Department of Veterans Affairs

*Open Source Electronic Health Record Services*

MTools IDE

System Design Document



Version 0.1

June 2013

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Revision | Description | Author |
| 06/19/2013 | 0.1 | Initial Draft | Jimmy Spivey |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1. Introduction 4](#_Toc359498275)

[2. Enhancing the original MTools plugins 4](#_Toc359498276)

[2.1. MEditor 4](#_Toc359498277)

[2.2. MDebug 4](#_Toc359498278)

[3. Implementation specs for MTools features 4](#_Toc359498279)

[3.1. MEditor 4](#_Toc359498280)

[3.2. MDebug 5](#_Toc359498281)

[3.2.1. Concurrent processing of the Eclipse UI concerns and the XTDEBUG backend concerns 5](#_Toc359498282)

[3.2.2. Case example of how to work with the separate UI and Core plug-ins 5](#_Toc359498283)

[4. MTools Class Diagrams 6](#_Toc359498284)

[4.1. MEditor 6](#_Toc359498285)

[4.2. Mdebug 8](#_Toc359498286)

[5. Appendix 9](#_Toc359498287)

[5.1. Acronyms and Definitions 9](#_Toc359498288)

[5.2. Software Licenses 11](#_Toc359498289)

[5.2.1. Software under License 11](#_Toc359498290)

[5.2.2. License Locations 11](#_Toc359498291)

# Introduction

The Department of Veterans Affairs (VA) has contributed the latest U.S. Department of State Freedom of Information Act (FOIA) release of the Veterans Health Information Systems and Technology Architecture (VistA) codebase to Open Source Electronic Health Record Agent (OSEHRA), the custodial agent that serves as the central governing body of a new open source community. The Open Source Electronic Health Record (EHR) Services project includes VistA Data Comparison, VistA System Test Platform, VistA Refactoring, VistA System Test Scripts, Veterans Benefits Administration (VBA) System Test Platform, Eclipse Plug-In Tool, and VistA Meaningful Use Certification.

# Enhancing the original MTools plugins

## MEditor

MEditor’s routine saving and loading to and from the server was re-implemented from scratch. This also involves the logic for backup, comparing routines, hierarchal directory support and the dialogs that occur when saving or loading. These were refitted completely because the prior code was too difficult to maintain. It had grown and become patched over many times, obscuring any cleanness to its design. The commit differences on the git repository can clearly show the new code replacing the old code in detail.

## MDebug

The original debug plugin was migrated to two new plugins. Only a very small portion of the original code remains in the new plugins. Many new features were added, and all the existing features were brought over. The main difference is that the prior plugin was written as Eclipse Actions, which would then update and change Eclipse Views and their SWT components. The GUI was not as finely presented as other debuggers, and being able to present highly readable details for a debugger is very important. So the migration was to the Eclipse Debug Model. This is how all other languages which use Eclipse implement debuggers. It provides a widely adopted and familiar debug GUI and behavior, as well as fast response and lots of information. Additionally there was no other viable way to get the newer debug features that were desired, specifically adding breakpoints to a line in MEditor.

# Implementation specs for MTools features

## MEditor

Eclipse, as a GUI based IDE, works from a single main or UI thread. It also manages a thread pool to handle background processing. Only the main or UI thread can create or update SWT components. Generally, all other processing should occur in background jobs, otherwise the main thread will have to wait for the non SWT processing to finish before the user can do anything to the Eclipse workbench window. Although background jobs cannot directly update SWT components, they can schedule a job for the UI thread to do any SWT widget creation or updates.

Given Eclipse’s multithreading workflow, changes were made to MEditor to put much of the actions and other processes that it creates into background jobs. This enhances the UI response when editing text or clicking any of the MEditor actions. As for MDebug, it does not rely on these contrived Eclipse Actions, but instead uses the Eclipse Debug Platform puts all of its actions into background jobs already. Therefore clicking an action, such as “step over” or “resume” doesn’t cause Eclipse to hang.

## MDebug

### Concurrent processing of the Eclipse UI concerns and the XTDEBUG backend concerns

MDebug was created by following an article on how to create a custom Eclipse Debugger for any language ([eclipse.org/articles/Article-Debugger/how-to.html](http://eclipse.org/articles/Article-Debugger/how-to.html)). This article was written in 2004 and is partially out of date with regards to asynchronous processing. It mentions that the step feature in Eclipse is invoked from the UI thread, which is incorrect with the version of Eclipse we are using. In our version, stepping is invoked as a background job and will not cause Eclipse’s workbench window to hang, even if the back-end debug system is synchronous.

In our case, the XTDEBUG RPC calls are synchronous; they send a single request and wait for its response to come back. However this is all processed in an asynchronous job on Eclipse, so the UI will not have any hanging problems. However it is also worth noting that although multiple jobs can run concurrently, access to the RPC **must not be** concurrent because the MUMPS background process can only handle a single request at a time. The java keyword “synchronize” is used on the XtdebugHandler class to make sure that multiple Eclipse jobs do not call this RPC while it is processing a current job. So in conclusion, the Eclipse UI thread does not hang, because stepping is processed as an asynchronous Eclipse background job.

### Case example of how to work with the separate UI and Core plug-ins

A custom console for handling READ and WRITE commands was added to MDebug. This console relies on the Eclipses Console View and Console Manager components, and not the Debug Platform. As a result of that, the plumbing to get this rendering and working on the screen is not provided by default. Therefore implementing it required less lightweight delegate based classes.

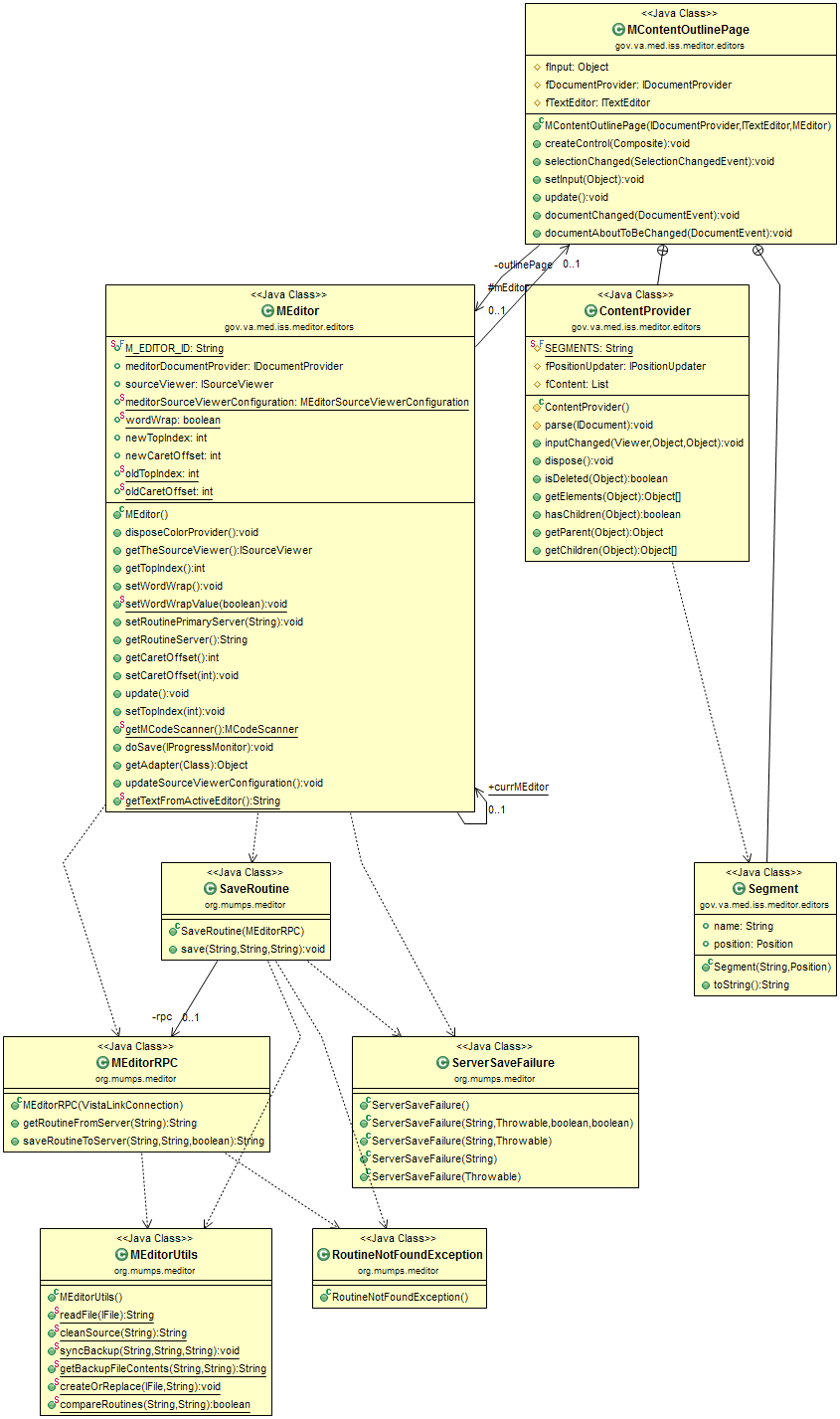
This custom console has implemented in the UI plug-in mostly, with some UI agnostic listener classes existing in the core plugin. It works by registering a Launch Configuration listener to the MDebug UI plug-in. This listener will react if any new launch configurations are launched by the user. Custom listeners for our custom console are added to both the Debug Target and the console. This is because the debug target needs to listen for when the console has finished collecting input, and what that input is. And the console needs to know when it is ready to receive input and if any output is to be written. These listeners are registered in a generalized way, as opposed to having the classes directly invoke each other by the specific Java class name, which it cannot see anyway. This is because it is not possible for the core plug-in to see the UI console class directly, as its specific java type. So the console instead implements a core listener and is registered to the debug target, a class from the core plug-in. This type of event based processing is common for design paradigms which completely separate the UI and core or model concerns into separate projects.

# MTools Class Diagrams

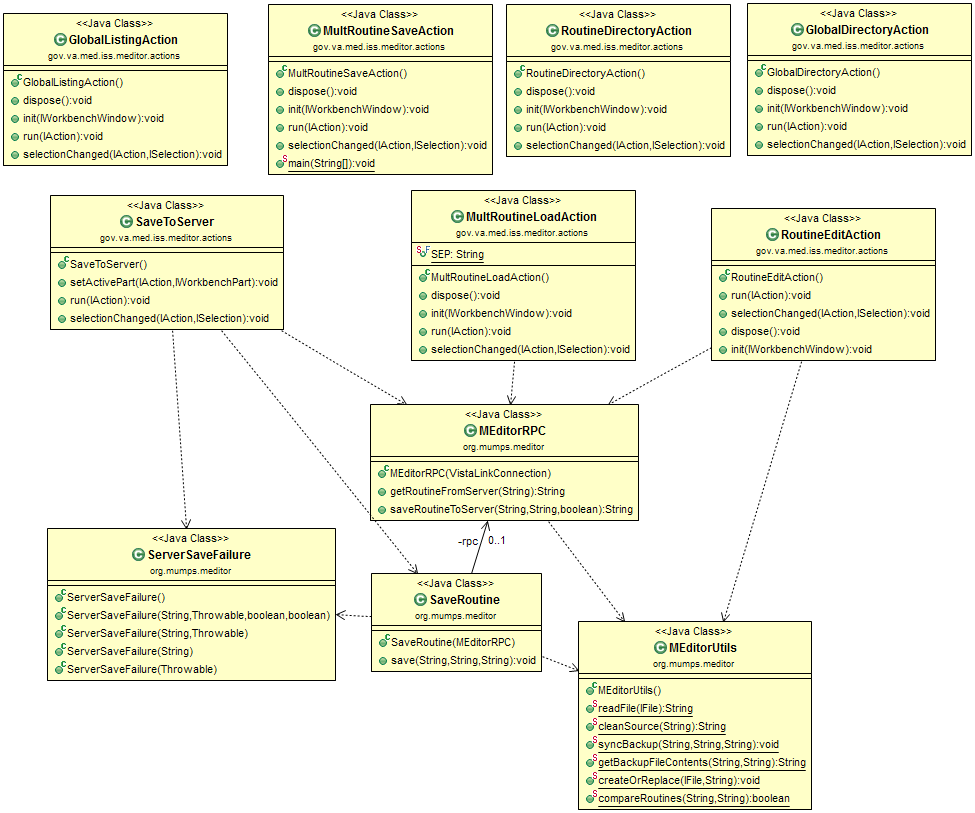
## MEditor

The MEditor plugin has an almost purely procedural implementation, in lieu of an object oriented approach. This means that instead of many objects relating to each other and each handling separate responsibilities, the code is organized into procedures, java methods, only. These procedures are spread into a logical class which has many procedures related to it, but rarely any instance (object) variables. Some use instance variables but the object is often a singleton.

The MEditor class diagrams show dependencies, whereas the Mdebug do not. This is because the MEditor is procedural, and we want to see if the classes depend (call) each other. Otherwise since there aren’t any instantiation relationships in the MEditor classes, there would be no relations to show in the diagram. The diagram bellow is for loading and saving routines from and to the server.



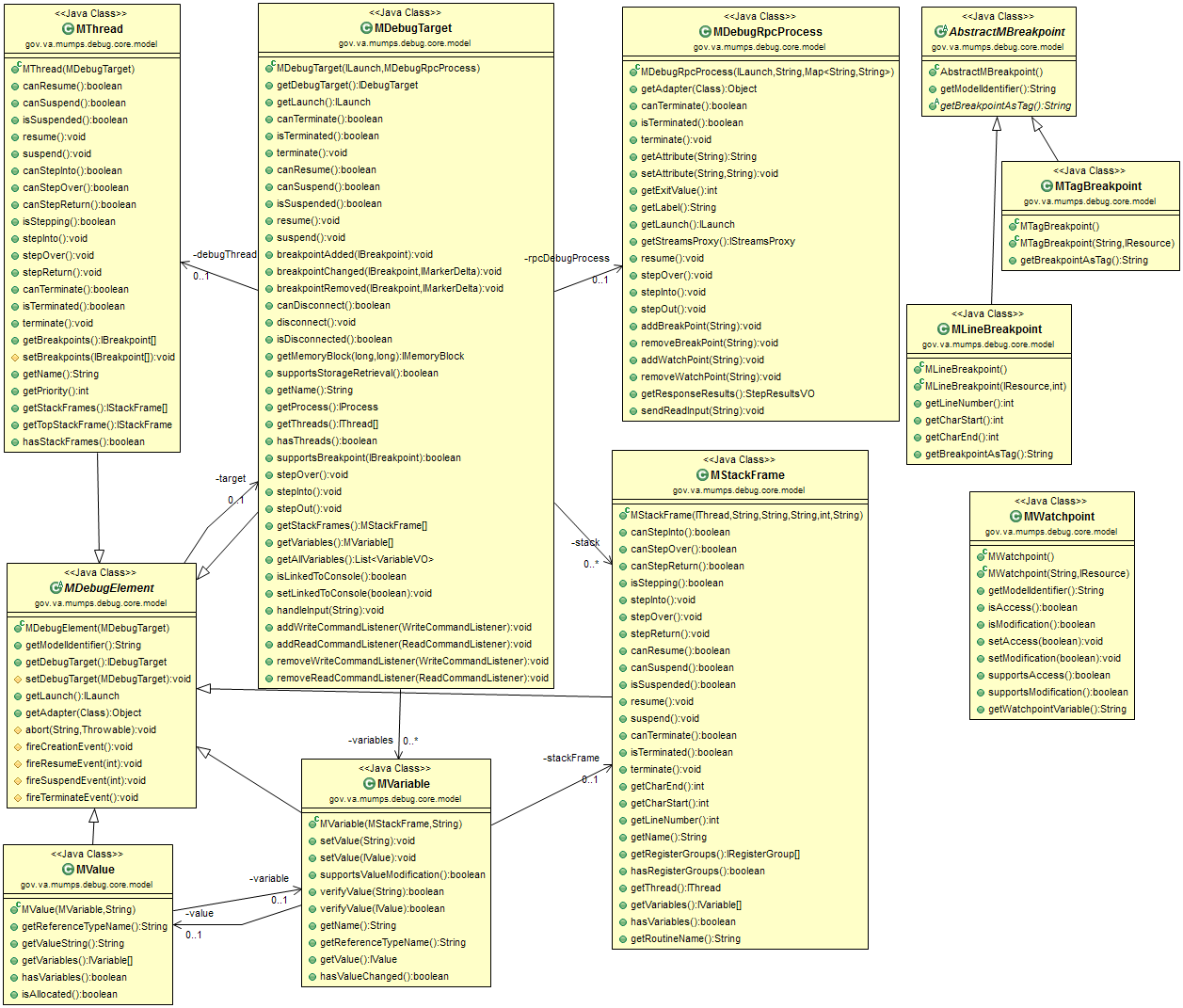
The classes in the bellow class diagram are for all of the MEditor actions. Actions are the icons that the user can select from the VistA menu, such as “Global Directory”. This also includes Routine Load, the class RoutineEditAction. Many of the actions are self-contained and do not rely on procedures from other classes.



## Mdebug

Mdebug is implemented as 2 plug-ins, a core and a ui (user interface) plug-in. These plug-ins are quite different in design than the other plugins, because they work so closely with the Eclipse Platform and its underlying classes. These classes are Object Oriented in design, and as a result of that they also rely on polymorphism. The Eclipse framework will handle most of the heavy lifting, and delegate the implementation at various points to Mdebug. Because of this separation of duties, Mdebug’s can be implementation can be described as several small delegate implementation scattered around, being called by Eclipse at several points. Whereas MEditor is just a few single entry points, which handle all the work with little use of Eclipse’s platform, and is procedural instead of object oriented.

Despite Mdebug having mostly a delegate based, lightweight implementation, there is at least one area where it must do heavy lifting. It cannot possibly understand how to debug any given language, so all of those implementation details are defined in various xtdebug packages and utilized in the MdebugTarget class. This class is in the core plugin, and there are many other classes which relate to it, as seen in figure TODO. MdebugTarget, and those classes which relate to it, are where the debugger is implemented at a core or model level. The figure bellow shows Mdebug’s Model objects.



# Appendix

## Acronyms and Definitions

|  |  |
| --- | --- |
| Branch | Exists in a repository. Contains a set of revisions in a chronological order. There may be multiple branches in a repository, tracking the same files in parallel. These branches may later be merged into the main branch. |
| Background job | In eclipse, a background job is processed by a thread pool allows the UI window to be responsive. Refer to: http://www.vogella.com/articles/EclipseJobs/article.html |
| Eclipse | An IDE primarily used for Java software development. |
| Eclipse View | A tab inside of Eclipse which provides application features to aid in software development. Eclipse provides many by default (e.g. search, directory explorer and console). RASR and JCTerm Plug-in provide their functionality inside of their own Eclipse Views. |
| Eclipse Plug-in | An extension to the Eclipse application, which can be installed. It gives Eclipse new features for software development. |
| EPL | Eclipse Public License |
| Fork | A copy of source code from one software project which creates a new separate project. Unlike a branch, there is no absolutely no intention of merging this back into its parent. Additionally, unlike a branch, it is a new project with new goals. |
| Git | A widely used SCM software. It is used to store and track the code for Mtools well. It is a distributed version control system. |
| Git Repository | In Git, a repository is where all the code history is stored. The code itself, at any time, is created from this meta-information. |
| GUI | Graphical user interface, as opposed to a text only based interface |
| IDE | Integrated Developer Environment. A robust, text editing application which allows software developers to write and test code. |
| Model object | In this paper, data-centric classes which encapsulate closely related items. Under Eclipse, these objects are separated from interface related concerns, and may exist solely in core or model plug-ins that are separate from UI plug-ins. The UI plug-ins depend on the model or core plug-ins. |
| Open Source | Software which is licensed under an open source license. This typically allows unrestricted modification and distribution of such licensed software. |
| RPC | Remote Procedure Call. Often using network connection, allows remote invocation and result gathering of a process. In this paper, it refers to XTDEBUG and how it is invoked from the Eclipse plug-ins. |
| SCM | Source Code Management |
| Software revision | A set of changes made to a software’s source code. One or more (typically the latter) revisions make up a software version. |
| SWT | Standard Widget Toolkit. A GUI framework that Eclipse uses to render and handle GUI’s. |
| Terminal Emulator | An application that renders text-based user interfaces and accepts input from a command line. No graphics, only text is supported. |
| Thread | In this paper, the term thread specifically refers to threads used in the Java language. It allows for con-current processing in Java, such as handling a user interface and also parsing syntax in the background. |
| Thread pool | A thread pool is a collection of threads that operate on tasks in their queue. The major advantage is that it is re-uses threads instead of creating them and throwing them away. |
| Version Control System | An application which manages all revisions and branches of revisions for a software project. |
| VistA | Veterans Health Information Systems and Technology Architecture |
| XTDEBUG | A MUMPS routine that communicates with the eclipse plug-in MDebug. |

## Software Licenses

### Software under License

|  |  |
| --- | --- |
| MTools | Apache License, Version 2.0 |
| VistA-FOIA | Apache License, Version 2.0 |

### License Locations

|  |  |
| --- | --- |
| Eclipse Public License v 1.0 | http://www.eclipse.org/legal/epl-v10.html |
| Apache License, Version 2.0 | http://www.apache.org/licenses/LICENSE-2.0.html |