Multi Operations Specimen Tester MOST Framework for AIT Testing and Operations

Antonio Pepiciello[(1)], Dr. Carsten Reese[(2)], Dr. Gilberto Arantes[(3)]

(1)
Universitätsallee 27-29
28359 Bremen
GERMANY
Email: antonio.pepiciello@ohb.de

(2)
Universitätsallee 27-29
28359 Bremen
GERMANY
Email: carsten.reese@ohb.de

(3)
Universitätsallee 27-29
28359 Bremen
GERMANY
Email: gilberto.arantes@ohb.de

INTRODUCTION

MOST was born of frustration.
In the early 2012 software AIT team started building tools in TOPE language to automatize test procedure and to support activity in clean room. The objective was test different spacecraft models using the **C**entral **C**heckout **S**ystem (CCS) by TERMA. Each tool was more focused on the specific model (AVM, EM, PFM, FM) than on the problem it was supposed to solve. It ended up re-building for each satellite model the same tool. This became increasingly expensive in term of hours and frustrating.
The only solution was a component approach based on reusability. Rather than building an application as self-contained monolith (a huge ball of mud), there was a need to find a way to divide applications into smaller, reusable components. The concept of **A**dva**N**ced **T**ester (ANT) was born. Ideally each ANT would be small enough to be implemented in short time, it would be capable to fix one specific problem (logging, monitoring parameters and events), and complex and interesting applications could be created by assembling ANTs: Anthill paradigm.
The component-based approach requires a powerful glue for assembling the components, and a shared software framework. It was based on best practice and patterns, could provide that glue. Out of this thinking grew the **M**ulti **O**perations **S**pecimen **T**ester (MOST) framework. In this paper, a specimen is defined as the subject under test e.g. spacecraft, SCOES, payloads, etc.

ALLEVIATING THE COMPLEXITY OF EGSE TOOLS WITH FRAMEWORK AND PATTERN

A framework is a reusable, “semi-complete” application that can be specialized to produce custom applications [1]. A pattern represents a recurring solution to a software development problem and helps to build on collective experience.
Complexity is a measure of understandability, and lack of understandability leads to errors [2]. A software system that is more complex may be harder to specify, harder to design, harder to implement, harder to verify, harder to operate, risky to change, and/or harder to predict its behaviour.
It is important to distinguish between “essential complexity” and “incidental complexity” [3]. Essential complexity arises from the problem domain and mission requirements, and can only be reduced by de-scoping. Incidental complexity arises from choices made about design and implementation, and can be reduced by making wise choices. MOST has been designed to manage incidental complexity. MOST’s architecture provides helpful abstractions and patterns that promote understanding, solve general domain problems.
In MOST’s design, unnecessary growth in complexity has been curtailed in particular by:

- Modular software architecture to isolate concerns and allow composition of individually tested elements.
- Use design patterns to capture sound solutions to recurring engineering needs, so less time spent reinventing wheel, freeing resources to work on more important things.

Moreover MOST constitutes a “grass roots” effort to build on the collective experience of skilled **A**ssembly **I**ntegration **T**est (AIT) engineers and software engineers. Such experts already have solutions to many recurring problems in AIT/EGSE. MOST captures these proven solutions and aims to provide a common set of building blocks called AdvaNced Tester (ANT) to develop automated scripts, as well as complete implementation of reports and events monitoring.
Monitoring and control a specimen during AIT/EGSE tests is a complex problem due to the huge number of information to have under control. Moreover as the size of the information increase (more telemetries, events, telecommands and out of limits) the perceived complexity increase exponentially due the human cognitive limitations.
MOST framework is an abstraction of the real monitoring and control, and it is constructed by leaving out unnecessary details and complexity. Abstracting helps to see the forest for the trees, allowing the tester to focus on, capture, document, and communicate only the important aspects of the system under test.

SOFTWARE ARCHITECTURE

MOST has been divided in three components or layers implementing the **M**odel **V**iew **C**ontrol (MVC) pattern [4]. The aim is to make the application in the problem domain independent from the user interface and the specific **M**ission **I**nformation **B**ase (MIB).
The tree components are:

- The **model** encapsulates core data from the Mission Information Base and functionality. The model is independent of specific output representations or input behaviour.
- The **view** displays information to the user. A view obtains the data, it displays the data from the model. There can be multiple views of the model.
- The **controller** monitors the changes of the entity in the model (telemetry changes) and notify those changes to the view.

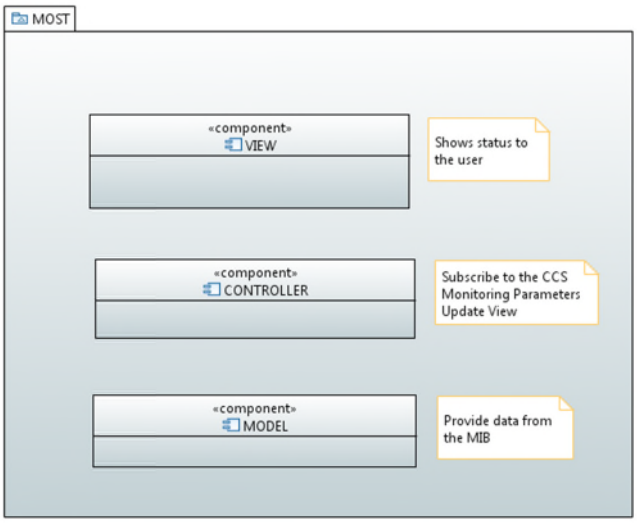

Fig. 1. Model View Controller (MVC)

An easy way to understand MVC: the model is the associated processing, the view is the window on the screen, and the controller is the glue between the two.
The three components give a static view on the design and are the starting point to define the lower-level ones, down to the software interfaces/classes. The top-down approach is vital for controlling complexity.

Interfaces Context

Underneath MOST is the TOPE language. TOPE stands for “**T**est and **O**perational **P**rocedure **E**xecutive”. MOST uses TOPE as a plug to connect and interact with the Central Checkout System (CCS).
The TOPE language was designed as an extension to TCL (**T**ool **C**ommand **L**anguage) by Siemens Austria for the European Space Agency and deployed in partnership with Terma for numerous European spacecraft test systems. TOPE is designed to support automated testing, commanding and monitoring of spacecraft and their subsystems through exchange of telecommands and telemetry, in the form of packets. Incoming telemetry is transformed into telemetry parameters through a process of extraction, calibration, and limit checking. Telecommands, which are sent as binary

packets, can be generated from mnemonic information. The transformation between binary and symbolic information and vice-versa is configured using a MIB.
Any test script based on MOST is an application with an embedded TOPE interpreter. The test script can be extended and customized by the end user (operator, test conductor, test engineer) in various ways. When used to its fullest, the running test script based on MOST provides much more than a GUI to show the result of a test, especially in the early stages of the test process. It can be used to isolate and test a subroutine in a library, to quick debug TCL code, to notify a real-time value of a telemetry parameter, to send telecommands, to extend the log with users' comments, etc. MOST has a well-defined interface making it easy to extend with new mission specific libraries.
MOST does not access directly to the MIB files or to the telemetries values stored in the archives, but it uses the CCS TOPE package to have those services.

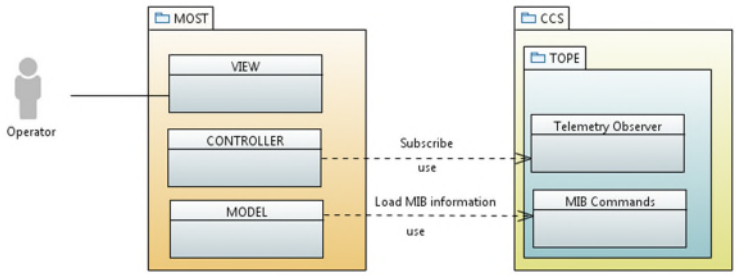

Fig. 2. Interaction between the Operator, MOST and CCS

The Controller implementation is based on the ***subscribe*** TOPE function. MOST can subscribe to the parameters, and receive the results asynchronously directly from the CCS.
MOST uses an efficient way to wait for many update: a callback function defined in the Controller is called by the CCS whenever a variable is updated. Then the Controller will update the View with the new telemetries values. One benefit of this approach is that no updates can be missed.
MOST implements in TOPE/TCL the Observer Design Pattern [5]. The CCS (the Subject) is the "Keeper" of the data coming from the Specimen in terms of TM packets and parameters. The Controller plays the role of Observers. The Controller registers itself with the CCS. Whenever data (telemetry parameters/packets) changes, the CCS broadcasts the Controller the changes. In this way MOST does not need to know how the data change, but only to be notified by the CCS about the change.
A key principle is that the observer does not know anything about the observers. It "publishes" a change and the observers get notified of the change.

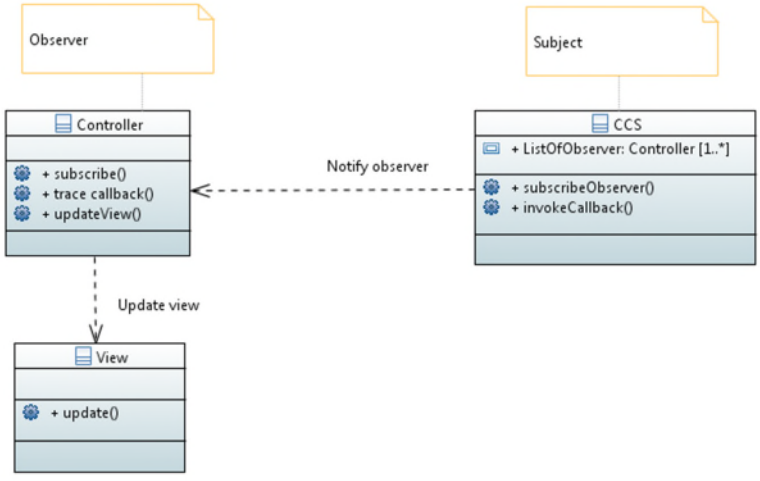

Fig. 3. MOST observer pattern implementation

Software Components Design

The software components are logically grouped in three different layers. A layer generally offers functionality to the layer above and utilizes functionality from the layer below.
Each layer contains different modules/interfaces and encapsulates a dedicated functionality.
The interfaces providing this functionalities are mainly divided into Panes, Kernels, Factories, Status, and Descriptors where:

- **Panes** handle the graphical part in the view.
- **Kernels** handle the specific business logic: they are the workers.
- **Factories** handle the creation of specific objects.
- **Status** store the actual status of a specific object.
- **Descriptors** handle the descriptions of the object in terms of attributes

Each software layer provides and requires a clear defined interface to and from other components and thus can be developed, tested and extended independently from each other. Moreover, this architecture makes MOST usable in different missions.
Each layer forms a separate item in the version control system.

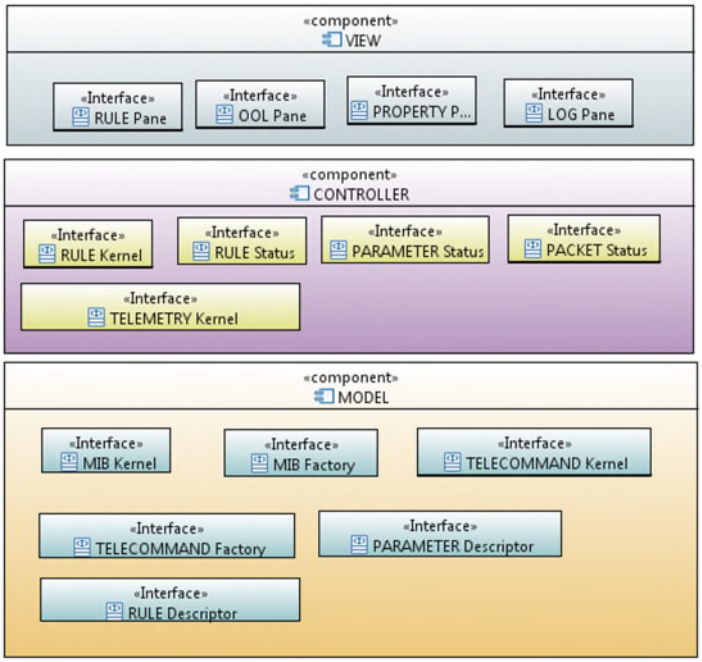

Fig. 4. Software components decomposition in interfaces

The separation of the model (MIB) from the view and controller components allows multiple views of the same model. If the user changes the model via the controller of one view, all other views dependent on this data should reflect the change. To achieve this, the controller catches the changes from the CCS and notifies all views whenever its data changes. The views in turn retrieve new data from the model and update their displayed information. This solution allows to change a subsystem of the application without causing major effects to other subsystems. For example, you can change from a non-graphical to a graphical user interface without modifying the model subsystem. You can also add support for a new input device without affecting information display or the functional core. All versions of the software can operate on the same model subsystem independently of specific ‘look and feel’.

Model Layer

The MIB contains all the static mission data: it defines the telemetries and telecommands characteristics of the mission. This included the specification of length and type of parameters, out of limits, verification steps and validity. The Model Layer abstracts the MIB information in a set of interacting objects like: telecommand, telemetry, out of limit and rule. When the system is analysed, developed and implemented in term of natural objects it becomes easy to understand the design and the implementation. Moreover, if the MIB changes only the Model Layer has to be update.

The Model Common Core is a set of objects that describe entities like telemetry parameters and telemetry packets that can be used across mission in the same way. It implements the singleton pattern [8] in combination with the abstract factory pattern [6]

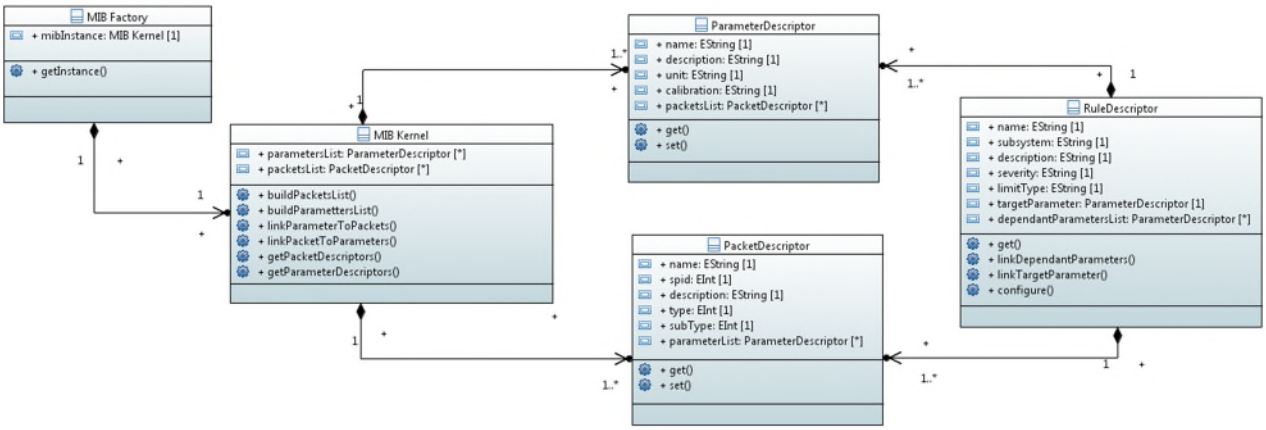

Fig. 5. Model layer common core

The Model Specific Core contains objects to manage telecommands whose behaviour have to be customized to specific mission, like enabling housekeeping report.

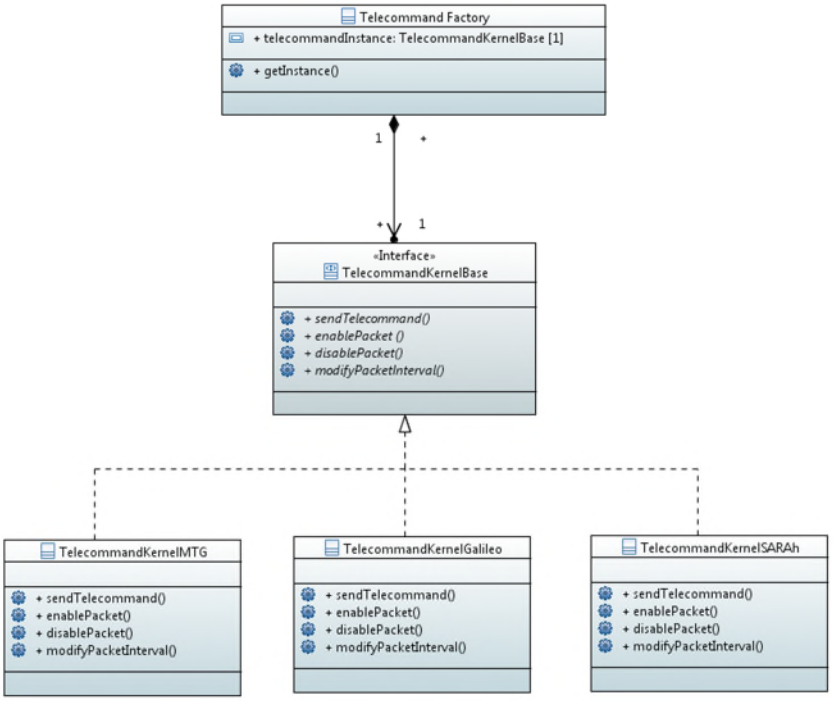

Fig. 6. Model Layer Specific Core

Controller Layer

The Controller layer reduces the coupling between the Model and the View. It is in charge to connect to the CCS and receive notifications from telemetries update (push-based notification).

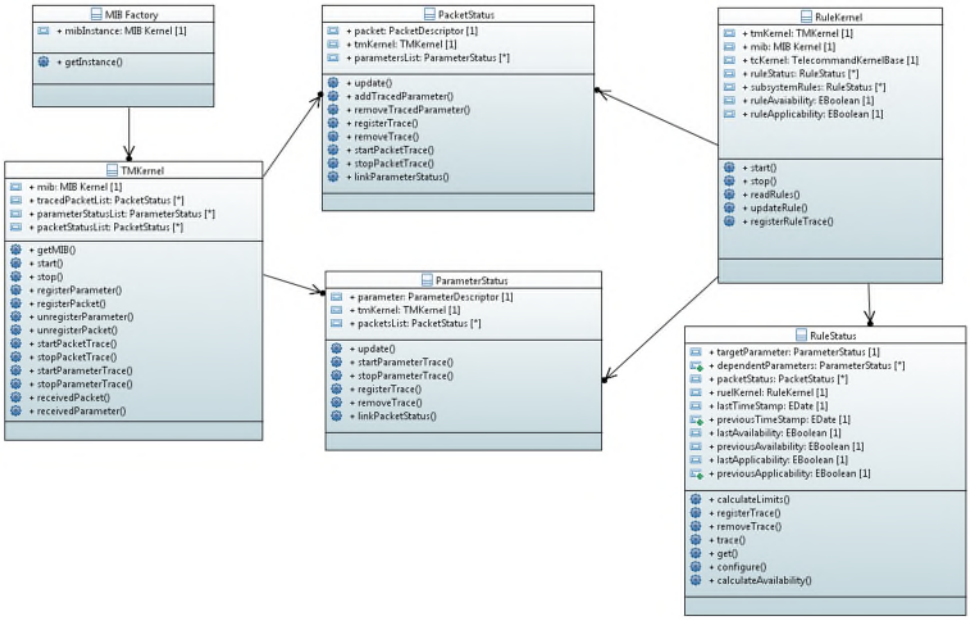

Fig. 7. Controller Layer

View Layer

The View presents to the user the information about telemetry parameters, telemetry packets and rule.

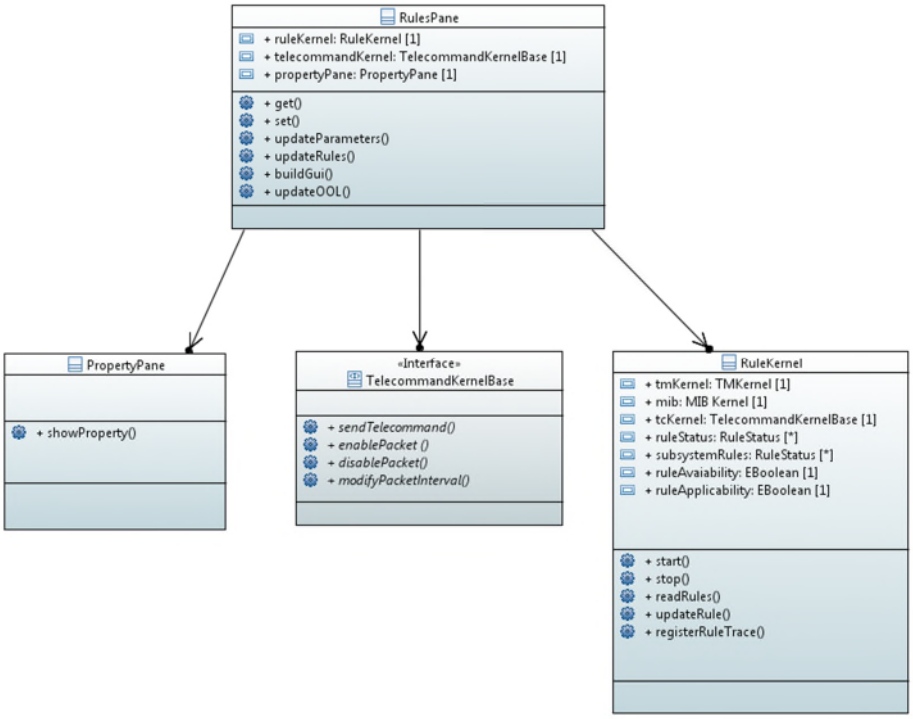

Fig. 8. View Layer

LEAVING A LEGACY

MOST has been deployed in several mission (Galileo FOC, MTG, EnMAP, SARah, EDRS-C) based on the CCS developed by Terma. MOST is not only contributing to make the EGSE AIT activities more efficient and effective, but it has spread a new mind-set. The Test Requirement Specification (TRS), the Test Procedure (TP) and the AIT Software tools have been seen as different aspects of the same process. An attempt to codify and test a specimen's requirement. Each flows direct into the next, passing through the natural boundaries between Subsystem engineering tasks and AIT tasks. The established process has encouraged feedback from testing the specimen using MOST into the Test Procedure Specification. By doing that, a natural flow of knowledge has been moved from different actors (System Engineers, AIT Engineers and Software Engineers) reducing the risk to duplicate information and increasing traceability from requirements specification and AIT test reports. The way it has been possible, was to put the abstraction of the test in MOST and the details in metadata (e.g. MIB). MOST knows what to do (the test to perform) and is actually the kernel of the test. The metadata contain the details about the specific specimen, and are the only to change. This solution has improved not only the reusability of tools, but also the communication between teams.

Fig. 9. MOST's deployment

CONCLUSION

The MOST framework described in this paper illustrates how the development of AIT/EGSE software tools can be simplified and unified. The key of the success of MOST is its ability to capture common software design patterns and to consolidate those recurrent solutions into flexible framework components. This concept efficiently encapsulate and enhance low-level specimen's commanding and controlling.
MOST's design does not espouse any revolutionary design ideas, but is based on sound common sense and it is now a toolbox with a rich set of libraries and tools (ANTs). Tools amplify your talent [9]. The better your tools, and the better you know how to use them, the more productive you can be.

REFERENCES

[1] https://en.wikipedia.org/wiki/Software_framework
[2] ECSS-E-HB-40A (Software engineering handbook) Issue A.
[3] F. P. Brooks, *The Mythical Man-Month*, Addison-Wesley, 1975.
[4] https://en.wikipedia.org/wiki/Model-view-controller
[5] https://en.wikipedia.org/wiki/Observer_pattern
[6] https://en.wikipedia.org/wiki/Factory_method_pattern
[7] https://en.wikipedia.org/wiki/Abstract_factory_pattern
[8] https://en.wikipedia.org/wiki/Singleton_pattern
[9] A. Hunt, D. Thomas, *The Pragmatic Programmer: From Journeyman to Master,* Addison-Wesley, 2000