange and e-mail. Virtual
Network Computer (VNC) has been used in some systems to provide sharing at the screen level.

In contrast, MCT operates at the level of user objects (described below) and collaboration is enabled at the same fine-grained level as is composition—at the level of user objects. This gives users almost unlimited control over what they share—from individual telemetry parameters to whole composed displays.

4. MCT Overview

From a user’s perspective, MCT is a collection of composable user objects, which we sometimes refer to as “things.” A user object is a functional piece of software, such as a telemetry value, a procedure step, a timeline, or an activity. It is something that users can manipulate, compose, inspect, share, and edit. All the MCT user objects are presented to users in a single unified environment (Figure 1).

There have been multiple past approaches to compositing, manipulating, and sharing information. A comprehensive review would be beyond the scope of this paper. MCT took its principal early inspiration from Apple and IBM’s OpenDoc project [1, 2, 3]; from Smalltalk/Squeak [4, 5]; from Bellcore’s multiplatform graphical user interface style [6]; and from the IBM OS/2 Workplace Shell, which was part of IBM’s broader Common User Access object oriented style graphical user interface guidelines [7].

We started with a small set of core, simple ideas that remain the driving goals for MCT:
- Compositions instead of applications: Software shall be compositions of user-composable objects.
- The user shall not have to start and stop applications; they shall have all the user objects they need in one user environment, from which they may create functional compositions by drag and drop.
- User objects shall be as fine-grained as necessary for user tasks. Composability and sharing shall be possible at the level of individual user objects.
- Instead of creating pre-packaged software in the form of monolithic applications, MCT shall present the user with a single environment (though not restricted to just a single window) that contains the objects they need to perform their tasks.

5. MCT’s Users and Their Need for Collaboration Support

There is a wide range of potential users of software built with MCT technology. The particular products currently being built on that foundation are for the ground-based flight controllers of NASA spacecraft, starting with controllers of the ISS. Each ISS flight controller serves in a job—a “discipline” such as PHALCON that manages the ISS power. Most disciplines have computer workstations in at least one console in a Multi-Purpose Support Room (MPSR). Some disciplines also have a console in the Flight Control Room (FCR). There can be more than one flight controller sitting simultaneously at a given discipline’s MPSR console, and those people must collaborate with each other in person and by sharing information across their individual workstations.

In contrast, in the Flight Control Room (the room seen on television) usually there is only one flight controller at a time at each discipline’s console. But that person must collaborate with the other flight controllers from the same discipline who are sitting in the MPSR. Collaboration between those rooms is by voice loop audio circuit, telephone, and computer-mediated communication.

All flight controllers in both rooms must collaborate with flight controllers from other disciplines, both in person and remotely. Even when flight controllers are sitting in the same room, sometimes they have difficulty collaborating in person because their consoles are too far apart.

A major collaboration need is for all flight controllers to see the same, accurate information despite the different configurations of that information on their displays. There is a tension between the flight controllers’ need to see that information in consistent ways to facilitate pattern recognition, and their need to rearrange that information to support non-nominal-situation decision-making.

6. MCT’s User Experience Style

A user object is any chunk of information that users can view and manipulate as a concrete “thing” to some degree independently of other information. Some examples are ISS telemetry elements and collections of telemetry elements, but MCT objects could be physical parts, procedures, steps in procedures, commands, or documents, or people—all would be treated identically as user objects. Figure 1 has examples of user objects that are collections (e.g., Subsystem B) and telemetry (e.g., PDU2 VTC1 Pwr Bus On Off Stat MDM). Here is how those user objects are shown to users:
- User Object ...
  - has a single Nucleus that users cannot see, ...
    - which appears in the user interface as one or more Manifestations, ...
    - each of which is filled with one View at a time, but can be switched to other Views.
Figure 1. Multiple presentations of user objects, united in user interface style and in the underlying architecture and code.

Each user object’s “nucleus” is the user’s mental model version of the “model” in the developer’s model-view-controller design pattern. Users see the information that is contained in the nucleus only in an enclosure (the “manifestation”) such as a table row, tree row, icon, whole window, or panel as exemplified in Figure 1. Looking at different manifestations of a single object is like simultaneously looking at two windows of the same document within a word processor. A “manifestation” is the user’s mental model version of a “view” in the developer’s model-view-controller design pattern. Changes made to nuclear attributes that are made in any of that object’s manifestations are reflected in all manifestations of that object seen by all users. This means users never have to go find the “real” object first, because no manifestation is any more real than any other; all the manifestations equally are aliases for the one nucleus.

Removing a manifestation, such as closing a window or removing a tree row, does not affect the object’s nucleus. But deleting the object’s nucleus causes all manifestations of that nucleus to vanish, because there is nothing left for them to manifest.

Figure 2 schematically illustrates the relationships among a user object’s nucleus, manifestations, and views. “User Object A” is a telemetry element that has a voltage property. Its nucleus is stored centrally (thus available to all users) and is not directly visible, so it is shown in the figure as an oval rather than a user-visible window or panel. The nucleus stores everything that makes up that user object:

- **Characteristics of User Object A**
  - **Properties:**
    - Units of the data value of object “45”
    - Base displayed name of “User Object A”
  - **Children** (related user objects):
    - User Object B
    - User Object C
- **View definitions—views of the above characteristics:**
  - “Canvas 1” view, shown as left oval in big oval
  - “Canvas 2” view, shown as right oval in big oval

None of the above can be seen directly by users. What can be seen directly are manifestations of User Object A. Figure 2 shows two window manifestations of User Object A above the nucleus oval; the object’s name is leftmost in the window’s title. The lower left window is filled with the “Canvas 1” view whose definition is shown as the left oval inside the nucleus. The upper right window is filled with the “Canvas 2”
view. The view names are the right portions of the windows’ title bars. User Object A is a child of User Object D, which manifests as the top left window, showing its “View 3” view. (Its nucleus oval is not shown here.) In that window, User Object A manifests as a panel that is filled with the “Canvas 1” view.

All those terms and concepts are from the users’ point of view. How those are architected and implemented may or may not map directly onto those GUI terms and concepts [8]. For example, “view” in the GUI is not necessarily synonymous with the under-the-covers term “view” in “model-view-controller.”

7. Collaboration Support by MCT

The manifestations (e.g., windows) of a given user object do not directly depend on each other. But they stay synched with each other by staying synched with the nucleus. That is true even of manifestations on different users’ workstations, because objects are stored centrally. This supports users collaboratively seeing and editing common user objects. But manifestations of the same object do not entirely match, because superficial aspects of the manifestations are independent; some examples are the sizes, shapes, and scroll bar positions of windows.

Those inherent abilities for all users to simultaneously see and edit the same user object are restricted in practice. Visibility of objects is restricted for reasons of security, privacy, and safety. Policies allow fine-grained control over which objects can be seen by whom under what circumstances, and even over which characteristics of an object can be seen by whom under what circumstances.

Ability to edit objects is restricted for the same reasons that viewing is restricted, but also because the ISS flight controllers highly value the stability of their displays and want to encourage the user who is editing to carefully check the changes before forcing those changes on other users. So MCT’s policies distinguish between private objects—those visible even in principle to only one user—and public objects—those visible in principle to multiple users even if only one user happens to have that object displayed at the moment. When a user has multiple manifestations of their private object on a single workstation, all those are editable and changes made in one immediately appear in all the others. That is because only that one user can see the changes to a private object, and that one user presumably knows what they are doing in all their windows.

But MCT’s current policies for JSC restrict a public object to having only one manifestation editable (“unlocked”) at a time, and for the editing user to explicitly “commit” those changes to the nucleus before those changes appear in other manifestations. That policy of one unlocked manifestation at a time applies even to the manifestations of a shared object on the same workstation, just to keep the user’s mental model simple. Allowing only one unlocked manifestation at a time also prevents clashes of edits that otherwise could occur by one user changing something that another user already has changed but has not yet committed.

MCT also enhances collaboration among developers of additional MCT abilities, and among NASA centers, as described in [8].

8. MCT’s Objects

8.1. User Objects versus Mashups and Web Widgets

A common form of user composition today is the mashup of web widgets, each piece of which is essentially an HTML representation of a URL. You might consider each widget to be a view, but if so it is a view of another web page or maybe an application, but usually not a view of an underlying user object and usually not even a view of an underlying system object. Web widgets can be almost anything… a list of documents on a server, a calendar, e-mail, a weather report, or a satellite ground track. Web widgets function as independent windows composed in a container, rather than as MCT’s related set of user objects that simulate their real-world counterparts.

The underlying “objects” on the web page don’t communicate with each other. For example, I can’t add a URL for a stock ticker and then somehow flip the view to a 2D plot for the stock without a lot of work. While the mashup is a form of composite functionality, its behavior lacks the consistency of user objects. MCT’s user objects represent real objects from the applicable domain, and they behave according to a consistent set of rules.

Mashups thereby do not well support the presentation of user objects as coherent objects. That incoherence prevents users from semi-consciously relying on their mental models of their work domain. Instead users must consciously map their mental model of a given work domain user object onto a fragmented set of Web widgets. Of course it is possible to use Web widgets to present user objects coherently. For example, with REST, the representation of a resource could be a representation of that resource as a user object. Whereas the MCT framework creates the structures needed to represent user objects, web widgets do not. Coherence and consistency of user object behavior in web widgets would have to be created by developers.
Figure 2. Schematic of user object nucleus, which users cannot see directly; window and panel manifestations of the object; and views that fill the manifestations.
The infrastructure for user objects already exists in the MCT technology. MCT is a platform that maintains the coherent identity of each user object that it manifests in the GUI, and that allows users to manipulate those coherent objects by way of manipulating their multiple GUI manifestations.

The most fundamental barrier to implementing MCT in web browsers is performance. Our tests show a two-orders-of-magnitude deficiency in drawing the thousands of frequently changing data plot lines that are needed for monitoring ISS, alongside the thousands of changing alphanumeric values. In contrast, MCT is scaled sufficiently large to handle the huge display requirements of ISS operators even with modest computing hardware. For more about why initially we are not using web browsers as MCT’s platform, see [8].

Nonetheless, the MCT team is evaluating the feasibility of using web browsers, mobile phones, and other devices as platforms for subsets of MCT functions—providing such devices with information from web services provided by the main MCT installation. We have demonstrated the display (not composition) of MCT user objects in a browser and on an iPad.

8.2. Programmers and Composers

In MCT, every user interface entity is a user object and every user is a potential composer. In Smalltalk [4], everything is an object and every user is a potential programmer. With regard to Smalltalk we are talking about “object” from a programmer’s perspective, not from a user’s perspective.

The early MCT prototypes were written in the Squeak implementation of Smalltalk [5]. Squeak provided a fast path to developing the core composability of MCT, and its ability to quickly modify objects was a significant advantage in that context. Everything was a composable user object and also a Squeak object. The question of how closely the implementation object model and the user object model should correspond is beyond the scope of this paper. However, based on the rapid development of our Squeak prototype, it appears that a close correspondence is an advantage.

The underlying assumption that everyone is a potential programmer who may want to modify the behavior of their objects does not hold in mission operations. We don’t think any of us would want to get on a plane thinking that the flight crew had just modified the core behavior of the objects that make up the plane’s flight control software.

In Mission Control, software is certified to ensure accurate information is presented to the flight controllers. Using MCT, that certification will be done at the user object level (see Figure 6) rather than always at the complete display level as is current practice. The certification status of each user object is stored as a property of that object, and so is used by the MCT system applying the relevant policies to control visibility and modifiability of that object, in just the same way that other policies use other properties. Both the container user objects and the atomic user objects have their own certification statuses. Big savings in time, expense, and risk of certifying new displays come from composing new displays (container objects) from already-certified objects (either atoms or other containers); only the new, surrounding display object need be newly certified. One example of a certification workflow is described in [8], but others can be built simply by writing new policies, perhaps creating other properties to be used by those policies, and perhaps creating other containers for holding objects having different values of those certification properties.

8.3. Nearly Everything is a User Object

Which information qualifies as an MCT “user object” depends on the users’ tasks, situations of use, and mental model of their task domain. In contrast to Squeak, MCT does not have an under-the-covers object model where everything created by developers is an object. MCT is written in Java, using the MCT component model. Typically each MCT component is a user object, but that is not enforced in the code. Deciding on the user objects can be done only by design process; the major process the MCT team uses is the “The Bridge” [9] participatory design method. A user object is a chunk of information that the user wants to see as a coherent whole, and possibly wants to create, move, copy, edit, and delete. An example is that a car’s tires are child objects, but the color is just a property.

The bottom line is that nearly everything users think of as a “thing” ends up being presented as a user object, because that makes the user interface match the users’ mental model of the work domain. All manifestations can be treated the same way insofar as the GUI fundamentals go—for example, inspected, opened into windows, views changed, dropped into other objects to form compositions, and resized. MCT’s simplifying, bottom-line guidance for users is that if something looks like an object, or if the user’s domain knowledge makes the user think of it as an object, then the user should try treating it like an object. Even if it is not and object, the user can’t hurt anything by trying. MCT’s motto is “A thing is a thing is a thing.”
9. Architecture

Architecture is not the primary focus of this paper, but we will give a brief overview; for more see [8]. MCT uses industry standards and avoids depending on proprietary technologies. MCT is written in Java (Figure 3). We use Eclipse as an integrated development environment (IDE), but MCT does not utilize Eclipse-specific features such as the rich client platform, so MCT is not bound to Eclipse. We have also demonstrated the ability to use NetBeans. GUI components are written in Java Swing. Java was chosen because it is a multi-platform enterprise-class language with industry-wide support.

The architecture is modular. The plug-in architecture is the Open Service Gateway Initiative (OSGi) (Figure 3), which gives us significant operational flexibility by allowing development of functionality in OSGi bundles. For example, the plot capability is a bundle that can be added and removed easily. So far we have used that capability to simplify development and testing, to deliver different functions to different NASA customers, and to troubleshoot if one plot package has a problem.

User compositions are stored in a MySQL database (redundantly), and local copies of metadata are stored in a Berkeley database (Figure 4). An API is provided so developers can write their own components. Our internal developers use the same API that external developers do.

MCT uses a simple component model that "wraps" the model and its views in a component. Typically a component corresponds to a user object’s model (nucleus) and all of its views (manifestations). Component reuse is achieved by building components that are applicable to the space domain across multiple mission classes. A simple example is a telemetry component. Once a telemetry component is built, it may be reused for different flight control uses on the same mission and even for different missions. The advantage of reuse is that where once multiple applications were required, now the same user object can be reused in different operational contexts. In our example, a telemetry user object may be used for spacecraft health monitoring, and may also be embedded in a procedure to verify the steps, but without reuse the telemetry would have been recreated in a separate telemetry application.

Tags of objects and of individual values of attributes support not just predefined organizations of objects, but also searches and ad hoc organizations created by users, not just by developers.

User objects can communicate directly with each other. For example, a telemetry object can have its data value translated from a number to a string for display (e.g., 1 displays as "On" and 0 as "Off") by that object being associated with a display enumeration object that inputs the telemetry object’s numeric value and displays the mapped string.

We designed MCT from its inception to be general enough for space mission operations not just for ISS, but for any mission at any NASA center, university, company, or other organization. Much of that extensibility is provided by the ability to add and replace plug-ins, but with such a broad target market, eventually MCT’s framework must be revised. The core MCT team at NASA Ames Research Center will maintain the integrity of the platform, and manage contributions to the platform and to the pool of plug-ins similar to what is done for other open source projects such as Mozilla. For example, the MCT team already has a user experience design guide.

10. Other Component Software

Comparison of MCT to other component software would take an entire paper devoted to that topic, but here we compare it to at least a few. A difference between MCT and mashups is that MCT allows users to configure and persist objects and views, but mashups are isolated from each other. The only commonality across the pieces in a mashup is their presence in the same web browser page.

MCT meets [10]’s definition of extreme tailorable: composability of existing user objects into other objects; extensibility by adding user objects and adding views to existing objects; and vertical openness by allowing plug-ins from different software producers. MCT’s use of OSGi allows plugging and unplugging without restarting MCT. MCT even allows plugging in software that was not developed with MCT in mind, though in that case the MCT developers must do at least some work to encapsulate that software to become an MCT plug-in; for example, an MCT developer once replaced MCT’s commercial plotting package with another after only a couple hours work.

A difference between MCT and CoCoWare [10] is that MCT’s components are representations of chunks of information that are meaningful to users in the context of their tasks and work situations [9]. But CoCoWare’s components are whole applications. Note that a software component is a unit of composition. Beyond that simple definition, the granularity and functionality of components varies widely. Although FlexiBeans components are more fine-grained than CoCoWare’s components [11, p. 17], they still are mostly functions or, if data, usually they are decomposed or are technical underpinnings of what MCT presents as user objects.
That difference between MCT and other approaches is due to differing approaches to tailoring. MCT does not try to let users work very deeply in what traditionally has been the domain of professional computer programmers by directly interacting with the computer. MCT does let users intrude somewhat into that arena, but mostly by creating data objects and views of predefined types (e.g., creating an empty container or evaluator, creating a plot view); adjusting settings of predefined...
attributes of those; and creating directional, labeled relationships and communication among user objects. MCT definitely does support deeper and more technical tailoring by users, but not by having users do it themselves—instead by having users design collaboratively with the MCT team’s professional developers and user experience designers as described in the next paragraph. That approach was chosen quite deliberately [12].

11. Deployment and Use

MCT was designed not simply by responding to a list of users’ requirements, nor even based on simply observing and questioning users. Instead, MCT was designed in collaboration with users, with users helping construct and test dynamic paper prototypes [9], then testing computerized partial prototypes, then testing a series of initially rough and then progressively more complete versions of the real coded product, often with the nightly build. In addition, over a hundred users gave feedback to demonstrations of MCT. Much of that was done in labs in JSC’s MCC, over several years, until testing, improvement, and completion were sufficient to deploy in the MCC’s uncertified operations area at the end of Summer 2010. MCT was exercised and evaluated there by a small number of flight controllers for nine months, and then access was given to more than a dozen others. Certification for mission use will take place in 2011. Operators are now building their first displays for real use instead of evaluation. These early displays closely mimic existing displays, but we expect that as operators become familiar with the power of fine-grained compositions, they will build novel ones.

Three levels of permissions are defined corresponding to the user organization’s policies, allowing users to be composers, administrators, or users without composition privileges.

How much will users act as software composers? How many users will compose rather than merely consume? Will the new abilities facilitate large-scale collaboration across the NASA centers and other institutions? Will knocking down the walls of monolithic software and replacing them with composable user objects help development of new operations concepts and capabilities? These questions will be answered in the coming months and years.

As of Spring 2011, we have demonstrated enough of our platform that enables software to be built from user compositions, that we have won acceptance from sufficient key stakeholders to fund and deploy that software and to begin its use in an operational environment. Nearly all users have been enthusiastically asking for MCT to be available to them as soon as possible. That is largely because MCT was not designed and built first and then thrown over the wall to see what users think. Instead, we used an iterative, participatory approach to ensure and improve users’ satisfaction during the entire design and development process.

Of course, with expansion of the number of actual users come more feature requests, bug reports, and changes of opinion about aspects that seemed like good ideas to the flight controllers when they were designing and then just playing with the prototypes. Fortunately, MCT’s modular GUI style and underlying modular architecture and implementation allow us to continually meet those challenges, and the requirements, and the capabilities, and the changes, and the requests, and the desire.

12. References