COORDINATED HIGHWAYS ACTION RESPONSE TEAM
STATE HIGHWAY ADMINISTRATION

R1B1 GUI Detailed Design

Contract DBM-9713-NMS
TSR # 9901961
Document # M361-DS-004R0

January 21, 2000
By
Computer Sciences Corporation and PB Farradyne Inc

Table of Contents

1 Introduction 1-1
1.1 Purpose 1-1
1.2 Objectives 1-1
1.3 Scope 1-1
1.4 Acronyms 1-1
1.5 References 1-2
1.6 Design Process 1-2
1.7 Design Tools 1-3
1.8 Work Products 1-3
2 Software Architecture 2-1
3 Use Cases 3-1
3.1 Release1UseCaseDiagram (Use Case Diagram) 3-1
4 Classes 4-1
4.1 R1B1GUIClassDiagram (Class Diagram) 4-1
4.2 DataModelClasses (Class Diagram) 4-5
4.3 NavigatorClasses (Class Diagram) 4-10
4.4 GUIDMSClasses (Class Diagram) 4-13
4.5 GUIDictionaryModuleClasses (Class Diagram) 4-18
4.6 GUIPlanClasses (Class Diagram) 4-21
4.7 GUIUserManagementClasses (Class Diagram) 4-24
4.8 UtilityClasses (Class Diagram) 4-26
4.9 SystemInterfaces (Class Diagram) 4-31
4.10 JavaClasses (Class Diagram) 4-36
4.11 CORBAClasses (Class Diagram) 4-38
5 Sequence Diagrams 5-1
5.1 GUI:ChangeUserBasic (Sequence Diagram) 5-1
5.2 GUI:CommandObjectBasic (Sequence Diagram) 5-2
5.3 GUI:ConfigurePreferencesBasic (Sequence Diagram) 5-3
5.4 GUI:DiscoveryBasic (Sequence Diagram) 5-4
5.5 GUI:LoginBasic (Sequence Diagram) 5-5
5.6 GUI:LogoutBasic (Sequence Diagram) 5-6
5.7 GUI:MakeMenuMultipleSelect (Sequence Diagram) 5-7
5.8 GUI:MakeMenuNoneSelected (Sequence Diagram) 5-8
5.9 GUI:MakeMenuSingleSelect (Sequence Diagram) 5-9
5.10 GUI:ShutdownBasic (Sequence Diagram) 5-10
5.11 GUI:StartupBasic (Sequence Diagram) 5-11
5.12 GUI:EventUpdatePushedBasic (Sequence Diagram) 5-13
5.13 GUI:SystemCommandBasic (Sequence Diagram) 5-14
5.14 DataModel:AttachObserver (Sequence Diagram) 5-14
5.15 DataModel:ObjectAdded_ (Sequence Diagram) 5-16

5.16 DataModel:ObjectRemoved (Sequence Diagram) 5-17
5.17 DataModel:ObjectUpdated (Sequence Diagram) 5-18
5.18 DataModel:UpdateObservers (Sequence Diagram)...5-19
5.19 Navigator:AddNavigables (Sequence Diagram)...5-20
5.20 Navigator:Initialize (Sequence Diagram) 5-21
5.21 Navigator:RemoveNavigables (Sequence Diagram) 5-22
5.22 Navigator:TreeSelectionChange (Sequence Diagram) 5-23
5.23 GUIDMSModule:AddDMS (Sequence Diagram)...5-24
5.24 GUIDMSModule:AddMessageToLibrary (Sequence Diagram) 5-25
5.25 GUIDMSModule:CreateMessageLibrary (Sequence Diagram)..................................5-26
5.26 GUIDMSModule:DeleteMessageLibrary (Sequence Diagram)..................................5-27
5.27 GUIDMSModule:DeleteStoredMessage (Sequence Diagram) 5-28
5.28 GUIDMSModule:Login (Sequence Diagram)..5-28
5.29 GUIDMSModule:Logout (Sequence Diagram)...5-29
5.30 GUIDMSModule:EditLibraryMessage (Sequence Diagram).....................................5-31
5.31 GUIDMSModule:SetMessageLibraryProperties (Sequence Diagram).......................5-32
5.32 GUIDMSModule:BlankDMS (Sequence Diagram)...5-33
5.33 GUIDMSModule:CreateNewPlanItem (Sequence Diagram).....................................5-34
5.34 GUIDMSModule:DeleteDMS (Sequence Diagram) 5-35
5.35 GUIDMSModule:Shutdown (Sequence Diagram)...5-36
5.36 GUIDMSModule:Startup (Sequence Diagram)..5-37
5.37 GUIDMSModule:Discovery (Sequence Diagram)...5-38
5.38 GUIDMSModule:ForcePoll (Sequence Diagram)..5-39
5.39 GUIDMSModule:ModifyDMSSettings (Sequence Diagram)....................................5-40
5.40 GUIDMSModule:PutOnline (Sequence Diagram)...5-41
5.41 GUIDMSModule:Reset (Sequence Diagram) 5-42
5.42 GUIDMSModule:SetMessage (Sequence Diagram) 5-43
5.43 GUIDMSModule:ShowTrueDisplay (Sequence Diagram) 5-44
5.44 GUIDMSModule:TakeOffline (Sequence Diagram)..5-45
5.45 GUIDictionaryModule:DictionaryProperties (Sequence Diagram) 5-46
5.46 GUIDictionaryModule:Discovery (Sequence Diagram) 5-47
5.47 GUIDictionaryModule:EventHandling (Sequence Diagram).....................................5-48
5.48 GUIDictionaryModule:Shutdown (Sequence Diagram)..5-49
5.49 GUIDictionaryModule:Startup (Sequence Diagram) 5-50
5.50 GUIPlanModule:ActivatePlan (Sequence Diagram) 5-51
5.51 GUIPlanModule:AddPlan (Sequence Diagram)...5-52
5.52 GUIPlanModule:CreatePlanItem (Sequence Diagram) 5-53
5.53 GUIPlanModule:Discovery (Sequence Diagram) 5-54
5.54 GUIPlanModule:PlanItemAddedEvent (Sequence Diagram) 5-55
5.55 GUIPlanModule:PlanItemRemovedEvent (Sequence Diagram)................................5-56
5.56 GUIPlanModule:PlanRemovedEvent (Sequence Diagram) 5-57
5.57 GUIPlanModule:RemovePlan (Sequence Diagram) 5-58
5.58 GUIPlanModule:PlanAddedEvent (Sequence Diagram)...5-59
5.59 GUIPlanModule:RemovePlanItem (Sequence Diagram) 5-60
5.60 GUIPlanModule:Shutdown (Sequence Diagram)...5-61

5.61 GUIPlanModule:Startup (Sequence Diagram) 5-62
5.62 GUIUserManagementModule:AddUser (Sequence Diagram) 5-63
5.63 GUIUserManagementModule:ConfigureRoles (Sequence Diagram) 5-64
5.64 GUIUserManagementModule:ConfigureUsers (Sequence Diagram) 5-65
5.65 GUIUserManagementModule:CreateRole (Sequence Diagram) 5-66
5.66 GUIUserManagementModule:DeleteRole (Sequence Diagram) 5-67
5.67 GUIUserManagementModule:DeleteUser (Sequence Diagram) 5-68
5.68 GUIUserManagementModule:ForceLogout (Sequence Diagram) 5-69
5.69 GUIUserManagementModule:GrantRole (Sequence Diagram) 5-70
5.70 GUIUserManagementModule:Login (Sequence Diagram) 5-71
5.71 GUIUserManagementModule:Discovery (Sequence Diagram) 5-72
5.72 GUIUserManagementModule:ModifyRole (Sequence Diagram) 5-73
5.73 GUIUserManagementModule:RevokeRole (Sequence Diagram) 5-74
5.74 GUIUserManagementModule:Startup (Sequence Diagram) 5-75

6 GUI Screen Captures 6-1

6.1 GUI:ScreenAccess (State Chart) 6-1
6.2 Change Own Password Dialog 6-2
6.3 Command Status View 6-2
6.4 Create Role Dialog 6-3
6.5 Create User Dialog 6-3
6.6 Dictionary Properties Dialog 6-4
6.7 DMS Message Editor Dialog 6-5
6.8 DMS Message Library Properties Dialog 6-6
6.9 DMS Properties Dialog 6-6
6.10 DMS Stored Message Item Properties Dialog 6-7
6.11 GUI Toolbar 6-7
6.12 Login User Dialog 6-8
6.13 Plan Properties Dialog 6-8
6.14 Role Configuration Dialog 6-9
6.15 Transfer Controlled Resources Dialog 6-10
6.16 User Configuration Dialog 6-10
6.17 Manage Logins Dialog 6-11

Table of Figures

Figure 3-1. Release1UseCaseDiagram (Use Case Diagram) 3-1
Figure 4-1. R1B1GUIClassDiagram (Class Diagram) 4-1
Figure 4-2. DataModelClasses (Class Diagram) 4-6
Figure 4-3. NavigatorClasses (Class Diagram) 4-10
Figure 4-4. GUIDMSClasses (Class Diagram) 4-13
Figure 4-5. GUIDictionaryModuleClasses (Class Diagram) 4-18
Figure 4-6. GUIPlanClasses (Class Diagram) 4-21
Figure 4-7. GUIUserManagementClasses (Class Diagram) 4-24
Figure 4-8. UtilityClasses (Class Diagram) 4-26
Figure 4-9. SystemInterfaces (Class Diagram) 4-32
Figure 4-10. JavaClasses (Class Diagram) 4-36
Figure 4-11. CORBAClasses (Class Diagram) 4-38
Figure 5-1. GUI:ChangeUserBasic (Sequence Diagram) 5-1
Figure 5-2. GUI:CommandObjectBasic (Sequence Diagram) 5-2
Figure 5-3. GUI:ConfigurePreferencesBasic (Sequence Diagram) 5-3
Figure 5-4. GUI:DiscoveryBasic (Sequence Diagram) 5-4
Figure 5-5. GUI:LoginBasic (Sequence Diagram) 5-5
Figure 5-6. GUI:LogoutBasic (Sequence Diagram) 5-6
Figure 5-7. GUI:MakeMenuMultipleSelect (Sequence Diagram) 5-7
Figure 5-8. GUI:MakeMenuNoneSelected (Sequence Diagram) 5-8
Figure 5-9. GUI:MakeMenuSingleSelect (Sequence Diagram) 5-9
Figure 5-10. GUI:ShutdownBasic (Sequence Diagram) 5-10
Figure 5-11. GUI:StartupBasic (Sequence Diagram) 5-12
Figure 5-12. GUI:EventUpdatePushedBasic (Sequence Diagram) 5-13
Figure 5-13. GUI:SystemCommandBasic (Sequence Diagram) 5-14
Figure 5-14. DataModel:AttachObserver (Sequence Diagram) 5-15
Figure 5-15. DataModel:ObjectAdded_ (Sequence Diagram) 5-16
Figure 5-16. DataModel:ObjectRemoved (Sequence Diagram) 5-17
Figure 5-17. DataModel:ObjectUpdated (Sequence Diagram) 5-18
Figure 5-18. DataModel:UpdateObservers (Sequence Diagram) 5-19
Figure 5-19. Navigator:AddNavigables (Sequence Diagram) 5-20
Figure 5-20. Navigator:Initialize (Sequence Diagram) 5-21
Figure 5-21. Navigator:RemoveNavigables (Sequence Diagram) 5-22
Figure 5-22. Navigator:TreeSelectionChange (Sequence Diagram) 5-23
Figure 5-23. GUIDMSModule:AddDMS (Sequence Diagram) 5-24
Figure 5-24. GUIDMSModule:AddMessageToLibrary (Sequence Diagram) 5-25
Figure 5-25. GUIDMSModule:CreateMessageLibrary (Sequence Diagram) 5-26
Figure 5-26. GUIDMSModule:DeleteMessageLibrary (Sequence Diagram) 5-27
Figure 5-27. GUIDMSModule:DeleteStoredMessage (Sequence Diagram) 5-28
Figure 5-28. GUIDMSModule:Login (Sequence Diagram) 5-29
Figure 5-29. GUIDMSModule:Logout (Sequence Diagram) 5-30
Figure 5-30. GUIDMSModule:EditLibraryMessage (Sequence Diagram) 5-31
Figure 5-31. GUIDMSModule:SetMessageLibraryProperties (Sequence Diagram) 5-32

Figure 5-32. GUIDMSModule:BlankDMS (Sequence Diagram)....5-33
Figure 5-33. GUIDMSModule:CreateNewPlanItem (Sequence Diagram)....5-34
Figure 5-34. GUIDMSModule:DeleteDMS (Sequence Diagram)....5-35
Figure 5-35. GUIDMSModule:Shutdown (Sequence Diagram) 5-36
Figure 5-36. GUIDMSModule:Startup (Sequence Diagram)....5-37
Figure 5-37. GUIDMSModule:Discovery (Sequence Diagram)....5-38
Figure 5-38. GUIDMSModule:ForcePoll (Sequence Diagram)....5-39
Figure 5-39. GUIDMSModule:ModifyDMSSettings (Sequence Diagram)....5-40
Figure 5-40. GUIDMSModule:PutOnline (Sequence Diagram)....5-41
Figure 5-41. GUIDMSModule:Reset (Sequence Diagram)....5-42
Figure 5-42. GUIDMSModule:SetMessage (Sequence Diagram)....5-43
Figure 5-43. GUIDMSModule:ShowTrueDisplay (Sequence Diagram)....5-44
Figure 5-44. GUIDMSModule:TakeOffline (Sequence Diagram)....5-45
Figure 5-45. GUIDictionaryModule:DictionaryProperties (Sequence Diagram)....5-46
Figure 5-46. GUIDictionaryModule:Discovery (Sequence Diagram)....5-47
Figure 5-47. GUIDictionaryModule:EventHandling (Sequence Diagram)....5-48
Figure 5-48. GUIDictionaryModule:Shutdown (Sequence Diagram)....5-49
Figure 5-49. GUIDictionaryModule:Startup (Sequence Diagram)....5-50
Figure 5-50. GUIPlanModule:ActivatePlan (Sequence Diagram)....5-51
Figure 5-51. GUIPlanModule:AddPlan (Sequence Diagram)....5-52
Figure 5-52. GUIPlanModule:CreatePlanItem (Sequence Diagram)....5-53
Figure 5-53. GUIPlanModule:Discovery (Sequence Diagram)....5-54
Figure 5-54. GUIPlanModule:PlanItemAddedEvent (Sequence Diagram)....5-55
Figure 5-55. GUIPlanModule:PlanItemRemovedEvent (Sequence Diagram)....5-56
Figure 5-56. GUIPlanModule:PlanRemovedEvent (Sequence Diagram)....5-57
Figure 5-57. GUIPlanModule:RemovePlan (Sequence Diagram)....5-58
Figure 5-58. GUIPlanModule:PlanAddedEvent (Sequence Diagram)....5-59
Figure 5-59. GUIPlanModule:RemovePlanItem (Sequence Diagram)....5-60
Figure 5-60. GUIPlanModule:Shutdown (Sequence Diagram)....5-61
Figure 5-61. GUIPlanModule:Startup (Sequence Diagram)....5-62
Figure 5-62. GUIUserManagementModule:AddUser (Sequence Diagram)....5-63
Figure 5-63. GUIUserManagementModule:ConfigureRoles (Sequence Diagram)....5-64
Figure 5-64. GUIUserManagementModule:ConfigureUsers (Sequence Diagram)....5-65
Figure 5-65. GUIUserManagementModule:CreateRole (Sequence Diagram)....5-66
Figure 5-66. GUIUserManagementModule:DeleteRole (Sequence Diagram)....5-67
Figure 5-67. GUIUserManagementModule:DeleteUser (Sequence Diagram)....5-68
Figure 5-68. GUIUserManagementModule:ForceLogout (Sequence Diagram)....5-69
Figure 5-69. GUIUserManagementModule:GrantRole (Sequence Diagram)....5-70
Figure 5-70. GUIUserManagementModule:Login (Sequence Diagram)....5-71
Figure 5-71. GUIUserManagementModule:Discovery (Sequence Diagram)....5-72
Figure 5-72. GUIUserManagementModule:ModifyRole (Sequence Diagram)....5-73
Figure 5-73. GUIUserManagementModule:RevokeRole (Sequence Diagram)....5-74
Figure 5-74. GUIUserManagementModule:Startup (Sequence Diagram)....5-75
Figure 6-1. GUI:ScreenAccess (State Chart)....6-1

1 Introduction

1.1 Purpose

This document describes the detailed design of the Chart II Graphical User Interface (GUI) application for Release 1, Build 1. This design is driven by the Release 1, Build 1 requirements as stated in document CHARTII-RS-001-00, "***CHART II System Requirements Specification For Release 1 Build 1***."

1.2 Objectives

The main objective of this design is to provide software developers with a framework in which to provide implementation of the software components used to satisfy the requirements of Release 1, Build 1 of the Chart II system user interface. This document focuses on the client side of each of the system use cases.

1.3 Scope

This design is limited to Release 1, Build 1 of the Chart II system and the requirements as stated in the aforementioned requirements document.

1.4 Acronyms

The following acronyms appear throughout this document:

CORBA	Common Object Request Broker Architecture
DMS	Dynamic Message Sign
FMS	Field Management Station
GUI	CHART II Graphical User Interface application.
IDL	Interface Definition Language
OMG	Object Management Group
ORB	Object Request Broker
R1B1	Release 1, Build 1 of the CHART II System
UML	Unified Modeling Language

1.5 References

CHART II System Requirements Specification For Release 1 Build 1, document number CHARTII-RS-001-00, Computer Sciences Corporation and PB Farradyne, Inc.

CHART II High Level Design For Release 1 Build 1, document number M361-DS-001R0, Computer Sciences Corporation and PB Farradyne, Inc.

CHART II GUI High Level Design For Release 1 Build 1, document number M361-DS-003R0, Computer Sciences Corporation and PB Farradyne, Inc.

The Common Object Request Broker: Architecture and Specification, Revision 2.2, OMG Document 98-02-33.

Martin Fowler and Kendall Scott, *UML Distilled,* Addison-Wesley, 1997.

1.6 Design Process

Object oriented analysis and design techniques were used in creating this design. As such, much of the design is documented using diagrams that conform to the Unified Modeling Language (UML), a de facto standard for diagramming object oriented designs.

The design process is very iterative, with each step possibly causing changes to previous steps. Listed below is the process that was used to create the work products contained in this document:

- The team started by determining the system uses that would need designing. It was determined that the GUI detailed design should model the client side of each transaction in the system use cases. Thus, the use case diagram from document M-361-DS-001R0, "***CHART II High Level Design For Release 1 Build 1***" was used.
- The team then created GUI screens that would be displayed to the user to collect and convey necessary information. Captured images of these screens are contained in Appendix A of this document.
- A straw man class diagram was created with major entities evident in the use cases being listed as possible classes in the system. High level relationships between the classes were discovered and documented on the class diagram.
- Additional class diagrams were added to depict third-party software objects and interfaces which were needed in order to convey the intent of the design. Only classes and methods shown on the sequence diagrams are included in diagrams for third party products.
- A sequence diagram was created for each use case, showing how the classes on the class diagram would be used to perform the use case. This often involved changes to the class diagram, such as adding classes, moving responsibilities between classes, or adding operations to a class. Creation of the sequence diagrams frequently uncovered details that required other sequence diagrams, or even proposed GUI screens to be modified.

- After the process of creating sequence diagrams and associated changes to the class diagram, internal reviews were used to resolve remaining issues.

1.7 Design Tools

The work products contained within this design are extracted from the COOL:JEX design tool. Within this tool, the design is contained in the Chart II project, Release 1 configuration, SystemDesign phase, system version R1B1GUI.

1.8 Work Products

This design contains the following work products:

- UML Class diagrams showing the high level software objects that will allow the system to accommodate the uses of the system described in the Use Case diagram.
- UML Sequence diagrams showing how the classes interact to accomplish a particular system function.
- GUI Screen Captures showing the information that will be conveyed to the operator and the data elements the operator may modify.
- A UML state diagram modeling GUI screen navigation. This diagram shows which GUI screens can be accessed from other GUI screens.

2 Software Architecture

For a thorough discussion of how the CHART II GUI fits into the architecture of the CHART II system please refer to the Software Architecture section of document M-361-DS-003R0, "***CHART II GUI High Level Design For Release 1 Build 1***."

The CHART II GUI application has been separated into a core GUI library which provides basic functionality needed by all portions of the GUI, a data model which provides an implementation of the subject-observer design pattern, and many installable modules which provide application functionality. For the R1B1 version of the application, the GUI will be comprised of the following installable modules. The DMS control module provides all DMS specific functionality including DMS control and configuration, DMS message library creation and modification and the creation and modification of DMS stored message plan items. The Dictionary module provides an interface for modifying the contents of the system dictionary. The Plan module provides an interface for creating a new plan and delegates creation of plan items to other modules. The User Management module provides interfaces for configuring system users and roles.

3 Use Cases

3.1 Release1UseCaseDiagram (Use Case Diagram)

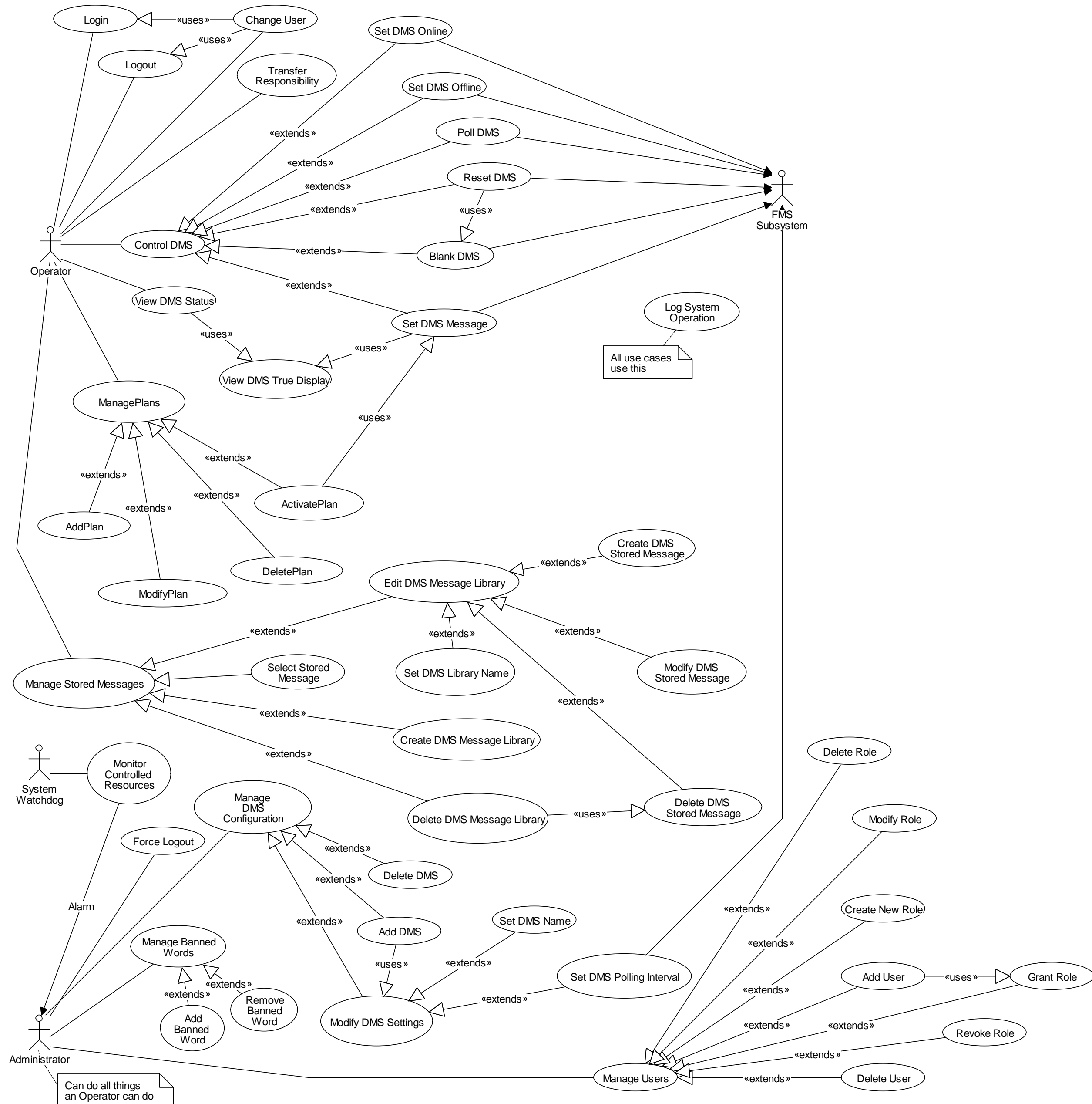


Figure 3-1. Release1UseCaseDiagram (Use Case Diagram)

4 Classes

4.1 R1B1GUIClassDiagram (Class Diagram)

This class diagram depicts the core classes and interfaces necessary to provide an extensible GUI application framework for future CHART II development. Included are details of objects served from the GUI application, an installable module framework, a core data model that provides the framework for window updates when objects change, and a framework for system preference configuration.

Figure 4-1. R1B1GUIClassDiagram (Class Diagram)

4.1.1 CommandFailureView (Class)

This window will store the descriptions of command failures that have occurred. The failures will be moved out of the command status view into the command failure view so that failures

will not clutter up the command status view. This view also provides the user with a convenient mechanism for reviewing system requests and commands which have failed, when they failed, and the reasons for their failure.

4.1.2 CommandStatus (Class)

The CommandStatus class is used to allow a calling process to be notified of the progress of an asynchronous operation. This is typically used by a GUI when field communications are involved to complete a method call, allowing the GUI to show the user the progress of the operation. The long running operation calls back to the CommandStatus object periodically as the command is executed and makes a final call to the CommandStatus when the operation has completed. The final call to the CommandStatus from the long running operation indicates the success or failure of the command.

4.1.3 CommandStatusImpl (Class)

This class is the implementation of the CommandStatus CORBA interface. It will be created and passed to a server when a command is to be executed so that the GUI can stay updated as the command is executing.

4.1.4 CommandStatusView (Class)

This class is a window that will monitor the status of all pending commands that are in the process of being executed. This class is an observer of the data model and, as such, will be updated when objects are added, removed, or updated in the system.

4.1.5 CosEvent. PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

4.1.6 java.awt.event. ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

4.1.7 InstallableModule (Class)

This class is the basic interface that all installable modules must implement. It contains functionality that all modules must support to be installable modules. This includes functionality for startup, shutdown, login, logout, and the handling of system and user preferences.

4.1.8 PreferenceHelper (Class)

This class is delegated all of the system and user preference functionality for the GUI. This includes registration, reading, writing, and accessing the preference values.

4.1.9 Menuable (Class)

This interface allows an object to provide menu item strings and receive commands when the corresponding menu items are clicked on. It supports both single selection and multiple selection of Menuable objects. The getSSMenuItems() method should return the menu items to display if the object is singly selected. The getMSMenuItems() method should return the menu items that the Menuable object wishes to display if other Menuable objects are selected. The access token is passed to these methods to allow the Menuable object to check the user's access rights before supplying the strings, so the user's actions may be restricted.

4.1.10 SystemPreferenceSupporter (Class)

This interface will be implemented by any InstallableModules that wish to support their own system preferences. They will be given a chance at startup to register their preferences. Then, after the preferences have been read in, the modules can get access to the values when their startup() method is called.

4.1.11 UserLoginSession (Class)

The UserLoginSession class is used to store information about a user that is logged into the system. This object is served from the GUI and provides a means for the servers to call back into the GUI process.

4.1.12 UserPreferenceSupporter (Class)

This interface will be implemented by any module that wishes to have user-specific preferences. The module will be called before InstallableModule.loggedIn() to allow the preferences to be registered. Then the preferences will be read, and will be available to the modules in the loggedIn() method. Any module supporting user preferences should also supply property page(s) so that the user can change the values of the preferences.

4.1.13 UserPreferenceSheet (Class)

This class is a property sheet that is used to present the user with the current user preferences. It will be built by the GUI, and each installable module that supports user preferences will have a chance to add one or more property pages to the property sheet.

4.1.14 GUI (Class)

This class is a singleton that contains all of the centralized functionality in the GUI. This includes startup, shutdown, login, and logout. It manages the installable modules and controls all functionality that requires the modules to be called. In addition, it stores all of the CORBA object wrappers in the DataModel, which allows access to the objects and supports an update mechanism to notify interested observers whenever the objects change.

4.1.15 GUIDMSModule (Class)

This module is an installable GUI module that handles all of the DMS-specific functionality.

4.1.16 GUIDictionary Module (Class)

This class is an installable GUI module that handles all of the dictionary-specific functionality in the GUI.

4.1.17 GUIPlan Module (Class)

This is an installable GUI module that handles the Plan functionality in the GUI. Other modules that support plan items must attach their PlanItemCreationSupporters to the GUIPlanModule at startup. The plan module will then call them as necessary when it is necessary to create a specific type of GUIPlanItem.

4.1.18 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

4.1.19 GUIUserManagement Module (Class)

This class implements the InstallableModule interface and performs functionality for managing user rights. It can be called to configure the roles and users, or to force a logout.

4.1.20 PreferenceRegisterable (Class)

This interface will be implemented by any class that wishes to support the registration of preferences (otherwise known as properties or options). A preference is a key that has an associated value. When registering a preference, a default value must be provided in case the preference does not exist or cannot be read. The implementing class should store the registered preferences until such a time as they can be read in. After the preferences are read in, the implementing class should provide access to the preferences.

4.1.21 DiscoveryThread (Class)

This thread is used by the GUI to check for new event channels and served CORBA objects. It will periodically call the GUI to find event channels and objects.

4.1.22 EventConsumerGroup (Class)

This class represents a collection of event consumers that will be monitored to verify that they do not lose their connection to the CORBA event service. The class will periodically ask each consumer to verify its connection to the event channel on which it is dependent to receive events.

4.1.23 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

4.1.24 ModelObserver (Class)

This interface must be implemented by any object that would like to attach to the DataModel as an observer and get updated as system objects are added, deleted or changed.

4.1.25 GUINavigatorDriver (Class)

This class handles all of the Navigator-specific functionality for the GUI.

4.1.26 UserLoginSessionImpl (Class)

This class is the implementation of the CORBA UserLoginSession interface. It will be served from the GUI and will be passed to the OperationsCenter on login. It will also store the access token returned from the OperationsCenter.

4.1.27 CommandStatusHandler (Class)

This class provides functionality that allows the modules to deal with CommandStatus objects for calling asynchronous methods without performing the housekeeping associated with serving these objects. It provides a method for creating a CommandStatus object which will create the object, attach it to the ORB, add it to the data model, and observe the data model waiting for the CommandStatus object to complete. When it completes, this object will disconnect it from the ORB and remove it from the data model.

4.1.28 GUIToolBar (Class)

This class will hold all of the top-level buttons and will be the launching point for invoking the functionality of the CHART2 system. It will be created at startup, and each module may add any toolbar buttons at that time. At Login, modules that have added toolbar buttons at startup should enable any toolbar buttons that should be enabled (depending on access rights). The buttons will be disabled by the GUI after they are added at startup and again at logout.

4.1.29 NavigatorSupporter (Class)

This interface must be implemented by any subsystem that supports invoking the Navigator. It must be able to supply the Navigable objects, and also can support user interaction with the selected Navigable objects through menus and drag/drop.

4.2 DataModelClasses (Class Diagram)

The data model classes represent a collection of objects that, when altered via the DataModel, will notify observers that they have been modified. The notification will be delivered in the form of a call to the observer's update() method and will include a collection of changes that have occurred in the system in the preceding interval. Each change is either an object added change,

an object removed change, or an object updated change. If the change is an object updated change it may include hints that help an observer determine if it needs to take any action based on the change.

Figure 4-2. DataModelClasses (Class Diagram)

4.2.1 ChangeCollection (Class)

This class represents a collection of object changes. All object changes in the collection must be for objects of the same type. This allows an observer to look at one object in the collection and determine if it is interested in changes to this type of object. If the observer is not, it may ignore the entire collection.

4.2.2 GUIModelObserver (Class)

Interface to be implemented by GUI components that would like to observe changes to the data model. Observers of this type will be notified of changes on the GUI event dispatch thread.

4.2.3 GUIUpdater (Class)

This class is used to send all changes to GUIModelObservers in the GUI event dispatch thread. It does this by storing the changes until the dispatch thread calls the run() method.

4.2.4 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

4.2.5 Identifier (Class)

Wrapper class for a CHART2 identifier byte sequence. This class will be used to add identifiable objects to hash tables and perform subsequent lookup operations.

4.2.6 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

4.2.7 java.util.Hashtable (Class)

This class implements a hashtable, which is a data structure that maps keys to values. Any non-null object can be used as a key or as a value. Objects used as keys implement the hashCode method that is inherited by all objects from the java.lang.Object class.

4.2.8 ModelChange (Class)

This class is used to convey changes to observers of the DataModel. It contains all ObjectChanges for a particular update priority level for a particular period of time.

4.2.9 ModelObserver (Class)

This interface must be implemented by any object that would like to attach to the DataModel as an observer and get updated as system objects are added, deleted or changed.

4.2.10 ObjectAdded (Class)

This class indicates that the object to which this class refers has been added to the model during this period.

4.2.11 ObjectChange (Class)

This class represents the changes to a particular object stored in the DataModel for a particular period. The change may be that this object was added to the model, removed from the model, or updated during this period.

4.2.12 ObjectRemoved (Class)

This class indicates that the object to which it refers has been removed from the model. Therefore, any observers receiving an ObjectRemoved must remove all stored references to the object.

4.2.13 ObjectUpdated (Class)

This class indicates that an object that was already in the model has been updated. The update may be specific to certain parts of the object, and the UpdateHint objects are used to specify which data members within the object were changed. If there are no hints in the ObjectUpdated, it signifies that the entire object has been changed so the observer must query the object for any data members that it is displaying.

4.2.14 UpdateHint (Class)

This interface must be implemented by all objects that are to be used as update hints.

4.2.15 UpdatePriorityLevel (Class)

This class represents a particular priority update level. When an observer attaches to the data model an update priority level is specified. The system currently supports five levels of priority ranging from real time updates for animated displays to delayed updates for windows which can tolerate not being notified for a significant period of time when a change occurs to the system data model. Each time an object is modified it is added to the ChangeCollection for all priority levels. The notification of observers simply happens at longer and longer intervals as the priority level decreases. Thus, an observer of the data model connected at real time may be updated three times in one second while a lower priority observer may only be updated once at the end of the second. However, both observers will be told about the exact same changes that occurred during the second.

4.3 NavigatorClasses (Class Diagram)

Figure 4-3. NavigatorClasses (Class Diagram)

4.3.1 GUI (Class)

This class is a singleton that contains all of the centralized functionality in the GUI. This includes startup, shutdown, login, and logout. It manages the installable modules and controls all functionality that requires the modules to be called. In addition, it stores all of the CORBA object wrappers in the DataModel, which allows access to the objects and supports an update mechanism to notify interested observers whenever the objects change.

4.3.2 GUINavigatorDriver (Class)

This class handles all of the Navigator-specific functionality for the GUI.

4.3.3 java.util. Hashtable (Class)

This class implements a hashtable, which is a data structure that maps keys to values. Any non-null object can be used as a key or as a value. Objects used as keys implement the hashCode method which is inherited by all objects from the java.lang.Object class.

4.3.4 ModelObserver (Class)

This interface must be implemented by any object that would like to attach to the DataModel as an observer and get updated as system objects are added, deleted or changed.

4.3.5 Navigable (Class)

This interface will be implemented by any class that supports the Navigator on either the left or right side (the tree or list view). This includes the functionality common to both the tree and list.

4.3.6 Navigator (Class)

This class represents one instance of the Navigator window. It supplies methods for opening the Navigator window and for maintaining the collection of Navigables after the Navigator is open.

4.3.7 NavigatorSupporter (Class)

This interface must be implemented by any subsystem that supports invoking the Navigator. It must be able to supply the Navigable objects, and also can support user interaction with the selected Navigable objects through menus and drag/drop.

4.3.8 NavList (Class)

This class represents the right hand side of the Navigator window (the list or report). It contains functionality for changing the NavTreeDisplayable to refill the list, and also for maintaining the Navigables in the list after the Navigables belonging to the NavTreeDisplayable are already displayed.

4.3.9 NavListDisplayable (Class)

This interface must be implemented by any object to be displayed on the right hand side of the Navigator window, in the list view. In addition to the Navigable methods, it must also support getting and comparing the strings for a given property (column) in the list.

4.3.10 javax.swing.table. AbstractTableModel (Class)

This class provides a base implementation of the TableModel interface. This data structure will be used to supply a JTable with data.

4.3.11 javax.swing.tree. DefaultTreeModel (Class)

This class is the data structure that is used as a foundation for the JTree class.

4.3.12 javax.swing.tree. MutableTreeNode (Class)

This interface extends the TreeNode interface and provides the ability to add and remove children from nodes. It may be used in a TreeModel.

4.3.13 NavTableModel (Class)

This class will serve as the data structure for the right hand side of the Navigator, and will be the foundation of the JTable which will display the data stored in the model.

4.3.14 NavTree (Class)

This class represents the left hand side of the Navigator window - the tree view. It contains functionality for maintaining the NavTreeDisplayable objects that are in the tree.

4.3.15 NavTreeDisplayable (Class)

This interface must be implemented by any objects that are to be added to the left side of the Navigator (the tree view). This contains all of the functionality to support the tree data structure and also provides the property list (column headers) which will be displayed in the list view when the NavTreeDisplayable is selected.

4.3.16 NavTreeModel (Class)

This class will provide the data structure that will support the tree structure on the left hand side of the Navigator.

4.4 GUIDMSClasses (Class Diagram)

Figure 4-4. GUIDMSClasses (Class Diagram)

4.4.1 CosEvent.PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

4.4.2 DefaultMultiFormatter (Class)

This class is the default implementation of the MultiFormatter interface that is used by the DMSMessageEditor upon creation. It is possible to change the parsing used by the editor by setting the installed MultiFormatter. This implementation formats the messages using a very simple algorithm which left justifies each line and fills pages from the top down.

4.4.3 DMS (Class)

This class represents a Dynamic Message Sign (DMS). It has attributes and methods for controlling and maintaining the status of the DMS within the system.

4.4.4 DMSLibraryNavGroup (Class)

This class provides a navigator group under which all DMS message libraries will appear.

4.4.5 DMSNavGroup (Class)

This class provides a navigator group under which all DMS objects will appear.

4.4.6 DMSMessageLibraryProperties (Class)

This class implements a dialog box that will be used to modify the properties of a GUIDMSMessageLibrary.

4.4.7 DMSMessageView (Class)

This class implements a window which is capable of displaying any MULTI formatted DMS message as a pixmap in order to give the operator an idea of how the message will look on a particular sign.

4.4.8 DMSStoredMsgItem (Class)

This class represents a plan item that is used to associate a stored DMS message with a specific DMS. When the item is activated, it sets the message of the DMS to the stored message to which it is linked.

4.4.9 DMSPropertiesDialog (Class)

This class is responsible for the display and processing of the DMS Properties dialog box that allows a user to configure the properties of a particular DMS.

4.4.10 GUIDMS (Class)

This class provides a wrapper for the DMS interface. This is the class that the other GUI components will interact with in order to perform GUI operations such as create a menu for the DMS.

4.4.11 GUIDMSModule (Class)

This module is an installable GUI module that handles all of the DMS-specific functionality.

4.4.12 GUIDMSStoredMsgItem (Class)

This class wraps a plan item for a DMS. This is done to cache the data locally as well as to give the item GUI-specific functionality.

4.4.13 GUIPlan (Class)

This class is a GUI wrapper for the Plan object. The wrapping is done to cache the data locally for faster access, as well as to give the Plan some GUI-specific functionality such as menus and command handling.

4.4.14 DMSStoredMessage (Class)

This class represents a stored DMS message that is created by the DMS Message Editor and stored in the database. It can be displayed on multiple DMS models and contains an attribute stating the minimum width of a sign that can display the message in its entirety.

4.4.15 GUIDMSStoredMessage (Class)

This class will wrap a DMSStoredMessage object in order to allow it to perform GUI specific functionality. It delegates operations to the wrapped object.

4.4.16 Identifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

4.4.17 GUIPlanItem (Class)

This is a base class for all of the GUIPlanItem classes. Each GIUPlanItem object will serve as a GUI wrapper to cache the plan item data locally and also to handle all user interaction in the GUI, such as menus and command handling.

4.4.18 InstallableModule (Class)

This class is the basic interface that all installable modules must implement. It contains functionality that all modules must support to be installable modules. This includes functionality for startup, shutdown, login, logout, and the handling of system and user preferences.

4.4.19 Menuable (Class)

This interface allows an object to provide menu item strings and receive commands when the corresponding menu items are clicked on. It supports both single selection and multiple selection of Menuable objects. The getSSMenuItems() method should return the menu items to display if the object is singly selected. The getMSMenuItems() method should return the menu items that the Menuable object wishes to display if other Menuable objects are selected. The access token is passed to these methods to allow the Menuable object to check the user's access rights before supplying the strings, so the user's actions may be restricted.

4.4.20 NavTreeDisplayable (Class)

This interface must be implemented by any objects that are to be added to the left side of the Navigator (the tree view). This contains all of the functionality to support the tree data structure and also provides the property list (column headers) which will be displayed in the list view when the NavTreeDisplayable is selected.

4.4.21 MultiFormatter (Class)

This interface must be implemented by classes which convert plain text DMS messages to MULTI formatted messages.

4.4.22 MultiParseListener (Class)

A MultiParseListener works in conjunction with the MultiConverter to allow an implementing class to be notified as parsing of a MULTI message occurs. An exemplary use of a MultiParseListener would be the MessageView window that will need to have the MULTI message parsed in order to display it as a pixmap.

4.4.23 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

4.4.24 DMSMessageEditor (Class)

This class is responsible for allowing an operator to set the current message on a DMS. It also updates a MessageView to allow the operator to preview the message as it will look on the selected sign, prior to sending the message to the sign controller.

4.4.25 GUIDMSMessageLibrary (Class)

This class wraps a DMSMessageLibrary object in order to allow it to perform GUI specific operations. It will delegate operations to the wrapped object.

4.4.26 DMSMessageLibrary (Class)

This class represents a logical collection of stored DMS messages that are stored in the database.

4.4.27 DMSStoredMsgItemProperties (Class)

This class will provide a dialog that will allow an operator to edit the properties of a DMSStoredMessageItem.

4.4.28 GUIDictionary (Class)

This class is a GUI wrapper for the Dictionary class. It adds functionality for caching the data and for adding GUI-specific functionality such as menus and Navigator support.

4.4.29 PlanItem (Class)

This class represents an action within the system that can be planned in advance. This abstract class is subclassed for specific actions that can be planned in the system.

4.4.30 DMSTrueDisplay (Class)

This class is responsible for recognizing that a DMS message has changed and updating the associated MessageView in order to convey that change to the operator.

4.4.31 GUIModelObserver (Class)

Interface to be implemented by GUI components that would like to observe changes to the data model. Observers of this type will be notified of changes on the GUI event dispatch thread.

4.4.32 java.awt.event.ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

4.4.33 NavListDisplayable (Class)

This interface must be implemented by any object to be displayed on the right hand side of the Navigator window, in the list view. In addition to the Navigable methods, it must also support getting and comparing the strings for a given property (column) in the list.

4.4.34 PlanItemCreationSupporter (Class)

This interface must be implemented in any modules that wish to support the plan module. The modules must attach their PlanItemCreationSupporters at startup. The GUIPlanModule will then call the supporter when it is time to display the Plan menu or to create a specific type of plan item or GUIPlanItem.

4.4.35 java.awt.event.KeyListener (Class)

Interface that a class must realize in order for objects of that class to be notified when the user presses a key.

4.4.36 SHAMultiFormatter (Class)

This class is the implementation of the MultiFormatter interface that will use the MDSHA defined message formatting algorithm. It will be aware of the desired justification, line ordering per page and special word connection requirements of this algorithm.

4.5 GUIDictionaryModuleClasses (Class Diagram)

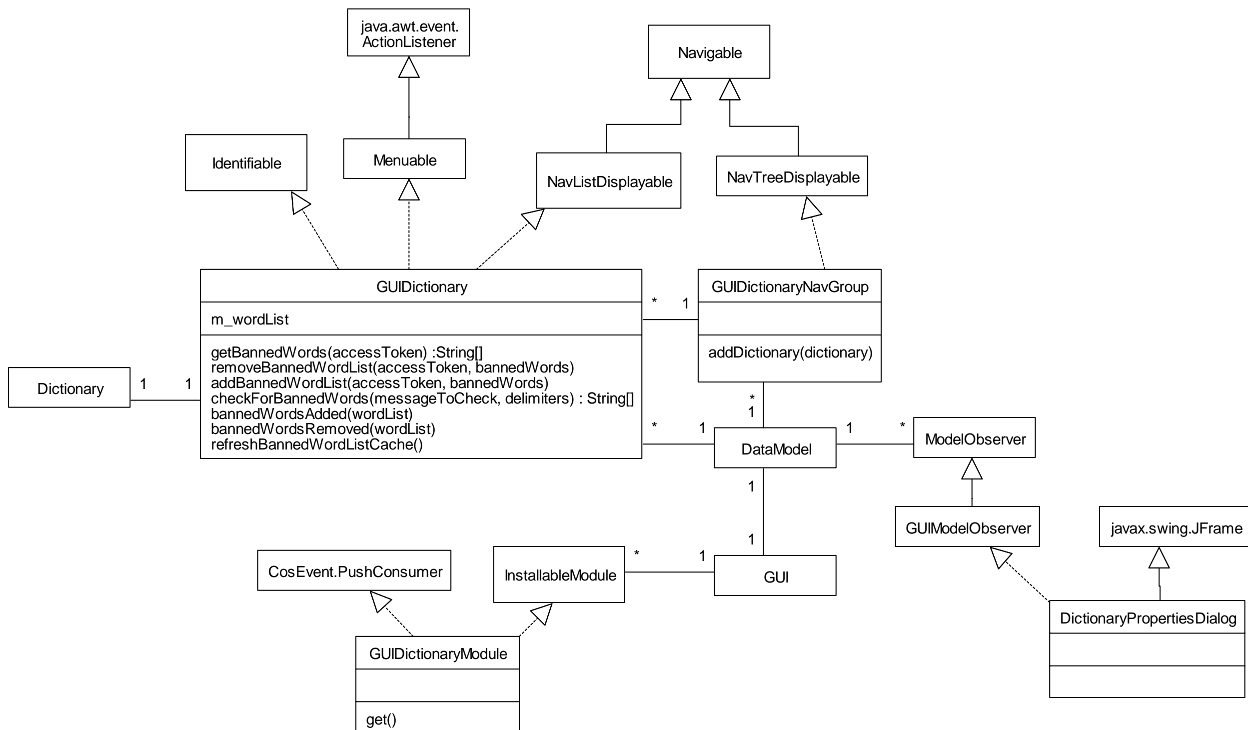


Figure 4-5. GUIDictionaryModuleClasses (Class Diagram)

4.5.1 CosEvent.PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

4.5.2 Dictionary (Class)

This class is used to check for banned words in a message that may be displayed on a DMS. In addition to methods for checking the words, it has methods to allow the contents of the dictionary to be changed.

4.5.3 DictionaryPropertiesDialog (Class)

This dialog is the editing interface which allows the user to view, add, and remove banned words from a given dictionary.

4.5.4 GUIDictionary (Class)

This class is a GUI wrapper for the Dictionary class. It adds functionality for caching the data and for adding GUI-specific functionality such as menus and Navigator support.

4.5.5 GUIDictionaryModule (Class)

This class is an installable GUI module that handles all of the dictionary-specific functionality in the GUI.

4.5.6 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

4.5.7 GUIDictionaryNavGroup (Class)

This class is used to support the required Navigator functionality to group the GUIDictionaries together for the purpose of being displayed together under one branch of the Navigator tree.

4.5.8 GUIModelObserver (Class)

Interface to be implemented by GUI components that would like to observe changes to the data model. Observers of this type will be notified of changes on the GUI event dispatch thread.

4.5.9 Identifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

4.5.10 InstallableModule (Class)

This class is the basic interface that all installable modules must implement. It contains functionality that all modules must support to be installable modules. This includes functionality for startup, shutdown, login, logout, and the handling of system and user preferences.

4.5.11 javax.swing.JFrame (Class)

Java class that displays a frame window.

4.5.12 Menuable (Class)

This interface allows an object to provide menu item strings and receive commands when the corresponding menu items are clicked on. It supports both single selection and multiple selection of Menuable objects. The getSSMenuItems() method should return the menu items to display if the object is singly selected. The getMSMenuItems() method should return the menu items that the Menuable object wishes to display if other Menuable objects are selected. The access token is passed to these methods to allow the Menuable object to check the user's access rights before supplying the strings, so the user's actions may be restricted.

4.5.13 GUI (Class)

This class is a singleton that contains all of the centralized functionality in the GUI. This includes startup, shutdown, login, and logout. It manages the installable modules and controls all functionality that requires the modules to be called. In addition, it stores all of the CORBA object wrappers in the DataModel, which allows access to the objects and supports an update mechanism to notify interested observers whenever the objects change.

4.5.14 java.awt.event. ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

4.5.15 ModelObserver (Class)

This interface must be implemented by any object that would like to attach to the DataModel as an observer and get updated as system objects are added, deleted or changed.

4.5.16 Navigable (Class)

This interface will be implemented by any class that supports the Navigator on either the left or right side (the tree or list view). This includes the functionality common to both the tree and list.

4.5.17 NavListDisplayable (Class)

This interface must be implemented by any object to be displayed on the right hand side of the Navigator window, in the list view. In addition to the Navigable methods, it must also support getting and comparing the strings for a given property (column) in the list.

4.5.18 NavTreeDisplayable (Class)

This interface must be implemented by any objects that are to be added to the left side of the Navigator (the tree view). This contains all of the functionality to support the tree data structure and also provides the property list (column headers) which will be displayed in the list view when the NavTreeDisplayable is selected.

4.6 GUIPlanClasses (Class Diagram)

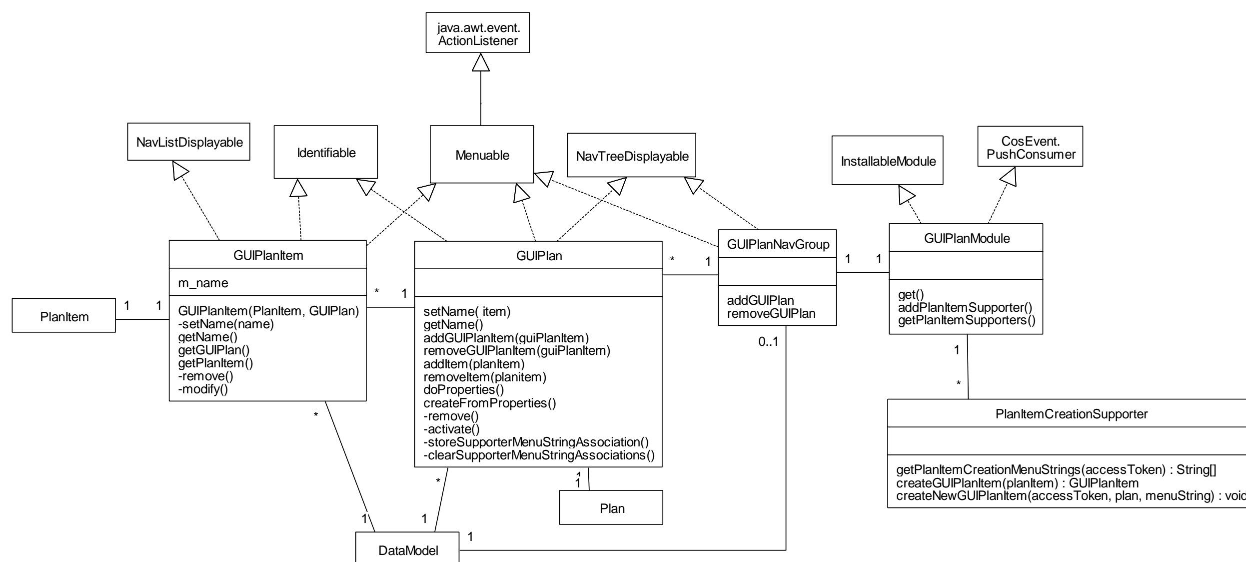


Figure 4-6. GUIPlanClasses (Class Diagram)

4.6.1 DataModel (Class)

The data model class serves as a collection of objects. It provides an efficient lookup mechanism for locating any object, and methods that allow for the retrieval of all objects of a particular type. Additionally, this class provides the ability to attach observer objects that are notified when objects are added to or removed from the model. Objects may also notify the DataModel that they have been modified. The model will periodically notify all attached observers of the changes to objects in the model.

4.6.2 GUIPlan (Class)

This class is a GUI wrapper for the Plan object. The wrapping is done to cache the data locally for faster access, as well as to give the Plan some GUI-specific functionality such as menus and command handling.

4.6.3 GUIPlanItem (Class)

This is a base class for all of the GUIPlanItem classes. Each GIUPlanItem object will serve as a GUI wrapper to cache the plan item data locally and also to handle all user interaction in the GUI, such as menus and command handling.

4.6.4 GUIPlanModule (Class)

This is an installable GUI module that handles the Plan functionality in the GUI. Other modules which support plan items must attach their PlanItemCreationSupporters to the GUIPlanModule at startup. The plan module will then call them as necessary when it is necessary to create a specific type of GUIPlanItem.

4.6.5 PlanItem (Class)

This class represents an action within the system that can be planned in advance. This abstract class is subclassed for specific actions that can be planned in the system.

4.6.6 CosEvent. PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

4.6.7 GUIPlanNavGroup (Class)

This class is the branch in the Navigator tree that contains all of the GUIPlan objects. It will provide functionality for displaying a menu for creating new plans.

4.6.8 Menuable (Class)

This interface allows an object to provide menu item strings and receive commands when the corresponding menu items are clicked on. It supports both single selection and multiple selection of Menuable objects. The getSSMenuItems() method should return the menu items to display if the object is singly selected. The getMSMenuItems() method should return the menu items that the Menuable object wishes to display if other Menuable objects are selected. The access token is passed to these methods to allow the Menuable object to check the user's access rights before supplying the strings, so the user's actions may be restricted.

4.6.9 Identifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

4.6.10 NavTreeDisplayable (Class)

This interface must be implemented by any objects that are to be added to the left side of the Navigator (the tree view). This contains all of the functionality to support the tree data structure and also provides the property list (column headers) which will be displayed in the list view when the NavTreeDisplayable is selected.

4.6.11 Plan (Class)

This class has a collection of Plan Items that it maintains. It has functionality for changing the plan items, and also allows the plan to be activated, which has the effect of activating each plan item in the plan.

4.6.12 InstallableModule (Class)

This class is the basic interface that all installable modules must implement. It contains functionality that all modules must support to be installable modules. This includes functionality for startup, shutdown, login, logout, and the handling of system and user preferences.

4.6.13 java.awt.event. ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

4.6.14 NavListDisplayable (Class)

This interface must be implemented by any object to be displayed on the right hand side of the Navigator window, in the list view. In addition to the Navigable methods, it must also support getting and comparing the strings for a given property (column) in the list.

4.6.15 PlanItemCreationSupporter (Class)

This interface must be implemented in any modules that wish to support the plan module. The modules must attach their PlanItemCreationSupporters at startup. The GUIPlanModule will then call the supporter when it is time to display the Plan menu or to create a specific type of plan item or GUIPlanItem.

4.7 GUIUserManagementClasses (Class Diagram)

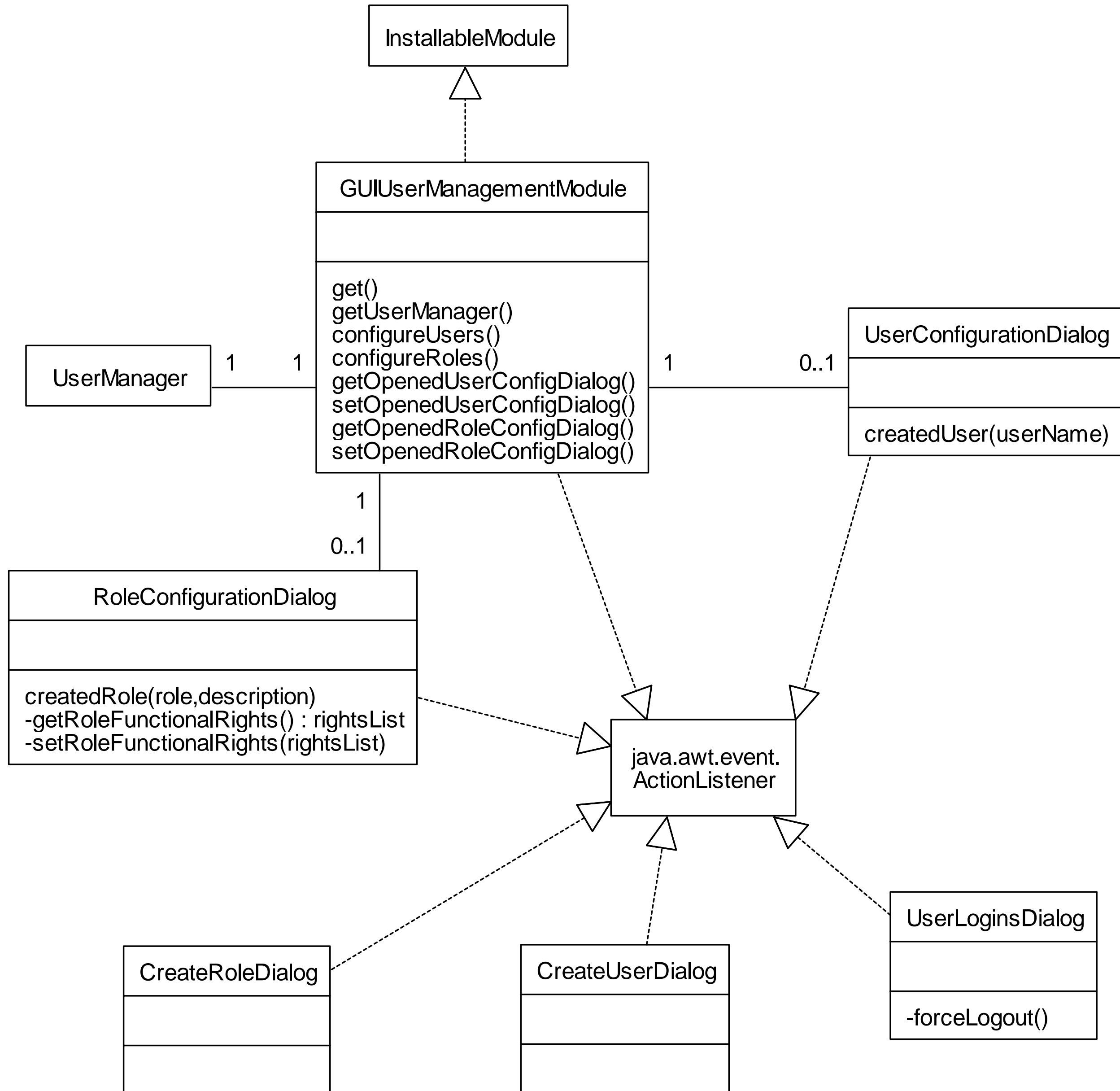


Figure 4-7. GUIUserManagementClasses (Class Diagram)

4.7.1 CreateRoleDialog (Class)

This dialog allows the administrator to create a new role.

4.7.2 CreateUserDialog (Class)

This dialog allows the administrator to create a new user.

4.7.3 GUIUserManagementModule (Class)

This class implements the InstallableModule interface and performs functionality for managing user rights. It can be called to configure the roles and users, or to force a logout.

4.7.4 InstallableModule (Class)

This class is the basic interface that all installable modules must implement. It contains functionality that all modules must support to be installable modules. This includes functionality for startup, shutdown, login, logout, and the handling of system and user preferences.

4.7.5 java.awt.event. ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

4.7.6 UserLoginsDialog (Class)

This dialog displays a list of currently logged in users and allows the administrator to force one or more users to be logged out.

4.7.7 UserManager (Class)

The UserManager provides access to data dealing with user management. This includes users, roles, and functional rights. The UserManager is largely an interface to the User Management database tables.

4.7.8 RoleConfigurationDialog (Class)

This dialog allows the administrator to configure the roles in the system. It supports the Create Role, Delete Role, and Set Role Functional Rights functionality. If the user does not have role configuration rights, all editing functionality will be disabled.

4.7.9 UserConfigurationDialog (Class)

This dialog allows the administrator to view or configure the users' roles, assuming the roles have been defined. It supports the Create User, Delete User, Change User Password, Grant Role and Revoke Role functionality. If the user has view rights but not configuration rights, all configuration abilities will be disabled.

4.8 UtilityClasses (Class Diagram)

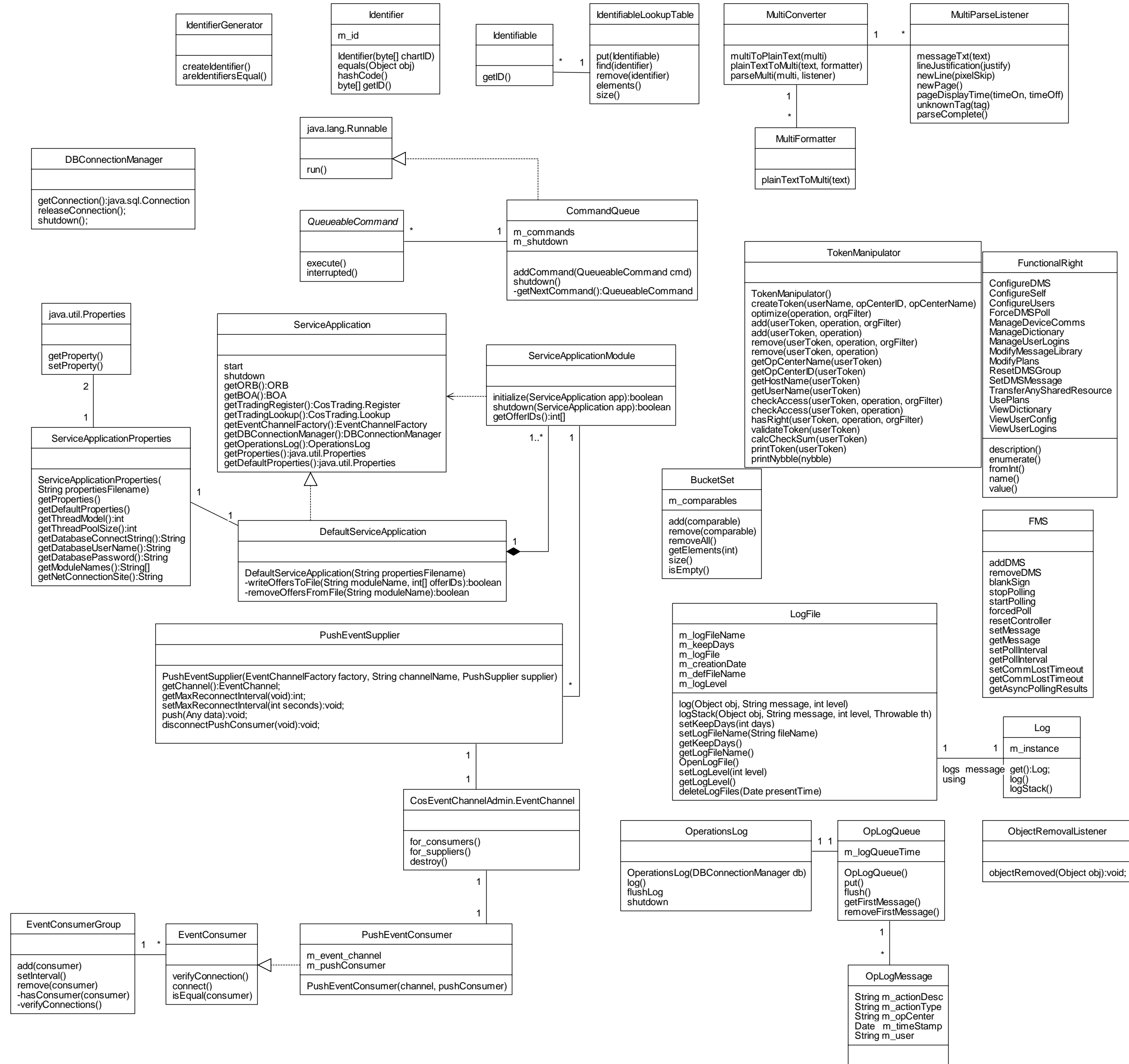


Figure 4-8. UtilityClasses (Class Diagram)

4.8.1 CosEventChannelAdmin.EventChannel (Class)

The event channel is a service that decouples the communication between suppliers and consumers of information.

4.8.2 DBConnectionManager (Class)

This class implements a database connection manager that manages a pool of database connections. Any CHART II system thread requiring database access gets a database connection from the pool of connections maintained by this manager class. The connections are maintained in two seperate lists namely, inUseList and freeList. The inUseList contains connections that have already been assigned to a thread. The freeList contains unassigned connections. This class assumes that an appropriate JDBC driver has been loaded either by using the "jdbc.drivers" system property or by loading it explicitly. The class has a monitor thread that is started by the constructor. This connection monitor thread periodically checks the inuseList to see if there are connections that are owned by dead threads and move such connections to the freeList. The connection monitor thread is started only if a non-zero value is specified for the monitoring time interval in the constructor.

4.8.3 DefaultServiceApplication (Class)

This class is the default implementation of the ServiceApplication interface. This class is passed a properties file during construction. This properties file contains configuration data used by this class to set the ORB concurrency model, determine which ORB services need to available, provide database connectivity, etc. The properties file also contains the class names of service modules that should be served by the service application. During startup, the DefaultServiceApplication instantiates the service application module classes listed in the properties file and initializes each.

The DefaultServiceApplication maintains a file of offers that have been exported to the Trading Service. Each module must provide an implementation of the getOfferIDs method and be able to return the offer ids for each object they have exported to the trader during their initialization. The DefaultServiceApplication stores all offer IDs in a file during its startup. Each module is expected to remove its offers from the trader during a shutdown. If the DefaultServiceApplication is not shutdown properly, it uses its offer ID file to clean-up old offers prior to initializing modules during its next start. This keeps multiple offers for the same object from being placed in the trader.

4.8.4 EventConsumer (Class)

This interface provides the methods that any EventConsumer object that would like to be managed in an EventConsumerGroup must implement.

4.8.5 EventConsumerGroup (Class)

This class represents a collection of event consumers that will be monitored to verify that they do not lose their connection to the CORBA event service. The class will periodically ask each consumer to verify its connection to the event channel on which it is dependent to receive events.

4.8.6 CommandQueue (Class)

The CommandQueue class provides a queue for QueuableCommand objects. The CommandQueue has a thread that it uses to process each QueuableCommand in a first in first out

order. As each command object is pulled off the queue by the CommandQueue's thread, the command object's execute method is called, at which time the command performs its intended task.

4.8.7 FMS (Class)

This class represents the CHART II system's interface to the FMS SNMP manager. Most methods included in this class have an associated method in the FMS SNMP Manager DLL provided by the FMS Subsystem. The other methods in this class exist to provide easier interface to the DLL. As an example, this class contains a blankSign method that actually calls setMessage on the FMS Subsystem with the message set to blank and beacons off.

4.8.8 FunctionalRight (Class)

This class acts as an enumuration that lists the types of functional rights possible in the CHART2 system. It contains a static member for each possible functional right.

4.8.9 Identifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

4.8.10 Identifier (Class)

Wrapper class for a CHART2 identifier byte sequence. This class will be used to add identifiable objects to hash tables and perform subsequent lookup operations.

4.8.11 OpLogQueue (Class)

This class is a queue for messages that are to be put into the system's Operations Log. Messages added to the queue can be removed in FIFO order.

4.8.12 PushEventSupplier (Class)

This class provides a utility for application modules that push events on an event channel. The user of this class can pass a reference to the event channel factory to this object. The constructor will create a channel in the factory. The push method is used to push data on the event channel. The push method is able to detect if the event channel or its associated objects have crashed. When this occurs, a flag is set, causing the push method to attempt to reconnect the next time push is called. To avoid a supplier with a heavy supply load from causing reconnect attempts to occur too frequently, a maximum reconnect interval is used. This interval specifies the quickest reconnect interval that can be used. The push method uses this interval and the current time to determine if a reconnect should be attempted, thus reconnects can be throttled indepently of a supplier's push rate.

4.8.13 IdentifiableLookupTable (Class)

This class uses a hash table implementation to store Identifiable objects for fast lookups.

4.8.14 ObjectRemovalListener (Class)

This interface is implemented by objects that wish to be notified of objects being removed from the system. This is typically used by objects that store a collection of other objects, such as a factory, to allow them to remove objects from their collection when the object is to be removed from the system.

4.8.15 BucketSet (Class)

This class is designed to contain a collection of comparable objects. All of the objects added to this collection must be of the same concrete type. Each element in the collection has an associated counter that tracks how many times this element has been added. It is then possible to get only the elements which have been added to the collection n times where n is a positive integer value. This class is very useful for creating GUI menu's for multiple objects as it allows all objects to insert their menu items and then allows the user to get only those items which all objects inserted.

4.8.16 IdentifierGenerator (Class)

This class is used to create and manipulate identifiers that are to be used in Identifiable objects.

4.8.17 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

4.8.18 java.util.Properties (Class)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its "defaults"; this second property list is searched if the property key is not found in the original property list.

4.8.19 Log (Class)

Singleton log object to allow applications to easily create and utilize a LogFile object for system trace messages.

4.8.20 LogFile (Class)

This class creates a flat file for writing system trace log messages and purges them at user specified interval. The log files created by this class are used for system debugging and maintenance only and are not to be confused with the system operations log which is modeled by the OperationsLog class.

4.8.21 OperationsLog (Class)

This class provides the functionality to add a log entry to the Chart II operations log. At the time of instantiation of this class, it creates a queue for log entries. When a user of this class provides a message to be logged, it creates a time-stamped OpLogMessage object and adds this object to the OpLogQueue. Once queued, the messages are written to the database by the queue driver thread in the order they were queued.

4.8.22 PushEventConsumer (Class)

This class is a utility class that will be responsible for connecting a consumer implementation to an event channel, and maintaining that connection. When the verifyConnection method is called, this object will determine if the channel has been lost and will attempt to re-connect to the channel if it has.

4.8.23 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

4.8.24 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a ChartII service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, BOA, Trader, and Event Service.

4.8.25 ServiceApplicationProperties (Class)

This class provides methods that allow the DefaultServiceApplication to access the necessary properties from the java properties configuration file. It also provides a default properties file which can be retrieved by anyone holding a ServiceApplication interface reference. This gives each installed service module the opportunity to load default values before retrieving property values from the properties file.

4.8.26 TokenManipulator (Class)

This class contains all functionality required for user rights in the system. It is the only code in the system that knows how to create, modify and check a user's functional rights. It encapsulates the contents of an octet sequence which will be passed to every secure method. Secure methods should call the checkAccess method to validate the user. Client processes should use the check access method to verify access and optimize to reduce reduce the size of the sequence to only those rights which are necessary to invoke the secure method. The token contains the following information: Token version, Token ID, Token Time Stamp, Username, Op Center ID, Op Center IOR, and functional rights.

4.8.27 MultiFormatter (Class)

This interface must be implemented by classes which convert plain text DMS messages to MULTI formatted messages.

4.8.28 MultiParseListener (Class)

A MultiParseListener works in conjunction with the MultiConverter to allow an implementing class to be notified as parsing of a MULTI message occurs. An exemplary use of a MultiParseListener would be the MessageView window that will need to have the MULTI message parsed in order to display it as a pixmap.

4.8.29 MultiConverter (Class)

This class provides methods that perform conversions between the DMS MULTI mark-up language and plain text. It also provides a method that will parse a MULTI message and inform a MultiParseListener of elements found in the message.

4.8.30 OpLogMessage (Class)

This class holds data for a message to be stored in the system's Operations Log.

4.8.31 QueueableCommand (Class)

A QueuableCommand is an abstract class used to represent a command that can be placed on a queue for asynchronous execution. Derived classes implement the execute method to specify the actions taken by the command when it is executed.

4.9 SystemInterfaces (Class Diagram)

This class diagram shows the interfaces from the High Level Design that are defined using IDL. These interfaces are included as reference and are included on other class diagrams in this design.

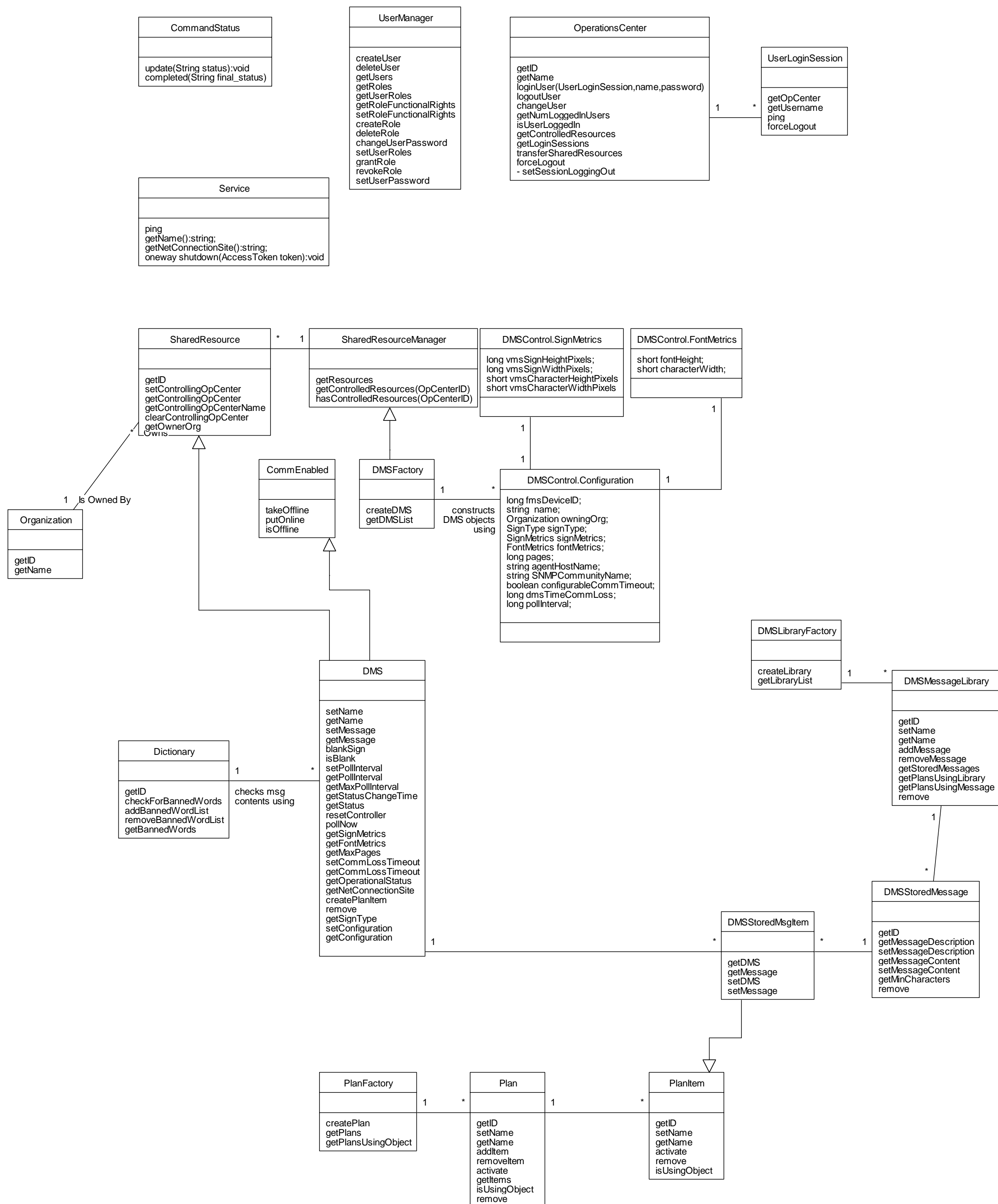


Figure 4-9. SystemInterfaces (Class Diagram)

4.9.1 CommandStatus (Class)

The CommandStatus class is used to allow a calling process to be notified of the progress of an asynchronous operation. This is typically used by a GUI when field communications are involved to complete a method call, allowing the GUI to show the user the progress of the operation. The long running operation calls back to the CommandStatus object periodically as the command is executed and makes a final call to the CommandStatus when the operation has completed. The final call to the CommandStatus from the long running operation indicates the success or failure of the command.

4.9.2 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can have their communications turned on or off. This typically only applies to field devices.

4.9.3 Dictionary (Class)

This class is used to check for banned words in a message that may be displayed on a DMS. In addition to methods for checking the words, it has methods to allow the contents of the dictionary to be changed.

4.9.4 DMS (Class)

This class represents a Dynamic Message Sign (DMS). It has attributes and methods for controlling and maintaining the status of the DMS within the system.

4.9.5 UserLoginSession (Class)

The UserLoginSession class is used to store information about a user that is logged into the system. This object is served from the GUI and provides a means for the servers to call back into the GUI process.

4.9.6 DMSControl.Configuration (Class)

This typedef defines data that is used to identify the configuration of a DMS in the system.

4.9.7 DMSControl.FontMetrics (Class)

This typedef is included in the IDL to specify the data to be passed to/from operations to initialize or query the size of the font used by a DMS.

4.9.8 DMSControl.SignMetrics (Class)

This typedef is included in the IDL to specify the data included in operations that initialize or query the size of a DMS.

4.9.9 DMSStoredMsgItem (Class)

This class represents a plan item that is used to associate a stored DMS message with a specific DMS. When the item is activated, it sets the message of the DMS to the stored message to which it is linked.

4.9.10 OperationsCenter (Class)

The OperationsCenter represents a center where one or more users are located. This class is used to log users into the system. If the username and password provided to the loginUser method are valid, the caller is given a token that contains information about the user and the functional rights of the user. This token is then used to call privileged methods within the system. Shared resources in the system are either available or under the control of an OperationsCenter. The OperationsCenter keeps track of users that are logged in so that it can ensure that the last user does not log out while there are shared resources under its control. This list of logged in users is also available for monitoring system usage or to force users to logout for system maintenance.

4.9.11 DMSLibraryFactory (Class)

This class is used to create new DMS libraries and maintain them in a collection.

4.9.12 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center.

4.9.13 DMSFactory (Class)

The DMSFactory provides a means to create new DMS objects to be added to the system.

4.9.14 DMSMessageLibrary (Class)

This class represents a logical collection of stored DMS messages which are stored in the database.

4.9.15 DMSStoredMessage (Class)

This class represents a stored DMS message that is created by the DMS Message Editor and stored in the database. It can be displayed on multiple DMS models and contains an attribute stating the minimum width of a sign that can display the message in its entirety.

4.9.16 Organization (Class)

The Organization class represents an organization that participates in the Chart system through ownership of shared resources. The Organization can be used in conjunction with functional rights to determine the level of access users have to shared resources owned by a given organization. This allows access to be granted to a user to perform controlled operations on shared resources owned by one organization but not another.

4.9.17 PlanFactory (Class)

This class creates, destroys, and maintains the collection of plans that can be used in the system.

4.9.18 Service (Class)

This interface is implemented by all services in the system that allow themselves to be shutdown externally. All implementing classes provide a means to be cleanly shutdown and can be pinged to detect if they are alive.

4.9.19 Plan (Class)

This class has a collection of Plan Items that it maintains. It has functionality for changing the plan items, and also allows the plan to be activated, which has the effect of activating each plan item in the plan.

4.9.20 PlanItem (Class)

This class represents an action within the system that can be planned in advance. This abstract class is subclassed for specific actions that can be planned in the system.

4.9.21 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

4.9.22 UserManager (Class)

The UserManager provides access to data dealing with user management. This includes users, roles, and functional rights. The UserManager is largely an interface to the User Management database tables.

4.10 JavaClasses (Class Diagram)

This package is included for reference to classes included in the Java programming language that are used in class and sequence diagrams for other packages within this design.

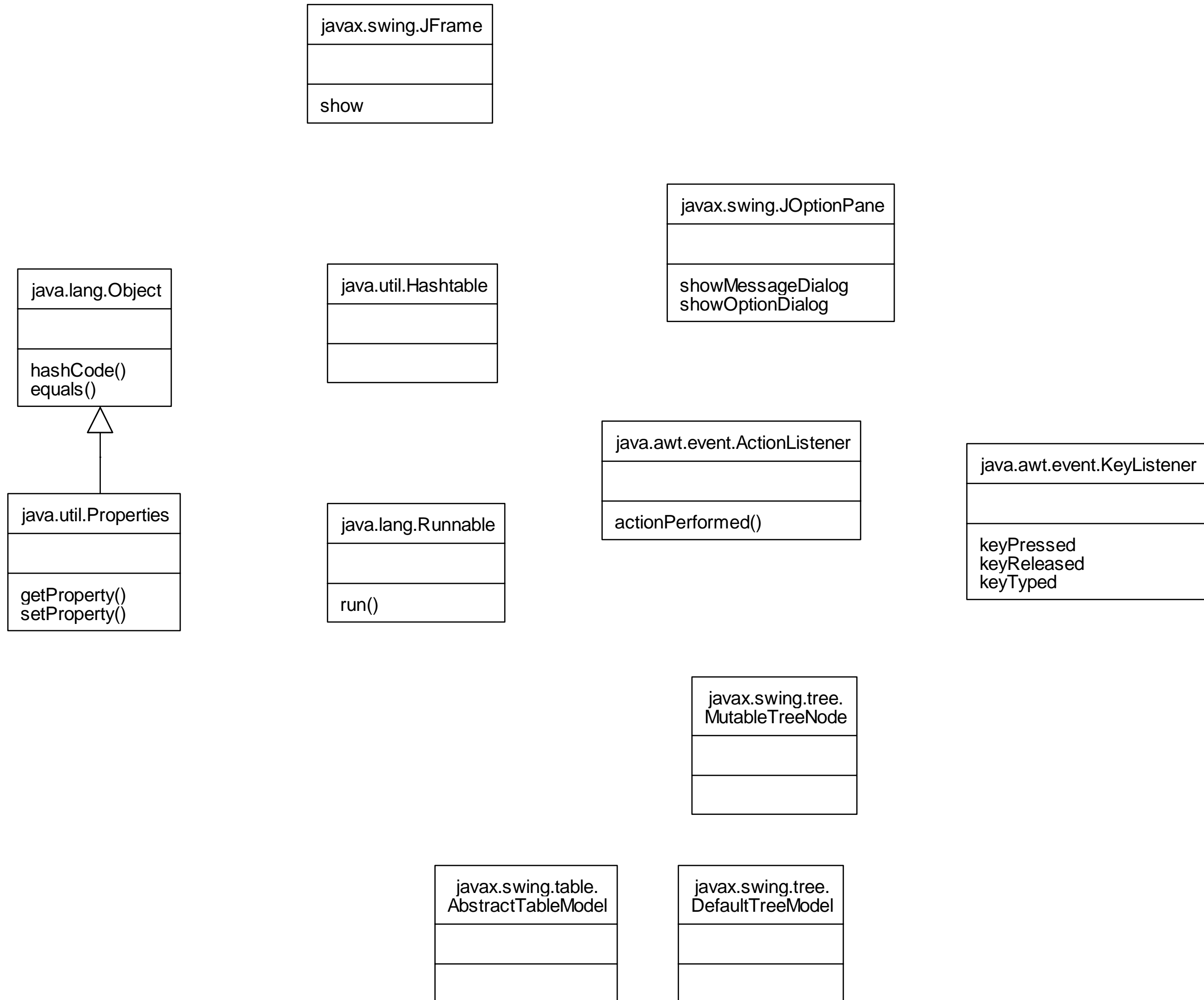


Figure 4-10. JavaClasses (Class Diagram)

4.10.1 java.awt.event.ActionListener (Class)

This interface listens for actions such as when a menu item or button is clicked. For menu items, it is attached to menu items when the menu is built.

4.10.2 java.lang.Object (Class)

This is the base class from which all Java classes inherit.

4.10.3 java.lang.Runnable (Class)

This interface allows the run method to be called from another thread using Java's threading mechanism.

4.10.4 java.util.Hashtable (Class)

This class implements a hashtable, which is a data structure that maps keys to values. Any non-null object can be used as a key or as a value. Objects used as keys implement the hashCode method that is inherited by all objects from the java.lang.Object class.

4.10.5 java.util.Properties (Class)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its "defaults"; this second property list is searched if the property key is not found in the original property list.

4.10.6 javax.swing.table. AbstractTableModel (Class)

This class provides a base implementation of the TableModel interface. This data structure will be used to supply a JTable with data.

4.10.7 javax.swing.tree. DefaultTreeModel (Class)

This class is the data structure that is used as a foundation for the JTree class.

4.10.8 javax.swing.tree. MutableTreeNode (Class)

This interface extends the TreeNode interface and provides the ability to add and remove children from nodes. It may be used in a TreeModel.

4.10.9 java.awt.event.KeyListener (Class)

Interface that a class must realize in order for objects of that class to be notified when the user presses a key.

4.10.10 javax.swing.JFrame (Class)

Java class that displays a frame window.

4.10.11 javax.swing.JOptionPane (Class)

This class is used to display popup messages to an end user.

4.11 CORBAClasses (Class Diagram)

The CORBAUtilities package exists to provide reference to classes that are supplied by the ORB Vendor and are referenced by other packages' class or sequence diagrams.

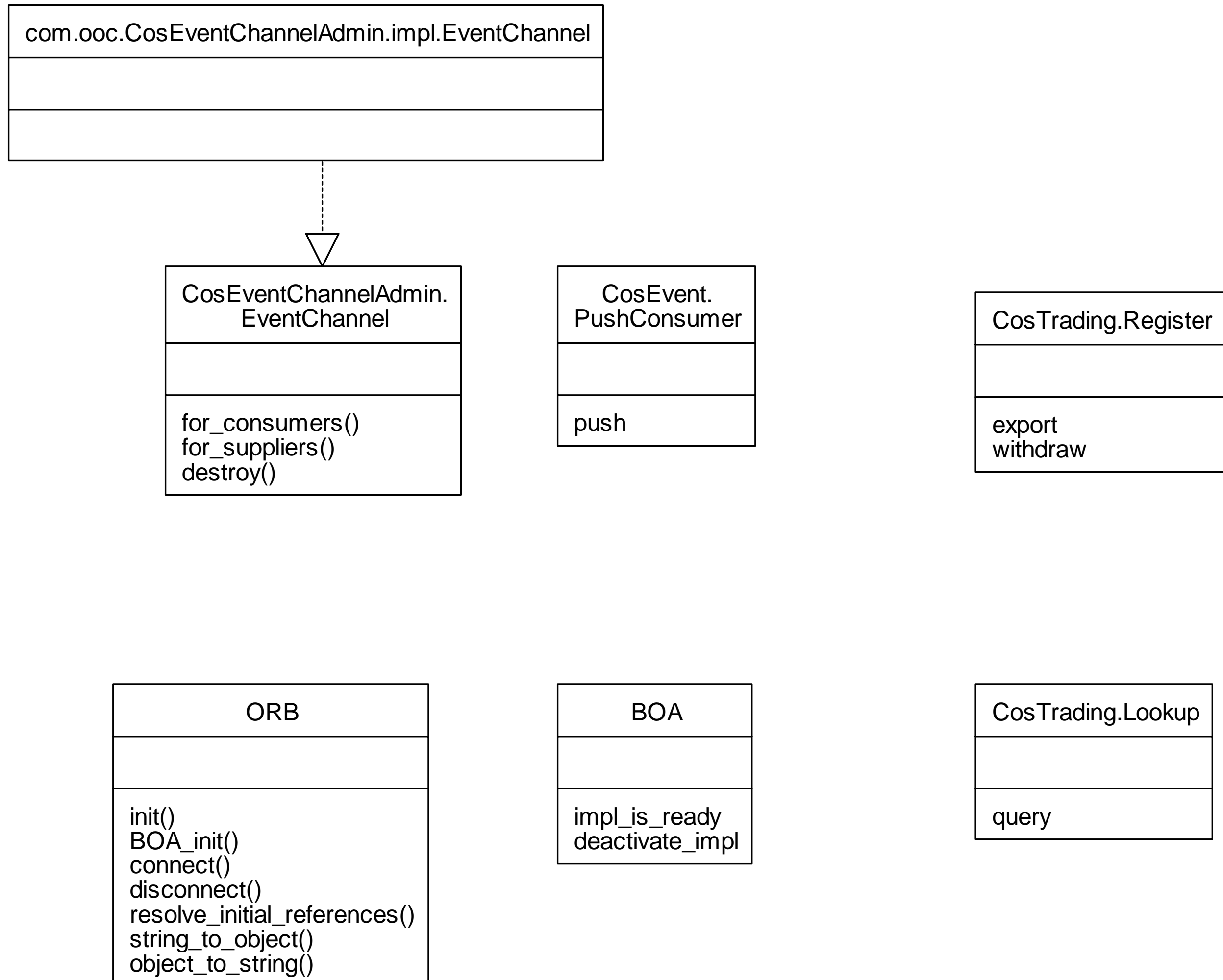


Figure 4-11. CORBAClasses (Class Diagram)

4.11.1 BOA (Class)

The BOA (Basic Object Adapter) is a class that assists implementation objects in using the ORB. Typical services provided include attaching and detaching object implementations to and from the ORB and generation of object references.

4.11.2 com.ooc.CosEventChannelAdmin.impl.EventChannel (Class)

This class is the ORB vendor's implementation of a CORBA event channel. The event service provided by the vendor simply serves one of these objects. The Extended Event Service serves a factory that allows multiple instances of the vendor supplied event channel to be created.

4.11.3 CosEventChannelAdmin. EventChannel (Class)

The event channel is a service that decouples the communication between suppliers and consumers of information.

4.11.4 CosEvent. PushConsumer (Class)

The PushConsumer interface is the interface to an event channel that a supplier of information uses to push event updates to consumers who have previously attached to the channel.

4.11.5 CosTrading.Lookup (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Lookup is the interface that applications use to discover objects that have previously been published.

4.11.6 CosTrading.Register (Class)

The CORBA trading service is an application that CORBA servers and clients use for object publication and discovery respectively. The CosTrading.Register is the interface to the trading service that server applications use to publish objects in order to make them available for client applications to discover.

4.11.7 ORB (Class)

The CORBA ORB (Object Request Broker) provides a common object oriented, remote procedure call mechanism for inter-process communication. The ORB is the basic mechanism by which client applications send requests to server applications and receive responses to those requests from servers.

5 Sequence Diagrams

5.1 GUI:ChangeUserBasic (Sequence Diagram)

This diagram shows the steps that will be taken in the GUI when a user change occurs without first logging out. The new user will be logged in and the previous user will be logged out, then all windows are closed and the new user's preferences are loaded to replace the previous preferences. If the changeUser command fails, the previous user will still be logged in and the new user will not be logged in.

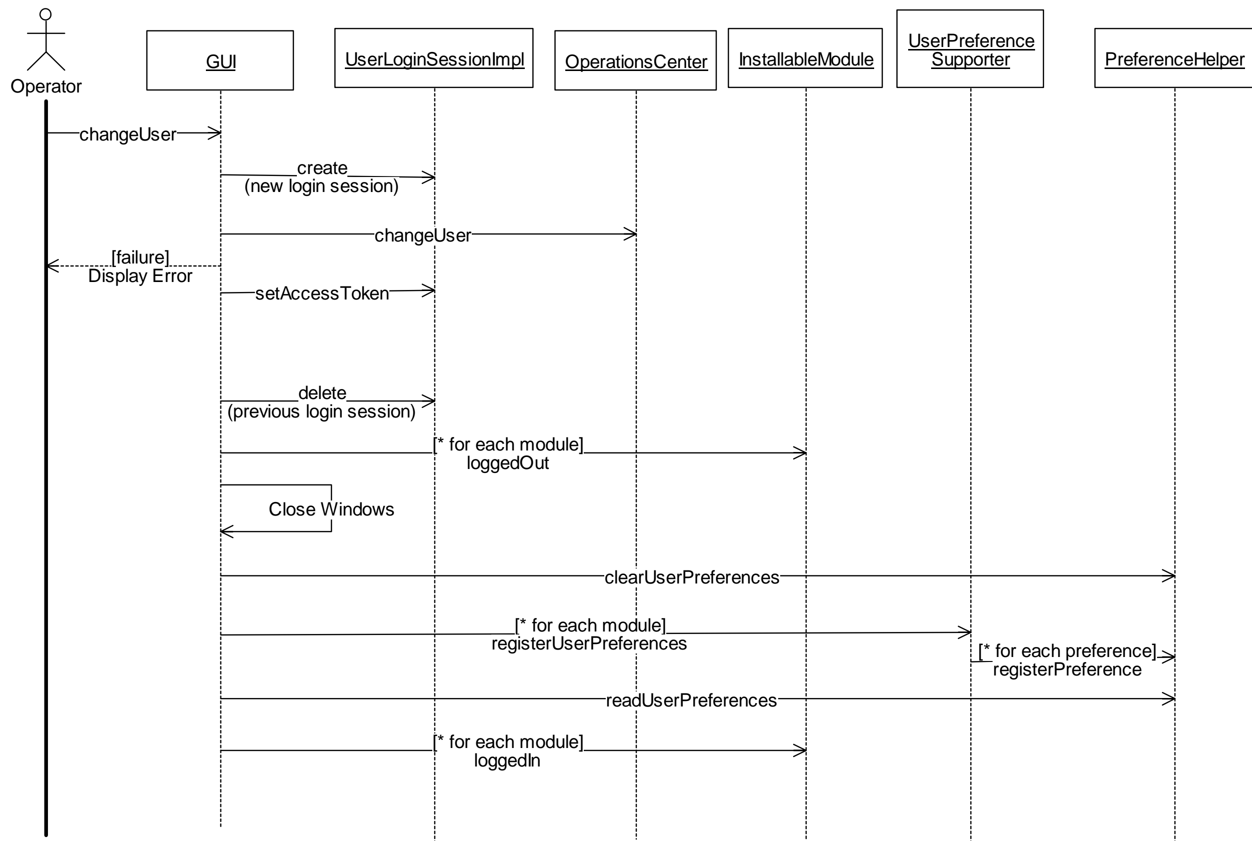


Figure 5-1. GUI:ChangeUserBasic (Sequence Diagram)

5.2 GUI:CommandObjectBasic (Sequence Diagram)

This diagram shows the basic steps involved in issuing a typical command to an object. The context menu is built when the user right clicks on one or more selected objects. At this time the GUI wrapper object will be added as an ActionListener and will receive the command if any of its menu items are clicked on. (See the sequence diagrams GUI:MakeMenuSingleSelect and GUI:MakeMenuMultipleSelect for more details). If a long-running command is invoked, the object will create a CommandStatusImpl object, put it in the DataModel, and pass it to the server so that the server can call back as the command is completed. When the server calls the CommandStatusImpl's completed() method, the CommandStatusImpl will remove itself from the DataModel. If the command fails, the Command Failures window will add the command status to its displayed list.

Figure 5-2. GUI:CommandObjectBasic (Sequence Diagram)

5.3 GUI:ConfigurePreferencesBasic (Sequence Diagram)

This diagram shows the steps necessary to change user preferences (a.k.a. options or properties). When the user clicks on the "Preferences" menu item, the GUI will ask each installed module to return zero or more property pages. The GUI will then construct a property sheet, add the pages to the propery sheet and show it. Each module can call the GUI to retrieve the values to use in the property pages. If the user clicks on OK or Apply, the GUI will ask each module to validate the property page that it supplied. If any fields are invalid, the module should set the focus to the offending control and return false so that the user can correct the mistake. If all pages are valid, the GUI will ask each module to handle the property page. This will cause the modules to call the GUI's setUserPreference(). When all preferences are set, the property sheet will call the GUI to save all of the preferences.

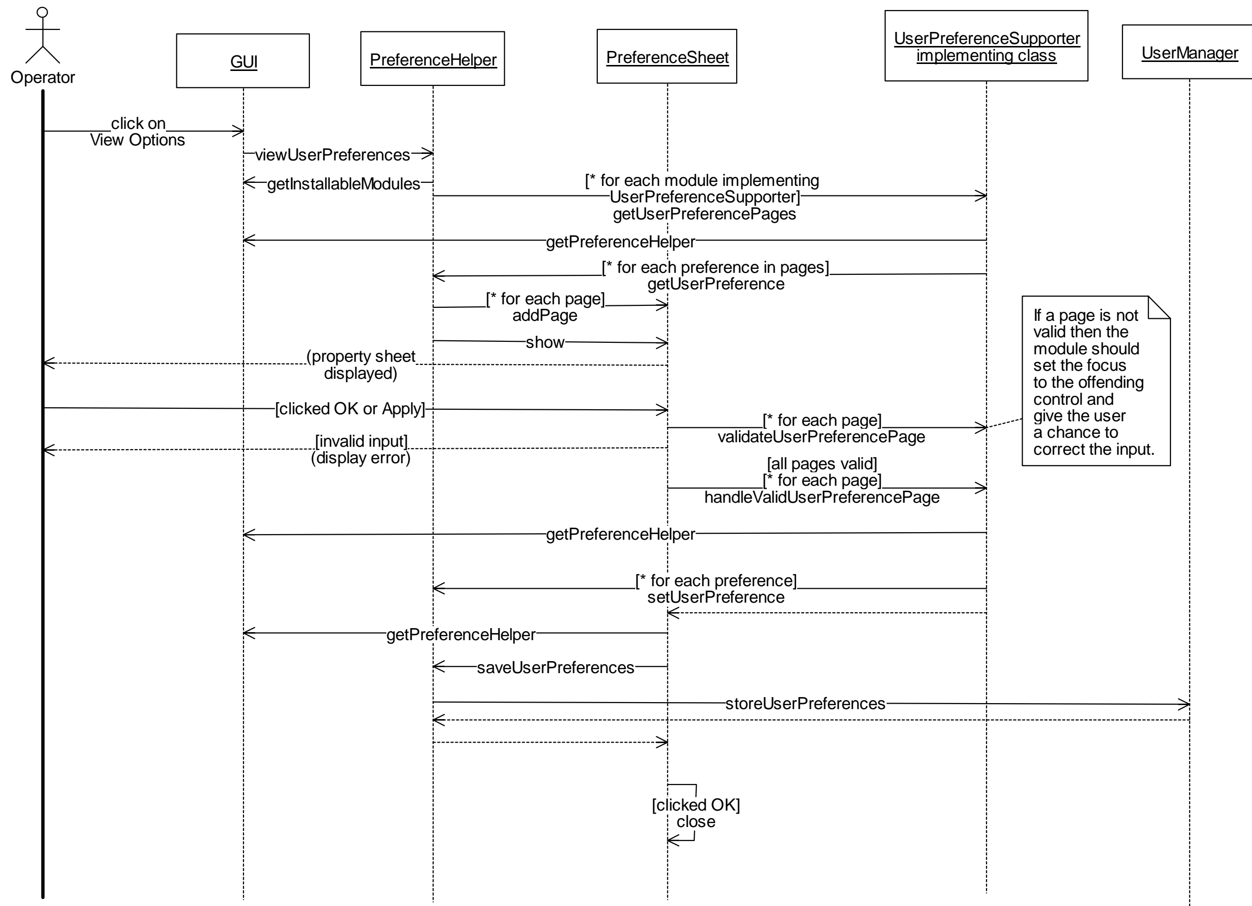


Figure 5-3. GUI:ConfigurePreferencesBasic (Sequence Diagram)

5.4 GUI:DiscoveryBasic (Sequence Diagram)

This diagram shows the ongoing discovery of event channels and served CORBA objects. In the GUI's startup, it will start the DiscoveryThread, which will periodically search for new event channels and objects until the GUI shuts down. The event channels are discovered before the objects to prevent the dropping of events just after the objects are discovered. First, the GUI looks for resource watchdog event channels, which will inform the user if resources are controlled by an Operations Center which does not have any logged in users, then it asks the modules to look for the module-specific event channels. If any event channels are found, they are added to the EventConsumerGroup, which will maintain the connection to the event channel if the event service goes down and is restarted. The GUI will then ask each module to discover the objects that it is interested in. Each module will look up the object factory in the trader, and ask the object factory for all of its objects. Then, the module will check whether the CORBA object already has a GUI wrapper object stored in the DataModel. If it doesn't, it will create a new wrapper object and add it to the DataModel. Any ModelObservers that are attached to the DataModel will be subsequently informed of the new wrapper objects.

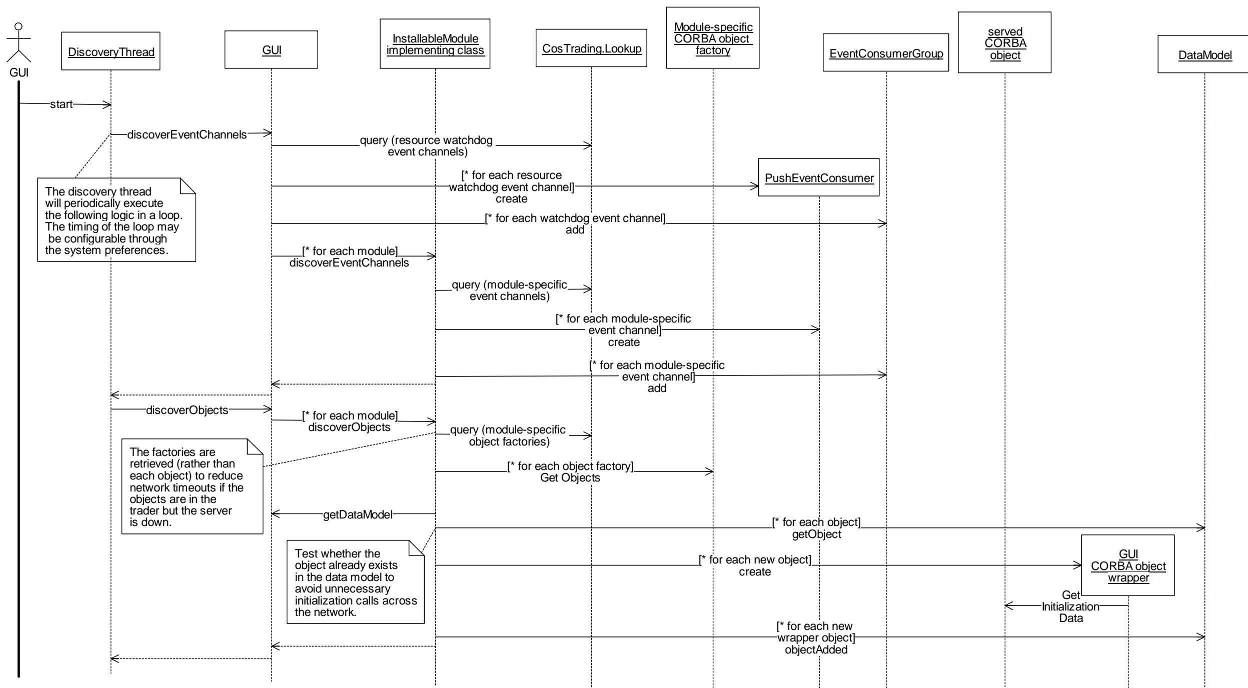


Figure 5-4. GUI:DiscoveryBasic (Sequence Diagram)

5.5 GUI:LoginBasic (Sequence Diagram)

This diagram shows what steps must be taken at login. The GUI creates a UserLoginSessionImpl and passes it to the OperationsCenter for login. The GUI will then store the AccessToken in the UserLoginSessionImpl for later use. The GUI will then ask each module to register its user preferences, then it will read the preferences and call each module's loggedIn() method.

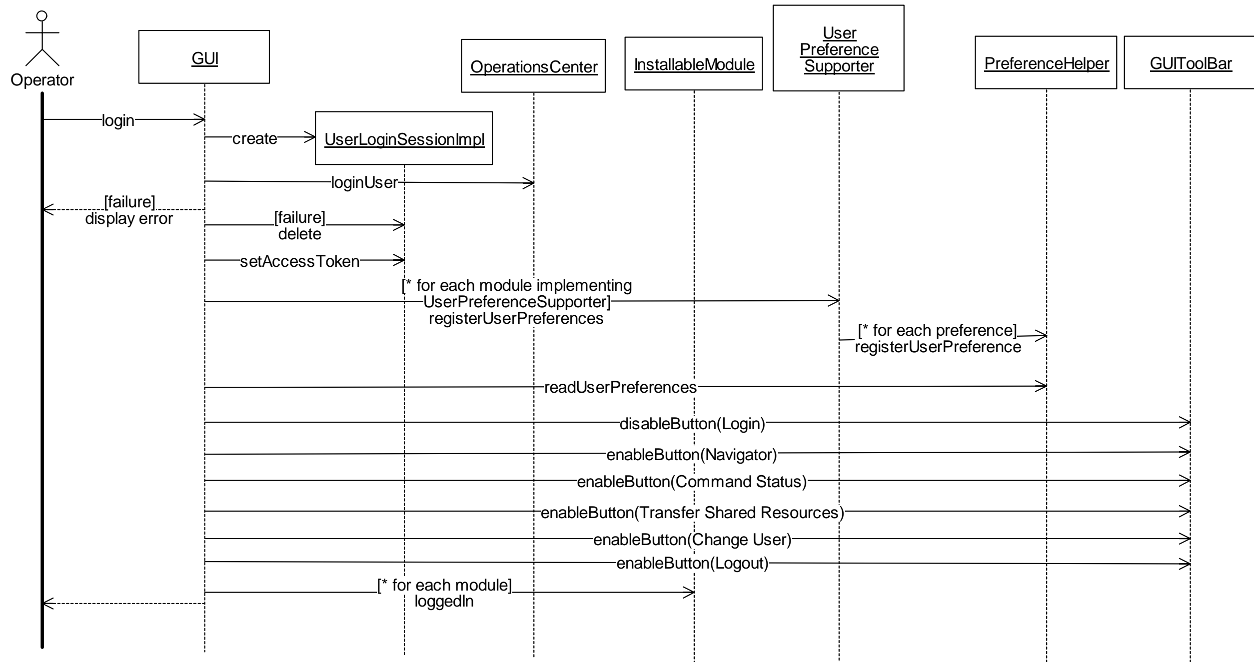


Figure 5-5. GUI:LoginBasic (Sequence Diagram)

5.6 GUI:LogoutBasic (Sequence Diagram)

First the GUI will call the OperationsCenter to log the user out. If any shared resources are still assigned to the Op Center, the logout will fail and the user will need to transfer the shared resources to another Op Center. If successful, the GUI will call each installable module's loggedOut() method, then it will close all windows and clear the user's preferences.

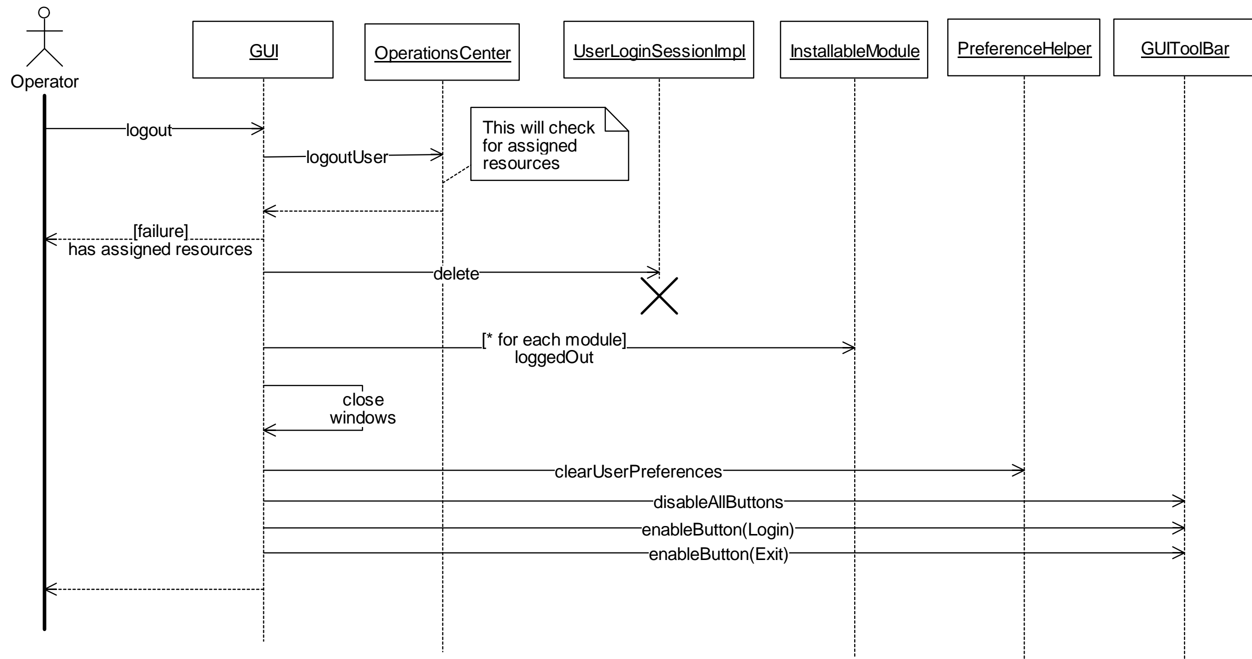


Figure 5-6. GUI:LogoutBasic (Sequence Diagram)

5.7 GUI:MakeMenuMultipleSelect (Sequence Diagram)

This diagram shows how a menu is created when two or more GUI wrapper objects are selected. The GUI's makeMenu method determines that there are multiple objects selected, and it creates a BucketSet that it will use to count the menu items. Then it asks each selected object to supply the multiple-selection menu items. The menu item strings are put into the BucketSet and then retrieved. The only strings that are retrieved from the BucketSet are those which have the same number of instances as there are selected objects. The GUI then creates menu items for the strings and attaches all selected objects as ActionListeners to each representative menu item.

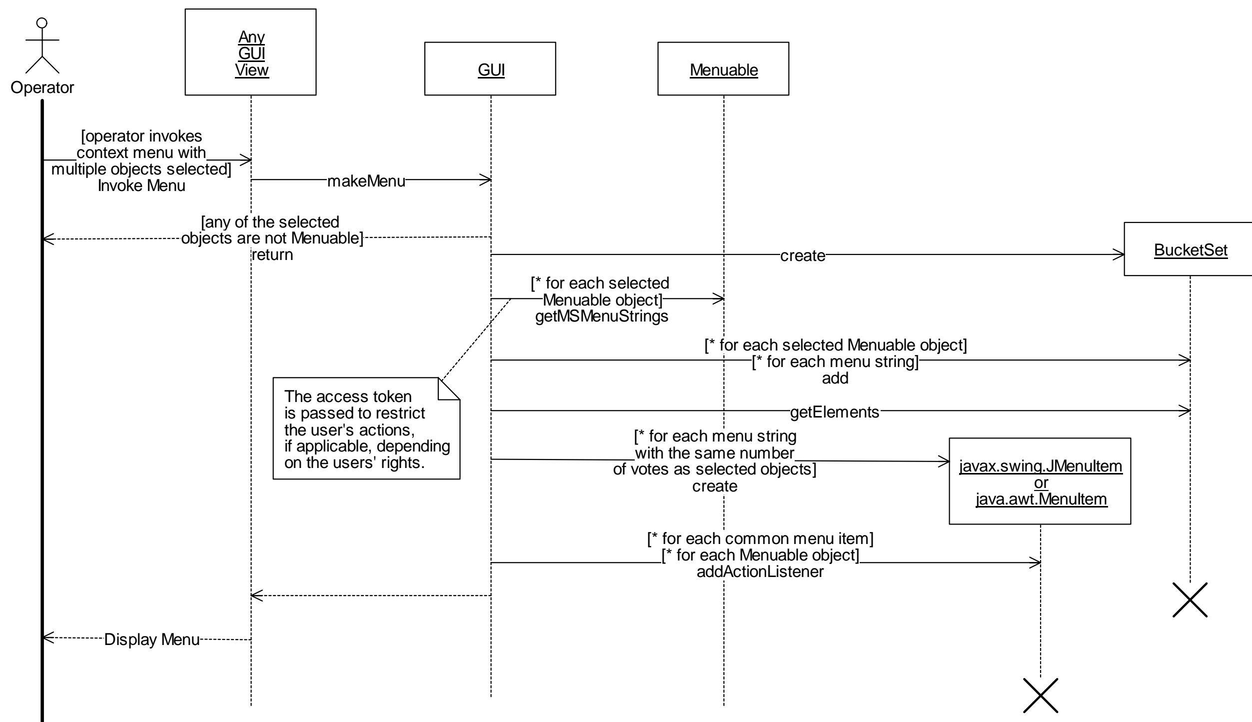


Figure 5-7. GUI:MakeMenuMultipleSelect (Sequence Diagram)

5.8 GUI:MakeMenuNoneSelected (Sequence Diagram)

This diagram shows how a menu is created when no GUI wrapper objects are selected. The GUI's makeMenu method determines that there are no objects selected, and the GUI then adds its own global menu items and calls each module to get their menu item strings. The GUI, and each installable module that supplies menu item strings, are then attached as ActionListeners to the menu items so that they will be called when the user clicks on the menu items.

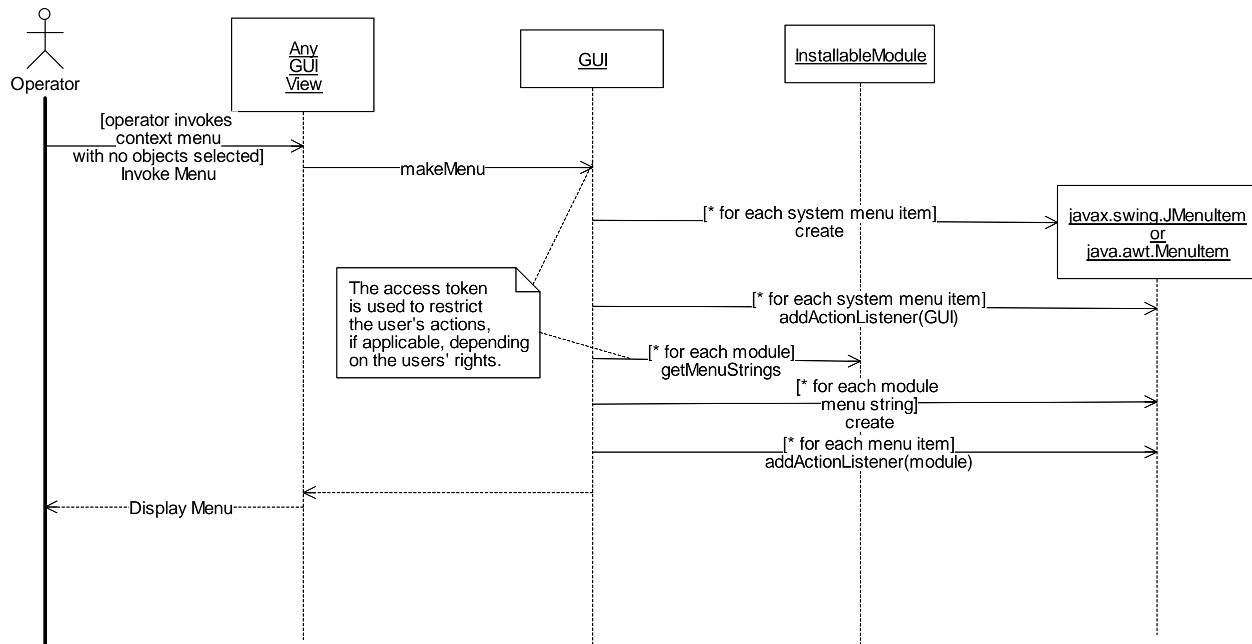


Figure 5-8. GUI:MakeMenuNoneSelected (Sequence Diagram)

5.9 GUI:MakeMenuSingleSelect (Sequence Diagram)

This diagram shows how a menu is created when exactly one GUI wrapper object is selected. The GUI's makeMenu method determines that there is one object selected, and it asks the Menuable object for the single-select menu item strings. The GUI will then create all of the menu items and attach the Menuable object as an ActionListener to each of the menu items.

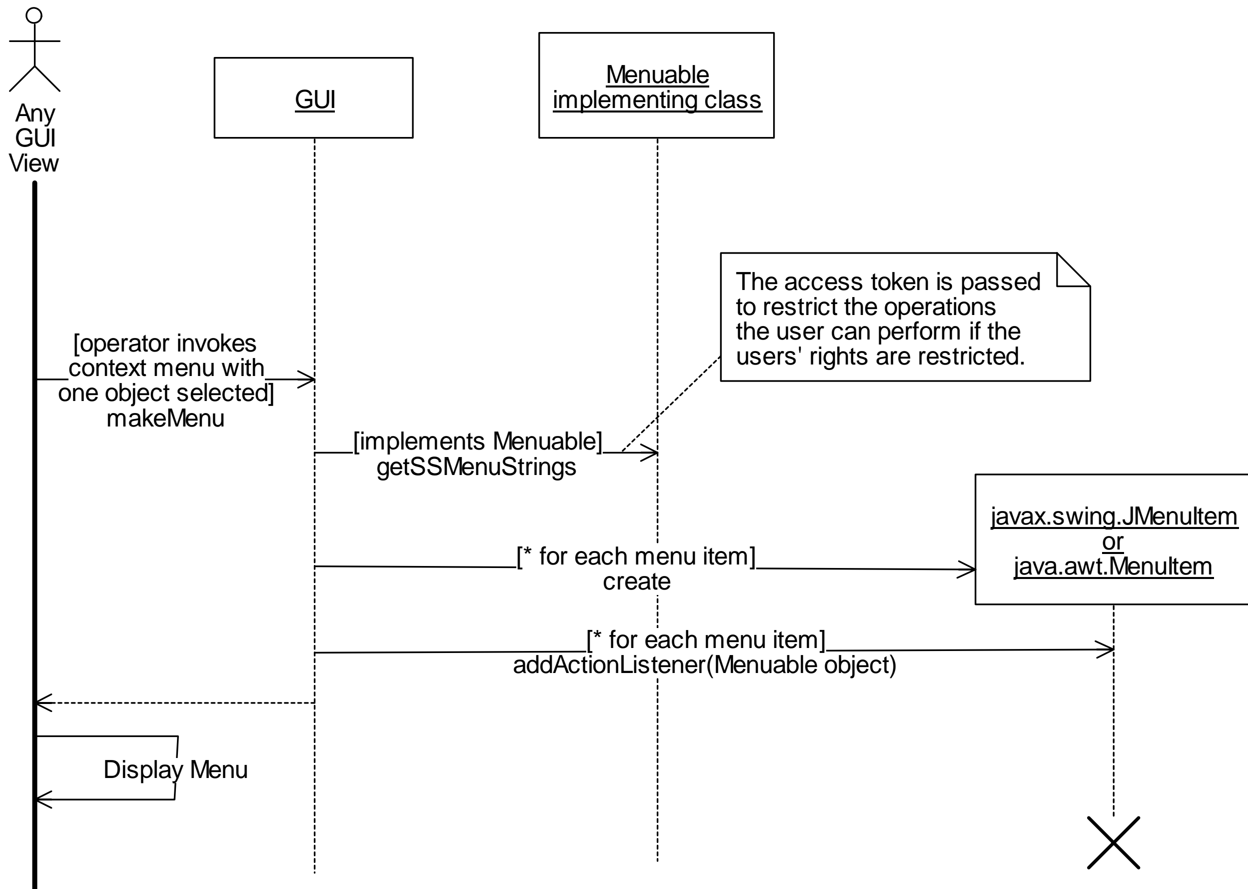


Figure 5-9. GUI:MakeMenuSingleSelect (Sequence Diagram)

5.10 GUI:ShutdownBasic (Sequence Diagram)

This diagram shows steps necessary for a shutdown. The operator either closes the GUIToolBar or clicks on the Exit button. Either of these actions will result in the GUI's shutdown method being called. The GUI will then call each module's shutdown() method. The GUI will then call deactivate_impl() to shut down the ORB.

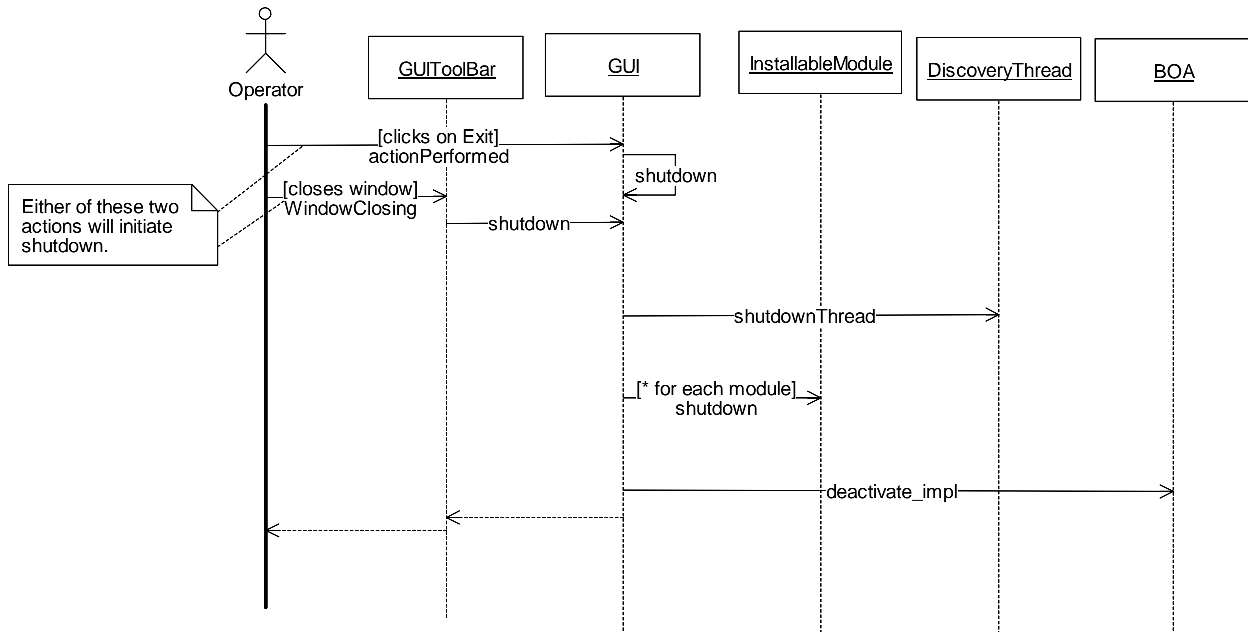


Figure 5-10. GUI:ShutdownBasic (Sequence Diagram)

5.11 GUI:StartupBasic (Sequence Diagram)

This diagram shows the sequence at startup. The GUI will find the names of the installable module classes and will use the Java class loader to instantiate them. Then it will call each module to register its system preferences and then it will read them in. Then the GUI will call each module's startup, at which time the modules will have a chance to query the system preferences that were read in. The GUI will create the GUIToolBar and add the general GUI buttons to it. The installable modules will also add their buttons in their startup methods. Then, the GUI will disable all of the buttons (except for Login and Exit) before showing the toolbar. The GUI will then create an EventConsumerGroup to monitor the health of all event channels. Then it will create a DiscoveryThread that will periodically look for event channels and objects. See the GUI:DiscoveryBasic sequence diagram for details on event channel and object discovery.

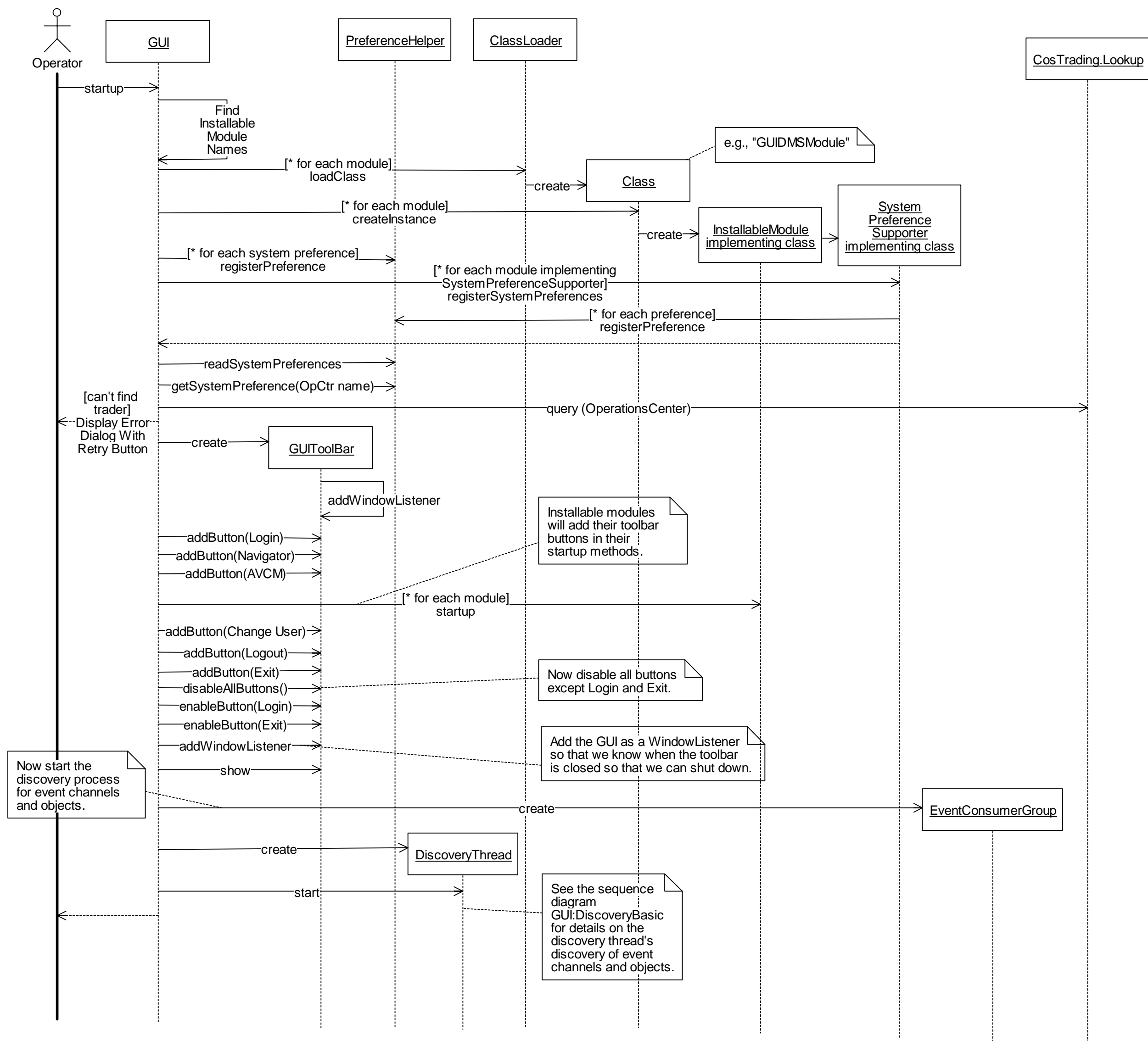


Figure 5-11. GUI:StartupBasic (Sequence Diagram)

5.12 GUI:EventUpdatePushedBasic (Sequence Diagram)

This diagram shows how updates to the served CORBA objects propagate to the GUI windows. The server will push the event data to the event service. The CORBA event service will then push the event data to the PushConsumer (which would typically be the GUI or an InstallableModule). The event data must contain some identification data so that the GUI wrapper object can be looked up in the DataModel. After the PushConsumer retrieves the GUI wrapper object from the DataModel, it will update any relevant data within the object and will call the DataModel one or more times with update hints to indicate what part of the object's data changed. The DataModel will accumulate all of the update hints for some short time period until it distributes them to all of the attached ModelObservers (which would typically be windows displaying the object data).

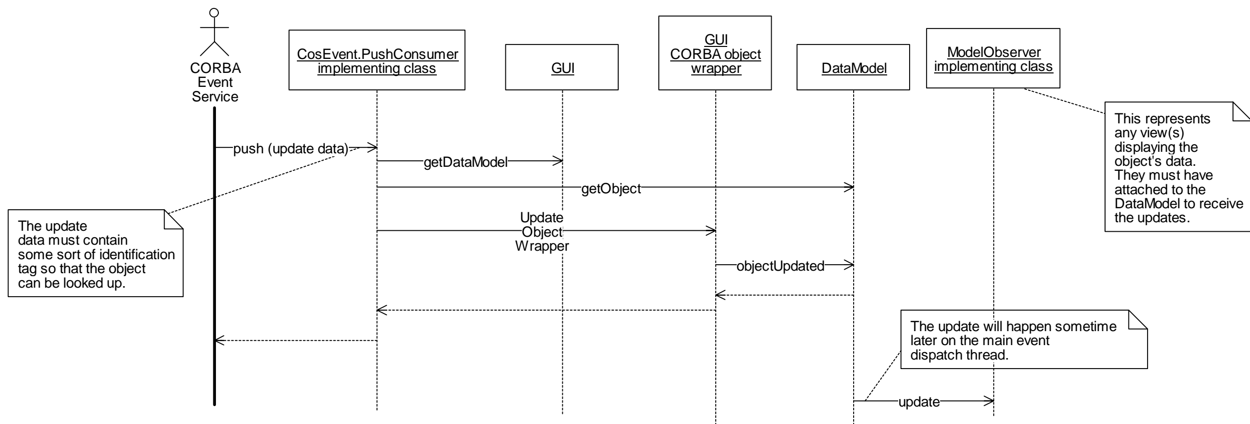


Figure 5-12. GUI:EventUpdatePushedBasic (Sequence Diagram)

5.13 GUI:SystemCommandBasic (Sequence Diagram)

This diagram shows how a system command is handled. A system command is one which does not apply to any served CORBA objects. (For those commands, see the GUI:CommandObjectBasic diagram.) First, a context menu is invoked by the user when there are no objects selected (see the GUI:MakeMenuNoneSelected for details on how the menu is made). The GUI, or an InstallableModule, will be attached to the menu items as an ActionListener when the menu is built. When the user clicks on the menu item, Java will invoke the actionPerformed() method of the ActionListener implementing class, which will allow the ActionListener to execute the command.

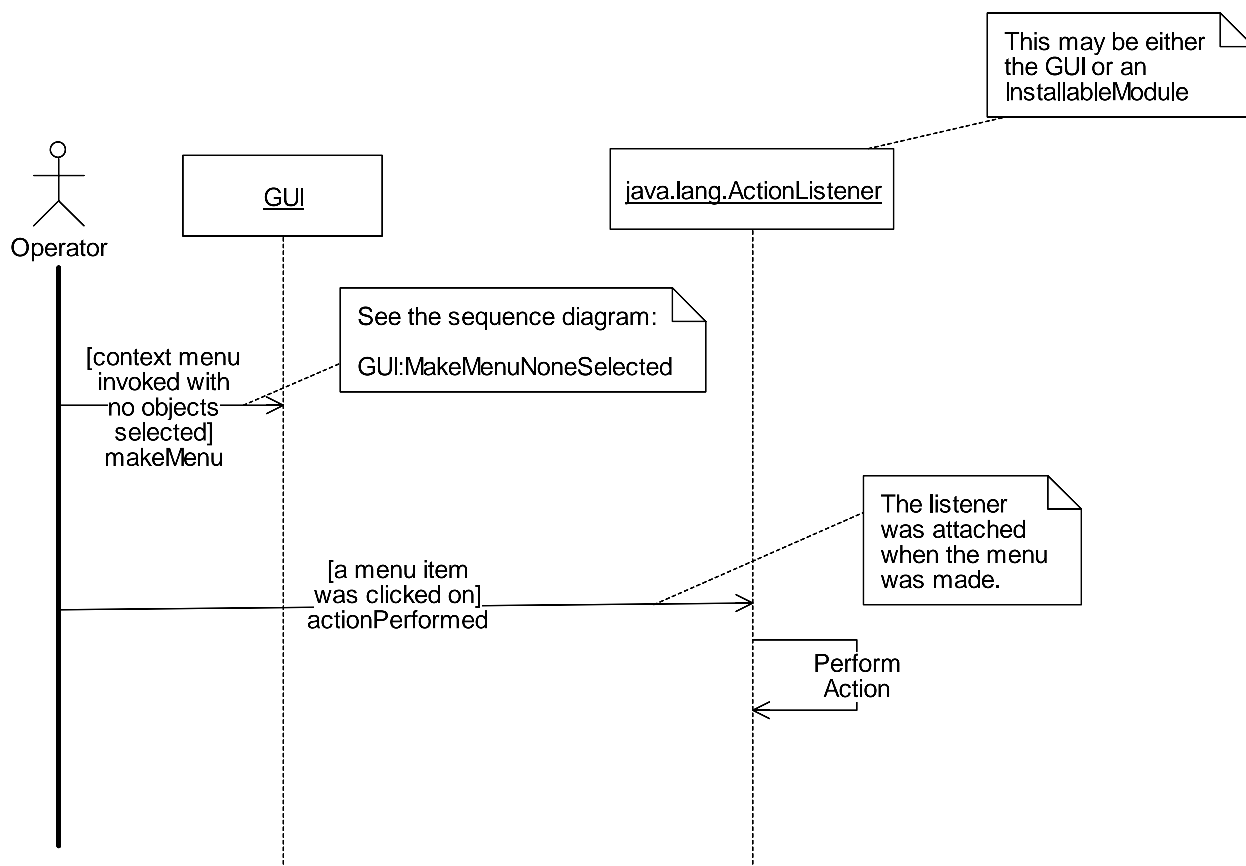


Figure 5-13. GUI:SystemCommandBasic (Sequence Diagram)

5.14 DataModel:AttachObserver (Sequence Diagram)

This diagram shows how an observer is attached to the DataModel for the purpose of receiving updates. The DataModel's attachObserver method is called, and if the priority level is supported by the DataModel, the observer will be attached at that priority level. The result of this is that the observer will be updated periodically (with the period depending on the priority level) after changes are made to the objects through the DataModel.

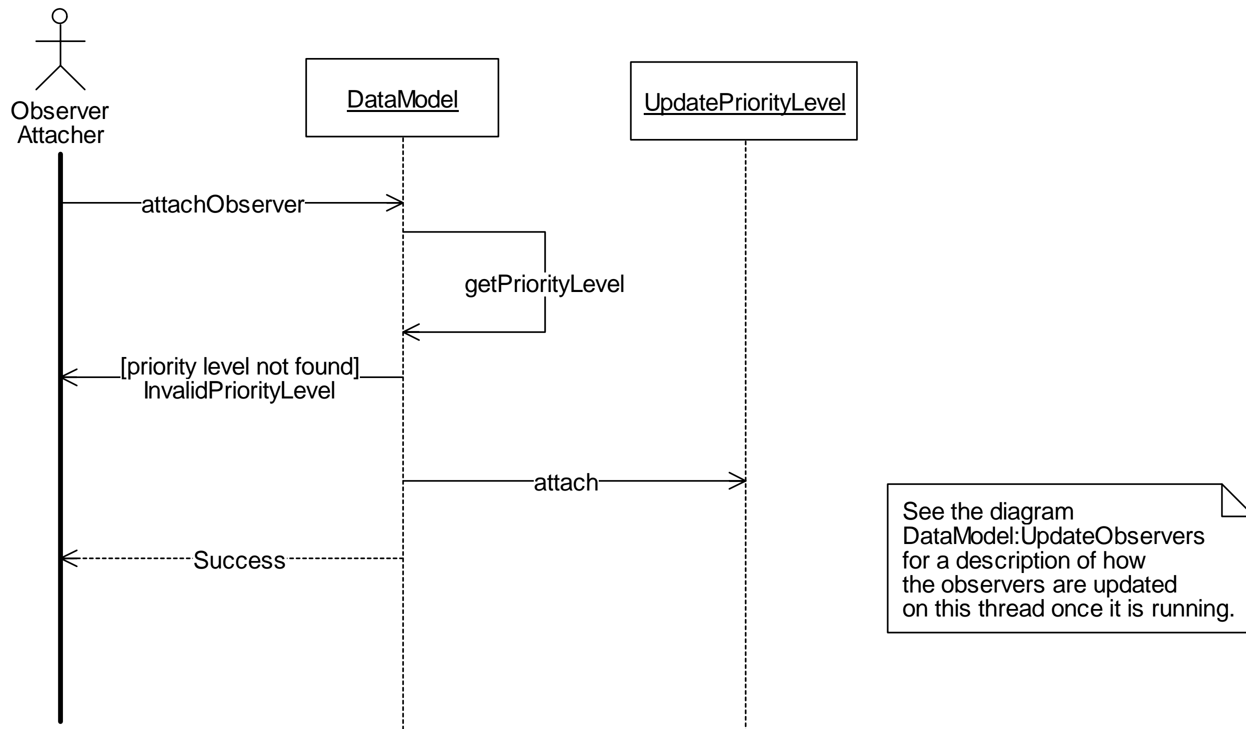


Figure 5-14. DataModel:AttachObserver (Sequence Diagram)

5.15 DataModel:ObjectAdded_ (Sequence Diagram)

This diagram shows the steps taken when an object is added to the DataModel. First, the Object and the Key are passed into the DataModel's objectAdded method. The DataModel checks whether the object was added before and if so, the object will not be added again. The DataModel then calls each of the PriorityLevel objects' objectAdded methods so that observers of all priority levels can be updated independently. The PriorityLevel object then checks its ChangeCollection objects to see if a ChangeCollection exists for the class of object which is being added. If not, it will create a ChangeCollection to store all changes for that class. The PriorityLevel then creates an ObjectAdded object to represent the change, then adds it to the ChangeCollection.

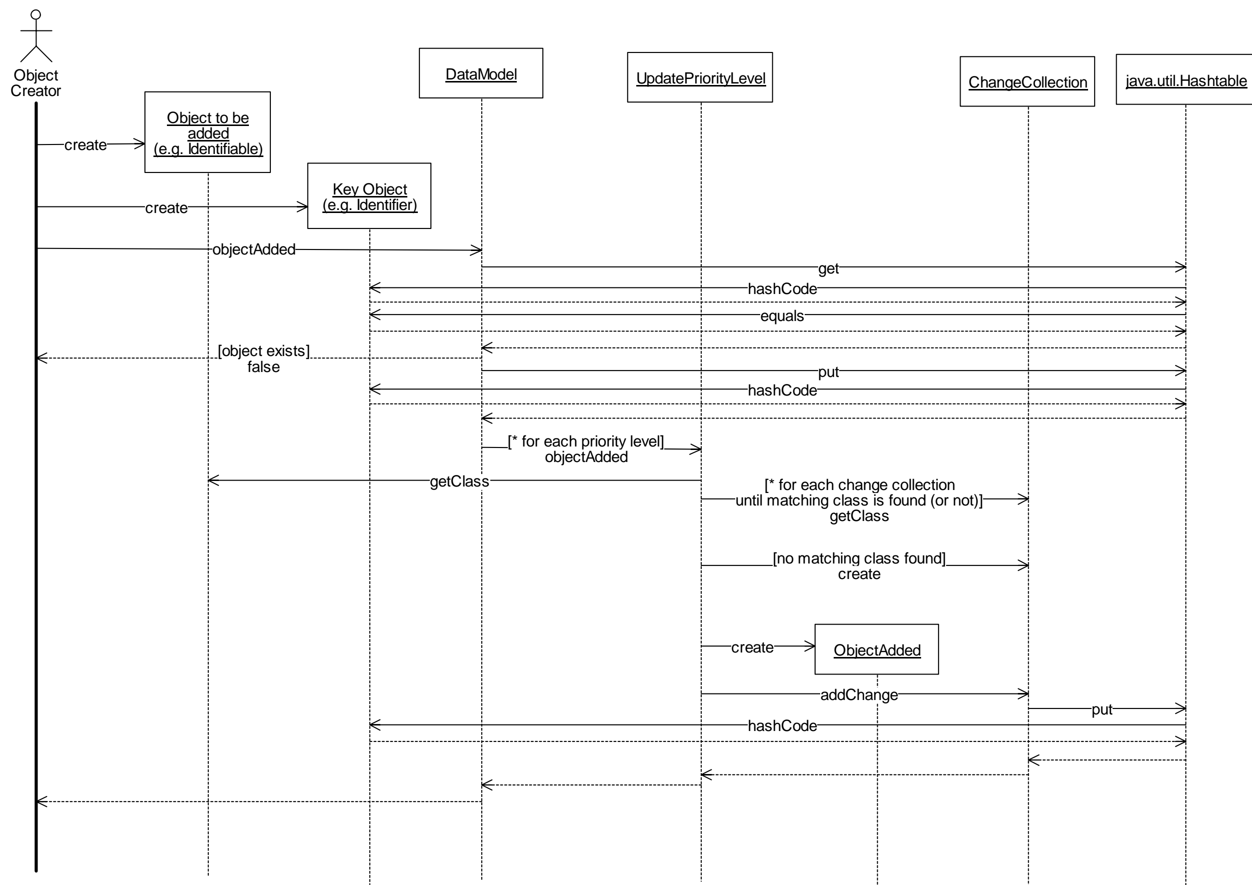


Figure 5-15. DataModel:ObjectAdded_ (Sequence Diagram)

5.16 DataModel:ObjectRemoved (Sequence Diagram)

This diagram shows what happens when an object is removed from the DataModel. The Key object is passed into the DataModel's objectRemoved method, which removes the stored object in the DataModel. If the object was removed (i.e., if it was found), the DataModel then calls the objectRemoved method for each UpdatePriorityLevel so that each priority level of observers will be updated independently. The UpdatePriorityLevel will check to see if it has a ChangeCollection to store changes for the class of the object. It will create a new ChangeCollection if necessary. The UpdatePriorityLevel will then create an ObjectRemoved object to represent the change. This object will be added to the ChangeCollection for the object's class. Java's garbage collection ensures that the object will not actually be deleted until the last reference to the object is removed; therefore, since object references are stored in the ChangeCollection objects, each object will exist at least until the last observer is updated on the lowest priority level. Observers have the responsibility to remove all of their references to the objects when their update method is called; otherwise, memory leaks will occur.

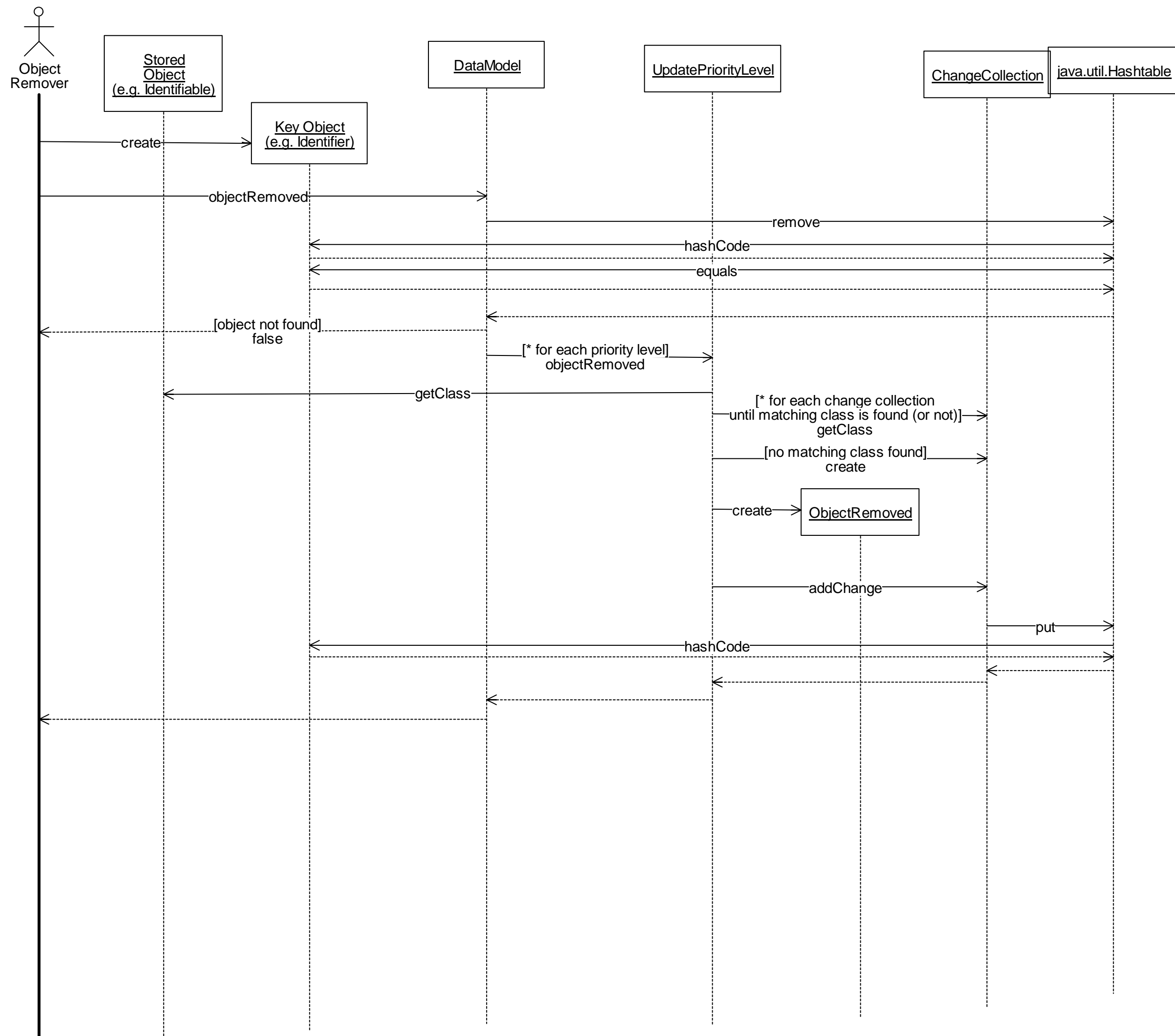


Figure 5-16. DataModel:ObjectRemoved (Sequence Diagram)

5.17 DataModel:ObjectUpdated (Sequence Diagram)

This diagram shows what happens when an object is updated through the DataModel. The caller passes in the Key object and an optional UpdateHint object. If an object is found with the Key, the DataModel will then call each UpdatePriorityLevel's objectUpdated method so that each priority level will be updated independently. The UpdatePriorityLevel checks to see if a ChangeCollection exists for the class of object that is being changed, and a ChangeCollection will be created if necessary. If theree is a previous change for the object and the existing change is ObjectRemoved or ObjectAdded, the update will be ignored. Otherwise, the update hint will be combined with the existing update hints (if any) so that the resulting hints are a union of all hints which have been accumulated. The changes will be distributed to the observers when the next period expires for the UpdatePriorityLevel.

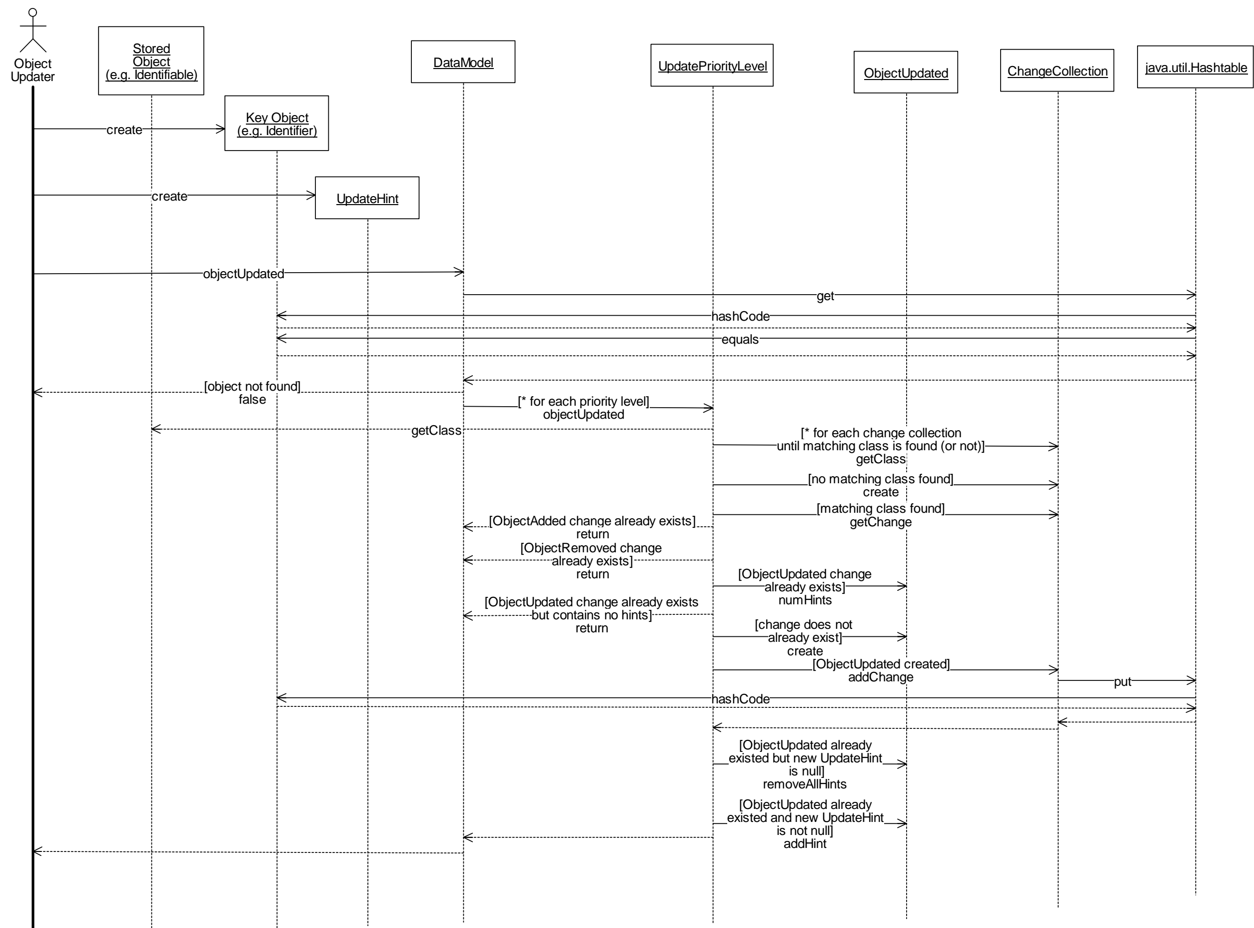


Figure 5-17. DataModel:ObjectUpdated (Sequence Diagram)

5.18 DataModel:UpdateObservers (Sequence Diagram)

This diagram shows how the observers are updated after changes have occurred to objects through the DataModel. The UpdatePriorityLevel thread decides that it's time to update the observers because the period has run out. It adds all of the changes that have been accumulated in the ChangeCollections and stores them in a ModelChange object. Then it distributes the ModelChange to all observers. If the observer is not a GUIObserver, it is updated on the UpdatePriorityLevel thread. However, GUIObservers must be updated on the main event thread, so the SwingUtility's invokeLater method is called to execute the update on the main event thread. After all observers are updated, the ChangeCollections are deleted to flush them. The UpdatePriorityLevel will then sleep until the next scheduled update.

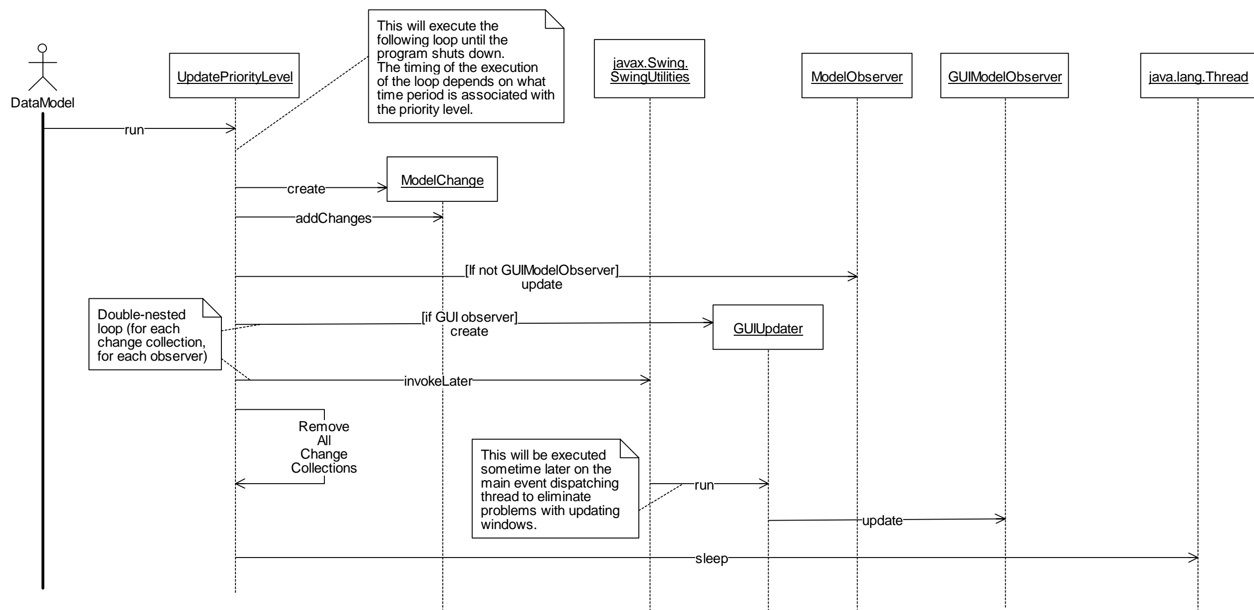


Figure 5-18. DataModel:UpdateObservers (Sequence Diagram)

5.19 Navigator:AddNavigables (Sequence Diagram)

This diagram shows what happens in the Navigator when Navigable objects are added. First, the Navigables are passed to the NavTree. The NavTree will then build a list of any NavTreeDisplayables to add. For each element in the list, it checks the hash table to determine whether the parent (if any) is already in the tree. If the parent is in the tree or there is no parent, a new MutableTreeNode will be created and inserted into the DefaultTreeModel, and the NavTreeDisplayable will be put into the hash table. Each NavTreeDisplayable that is added to the tree is removed from the list to be inserted. As long as one or more nodes were inserted during a given pass through the list, another pass is attempted (for the next level of the tree). Then the Navigables are added to the NavList. This will check each Navigable to see if it is a NavListDisplayable and if its parent is the selected NavTreeDisplayable. If both are true, the NavListDisplayable will be added to the list.

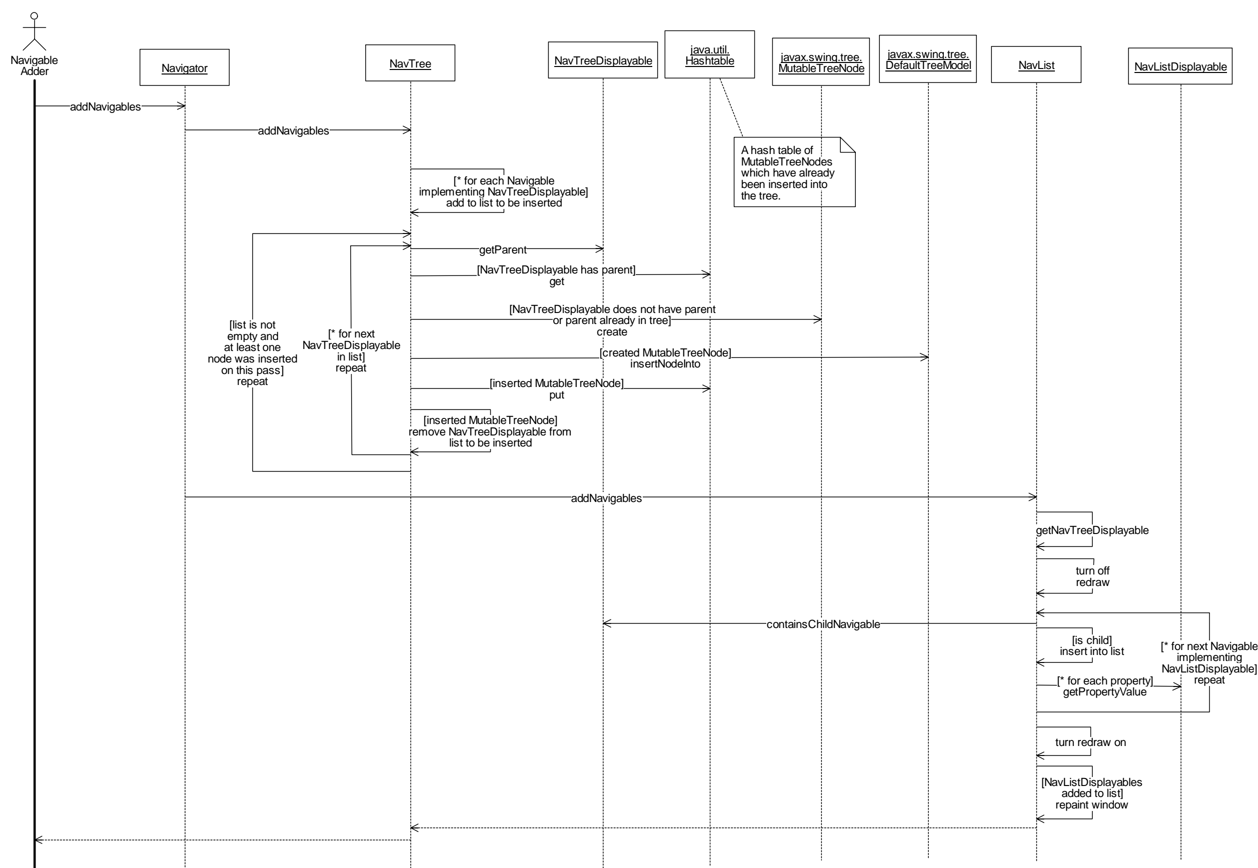


Figure 5-19. Navigator:AddNavigables (Sequence Diagram)

5.20 Navigator:Initialize (Sequence Diagram)

This diagram shows how the Navigator is initialized. The openNavigator method will create a new Navigator window and the tree and list views. The Navigator will then query the NavigatorSupporter to provide it with all Navigable objects. The Navigables are added to the NavTree (see the Navigator:AddNavigables diagram for details). Then the root node is set as the selected node in the NavTree. See the Navigator:TreeSelectionChange sequence diagram for details on the effects of this.

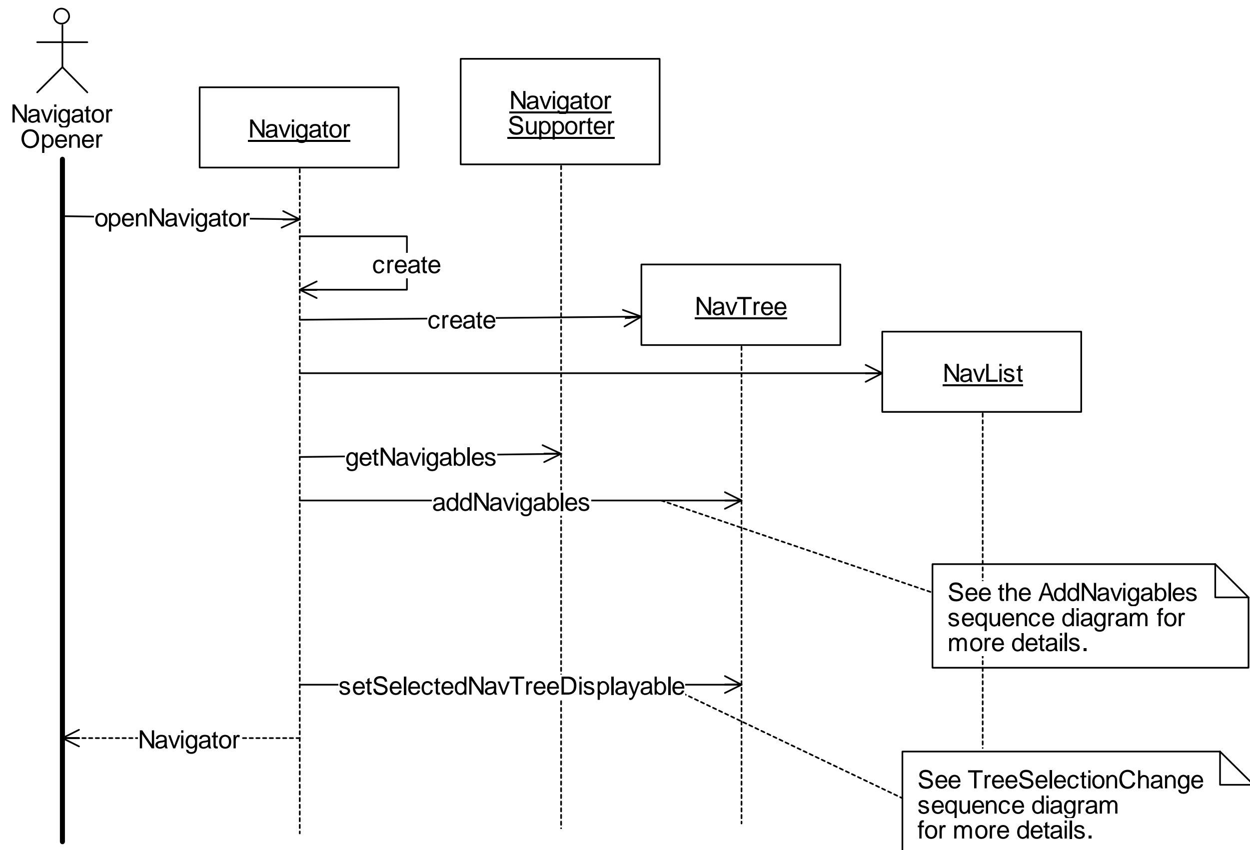


Figure 5-20. Navigator:Initialize (Sequence Diagram)

5.21 Navigator:RemoveNavigables (Sequence Diagram)

This diagram shows how Navigables are removed from the Navigator. Each NavTreeDisplayable to removed causes removeTreeNode to be called. This is a recursive call, which calls removeTreeNode first on each of its children. The children are removed first so that every tree node below the current node is cleaned out of the hash table. If the NavList is displaying the children of the node that is being destroyed, then we set the NavTreeDisplayable in the list to the parent. Then the NavTreeDisplayable is removed from the hash table and also from its parent. The Navigables to be removed are then passed to the NavList, which removes and NavListDisplayables in the list matching any of the Navigables to be removed.

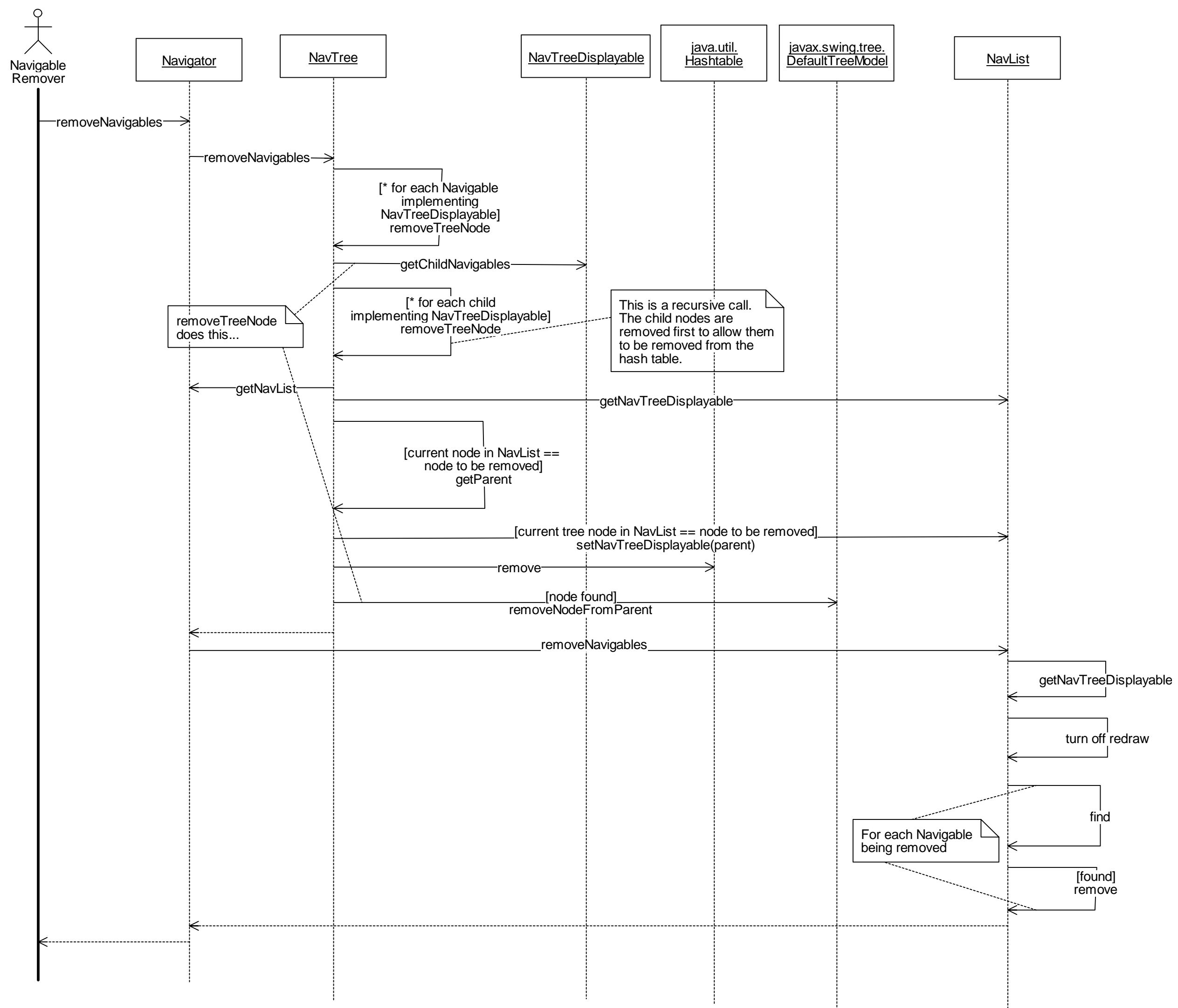


Figure 5-21. Navigator:RemoveNavigables (Sequence Diagram)

5.22 Navigator:TreeSelectionChange (Sequence Diagram)

This diagram shows what happens when a tree selection change takes place. The NavTree calls the NavList and sets the NavTreeDisplayable. This will cause all objects to be removed from the NavList. The NavList will ask the new NavTreeDisplayable for its properties (columns). Then the NavList will ask the NavTreeDisplayable for its children, which will all be inserted into the list. Each item inserted will be called for each column/property to supply the property value.

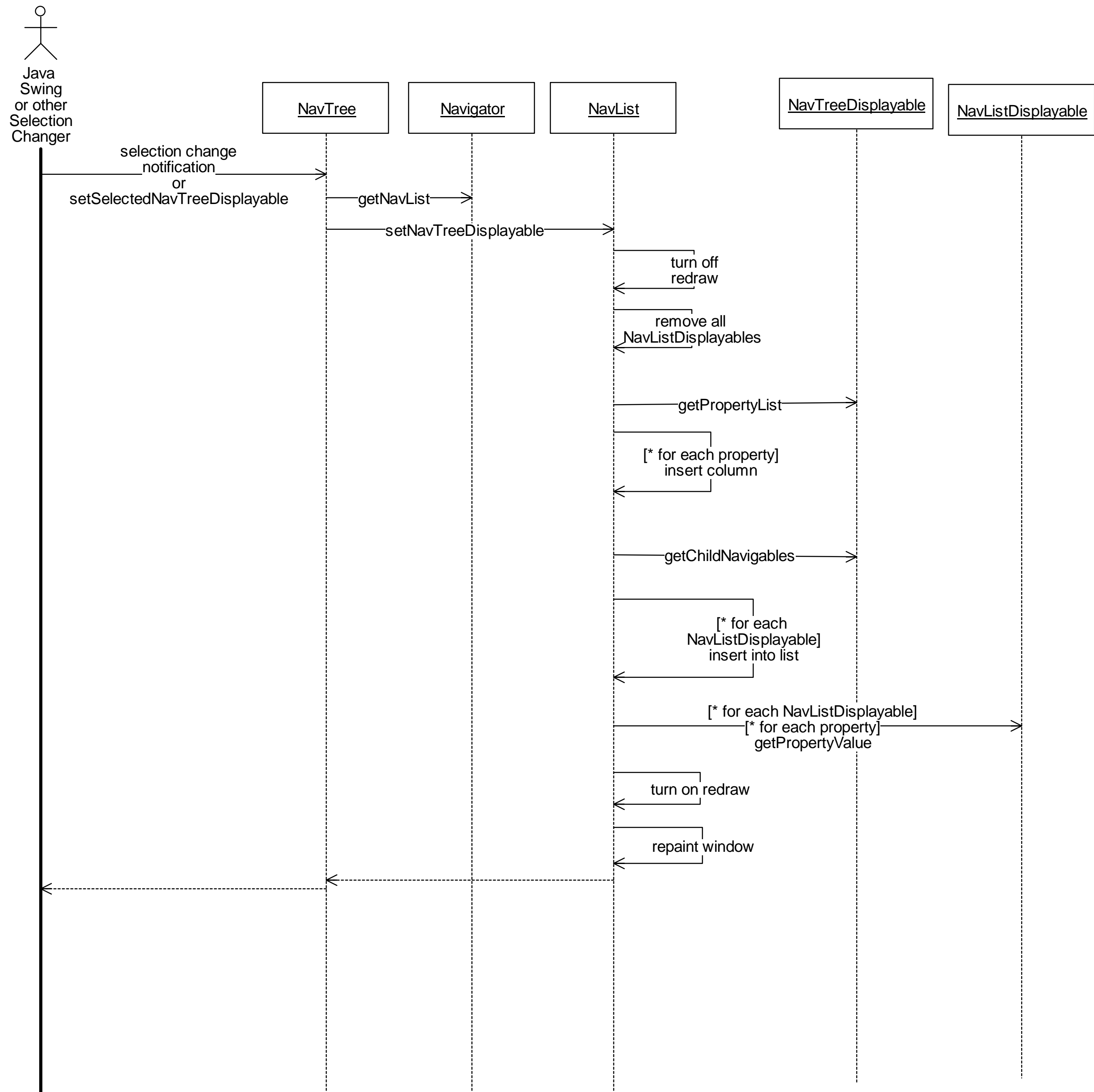


Figure 5-22. Navigator:TreeSelectionChange (Sequence Diagram)

5.23 GUIDMSModule:AddDMS (Sequence Diagram)

This sequence shows how an operator adds a new DMS to the system. The operator initiates this action by selecting Add DMS from the appropriate popup menu. If the user does not have the appropriate functional rights, this menu item will not be made available. The operator will then be shown a DMS properties dialog box with default configuration information which he/she may modify to alter the configuration of the DMS. When the user presses OK, the new DMS will be added to the system. If the DMS can't be added, the user will be shown a popup window describing the reason it could not be added. If the DMS is added successfully, a DMSAdded event will be pushed from the server through the DMS event channel and the new GUIDMS object will be added to the DataModel, which will update all windows after a short delay.

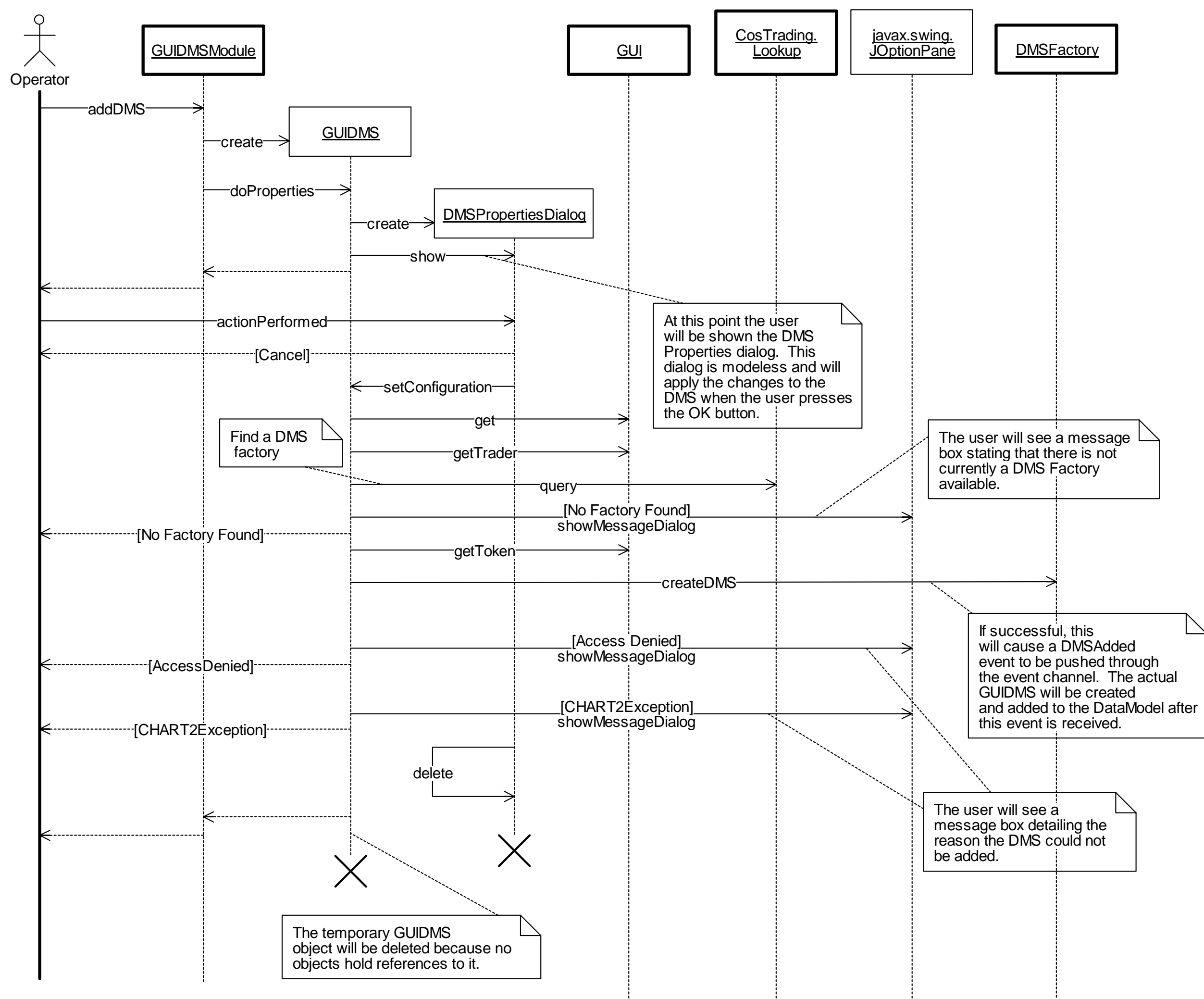


Figure 5-23. GUIDMSModule:AddDMS (Sequence Diagram)

5.24 GUIDMSModule:AddMessageToLibrary (Sequence Diagram)

This sequence shows how an operator adds a new stored message to a particular library. The operator initiates this action by right clicking on a library object and selecting "Add Message" from the popup menu. If the user does not have the appropriate functional rights, this menu item will not be made available. The operator will then be shown the DMSMessageEditor dialog box with default configuration information that he/she may modify to alter the name, category and message content of the stored message. When the user presses OK, the new message will be added to the library. If the message can't be added, the user will be shown a popup window describing the reason it could not be added. If the message is added successfully, an event will be pushed from the server through the the event service and the new GUIDMSStoredMessage object will be added to the DataModel, which will update all windows after a short delay.

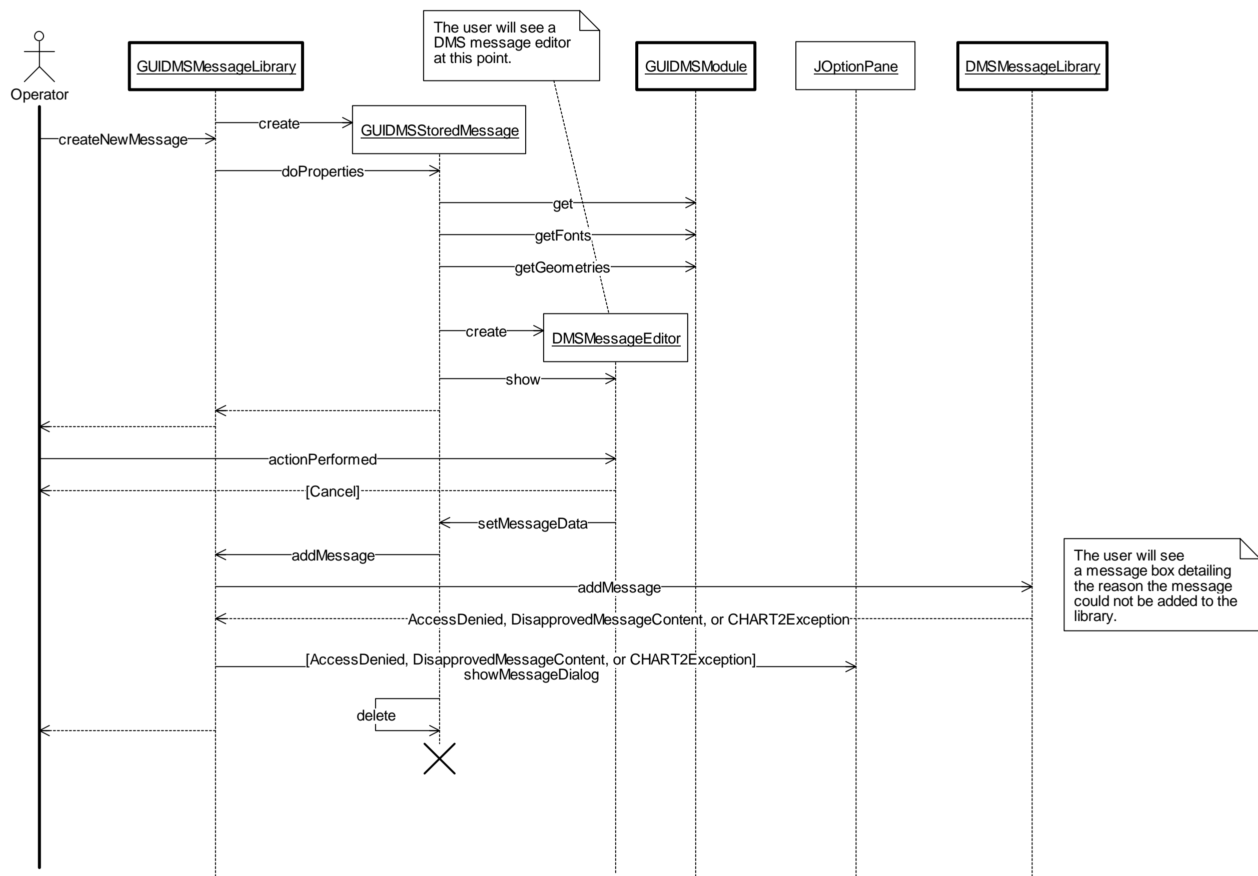


Figure 5-24. GUIDMSModule:AddMessageToLibrary (Sequence Diagram)

5.25 GUIDMSModule:CreateMessageLibrary (Sequence Diagram)

This sequence shows how an operator may create a new message library. The operator initiates this sequence by right clicking on the "DMS Message Libraries" object in the Navigator window and selecting the "Add Message Library" menu item. This item will not be available if the user does not have the correct functional rights. The user will be shown a DMSMessageLibraryProperties dialog that will allow him/her to name the library. The system will then attempt to create the new libary with the specified name. If the library is created successfully, it will be available for use. If an error is encountered, the user will be notified via a popup window that describes the error condition.

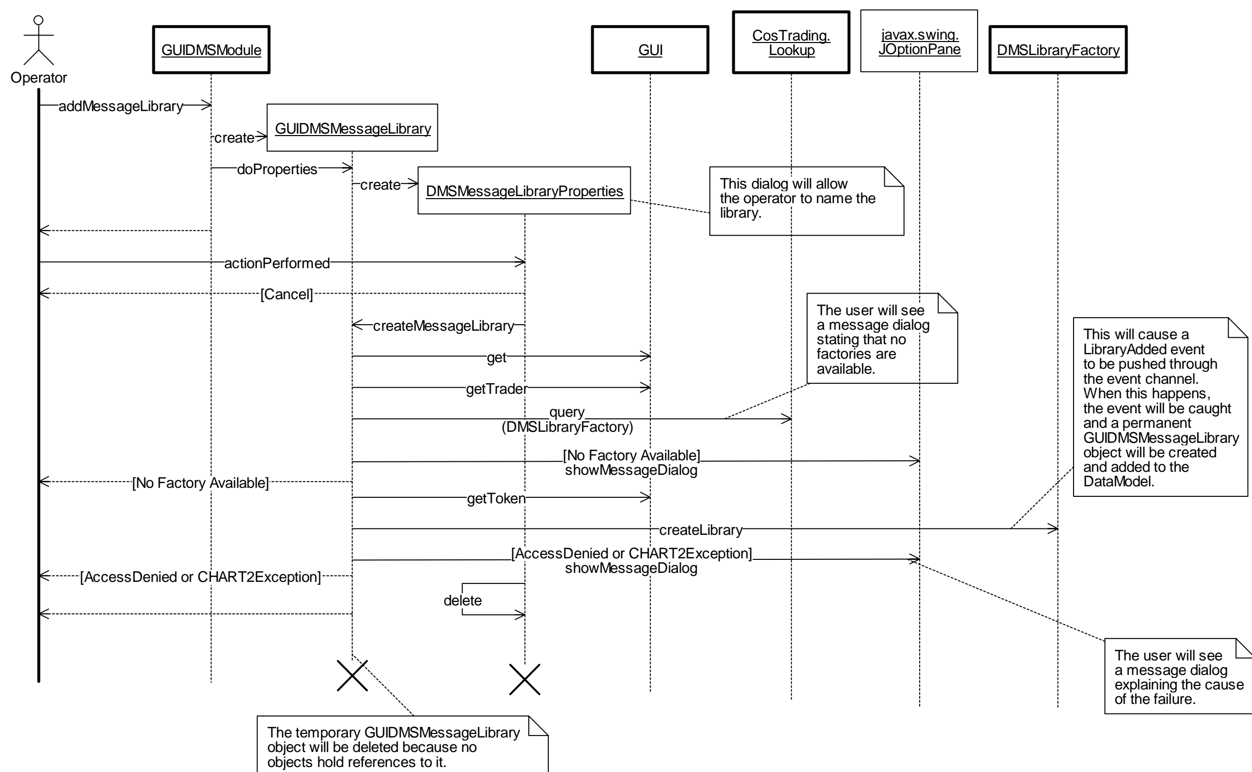


Figure 5-25. GUIDMSModule:CreateMessageLibrary (Sequence Diagram)

5.26 GUIDMSModule:DeleteMessageLibrary (Sequence Diagram)

This sequence shows how an operator may remove a DMS Message Library from the system. The operator initiates this action by right clicking on the message library in a window and selecting the "Remove" menu item from the popup menu. If any messages in this library are currently being used in a plan, the user will be warned and asked if he/she would like to continue. If the user continues the library will be removed from the system. If any errors are encountered the user will be notified via a popup window describing the error. This action will delete all messages in the library as well as the library itself from the system.

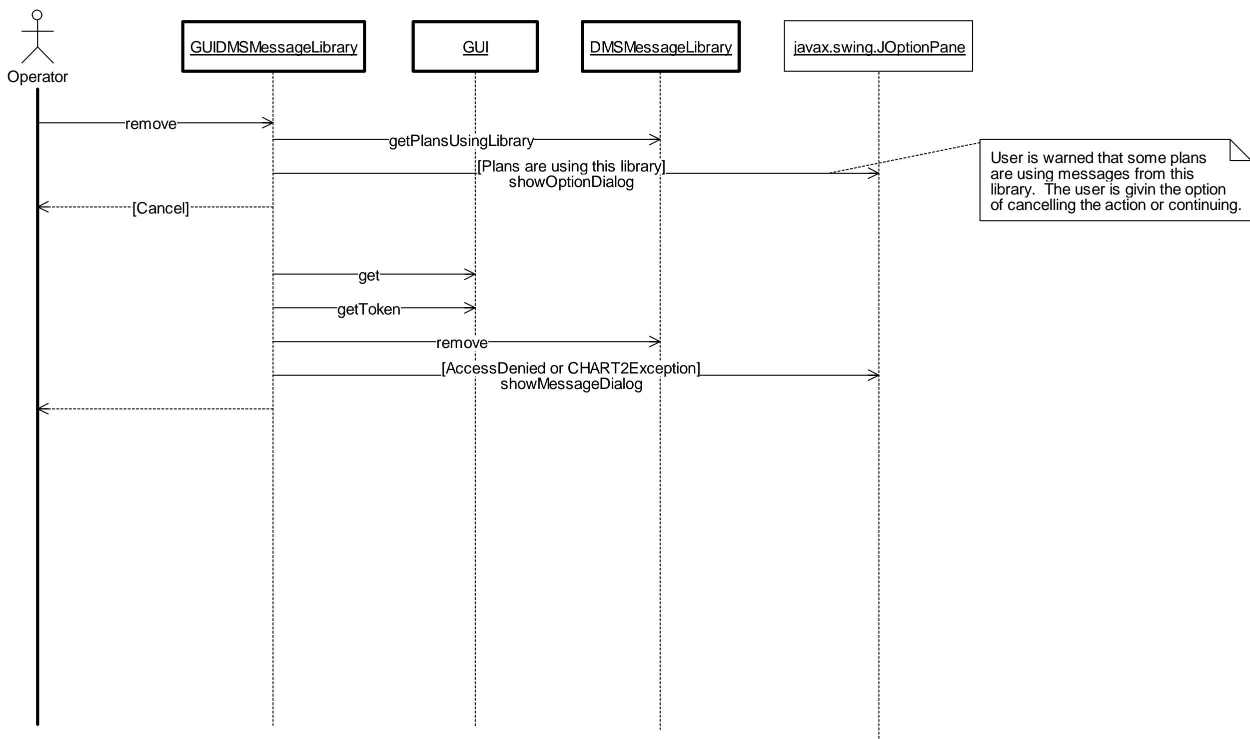


Figure 5-26. GUIDMSModule:DeleteMessageLibrary (Sequence Diagram)

5.27 GUIDMSModule:DeleteStoredMessage (Sequence Diagram)

This sequence shows how an operator may delete a stored DMS message from the system. The operator initiates this sequence by right clicking on a particular DMS message in a window and selecting the "Remove" menu item from the popup. The system will check if the message is currently being used by any plans. If it is, the user will be warned and will be given the option to cancel or continue. If the user continues, the message will be removed from the system. If any errors occur the user will be notified via a popup window which displays the details of the error.

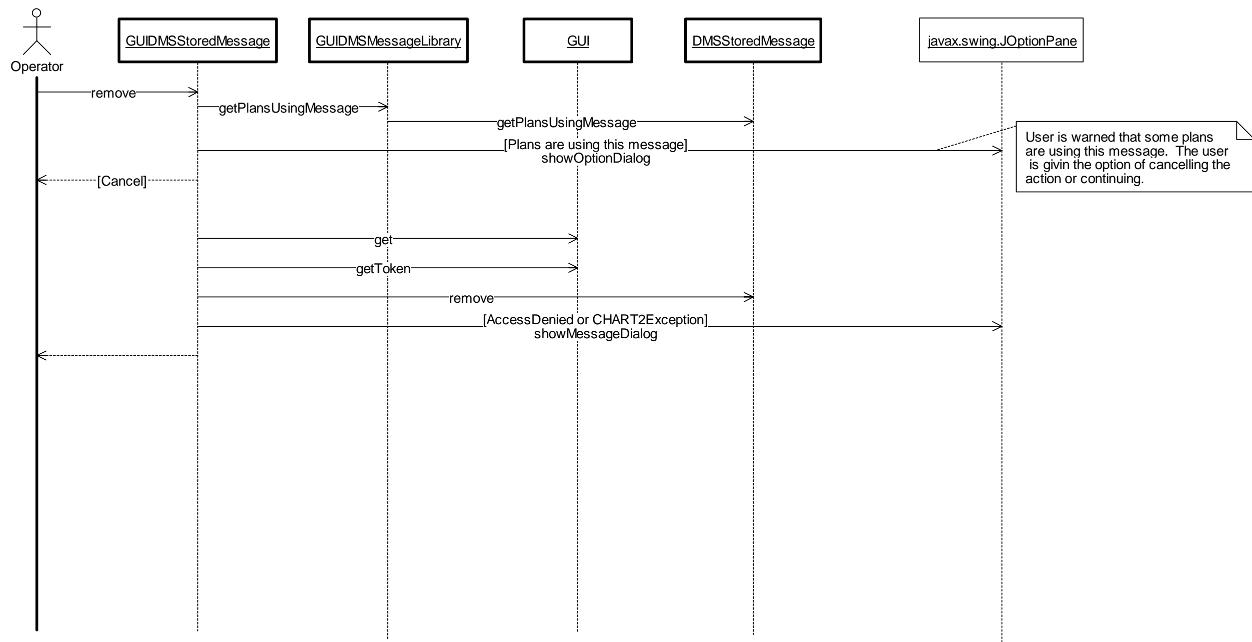


Figure 5-27. GUIDMSModule:DeleteStoredMessage (Sequence Diagram)

5.28 GUIDMSModule:Login (Sequence Diagram)

This sequence of events in initiated when a user logs in to the system using either the login or change user commands from the toolbar window. These commands cause the GUI:LoginBasic sequence or GUI:ChangeUserBasic to be performed. As part of either of these sequences, the GUI will call each of the installed modules giving them a chance to perform necessary operations to set up data specific to a particular user. The GUIDMSModule does not currently need to perform any processing when a user logs in.

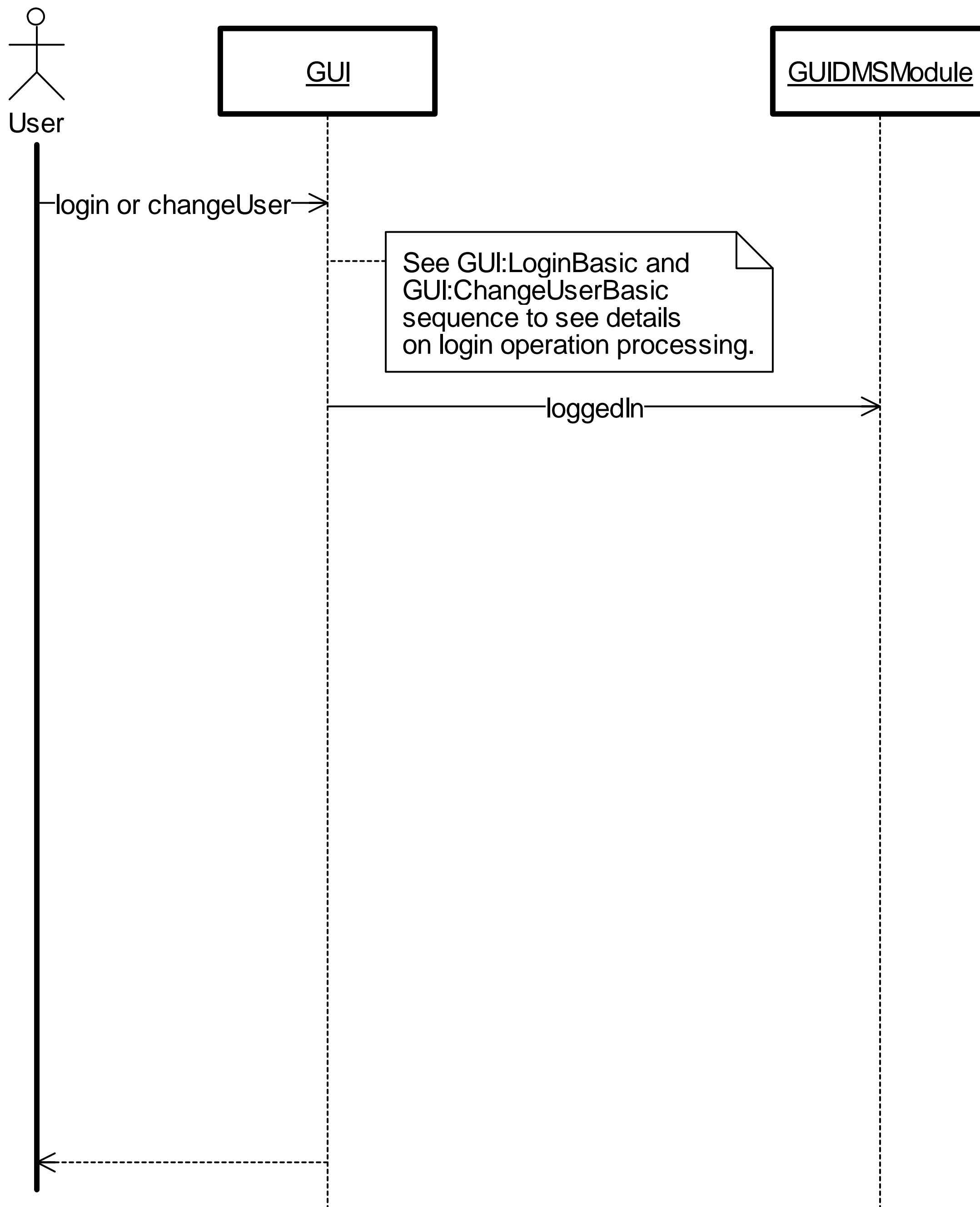


Figure 5-28. GUIDMSModule:Login (Sequence Diagram)

5.29 GUIDMSModule:Logout (Sequence Diagram)

This sequence of events in initiated when a user logs out of the system using either the logout or change user commands from the toolbar window. These commands cause the GUI:LogoutBasic or GUI ChangeUserBasic sequences to be performed. As part of these sequences, the GUI will call each of the installed modules giving them a chance to perform necessary operations to clean up data for a particular user. The GUIDMSModule does not currently need to perform any processing when a user logs out.

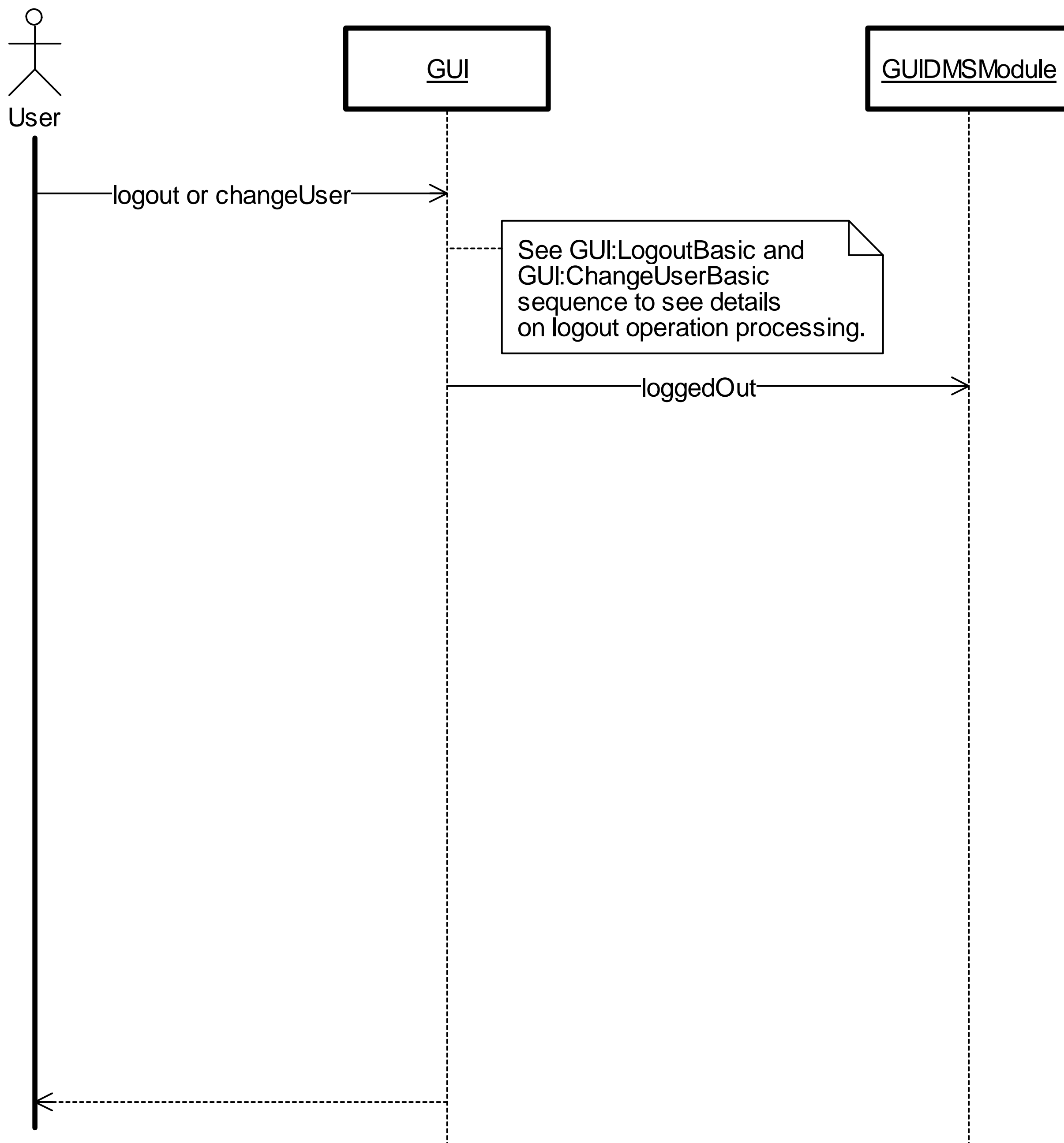


Figure 5-29. GUIDMSModule:Logout (Sequence Diagram)

5.30 GUIDMSModule:EditLibraryMessage (Sequence Diagram)

This sequence shows how an operator changes the name or message content of a stored DMS message. The operator initiates this action by right clicking on the stored message and selecting "Properties" from the popup menu. If the user does not have the appropriate functional rights, this menu item will not be made available. The operator will then be shown the DMSMessageEditor dialog box with the current message contents and names which he/she may modify. When the user presses OK, the message contents and name will be changed. If the message content or description could not be modified, the user will be shown a popup window describing the reason for the error. If the values are modified successfully, an event will be pushed from the server through the event service and the all windows will show the new information after a short delay.

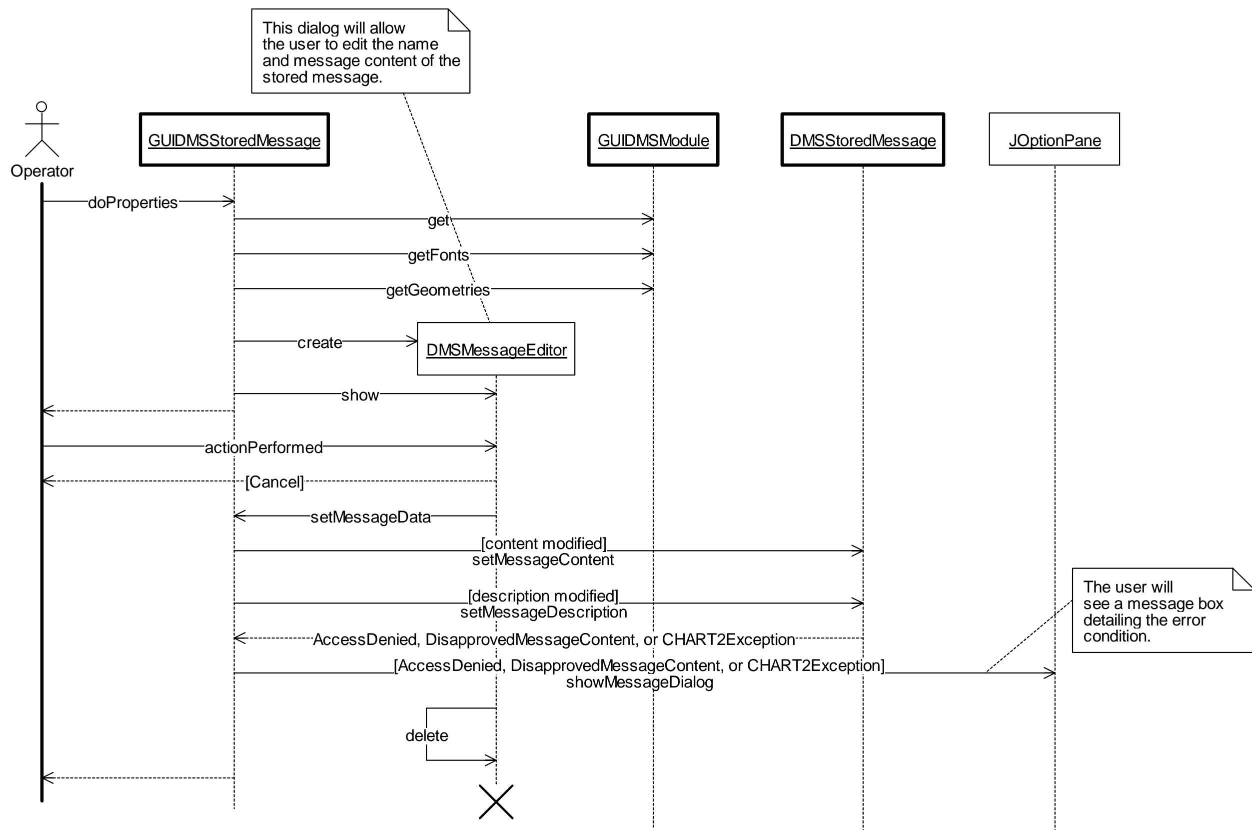


Figure 5-30. GUIDMSModule:EditLibraryMessage (Sequence Diagram)

5.31 GUIDMSModule:SetMessageLibraryProperties (Sequence Diagram)

This sequence shows how an operator may alter the properties of a DMSMessageLibrary that, for this release, will involve changing the name of the library. The user initiates this sequence by right clicking on a DMS message library object in a window and selecting the "Properties" menu item. The user will then be shown a DMSMessageLibraryProperties dialog that will be populated with the current name of the library. The operator may then type a new name for the library. When the user is done the library will be renamed. If any errors are encountered, the user will be notified via a popup window detailing the reason for the failure.

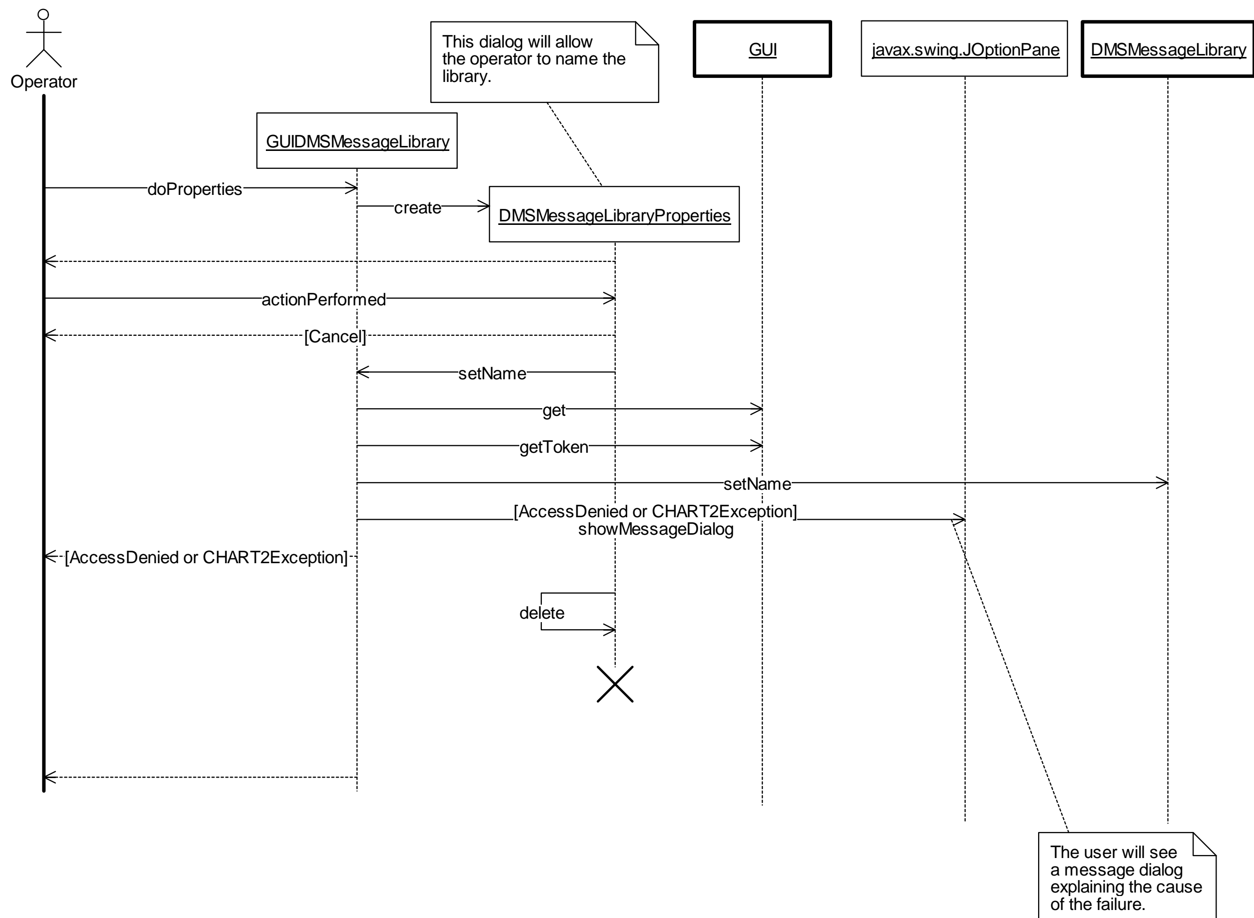


Figure 5-31. GUIDMSModule:SetMessageLibraryProperties (Sequence Diagram)

5.32 GUIDMSModule:BlankDMS (Sequence Diagram)

This sequence diagram shows how an operator blanks a particular DMS controller. The sequence is initiated when an operator right clicks on a DMS in the navigator and selects the Blank menu item. If the user does not have the appropriate functional rights, the DMS will not put the item in the menu when the menu is displayed. The access denied exception should never be encountered. It is handled here as a failsafe. Upon completion of this sequence the user will have a Command Status in the Command status view which will display the state of the operation as the server processes it. When the command completes and the DMS has been blanked, the navigator and other views will be updated with the new (blank) message information for the DMS.

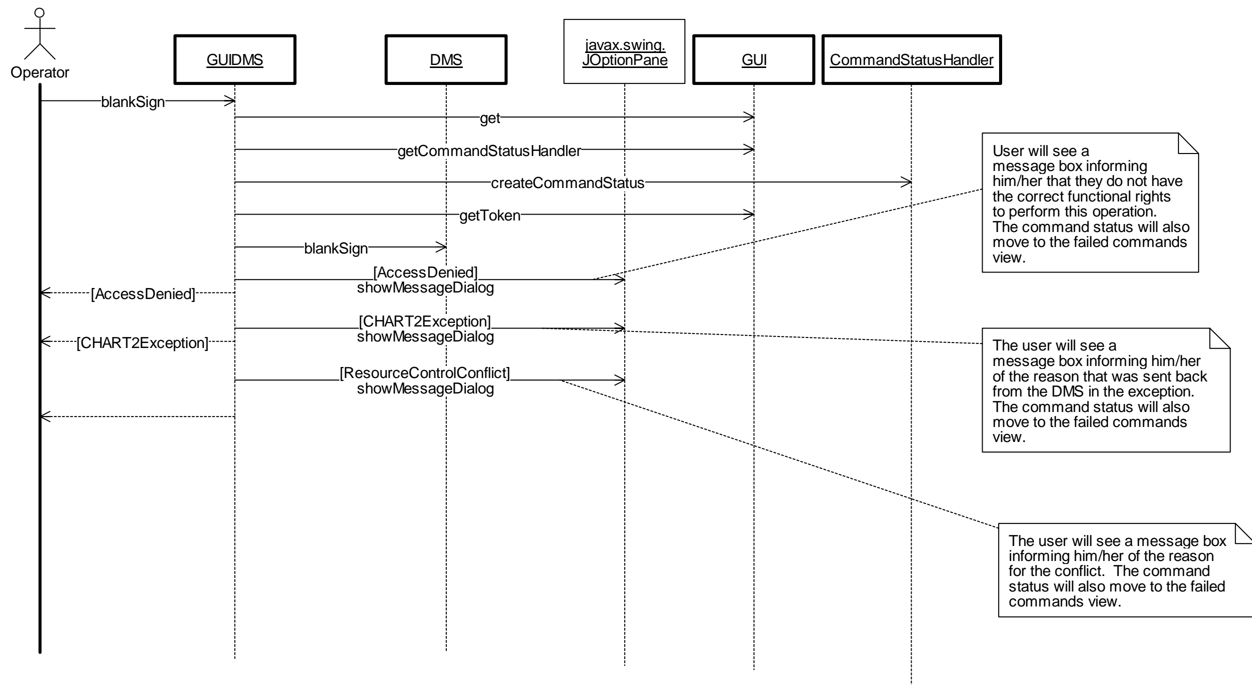


Figure 5-32. GUIDMSModule:BlankDMS (Sequence Diagram)

5.33 GUIDMSModule:CreateNewPlanItem (Sequence Diagram)

This sequence shows how an operator may add a new plan item to an already existing Plan. The operator initiates this sequence by right clicking on the Plan object in a window and selecting "Create DMS Stored Message Item" from the resulting popup. The operator will then be presented with the DMSStoredMsgItemProperties dialog that will allow him/her to associate a DMS with a library message. When the user is finished he/she will either see a popup window detailing an error condition which prevented the item from being added to the plan, or the item will be show as a part of the plan in the GUI windows.

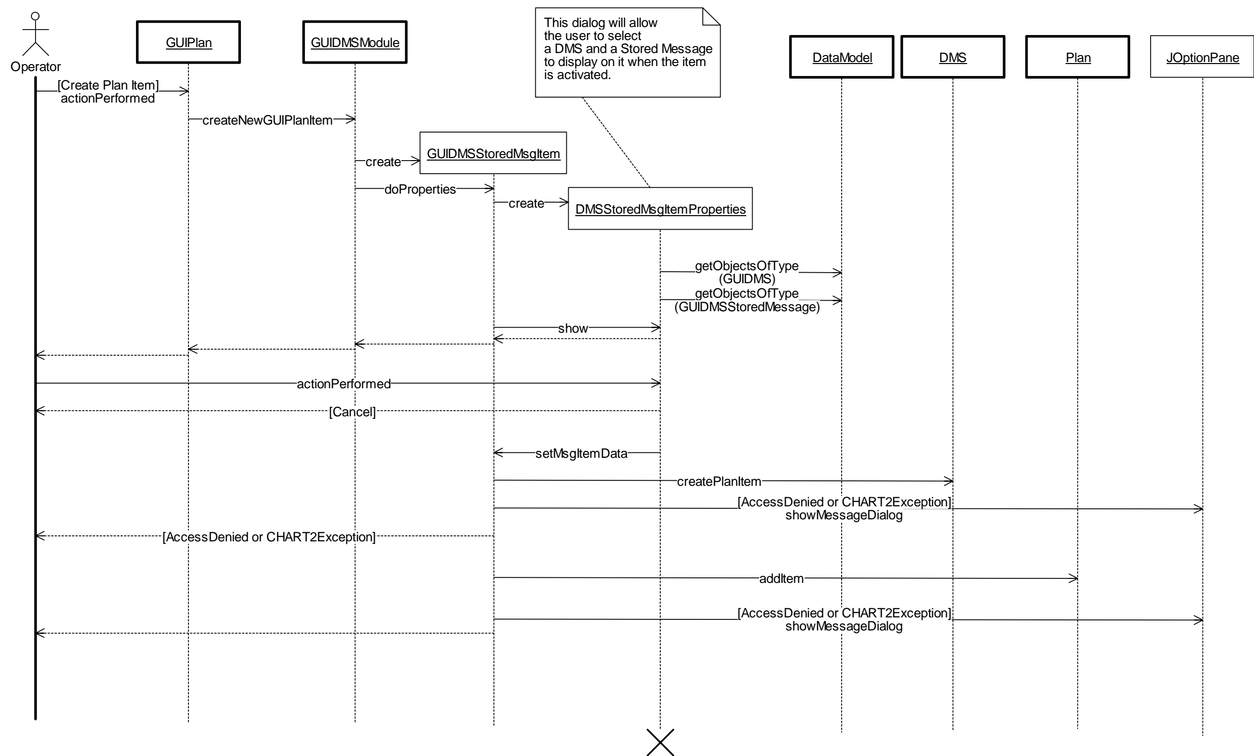


Figure 5-33. GUIDMSModule:CreateNewPlanItem (Sequence Diagram)

5.34 GUIDMSModule:DeleteDMS (Sequence Diagram)

This sequence diagram shows how an operator removes a DMS from the CHART II system. The sequence is initiated when an operator right clicks on a DMS in the Navigator and selects the delete menu item. If the user does not have the appropriate functional rights, the DMS will not put the item in the menu when the menu is displayed. The access denied exception should never be encountered. It is handled here as a failsafe. When this sequence completes the DMS will be removed from all views in the CHART II application and will no longer be available for use.

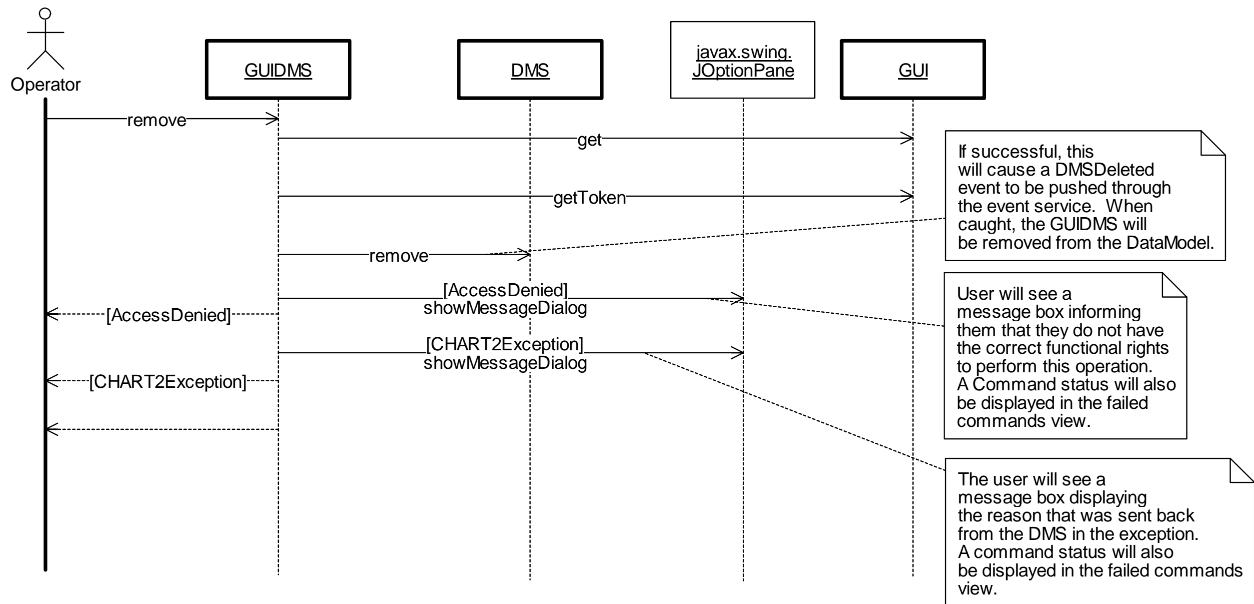


Figure 5-34. GUIDMSModule:DeleteDMS (Sequence Diagram)

5.35 GUIDMSModule:Shutdown (Sequence Diagram)

This sequence of events is initiated by a user closing the toolbar window of the CHART II GUI application.

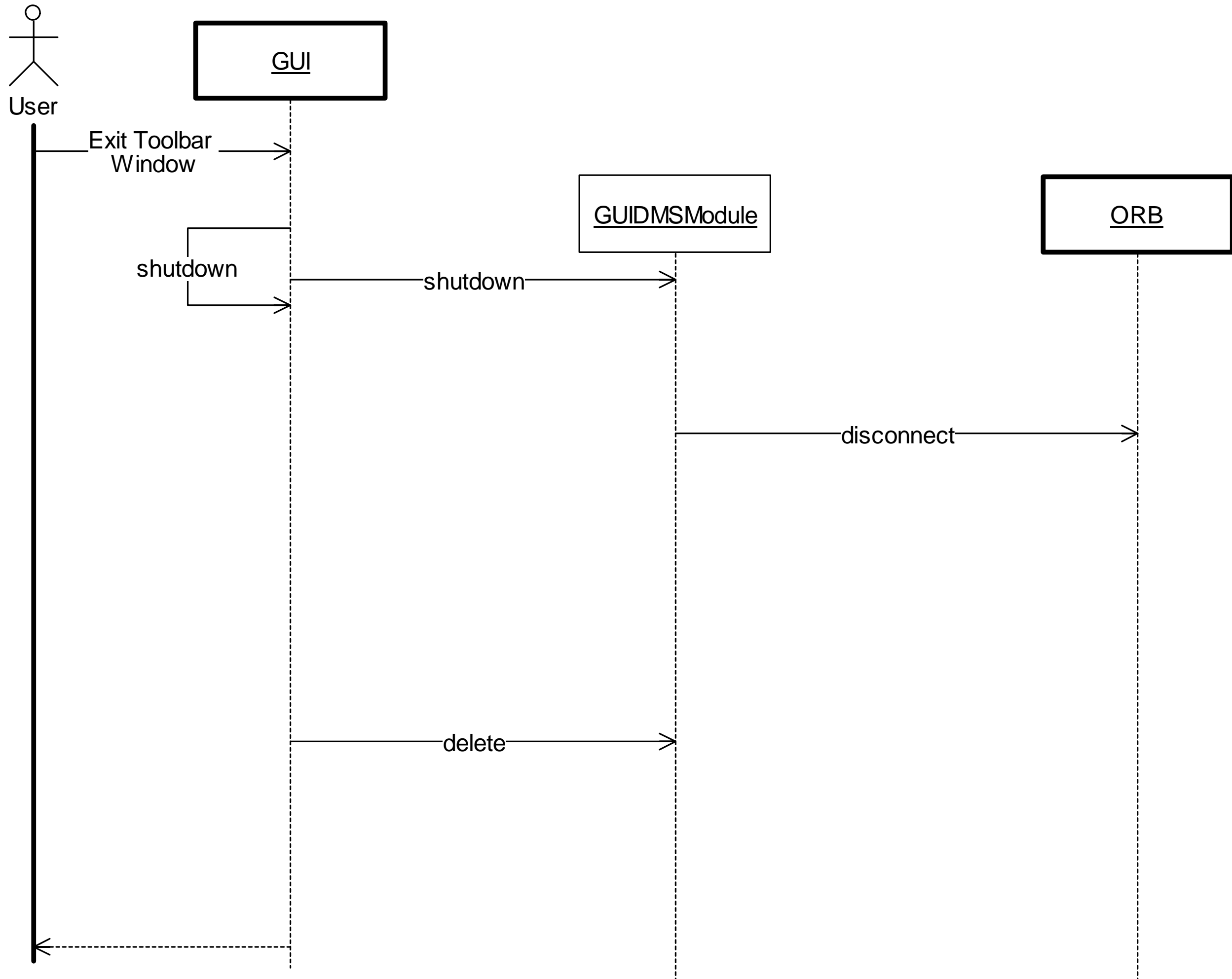


Figure 5-35. GUIDMSModule:Shutdown (Sequence Diagram)

5.36 GUIDMSModule:Startup (Sequence Diagram)

This diagram shows the actions of the DMS module when the GUI starts up. Because the DMS module will be the consumer of DMS related events from the event service, it must be connected to the ORB during startup. The Navigator groups for message libraries and DMSs are also created and added to the DataModel.

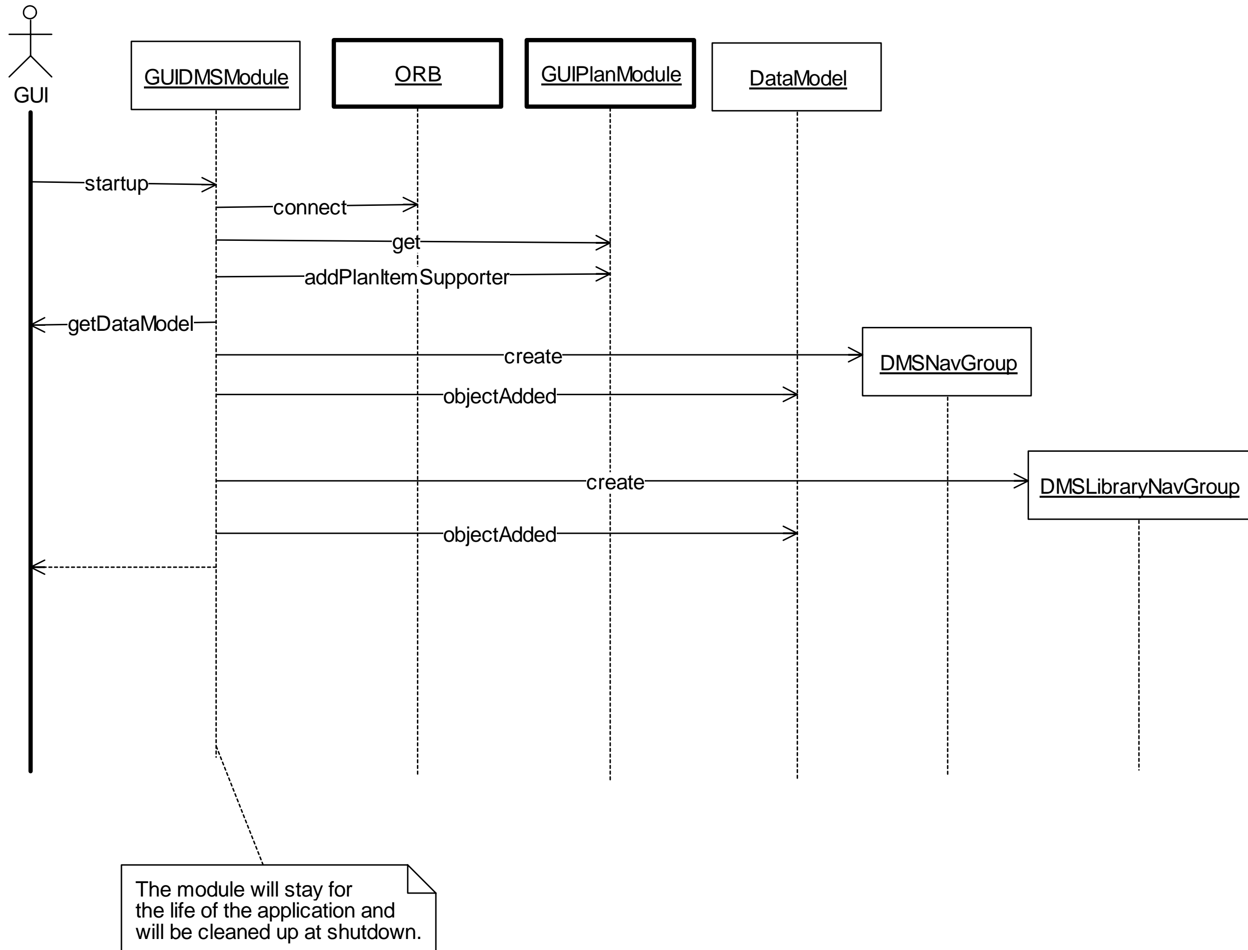


Figure 5-36. GUIDMSModule:Startup (Sequence Diagram)

5.37 GUIDMSModule:Discovery (Sequence Diagram)

This sequence shows how the GUIDMSModule will perform discovery of system objects and make objects that are discovered available to the operator. The operator does not need to take any action to initiate this sequence. The GUI Discovery thread will periodically call methods on each installed module that will cause that module to discover event channels and objects. The GUIDMSModule will discover DMS Event Channels, DMS objects, and DMS message library objects. Each time it discovers one of these objects it will check the DataModel to see if the object has previously been discovered. If it has, no action needs to be taken. If it has not, the object must be added to the DataModel. The object will then show up in all appropriate windows.

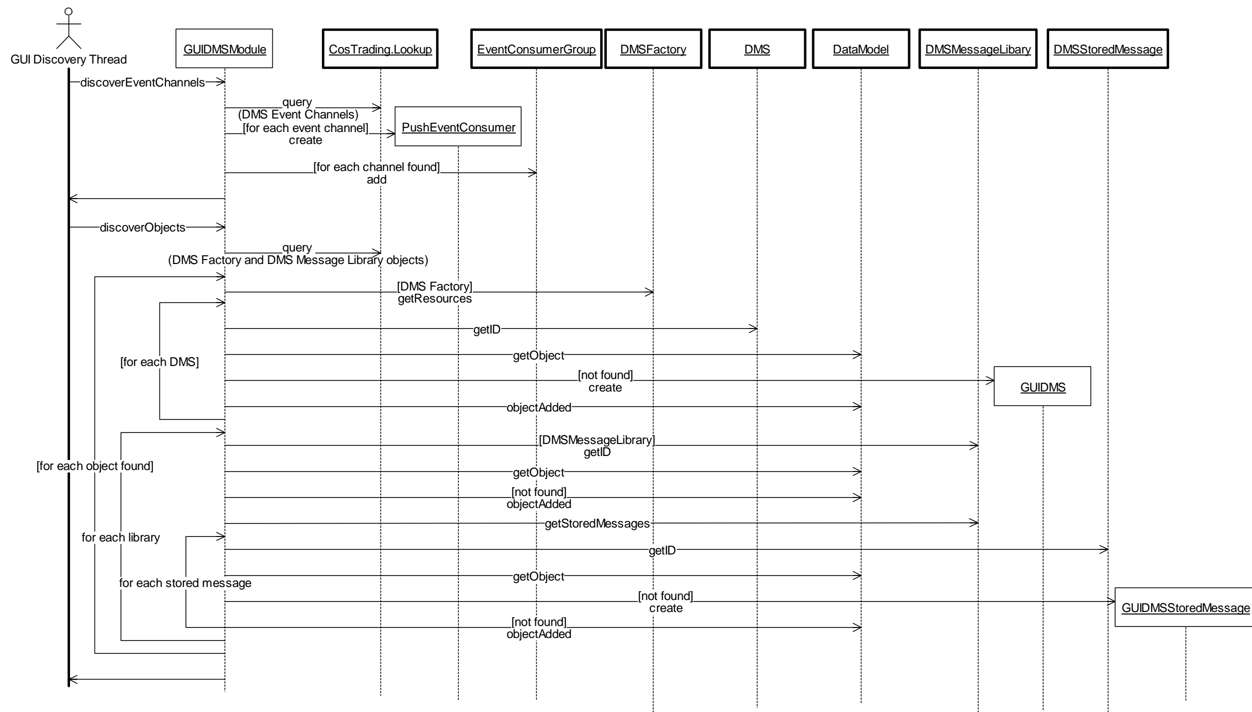


Figure 5-37. GUIDMSModule:Discovery (Sequence Diagram)

5.38 GUIDMSModule:ForcePoll (Sequence Diagram)

This sequence diagram shows how an operator forces the poll of a DMS to get an update of the signs current status information. The sequence is initiated when an operator right clicks on a DMS in the navigator and selects the Update Status menu item. If the user does not have the appropriate functional rights, the DMS will not put this item in the menu when the menu is displayed. The access denied exception should never be encountered. It is handled here as a failsafe. The user will see the Command Status in the Command Status view while it is being handled by the DMS service. Once the DMS has been polled, the user will see the new status of the DMS in the navigator.

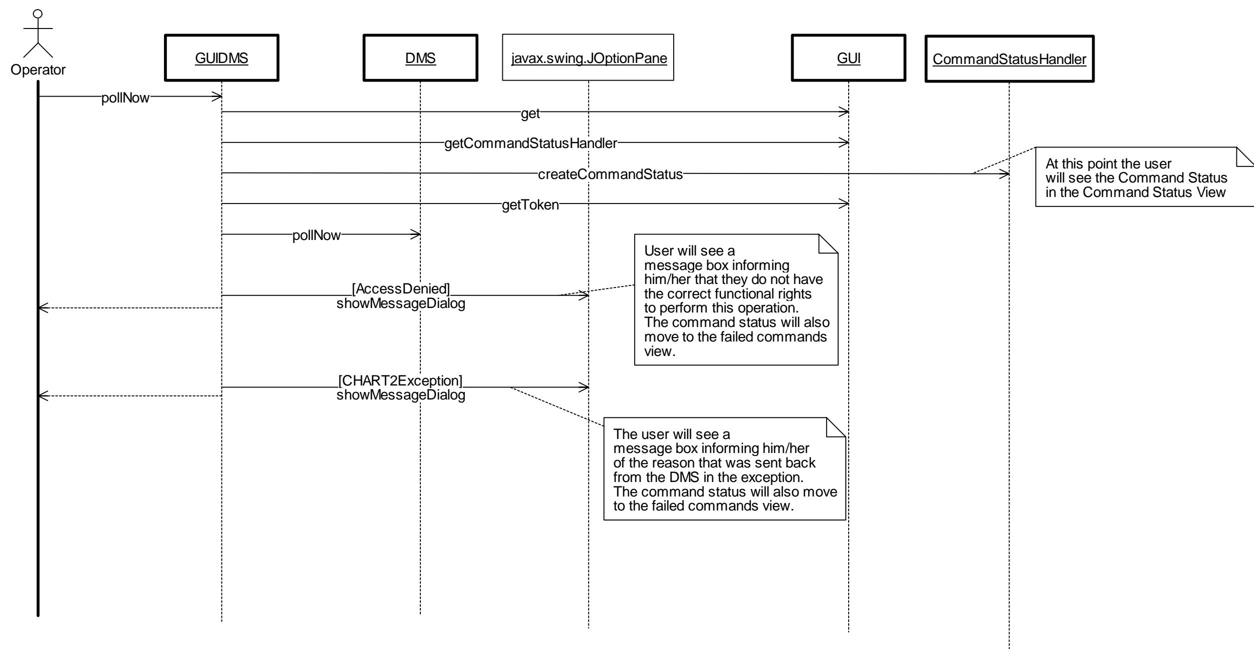


Figure 5-38. GUIDMSModule:ForcePoll (Sequence Diagram)

5.39 GUIDMSModule:ModifyDMSSettings (Sequence Diagram)

This sequence shows how an operator may alter the configuration of a DMS. The operator initiates this action by right clicking on the DMS in a window and selecting the "Properties" menu item. If the user does not have the appropriate functional rights, this menu item will not be made available. The operator will then be shown a DMS properties dialog box with the current configuration information for the selected DMS which he/she may modify to alter the configuration as appropriate. When the user presses OK, the DMS will be reconfigured. If the DMS configuration information is erroneous, the user will be shown a popup window describing the reason it could not be altered. If the operation is successful, the DMS will begin using the new configuration information.

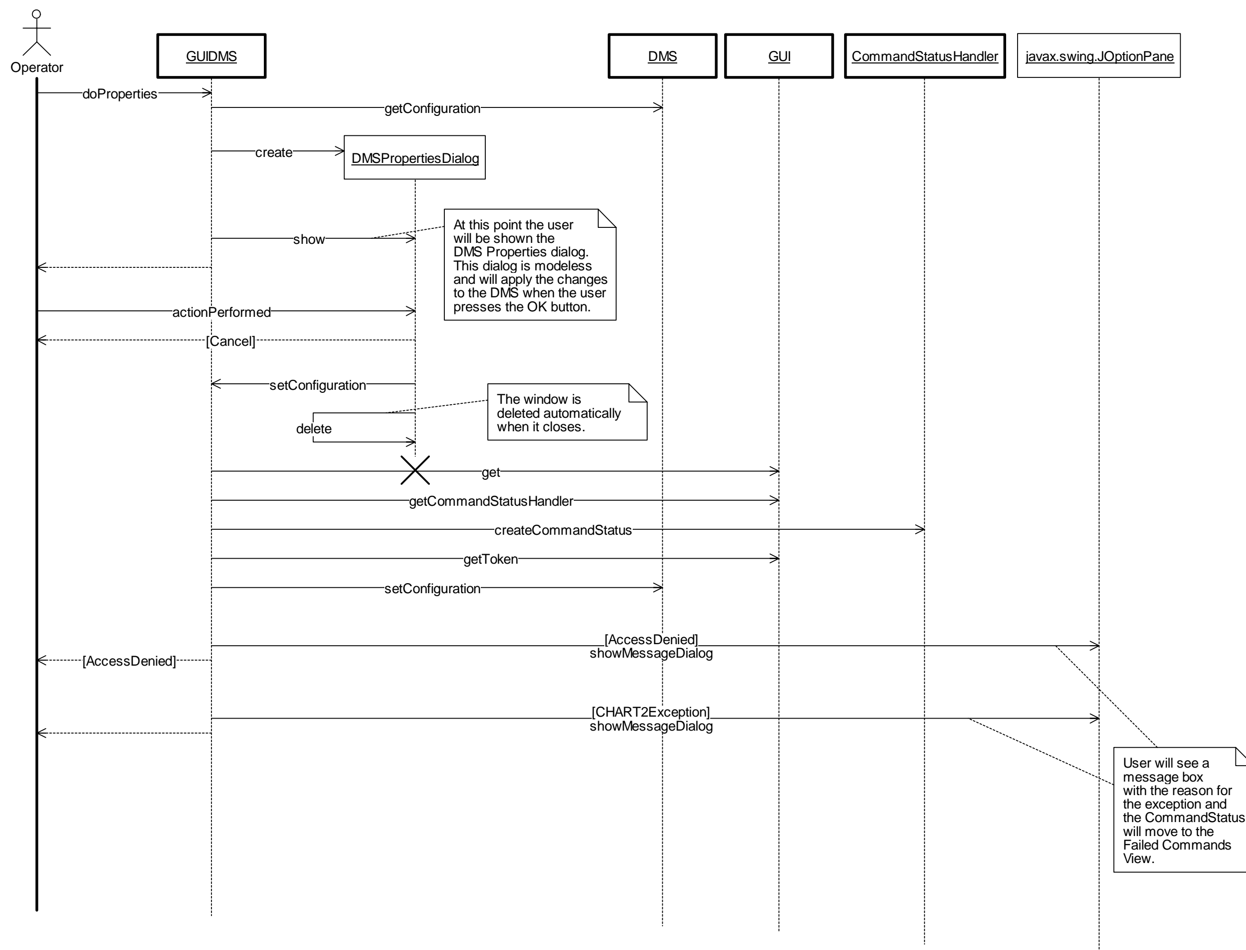


Figure 5-39. GUIDMSModule:ModifyDMSSettings (Sequence Diagram)

5.40 GUIDMSModule:PutOnline (Sequence Diagram)

This sequence diagram shows how an operator puts a DMS online. The sequence is initiated when an operator right clicks on a DMS in the navigator and selects the Put Online menu item. The DMS will not put the item in the menu when the menu is displayed unless the user has the appropriate functional rights, so the "access denied" should never be encountered. It is handled here as a failsafe.

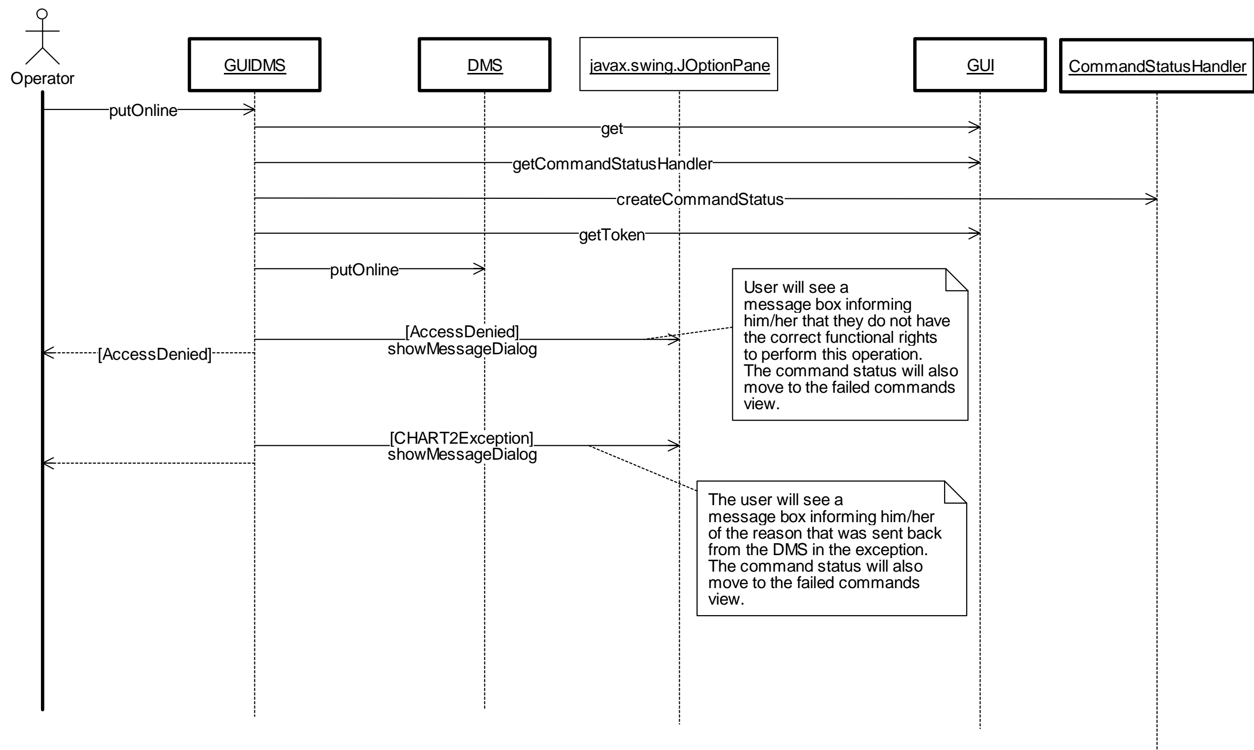


Figure 5-40. GUIDMSModule:PutOnline (Sequence Diagram)

5.41 GUIDMSModule:Reset (Sequence Diagram)

This sequence diagram shows how an operator resets a particular DMS controller. The sequence is initiated when an operator right clicks on a DMS in the navigator and selects the Reset menu item. If the user does not have the appropriate functional rights, the DMS will not put the item in the menu when the menu is displayed. The access denied exception should never be encountered. It is handled here as a failsafe. Upon completion of this sequence the user will have a Command Status in the Command status view which will display the state of the operation as the server processes it. When the command completes and the controller has been reset, the navigator and other views will be updated with the new status information for the DMS.

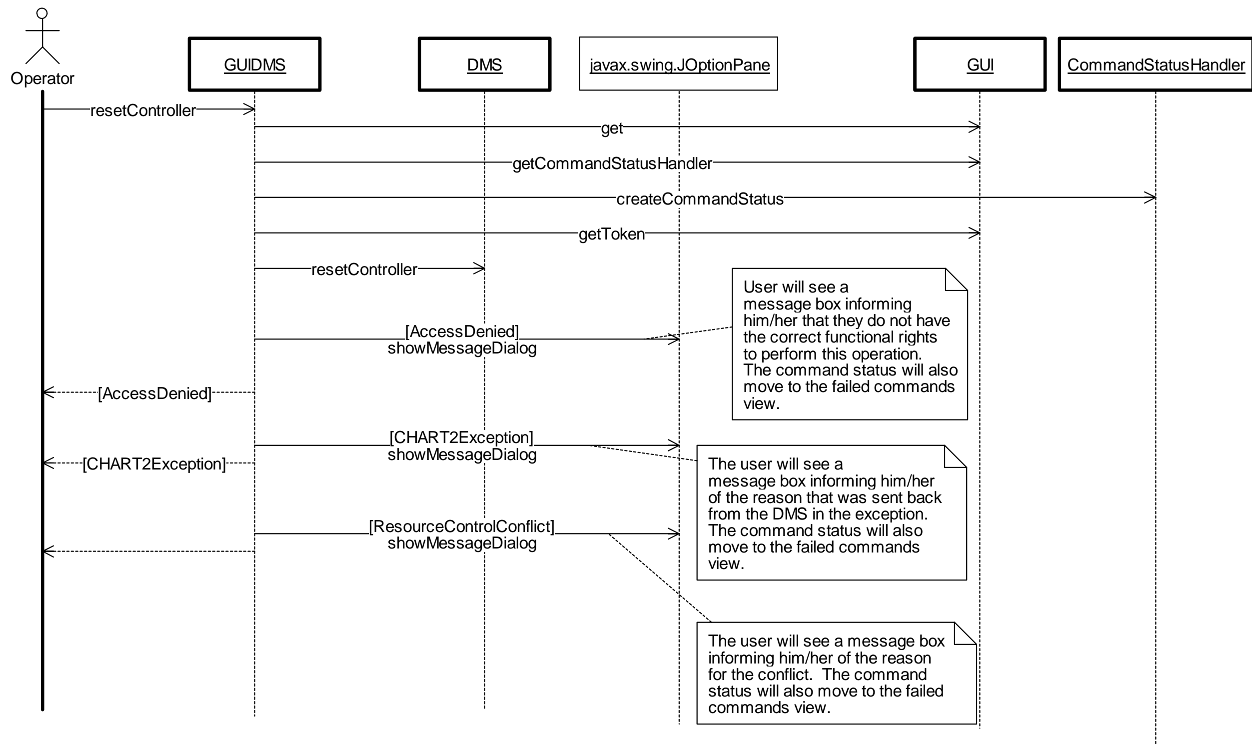


Figure 5-41. GUIDMSModule:Reset (Sequence Diagram)

5.42 GUIDMSModule:SetMessage (Sequence Diagram)

This sequence shows how an operator may change the current message being displayed on a particular DMS. The operator initiates this operation by right clicking on the appropriate DMS in a window and selecting the "Edit Message" item from the resulting popup menu. This item will not be available if the user does not have the appropriate functional rights. The user will then be shown a DMSMessageEditor dialog populated with the current message from that DMS. The user may use this dialog to type in a new message and preview what that message will look like formatted for the selected DMS. When the user is done, the new message will be sent to the DMS. If the message cannot be changed, the user will be notified via a popup message box and/or the failed commands view. If the message is altered, the new message will show up in all GUI windows displaying the message for that particular DMS.

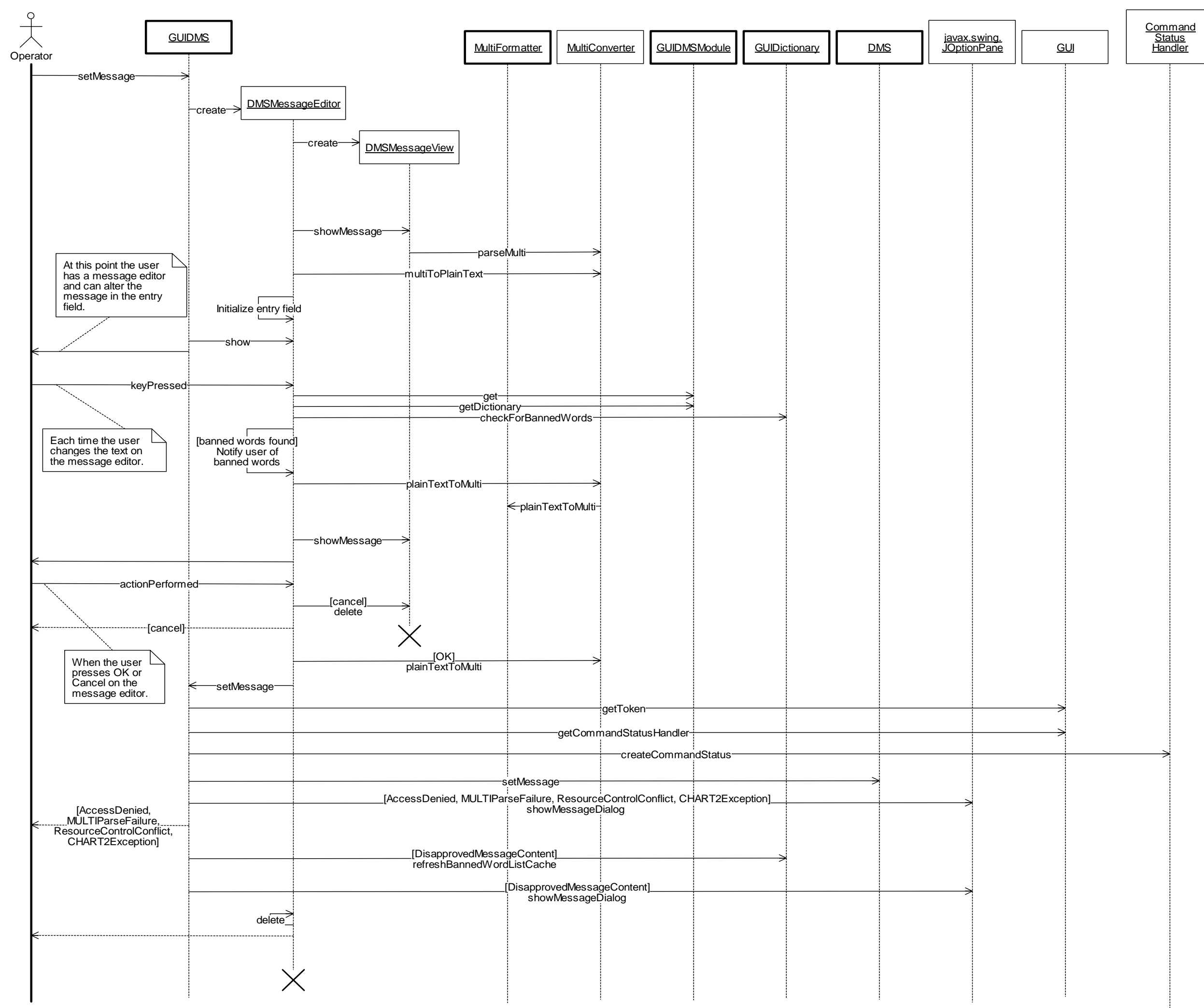


Figure 5-42. GUIDMSModule:SetMessage (Sequence Diagram)

5.43 GUIDMSModule:ShowTrueDisplay (Sequence Diagram)

This sequence shows how an operator may view the current message displayed on a particular DMS. The view will be formatted to show the message as it looks on the sign. The operator initiates this sequence by right clicking on the desired DMS in a window and selecting the "Show Display" menu item.

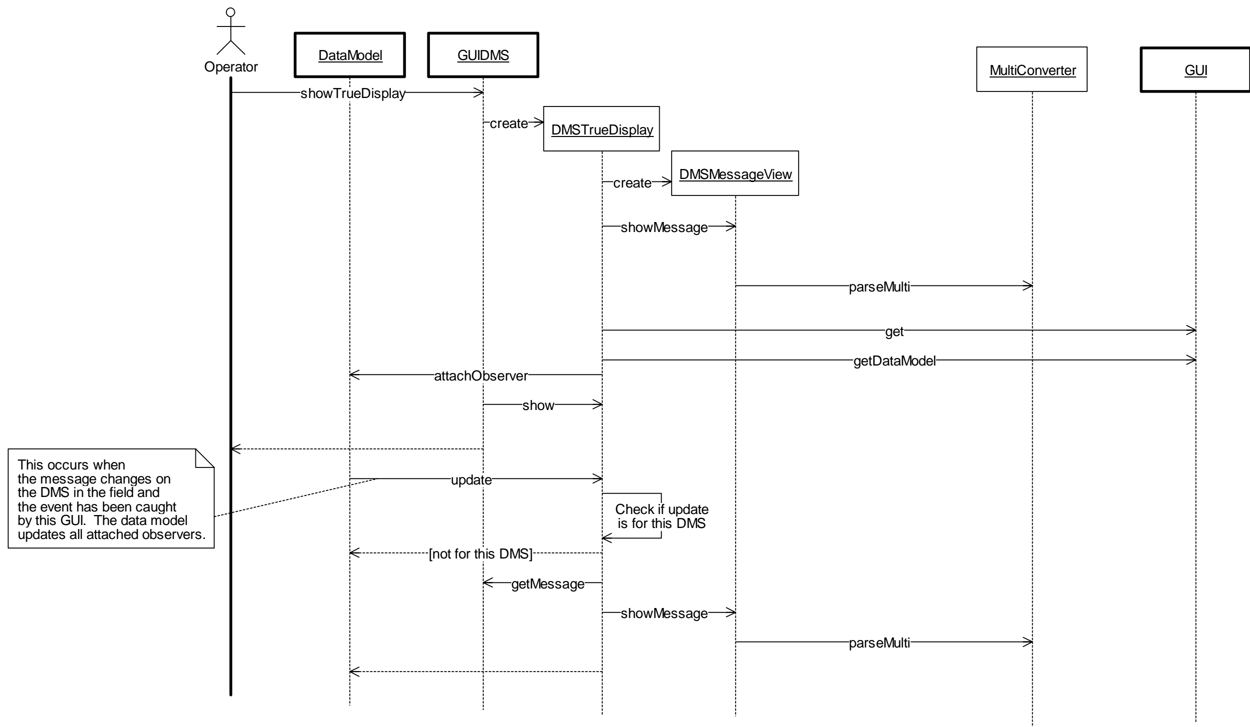


Figure 5-43. GUIDMSModule:ShowTrueDisplay (Sequence Diagram)

5.44 GUIDMSModule:TakeOffline (Sequence Diagram)

This sequence diagram shows how an operator takes a DMS offline. The sequence is initiated when an operator right clicks on a DMS in the navigator and selects the Take Offline menu item. If the user does not have the appropriate functional rights, the DMS will not put the take offline item in the menu when the menu is displayed. The access denied exception should never be encountered. It is handled here as a failsafe.

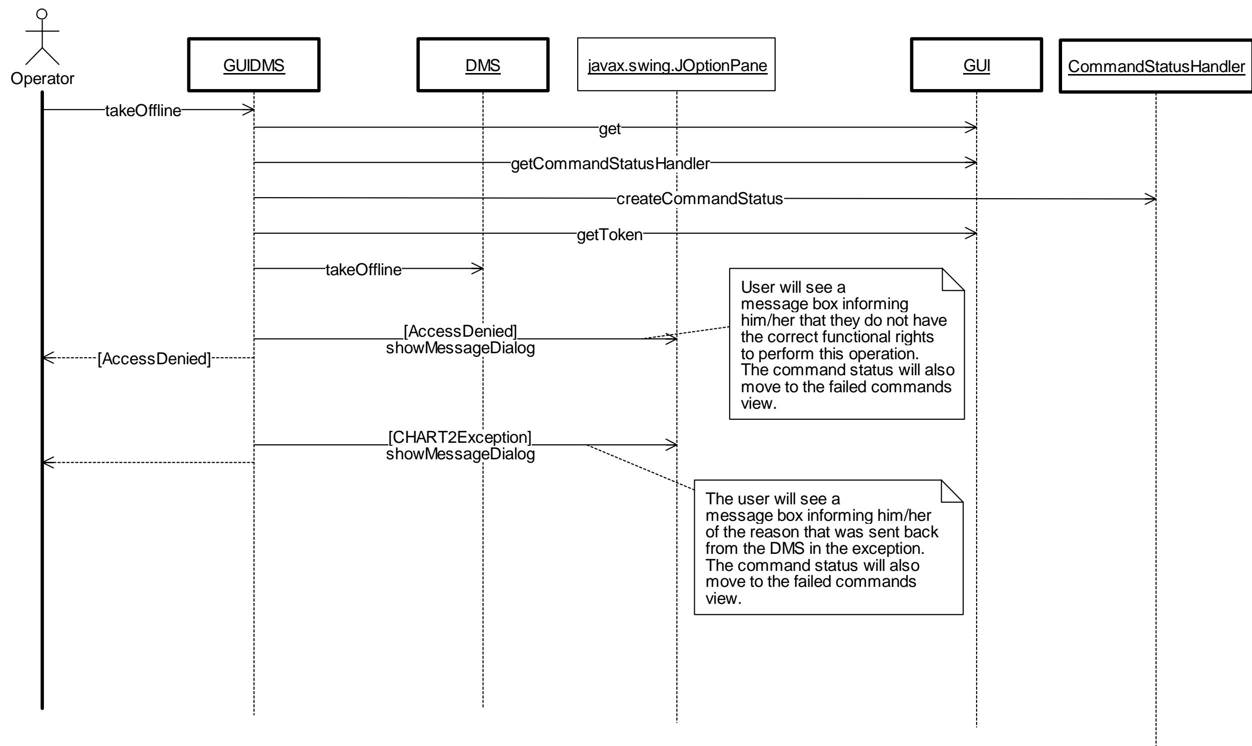


Figure 5-44. GUIDMSModule:TakeOffline (Sequence Diagram)

5.45 GUIDictionaryModule:DictionaryProperties (Sequence Diagram)

This diagram shows how the editing of the words in a given dictionary will be done. It begins with a user clicking on a menu item from the GUIDictionary's context menu. Since the GUIDictionary will be an ActionListener for the menu item, the GUIDictionary will be called and then creates the BannedWordsDialog. This dialog attaches itself as an observer to the DataModel in order to catch any updates to the word list (which will come through the event channel and then through the DataModel). It gets the list of banned words and displays them to the user. When the user provides a list of banned words to add or remove, the GUIDictionary will make a call to the served Dictionary object. If the words are added successfully, the Dictionary object will push an event through the Dictionary event channel. (See the EventHandling diagram for details.) The DataModel will then call the dialog's update() method, and the dialog will ask the GUIDictionary wrapper for the current list of words to display. Just before the dialog is closed, it will detach from the DataModel.

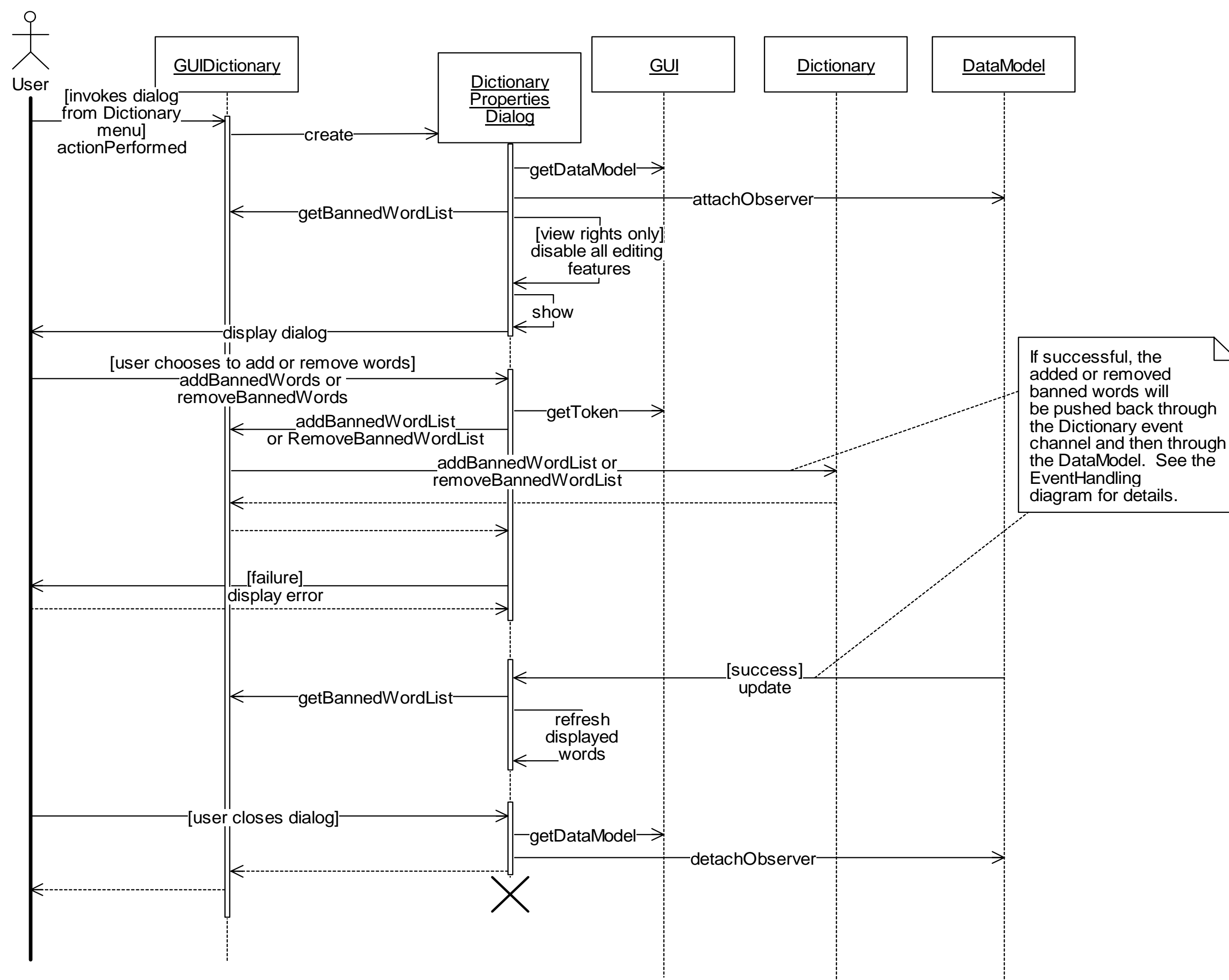


Figure 5-45. GUIDictionaryModule:DictionaryProperties (Sequence Diagram)

5.46 GUIDictionaryModule:Discovery (Sequence Diagram)

This diagram shows how the Dictionary event channels and Dictionary objects are discovered and added to the system. This will be a periodic process, and the GUI will call the GUIDictionaryModule repeatedly. When the GUI asks the module to discover event channels, it looks up the Dictionary event channels in the trader. It then creates a PushEventConsumer and adds it to the EventConsumerGroup, which actually attaches the consumer to the channel and reattaches it if the event service is restarted. (Duplicate channels will be ignored). The GUI then calls the module to discover objects. At this time the module will query the Dictionary objects in the trader. If any are found, it will create an Identifier to be used as a lookup key for use with the DataModel. If the GUIDictionary wrapper object does not already exist in the DataModel, it is created and added. Creating the wrapper will cause the new wrapper to initialize its data by making a remote call to the served Dictionary object. The GUIDictionary is then added to the GUIDictionaryNavGroup and the DataModel is called to propagate the changes to any interested observers such as the BannedWordsDialog.

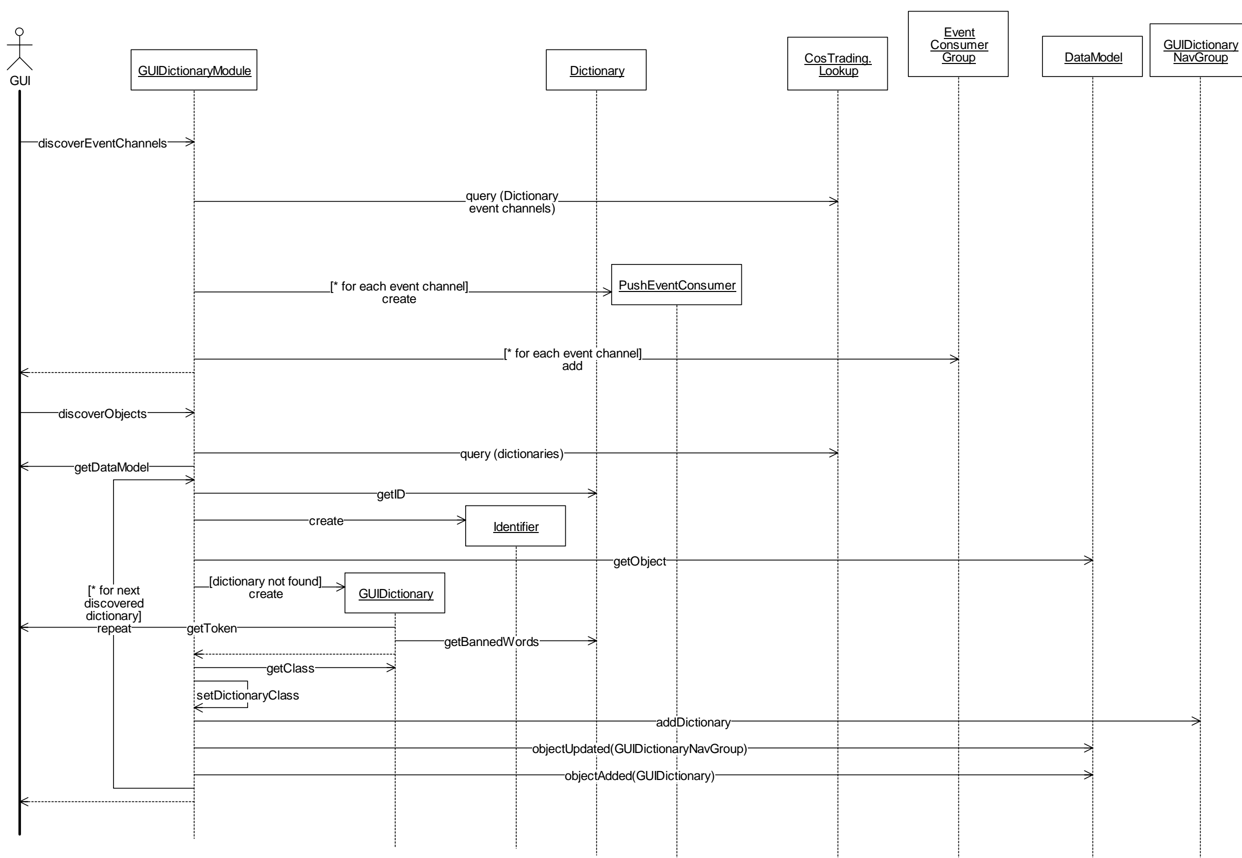


Figure 5-46. GUIDictionaryModule:Discovery (Sequence Diagram)

5.47 GUIDictionaryModule:EventHandling (Sequence Diagram)

This diagram shows how dictionary events are propagated through the GUI when they are pushed from the event channel. The ORB invokes the push method of the DictionaryEventConsumer. The event data contains a byte array identifier, which is used to create an Identifier object to get the GUIDictionary object from the DataModel. The words are added or removed from the wrapper's cache, and then the DataModel is called to update any observers that may be listening for updates, such as the BannedWordsDialog.

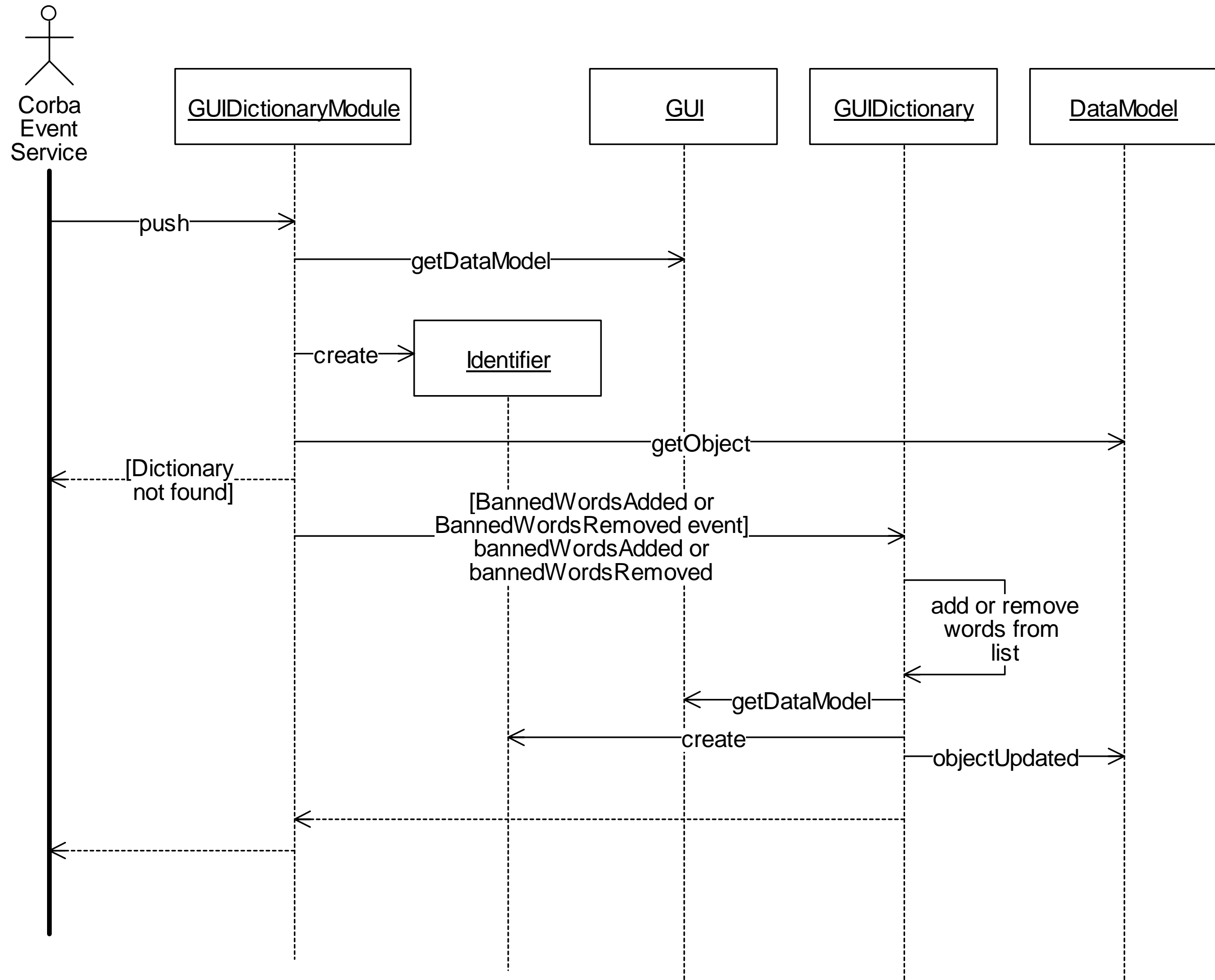


Figure 5-47. GUIDictionaryModule:EventHandling (Sequence Diagram)

5.48 GUIDictionaryModule:Shutdown (Sequence Diagram)

This diagram shows what happens at shutdown. The module disconnects from the ORB to clean up.

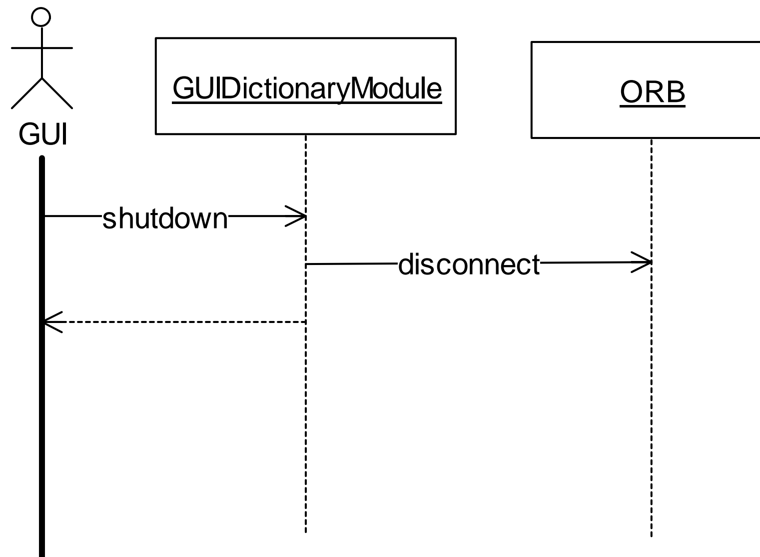


Figure 5-48. GUIDictionaryModule:Shutdown (Sequence Diagram)

5.49 GUIDictionaryModule:Startup (Sequence Diagram)

This diagram shows the steps taken to initialize the GUIDictionaryModule. The GUI will call the module's startup method. The module will create a GUIDictionaryNavGroup and add it to the DataModel so that the Navigator will display it. The module will store the group for later use. The GUIDictionaryModule will attach itself to the ORB so that it can serve as a PushConsumer to receive dictionary events.

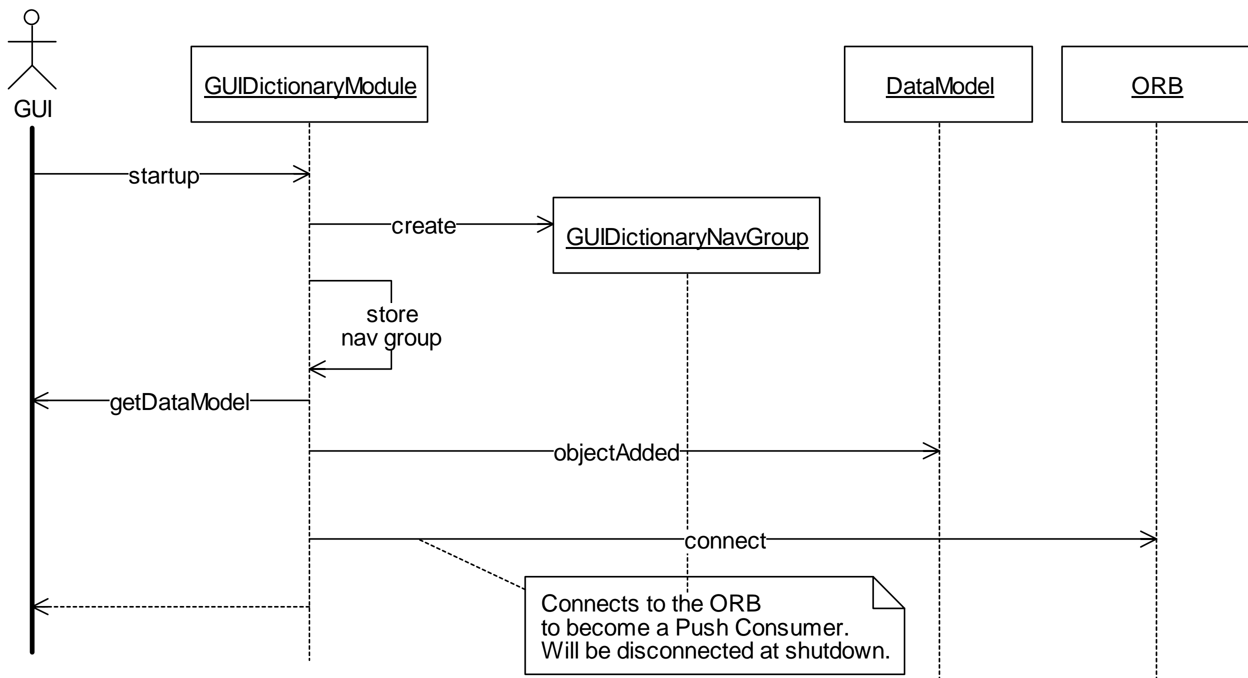


Figure 5-49. GUIDictionaryModule:Startup (Sequence Diagram)

5.50 GUIPlanModule:ActivatePlan (Sequence Diagram)

This diagram shows how a plan is activated. The user clicks on the GUIPlan and invokes the context menu. If the user has rights, the Activate Plan menu item will be displayed. When it is clicked on, th GUIPlan calls the CommandStatusHandler to create a CommandStatus and put it in the DataModel so that the Command Status View (or Command Failure View) will be notified of the command's progress. The GUIPlan then calls the Plan to activate it. The activation happens asynchronously, and the CommandStatus object is updated as the command progresses.

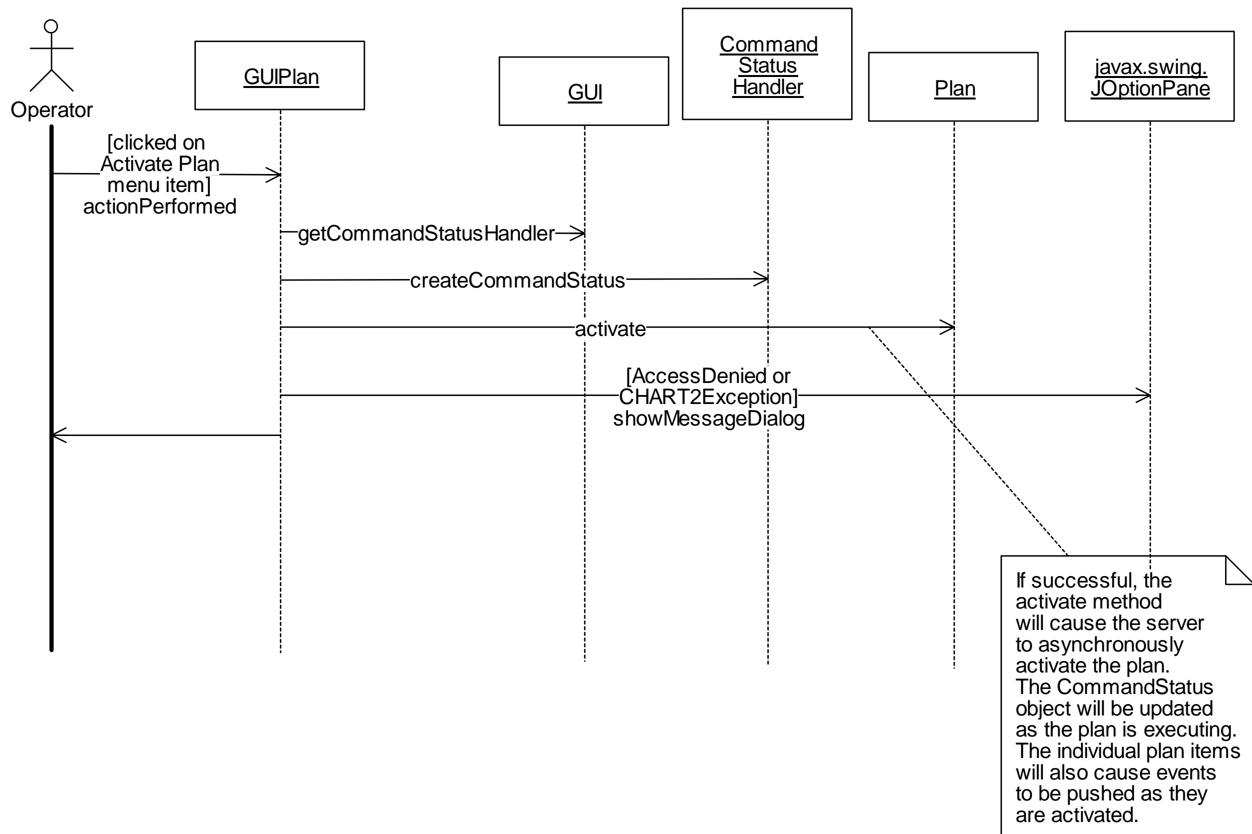


Figure 5-50. GUIPlanModule:ActivatePlan (Sequence Diagram)

5.51 GUIPlanModule:AddPlan (Sequence Diagram)

This diagram shows how a new plan is created. The user clicks on the Create Plan menu item in the GUIPlanNavGroup's context menu. (This menu item will only be displayed if the user has rights.) The GUIPlanNavGroup will create an uninitialized GUIPlan object with default properties and will call its doProperties method. This is a temporary object, used only for displaying the properties. The temporary GUIPlan will create a modeless PlanPropertiesDialog and display it. When the user clicks OK, the dialog will ask the GIUPlan to create a Plan from the properties entered from the dialog. The GUIPlan will then query all of the Plan Factories from the trader and will use the first one to create the Plan object. If the Plan was created, the PlanAdded event will be pushed from the plan server through the plan event channel to update all of the GUIs. See the GUIPlanModule:PlanAddedEvent diagram for more details.

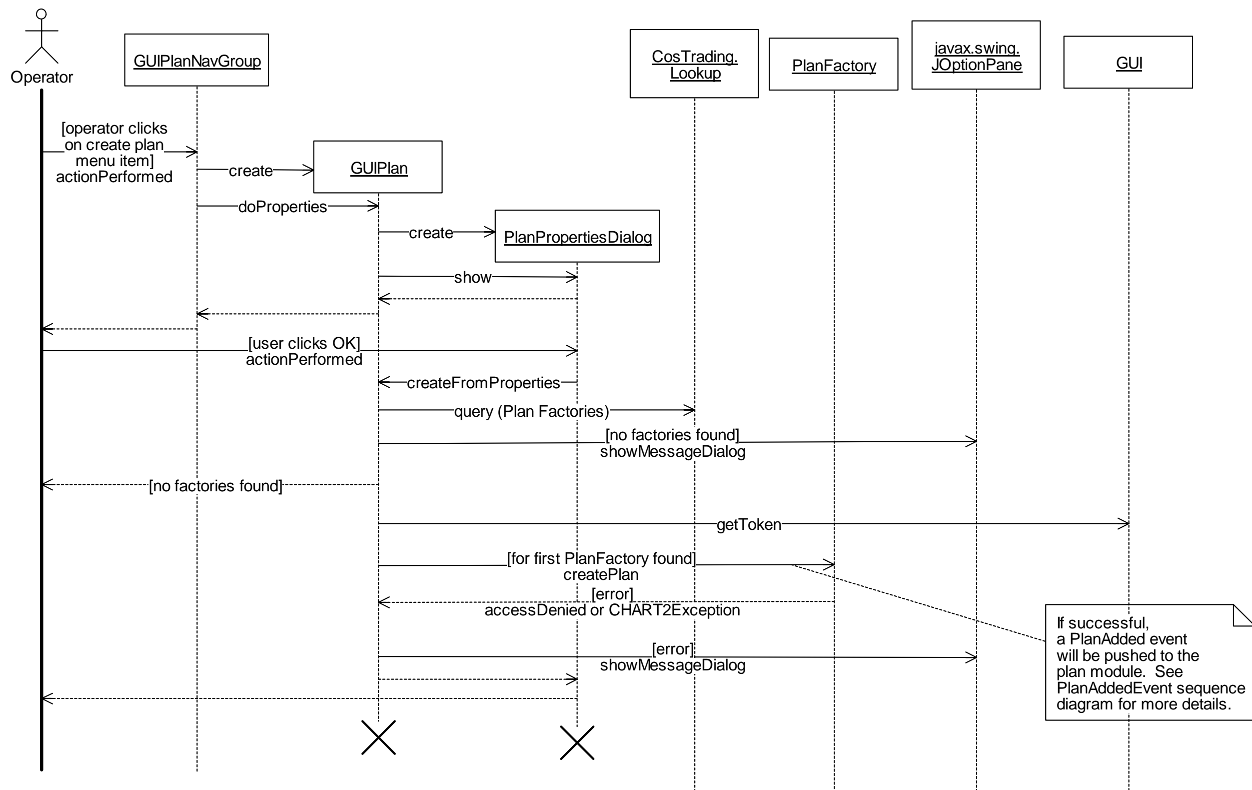


Figure 5-51. GUIPlanModule:AddPlan (Sequence Diagram)

5.52 GUIPlanModule:CreatePlanItem (Sequence Diagram)

This diagram shows how a plan item is created. When the user invokes the menu on the GUIPlan object, the GUIPlan object asks the GUIPlanModule for all of the attached PlanItemCreationSupporters. It then asks each of the supporters for the strings to use for the plan item creation menu items. Each string is associated with the supporter that supplied it, and the associations are stored in the GUIPlan object for use when a menu item is clicked on. When the user clicks on one of these menu items, the GUIPlan's actionPerformed method will be called, and the GUIPlan will find the matching string stored in the association, and will call the corresponding PlanItemCreationSupporter to create the new plan item. See the modules that support plan item creation for more details on how plan items are created.

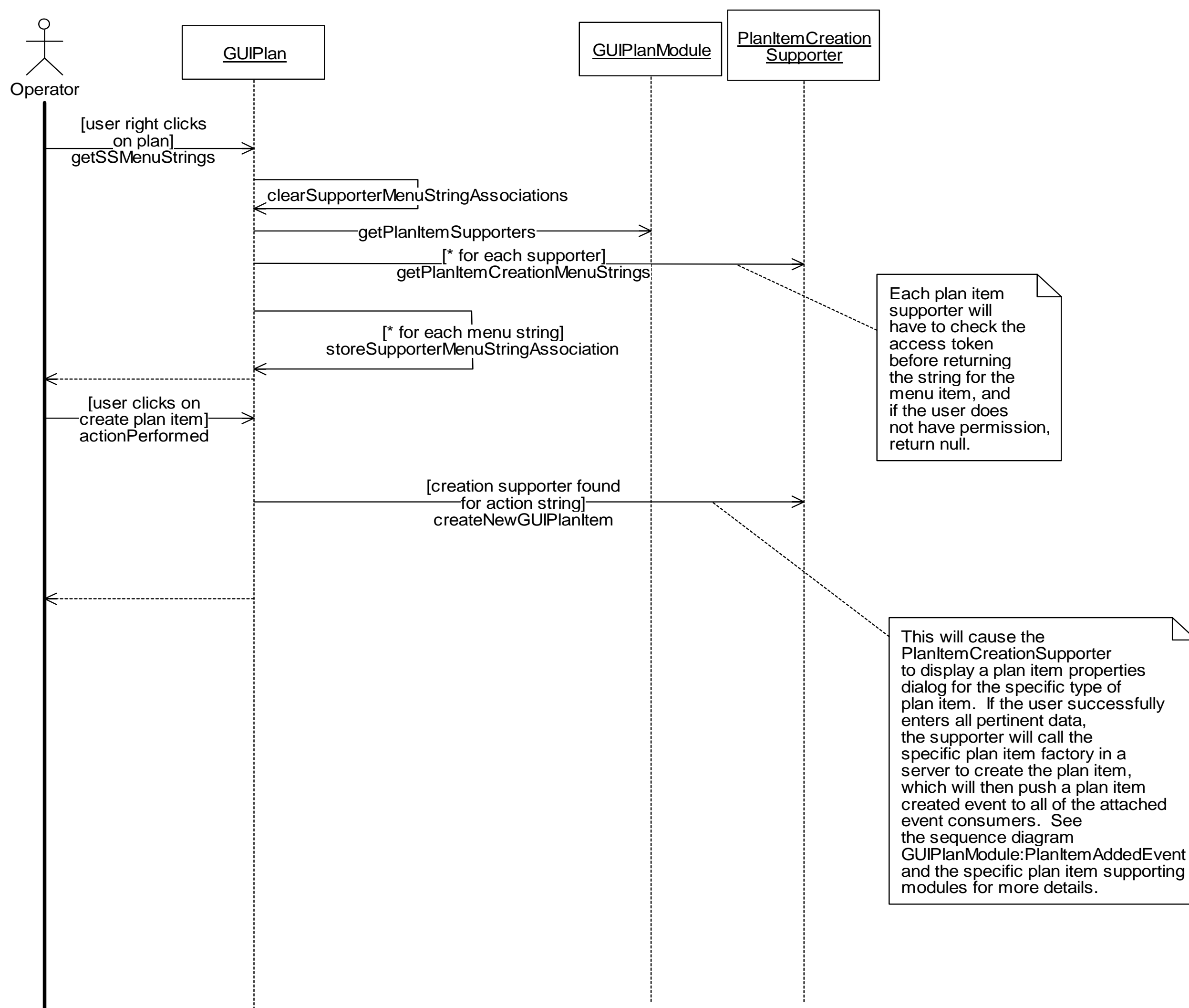


Figure 5-52. GUIPlanModule:CreatePlanItem (Sequence Diagram)

5.53 GUIPlanModule:Discovery (Sequence Diagram)

This diagram shows what happens during the discovery process, in which the module has a chance to find out about event channels and objects. The GUI will periodically call the module, first to discover event channels and then to discover objects. During the event channel discovery phase, the module looks for Plan event channels in the trader. If it finds any, it creates a PushEventConsumer and attaches itself to the Event Consumer Group. This will attach the module to the event channel and will reattach it automatically if the event service is restarted. If the module was previously attached to the event channel, it will be ignored. During the object discovery phase, the GUI calls the module to discover objects. The module will query the Plan objects in the trader. If the Plan does not already exist in the DataModel, a new GUIPlan wrapper object will be created and added to the DataModel. When the GUIPlan object is created, it asks the Plan for all of its PlanItems. For each item which is not already in the DataModel, it will call all of the attached PlanItemCreationSupporters and ask each one to attempt to create the specific type of GUIPlanItem wrapper object for the generic PlanItem object. Each PlanItemCreationSupporter will check whether the generic PlanItem object is of its own specific class of plan item. If so, the PlanItemCreationSupporter must create an object of its own specific class of GUIPlanItem object to wrap the PlanItem. If a wrapper object was created, it will be added to the DataModel. After a short delay, the changes made through the DataModel will update any windows that are attached to the DataModel.

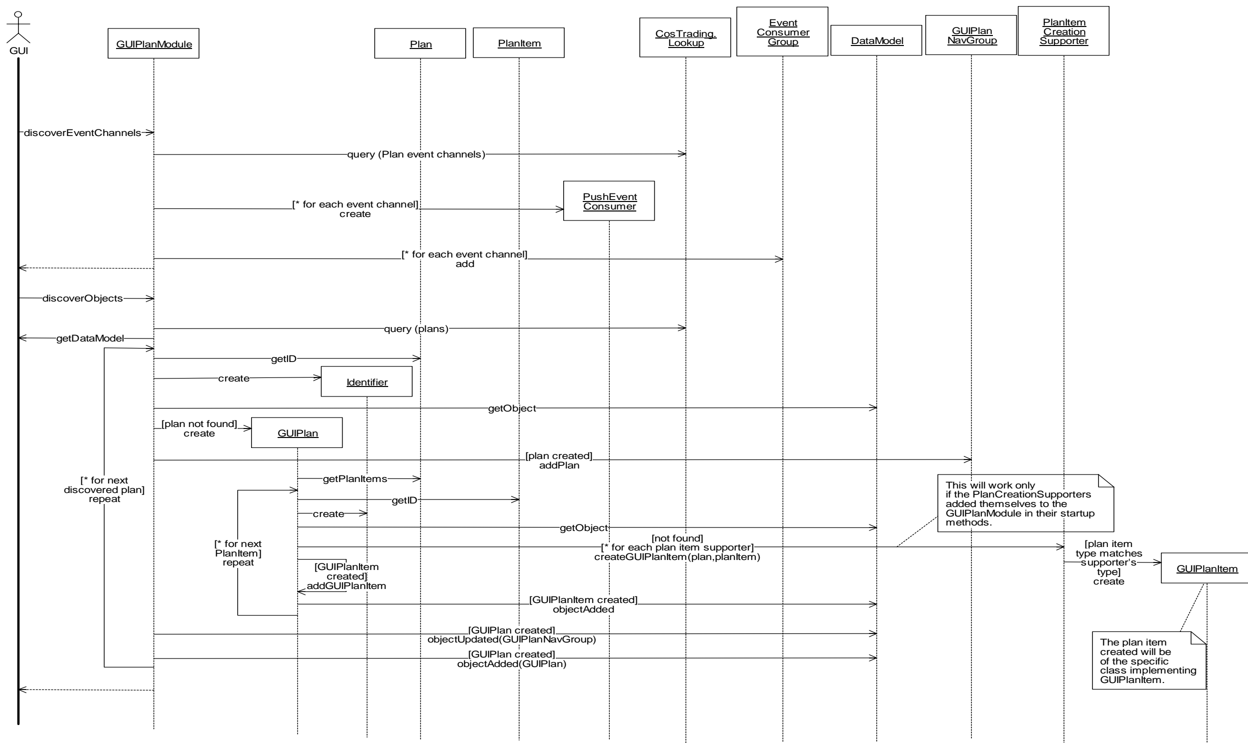


Figure 5-53. GUIPlanModule:Discovery (Sequence Diagram)

5.54 GUIPlanModule:PlanItemAddedEvent (Sequence Diagram)

This diagram shows the handling of the event after a new PlanItem has been created. First, the GUIPlan to which the new PlanItem belongs is retrieved from the DataModel. Then the module will ask each PlanItemCreationSupporter to attempt to create a specific GUIPlanItem wrapper object if the generic PlanItem is a correct type for the supporter. If a GUIPlanItem object was created by one of the creation supporters, it is added to the GUIPlan and to the DataModel. The GUIPlan is also updated through the DataModel to make sure that any windows will be updated.

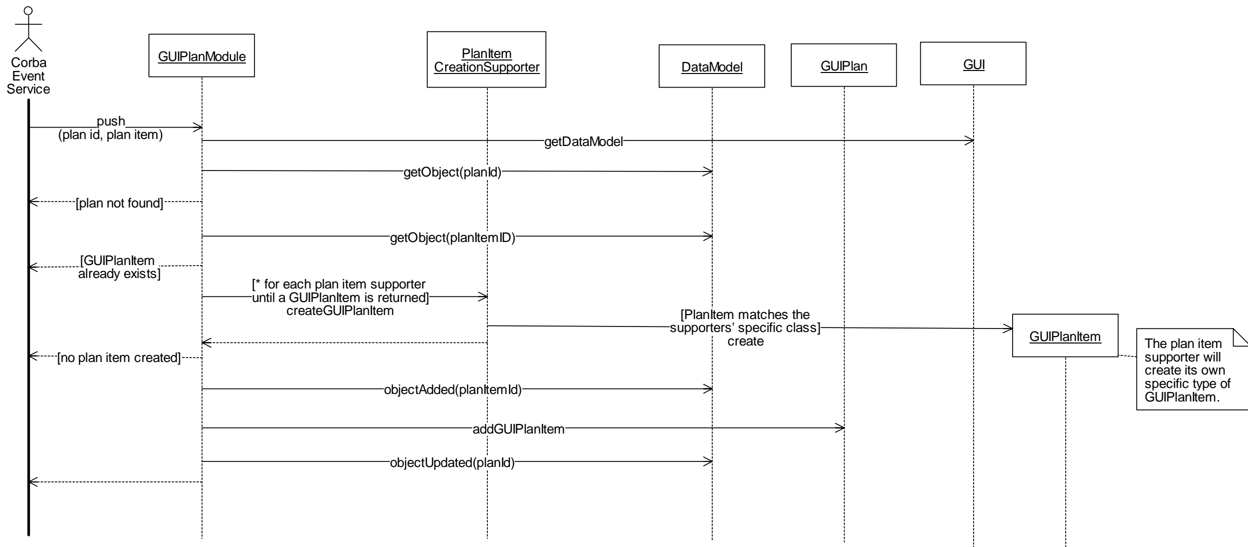


Figure 5-54. GUIPlanModule:PlanItemAddedEvent (Sequence Diagram)

5.55 GUIPlanModule:PlanItemRemovedEvent (Sequence Diagram)

This diagram shows how a PlanItemRemoved event is handled, after a plan item is deleted. The GUIPlanModule received the PlanItem identifier and looks up the GUIPlanItem object in the DataModel. If found, the module gets the GUIPlan and asks it to remove the GUIPlanItem from its collection. The GUIPlan object is then updated through the DataModel, and the GUIPlanItem is removed from the DataModel. Any attached observers (e.g., windows) will be updated after a short delay. The GUIPlanItem will then be removed from memory by Java when the observers remove their references to it.

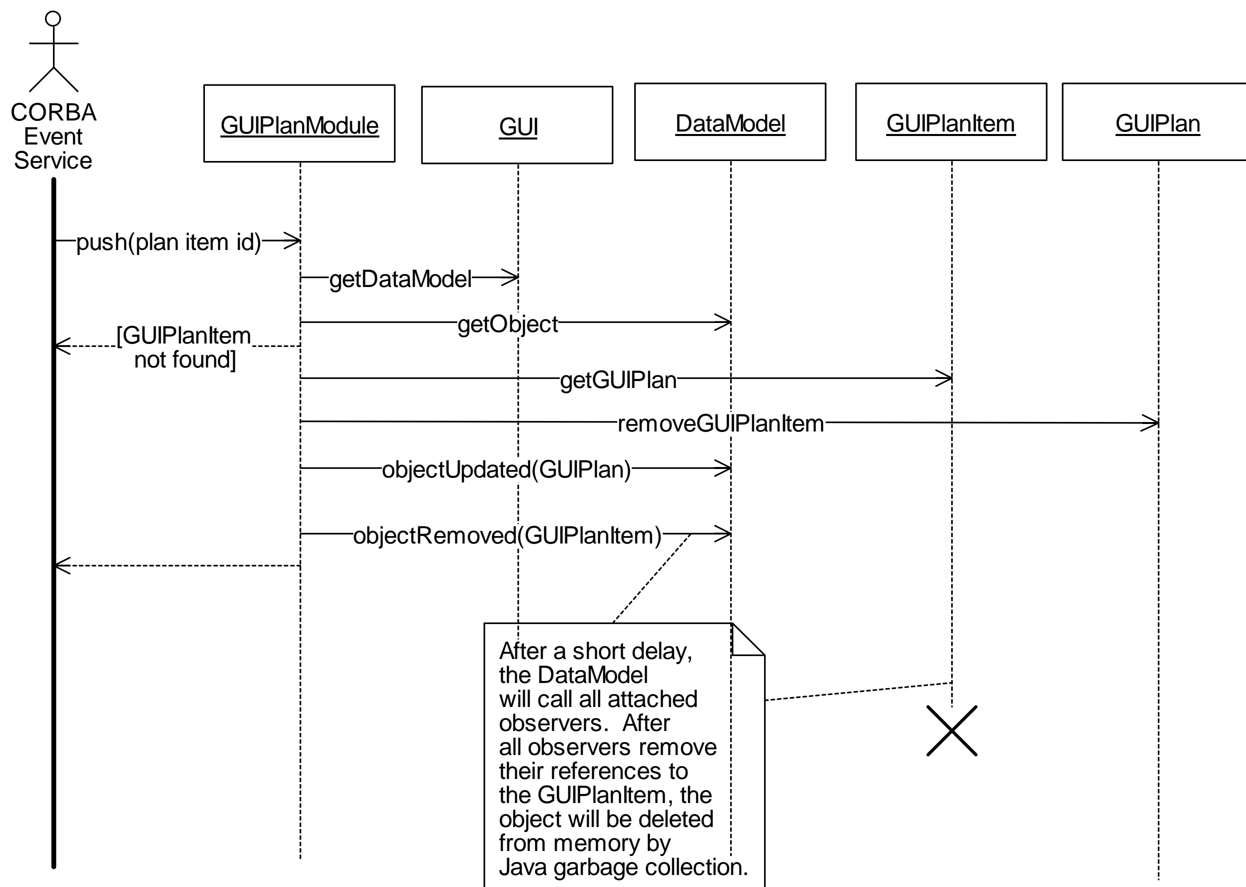


Figure 5-55. GUIPlanModule:PlanItemRemovedEvent (Sequence Diagram)

5.56 GUIPlanModule:PlanRemovedEvent (Sequence Diagram)

This diagram shows how a PlanRemoved event is handled. First, an attempt is made to get the GUIPlan object from the DataModel. If it exists, the GUIPlan is removed from the GUIPlanNavGroup. The GUIPlanNavGroup update notification is invoked through the DataModel, and the GUIPlan is removed from the DataModel. The DataModel will cause any attached observers to display the change.

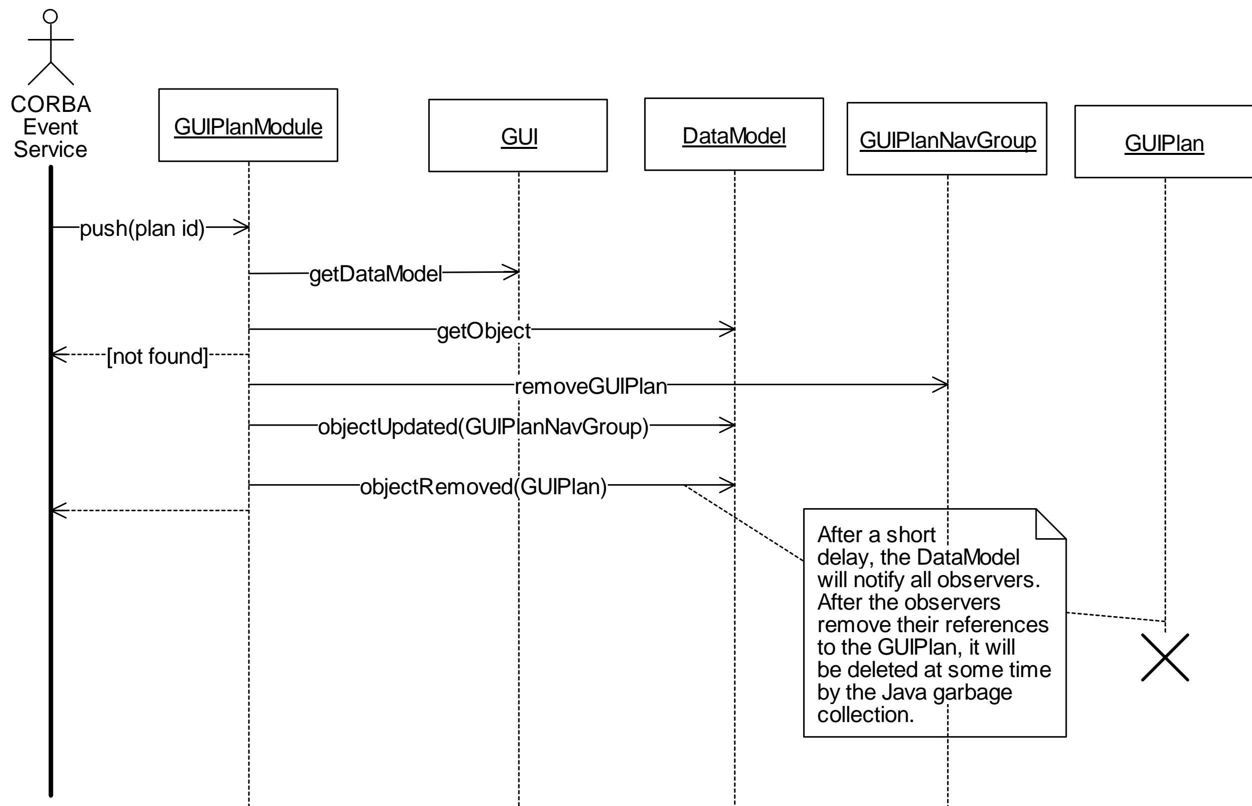


Figure 5-56. GUIPlanModule:PlanRemovedEvent (Sequence Diagram)

5.57 GUIPlanModule:RemovePlan (Sequence Diagram)

This diagram shows how a plan is removed from the system. The operator clicks on the Delete Plan menu item. The GUIPlan then gets the access token and calls the Plan to remove itself. If successful, it will cause the server to push a PlanRemoved event to be pushed through the event channel. See the diagram GUIPlanModule:PlanRemovedEvent for details on how the GUIs are updated after the plan is removed.

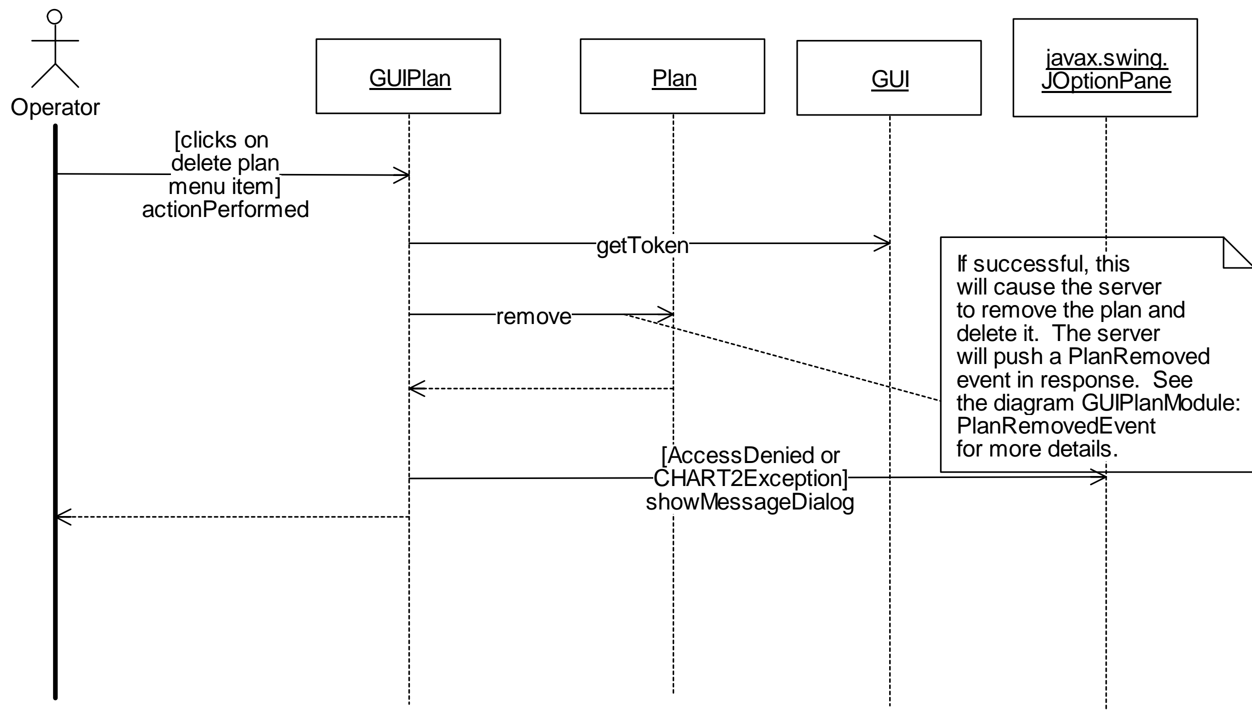


Figure 5-57. GUIPlanModule:RemovePlan (Sequence Diagram)

5.58 GUIPlanModule:PlanAddedEvent (Sequence Diagram)

This diagram shows how the event is handled when a Plan is added. The GUIPlanModule makes sure that the GUIPlan does not already exist in the DataModel, and assuming it does not, it creates the GUIPlan wrapper object for the Plan. The GUIPlan object is then added to the DataModel and the GUIPlanNavGroup, and the DataModel will update all attached observers to show the change.

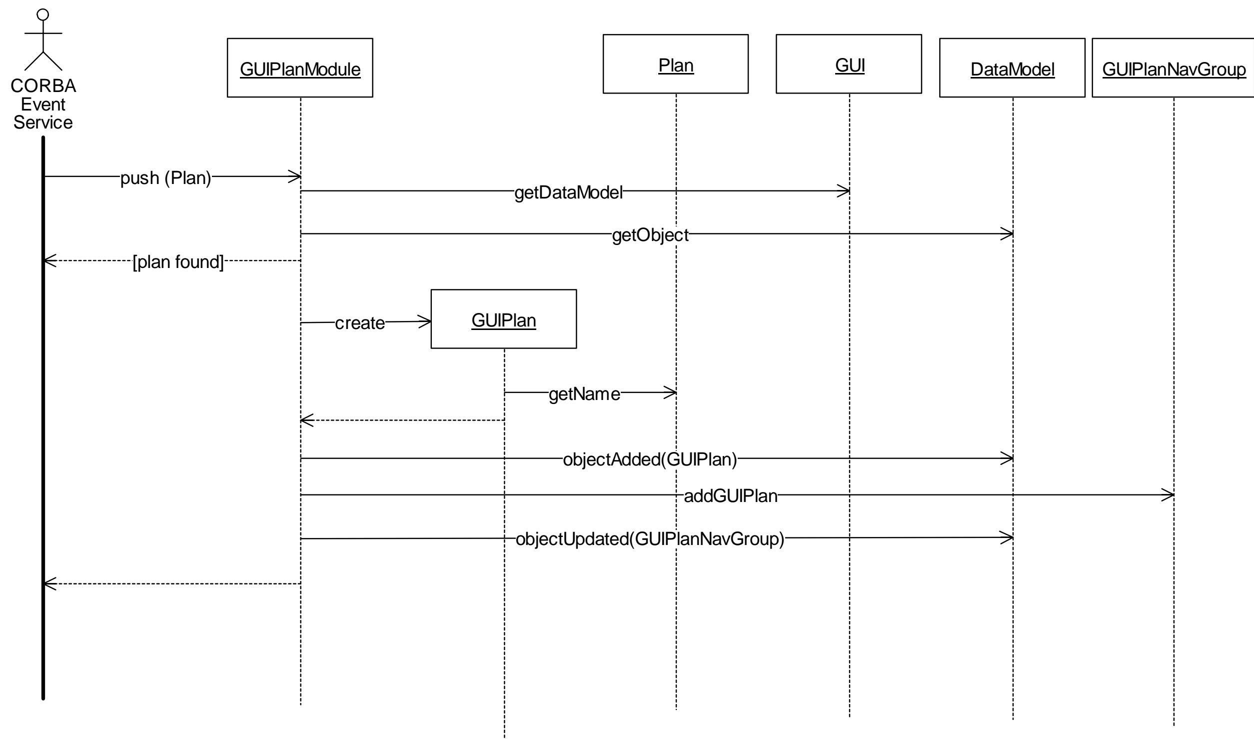


Figure 5-58. GUIPlanModule:PlanAddedEvent (Sequence Diagram)

5.59 GUIPlanModule:RemovePlanItem (Sequence Diagram)

This diagram shows how a plan item is removed from a plan and deleted. The operator selects the plan item and invokes the item's context menu, then clicks on Delete Item. The GUIPlanItem calls the GUIPlan that it is contained in to remove the item. The GUIPlan then calls the Plan to remove the item. The served Plan object will then remove the item and push a PlanItemRemoved event through the event channel. See the diagram GUIPlanModule:PlanItemRemovedEvent for more details on this event.

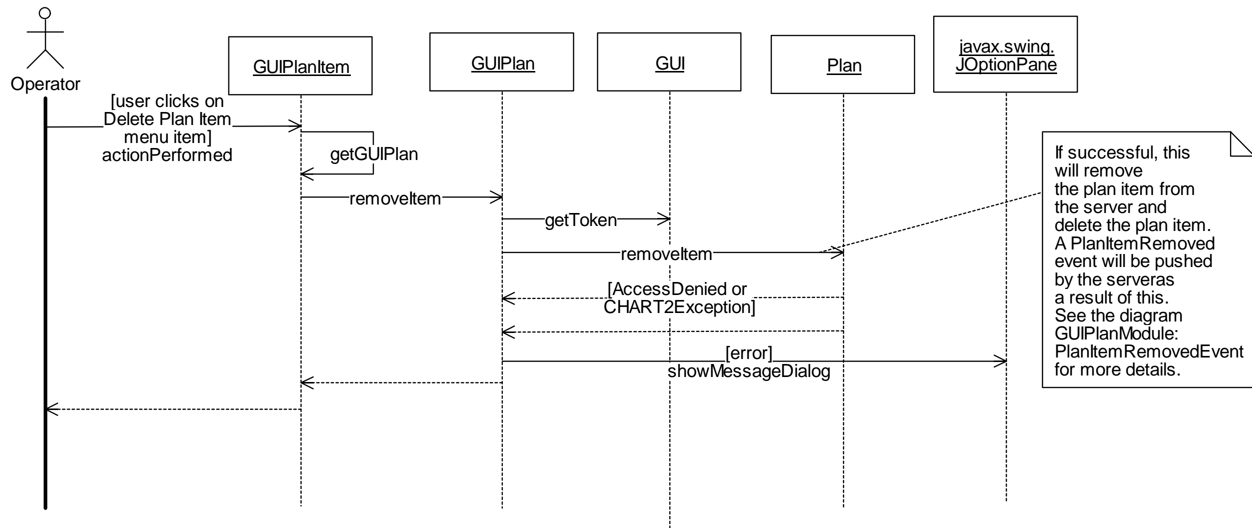


Figure 5-59. GUIPlanModule:RemovePlanItem (Sequence Diagram)

5.60 GUIPlanModule:Shutdown (Sequence Diagram)

When the GUI calls the module's shutdown method, the module disconnects from the ORB to clean up.

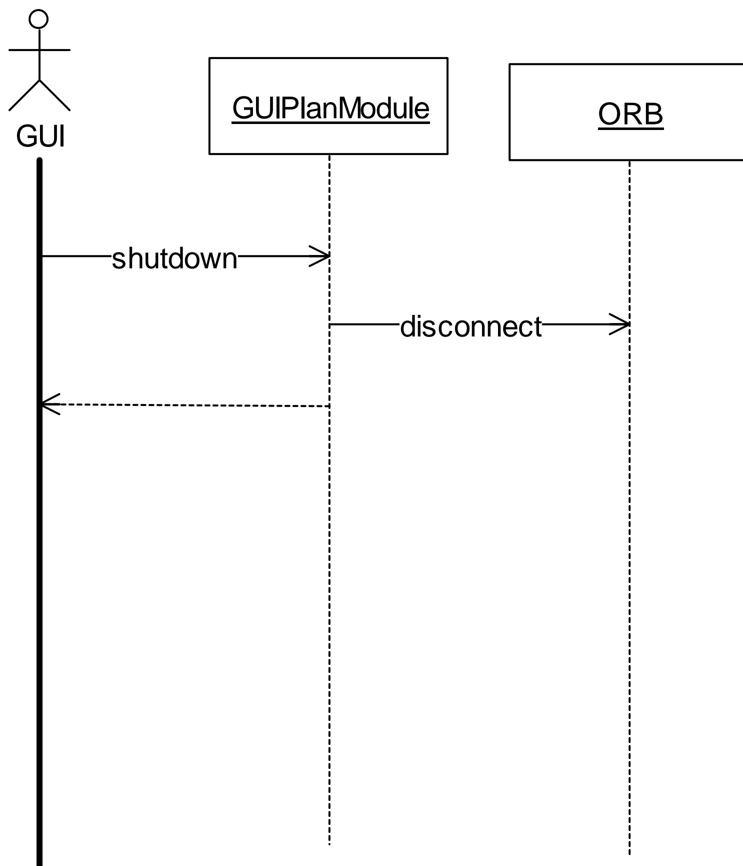


Figure 5-60. GUIPlanModule:Shutdown (Sequence Diagram)

5.61 GUIPlanModule:Startup (Sequence Diagram)

The startup for the GUIPlanModule begins when the GUI calls the startup method. At this time the module connects itself to the ORB so that it can be called as a PushConsumer. It also creates a Navigator group to hold the GUIPlan objects and adds the group to the DataModel. NOTE - Any modules wishing to support plan item creation should attach themselves to the GUIPlanModule in their startup methods.

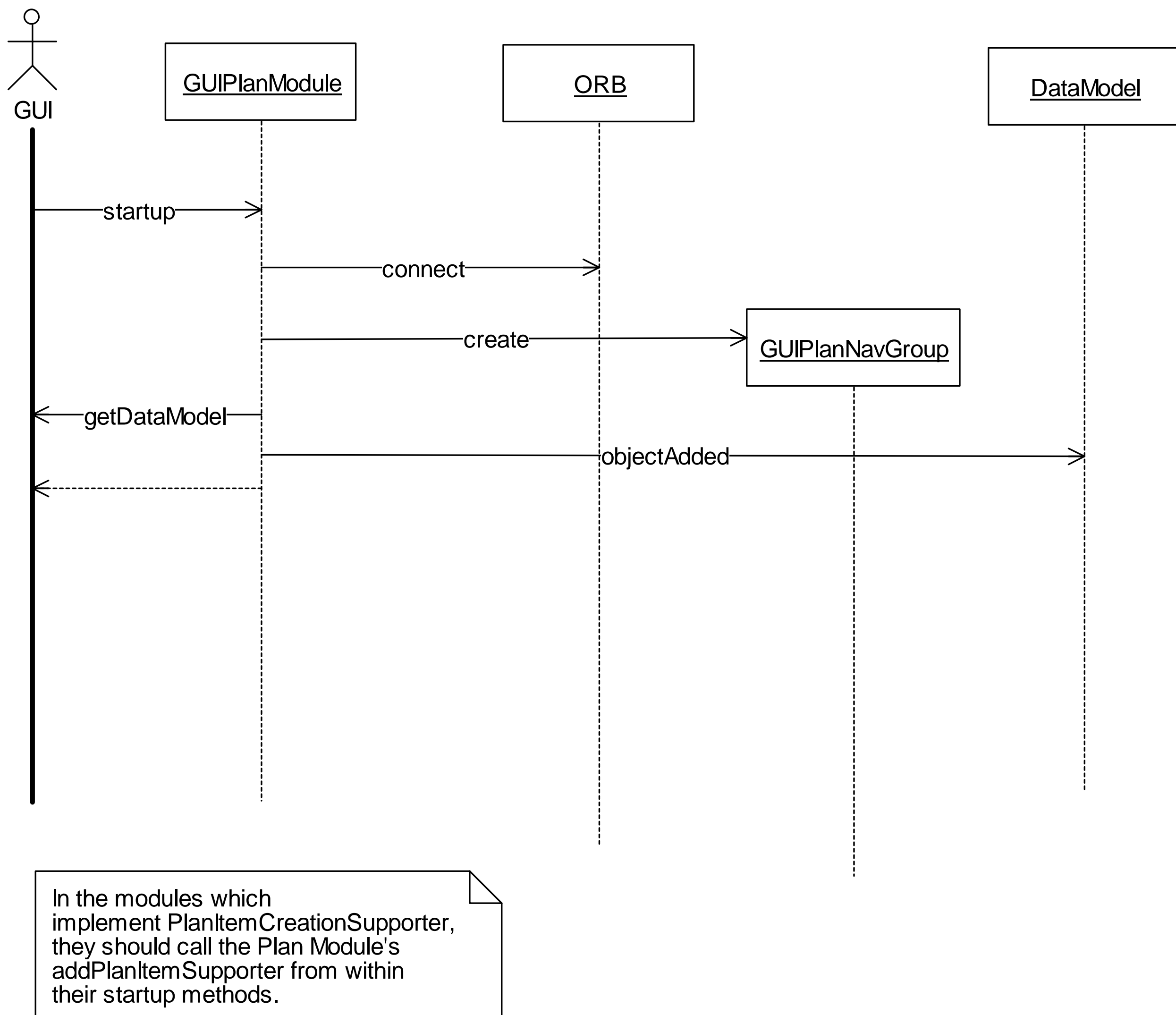


Figure 5-61. GUIPlanModule:Startup (Sequence Diagram)

5.62 GUIUserManagementModule:AddUser (Sequence Diagram)

This diagram shows how a user is added to the system. From the User Configuration Dialog, the administrator clicks on "New User", and the Create User Dialog is invoked. When the administrator clicks "OK," if the new password is the same as the confirmation password, the dialog will call the UserManager to create the user. If the user name or password is invalid, a message box will be displayed and the administrator will be given a chance to correct the mistake. If the user was successfully created, it will be added to the User Configuration dialog if it is still open.

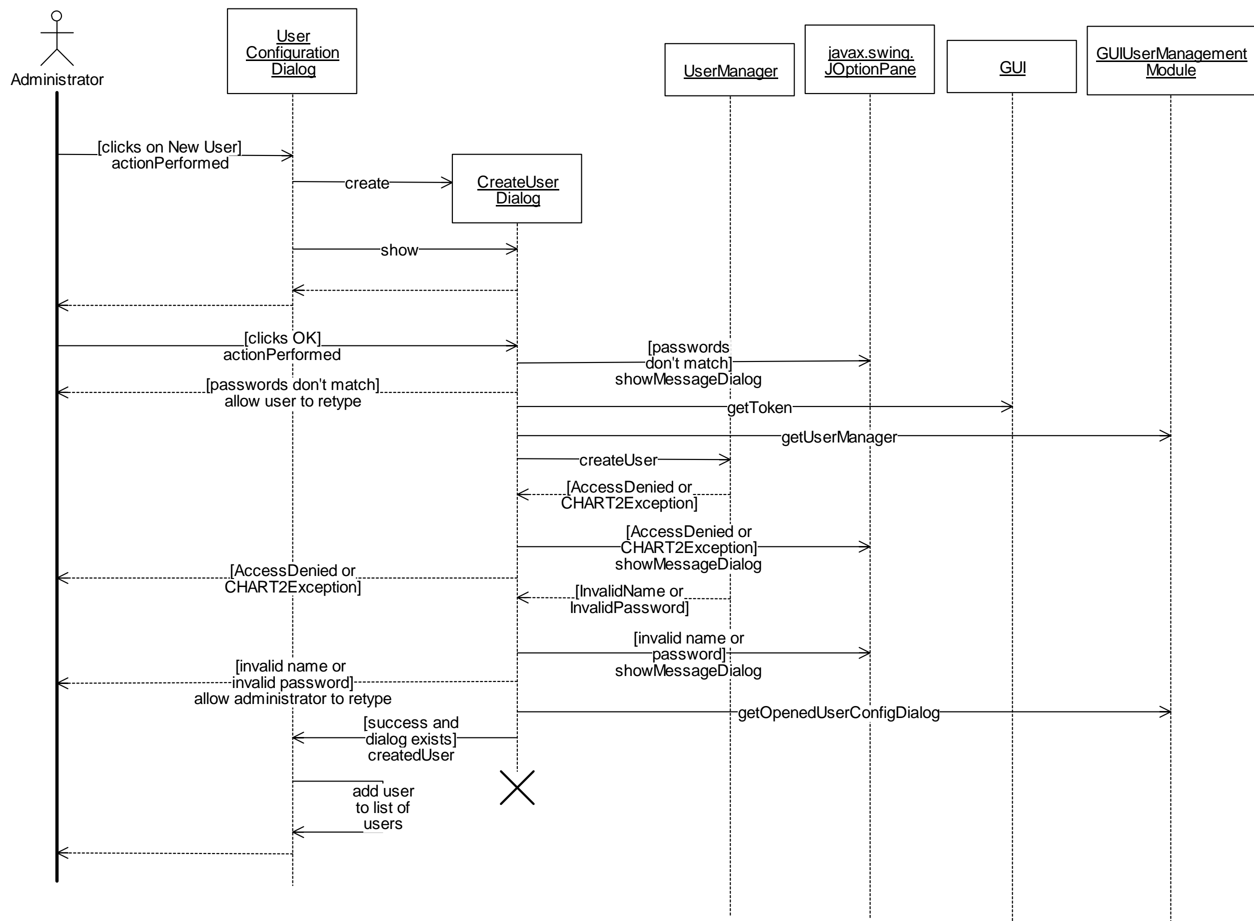


Figure 5-62. GUIUserManagementModule:AddUser (Sequence Diagram)

5.63 GUIUserManagementModule:ConfigureRoles (Sequence Diagram)

This diagram shows how the Role Configuration Dialog is invoked. The adminstrator clicks on the "Configure Roles" toolbar button. The GUIUserManagementModule then creates the Role Configuration Dialog. This gets the Organizations from the trader, and gets all of the roles. It then gets the functional rights for the first role in the list. It displays the roles, functional rights within a role, and organizations supporting a given functional right. If the user does not have the ConfigureRoles right, all editing features will be disabled.

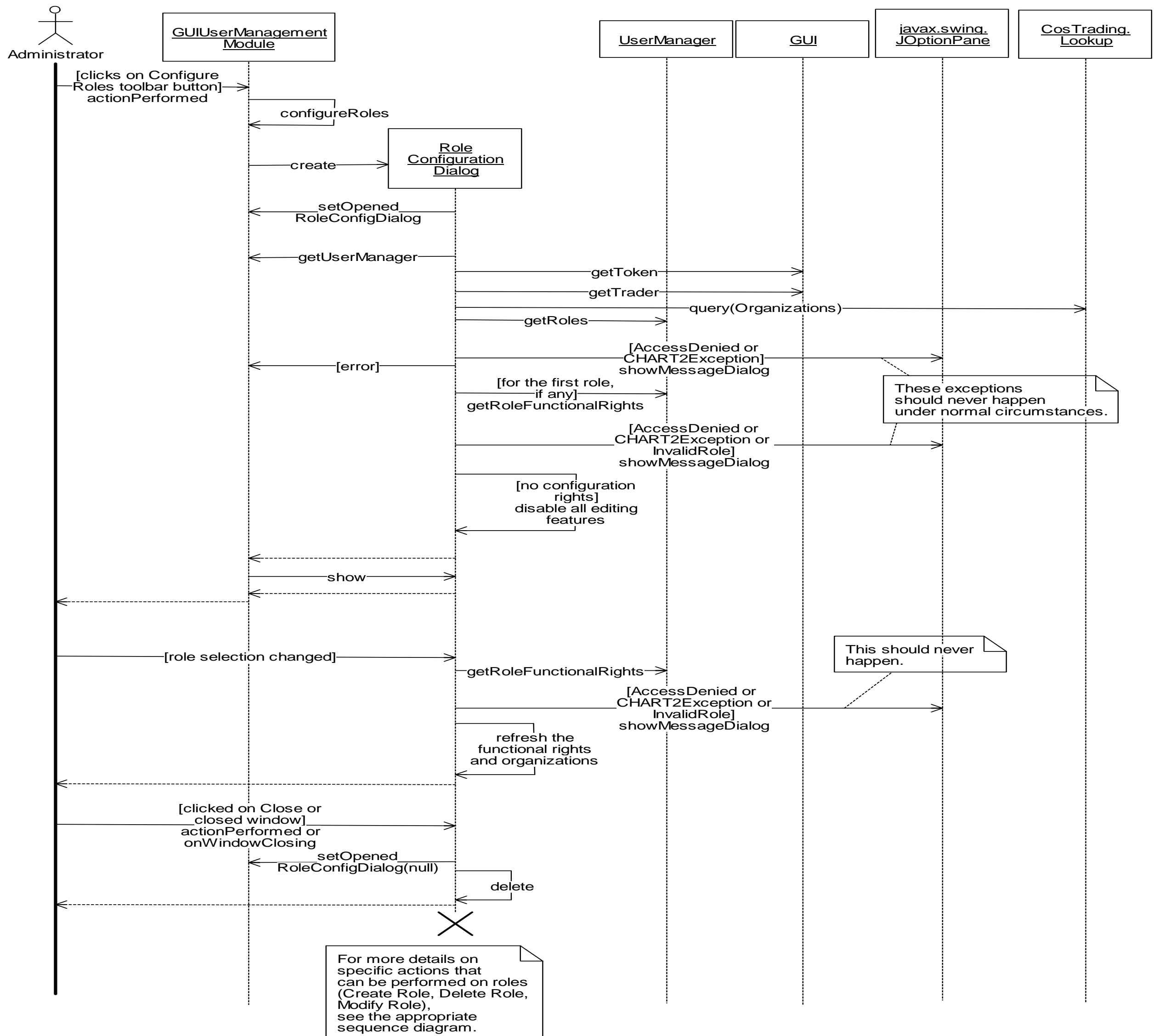


Figure 5-63. GUIUserManagementModule:ConfigureRoles (Sequence Diagram)

5.64 GUIUserManagementModule:ConfigureUsers (Sequence Diagram)

This diagram shows how the User Configuration Dialog is invoked. The user clicks on the "Configure Users" button from the toolbar, which will be disabled unless the user has the rights: ConfigureUsers or ViewUserConfiguration. The GUIUserManagementModule will create the UserConfigurationDialog, and it will call the UserManager to get the users and the user roles. If the user has ViewUserConfiguration rights only, all user configuration functionality in the dialog will be disabled.

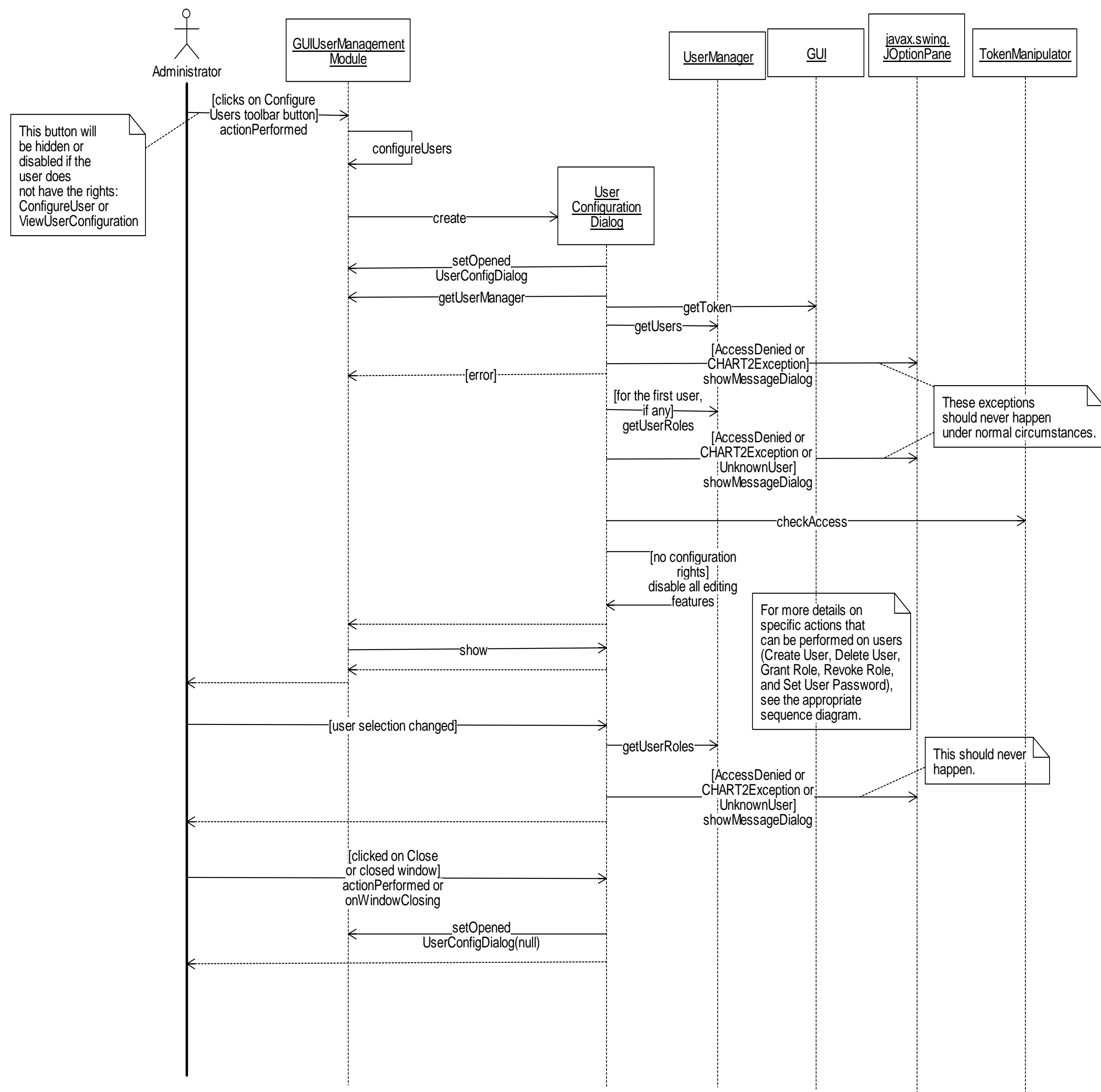


Figure 5-64. GUIUserManagementModule:ConfigureUsers (Sequence Diagram)

5.65 GUIUserManagementModule:CreateRole (Sequence Diagram)

This diagram shows how a role is added to the system. From the Role Configuration Dialog, the administrator clicks on "New Role", and the Create Role Dialog is invoked. When the administrator clicks “OK,” the dialog will call the UserManager to create the role. If the role is a duplicate, a message box will be displayed and the administrator will be given a chance to correct the mistake. If the role was successfully created, it will be added to the Role Configuration Dialog if it is still open.

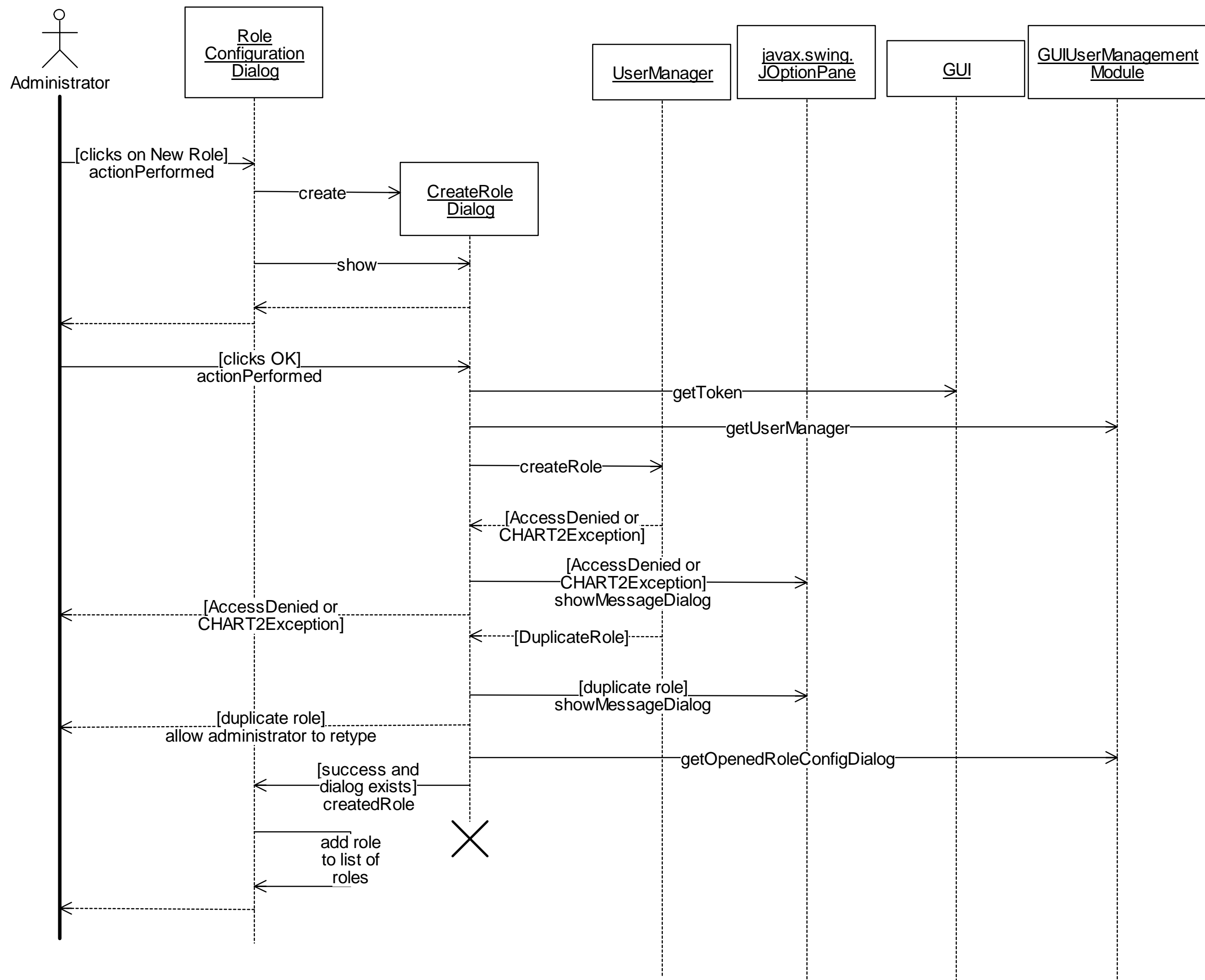


Figure 5-65. GUIUserManagementModule:CreateRole (Sequence Diagram)

5.66 GUIUserManagementModule:DeleteRole (Sequence Diagram)

This diagram shows how a role is deleted from the system. From the Role Configuration Dialog, the administrator selects a role and clicks on "Delete Role." The dialog handles the command and calls the UserManager to delete the role. If successful, the role is removed from the displayed list.

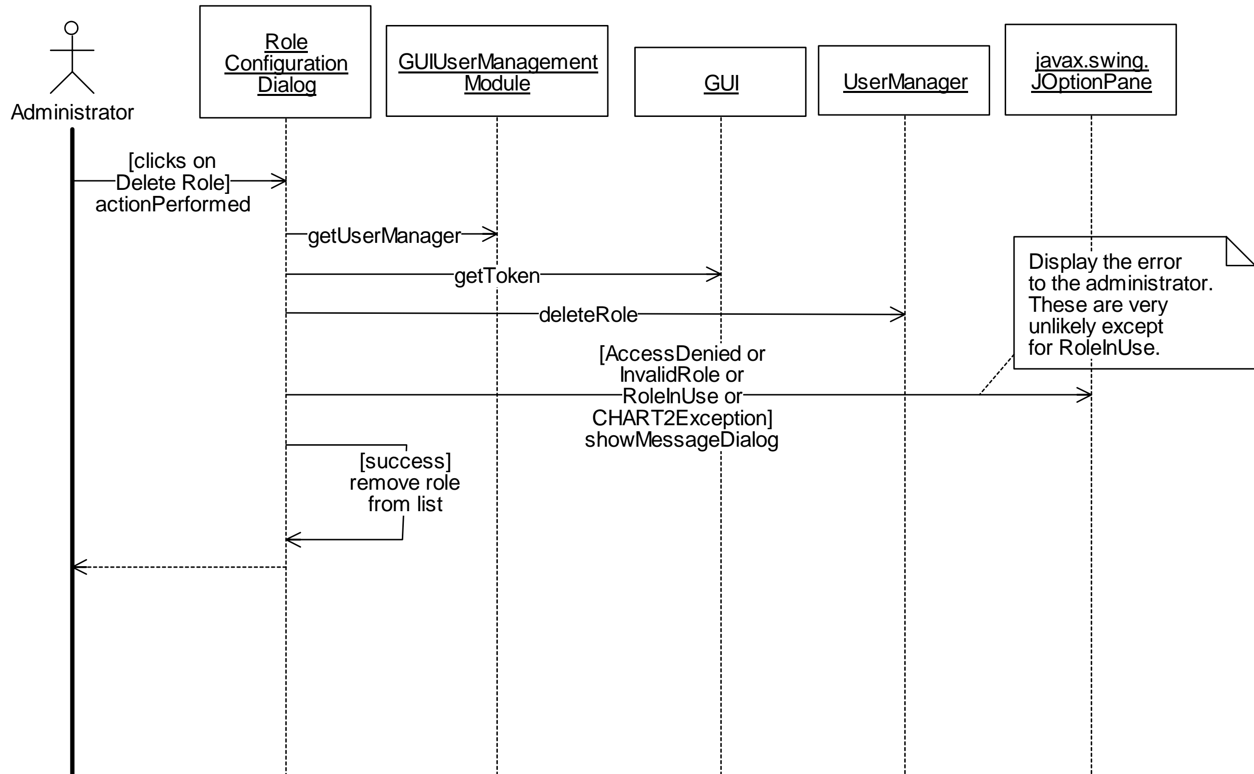


Figure 5-66. GUIUserManagementModule:DeleteRole (Sequence Diagram)

5.67 GUIUserManagementModule:DeleteUser (Sequence Diagram)

This diagram shows how a user is deleted from the system. The administrator selects a user and clicks on "Delete User" from the User Configuration Dialog. The dialog calls the UserManager, which deletes the user from the system. If the user is currently logged in, a message box will be displayed informing the administrator. If the user is successfully deleted, the user's name will be removed from the dialog.

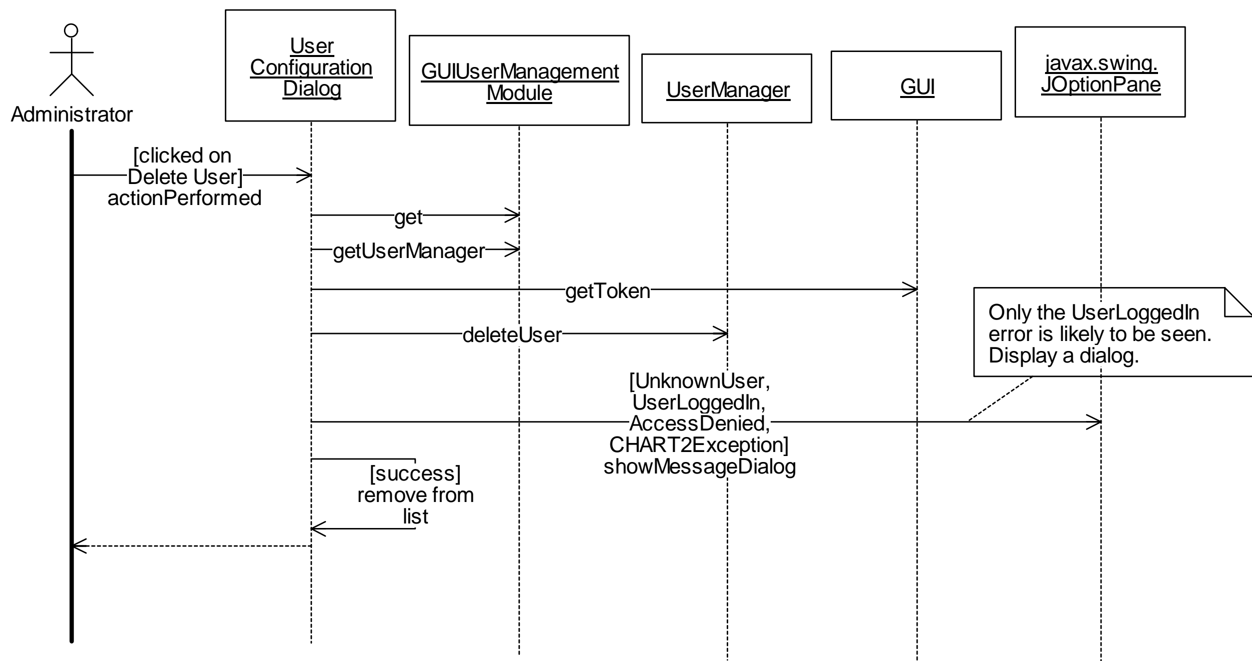


Figure 5-67. GUIUserManagementModule:DeleteUser (Sequence Diagram)

5.68 GUIUserManagementModule:ForceLogout (Sequence Diagram)

This diagram shows how the Force Logout command is performed. The administrator clicks on the Force Logout button on the toolbar. The GUIUserManagementModule then creates a ForceLogoutDialog, which displays all of the users from all of the Operations Centers. When the adminstrator selects a user and hits the Force Logout button on the dialog, the Operations Center will be called to log the user out.

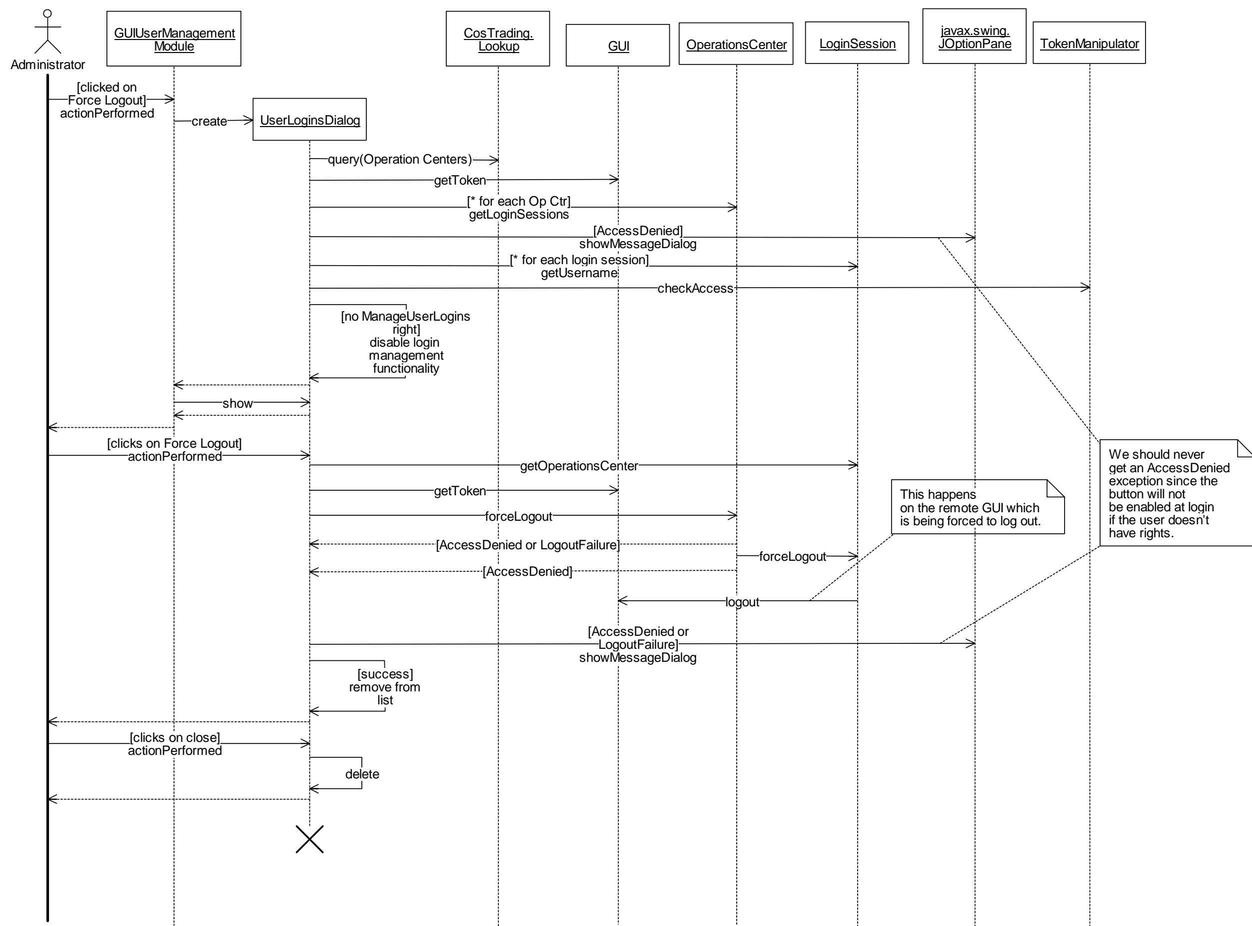


Figure 5-68. GUIUserManagementModule:ForceLogout (Sequence Diagram)

5.69 GUIUserManagementModule:GrantRole (Sequence Diagram)

This diagram shows how a role is granted to a user. From the User Configuration dialog, the administrator clicks on an (unchecked) role checkbox in the role list. The dialog will mark the role as checked, which assumes a successful operation. Then it will call the UserManager to grant the role. On failure, a message box will be displayed and the role will be unchecked.

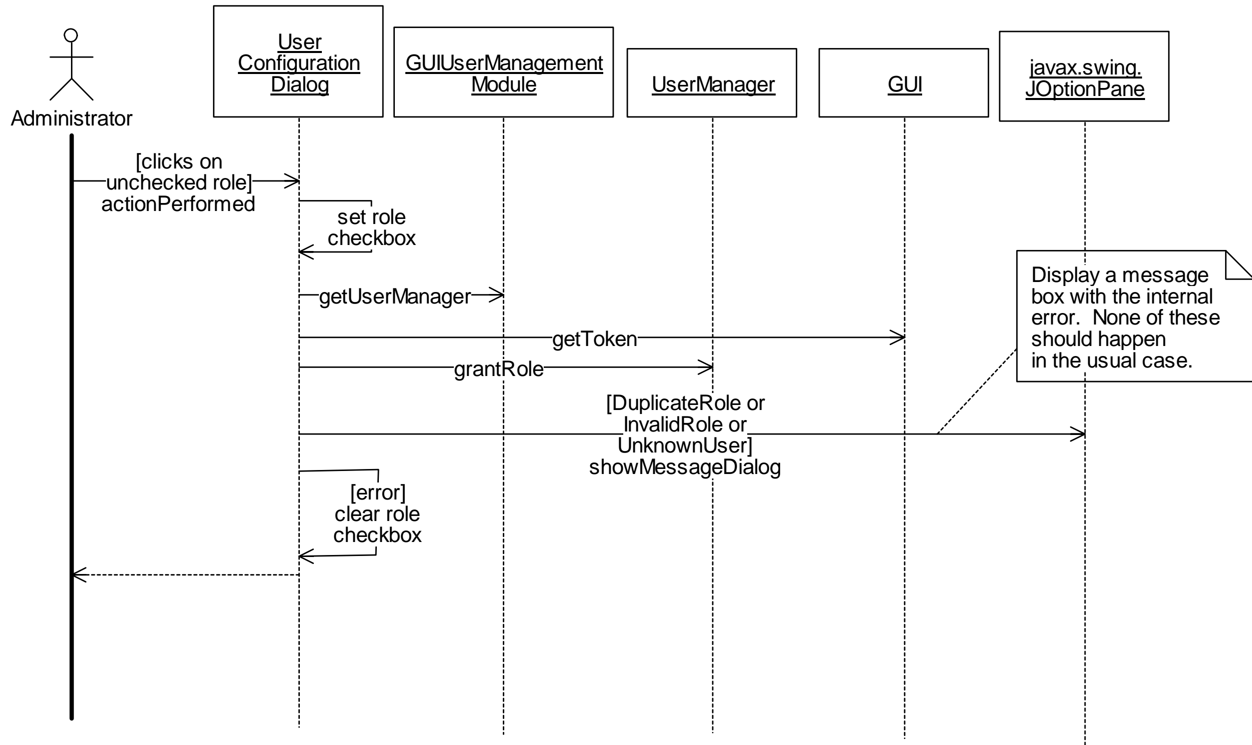


Figure 5-69. GUIUserManagementModule:GrantRole (Sequence Diagram)

5.70 GUIUserManagementModule:Login (Sequence Diagram)

This diagram shows the user-specific initialization that is done at login.

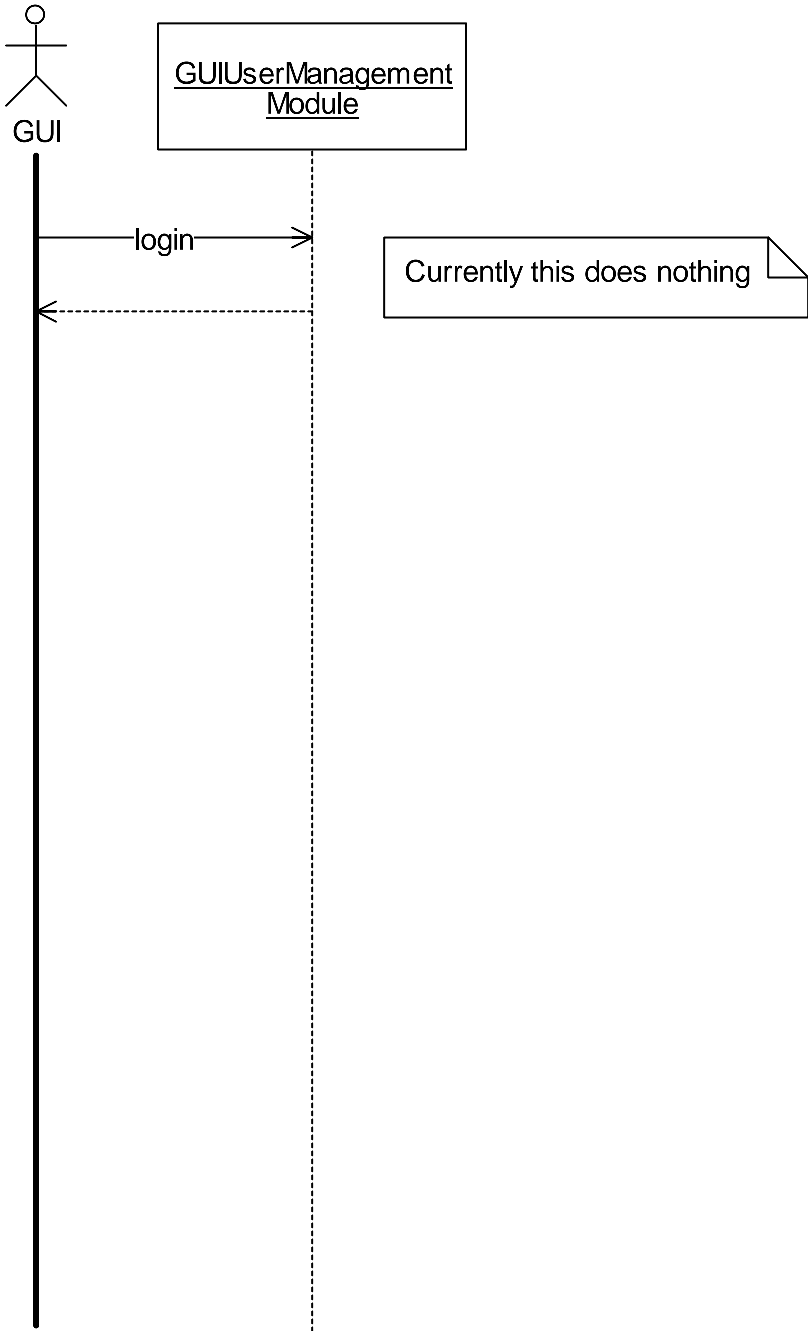


Figure 5-70. GUIUserManagementModule:Login (Sequence Diagram)

5.71 GUIUserManagementModule:Discovery (Sequence Diagram)

This diagram shows how UserManager objects are discovered. The GUI will call the GUIUserManagementModule to discover objects, and the module will query the trader for any published UserManager objects. Then it will try to ping each one until one responds, and if the ping is successful, it will store the UserManager for later use. Once a UserManager is stored, it will be pinged first before querying from the trader.

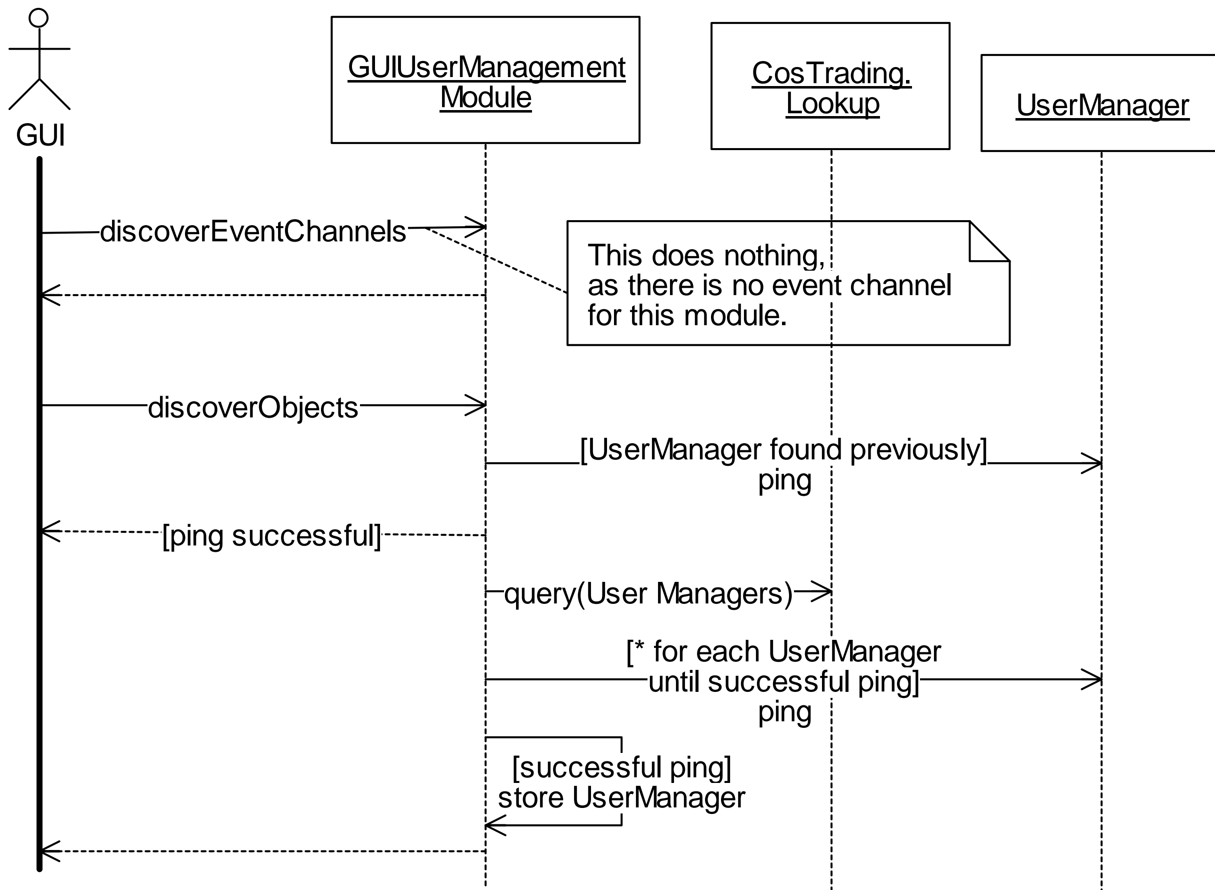


Figure 5-71. GUIUserManagementModule:Discovery (Sequence Diagram)

5.72 GUIUserManagementModule:ModifyRole (Sequence Diagram)

This diagram shows how roles are modified in the system. From the RoleConfigurationDialog, the adminstrator clicks on a functional right or an organization to toggle its presence in the role. The dialog retrieves all of the functional rights from its components, then sets the functional rights by calling the User Manager. If an error occurs, the correct functional rights for the role are retrieved from the User Manager, and the dialog is refreshed based on the correct rights.

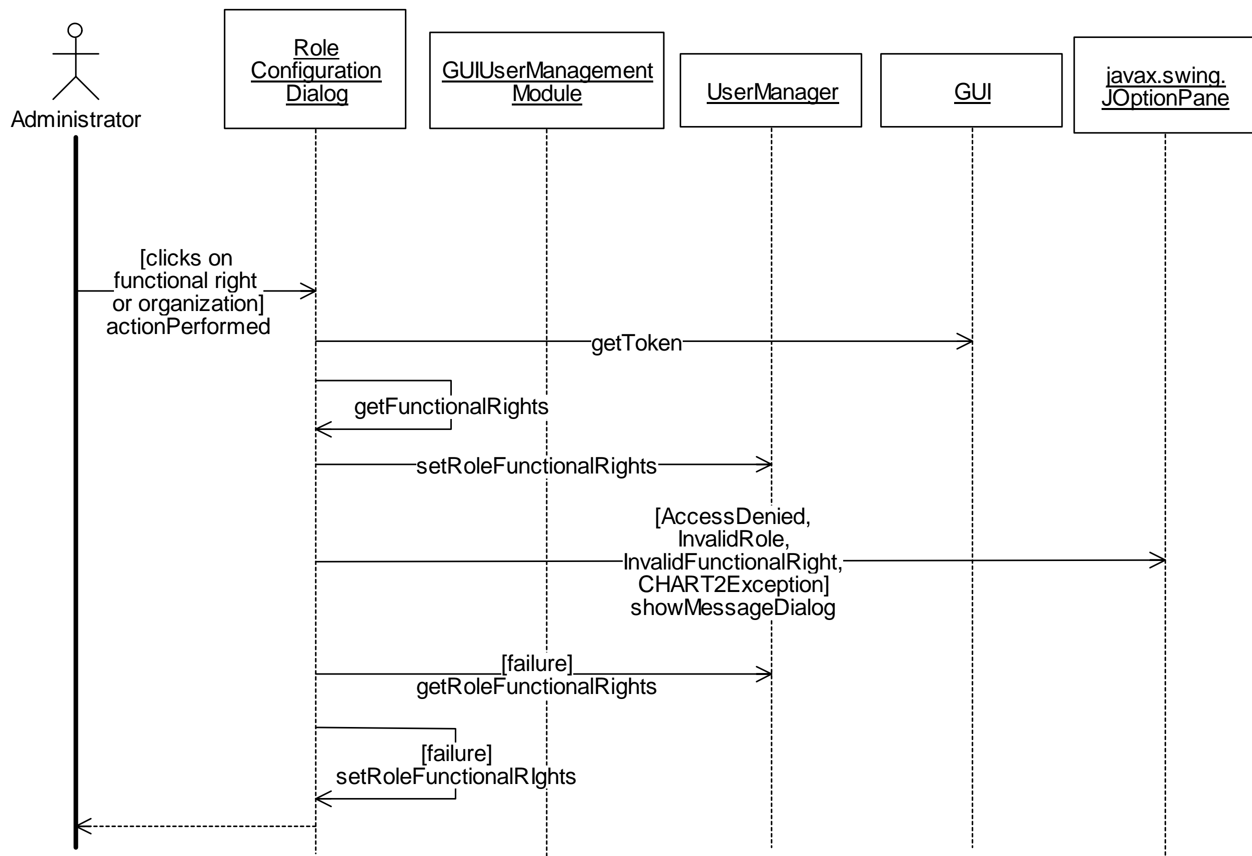


Figure 5-72. GUIUserManagementModule:ModifyRole (Sequence Diagram)

5.73 GUIUserManagementModule:RevokeRole (Sequence Diagram)

This diagram shows how a role is revoked from a user. From the User Configuration dialog, the administrator clicks on a (checked) role checkbox in the role list. The dialog will mark the role as unchecked, which assumes a successful operation. Then it will call the UserManager to revoke the role. On failure, a message box will be displayed and the role will be checked.

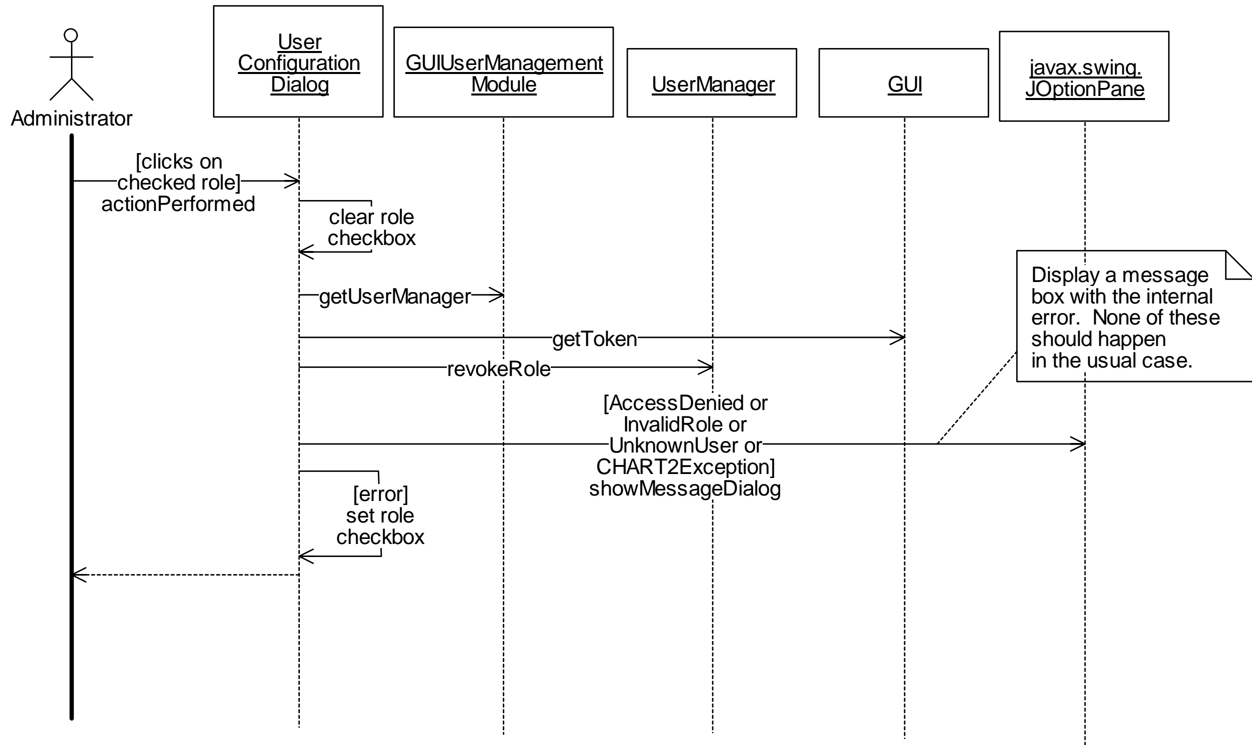


Figure 5-73. GUIUserManagementModule:RevokeRole (Sequence Diagram)

5.74 GUIUserManagementModule:Startup (Sequence Diagram)

This diagram shows the actions performed by the GUIUserManagementModule at startup.

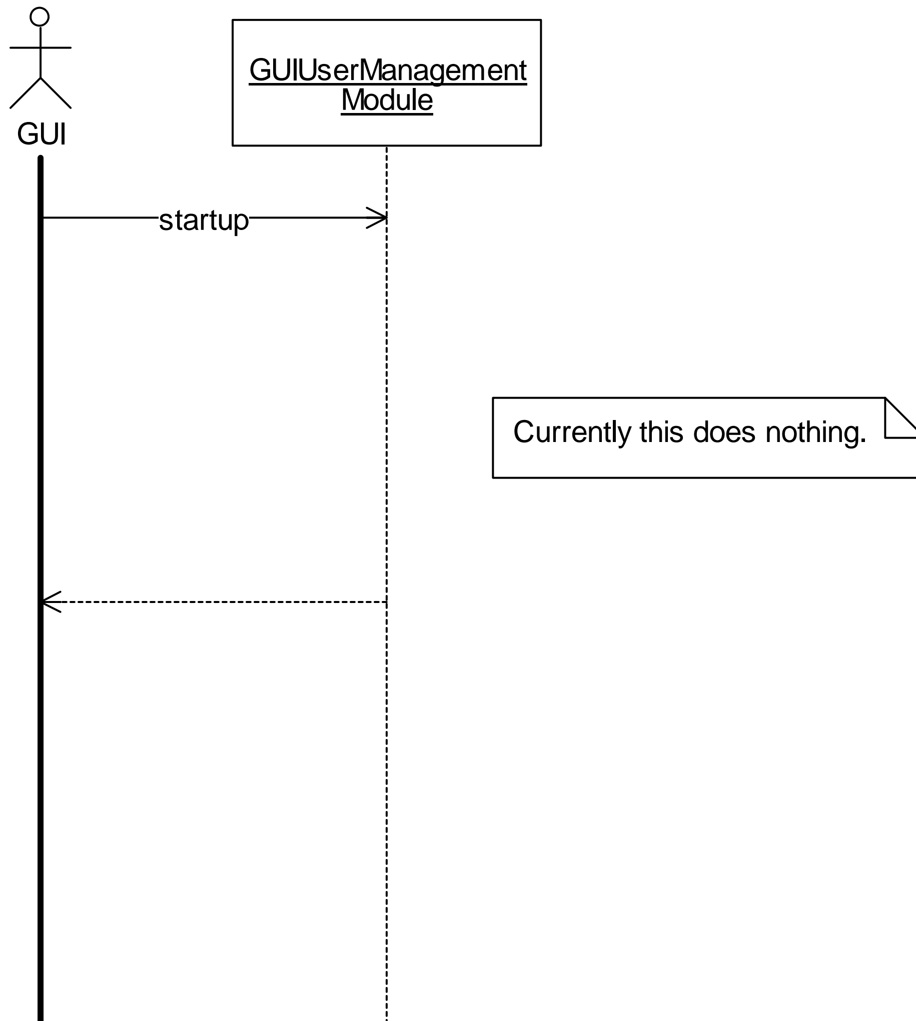


Figure 5-74. GUIUserManagementModule:Startup (Sequence Diagram)

6 GUI Screen Captures

6.1 GUI:ScreenAccess (State Chart)

This diagram shows how all of the major windows in the GUI may be invoked from a user's point of view. The GUI will create the GUIToolBar at startup, which is the main launching point for all second-level windows. The main commands are located on the toolbar, while less commonly used functionality and object-specific functionality is invoked from the Navigator.

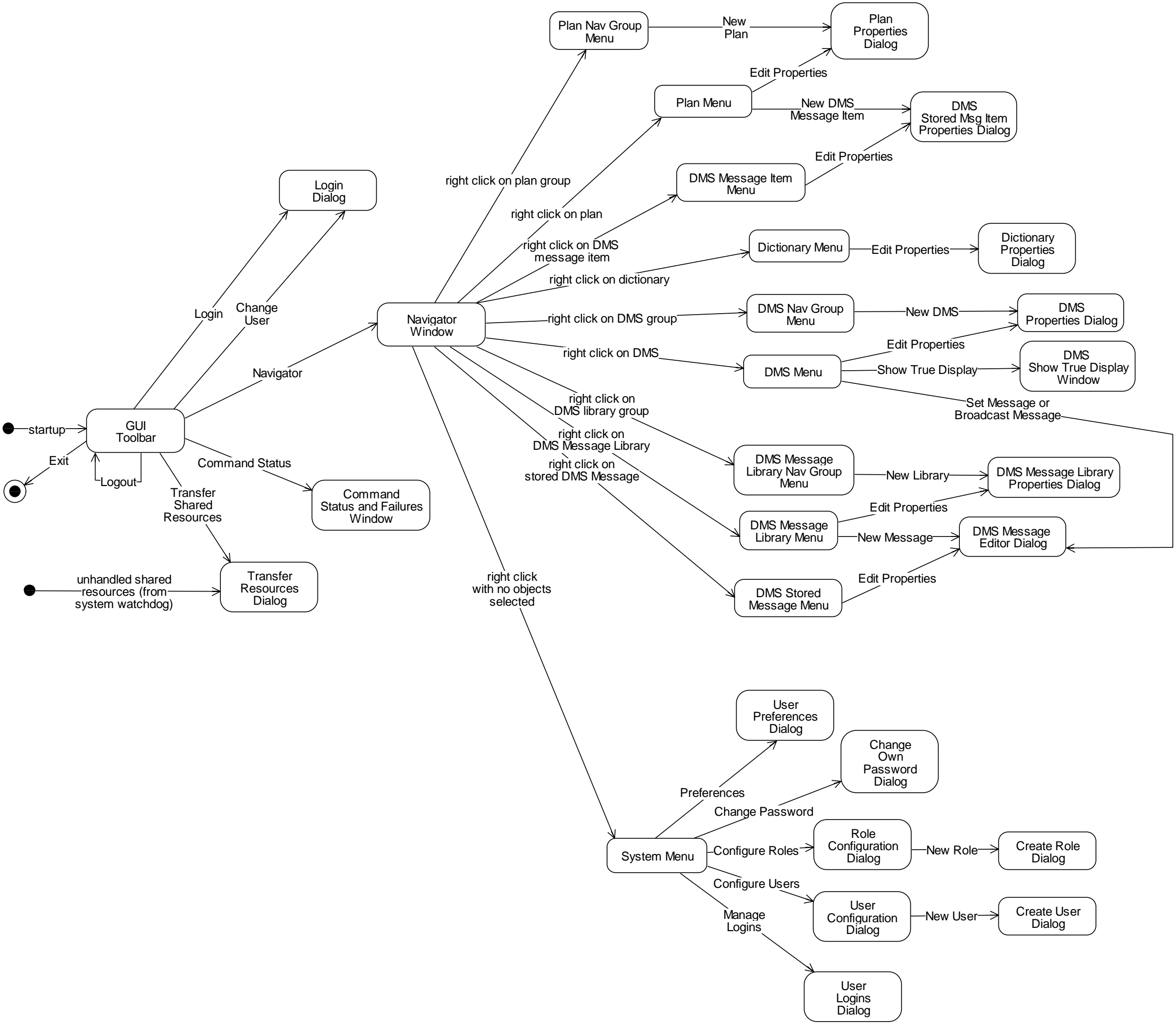
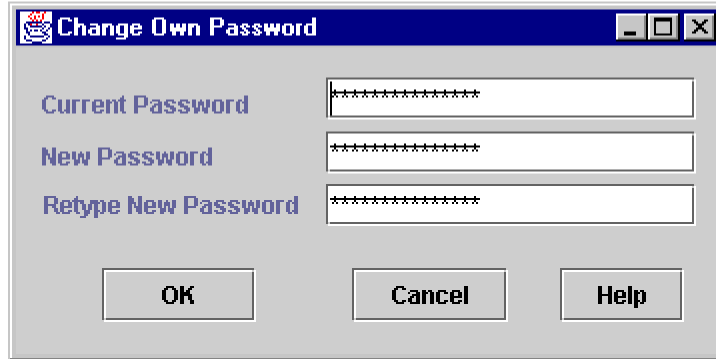


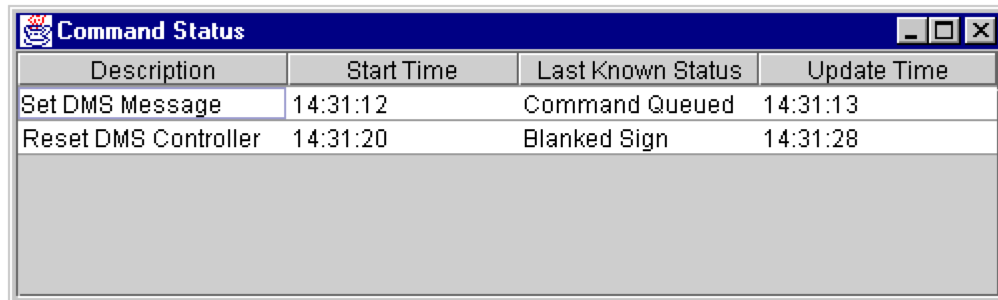
Figure 6-1. GUI:ScreenAccess (State Chart)

6.2 Change Own Password Dialog



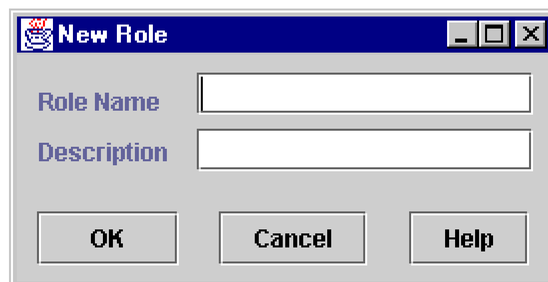
This dialog allows a user to change his/her own system password. The user must specify his/her current password along with a new password to use for future logins. The user must enter the new password twice to prevent typographical errors.

6.3 Command Status View



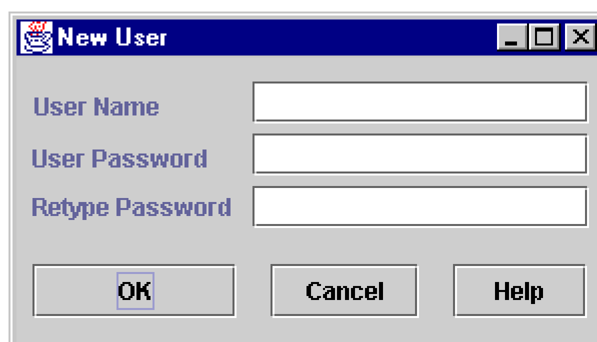
This view shows the current status of all currently running operations. The “Description” column contains a description of the command. The “Start Time” column contains the time/date that the command was issued. The “Last Known Status” column contains the text status last reported by the server process executing the command. The “Update Time” column contains the time/date that the server last updated the status of this command.

6.4 Create Role Dialog



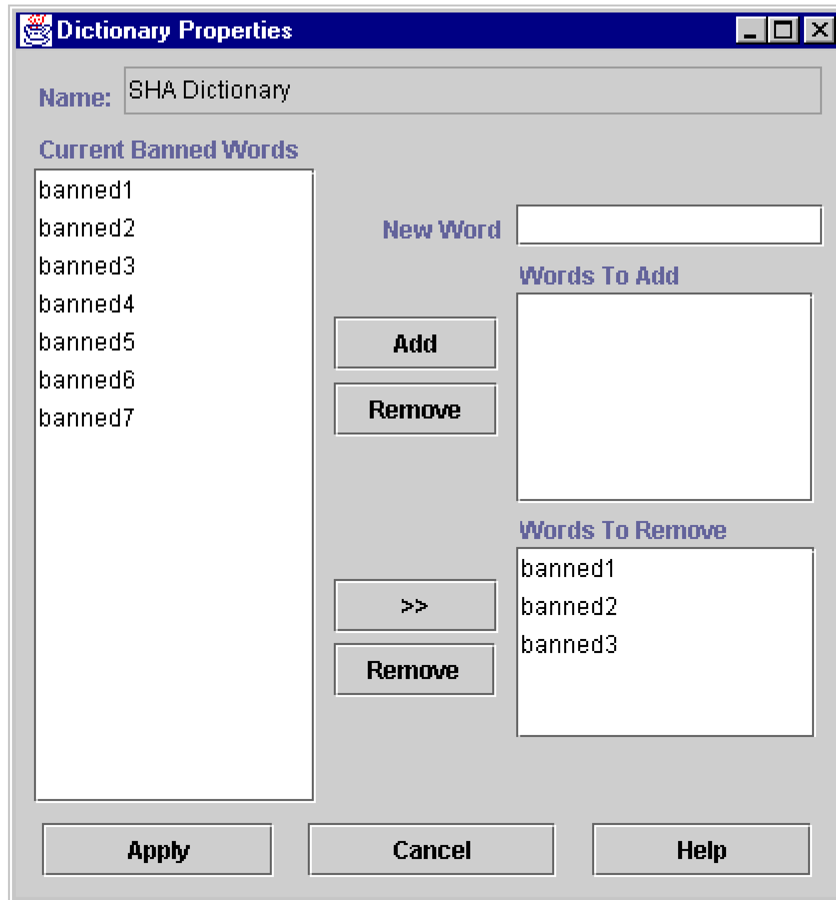
This dialog allows the user to specify the name and description of the role that he/she is adding to the system. This dialog is accessed by pressing the “New Role” button on the Role Configuration Dialog.

6.5 Create User Dialog



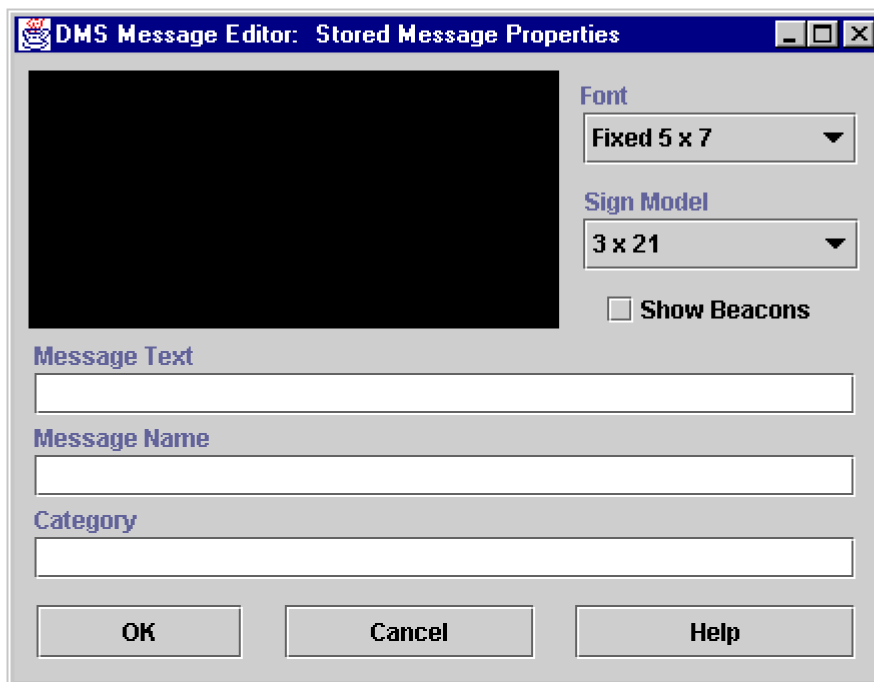
This dialog allows the user to specify the user name and password of the user that he/she is adding to the system. This dialog is accessed by pressing the “New User” button on the User Configuration Dialog.

6.6 Dictionary Properties Dialog



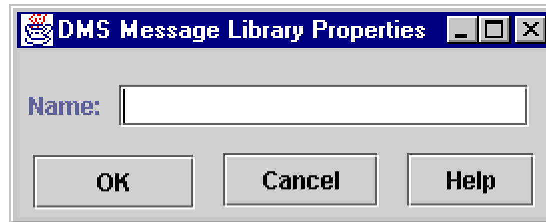
This dialog allows a user to view and alter the current list of banned words in the system dictionary. The "Current Banned Words" list always shows the current list of banned words in the system dictionary. The user may select any subset of this list and mark it for removal by pressing the ">>" button. The user may add a word to the banned words list by typing it in the "New Word" field and pressing the "Add" button. When the user presses the "Apply" button, the words to remove will be removed from the banned words list, the words to add will be added to the banned words list, and the current list of banned words will be updated. The user may press "Cancel" at any time to quit using the dialog. When the user cancels the dialog, any words to add and words to remove that have not been applied (by pressing the "Apply" button) will not alter the system dictionary.

6.7 DMS Message Editor Dialog



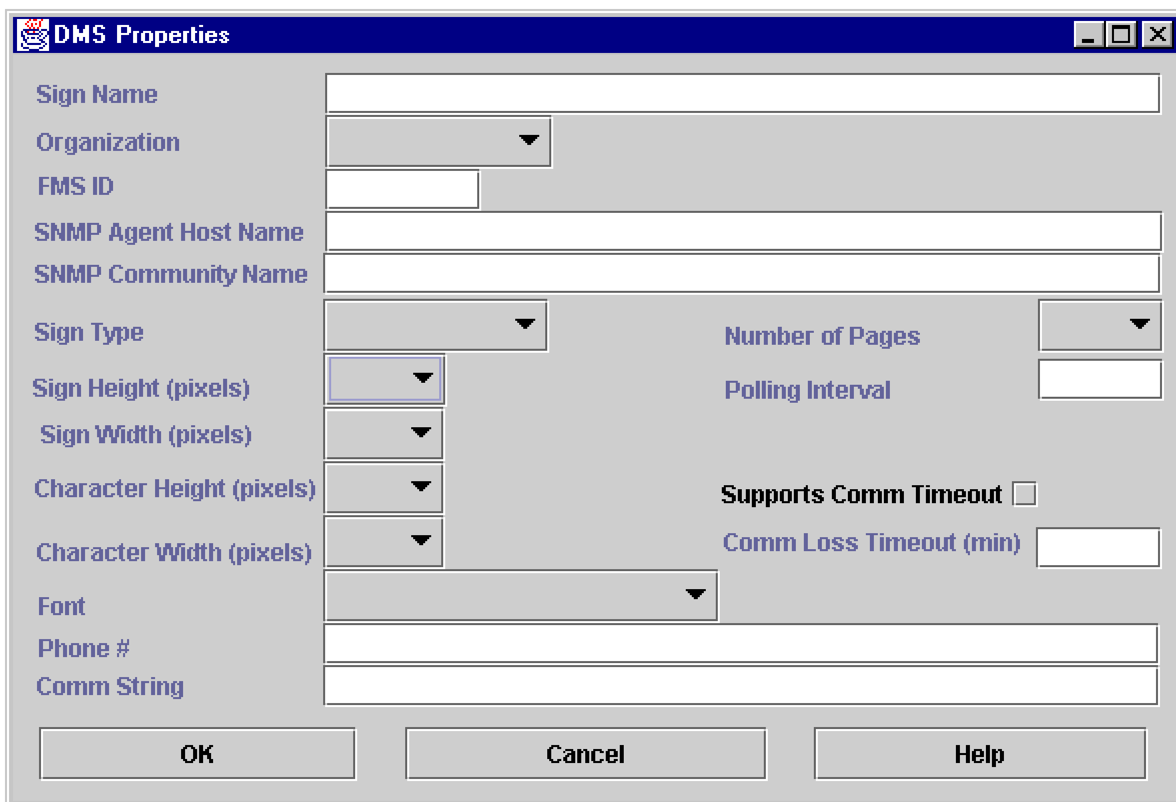
This dialog allows a user to edit the current message on a particular DMS, edit the text in a stored DMS message from a library, or broadcast a message to a number of DMS objects. If the user is editing the current message of a single DMS the “Category” and “Message Name” fields will not be visible and when the user presses OK the message will be sent to the DMS. If the user is editing a message to broadcast to multiple DMS objects, the “Category” and “Message Name” fields will not be visible and the message will be sent to all of the DMS objects when the user presses OK. If the user is editing a stored DMS message all of the fields will be visible and the message name, category and message contents will all be set when the user presses OK.

6.8 DMS Message Library Properties Dialog



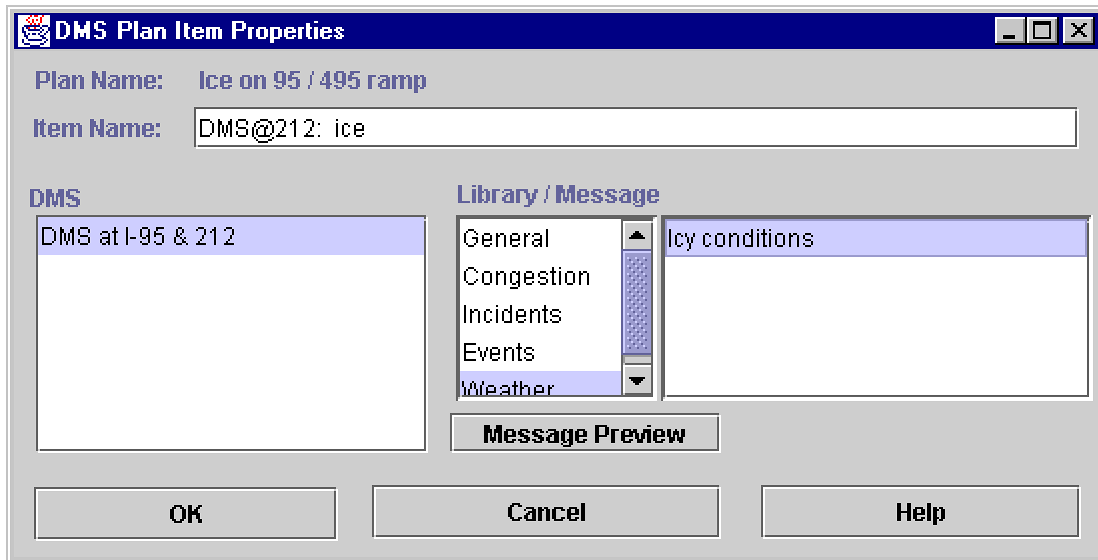
This dialog allows a user to set the name of a DMS message library. It is accessed by selecting "Properties" from the context menu of a DMS message library object.

6.9 DMS Properties Dialog



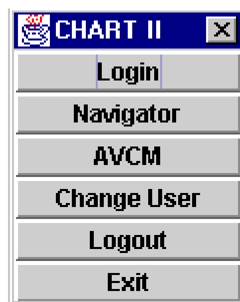
This dialog allows a user to set the configuration information for a particular DMS. This dialog is presented when the user opts to add a new DMS to the system, or when the user selects "Properties" from the context menu of a DMS. When the user presses "OK" the new configuration information is sent to the DMS server.

6.10 DMS Stored Message Item Properties Dialog



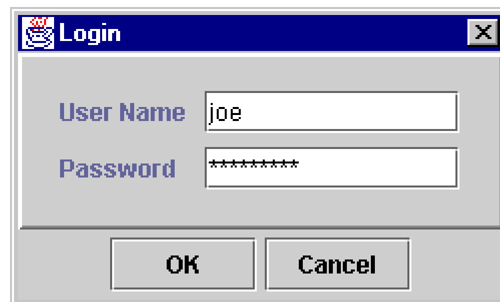
This dialog allows a user to select the DMS and stored message that will be used for a particular plan item. When the item is activated, the selected library message will be displayed on the selected DMS. The message preview button will cause a window to be displayed that shows the contents of the selected library message.

6.11 GUI Toolbar



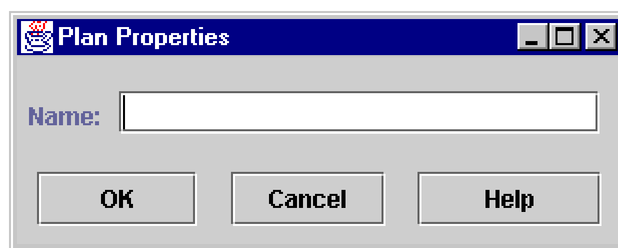
This window is the main window of the CHART II GUI application. It provides buttons that allow the user to perform system functions.

6.12 Login User Dialog



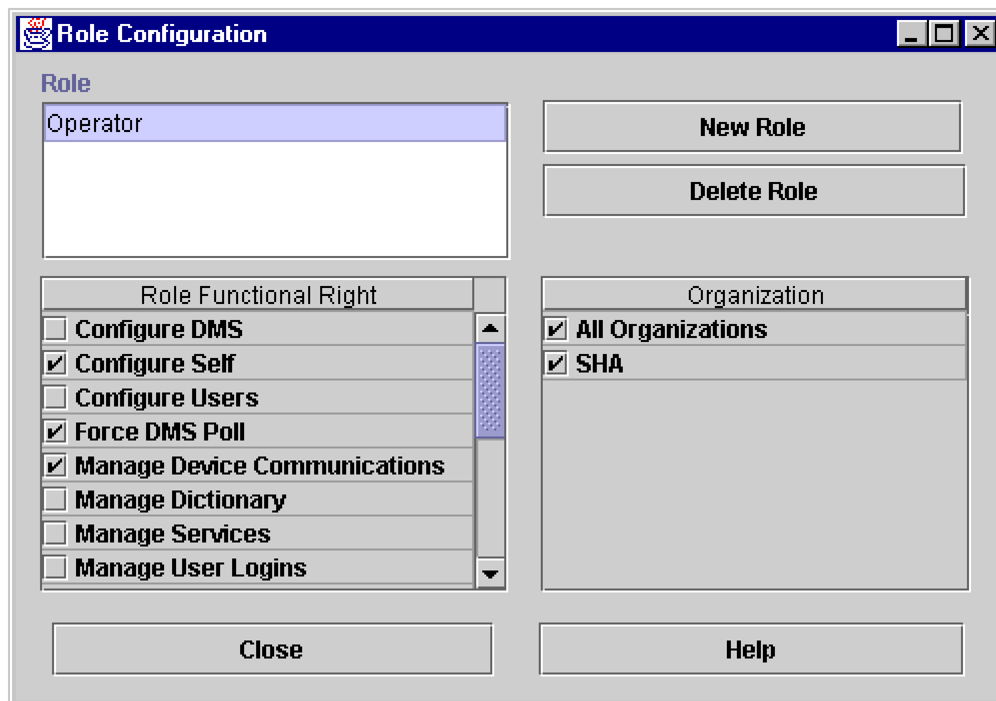
This dialog allows a user to enter his/her user name and password. This dialog is presented when a user opts to login or change user. In the case of a change user command, the user who is logging in for the next shift should provide his/her user name and password.

6.13 Plan Properties Dialog



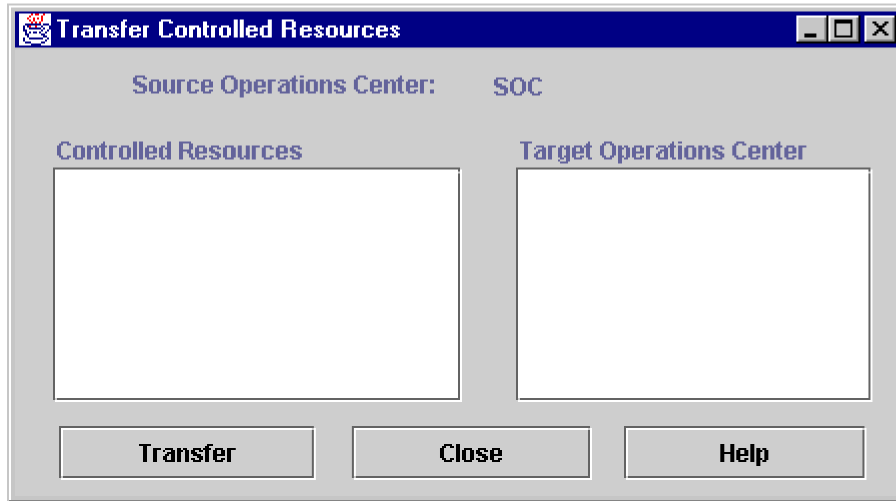
This dialog allows a user to alter the name of a system plan. It is accessed by selecting the "Properties" menu item from the plan object's context menu.

6.14 Role Configuration Dialog



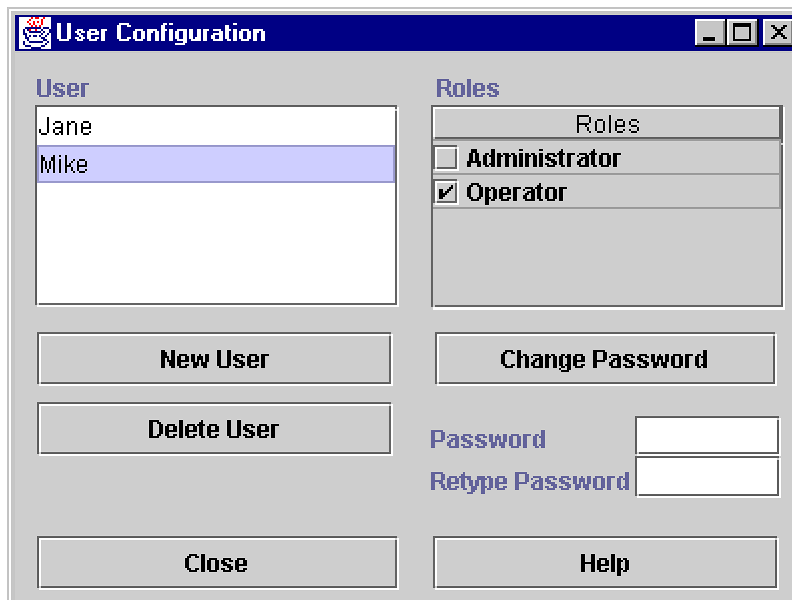
This dialog allows a user to create new roles, delete existing roles, and alter the functional rights assigned to a particular role. The user may select one role from the "Role" list. The "Role Functional Right" list will then update with the current functional rights for the selected role. When the user selects a particular functional right, the "Organization" list will update to provide a list of available organizations for that right. The user may then specify which organizations the right should be granted for.

6.15 Transfer Controlled Resources Dialog



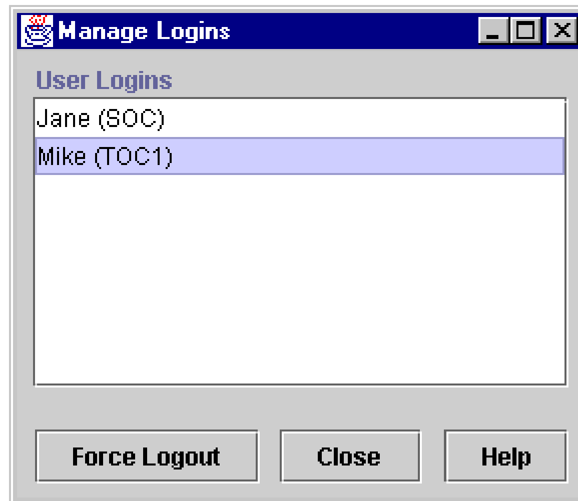
This dialog provides a list of controlled resources for a particular operations center. The user may select a group of controlled resources and a target operations center. When the user presses the "Transfer" button, the selected resources will be transferred to the target operations center.

6.16 User Configuration Dialog



This dialog allows a user to create new users, delete existing users, change a user's system password, and alter the roles that a user is allowed to utilize. The user may select a single user from the "User" list. The "Roles" list will then update by placing check marks next to those roles that the selected user may currently utilize. The user may alter the role assignments by checking or un-checking roles in this list.

6.17 Manage Logins Dialog



This dialog allows a user to force another logged in user to log out of the system. This dialog will present a list of all users currently logged in and their current operations center. Pressing "Force Logout" will force the currently selected users out of the system.