View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

# Agile Development Methods for Space Operations

Jay Trimble
*NASA Ames Research Center, Mountain View CA, 94035, USA*

Chris Webster
*University of California Santa Cruz, NASA Ames Research Center, Mountain View CA, 94035, USA*

**The Mission Control Technologies (MCT) Project, developed at NASA Ames Research Center in collaboration with Johnson Space Center (JSC), has shortened its software release cycle from months to weeks. This agile development cycle is similar to state-of-the-art practice in industry, with modifications for safety in a mission operations environment. Software builds are made available to the customer for evaluation daily. The customer can download these nightly builds to provide feedback on new features as they are rolled out. Formal deliveries are made every three weeks. A formal release for operations is ready every three months.**

## I. Introduction

MAINSTREAM industry software development practice has gone from a traditional waterfall process to agile iterative development that allows for fast response to customer inputs and produces higher quality software at lower cost. In consumer software, we see increasing trends towards agility and short release cycles. For example, Google releases a new version of the Chrome web browser in periods measured in weeks, not months or years. On the web, we see software deployed early for user evaluation, typically paired with analytics to evaluate usage patterns. How can we, the space ops community, adopt state-of-the-art software development practice, achieve greater productivity at lower cost, and maintain safe and effective space flight operations?

At NASA Ames, we are developing Mission Control Technologies (MCT) software, in collaboration with Johnson Space Center (JSC) and, more recently, the Jet Propulsion Laboratory (JPL). To meet customer requirements on schedule and within budget, we have built an integrated agile software design and development process that is much like modern industry practice, with modifications to ensure safety for space operations.

We have replaced the traditional requirements process and documentation with a participatory design process in which the customer acts as domain expert and design experts facilitate the process, starting with simple paper prototypes, then high-fidelity specifications, then code. We deliver an iteration to our customers every three weeks, with a full release every three months. Nightly builds are made available to facilitate daily customer feedback on our progress. This replaces traditional documentation and extensive reviews with direct and ongoing interactions in which progress is always visible. Technical documentation is maintained on the team Wiki, to which everyone may contribute.

Our software build process is automated. Each code commit triggers a build that runs the automated test suite, does a static code analysis, measures code coverage, and produces a binary distribution.

## II. Why Agile?

Agile development addresses some specific shortcomings of the traditional waterfall software development model. Agile shortens the delivery cycle, thereby breaking large software development projects into smaller, more easily manageable pieces. Agile facilitates direct and ongoing interactions between the developers and the customers, leading to more effective solutions. Using agile methods, software is available for download daily, so developers and customers always have the means to assess the state of the code.

### A. Replace Predictions with Actuals

In spite of decades of attempts to develop techniques to accurately estimate software development time, it remains very difficult to reliably predict how much software functionality can be developed in a given amount of

time. Agile development cycles mitigate the risks in predicting the time to develop software. Progress is measured by the state of the code, rather than through estimations and presentations. For MCT, this takes three forms. First, there is the nightly build. The customer can download and use the software daily, where they see the state of the software directly. Next there are iterations, which are delivered every three weeks. These are typically installed in a mission control test facility. After four iterations there is a release, which is ready for operational mission control certification.

### B. Manageable Deliveries

The longer the delivery cycle, the more code, the greater the complexity, and the larger number of tests that must be conducted. A longer cycle means more time between the specification of function and the delivery. The greater the time from specification to use, the greater the potential mismatch between user expectations and the actual product. A shorter cycle eases the testing burden by reducing the number of new features and potential regressions for any given delivery. While agile may not change the total number of tests over time, it distributes them into a manageable size for any given delivery.

### C. Development Team/Customer Interactions

The short delivery cycle of agile, coupled with the constant availability of working code for customer evaluation and feedback, makes possible closely coupled interactions between the development team and the customer. Customers have the option of downloading a build daily, from which they can provide daily feedback on feature design and bugs to the development team. This developer-customer interaction is particularly useful to verify new features as they roll out.

### D. Fast Response to Change

Agile’s shorter development cycle makes the team better able to respond to changes. The software is rolled out in stages, the users respond to what they see, and the development team can act on the feedback. This also helps with planning by using the software to determine when functionality is complete or needs more work.

### E. Team and Organizational Culture

The short and flexible cycles of agile development lend themselves to adaptation to the culture and expectations of the organization and the project. Note that there is no one or right way to implement agile development. While there are some core principles to adhere to, the specific cycles and modes of operations must fit the team(s) involved, the organizational culture, and the project’s goals.

## III. Delivery Cycles

The examples below show the MCT release cycle, iteration detail, and strategic road map. This is only one example of how to implement agile software development in a mission operations environment. As stated above, specific implementations of agile must be adapted to fit team and organizational culture and project goals.

### A. Iterations and Releases

Figure 1 shows the MCT release cycle. MCT is delivered to customers in three-week increments called iterations. Four iterations constitute a release. A release takes three months. The reason for the three-week iteration cycle is to allow enough time to add features that are meaningful for customer evaluation and verification, but to have a short enough cycle to keep testing manageable.

Iterations are installed and deployed in a test and evaluation area in the mission control environment, but not in the operational environment. Releases are ready for certification and installation into mission control.

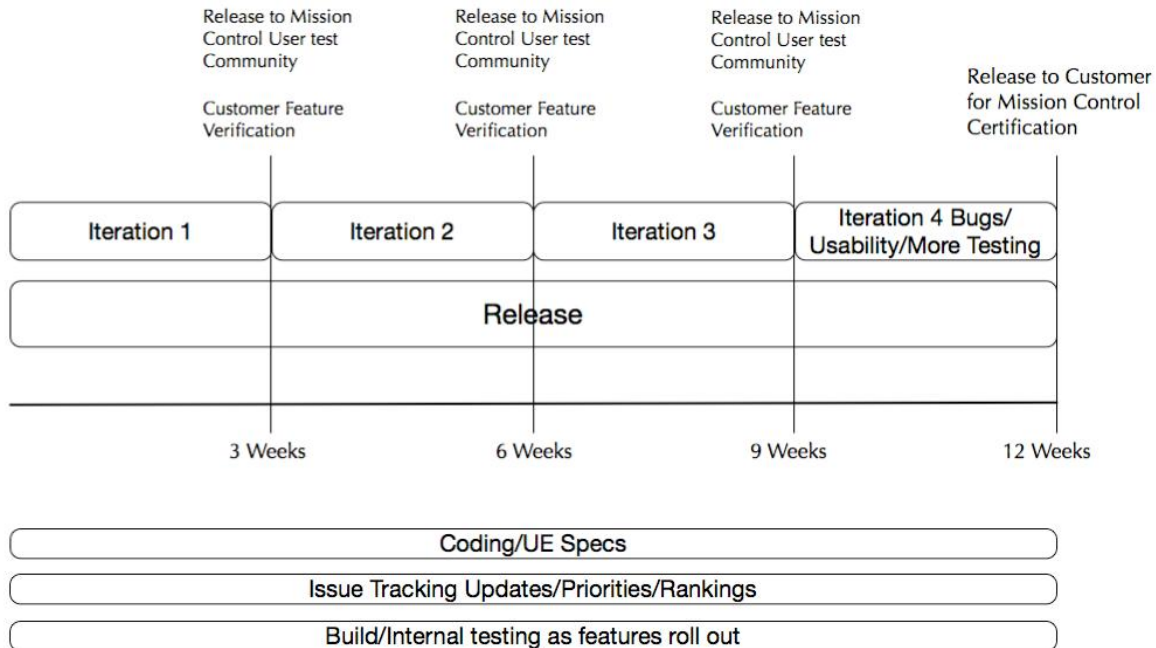


**Figure 1. The MCT release cycle**

### B. Progression

During a release, tasks should be worked in priority order. This may be counterintuitive, as the burndown rate can increase by working on easy tasks first; this temptation needs to be resisted to ensure the delivered software always contains the features that are most important to the customer. This will also help ensure features that need additional maturation can be worked on later in the release to improve customer usability. Harder features rolled out for testing by the end of iteration two have the highest probability of being mission ready. The emphasis for iteration four should be on testing and reliability. Ideally no new features would be developed in iteration four, but if this is not possible, any new features should be limited to simple ones. Note that difficult features may span multiple iterations or releases.

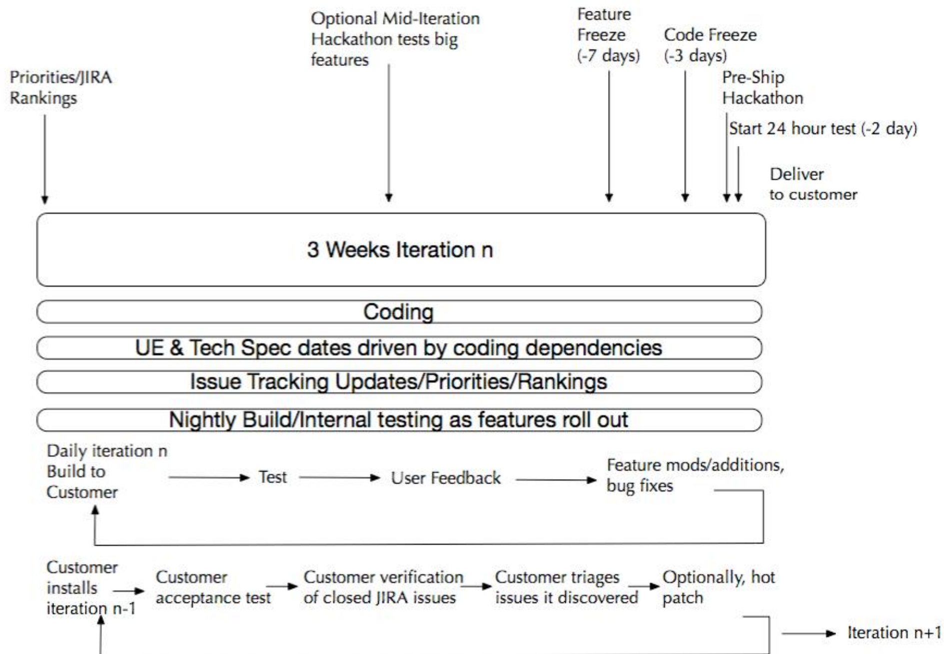


**Figure 2. Iteration detail**

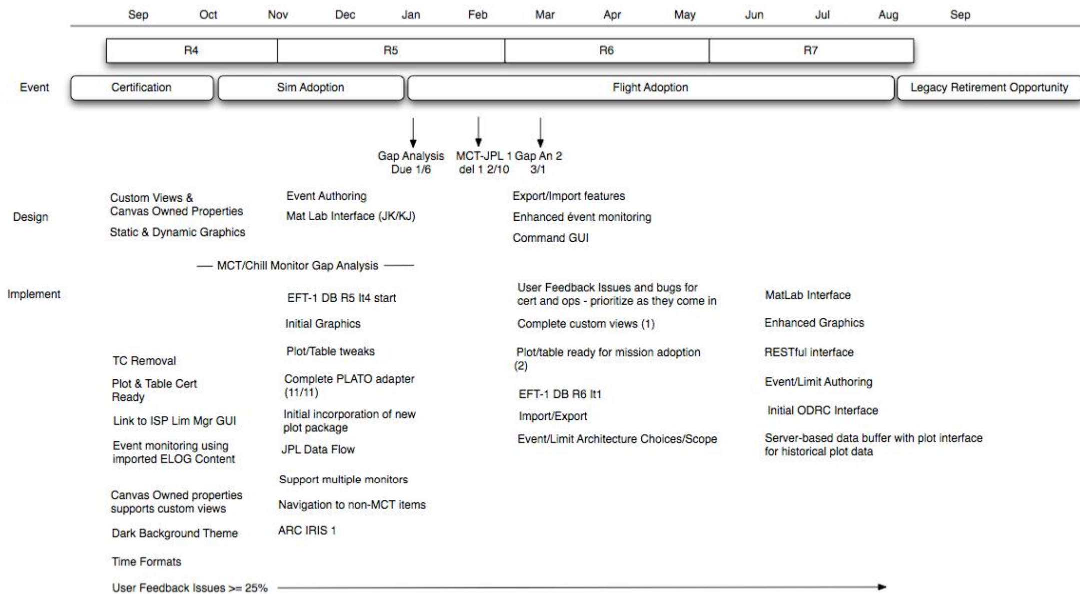


**Figure 3. Strategic road map**

### C. Iteration Detail

Figure 2 shows the detailed flow of an iteration, and the relationship of a current iteration, which we call iteration n, to the previous (iteration n-1) and subsequent (iteration n+1) iterations. A key principle of agile development is to always be working on the highest-priority issues. At the start of each release the issues are ranked by priority, using the project strategic road map as a guide.

Figure 3 shows an example of a road map. The map shows the high-level functional objectives relative to the timing of key events such as simulation and mission adoption, and serves as a guide for prioritization. There are discrete points where it is important to be clear on priorities. The start of a release and the start of each iteration can serve as check points for priorities. Given the continuous feedback loop between developers and customers, prioritization of issues is ongoing and happens continually. Be wary of altering the strategic road map. Such alterations can come at the price of team productivity.

### D. Coding is Ongoing, All the Time

Developers like to write code. They should be able to code constantly. However, code for delivery must fit into team guidelines. For each iteration, we have a feature freeze seven days before delivery, and a code freeze three days before delivery (see Fig. 2). The developers may still code for subsequent iterations after a freeze for the current iteration.

### E. Issue Priorities and Rankings

Traditional development methods are requirements driven. The team levying requirements writes a document and the developers write the code to perform the specified functions. While the MCT team has a high-level requirements document, that is not the primary driver of day-to-day work. Day-to-day work is driven by the issue tracking system. While we use JIRA for this, that is but one of many systems available.

### F. User Experience

What the developers build is specified in a user experience (UE) document. This is based on industry-standard human computer interaction methods and participatory design. We utilize a design process in which the customers participate as domain experts, and the design team guides the process.

In participatory design, we begin with simple representations of the software using sticky notes. Figure 4 shows a mockup of flight controller displays using sticky notes. The next step is a low- to medium-fidelity mockup (Fig. 5), followed by a high-fidelity UE specification (Fig. 6) which is used by the developers for coding. UE specifications are directly linked to the issue tracking system. Figure 7 shows two views of an MCT display.

The development of UE specifications is ongoing. The deadlines for user experience specifications are driven by the development dates. A user experience specification for any given feature is due by the start of development for that feature, with the dates being specified by the developers.

**Figure 4. Flight controller display mockups using sticky notes**

**Figure 5. Medium-fidelity display mockup**

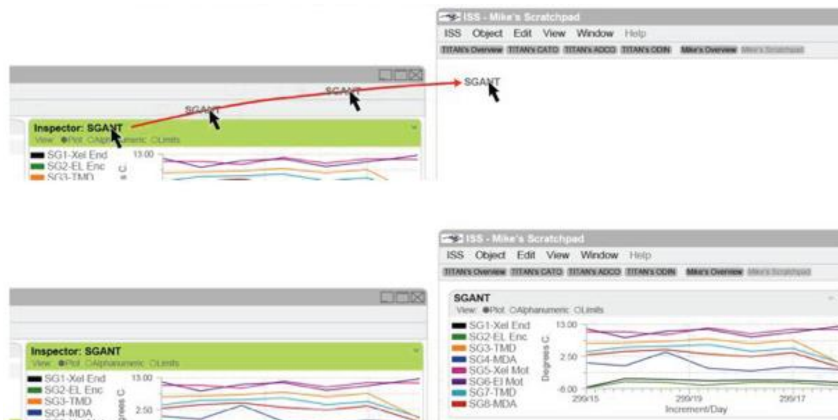


Figure 6. High-fidelity display mockup

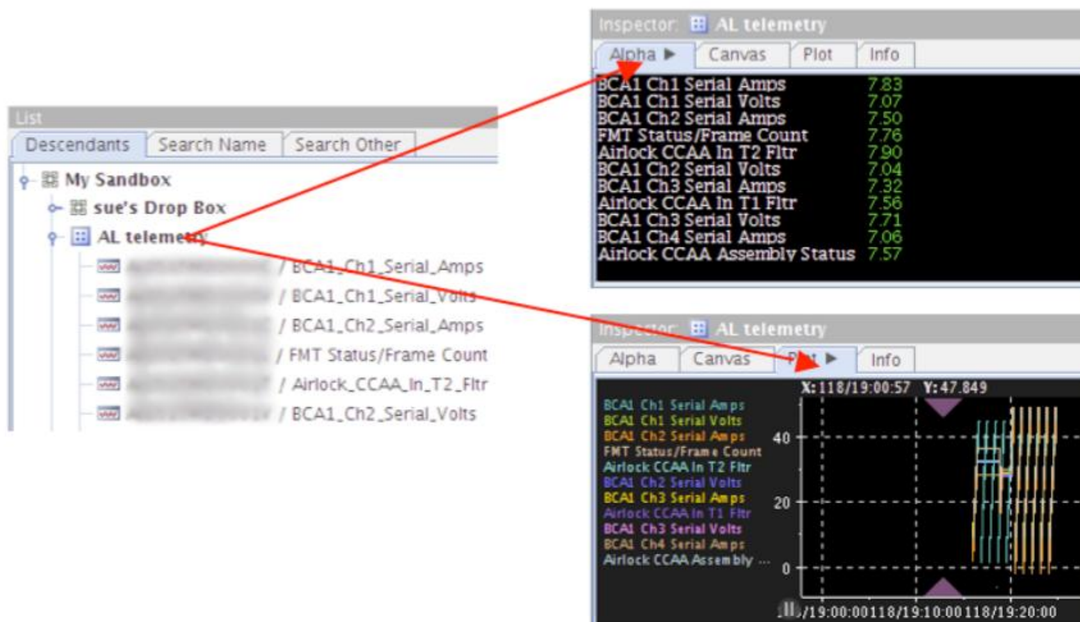


Figure 7. Completed MCT display

### G. Testing and Build Processes

Unit tests are developed along with code and must be present prior to integration of the code into the repository. Developers create white box tests including logic used in response to user interface inputs. These tests are run as part of the build, which is triggered by any integration into the repository but can also be run on a development machine to reduce the number of build failures. A build failure closes the code repository until the failure is resolved. The build consists of verifying that the code is syntactically correct, ensuring that there are no warnings, checking that the external developer interfaces are documented and have not inadvertently changed, and executing all the unit tests. The build also monitors code coverage for the unit tests and runs a static analysis on the source

code. Finally, the build runs an automated user interface test that serves as a smoke test for the product to ensure that basic user interface functionality always works. The automated user interface test suite is brittle, as the user interface is evolving; therefore, the smoke test exercises only a minimal amount of functionality.

Toward the end of an iteration, after the major features have been incorporated, a "hackathon" will take place. A hackathon involves multiple team members testing the software simultaneously in a lab environment. Before the software is delivered to the customer it must pass a hackathon, as well as a 24-hour long-duration test.

After an iteration is delivered, the customer verifies that the new features meet their requirements and that the software does not have bugs when run in the mission control test environment. So during any iteration n, the features from iteration n-1 are being verified by the customer.

## IV. Core Lessons

For agile development, there is no set routine that guarantees success. Teams, team culture, circumstances, and goals differ, and it's important to adapt the team practice to fit all of these considerations. There are some core lessons learned that should be adaptable to most situations.

### A. Regular Deliveries - The Train Must Leave on Time

It is important to maintain a regular shipping schedule. Do not delay shipment because one or more features are not complete. Features that are not ready ship in a later iteration. One approach to features that are not ready but in the code line is to allow these features to be turned off, thus a feature can be turned on for evaluation but off for general use.

Only delay shipments if there is a "blocker" bug, meaning that the software (1) simply doesn't work or (2) an enabling and required feature is not working that renders the software useless to the customer and that feature can be fixed within two days.

The key to never having to delay a shipment for a feature is that the software is shipped on a regular and frequent schedule. If a feature is not ready for inclusion in one shipment, the feature doesn't ship. But the next chance to ship is only weeks away.

### B. Internal and External Feature Testing and Verification

Test features as they are rolled out during development; don't let them pile up and wait until the end. The developers test in their development environments and in a lab environment, but not in the operational environment. The customer tests features as they roll out. If customer testing is done outside the operational test environment it can be used to provide feedback on feature satisfaction. If the testing is done in the operational test environment it provides further verification of function in the customer environment. Final verification of function in the customer environment can only be done in the full operations environment. Perform the final operational tests in the operational environment outside of an actual mission.

### C. Constantly Prioritize Consistent With the Strategic Road Map

The interactive nature of agile development means that the development team is constantly getting new requests and updated priorities. The goal is to have the team always focused on the highest-priority issues. As feedback comes in, priorities may change. Tactical priority changes that happen during a release should not violate the priorities set forth in the strategic road map.

### D. Don't Alter the Strategic Road Map on Short Time Scales

The strategic road map (see Fig. 3) points the way for the project and the team. Changes to this road map should be kept to a minimum. Ideally the road map would be re-evaluated on an annual basis with few changes during the year.

### E. Measure Progress by Working Code, not Reviews

The measure of progress is working code. Reviews and presentations are an inexact model of progress. Working code shows just what's really been accomplished.

### F. Estimations and Actuals

Once again, working code is the signpost. Over the course of development, everyone will see the rate of progress. A comparison of working functions to the road map will show the state of the project.

### G. Automate Build Processes

To achieve the speed required for agile development, automated build processes are essential. See above, Testing and Build Processes.

## V. Operational Safety Considerations

Agile development methods are commonly used in industry. There are elements of industrial practice that must be modified to safely fit a mission operations environment. In industry, particularly with web-based software, extended beta cycles are used to mature software, collect user feedback, and measure key performance indices needed for a successful operational deploy. On the web, beta software is commonly used by users to perform critical tasks. We do not have this luxury in a mission environment.

### A. Do *not* Use Uncertified Software for Mission Decisions

Software that is not ready cannot be deployed into mission operations. For agile development in a mission operations environment users need access to the software through development, but it must not be used in an operational decision capacity, meaning don't use the software to make mission decisions, use it for evaluation only.

There are multiple possible approaches here. Users can download the software in development to their work computers rather than their mission operations computers. A mission operations test and evaluation area can be set up to use the software outside of an operational decision area. The software may be made available for restricted non-flight activities in a mission operations area, for non-mission-decision evaluation.

### B. Updates Must be Planned and Visible

Industry practice for software updates varies with usage, operating system, and deployment environment. Web software updates are done in the background, typically with no effort on the part of the user and often without the user's knowledge. A user of web software often logs on one day and finds features and changes that were not there the previous day.

For client software, standard practice is to prompt the user that updates are ready for installation. However, there are instances, such as Google Chrome, where the updates are done in the background with no effort or permission required from the user.

For mission operations, users train to a set of software features. We do not consider it safe to perform updates without user knowledge and understanding of changes. For mission operations, updates should be planned, so users can be notified and trained on new capabilities before mission use.

## VI. Estimates for Management

The predominantly hardware-based culture of spaceflight expects firm schedules for deliverables, with ship dates and adequate margins. When a launch date or a key mission event is driving software requirements, the readiness dates must be met. To achieve this, adequate margins must be present. Given the declining budgets in the current space program, proper margins for traditional software development are ever harder to maintain. With agile development the problem of predicting the rate of progress over a long period of time is lessened by the ability to always show the customer the state of the software functionality. While adequate margins are still required to meet firm operational dates, the problem of prediction is lessened by the ability to constantly monitor progress.

## VII. Conclusion

Agile development can provide numerous benefits over traditional development methods. Our experience has shown that agile development produces a better solution for the customer, creates a more effective and unified customer-developer team, and produces better results at lower cost than traditional methods. A common assumption is that agile development is faster than traditional development. However, agility refers to the ability to act and move quickly, to react to situations and change if needed, rather than sheer speed. The focus on working software over documentation increases developer productivity. The developer-customer team can react and move more quickly with agile, and because the process is iterative, continuous, and integrated, the result is a more productive and effective team and product.