MCT Project Technical Specification

Jack Hodges
August 14, 2008

MCT Project Engineering Team: Jack Hodges, Alan Tomotsugu, Dennis Heher, Irene Smith, Nija Shi, Antonio Si, Alex Voskoboynik

For Internal Distribution Only
NASA Ames Research Center, 2008.

Contents

Abstract

This document serves to define the basic technical requirements for the systems, mechanisms that`will collectively be called MCT, and the associated tools that allow NASA Mission Control application developers to use the MCT Framework. This project consists of a multi-generation design/development effort targeted at replacing the current mission control software with new versions that perform the same kinds of tasks as before but in a dramatically more effective/efficient manner. At the most outward/visible level the new model introduces a new [default] client look and feel but under the covers it changes the entire way applications are built and how mission flight controllers interact with them.

The functional requirements/constraints for the project will be specified by the combination of the functional use cases for the client, the user interface (UI) elements, the storyboards of pages that will implement the use cases pertinent to MCT mechanisms, systems, and tools. This document represents the engineering response to the requirements defined in the MCT UE Architecture Specification, vers 1, dated 10/31/2007.

This document is both long and complex. The table below provides some suggested guidance for who should read what parts of the document.

Who Should Read	Document Section	Pages
PI, HCI	Abstract	5
	Discussion	7 - 22
HCI, QA	Abstract	5
	Discussion	7 – 22
	Framework use cases	30, 46, 91, 98, 108, 122, 131, 151, 171, 182, 189, 200, 209, 217, 230, Appendix A
Engineering Managers	Abstract	5
	Discussion	7 – 22
	Component Model	26 - 31
	Component Toolkit	42 - 49
	Component Library	89 - 91
	Information Semantics Manager	94 - 98
	User Platform	106 - 108
	Configuration Management	120 - 124
	Event Handling	129 - 131
	Identity Manager	135 - 138
	Rule Engine	146 - 151
	Constraint Validation	166 - 168
	Composition	169 - 171
	Data Validation	178 - 182
	Messaging	187 - 189
	Persistence Management	196 - 200
	Policy Management	206 - 209
	External Services	211 - 217
	Localization and Internationalization	225 - 227
	Packaging and Deployment	228 - 230
Development Engineers	Entire document (as appropriate)	

Chapter 1 Project Overview

This chapter presents the MCT project engineering effort from an outward perspective. That is, in terms of the outward objectives, constraints, and requirements. It addresses these in terms of possible approaches and selected solution strategies, and goes on to address the mechanisms necessary to achieve the objectives within the stipulated constraints and requirements. The chapter ends with an outline of the chapters to follow.

Objective

Mission Control Technologies (MCT) is a framework for developing NASA mission control applications. MCT libraries consist of systems and tools supporting a wide variety of mission types, along with a flexible way of using applications that makes it easier for flight controllers to work with mission control applications.

Anatomy of a Mission Control Application

The manner in which constraints/requirements shall manifest themselves in the MCT architecture and implementation choices are best appreciated by looking at an example application. How does it currently work and what are its limitations? How can integration make developing it and other applications better, easier, faster?

Using the Titan telemetry monitoring position as an example, the user interface displays a large amount of data from a single data source (ISP) using multiple MSKView and RTPlot instances. Each of these instances is autonomous from the next, and the manner in which data is described using these applications is unary; the data can be viewed in a single way and cannot be manipulated in any manner. To see data for a particular object in a different way a different application must be launched. The user must configure and control multiple independent applications. The layout and control of these applications is offline, manual, awkward, and cannot easily be shared without sharing the console itself. This approach also requires multiple engineering teams to support development of the separate applications, resulting in long lead times to test and deploy changes.

It would be much easier on the user if he could view objects in different ways, at will, w/o changing the application. It would also be nice if the user could change the layout/orientation/size of objects to suit a specific event, to drill down into the data to view it more deeply, or to compose object views in ways not currently present in the interface, and have those compositions remembered for future use; all in real time.

A monitoring application is intended to provide the ability to monitor space-borne sensor telemetry. Sometimes a Titan (or other position) must view information about the sensor the telemetry is associated with, the device the sensor is attached to, or even diagnostic processes involving the sensor or device. MSKView and RTPlot can display telemetry values only, and it is up to other data sources and applications to provide abstracted conceptual information to a console position, further complicating the computational environment of the console controller.

It would be nice if all information about a mission, both conceptual and state information, were organized by a single information repository and made available to mission applications at any level of abstraction needed by the console controller on an as-needed basis.

The user interface for Titan is itself brittle in that it cannot be easily applied to other missions or even to other monitoring positions. This means that every station/mission must be developed independently and that any changes are costly to make and maintain.

It would be nice if the underlying object models and their UI counterparts were flexible enough to support Titan but also to support any other telemetry or mission profile. This would require a different, generic, model for how to connect views to data sources, how to represent, configure, and lay out user interfaces, how to validate data and events, and how to relate data to metadata.

Titan consoles display information representing data objects that cannot change. For some missions this is probably critical, but for others it may be necessary or even imperative that the data models or even the mission definition change after the application is deployed, so it would be nice if the underlying object models and the user interface be capable of dynamically adapting to changes in object definition without rebuilding and recertifying the software.

Overview of Problem

Applying the example case above, there are four motivations for designing/developing the MCT Framework and associated component library:

- Current telemetry monitoring systems are brittle and inflexible. Users need systems where they can view information, or aggregate information, at different abstraction levels and in different ways, without the need for specialized (non-integrated and inflexible) applications for viewing the information or having to predevelop the viewing methods. This can reduce the overall size and complexity of the system, maintenance, and reduce training (and retraining) times.
- Current mission control systems acquire data from disparate sources with specialty software, but adding access to different data types using different networking strategies should be easily supported. This integrated and flexible approach should enable mission application developers to use a single framework for many if not all mission control applications.
- Currently mission control systems must be accessed in special locations, making collaboration and interaction both difficult and awkward. Mission applications should be available anywhere and anytime as long as the person requiring access has the right permissions, plays the right roles, and the connection is adequately secured.
- The models used to represent mission conceptual information must be suitably versatile that they can adapt to changing NASA needs and requirements, even during a mission and including the mission itself.

Although the monitoring application identifies general motivation for MCT, the collection of other mission control tasks, such as command and control, planning, and analysis simply makes a stronger case for the MCT approach. In fact, there is no reason why the MCT Framework couldn't provide the backbone for creating generic application user interfaces.

General Requirements of the Project

MCT is a project that is intended to address a wide range of real world problems, and in particular to be demonstrated within the scope of telemetry monitoring applications. The requirements that must be met by any architecture, design, implementation, and deployment fall into two categories: User Experience requirements, and System requirements.

User Experience Requirements

There are 5 categories of general User Experience requirements that will inform the basic architectural approach: (1) fine-grained components, (2) object function based on object usage, (3) composability and design, (4) interoperability and adaptability, and (5) core MCT components.

Fine-Grained Components

The notion of fine-grained componentry is emergent. In traditional systems, a functionality is defined and the objects and views associated with that functionality are developed as an application. In object oriented interfaces, a component is defined, with both its model and view aspects, and it can be combined with other objects in different ways without a specific application or implied functionality. There are limits to how flexible these combinations should or can be, but the focus is shifted from building applications to build components. There is also a range of real vs. implied separation of model from view, but that distinction isn't required at the architectural level.

Object Function Based on Object Usage

The notion of component is associated with what a user associates with something they think that they should be able to manipulate on a screen. Under the covers has model properties and it has view properties but the user sees them as a combination. In MCT there is a requirement that a component take on a view that is appropriate to the context in which the component is being applied by the user. That is, the component's view is context-sensitive. Moreover, the component's functionality should be malleable to the point where it can take on functionality it wasn't explicitly imbued with on creation, at run time, if needed.

Composability and Design

A fundamental aspect of an object oriented GUI is that components are building blocks, but in concert with this idea must be the answer to the question "what can we do with components?" Composability is an action of aggregating (or disaggregating) components into different, or even new, visual contexts. When component templates are composed into an environment the composition would be considered design, since something is produced where previous nothing existed. Since a container must in a design context be considered a containee it is imperative that component be capable of becoming either.

Interoperability and Adaptability

An essential aspect of MCT is that objects be sharable with others, whether those others are performing the same or different tasks, and that, to the user, the shared object seem as though it is the only one of its kind.

Core Components

MCT is intended to provide a visual baseline from which other visualization contexts can be constructed. As such, it makes use of containers and other object types that do not exist in previously-defined widget sets. It is imperative to this approach that all GUIs be constructed from this baseline or core set of components, and that the look and feel of such components be dictated by user experience requirements to the greatest extent possible.

System Level Requirements

MCT visualizations will be used by people and so there are user centered requirements that drive its design. At the same time, MCT will be used in environments where there are system-level requirements that will additionally inform the design. There are ten categories of system-level requirements that can inform the MCT architectural decisions: (1) certification, (2) maintainability, (3) extendability/evolvability, (4) flexibility/interoperability,

(5) modularity, (6) configurability, (7) support for 3rd party applications, (8) scalability, (90) performance, and (10) rapid prototyping.

Component Certification

A reality of NASA is a finite budget, set of resources, and time. It is imperative that the MCT design guarantee that components be discrete enough to minimize recertification. This requirement mandates that models be certifiable, that views be certifiable, and that processes that operate on components are certifiable. In this manner, only those items that change must be recertified, rather than the whole system, which should reduce recertification times dramatically.

Minimum Maintenance Footprint

As part of the budget reality, a requirement of the MCT project is that the maintenance requirements be minimized. This can be translated into an interaction of several metrics but they are all associated with reducing software complexity and the number of build/recertification cycles the software must undergo in order to extend framework functionality. The most common ways to reduce complexity are associated with standard object oriented mechanisms (encapsulation, inheritance, polymorphism), and these must always be applied. Additionally, if decoupling functional subsystems can be achieved, and if the way the framework manages interactions between functional subsystems can be designed to limit dependencies, then the framework complexity is reduced. Another way to reduce complexity is to isolate functional subsystems from the specific manner in which the functionality is provided, through adapters. This approach enables the framework to adopt new mechanisms without changing the internals. Finally, if declarative mechanisms can be employed then it is possible to modify or add functionality without changing the codebase. Along with declarative mechanisms usually comes a central processing mechanism, and this is also a simplification in its own right. Employing each of these strategies can reduce software complexity and thus reduce the maintenance footprint.

Easily Extendable/Evolvable

Software is most easily extended when functional capabilities are discrete and reusable. If for example some of the business logic associated with how components interact, which is very traditionally very brittle, can be removed from component code, then component functionality is more easily extendable. If the logic itself is broken down into reusable logic elements, then the logic becomes more easily extendable.

Flexible/Interoperable

Flexibility is associated with sharing functionality and component definitions across mission profiles and this is really at the crux of the Constellation project and information architecture. If component definitions can be built into a commonly shared repository, along with the instance definitions, then anyone using MCT should be able to share these components. If these definitions can be removed from the code then the code becomes much more flexible.

Modular

All software systems benefit from modular design and implementation. Modularity is associated with functionality, so it is desirable to divide the overall functional task into systems, packages, and classes that are themselves responsible for specific tasks but are

also, through their design APIs, autonomous with respect to other functional units. This is, of course, essential to building maintainable code.

Configurable

Configurability takes on several flavors in modern software. It is important to be able to configure which adapters to use for a particular subsystem, which subsystem components to use, to configure attributes of those components, and to configure user-specific attributes. The greater the degree of configurability the more flexible the system will become, and this is an important capability for MCT.

Support 3rd-Party Data Sources

MCT must support the acquisition and integration of realtime telemetry, audio, and video data, and repository data from many sources and types. It is important that these sources be controllable and isolated from the internals of the framework.

Scalable to 100,000 Real-time Parameters

From a telemetry-monitoring point of view MCT must be able to support the full library of telemetry parameters. It is impossible to view more than about 1,500 values in any given view but some of those values may be composites/aggregates of many telemetry parameter values, and it must be possible for flight controllers to construct views from arbitrary collections of telemetry parameters.

Performance

It is essential that MCT provide a user experience that is at least as responsive as current MCCS applications support.

Rapid Prototyping

One of the design considerations of MCT is composability and reuse of composed aggregates as design entities. This requires the ability to use composition to construct both coarse and generalized components that can be shared and reused, and the capability must be integrated into the way MCT works for all users.

Constraints

Using a new way of representing component views, the user should be able to view and use any existing data object available, and to view, use, and compose (i.e., aggregate) these objects in any way they are capable of being viewed, used, or composed. This interface style must be configurable, customizable, and highly adaptive while also being highly scalable. The associated framework must satisfy the following thirteen constraints:

- The framework/implementation must be capable of supporting all data formats, networking protocols, and existing NASA data sources in a plug and play manner.
- The framework/implementation must support rapid application prototyping, development, release, and maintenance.
- The framework/implementation should be able to extend its functionality by using open source, standardardized, communications protocols wherever possible.
- The framework subsystems and associated components should be loosely coupled, have discoverable semantics, and thus support a service-oriented architectural approach.

- Access in the framework/implementation should be controlled by permissions and role-based identity management.
- The client interface response time (i.e., framework implementation performance) must support real-time mission control requirements.
- The components and conceptual models (ontologies) should be highly distributed to support interoperability and data redundancy.
- The framework/implementation should eliminate the problem with client localization strings which is pervasive across country borders and cultures.
- Applications should be easily constructible and customizable by users, or role groups, to add/remove functionality and interface components.
- The framework/implementation should support data type validation.
- The framework/implementation should support behavioral (i.e., business rules) validation.

Possible Solution Strategies

The constrained scope of the MCT framework narrows the possible implementation paradigms significantly. The needs to support a wide variety of data formats and interaction protocols, maximal configurability, and decoupling suggests the use of a declarative language such as XML. The constraint satisfaction requirement suggests a rule-based approach which is in line with the use of semantic web technologies.

The most general approach decouples the component representation and services from their content sources, their data sources, and from their GUI representations. This requires a framework that distinguishes and isolates network communications from local processing, and isolates the transformation from generic declarative (e.g., XML) formats to local implementations in one place. It enables object representations to reside anywhere and in any format without the local system being aware (except through performance).

Within the contexts of these constraints there is a great deal of flexibility in design choice. For example, should an off the shelf messaging layer such as ECF (Eclipse Communications Framework, Mantaray, or ActiveMQ) be used or should one be developed to an appropriate API. The same can be said for the component model development. Which rule engine should be used, what rule representation language should be used, and what semantic query engine/language should be used (among many) needs to be decided based on functionality, flexibility, performance, cost, and ease of use/integration. It is the role of the framework to define the appropriate interfaces for these systems and services, and to provide a reference implementation that demonstrates and grounds the framework functionality.

Proposed Solution Strategy

The solution strategy that provides the front-end flexibility required for client applications and meets the other project constraints can be divided into two general components: (1) application development tools, and (2) the underlying framework which supports application development. The framework/infrastructure can be partitioned roughly into three layers: (1) UI-specific components and mechanisms/services, (2) model-specific components and mechanisms/services, and (3) communications/services-specific components and mechanisms/services. These layers can be roughly matched to a Model/View-Controller (M/VC) paradigm, where the View component maps to the GUI-

specific components, the Model component maps to the model-specific components, and the Controller component maps to a model of component interaction. The framework is intended to work as follows:

- The MCT framework elements are initialized and managed by an executive subsystem called the User Platform, particularly for authentication, component lifecycle, semantics, communication, and representation. The remaining framework components are initialized while the user's view is loaded and configured.
- An object/data server delivers the baseline mission policies, configuration, GUI, object, and constraint models (formatted in XML and adhering to an ontological model of supported components and parameters) to the client host. These files completely describe the user baseline view being constructed. The object/data server must work in failover mode to guarantee availability and performance.
- Another server delivers the content model content based on ontological descriptions. This is done using an RDF/RDFS/OWL query engine. The ontology server must work in failover mode to guarantee availability and performance.
- A data source delivers the real-time data associated with the mission to be made available using a data model proxy to the UI representations managed through a component registry. Both synchronous and asynchronous data types must be supported in MCT. Data validation is performed according to mission policies at acquisition time.
- An information manager parses OWL descriptions and creates MCT representations which are managed by the User Platform. As part of this process, the application-generic components are mapped to their application-specific representations.
- The page is rendered on the client host in the locale/time zone of choice.
- The user performs normal client-type actions (e.g., button presses, selections, text input, tab selections, etc.). These are handled by the MCT representations that interact with the application-specific components.
- The interface enforces GUI-specific data constraints, and catches and handles any violations between GUI-specific data constraints. Changes in data are cached locally and persisted to the network at configurable intervals.
- Changes to the interface model are mediated by MCT composition. Viable changes are cached locally and persisted to the network at configurable intervals.

Project Deliverables

MCT Engineering will produce an MCT reference implementation which meets the spirit of the use cases outlined in the MCT UE Architectural Requirements (currently Version 3). MCT Engineering will also implement artifacts that comply with the current MCT Functional Specification to the extent supported by the framework and available resources with a focus on architectural integrity.

Outline

The remainder of this document presents the architectural and functional requirements of the proposed MCT framework and reference implementation. These include the mechanisms needed to support application development, the tools that must be

developed to support application development, the implementation of those tools, and the design of the mechanisms used to implement the project. Each will be presented as a separate chapter following a general introduction to the framework. Within the context of each chapter the following general questions will be answered:

- What is the function of this framework component
- Why is this framework component needed
- What are the constraints and requirements that inform this framework component's design
- Where does this framework component fit into the larger MCT functional picture
- How flexible/autonomous must this framework component be
- What design approaches are feasible, what approach is recommended, and why
- What use cases must be supported by this framework component
- General workflow for this framework component
- Framework component design overview and block diagram
- Appropriate and annotated UML to enable development of a reference implementation (class diagrams, state diagrams, sequence diagrams, etc.)

Introduction to the MCT Platform Architecture

Mission Control Technologies (MCT) is the project name for the tools and associated framework used to implement a NASA next-generation mission control visualization environment. As such, it supports the implementation of existing functional capabilities, such as those supporting a Titan Telemetry Monitoring position, but can also be used to implement other telemetry monitoring positions and capabilities other than telemetry monitoring. While the framework is being designed and implemented, the baseline features, and look and feel for client applications, are also undergoing revision.

The MCT implementation will be multi-phased. In its earlier stages, many of which have been completed at the time of this writing, several demonstration prototypes will be presented that illustrate the approach. Stage 2 wraps up the lessons learned during prototyping and presents a beta-level framework architecture and set of visualization capabilities ready for test deployment. Stage 2 is currently underway. In stages 3 and beyond remaining features will be added, the system will be performance tuned, documentation and training materials will be written, and tools required for developing particular visualization environments will be implemented.

General Approach and Background

Fundamental to the MCT approach is the notion of a simple composable element that has no outward functionality but can take on the functionality of standard building blocks that have model and view characteristics. Support for such a dynamic component requires that there be no preset definition of its capabilities. Essentially the concept of component is that of a shell, with the potential to function in any way, but not knowing until run time what functionality it will implement. The architecture of such a system must thus be defined around the component itself, since both “lower” (those relating to the component structure itself) and “higher” (those relating to how components acquire and make use of functionality) architectural levels depend on this organization. A general structure for a system that can address this requirement is depicted in Figure 1:

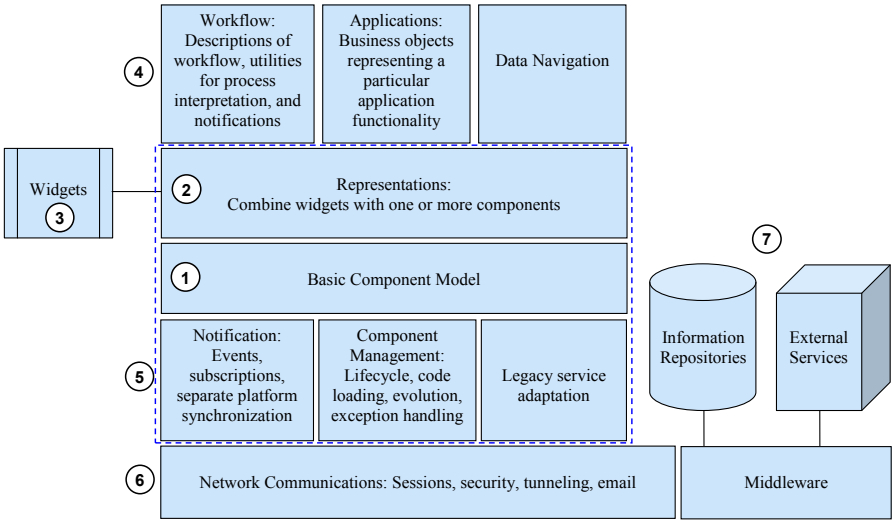


Figure 1: General component-based system architecture including application, widget, data, and external services contributions.

The central (component) layer (at **1**) represents the foundation of the system. Moving up in the diagram, all higher-level components are themselves composed of components. Representation components (at **2**) are comprised of a GUI-level façade (from widgets, at **3**) and data-level (or model) components. Representation components are used at the application level (at **4**) to construct or modify application interfaces. Moving down in the diagram, components are managed and interact (at **5**), and this business layer sits on top of a general communications and middleware layer (at **6**) which provides access to information repositories and external services (at **7**).

Although this figure represents a viable approach it doesn't provide a direct mapping to the MCT model. It does show the general interaction/dependency between functional components that would utilize a component model and should be appreciated at that level of granularity.

MCT is intended to provide a component-based framework for constructing mission control applications. In this capacity, it provides for a general component model, representation library/dictionary of prebuilt components, and set of component-based services and subsystems. These are outlined by the dotted square in the general component-based architecture above.

If we think of the MCT Framework as providing the mechanisms needed to create applications and interact with external resources, then as long as the MCT framework provides a single interface to application developers the MCT framework can be thought of as a black box. Moreover, the MCT Framework is designed to be used in a collaborative environment in a peer-to-peer manner, thus providing access between the components managed by the framework and across the platform network. This arrangement and way of viewing the MCT framework is shown as a general topological diagram in Figure 2:

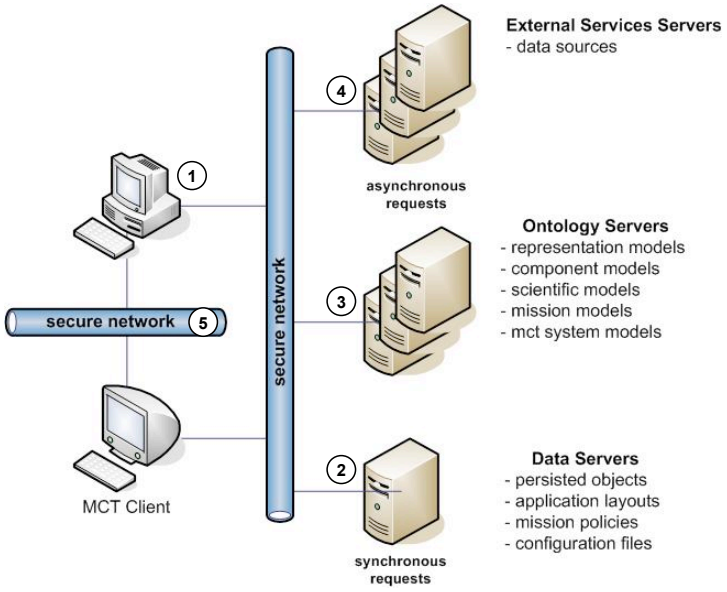


Figure 2: MCT framework as a client component in a network topology.

This figure illustrates the MCT framework as encapsulated in a client host (at **1**). In this figure the client host represents the general component framework from Figure 1. Five types of network interactions are described. First, when the application/framework is launched, application policies, configuration files, user interface descriptions, and persisted component instances must be loaded, generally from some external data source (**2**). These are generally read-only operations and can use synchronous requests. Second, periodic changes to application objects are persisted back to the data server and these operations are also synchronous in nature. Third, when component semantics are initialized, or change, the framework must synchronize the local models with those on a network of ontology servers (at **3**). These interactions can be synchronous or asynchronous. Fourth, to populate UI representations with data, the framework must interact with a network of service providers (**4**). Since these interactions can be updated at any time they can be either synchronous or asynchronous interactions. Finally, the components that are managed by the framework can interact with one another in a publish/subscribe manner or they message one another more directly in a peer-to-peer manner (at **5**). These communications can be synchronous or asynchronous.

Mechanisms/Subsystems

Sixteen interacting mechanisms/subsystems together provide the functionality required to develop mission control applications using MCT and thus comprise the MCT Framework functional capabilities:

1) **Component Model**: This mechanism provides the component functionality required for the MCT framework, with support for dynamic discovery, adaptation, data validation, persistence, and messaging. The component model interacts with the external services mechanism to acquire and manage real-time mission data, through

the information semantics mechanism to acquire/update component models, through the data server to manage persisted components, and through the user platform to provide component information to user interface representations.

2) **Component Management Mechanism:** This mechanism enables a generic and configurable description of model and user interface elements that will support the design requirements of different mission control applications. The proposed implementation uses a set of OWL-based descriptions to define the models and to describe the interfaces, provides the mechanisms for constructing, managing, and interacting with components from them. It also provides a library of baseline component functionality.

3) **Information Semantics Mechanism:** This mechanism manages content models by interacting with the ontology server to acquire and update the various semantic models and to manage intermediate representations. The proposed implementation uses a JENA-like OWL/RDF query engine and a distributed connection to an ontology server.

4) **User Platform:** This system manages MCT component services and provides the framework interface to applications and external services. It is responsible for all component and subsystem lifecycles and is the access point to external interests and applications. It interacts directly with the component model, the information semantics manager, the security system, and the messaging systems. The User Platform directly manages the configuration management, policy management, and data validation mechanisms.

5) **Configuration Management Mechanism:** This mechanism ensures that system parameters and components can be properly configured at runtime.

6) **Policy Management Mechanism:** This mechanism ensures that all framework services and subsystem attributes can be controlled at runtime.

7) **Persistence Management and Caching Mechanism:** This mechanism ensures that appropriate information is stored in a persistent repository. It guarantees session-level integrity and interacts with the policy management mechanism.

8) **Authentication and Identity Management System:** This mechanism manages access to the application and to components, as well as to system resources and external resources.

9) **Event Handling Mechanism:** This mechanism represents and processes events based on actions from users of the UI or by the framework. This mechanism will be discussed in parallel with the UI representation and exception handling mechanisms.

10) **Exception Handling Mechanism:** This mechanism handles exceptions in a consistent and coherent manner. The exception handling mechanism overrides the built-in exception handling mechanism in Java, as well as how the exceptions are relayed to the user. This mechanism will be discussed in parallel with the UI representation and event handling mechanisms.

11) **Validation Mechanism:** This mechanism validates the Component Model instance and data values against their definitions and allowable data types/ranges. This mechanism will be discussed in parallel with the constraint satisfaction and composition engine mechanisms.

12) **Rule Engine Mechanism:** This mechanism provides a rule-based capability that can be applied against system states in several capacities, particularly composition and constraint satisfaction.

13) **Constraint Satisfaction Mechanism:** This mechanism represents the data constraints across multiple objects, and may require action when the user makes selections and modifications to the user interface. This mechanism makes use of the rule engine mechanism and will be discussed in parallel with the composition mechanism.

14) **Representation Composition Mechanism:** This mechanism enables composition of user interface representations in the application based on user actions such as dragging and dropping (or other action types). This mechanism makes use of the rule engine mechanism and will be discussed in parallel with the constraint satisfaction mechanism.

15) **Messaging Mechanism:** This mechanism dictates how information is conveyed between various distributed elements, whether they are components, external services, data, or ontologies.

16) **Localization/Internationalization Mechanisms:** These mechanisms ensure that localized strings and internationalized structure are supported in the new interface.

Tools

The MCT framework cannot be effectively/efficiently used by application developers or integrators without the following six development tools:

1) **GUI Design and Layout Editor:** This tool provides the application developer with the ability to design a mission application interface from available representation elements (palette), data models (model components), and constraint models (rules), to lay them out according to user/mission requirements. The tool can either be part of a runtime application or be used in an offline capacity. The result of this tool is declarative representations and resources that comprise the application.

2) **Rule and Composition Policy Editor**: This tool supports the construction of a workflow engine specific to developing application-specific rules or component-specific composition policies and produces rule descriptions that can be integrated into the appropriate rule base for a particular application.

3) **Configuration Management and Workflow Editor**: This tool provides the ability to define mission-specific polices, to configure a mission-specific application, and to define workflow within the application and associate it with user and component roles.

4) **Component Development Tool**: This tool provides MCT framework developers with an integrated environment for developing new library components.

5) **Role Editor**: This tool allows mission personnel to create, edit, configure, and manage user role identities and environments, and to integrate them into the security/authentication mechanism for MCT and thus provide access to different levels of mission workflow.

6) **System Integration/Management Tool**: This tool allows the MCT developer or mission integrator to configure the resources required to integrate a mission

application into network resources, to monitor its runtime behavior, and to tune its performance.

Using these tools and components, any application can be constructed to run on top of the MCT Framework. It may be possible to combine some/all of these tools into a single tool or even into the environment. For example, it would be very reasonable to combine the UI design and layout tool, the component development tool, the rule construction tool, and the workflow engine into the same environment.

Returning to the anatomy of the MCT framework-based application from a functional and interaction point of view, the system can be thought of in terms of four interacting aspects: (1) a set of resources associated with the application, (2) a set of tools used to create those resources, (3) the framework that supports and manages the application functionality, and (4) external sources, as shown in Figure 3:

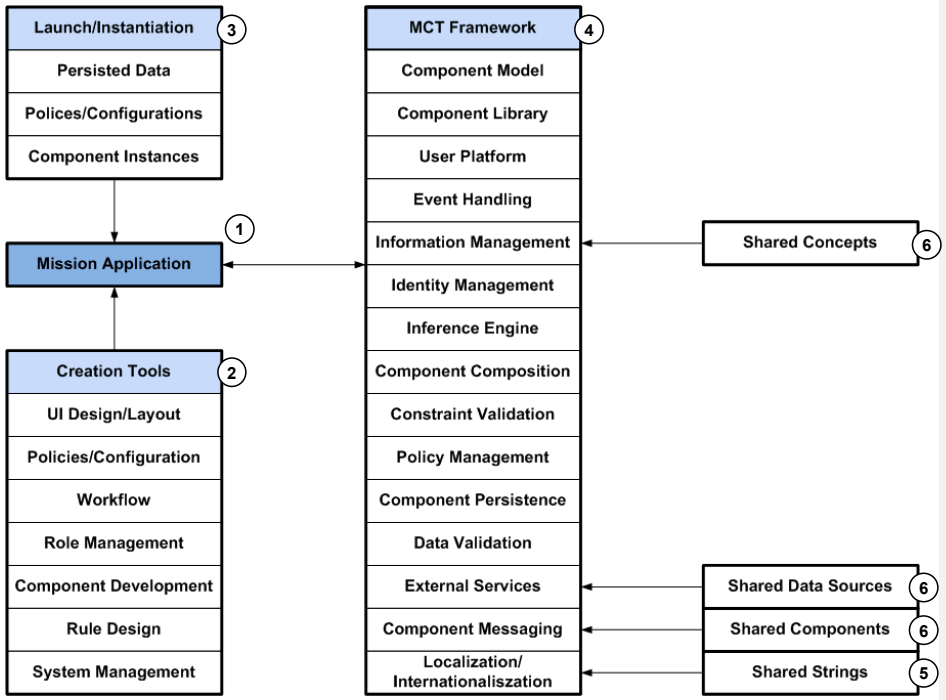


Figure 3: MCT framework and tools interaction with generic mission control application.

A generic mission control application (at **1**) is designed and implemented using one or more of the development tools (at **2**) previously described. The resulting application is represented as a number of artifacts (e.g., XML resources, images, files) that are part of the MCT application deployment package (at **3**). At launch time, these resources are loaded into the application, which instantiates/launches the User Platform and other MCT Framework components (at **4**), thus providing the backbone functionalities described in Figures 1 and 2. The User Platform constructs all of the services and subsystems, configures them, starts them, and makes them ready for operation. These MCT Framework functionalities provide mechanisms that enable an application to adapt

dynamically to changing conditions or user intentions. When the application is rendered, a localization component translates display strings into the configured locale and encoding. These localizations can be retrieved from a localization server (at **5**) at build time. Similarly, shared concepts and components, originating from an ontology server, and shared data, originating from a persistence storage, are retrieved at/during runtime (at **6**).

The chapters following will discuss the architecture and implementation strategy of each subsystem or subsystem group in the MCT Framework. Tools design, where appropriate, will be discussed in separate documents.

Project Organization

The functional components associated with the MCT project have been identified. The organization of the project, and the construction of new components, should follow a consistent organizational approach. To clarify this approach the file system organization for the project will be presented, followed by the organization for Eclipse development.

MCT Project File System

The MCT repository is comprised principally of Eclipse packages. The CVS file structure is shown in Figure 4:

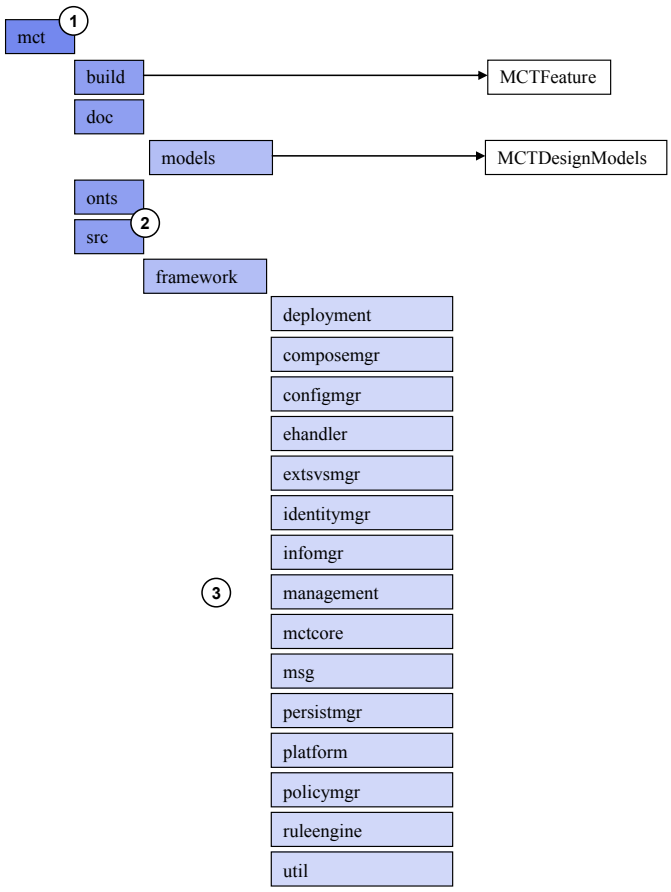


Figure 4: MCT file system architecture in CVS source code repository.

The root directory is named mct (at **1**). Under this directory there are 4 subdirectories: build, doc, onts, and src, as described in Table 1:

Directory Name	Directory Content Description
build	Content relating to the building of the project
doc	Content relating to project design
onts	Project ontologies (until managed by an ontology server)
src	Project source code

Table 1: MCT CVS upper directory structure.

The important thing about the build directory is that it stores information about how the source code is built in the nightly builds. The doc directory is where javadoc and design models are stored. The design models are created from IBM's Rational Software

Architect. Some are reverse engineered from existing code while others are generated UML diagrams for new, unwritten, mechanisms or subsystems.

The onts directory is where project-related ontologies (.owl format) are stored, although they can also be found in applications such as the infomodel demo.

The src directory is the source code repository. The src (at **2**) directory is also divided into 4 subdirectories: apps, framework, proto, and tools, Table 2:

Directory Name	Directory Content Description
apps	Application packages (planning and telemetry)
framework	Framework packages + deployment
proto	Developer private source code directories
tools	Helper applications and demos

Table 2: MCT CVS src directory structure.

The apps directory holds the applications that have been developed for MCT. There are no applications in the tree that work with the current codebase.

The proto directory is temporary storage for developer experiments and is part of the repository only as a backup to what developers are working on before it gets integrated. This directory should probably be removed and developers perform their work on branches.

The tools directory is for stand alone applications used to support the framework, as well as demonstration code such as the UE group demos.

The framework directory holds all of the packages associated with MCT. The framework directory is divided into 14 subdirectories, associated with the 12 functional areas in the project: composemgr, configmgr, ehandler, extsvsmgr, identitymgr, infomgr, mctcore, msg, persistmgr, platform, policymgr, and ruleengine. These packages are described below in Table 3:

Package Name	Package Functionality
composemgr	Composition management
configmgr	Configuration management
constraintmgr	Constraint validation management
ehandler	Exception handling, logging and tracing
extsvsmgr	External services (3rd party connectivity)
identitymgr	Identity management
infomgr	Information semantics management
management	Management console
mctcore	Common interfaces and component/role functionality
msg	Messaging and Component sharing
persistmgr	Persistence management
platform	Framework services platform
policymgr	Policy management
ruleengine	Rule engine
util	Utilities

Table 3: MCT CVS framework directory structure.

Each directory holds both related code for implementing the associated functionality and related testing code. The composemgr directory holds the code associated with

composition and is based on the rule engine. The configmgr directory holds the configuration manager. Constraintmgr hods the code associated with constraint validation and is based on the rule engine. The ehandler directory holds exception handling for logging and tracing. The extsvsmgr directory holds the external services functionality as well as adapters to various 3rd party data sources (particularly ISP and ODRC). The identitymgr holds the identity management functionality. The infomgr directory holds the packages associated with the information semantics manager and related utilities (Jena and RDFGateway packages). Management is an external Java console utility for watching the MCT runtime. It is included in the codebase packages but is not part of the platform startup sequence. Mctcore holds all the code specific to the core component and role model, as well as common interfaces used by all packages, and forms the foundation of the MCT Framework. The msg directory holds the messaging functionality and related adapters (such as P2P and Pub/Sub). The persistmgr directory is a transparency layer for persistence, along with adapters to various persistence management services (XML, OWL, RDBMS). The platform directory holds the code that manages the framework and its services, manages component and role instance life cycle and related services (creation, registration, validation), and manages GUI-specific code and operations. Policymgr holds the code associated with policy management and also makes use of the ruleengine. The ruleengine directory holds the rule engine and related code. The util directory holds commonly used utilities such as XMLFile import and export and property management.

In addition to these packages the framework directory includes a directory called deployment which manages MCT deployment but isn't part of the framework functionality.

MCT Package Organization

Packages in MCT are intended to use the package structure depicted in Table 4:

Package Name	Package Functionality
[package_name]	Primary package functionality interfaces and classes
[module_name]	Related but not primary functionality interfaces and classes
construct	Factories
constants	Constants associated with the package
context	Context for platform-level access
exception	Any package-specific exception classes
test	Test code and unit tests
utils	Utilities local to the package

Table 4: MCT package architecture.

Chapter 2 Component Model

This chapter describes the foundation of the MCT project: the component model. The component model provides a flexible platform from which to construct dynamic and adaptive objects. Issues associated with the design approaches required of such a model, along with a presentation of the representation approach, complete the chapter. The chapter following describes a management layer that handles component descriptions, their conversion from/to an information store, and their runtime manipulation.

Introduction to the Component Model

The Component Model is used to define and create object instances that can be manipulated in visualization environments. These instances can take any form that could be used in any application, from data models to user interface components and views. What forces the design approach of the Component Model is that it must be flexible enough to adapt to a changing definition of any one of the objects it is used to represent and instantiate. That is, if the definition of what constitutes what a telemetry component is or how it behaves changes, an application built with MCT must adapt to the change without requiring a complete rework of the code.

The Component Model forms the foundation of the MCT project because these requirements on its generality and flexibility allow it to represent both the data model and the visual representation elements of the visualization. This degree of generality and flexibility arise because the Component Model isn't based on a pre-defined set of class/data structures for components but, rather, on a flexible and adaptable definition of function that can be shared across components, and added to or removed from the component on demand without changing the component definition. In simple terms this means that blocks of functionality (including attributes and behaviors) can be added to or removed from a component at any time, making component semantics and adaptive to changing semantic requirements.

In MCT, component model definitions are maintained outside of the framework so that the definitions can be shared with other clients, and the component model itself is flexible enough to adapt to changing externally-defined information models.

Model-Specific Implementation Issues

The rationale behind the Component Model design approach is that the exact structure and function of any particular component might not be known at build time; only at run time, and that it might change in some manner during runtime operation. This amount of generality/flexibility requires a component's structure and behaviors to be actively managed, since they cannot be known at build time. This structure imposes 6 constraints on the component model design and implementation, as enumerated below:

- The component model must support dynamic addition and removal of role instances because roles represent functionality in MCT. Components themselves only provide structure continuity.
- The component model must be able to differentiate operations applied to the correct role.
- Components must be able to determine which role attributes and behaviors they satisfy.
- The component model must support forward and backward referencing of components, roles, and their attributes as they can be referenced before and after they are created or instantiated. This means that component and role registries must be maintained.
- Components and roles must be provided with a unique identifiers based on a published semantics since the component can be constructed at run time and clashes would be unacceptable to the behavior of an application.

- Access operations must be defined on the component and role registries for fast lookup on the component or role.

Design Limitations Imposed by Constraints

These constraints put severe limitations on the architecture chosen to implement the component model. In addition, the following 10 capabilities/requirements must be satisfied by the component model architecture:

- A component's functionality is determined by the context of its use, not by its definition.
- The combined component/role model can be used to represent any object's functions.
- Component instances can have constituent functionality (roles) added or removed at runtime without behavioral failure in the application (though the behavior supported may be degenerative).
- Component instances can react to changes in one another.
- Component semantics is decoupled from component instance descriptions. An ontological repository (or repositories) may be used to verify and maintain component semantics.
- Component and role instances are persistent, both locally and to a persistence store.
- Component and role instances are updated/synchronized at appropriate intervals, both their semantics and their content.
- Components and role instances have a configurable/enforceable access policy.
- Components and instances have a configurable/enforceable modifiability policy.
- Operations on components and roles are policy based.

The component model architecture/design will be discussed with these constraints and requirements in mind.

Component Model Requirements and Use Cases

The requirements an illustrative use cases associated with the Component Model are provided in Table 5:

Requirement	Use Cases?	Related Use Cases
CM1: There shall be a single component type to represent both model and view content.	Yes	• Create component
CM2: Component View roles shall have a GUI.	Yes	• Add guiSpec • Remove guiSpec • Attach guiSpec • Detach guiSpec
CM3: A component shall be able to hold state, including references to other components.	Yes, requires message-based assignment mechanism	• Message/System changes Component Value
CM4: Component functional roles and	Yes, requires	• Message/System retrieves

constituents shall be examinable (access to structure, behavior, and values) at runtime.	message-based access mechanism	Component Value
CM5: Component shall include the ability to delegate some state and/or behavior to parent or child components.	No, requires the ability to sub-structure components	
CM6: The component state shall be understood as a mapping from names to values. (Reference through component structure)	Yes, defines how models are managed through attributes	• Message/System to Component
CM7: A component shall be dynamically extendable through the addition of functional roles.	Yes	• Message add Component attribute name • System add Component name
CM8: The annotation of component state shall be possible.	Yes	• Message/System annotate Component
CM9: Component values shall be typed through annotation (e.g., other components, behavior actors, primitive programming language types, or object programming language [reference] types).	Yes	• Message/System add Component type (specialization of annotation for type field)
CM10: The state of a component shall be dynamically restricted by removing functional roles.	Yes	• Message/System remove Component attribute or behavior
CM11: Named role predicates shall be used to define sets of attributes and behaviors that describe capabilities that a component may offer.	Yes	• Component plays Role
CM12: Component roles shall inherit attributes/behaviors from their parent roles	Yes	• Component constructed with Role (sees Ancestors)
CM13: Components will have policy-based assignable/configurable rights.	Yes	• System constructs Component
CM14: The system shall support a fundamental set of operations on component roles including retrieval (find), adding a role or roles (add), and removing a role or roles (remove).	Yes	• Message/System changes Component Value • Message/System retrieves Component Value
CM15: The construction of context-specific implementations of the component shall be provided (factory).	Yes	• System constructs Component
CM16: It shall be possible for components to be developed independently (outside the MCT IDE), based on existing components, for inclusion into the registry without developing new java code. (declarative design capability).	Yes	• System loads Component • System saves Component (user version) • System updates/synchronizes Component to peers
CM17: Components shall be able to use other components as prototypes such that behaviors and attributes from the prototype are transferred to the component being constructed.	Yes	• System constructs Component from Prototype
CM18: A child component shall have access to its parent (or prototype parent) that was used during the child's extension or creation (for compliance).	Yes (lower priority impl, 2 cases: load time and runtime)	• System verifies Component prototype compliance

CM19: It shall be possible to label the roles of a component as not being prototyped during child component creation or extension.	Yes	• System creates Component from Prototype with Restrictions
CM20: The system shall provide base code that facilitates the wrapping of GUI widgets with View roles.	Yes	• System creates GUI-based Component
CM21: Component values shall be verified/validated when changes are made.	Yes	• System verifies Component value
CM22: Component values shall be set, when they are rendered, in the appropriate localization and internationalization.	Yes	• System localizes Component strings
CM23: Components will support accessibility requirements as set forth by NASA policy (e.g., 508B).	Yes	• System supports accessibility requirements
CM24: A component may satisfy multiple roles.	No	
CM25: A component may be presented by multiple views.	No	
CM26: Components shall support composition.	No	
CM27: Changes to a model shall be communicated to all active model presentations.	Yes	• Change component model value
CM28: Roles are named descriptions of sets of functionally-related attributes and behaviors.	No	
CM29: Components can be defined to work with any number of views.	No	
CM30: Components can have any number of currently presented views.	No	
CM31: Components can have any number of inspection views.	No	
CM32: Components can be used as palette items (templates) to create new component instances.	No	
CM33: Components shall organize and have access to their parents and children.	No	

Table 5: Component Model requirements and use cases.

The first column represents the engineering requirement, the second table identifies whether the requirement is associated with use cases, and the third column illustrates some of the use cases associated with the requirement. Requirements highlighted in red are being removed. Requirements highlighted in yellow are lower priority. This format will be used throughout the document. Use case primary scenarios are fleshed out in Appendix A.

Component Types and Role Types

The MCT Component is a building block but this building block has no functionality of its own. The functionality, and therefore the type, of a component is derived from the roles it plays. That is, the component structure is so general that it only has the *capability* to describe functionality and has no functionality bound to it at compile time.

Model Roles and View Roles

In MCT visualization interfaces are composed of components. Visualization interfaces are constructed of user interface widgets and bound to data sources through data or domain

models. Since every view is composed of components, there is an equivalent mapping. The user interface is constructed from a View Role – which has a mapping to one or more user interface widgets and data models. The data and domain models are constructed using a Model Role. Both are defined using the same type of structure, a Role. View Roles are constructed from view prototypes and roles, while Model Roles are constructed from model prototypes and roles. A component may have a Model Role but it must have at least one View Role.

Component Structure

Since a component has no functionality of its own, a component that satisfies a role is *functionally equivalent* to any other component that satisfies the same role. There are three ways to construct a role to have a particular functionality. First a raw component can be created and have all the necessary roles added. Second, a component can be created organically, by combining roles. Third, a component can be created from pre-existing components (i.e., prototypes) and then adding additional roles as needed. A prototype is an existing component that satisfies functional capabilities; it is a starting point. Instead of creating a component from a pure/empty component every time, if a prototype is known that resembles a desired component then it can serve as a starting point in creating a new component. The component's functional capabilities can then be extended by adding roles having the new functionality. A component with arbitrary functionality can thus be constructed from a combination of prototypes and/or roles.

The component model also provides the foundation structure required for constructing component instance templates. An instance template is simply a partly uninstantiated component instance that can be used to create any number of component instances in a graphical context - like a design palette item.

Foundational Structure

The **Component** is the structural building block of MCT, and is a tuple comprised of the following structured characteristics: componentPart, componentParts, definedRoles, currentRoles, inspectors, and preferredView, as shown in the schema definition in Figure 5:

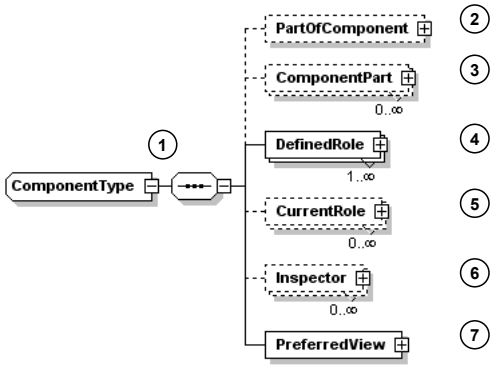


Figure 5: Component structure shown with properties and property range values.

Component (shown as ComponentType, at **1**) is comprised of a single PartOfComponet (itself a ComponentType, at **2**), and a collection of ComponentPart (also ComponentType, at **3**). These represent the parent or containing component, and its children, respectively. A Component is also associated with three Role collections: a collection of DefinedRole (at **4**), which defines what roles are admissible for usage on this component type; a collection of CurrentRole (at **5**), which describes which roles are actually associated with this component instance; and a collection of Inspector (at **6**), which describes which View role instances are acting as inspectors on this component instance. A component also has a PreferredView (at **7**) that defines which of the DefinedRole Views is the default presentation of the component.

These same relationships are shown as a relationship diagram in Figure 6:

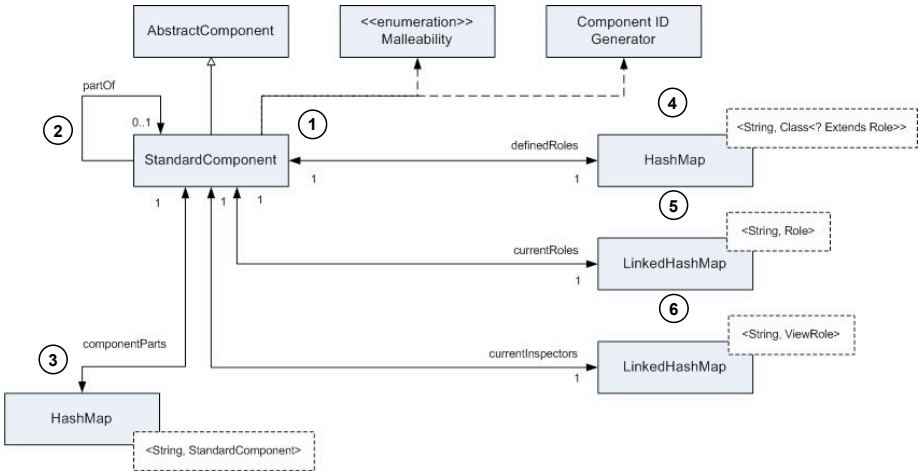


Figure 6: StandardComponent design.

The structure of StandardComponent (at **1** above) is that it has a parent (at **2**) and parts (at **3**) that define its reference hierarchy. It has definedRoles (at **4**), currentRoles (at **5**), and currentInspectors (at **6**) that define its functional hierarchy. The defined roles list defines what role *types* the component *can* play. The current roles list articulates what role *instances* are actually associated with the component instance right now. The current inspectors list is a subset of the current roles list and articulates which of the current roles are to be used for inspection purposes.

Roles

A **Role** is the *functional* building block of the architecture. It defines the attribute and behavior descriptions a component requires to function a particular way, as shown in Figure 7:

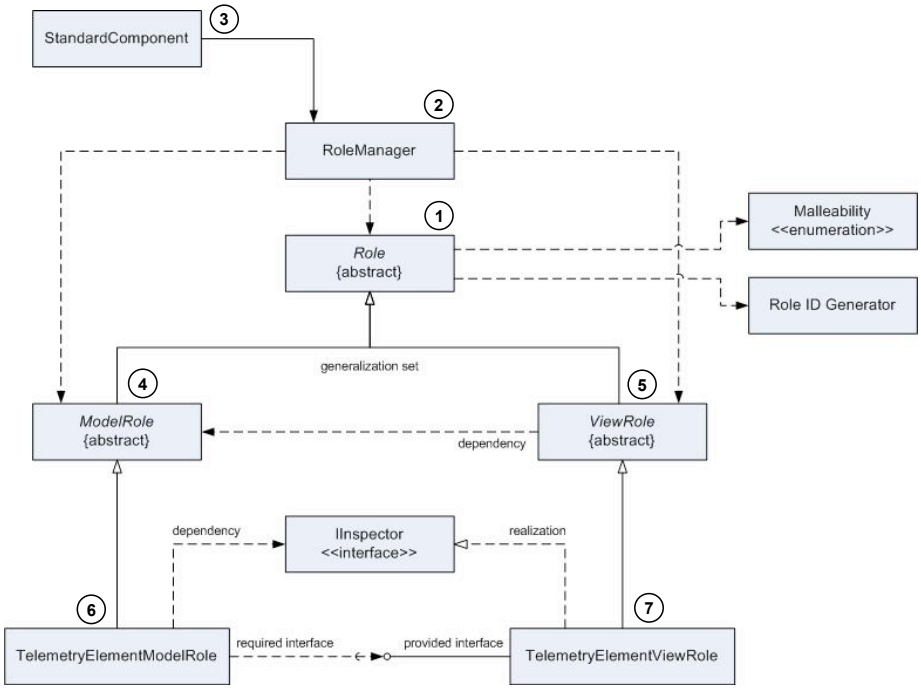


Figure 7: Role structure shown with properties and property range values.

A Role (at **1**) is an abstract definition that defines a set of attributes and behaviors. Roles are managed by a RoleManager (at **2**) which is accessible by a particular component (at **3**) but also system wide. There are two Role types: Model Role (at **4**), and View Role (at **5**). Model roles map to domain objects. The figure provides an illustration where a Sensor's telemetry might be modeled with a type of Model Role called a TelemetryElementModelRole (at **6**) and, in one incarnation, visualized with a View Role type called TelemetryElementViewRole (at **7**). Because of the generic form of the component model, roles provide the only notion of inheritance that can be applied to components. A particular component can realize any number of roles, so a component is defined by its roles and their ancestry.

Model Roles

In addition to the accessors/mutators associated with the data attributes of the domain object, the operations defined on all Model roles are:

changed: Determines whether any of the Role attributes have changed value.

load: Loads the domain data (calls getObject).

updateSource: Saves domain values to source (calls setObject).

getObject: Retrieves the object from the data source.

setObject: Updates the object on the data source.

getData: Retrieves the attribute information from this domain object (used by model and View, calls getOjbect).

equals: Standard comparator.

getError: Retrieves the error value.

setError: Assigns the error value.

getFieldsMask: Retrieves the error mask.

setFieldsMask: Assigns the error mask.

reinitializeFieldsMask: Reinitializes the error mask.

setSavedFieldsMask:

getFieldNames:

revertUnsavedData: If a problem occurs in a save then revert to the buffered values.

Of these, load, updateSource, getError, setError, getFieldsMask, and setFieldsMask have generic definitions and are implemented in the ModelRole class. The changed, getObject, getData, equals, setObject, reinitializeFieldsMask, setSavedFieldsMask, getFieldNames, and revertUnsavedData methods can be defined on the Abstract model class as long as the domain attributes are defined there. The getObject and setObject methods make calls to external resources. The Role developer needs to construct the Abstract[Foo]Model class because this is where the basic domain definition occurs, but then doesn't have to clutter the concrete class with these method definitions.

View Roles

All View roles must implement either the IActionDelegate interface or the IScreenDelegate interface (which itself implements IActionDelegate). The operations defined on these two interfaces are described below:

call: Defined in IActionDelegate, responds to action events in the GUI and maps <Action> tagged items in the GUI definition to operations defined in the View class. The call() method is invoked by the ActionMgr doAction method.

initializeGUIModel: Defined in IScreenDelegate. Responsible for acquiring the current model values associated with the GUI that are needed in the class.

unSetGUIModelInitialized: Defined in IScreenDelegate. Resets the flag that says the GUI model is initialized.

reinitializeGUIModel: Defined in IScreenDelegate. Returns model to previously saved values.

revertGUIModelData: Defined in IScreenDelegate. Returns any changed models to previously saved values.

modelValuesChanged: Defined in IScreenDelegate. Determines whether any model values have been modified.

Of these, unSetGUIModelInitialized is generic and is defined on ViewRole. The call, initializeGUIModel, reinitializeGUIModel, revertGUIModelData, and modelValuesChanged operations require domain definition, so they are defined in a concrete class.

Prototypes and Parts

A prototype is defined as a template component that realizes a set of roles. It is a lightweight starting point for constructing components. Neither prototypes nor templates are Component types. They simply reference Component types or Component instances.

A part, however, is a distinct, autonomous, component that provides necessary functionality to another component. As a component, a part must satisfy at least one role.

Parts have physical counterparts. The parts of an automobile may have an autonomous function, but they may also contribute to the functionality of the overall automobile. An attributional example would be the weight (a property) of a component. The weight of an automobile could be obtained by the combined weight of its parts (i.e., the chassis, body, engine, etc.). A behavioral example woult be the function of a component. The automobile functionality as a mode of transportation cannot be achieved without the functionality of its engine, so the engine is clearly an automobile part. This analogy can be applied to MCT components and their parts except that the parts referred to in MCT are user objects. An example would be a Housing component. A Housing has parts representing a menu area, a directory area, an inspection area, a content area, and a control area. Each of these 'areas' represents a functionally autonomous component but it also contributes to the functionality of the housing.

Component Access/Visibility

In a standard java OOP paradigm object access would be defined by the developer using the standard mechanisms of encapsulation and inheritance. Any operations allowed through these mechanisms would be acceptable at the code level. In MCT there are additional access mechanisms imposed by the integration of identity and policy management into component manipulation. The baseline mechanism to support such evaluations is provided in the component/role model.

In MCT component access/visibility can be defined and enforced at the following four levels:

- **Mission/System** – Coarsest level, applies visibility to across an entire mission or view, or at the system level which would mean all Components.
- **Group** – Group membership (e.g., identity, role) defines component access to a subgroup of components. For example, all components that satisfy a particular role, or an operation initiated by a user satisfying a particular identity.
- **Component** – Access is defined at the component level, meaning that it is controlled by the component type or component instance.
- **Attribute** – Access is defined at the attribute level.

It has been decided that Component instance and Attribute level access control are too granular and so, for the time being, access will be controlled at the system, mission, group, and component type levels. System/Mission level access control means that all components would have the same access permissions. Group level means that all components that belong to a particular group would have the same access permissions. Group membership can be divided into two groups: system and role. System membership would allow MCT systems to operate on components. Role membership means that any component that satisfies a particular role would share the same permissions. Likewise, a user role could be treated in the same manner. Component Type level means that all component instances of a particular type would share the same permissions.

Component Malleability

The flip side of component access is component malleability – i.e., when can a component be modified and in what manner. In MCT there are five malleability types:

- **Mutable** – All modification operations (add/replace/remove) can be performed on the component
- **Additive/Replaceable** – Component fields/values can be added, or their values changed
- **Additive** – Component fields/values can be added
- **Replaceable** – Component field values can be changed
- **Immutable** – No modification operations are allowed

Unlike access permissions, which are defined in one way only (to read), component malleability applies both to the type of malleability and to the enforcement level. The enforcement levels are the same as for component access.

Although both component access and malleability are terms associated with components, these concepts are implemented by the Identity Management system and enforced in the User Platform.

Component Persistence

Persistence means to save component instances (structure and data) to a long-term repository for later use. Persistence can be used for failure recovery, for faster startup, for customization, and for distributed operation. Components are persisted at regular intervals and what defines regular, as well as to decide which data source to use when loading a component instance, are topics for consideration. Component persistence isn't handled by the component model, but by the User Platform, but serialization may be part of the component model. This topic is taken up in greater detail in Chapter 15.

Component Synchronization

Synchronization means to bring conceptual models[1], data models, or domain models into accordance across the MCT client network. Component semantics[2] are loaded at launch time, when the conceptual/semantic model changes, or when the local[3] model changes and needs to be synchronized with the conceptual model. The first case is not a case of synchronization because there is no model loaded yet. The second case is initiated by the ontology server and requires the Information Semantics Manager to interact with the component model and application component instances to synchronize. The third case would be when, for example, someone created a composition that needed to be synchronized with the ontological definitions. In all cases, the synchronization will be mediated by the Information Semantics Manager and the User Platform. This topic is taken up in greater detail in Chapter 5 and Chapter 6.

[1] Conceptual models are used to represent any domain concept. In this context a component model or a data model are simply types of conceptual models targeted at components and telemetry. Models in this context can be likened to the schematic model in an MVC paradigm.
[2] This refers to the component conceptual model.
[3] This could be the result of an action by a user in a particular application.

Component Type Checking

Component values need to be validated both at load time and when changed during operation. The best approach is to define validations using an XML or information model and then the validations can be changed at load time or runtime without rebuilding or redeploying the system. At runtime the validation models have been created and any changes to fields can be checked. This topic will be discussed in detail in Chapter 13.

Component Constraint Satisfaction

Component to component logic constraints need a clear mechanism for identifying constraints. A rule-based approach makes a lot of sense for four reasons:

- **Decoupling** - It decouples component-to-component dependencies from the component code
- **Discrete/Modifiable/Updatable** – It places dependencies into single location.
- **Configurable** – Using an XML approach so that the rules can be modified without modifying the code.
- **Extendable** – New functionality can be implemented that can take advantage of the rules without changing them.

Moreover, since MCT is already using an rule engine to support composition, the rule engine (as a mechanism) is already available for performing constraint satisfaction.

This topic will be addressed in detail in Chapter 11.

Summary

The combination of parent and parts to support compositional structure, roles to define component functionality, and a message-passing mechanism to support component-component interaction together satisfy the constraints and requirements imposed on the MCT component model design. Other issues associated with components that are raised by this approach, such as management and validation, will be discussed in other chapters. The requirement that components be discoverable is handled by having their definitions reside in a declarative (and possibly ontological) form and shared repository apart from the MCT Framework.

Component Model Reference Implementation

The component model implements the basic functionality associated with the component functional capabilities: flexibility, discoverability, and messaging. The component model reference implementation constitutes the central and foundational building block of the MCT Framework and thus its interaction with the framework is of prime importance in discussing its implementation. Beyond its integration, the component model is designed around the delivery of five features:

- **Component Structure:** Basic structural aspects supporting required component functionality.
- **Component Lifecycle:** Basic functionality associated with managing specific components.

- **Component Messaging:** Basic functionality associated with component-component interactions.
- **Role Behaviors**: Basic functionality associated with component behaviors.
- **Component Management**: Basic functionality associated with managing components.

Component Model Dependencies

The Component Model interacts directly with five subsystems, as depicted in Figure 8:

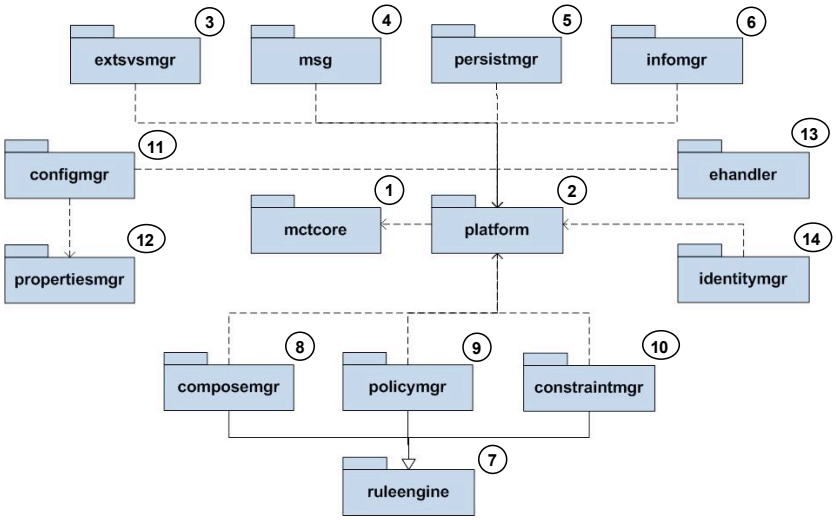


Figure 8: MCT Component Model system dependencies.

The Component Model, illustrated by the mctcore package (at **1**) is the framework aspect responsible for implementing the behavior defined by Component Model use cases. The UserPlatform, which is implemented in the platform package (at **2**) depends on the Component Model because it is responsible for constructing, loading, and managing component and role instances as well as to manage GUIs using the embedded component toolkit (*platform.comp.gui* package).

Four packages are responsible for interacting with the outside world: extsvsmgr, msg, persistmgr, and infomgr. Data from the External Services subsystem (extsvsmgr package, at **3**) is provided to application component instances through service adapters. Interactions between clients are mediated through a messaging backbone and implemented in the msg package (at **4**). Persisted definitions (component, role, gui, configurations) are provided through the Persistence Manager (implemented with the persistmgr package (at **5**). The Component Model is provided with declarative component definitions by the ontology server through the Information Semantics Management subsystem (and implemented by the infomgr package, at **6**).

Three subsystems are responsible for defining and evaluating component based logic using a rule engine (at **7**): composemgr, policymgr, and constraintmgr. Component aggregational logic is decoupled from component definitions through the use of a

Composition Manager that evaluates whether composition should take place at the point of composition using the composemgr (at **8**). Component operational logic is decoupled from component definitions through the use of a Policy Manager rule engine that dynamically evaluates operational fitness (policymgr package, at **9**). Those aspects of policy management that are specific to identity are further handled by the Identity Manager (identitymgr package, at **14**). GUI-based operational logic is decoupled from widget definitions through the use of a Constraint Validation rule engine in the constraintmgr package (at **10**).

Two utilities are used to support system configuration: configmgr and propertiesmgr. The Configuration Manager provides an integrated approach for configuring each subsystem (at **11**). The Properties Manager (propertiesmgr, at **12**) provides client-specific information that is difficult to define in properties.

Finally, framework wide exception handling, logging, and tracing are mediated by the Exception Handler (ehandler package, at **13**).

A closer look shows that the Component Model is comprised of several abstraction layers, as illustrated in Figure 9:

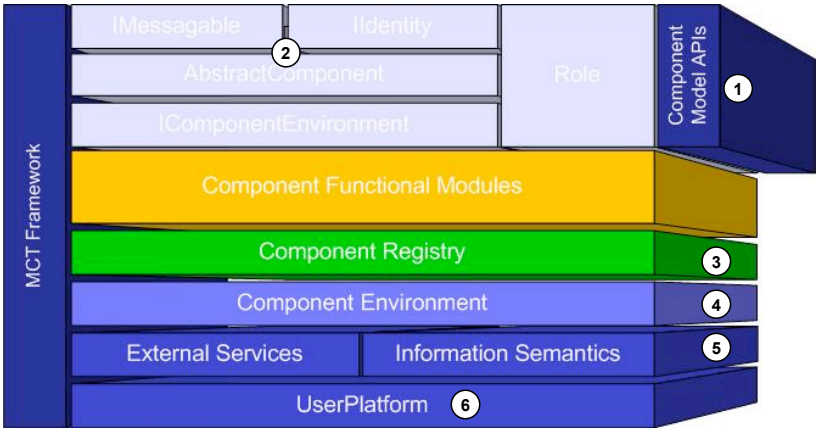


Figure 9: Component Model API relationships in MCT.

This figure shows how the Component Model fits into the MCT Framework. The underlying structure is comprised of several interfaces. The Component group (at **2**) represents the behaviors used to implement component functionality. Sitting above the core APIs and concrete classes is the Component Registry (at **3**), which is a service that manages and provides access to components at run time. The Component Registry is managed by the Component Environment (at **4**), which provides access to all framework services and subsystems to one another, such as External Services and Information Semantics (at **5**). The User Platform (at **6**) manages the MCT Framework.

IComponent is the root of the Component hierarchy. It provides the basic behavioral capabilities that all Component instances will share, as depicted in Figure 10:

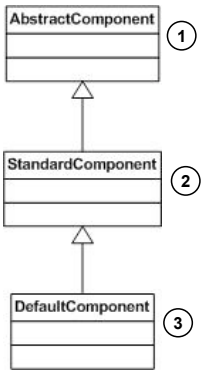


Figure 10: Component hierarchy.

The purpose of AbstractComponent (at **1**) is to define the required operations in COMP use cases. AbstractComponent is used as a parameter or argument across the framework because it can be/is implemented by any other Component abstraction. StandardComponent (at **2**) implements several of the AbstractComponent behaviors while leaving several unimplemented and thus required for subclass implementation. DefaultComponent (at **3**) is currently the baseline concrete Component implementation.

Although IComponent is directly associated with Components, it inherits functionality from 2 other interfaces, as shown in Figure 11:

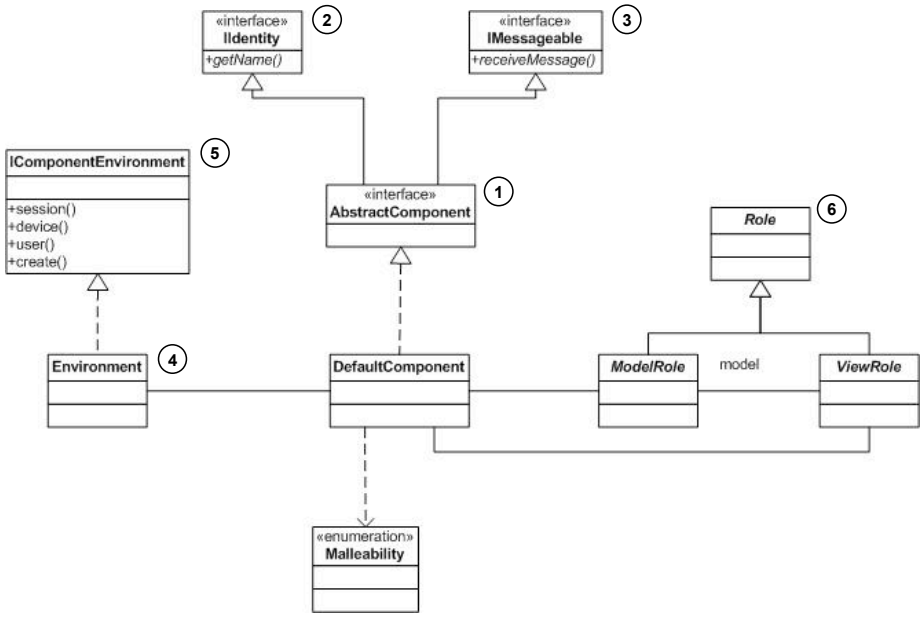


Figure 11: Component Model interfaces and interactions.

AbstractComponent (at **1**) extends the functionality of 2 interfaces. The IIdentity interface (at **2**) provides functionality for identifying a component. IMessable (at **3**) provides the MESG method used in use case definitions.

The baseline component layer and related interfaces are organized with a few others, such as IComponentEnvironment (at **5**). A part of the Role hierarchy (at 6) and the relationship of Component to the Environment (at 4) are shown in the figure to provide some glue.

Summary

This chapter has presented the component model as the foundation for building user interface elements and supporting services relating to them in the MCT framework. The chapter has described the essentials of the component model as well as its interactions with various functional capabilities and systems in the framework. The chapters that follow all make use of the component model either directly or indirectly.

Chapter 3 Component Toolkit

MCT is a visualization framework, meaning that it is intended to provide for the design, construction, and use of application interfaces for a variety of mission control needs. Fundamental to this goal is the encapsulation of component-specific functionality which can be abstracted away from application-specific implementations. The Component toolkit provides baseline functionality that is both generic and domain/context independent. The next chapter describes the library of components that reside on top of the component toolkit and provide domain-specific capabilities.

Introduction

The MCT Framework is comprised of many subsystems devoted to supporting the functionality required of a runtime visualization, but they are functionally autonomous and can be decoupled from visualization-specific operations. Likewise, there are aspects of the GUI functionality that can and should be decoupled from environment-specific contexts if they are to be truly reusable across multiple contexts. The component model provides the foundation for constructing components, but the baseline functionality supporting widgets, widget interactions, component creation, etc. are provided by the Component Toolkit, which is a subset of the platform package. The Component Toolkit builds on the Component Model to provide a set of reusable building blocks that form the corpus of any MCT visualization environment. Abstracted from the toolkit is the component library, which contains domain-specific model and view objects that extend the building block set and are particularly applicable to specific visualization environments.

User interface functionality can be viewed as a combination of all framework-provided capabilities, plus:

- **Foundation GUI Set:** The basic set of widgets that can be used to construct more functionally generalized widgets.
- **Baseline Component Functionality:** The model and view roles that are required in all MCT applications.
- **IO:** Input, parsing, and export of widget, role, and component definitions and descriptions.
- **Management:** Managing component models with respect to the Java implementation, including the binding of GUI widgets to MCT components and data sources as well as event handling between them.
- **Design:** Creating representation components by aggregating representation components and widgets.

Given that the component model underlies all MCT components, the binding of GUI widgetry to MCT components provides an access and management layer for the related widget and model layers.

Constraints on Component Toolkit Design

The Component Toolkit design must satisfy 9 constraints imposed by the way that component information is acquired, saved, and designed:

- GUI widgets, roles, and components are defined to satisfy an information model such as XML Schema.
- Instance (widget, role, and component) descriptions are provided in XML.
- Instance descriptions are validated using the information model at load time.
- Instance descriptions can be used for GUI configuration overrides.
- Instance descriptions are parsed into model, widget, and action information.
- Model, widget, and action information, events, and interactions with roles and components are managed by the Component Toolkit.
- Runtime instances can be exported back to XML and the information model.

- Instances can be composed into increasingly generalized and reusable instances.
- Application design and layout is possible using widgets and composition.

Core Toolkit Requirements

Within the scope of the outward constraints defined above reside the core functionalities that apply to all MCT components. Combined the baseline and core functionalities identify formal requirements and use cases that are presented in Table 6:

Requirement	Use Cases?	Related Use Cases
UIT1: Users can design new user objects by extending existing user object types.	Yes	• User compose user object from template objects
UIT2: MCT shall provide a transparency layer from MCT widgets, roles, and components to widget set, role, and component implementations (there will be a 1:1 mapping from MCT widget to implementation, but not a 1:1 mapping from MCT widget to a particular implementation.	Yes	• MCT supports widget functionality – implemented with Swing widget • MCT supports widget functionality – implemented with SWT widget
UIT3: All user objects shall be selectable.	Yes	• Entity select user object
UIT4: Users can choose actions on all user objects.	Yes	• Entity select menu option (also Entity right click on User Object) • User create object • User modify object • User delete object
UIT5: Selected user object properties shall be configurable.	Yes	• Entity edit user object • Entity configure user object
UIT6: User objects shall have at least one visualization.	Yes	• Entity render user object
UIT7: User objects shall have a preferred visualization.	Yes	• Entity render user object preferred visualization
UIT8: User objects can be created and controlled via user interface controls (e.g., menus, right clicking, keyboard shortcuts, composition).	Yes	• Entity select user object • Entity delete user object • Entity move user object • Entity copy user object • Entity paste user object

UIT9: User objects shall support high-level interactions such as selection, cut, copy, paste, and inspection as enumerated in table UIT3.	Yes	• Entity select user object displays inspector • Entity right click user object • Entity double click user object • Entity drag/drop user object • Entity hover over user object
UIT10: User objects shall support low-level interactions such as mouse and keyboard events, shortcuts.	Yes	• User move cursor • User cntl-s • User cntl-x • User cntl-z • User hover pointing device • User pointing device press • User pointing device release
UIT11: GUI widgets shall support nominal GUI widget properties: enablement/disablement, visibility, borders, layout management, accessibility, localization.	Yes	• Entity enables a representation • Entity disables a representation • Entity sets the visibility of a representation • Entity sets border • Entity removes border • Entity aggregates borders
UIT12: User objects can be hierarchical with respect to GUI containment (i.e., it maps 1:1 to the GUI containment hierarchy).	Yes	• Add Representation • Remove Representation • Get Parent Representation • Get Descendent Representation(s)
UIT13: User objects shall support copying/cloning.	Yes	• Ask if this representation is copyable/clonable • Copy this representation
UIT14: View role may be bound to a model role component.	Yes	• Entity assigns a model component to a representation • Entity removes a model component from a representation
UIT15: View roles may be bound to core GUI widgets.	No	
UIT16: View roles may be asked to rerender.	Yes	• Entity render representation
UIT17: Roles may be asked to update.	Yes	• Entity update representation
UIT18: User object creation, modification, and deletion are policy based.	Yes	• User create component • User modify component • User delete component
UIT19: User object creation, modification, and deletion policies make use of user	No	

identity as well as other information.		
UIT20: Views enabling component creation, modification, or deletion shall indicate whether the operation is permissible.	No	
UIT21: User commands are component role behaviors invoked by users.	No	
UIT22: User commands can be invoked on any component type.	No	•
UIT23: User commands may require commitment before they are executed.	No	•
UIT24:		•
UIT25:		•

Table 6: Component Toolkit requirements and use cases.

It should be noted that use cases identified in the table do not represent an exhaustive set of capabilities but an illustrative set of capabilities. This holds for all such tables. The intent of this table is to articulate the basic functional requirements on components at the toolkit level but not to focus on a particular type of component or operation. For example, the reader should refer to the section on Housings to see which user actions are required on housings, but should find no functionality that isn't supported by the table above, only specializations of same.

The Component Toolkit defines requirements for generic capability that spans all components used by the MCT Framework. These are divided into specific Model Role type requirements and View Role type requirements.

Model Role Requirements

MCT must implement at least 5 Model Role types to be a viable environment:

- **Collections**
- **Taxonomies**
- **Filters**
- **Monitors**
- **Events**

These model role types need to exist whether or not any domain-specific models (e.g., telemetry) are developed (as detailed in Chapter 4). The general Model role requirements are presented in Table 7:

Requirement	Use Cases?	Related Use Cases
A View's Model role state is persisted.		

Table 7: Model Role requirements and use cases.

View Role Requirements

MCT must implement at least 4 View Role types to be a viable environment:

- **Housings**
- **Inspectors**
- **Lists**
- **Plots**

These View role types need to exist whether or not any domain-specific models are developed (as detailed in Chapter 4). The general View role requirements are presented in Table 8:

Requirement	Use Cases?	Related Use Cases
UIT26: View role may be bound to a model role component.	Yes	• Entity assigns a model component to a representation • Entity removes a model component from a representation
UIT27: View roles may be bound to core GUI widgets.	No	
UIT28: View roles may be asked to rerender.	Yes	• Entity render representation
UIT29: View role GUI models shall be independent of the View role's Model role.		•
UIT30: View role GUI state is persisted.		•
UIT31: View roles have a displayable description attribute.		•
UIT32: View role GUIs allow the user to interact with the behaviors provided by the View		•

role and its Model role.	
UIT33: Contained View roles may have the same model as the containing View role, a different model, or no model at all.	•
UIT34: View roles have a standard set of parts: a single content part, a single control part, and a single extended control part.	•
UIT35: View role GUIs are composed from other View roles and GUI widgets.	•
UIT36: View roles support inspection.	•
UIT37: Inspection of a View role is controlled by policy.	•
UIT38: View role GUIs may use any supported layout manager.	•
UIT39: GUI widgets can be used to construct View roles.	•
UIT40: Only GUI widgets bound to View roles can exhibit View role behaviors.	•
UIT41: The GUI widget library will support layout managers.	•
UIT42: GUI widgets do not participate in composition.	•

Table 8: View Role requirements and use cases.

Dsdsd

Requirement	Use Cases?	Related Use Cases
UIT43: View role may be bound to a model role component.	Yes	• Entity assigns a model component to a representation • Entity removes a model component from a representation
UIT44: View roles may be	No	

bound to core GUI widgets.		
UIT45: View roles may be asked to rerender.	Yes	• Entity render representation
UIT46: View role GUI models shall be independent of the View role's Model role.		•
UIT47: Users can adjust a view's content area visualization via the view's control area.		•
UIT48: Users can adjust a view's content area visualization via the view's filter area.		•
		•

Component Toolkit Approach

The semantic intent of the Component Toolkit is to decouple basic functionality from the Component Model on one side and from the Component Library on the other. The goal is to provide a set of reusable and extendable widgets the code for which are never manipulated directly. At the core of this approach is the MCT requirement that all component instances have a declarative representation, because this requirement enables both import from and export to a repository and a uniform definition. Likewise, decoupling basic functionality from the component library implementations forces a uniform approach on development that encourages reusability.

A widget library *can* be used to construct an application interface directly in the standard coding way, through extension, but this produces a tight coupling between the constructed interface and the widget library and reduces reusability. A widget library can also be used through a translation layer so that the application layer has no direct dependency on the widget library. Using this latter type of approach the components, and in particular the component GUI aspects, can be represented declaratively and parsed into programmatic/procedural equivalents at load time. As a result, the programmatic/procedural equivalents are never manipulated directly. To accomplish this requires a very general, framework-level, set of programmatic components that GUI descriptions can be parsed into, but it also requires the removal of application-specific logic because that is the only way to achieve the necessary generality at the widget level. The strength of such an approach is that it is extremely durable and rarely requires modification except to add new baseline components or baseline component functionality. It also has the strength that any component's GUI can be constructed from the library of declarative components, even on the fly. The result is seven constraints on the development of such a library:

- **Reusability:** GUI widget components must be reusable across functional domains.
- **Extendability:** GUI widget component functionality must be extendable.

- **Flexibility:** Support for all controllable GUI widget parameters.
- **Durability:** GUI components must be stable.
- **Lightweight:** GUI components must be lightweight and yet support the MCT look and feel.
- **Complete**: The GUI widget library must support the necessary application domains.
- **Info Model based**: The component library should be constructable from declarative descriptions that include GUI parameters.

Within the scope of these constraints lie the parameters leading to a general component development approach. The approach is presented in Figure 12:

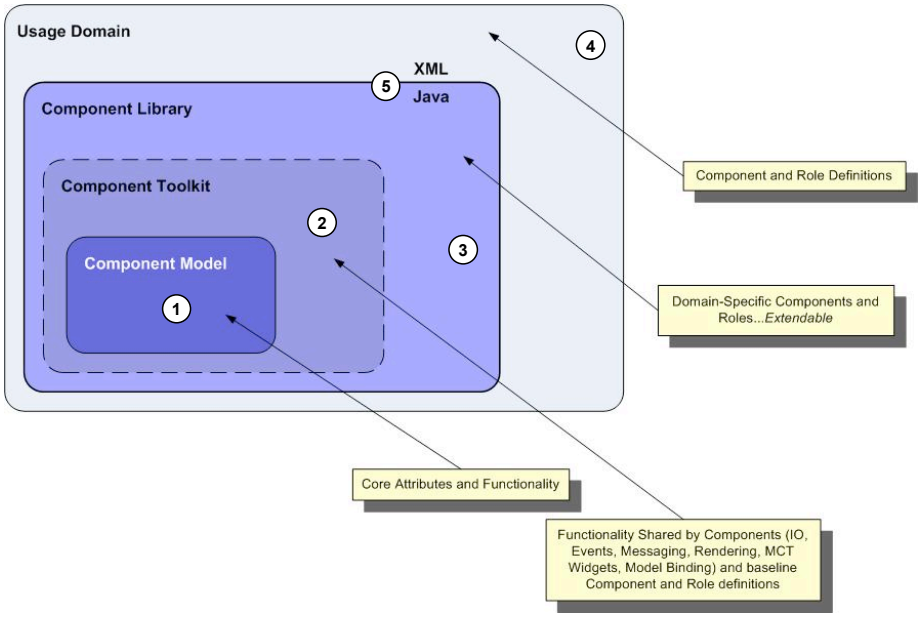


Figure 12: Component Toolkit layering in MCT Framework.

This figure illustrates the relationship between the Component Model (at **1**), which forms the basis for all component/role definitions and component/role-based services in the MCT Framework, and usage domains that make use of the framework (at **4**), mediated by the Component Toolkit (at **2**) and Component Library (at **3**). While the Component Model provides the core functionality needed to construct and manipulate a component, it doesn't have any functionality that could be useful in a usage domain.

The Component Toolkit (at **2**) provides a layer of functionality that is foundational with respect to usage domains that use MCT; building-block support for usage domain interface construction. The Component Toolkit it provides the basic widget, role, and component set needed to construct a usage domain along with the operational support to instantiate, manage, and manipulate them. This is the point where input and output become important, persistence, model binding, event handling, composition, and the like.

That is, service oriented. Domain-specifc Roles are not constructed at this level as it represents functionality that all roles must make use of; these are general capabilities associated with general MCT component types.

Closely related to this layer is a set of domain-specific role definitions that provide support for usage domain interface construction. These comprise the Component Library (at **3**).

The Component Model, Toolkit, and Library are Java implementations. When a usage domain's view is designed, it is done declaratively (at **5**). This decouples the usage domain from the toolkit implementation. Thus everything that is usage-specific except the library roles associated with a particular domain should follow this same strategy of decoupling, while everything that is application-generic will be toolkit specific.

User Objects, Model Roles, and View Roles

Every visual object in MCT is either a core widget or a User Object. A User Object is a visual representation of something meaningful to a user. User Objects are comprised of Roles (Model roles and View roles). The Model role is bound to a domain/data model and provides a value or values to the GUI for rendering. A view role provides the visualization for the model by: (1) holding a user interface description that defines how the GUI and model should interact (i.e., an abstracted GUI), and (2) linking to the actual GUI rendering code. The relationship between components, roles, and user interfaces is shown in Figure 13:

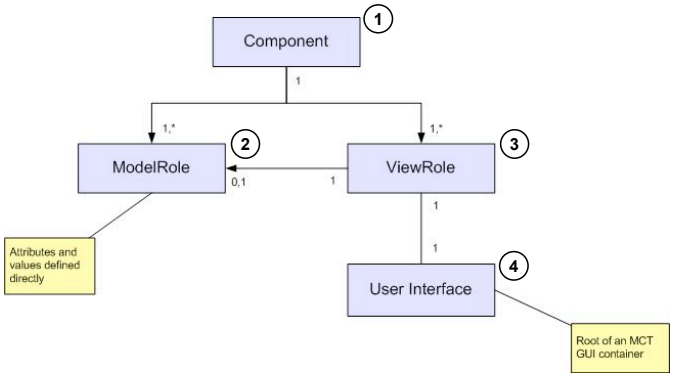


Figure 13: Component, model role, view role, and UI relationships.

Components (at **1**) can be associated with Model (at **2**) or View (at **3**) roles. It can have zero or more Model roles but generally has one. It can have one or more View roles. These may take the form of possible roles, current role instances, or inspector role instances. A Model Role may be associated with one or more View Roles, meaning that it can be viewed in any number of ways. A View Role, on the other hand, may or may not be associated with a Model Role. A View Role for a panel GUI, for example, would not be expected to have a model, but something organized by the panel would. A View Role, as a visualization, has a 1:1 relationship with a user interface's topmost container (at **4**).

Component Toolkit General Architecture

The Component Toolkit aggregates and directs the functionality provided in the MCT Framework toward component manipulation. At the same time, it provides component

creation and export services. In the former respect the Component Toolkit acts like a framework subsystem, while in the latter respect the Component Toolkit acts like a framework service. The relationship the Component Toolkit shares with the rest of the framework is illustrated in Figure 14:

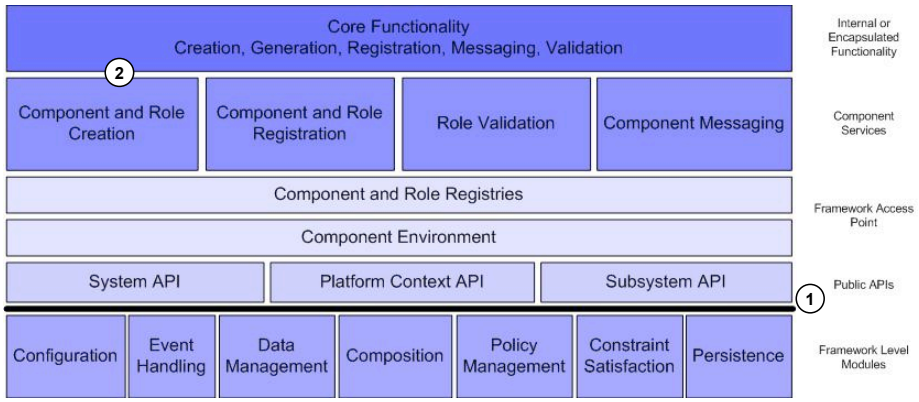


Figure 14: The Component Toolkit as an MCT Framework subsystem.

The Component Model is not represented in this figure, while all functionality above the dark black line (at **1**) is organized in the User Platform, and all functionality below the dark line (again, at **1**) represents framework subsystems. The User Platform is shown in some detail here to show the services provided and the associated APIs. The Component Toolkit, as a collection of functional capabilities, resides in the User Platform (at **2**). Much of what it provides takes the form of creation and export services but it is also responsible for managing the GUI layer and interactions between the GUI layer and the data model.

Component Toolkit Core GUI Widgetry

Underlying all MCT components is a backbone of GUI widgets that are combined to define a particular view role type. The structure illustrating the conversion of a declarative widget description (which can be extrapolated to view roles) is shown in Figure 15:

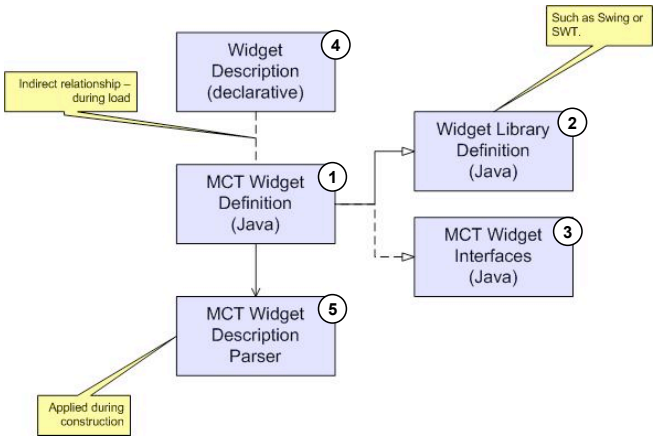


Figure 15: Relationship between widget layer and MCT components.

An MCT widget (e.g., MCTJButton, at **1**) will inherit properties from a standard widget library (e.g., JButton from Swing, at **2**) and other interfaces, such as one supporting parsing and generation capability (at **3**). From this definition, a widget declarative description (at **4**) can be parsed (or generated, at **5**) by the Java version of the widget in the Component Toolkit into the MCT widget.

Widget Foundation Set

At the core of the Component Toolkit is a set of GUI widgets that are defined for MCT. The implementation of these widgets takes the form of an established widget set (currently Swing), but the attributes and behaviors provided are those defined for MCT and thus there is a layer of transparency to the widget set used. It should be noted that the definition of the MCT GUI widget set only guarantees the functionality that the MCT Framework requires, rather than to preclude the use of functionality in the respective toolkit.

The widget set XML schema that is being used at present is illustrate at its topmost level in

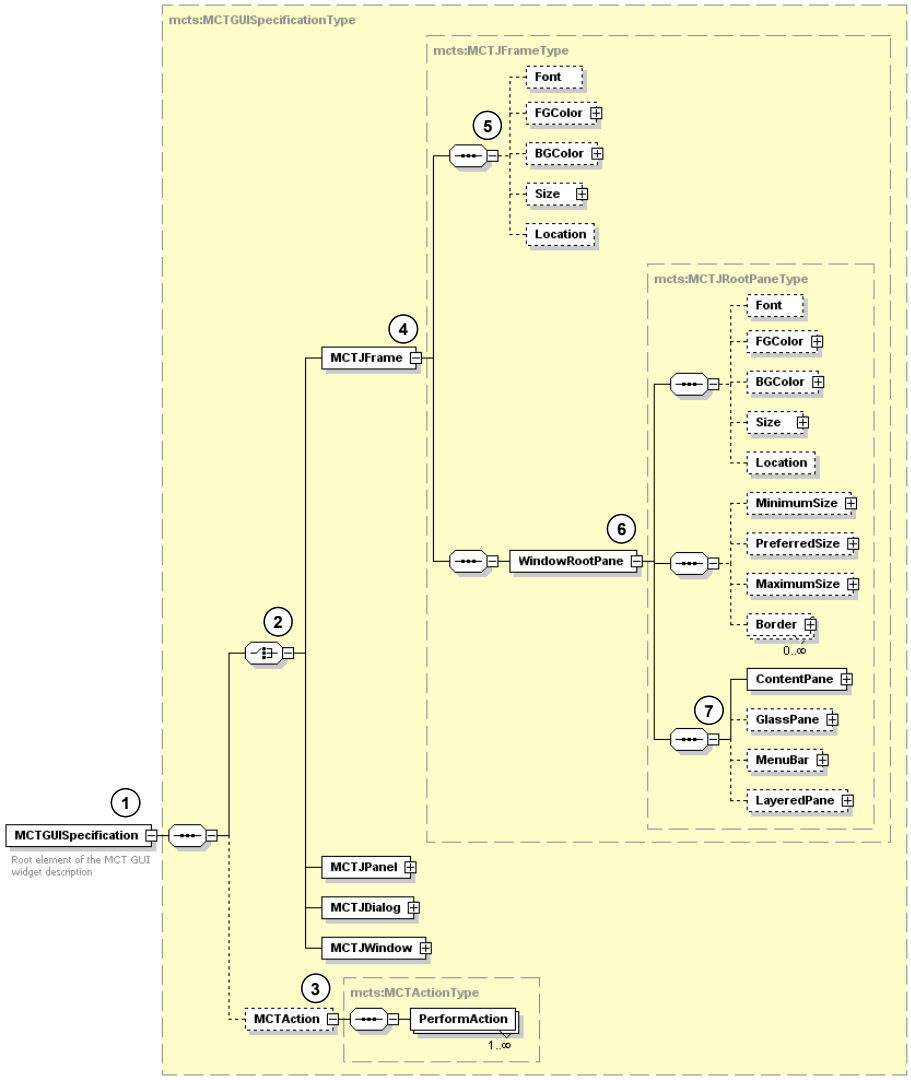


Figure 16: MCT GUI widget set schema. Only Swing equivalent widgets are shown here.

This figure shows that a MCTGUISpecification (at **1**) is comprised of one of the primary Swing GUI containers (at **2**): Frame, Panel, Dialog, or Window, along with any number of actions (MCTAction, at **3**). The figure has expanded the definition of MCTJFrame (at **4**) to show its constituents. One can see that MCTJFrame has basic elements of font, size, color, and location (at **5**) but that it also points to a root pane (at **6**). It is embedded in this structure, at the ContentPane, GlassPane, LayeredPane, or MenuBar levels (at **7**) that one would see the items allowed for composition into an MCTJFrame.

To make it clearer in an image, the constituents for MCTJPanel are shown in Figure 17:

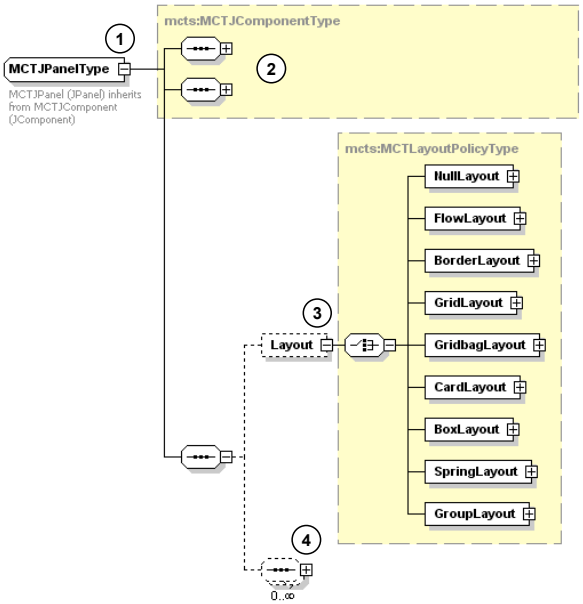


Figure 17: XML Schema representation of MCTJPanel.

In every GUI widget collection a Panel is a significant portion of any GUI description. It is a generic, frameless, GUI container. In this case, Panel is defined as MCTJPanel (extending the Swing JPanel, shown at **1**). MCTJPanel "inherits" from MCTJComponentType (shown at **2**), which is not shown opened up in this figure. The inheritance is shown in quotes because MCTJPanel extends the JPanel class, so it cannot extend any other class. The inheritance to MCTJComponent is noted because, during parsing and generation, MCTJComponent is referenced directly when constructing MCTJPanel. The Panel is comprised of a layout manager type (at **3**, plus constraints), and any number of "containees" (at **4**).

The constituents that can take the form of a containee are represented with an XML Schema construct called a substitution group, which simply means that any item in the substitution group can take the place of the main element (called MCTComponent). There can be any number of these items contained by the MCTJPanel container at any time, subject to the restrictions imposed by the layout manager. Members of the substitution group for MCTComponent are depicted in Figure 18:

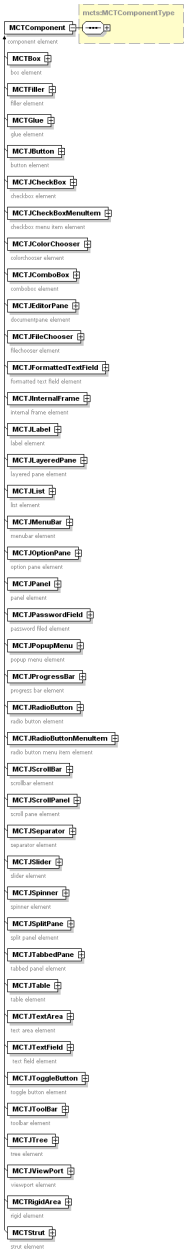


Figure 18: Membership of the MCTComponent substitution group.

Although this figure is not very readable, it represents those items that can be used in an MCT container. There is also a substitution group representing the items that can be contained by MCT Housing Roles (MCTHousingComponent), namely DirectoryArea, ControlArea, ContentArea, and InspectionArea.

Returning to MCTGUISpecification, this element, as mentioned previously, is a placeholder for a GUI container. MCTGUISpecification is referenced in the ViewRole element GUISpec.

Component and Role Foundation Set

Like the widget set, MCT has a schematic definition of what a component's structure looks like, as well as what a role's structure looks like. The associated XML schema for components and roles is shown in Figure 19:

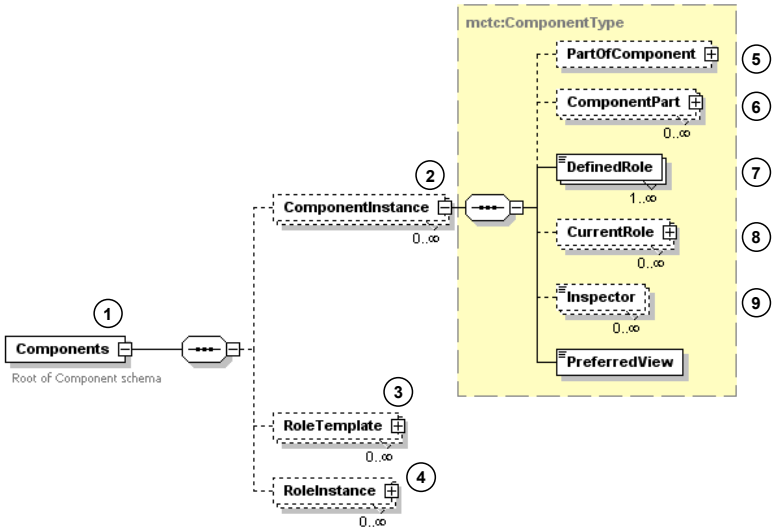


Figure 19: MCT component and role schema.

In this schema, the Components element (at **1**) is a placeholder much as GUISpecification is a placeholder for the GUI widgets schema. Under this element can be three lists: one of Component instances (at **2**), one of Role templates (at **3**), and one of Role instances (at **4**). Each component instance can be described as having a parent component (PartOfComponent, at **5**), a number of component parts (ComponentPart collection, at **6**), a collection of roles that are defined for the component (DefinedRole collection, at **7**), a number of currently used roles (CurrentRole collection, at **8**), a number of inspector roles (Inspector collection, at **9**). Among the attributes defined on Component are a mutability value and a preferred view role.

Roles are defined in hierarchies of Model Role and View Role. The Model Role hierarchy is shown in Figure 20:

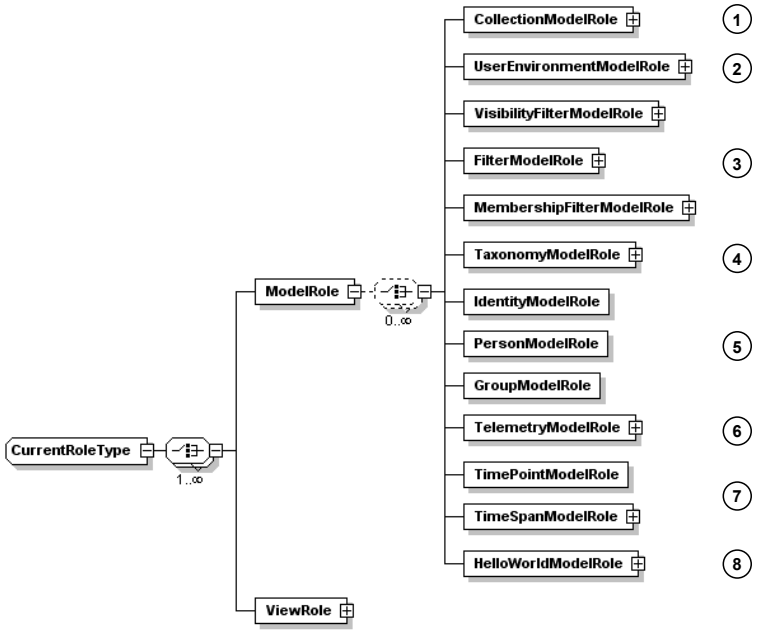


Figure 20: MCT Model Role hierarchy.

This hierarchy defines the core functionality of the MCT Framework from a domain model perspective. CollectionModelRole (at **1**) is used to describe general collections of objects. UserEnvironmentModelRole (at **2**) is a CollectionModelRole tailored to a tree structure. FilterModelRole (and similar, at **3**) are used to define algorithms that apply to collections to reduce their effective scope. TaxonomyModelRole (at **4**) is used to articulate the semantic organization of tree structures. IdentityModelRole (and similar, at **5**) is used to describe users. TelemetryModelRole (at **6**) is used to describe telemetry. This role is domain specific and can be moved to the component library eventually. TimePointModelRole and TimeSpanModelRole (at **7**) are used to describe planning elements. HelloWorldModelRole (at **8**) is really a SingleModelRole, the counterpart of CollectionModelRole.

The View Role hierarchy is shown in Figure 21:

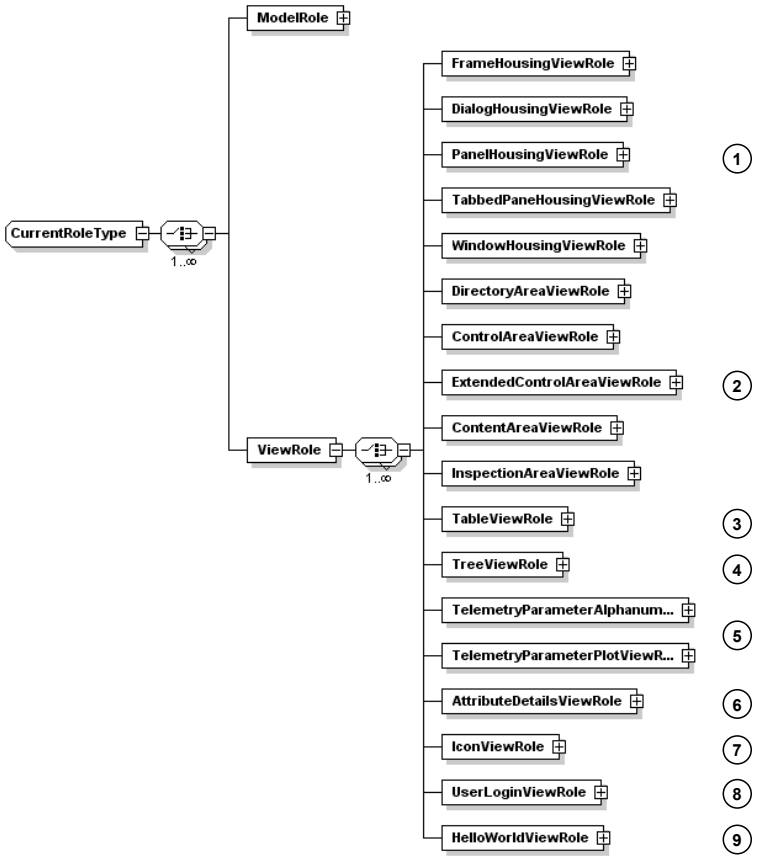


Figure 21: MCT View Role hierarchy.

This schema segment illustrates the currently-defined view types that constitute the MCT Framework core. The first group is the Housing group (at **1**) and represents container-like views (Frame, Dialog, Panel, TabbedPane, and Window). Fundamentally they implement their widget counterparts, but they have both specialized views and behaviors. The next group is the Parts group (at **2**). The parts group represents types of PanelHousingViewRole that make up the significant functional aspects of Housings. The first is DirectoryArea, which is used to display the organization of Components managed by the Housing. The second is ControlArea, which is used to manage the model data displayed in the ContentArea. The third is ExtendedControlArea which is a Frame version of ControlArea. The fourth is ContentArea, which is where the Components managed by the Housing are displayed. The fifth is InspectionArea, which is used to inspect any item in the DirectoryArea or ContentArea that is selected.

TableView (at **3**) is used to display CollectionModelRole content in a table. TreeViewRole (at **4**) does the same thing in a tree. The telemetry twins, alphanumeric and plot (at **5**) are used to display telemetry and are really library views rather than core views. AttributeDetailsView (at **6**) is an inspection view for telemetry metadata. IconView (at **7**) is

for display icons. UserLoginView (at **8**) will probably be removed. HelloWorldView (at **9**) is a test view and a precursor to TelemetryView.

Due to their importance in defining views, the HousingViewRole's GUI specification is further fleshed out in Figure 22:

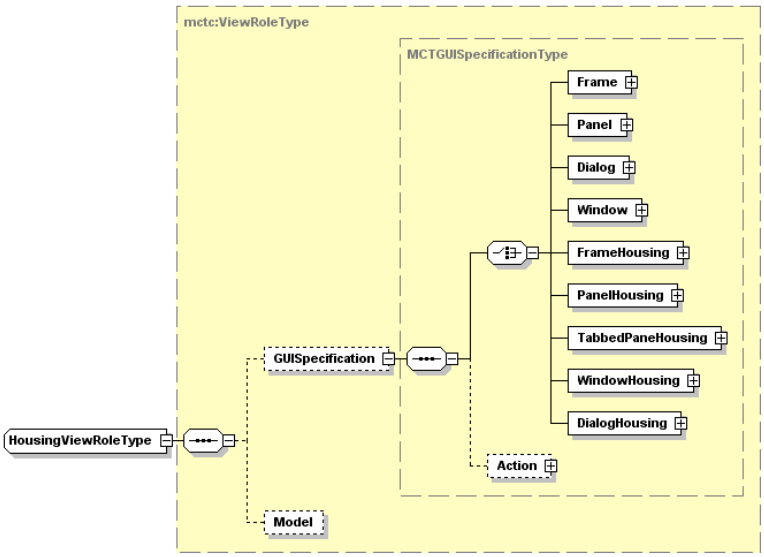


Figure 22: HousingViewRoleType schema definition.

Notice from this figure that GUISpecification can be implemented by Swing JFrame, JPanel, JDialog, or JWindow as well as MCT Housing widgets FrameHousing, PanelHousing, TabbedPaneHousing, and WindowHousing (which themselves have supporting widget specializations). We use Swing-independent element names to retain a widget library agnostic approach as much as possible while keeping enough so that Swing developers won't get confused. Fortunately widget names are now widely used in the industry.

FrameHousing is further decomposed in Figure 23:

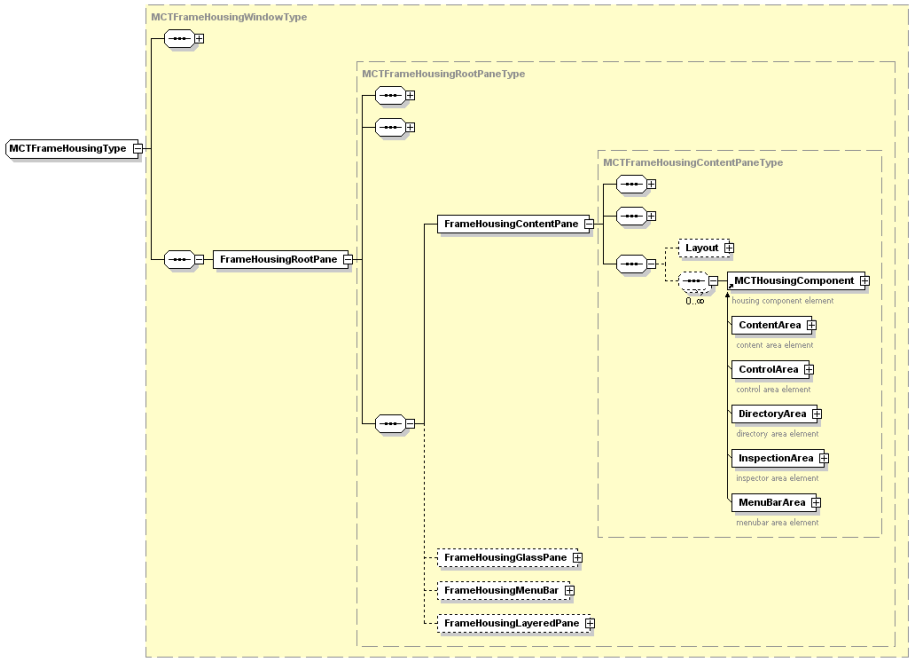


Figure 23: MCTFrameHousing widget.

This figure illustrates one of the differences between Swing containers and MCT Housings. A Swing container can contain any Swing object, while an MCT Housing can only contain a ContentArea, a ControlArea, a DirectoryArea, an InspectionArea, or a MenuBarArea. There is no specification made at this level as to the type of layout to be used, so it could be any layout that would support all 5 area types.

It should be noted that of these areas, the ControlArea and ContentArea map 1:1 to MCTJPanel in that they can contain any Swing widget[4]. The DirectoryArea can contain either an MCTJTree or an MCTJTree in an MCTJScrollPane. The InspectionArea can contain an MCTJTabbedPane. The MenuBarArea can contain an MCTJMenuBar.

It should come as no surprise that neither of these hierarchies is complete. They are a starting point, but they provide ample room for expression MCT views. Future versions will expand the baseline set of roles, while domain-specific specializations will be moved to the component library.

Together these schemata (MCTSwing and MCTComponents) define the structure of components, roles, and widgets in MCT.

An example fragment illustrating how these schemas translate to XML in MCT is depicted in

[4] At this time they cannot contain MCT components.

```
<?xml version="1.0" encoding="UTF-8"?>
<mctc:Components lookAndFeel="Windows"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xmlns:mct=http://mct
                 xmlns:mcts="http://mcts"
                 xmlns:mctc="http://mct/mctc"
                 xsi:schemaLocation="http://mct/mctc MCTComponents.xsd">
  <ComponentInstance id="Launcher"
                     name="Launcher"
                     longName="root containerl"
                     mutability="MUTABLE">
    <DefinedRole>FrameHousingViewRoleDefinedRole>
    <CurrentRole>
      <ViewRole id="FrameHousingViewRole_1" type="view">
        <FrameHousingViewRole id="FrameHousingViewRole_1" type="view">
          <GUISpecification lookAndFeel="Windows">
            <FrameHousing id="Launcher"
                          enabled="true"
                          title="Mission Control Technologies"
                          visible="false"
                          wid="800">
            </FrameHousing>
          </GUISpecification>
            <Action>
              <PerformAction method="close" name="MCT.close"/>
            </Action>
          </GUISpecification>
        </FrameHousingViewRole>
      </ViewRole>
    </CurrentRole>
    <PreferredView>FrameHousingViewRolePreferredView>
  </ComponentInstance>
</mctc:Components>
```

(1)

Figure 24: Sample component schema instance. GUISpecification collapsed.

This figure illustrates the construction of a GUI XML file that satisfies the definitions imposed by the MCTComponents and MCTSwing schemas. The GUISpecification (at **1**) is enclosed in a FrameHousingViewRole instance (which is inside a ViewRole, inside a CurrentRole, inside a ComponentInstance). The schema allows for any number of component instances to be defined within the file. It also allows for any number of defined role types (as strings) or current roles (fleshed out) to be defined within a component instance.

Component and Role References

It is an important requirement that this kind of XML description not force duplication on the developer. As such, it is imperative that both forward and backward referencing be supported in the parsing and management of components and roles. What this means is that if a role instance is used in one file it can be defined in another, or defined in one and used in another, or defined in one location within a file and used in another. In MCT, this is being implemented in such a way that if there are no elements inside a role or component definition it is assumed that the role or component id is a reference, whether or not that component or role has been defined (i.e., registered with the User Platform). It is assumed that if a reference is used that the role or component definition will be provided before the component or role is required.

Component/Role/GUI File Parsing and Generation

The parsing and generation of Component files (including their GUI definitions) is moderated by 2 interfaces defined in the mctcore.xml package: IMCTXMLParse and IMCTXMLGenerate. These interfaces define methods for each operation:

IMCTXMLParse:

parseXML
- parseXMLAttrs – parse an item's attributes
- parseXMLElts – parse an item's elements
- parseMCTXML – construct management relationships

IMCTXMLGenerate:

generateNode
- addAttrs – generate an item's attributes
- addElts – generate an item's elements

In each case, the parsing or generation is hierarchical; a top-level object is asked to parse or generate itself and it, in turn, asks any children to parse or generate themselves, down to terminal object types. At the same time, when an object is asked to parse or generate itself, it may in fact be extending another class, so before the object parses or generates itself it first parses or generates its parent. The main parse or generate operation is broken up into a parse or generation of the attributes followed by a parse or generation of its elements. The parsing operation takes a Node as an argument and produces a Java object as its result, which the generation operation takes a Java object as its argument and produces a Document element as its result.

Parse Workflow

Parsing is the process whereby descriptions of components, roles, and GUIs are converted from their declarative (XML) forms into their Java counterparts that will be used at runtime. The process whereby parsing is accomplished is illustrated in Figure 25:

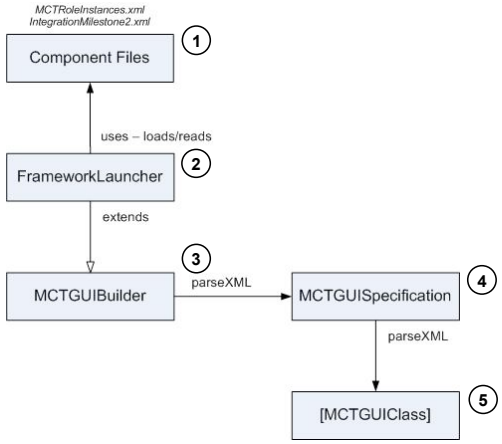


Figure 25: Parsing workflow.

It should be noted that this process only identifies the starting point. MCT is launched from an application currently called FrameworkLaunch, which invokes the MCT platform startup. After the framework is 'up' the component toolkit is invoked through the FrameworkLauncher class (at **2** above). This class acquires the names of and reads the two component (roles and gui) files (at **1**) in its launch method. FrameworkLauncher extends a class called MCTGUIBuilder (at **3**) that is responsible for managing any particular [primary] view. The actual parsing of Role definitions is handled by MCTRolesBuilder, while everything GUI related is handled by MCTGUIBuilder. The launch point for this class is the initialize method (which parses the GUI XML file), but the real action is in the initializeGUIs method, which is responsible for initiating the parse of MCTComponents, and in initializeGUIModels, which is responsible for initiating the parse of models for binding to the GUI widgets and to the Model role instances. As mentioned previously, every View Role is associated with a GUI definition. This definition is accessed through an MCTGUISpecification (at **4**) through a parseXML call at the Role level. Each MCTGUIClass (at **5**) has its own parse and generate definitions, and the parse and generation are both recursively defined. Once the components parse is initiated, it is a recursive process.

Parsing Root Operation - parseXML

The parse process is hierarchical and recursive. As an example, consider the parsing methods for MCTJButton as shown in Figure 26:

```
public Object parseXML(Node node) {
    MCTJButton button = (MCTJButton) createObject();
    String     osname = System.getProperties().getProperty("os.name");

    button.mMCTGUIInfo = ParserHelper.parseMCTGUIComponentInfo(node);

    parseXMLAttrs(node, button);
    parseXMLElts(node, button);
    parseMCTXML(node, button);

    return(button);
}
```

Figure 26: Example parseXML method, for MCTJButton.

There are four important things to note in this method. First, every widget type will have this root parse method. Second, every widget type will use ParserHelper to parse the MCT-specific attributes associated with MCTGUIComponentInfo. Third, every widget parser will call parseXMLAttrs and parseXMLElts, but not all parsers will have a call to parseMCTXML.

MCTGUIComponentInfo

The attributes associated with MCTGUIComponentInfo are: id, model, action, placeholder, parentRole, and parentComponent. It should be obvious to any Swing developer that these attributes would not be a part of any Swing widget, let alone all of them.

The id is necessary for referencing throughout MCT and, in normal Swing development, would be created in place.

The model attribute represents the model layer and is used to reference the domain model that will be used to bind the value of the widget. Again, in Swing widgets there are always

methods used to bind a value to the widget model, but they aren't uniform and they are invoked in place. The model attribute is uniformly available and managed, allowing for the notion of generalized widgets, but at the cost of some degree of control.

The action attribute is used for those widgets that have actions, such as buttons.

The placeholder attribute is used in cases where a widget needs to be swapped at runtime with another widget.

The parentRole and parentComponent attributes are used for component selection purposes.

All of these attributes are wrapped up inside an instance of MCTGUIComponentInfo that is bound to the widget after parsing and is thus available to the component during runtime operation.

Attribute Parsing – parseXMLAttrs

Every widget has some attributes that are defined on it that are used to construct or initialize the Swing widget. The selection of which attributes are included may, to the reader of the XML Schema, seem arbitrary, but it is not. To be an XML attribute the item must be a primitive object type: String, Boolean, integer, …the selection of which attributes to include is thus based on which constructors and mutators take primitive types as arguments. The second consideration is consistency. In many cases the definition of Swing attributes has not been consistent across the Sun library, but in the MCT schema an attempt has been made to improve consistency across classes, so the names may appear differently in the schema than they would in the Swing API.

Within the parseXMLAttrs method the nodes which are parsed are specific to the widget type being represented and parsed. The parse is recursive and hierarchical, so the root object is always passed into the parse method and the result of the parse is added into the root object. When the parse is complete, the root object is fully defined.

An example of parseXMLAttrs is shown, for MCTJButton, in Figure 27:

```
public void parseXMLAttrs(Node node, Object obj) {
    IMCTXMLParse btnParser = SchemaInstanceFactory.getParserInstance(SchemaClassConstants.ABSTRACT_BUTTON);

    btnParser.parseXMLAttrs(node, obj);
}
```

Figure 27: parseXMLAttrs for MCTJButton.

Notice that this method suggests that JButton has no attributes. When we look at the Swing JButton API we find that this is exactly true. JButton inherits attributes from AbstractButton, and that is exactly what parseXMLAttrs does; it parses the attributes from AbstractButton, which in turn parses additional attributes from JComponent, etc. (i.e., recursively up the abstraction hierarchy).

Developers will immediately recognize that the hierarchical parsing mechanism is identical to the nature of OOP object construction, where an object's parent or superclass is constructed before the object itself. Others will ask the question why, if when we construct a Swing JButton widget, all the attributes not defined on JButton are automatically assigned to the proper level of abstraction (i.e., AbstractButton, JComponent, etc.), do we have to do it manually in MCT. The simple answer is that in MCT we are parsing from XML into MCT widgets and we do not want to associate all attributes with a single layer of

abstraction that is completely different from the underlying widget set. This mechanism also allows MCT to use the underlying classes directly.

Another example is shown for MCTJComponent in Figure 28:

```
public void parseXMLAttrs(Node node, Object obj) {
    NamedNodeMap          attrMap              = node.getAttributes();
    IMCTXMLParse          containerParser      =
SchemaInstanceFactory.getParserInstance(SchemaClassConstants.CONTAINER);
    Node                  hAlignmentNode       = attrMap.getNamedItem(ATTR_H_ALIGNMENT);
    Node                  vAlignmentNode       = attrMap.getNamedItem(ATTR_V_ALIGNMENT);
    Node                  toolTipNode          = attrMap.getNamedItem(ATTR_TOOLTIP);
    Node                  opaqueNode           = attrMap.getNamedItem(ATTR_OPAQUE);
    Node                  autoscrollsNode      = attrMap.getNamedItem(ATTR_AUTO_SCROLLS);
    Node                  focusNode            = attrMap.getNamedItem(ATTR_FOCUS);
    JComponent            jc                   = (JComponent) obj;
    LocalizationResource localizationResource = LocalizationResource.getInstance();

    containerParser.parseXMLAttrs(node, obj);

    if (hAlignmentNode != null) {
        jc.setAlignmentY(getHAlignment(hAlignmentNode.getNodeValue()));
    }

    if (vAlignmentNode != null) {
        jc.setAlignmentX(getVAlignment(vAlignmentNode.getNodeValue()));
    }

    if (toolTipNode != null) {
     jc.setToolTipText(localizationResource.getString(toolTipNode.getNodeValue().trim()));
    }

    if (opaqueNode != null) {
        if (opaqueNode.getNodeValue().intern() == TRUE) {
            jc.setOpaque(true);
        } else {
            jc.setOpaque(false);
        }
    }

    if (autoscrollsNode != null) {
        if (autoscrollsNode.getNodeValue().intern() == TRUE) {
            jc.setAutoscrolls(true);
        } else {
            jc.setAutoscrolls(false);
        }
    }

    if (focusNode != null) {
        if (focusNode.getNodeValue().intern() == TRUE) {
            jc.requestFocus(true);
        } else {
            jc.requestFocus(false);
        }
    }
}
```

1 2 3 4 5

Figure 28: MCTJComponent instance of parseXMLAttrs.

In this version there are 6 attributes being parsed. We first acquire the attributes in the form of a NamedNodeMap (at **1**). Then we access the nodes we are interested in using predefined constants (at **2**). Before we do anything with these nodes, however, we first parse the superclass (at **4**). In this case the superclass is Container. The localization resource is also acquired (at **3**) since JComponent is where the tooltip is defined, so it will need to be localized. Once the superclass attributes have been parsed the nodes are each checked and the local object's attributes are assigned their parsed values.

Element Parsing – parseXMLElts

After a widget's attributes are assigned parseXML goes through another sequence where the widget's elements (structured objects) are parsed. An example of parseXMLElts is shown for MCTJButton in Figure 29:

```
public void parseXMLElts(Node node, Object obj) {
    IMCTXMLParse btnParser = SchemaInstanceFactory.getParserInstance(SchemaClassConstants.ABSTRACT_BUTTON);

    btnParser.parseXMLElts(node, obj);                                                    (1)

    MCTJButton button   = (MCTJButton) obj;
    NodeList   children = node.getChildNodes();

    for (int i = 0; i < children.getLength(); i++) {                                      (2)
        Node childNode = children.item(i);

        if (childNode.getNodeType() == Node.ELEMENT_NODE) {
            String childNodeName = childNode.getNodeName().intern();

            if (ELT_PARAMS == childNodeName) {                                            (3)
                MCTActionParams apInstance = new MCTActionParams();
                MCTActionParams ap         = (MCTActionParams) apInstance.parseXML(childNode);

                button.mActionParams = ap;
            }
        }
    }
}
```

Figure 29: MCTJButton version of parseXMLElts.

Notice that, like parseXMLAttrs, the parseXMLElts method starts with a recursive parse of its parent's elements (at **1**). Then it iterates through the child element nodes defined for it (at **2**). In this case, the only added elements defined on JButton are those associated the parameters of the action (at **3**).

GUI Management – parseMCTXML

The final pass in parseXML is a call to parseMCTXML. This call is not recursive, because its task is not a parsing task but a management one. parseMCTXML creates bindings between the widget just parsed and the aspects that will make it a full-fledged GUI widget within MCT. Consider the call to parseMCTXML for MCTJButton shown in Figure 30:

```
public Object parseMCTXML(Node node, MCTJButton button) {
    IMCTGUIComponentInfo info         = button.getMCTGUIComponentInfo();
    String               model        = info.getModel();

    ButtonActionAgent    agent        = new ButtonActionAgent(button, model);   (1)
    GUIAgentMgr          agentManager = GUIAgentMgr.getInstance();
    GUIAgentStore        agentStore   = agentManager.getCurrentAgentStore();    (2)

    if (agentStore != null) {
        agentStore.addAgent(agent);                                             (3)
    }

    String         action        = info.getAction();
    GUIActionMgr   actionManager = GUIActionMgr.getInstance();
    GUIActionStore actionStore   = actionManager.getCurrentActionStore();

    if (actionStore != null) {                                                  (4)
        ComponentActionParams cap = new ComponentActionParams(button, button.mActionParams);

        actionStore.addUIActionMapping(cap, action);
    }

    ActionListenerMgr actionListenerMgr = ActionListenerMgr.getInstance();
    ActionListener    actionListener    = actionListenerMgr.getCurrentActionListener();  (5)

    button.addActionListener(actionListener);

    GUIInfoMgr          infoManager = GUIInfoMgr.getInstance();
    MCTGUIComponentInfo ginfo       = new MCTGUIComponentInfo(info);
    GUIInfoStore        infoStore   = infoManager.getCurrentInfoStore();

    if (infoStore != null) {                                                    (6)
        infoStore.addCompInfo(button, ginfo);
    }

    return(button);
}
```

Figure 30: parseMCTXML for MCTJButton.

Several points need to be reviewed about parseMCTXML. The first is about agents. The first block in this method refers to the creation of an ButtonActionAgent instance. An action agent exists for widgets that are associated with actions. Clearly not all widgets are associated with actions. So far in MCT there are 18: JButton, JCheckBox, JComboBox, JEditorPane, JLabel, JList, JPasswordField, JProgressBar, JRadioButton, JSlider, JSpinner, JTabbedPane, JTable, JTextArea, JTextField, JToggleButton, JTree, and ViewPort. Action agents bind a widget and the associated model to property changes. In parseMCTXML the ButtonActionAgent is created (at **1**), for this button, and then it is stored in a global agent store (at **2** and **3**).

The second stage in parseMCTXML, for MCTJButton, is to get the action name from the MCTGUIComponentInfo object parsed earlier. First the global action manager is retrieved, along with the store for action bindings. Then a binding (ComponentActionParams) is created between the button and its action params. Finally this binding is related to the action name and stored in the action store. All of these steps are shown at **4**.

The third stage binds the button to its action listener. The global action listener manager is retrieved, along with the action listener, and the listener is added to the button (at **5**).

Finally, the global GUI info manager is accessed, along with its store, and a binding between the button and its GUI info is added to the store (at **6**).

GUI Binding to View Roles

As shown in the MCTComponents schema (Figure 22:), each ViewRole potentially has an element named GUISpecification. GUISpecification is a reference to a primary/top-level MCT/Swing container type. Each ViewRole will thus always refer to a Swing container, regardless of what other widgets are embedded in the container. The GUISpecification is the only reference point a role has to its GUI. The GUI has a back pointer to both its parent role and its parent component that are attributes of its MCTGUIComponentInfo object, so if the GUI is selected at run time both its parent role and component are available.

For every primary view there is a single root element or root component and its associated preferred view. In MCT, all component descriptions will be in a single XML file associated with this root element. If it becomes necessary to swap this view for another, then another XML file is parsed and instantiated. This provides a degree of flexibility while limiting the amount of overhead in what is parsed and rendered at any given time. Clearly it would not make sense to parse and render items that aren't in the current view. Not only that, but there may be many dependencies at play for a particular view that might not be shared by other views (such as localizations, rule groups, etc.).

So each primary view is associated with the GUI description of everything it contains (in the Swing context). This makes it much easier to decide what to parse. Of course, the top-level, or primary, component defines component parts and roles that it is associated with, and not just a GUI. The way the GUI is referenced by the primary view or Component is through its *preferred* view role. So the GUI that is parsed is the top-level MCT/Swing container of the primary view's preferred view role (whew!). The reason that this is articulated is that in each view role's definition there may be a separate GUISpecification and in only one of these should it be necessary to fully articulate the GUI. Any others should reference the GUI by id reference.

GUI Management

As mentioned in Figure 25:, there are two classes that are primarily responsible for managing the GUI: FrameworkModelMgr and MCTGUIBuilder. The launcher for a view extends MCTGUIBuilder so this class is responsible for constructing and managing a primary view. For example, it will be responsible for initiating the loading and parsing of all GUI-specific files (gui definition, model definition, validation, and customization), initializing the FrameworkModelMgr instance, initializing the localization resource bundle, creating the models, providing the means to reference GUIs and models, and managing subviews.

FrameworkModelMgr is defined and referenced inside MCTGUIBuilder and is responsible for managing the various models used by the GUI associated with a primary view, including validation.

Component Toolkit Models

The term *model* used in the context of MCT GUI widgets is a mapping between a single name and a single value (i.e., a name/value pair) that can be used anywhere in the view context but references a specific Model Role definition. This mechanism allows for the usage of an object aspect in numerous view contexts without referring to its original semantic definition and dependencies per se. A model is used in a GUI widget and must be mapped to the associated Model Role. The Component Toolkit is responsible for managing Model Role, model, and GUI interactions.

Model Role to model Naming Convention

In a GUI widget declarative form one can only reference a model by name (attribute values must be scalars), so the Model object/attribute naming approach must be collapsed in the model naming convention so that it uniquely references a Model Role instance and its attribute. Moreover, there are different kinds of models: some refer to the object itself while some refer to its attributes, some are read only, some are read/write, some are collections, and each of these should be managed differently.

Model Types

MCT implements one object-level model type (ModelValueHolder) and three attribute-level model types: (1) AttributeModel, which represents a scalar attribute but need not be buffered because it is not intended for buffered change; (2) BufferModel, which represents an attribute model whose value is expected to change and must be validated; and (3) ListModel, which represents a collection of values with a current value selected. The class diagram illustrating the relationships between these model types is shown in Figure 31:

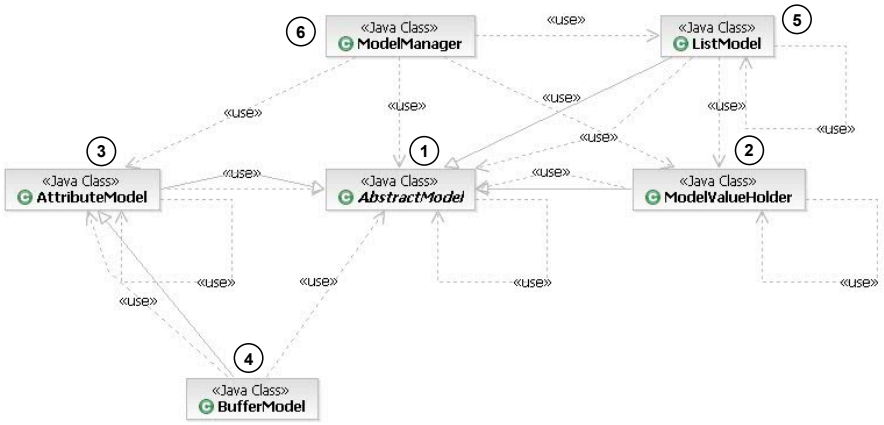


Figure 31: MCT model types.

The central class for this hierarchy is AbstractModel (at **1**), followed by ModelValueHolder (at **2**), AttributeModel (at **3**), BufferModel (at **4**), ListModel (at **5**), and a ModelManager (at **6**). The reason for having different model types is both that they perform different tasks in different ways, but more importantly because there is no point in searching a large collection of models the large percentage of which might not be applicable to a given task. Organizing models by what types of operations that might be performed on them is a way to improve performance.

AbstractModel

AbstractModel provides the skeleton for the model groups. Basically, it provides access and control over the name and value pair, but this extends to initialization, change detection, and validation of the value. AbstractModel implements PropertyChangeListener. There is a member called sig that represents the root name and is used to generate the

getter and setter method names which enable interaction with the Model Role. There are also methods for converting the value to and from its defined type.

ModelValueHolder

A ModelValueHolder represents a structured object. ModelValueHolder is a concrete class that implements the notion that a structured object can be assigned a value. For example, if we define a class Person, we could say:

```
Person me = new Person();
```

When we take a Person definition we are talking about a structured object that can be assigned a structured value, or accessed as a structured value, but only as a structured object and not as a collection of attributes. That is, we can access the object's value but not its attributes. It is a holder or reference.

A ModelValueHolder must have a method for accessing the object. In MCT this method is wrapped inside the <value>getterMethodName</value> tag in FrameworkModles.xml.

AttributeModel

AttributeModel is a concrete class for representing object attributes. It doesn't represent the parent holder, only a particular attribute. AttributeModel extends AbstractModel and overrides some of the default method implementations but adds methods associated with attribute names and values, along with the validate method. The attribute model must have an object reference such as illustrated below:

```
UserEnvironment.UserEnvironmentName
```

In this case UserEnvironment provides a reference to the object that has UserEnvironmentName as an attribute. The AttributeModel need not have an accessor because the model follows the bean convention that the accessors will be `"get" + AttributeName` and `"set" + AttributeName`, or `getUserEnvironmentName` and `setUserEnvironmentName`.

BufferModel

BufferModel is a concrete class for representing objects whose values change and need buffering. It extends AttributeModel and overrides some super class methods while providing methods for setting dirtiness, saving, and reverting to a previous valid value. The buffered model has a type reference in the GUI that looks as follows:

```
BUF::UserEnvironment.title
```

The `"BUF::"` is the only differentiation between the BufferModel and the AttributeModel during parse.

ListModel

ListModel is a concrete class for representing value collections (particularly arrays) where at any given time there may be a *selected* value. It extends AbstractModel. It has members and methods that differ somewhat from the previous classes due to its structure but are otherwise isomorphic. Like the ModelValueHolder, ListModel requires [possibly 3] method names:

```
<list>getterMethodName</list>

<current>currentItemGetterMethodName</current>
```

```
<setselected>selectedItemGetterMethodName</setselectedItem>
```

As mentioned, the <list> tag is used to identify the method name used to get the list contents. The <current> tag is used to identify the method name used to get the currently selected item in the collection, while the <setselected> tag is used to assign the currently selected item in the collection.

Model Use

The calling sequences that *connect* a data source to a GUI widget, which are mediated by the Component Toolkit, cannot be appreciated without looking more carefully at how the model layer structurally fits in with both Model roles and GUI widgets. The way models are organized in MCT GUI definitions and Model Roles is shown pictorially in Figure 32:

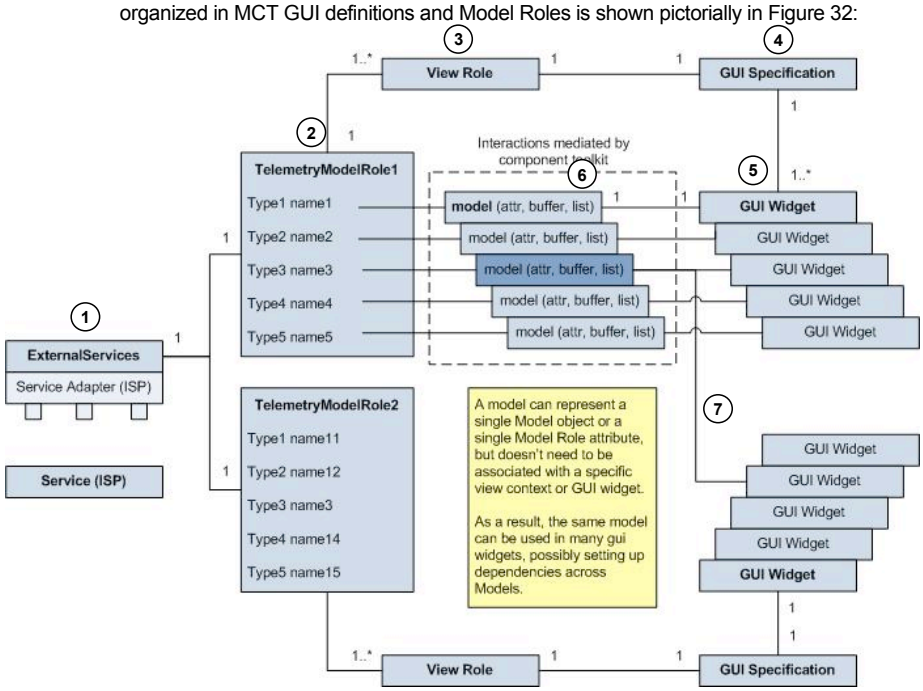


Figure 32: MCT Model Role and View Role relationships to GUI widgets and model elements.

This figure shows that the MCT ExternalServices API (at **1**) allows an external service such as ISP to populate any number of Telemetry Model Roles (such as at **2**). Each Model Role may be described, as could any POJO, by a number of attributes. These attributes define a semantic (domain) object and are required to recognize that object as a semantic whole. Each Model Role can be associated with many View Roles (at **3**). Each View Role is associated with a single GUI Specification (at **4**). That GUI specification can be comprised of any number of GUI Widget elements (at **5**), and each of those elements can be associated with a single model element (at **6**). That model element is unique, but it isn't unique to a particular GUI Specification (see for example, **7**).

An example of how models are used in a GUI is shown in Figure 33:

```
<Label horizontalAlignment="center" (1)
       id="HelloWorldLabel"
       text="foobar"
       model="BUF::UserEnvironment.UserEnvironmentName"> (2)
  <Font name="Dialog" size="10" style="bold"/>
  <MinimumSize>
    <Dimension height="16" width="27"/>
  </MinimumSize>
  <PreferredSize>
    <Dimension height="20" width="27"/>
  </PreferredSize>
</Label>
```

Figure 33: Example model usage in GUI file, in HelloWorldLabel.

Above is an example from a current demo GUI file illustrating the use of a BufferModel to bind the value of an MCTJLabel (using the Label element, at **1**) to a Model Role named UserEnvironment and attribute UserEnvironmentName (at **2**). Notice that model is an XML attribute and not an XML element. Model is part of the MCTGUIComponentInfo object that is parsed for each GUI widget. It contains the information necessary to deference the model to a model type, a Model Role, and a Model Role attribute. In this example, the environment name uses a BufferModel and the "BUF::" notation is used by the toolkit (specifically MCTGUIBuilder and FrameworkModelMgr) to create the BufferModel instance. The naming convention must be adhered to. If an object is referenced that doesn't exist in the role registry at the time the binding is needed then, in this case, the label will take the defined text value of "foobar" rather than the intended value of whatever is defined, or expected to be defined, in the UserEnvironmentName attribute. Likewise, for every attribute so referenced, there must be a Java bean reference to the attribute (i.e., getUserEnvironmentName and setUserEnvironmentName) or the dereferencing cannot complete. It is the responsibility of the model to convert "UserEnvironmentName" to the getter and setter method names.

A more elaborate diagram illustrating the relationship between Model Roles, View Roles, and models in the Component Toolkit is shown in Figure 34:

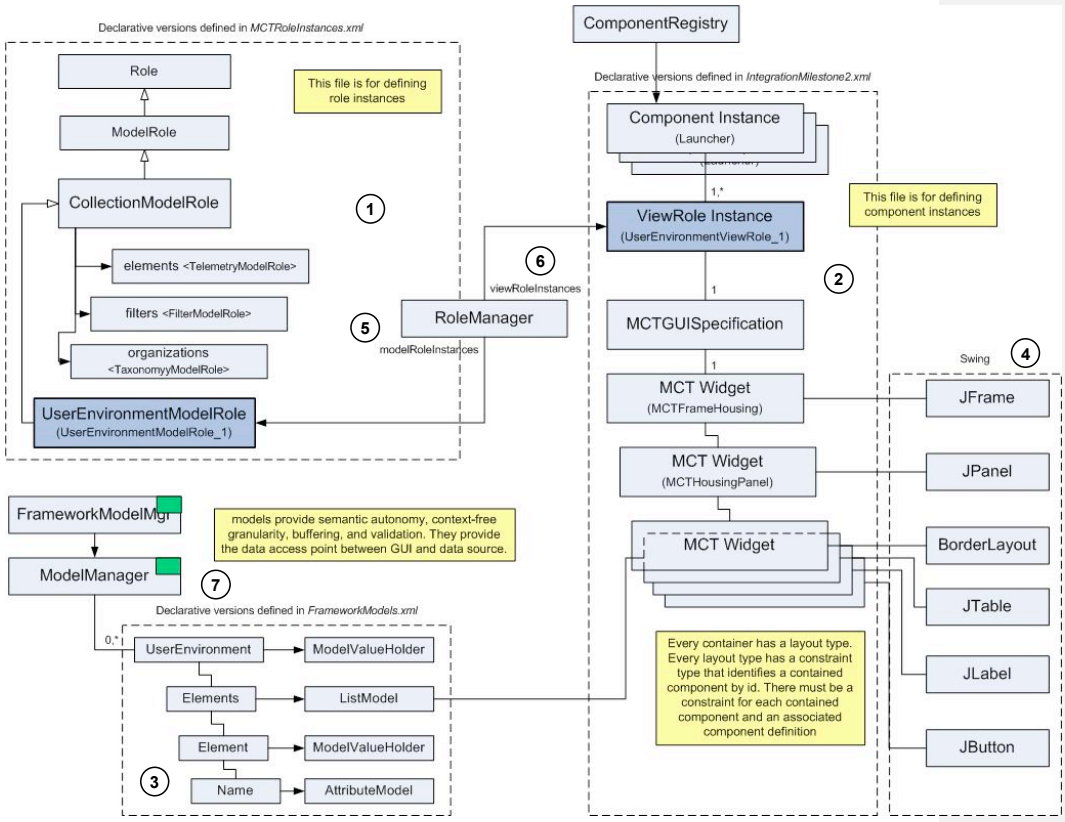


Figure 34: Model Role, View Role, and model relationships in MCT.

This is unfortunately a busy diagram, but there are 4 main areas, delimited by dotted-line boxes, that can be discussed: The classes that will be parsed into by reading the Components/GUI file, *MCTRoleInstances.xml* (at **1**), the classes that will be parsed into by reading the role template file, *IntegrationMilestone2.xml* (at **2**), the classes that will be parsed into by reading the models file, FrameworkModels.xml (at **3**), and the Swing components (at **4**). The diagram is based on an example using UserEnvironment as a basis.

As mentioned previously (from the MCTComponents XML Schema), UserEnvironmentModelRole extends the CollectionModelRole class that has elements, filters, and organizations. For the purposes of this discussion, we are interested only in the elements collection (currently defined on components that implement the TelemetryModelRole Role). When the parsing of *MCTRoleInstances.xml* is complete, all Role template definitions, particularly UserEnvironmentModelRole_1, will have been parsed and registered with the Role registry in its modelRoleInstances attribute (at **5**). The elements collection will be accessible by accessing the role from the RoleManager by id (provided in the XML file).

The contents of the *IntegrationMilestone2.xml* file are Component definitions along with their associated Role and GUI definitions particularly for the primary view. Of particular interest is the View Role UserEnvironmentViewRole_1, which during the parse will identify and parse the Role's MCTGUISpecification, which decomposes to MCT aggregate widgetry such as Housings and MCT Swing equivalent widgetry. The result of the parse will be the addition of these components and roles into the component and role registries, in the case of the role registry into the viewRoleInstances attribute (at **6**). The View Role instance and its GUI will then be accessible through the RoleManager by id (provided in the XML file).

The contents of the *FrameworkModels.xml* file are model definitions that are used in the Components/GUI file and map back to the Model Role definitions. In this example the model for UserEnvironment is shown, indicating the Elements ListModel, the Element within it ModelValueHolder, and the Name within Element AttributeModel. Models cannot be further divided into more granular items because of the way they are currently parsed. When the parse is complete the models are added into the ModelManager, which is managed by FrameworkModelMgr (at **7**). The models are accessible from these classes, but no by id because they have no id. The figure above will be reference in other diagrams that illustrate significant call sequences in the Component Toolkit that mediate between Model Role instances and GUI widgets. Prior to that we will look a little more closely at models and how they are constructed.

Model Definition

The model definitions used in GUI elements must be mapped to their semantic parent Model role instances. An XML file called *FrameworkModels.xml* is used to accomplish this, and this XML file is parsed in the same context as the GUI file. Currently all XML files in MCT are validated against a corresponding schema. This becomes problematic with the models file because the models file represents model instances and one would never want to construct a schema that represented instances because it is counterintuitive to what schema does well – define abstractions. For the time being we suspend validation of the models file but a better solution is to change the schema as shown in Figure 35:

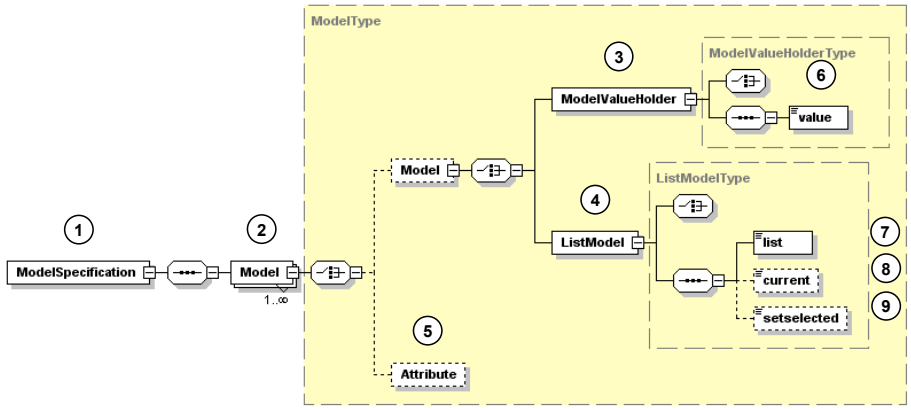


Figure 35: MCTModels.xsd schema.

The MCTModels schema has a root element called ModelSpecification (at **1**) which is comprised of any number of Model (at **2**) elements. Model can be of three types:

ModelValueHolder (at **3**), List (at **4**), or Attribute (at **5**). The ModelValueHolder additionally has a required *value* element (at **6**) that provides the name of the getter method defined in the Model Role used to get the object instantiation. List additionally has a required *list* element (at **7**) that performs the same role but for the list. It also has a *current* element (at **8**) that provides the getter method for the currently-selected item, and a *setselected* element (at **9**) that provides the setter method name to select an item. Any other Elements defined on the model represent attributes of the model.

An example illustrating a portion of a current model file (not explicitly following the schema shown above) is shown in Figure 36:

```
<?xml version="1.0" encoding="UTF-8"?>                          (1)
<ModelSpecification>                                            (2)
  <Models>
    <HelloWorldModelRole type="ModelValueHolder">
      <value>getHelloWorldModel</value>
        <Data/>
    </HelloWorldModelRole>
    <LoginModelRole type="ModelValueHolder">
      <value>getLogin</value>
      <Password/>
    </LoginModelRole>
    <UserEnvironment type="ModelValueHolder">                   (4)
      <value>getUserEnvironment</value>                         (5)
      <UserEnvironmentName/>                                    (6)
      <UserEnvironmentDescription/>                             (7)
      <Source type="ModelValueHolder">                          (8)
        <value>getSource</value>
        <SourceName/>
      </Source>                                                 (3)
      <Elements type="ListModel">                               (9)
        <list>getElements</list>                                (10)
        <Element>
          <Name/>
          <Value/>
        </Element>
      </Elements>
    </UserEnvironment>
  </Models>
</ModelSpecification>
```

Figure 36: Portion of FrameworkModels.xml.

So far, for the purposes of discussion, the only item in FrameworkModels.xml that is in use is UserEnvironment. The ModelSpecification as a whole (at **1**) is comprised of a collection of any number of Models (at **2**). The Model name must map to a Model Role delegate. Every model mapping *type* for the environment will appear in this (or a similar) file. The block shown at **3** illustrates a moderately complex model. Future versions of this file will provide more complex examples. The UserEnvironment model is bound to a UserEnvironmentModelRole instance. Not that UserEnvironment is of type ModelValueHolder (at **4**). This means that it is a structured object that can be assigned a value. ModelValueHolder is a class in the *platform.comp.gui.models* package of the

component toolkit. In this model mapping, to get the value for the model we provide a mapping to the Model Role's accessor method, "getUserEnvironment" (at **5**). This must map to a method defined on the definition associated with the object the UserEnvironment delegate references. Four attributes are defined for this Model in this example, and they are UserEnvironmentName (at **6**), UserEnvironmentDescription (at **7**), Source (at **8**), and Elements (at **9**). Clearly this is an early definition of UserEnvironmentModelRole since it doesn't have an appropriate definition of its collections or their types but it is acceptable for this discussion. For simple attributes the type of declaration shown in **6** and **7** is appropriate. When an attribute is structured, then a type is required. Thus Source (at **8**) and Elements (at **9**) require types. Whenever there is no type AttributeModel will be assumed to be the type. When the type is "ValueHolderModel" a *value* tag is required to tell the ValueHolderModel how to access the associated object value. When the type is "ListModel" a *list* tag (at **10**) is required to tell how to access the associated list attribute.

Model Parsing

The models file is parsed in the same context as the GUI file. In fact, the models file is read before the GUI file is parsed, from FrameworkLauncher.launch (FrameworkModelMgr.initModel). Since the launcher extends MCTGUIBuilder and has an instance of FrameworkModelMgr, the launcher defines initial values for file paths and assigns them into the FrameworkModelMgr instance. When initialization is performed the values have already been defined for which file to use for the models. The names to use are actually stored in *mct.properties*.

The parsing process starts with the FrameworkModelMgr.initModel() method as shown in Figure 37:

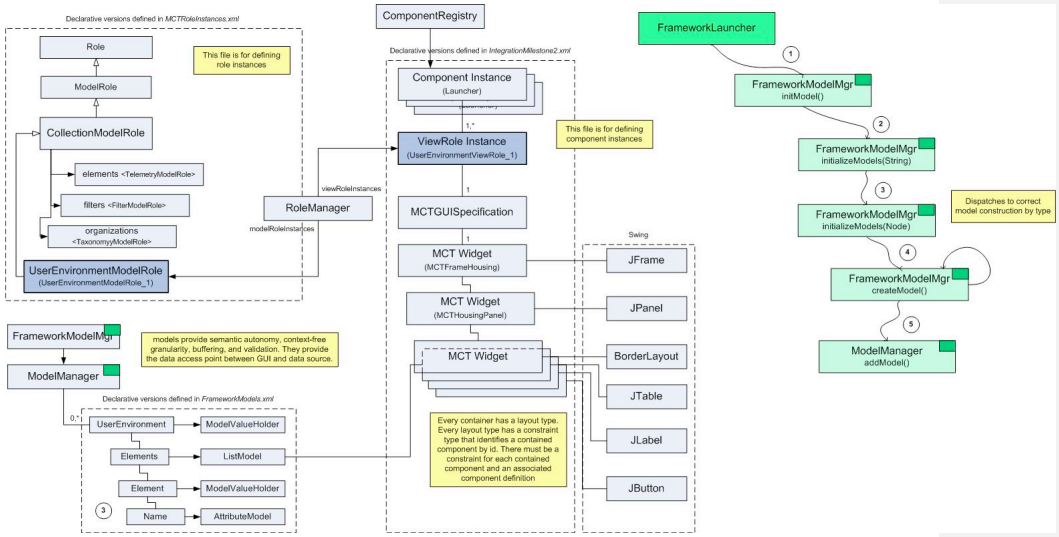


Figure 37: model parsing process.

Most of the model parsing process takes place in FrameworkModelMgr. The call to createModel parses the type attribute and dispatches to the correct model parser (createAttributeModel, createListModel, createValueHolderModel). These methods parse

the attributes and elements, assign them appropriately into said model type, and add the model into the ModelManager.

Model Initialization

After the Models, models, and GUIs have been parsed and registered the models can be initialized. This process entails checking to see if the model has been initialized and, if not, acquiring data from the Model. The initialization process is recursive and starts with the MCTGUIBuilder.initializeGUIModels() method, as shown in Figure 38:

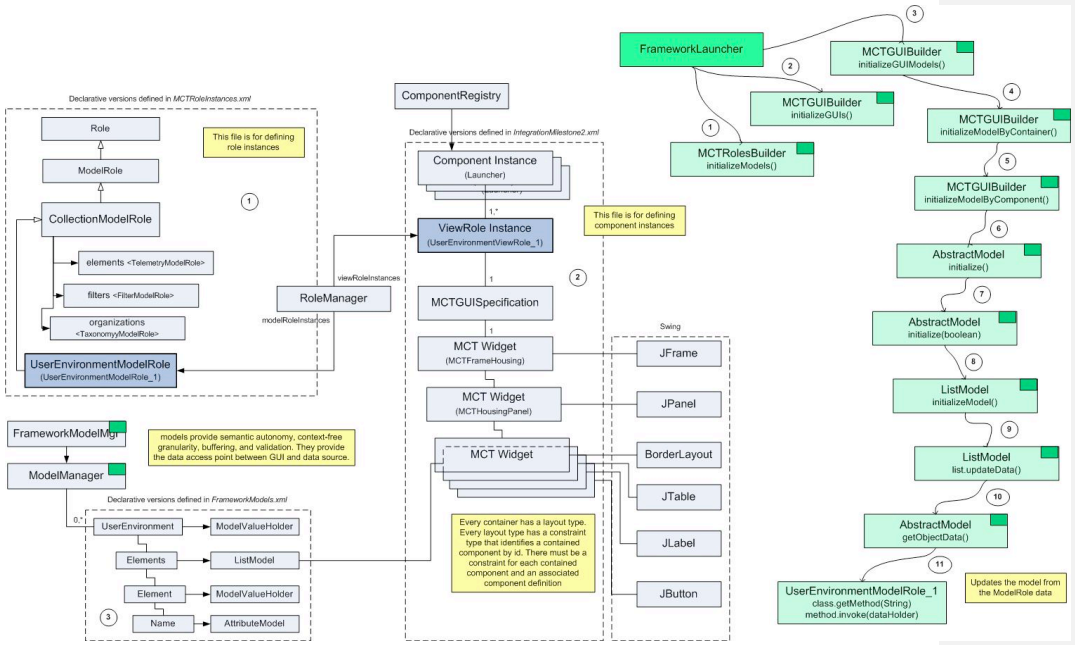


Figure 38: model initialization process.

As can be seen, the first few steps in FrameworkLauncher (highlighted in green) identify parsing processes identified earlier. The elaborated path (highlighted in light green) depicts the more significant steps in model initialization. The most significant steps are 8-11 where the model of interest (in this case ListModel) is asked to initialize, it is asked to call updateData, this calls AbstractModel.getObjectData(), which eventually uses the getter method name (parsed from the models file) and reflection to get the data value. Since AbstractModel implements PropertyChangeListener, and since the Swing widgetry all responds to PropertyChange events, initializing the model will also initialize the Swing widgetry that are associated with the model.

Model Management

The models are managed by primary view using the FrameworkModelMgr class. This class manages all of the models using another class called ModelManager. ModelManager is like our component and role registry except that it isn't global to the

framework. It is specific to a view. This helps to keep model management a bit more controllable.

Reference to models

Models are part of the component toolkit. As such they are specific to the GUI and its management and not a part of the MCT Component Model per se. They are referenced in two places: (1) in the GUI file itself, as previously shown, and (2) in the View roles that make use of that model. Generally speaking, in the view role there is a member for every model used in the GUI that is actionable (modifiable) because a standard operation on the view role is to check to see if anything has changed or to revert an invalid change. So any model that can be affected by such operations, which is often all of them, would have a local version in the view role definition.

Since every widget has change listeners (if not action listeners) any change to the model value would be 'heard' right away by the widget and it would be redrawn. As such, no direct reference to the widget is generally required. If so, however, widgets can be referenced by their unique ids and their models can be accessed as well. What needs to be added into the component toolkit is the ability to dynamically update the Model instance and have the associated models/GUI widgets react.

Model Role, model, and GUI Interactions

The previous section articulated where model references are found but not how they work. In this section two sequences are discussed: (1) GUI update, and (2) Model update. The former is fairly straight forward since the GUI/model integration is well defined. The latter scenario requires a ModelManager lookup for the related object model.

GUI/model/Model Interaction

Changes initiated at the GUI can be illustrated with a simple example of pressing a button in the GUI and updating a label, as shown in Figure 39:

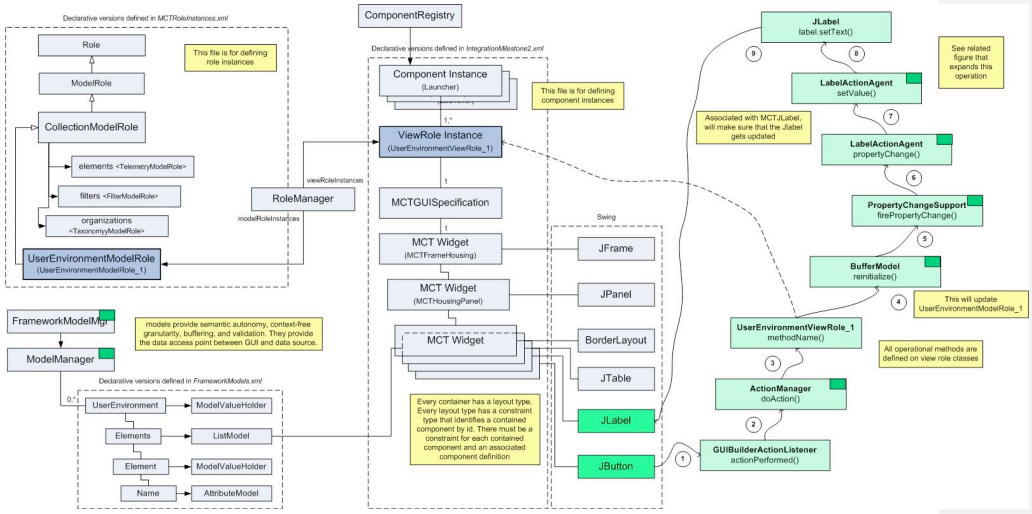


Figure 39: GUI->model->Model interaction in the Component Toolkit.

In this interaction, the GUI widget (MCTJButton -> JButton) will be selected (as highlighted in green) and the result will be a change in the JLabel object associated with MCTJLabel in the GUI (also highlighted in green). The sequence involves GUIBuilderActionListener, ActionManager, the associated ViewRole (UserEnvironmentViewRole_1), the associated model (BufferModel), and the MCTJLabel's action agent (LabelActionAgent).

When the initial button press is made the primary view's GUIBuilderActionListener responds: it gets the action object and calls doAction in the ActionManager with the name of the operation being performed. The ActionManager's role is to pair that widget (Swing) with the appropriate MCT widget and operation and to invoke the operation. To do this the class identifies the action scope (i.e., the View Role instance) that is capable of responding. This is always found in the XML for the action using the same nomenclature as for models. That is, ActionScope.ActionToPerform. All ViewRoles must implement a method called call() that is invoked with the ActionToPerform name, which dispatches to the appropriate method in the class. In this case the ActionScope is UserEnvironment, which points to UserEnvironmentViewRole_1, and the method dispatched to is shown in Figure 40:

```
public void doUpdateLabel() {
    ModelManager                  manager       = (ModelManager) FrameworkModelMgr.getInstance().getModelManager();
    ModelDelegateMgr              delegateMgr   = FrameworkModelMgr.getInstance().getModelDelegateManager();
    BufferModel                   ueNameDataBuf = (BufferModel) manager.getModel("BUF::UserEnvironment.UserEnvironmentName"); (1)
    DefaultUserEnvironmentModelRole uemr;

    uemr = (DefaultUserEnvironmentModelRole) delegateMgr.getDelegate("UserEnvironment"); (2)
    uemr.setUserEnvironmentName("This is our world");
    ueNameDataBuf.reinitialize(); (3)
}
```

Figure 40: doUpdateLabel example method.

This method illustrates one usage of models in that the model is directly retrieved from the model manager by model name (at **1**). The second step is to get the Model Role instance by delegate name (at **2**) and assign it a new value. Then the retrieved model is asked to reinitialize (at **3**), which will go out to the bound Model and retrieve the new value and fire a property change listener.

Returning to the trace in Figure 39: the property change is picked up by the LabelActionAgent, which gets the new value and calls setValue(). This call sets the JLabel text attribute and the sequence is complete. This example is illustrative of the general approach to GUI->GUI update in the framework.

Model/model/GUI Interaction

Changes initiated at the data source can be illustrated with a simple example of updating a Model Role attribute from ISP and updating the related GUI table, as shown in Figure 41:

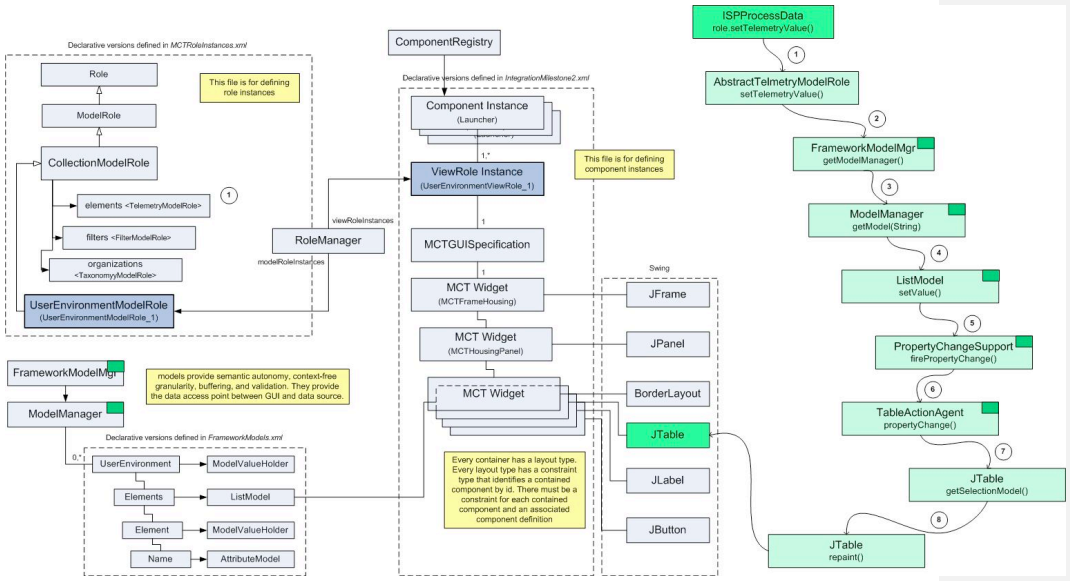


Figure 41: Model->model->GUI interaction in the Component Toolkit.

The sequence shown above has not yet been implemented, and has some drawbacks that should be mentioned. Most importantly, the way updates are currently performed is on the TelemetryModelRole instance and not on the collection in its parent (UserEnvironmentModelRole), but the model is defined as UserEnvironment.Elements or UserEnvironment.Elements.Element. If the former is used, then there will be more updates on the collection than the number of items in the collection. It is not certain whether the latter can be tried since there is no identifier for Element. Another problem is that the nomenclature for referencing a model is: DelegateName.AttributeName, but the delegate of interest is associated with the super class (UserEnvironmentModelRole) and not the current class (TelemetryModelRole). So the name must somehow be provided by the time the setTelemetryValue() operation is called.

If these hurdles can be overcome then the GUI update is straight forward. The model name is provided to the ModelManager and then setValue() is called on the resulting model. This call will also result in a call to firePropertyChange that will be picked up by TableActionAgent, resulting in the JTable model being updated (highlighted in green).

Of course, with respect to MCT, this is probably the most complicated example possible but it serves to illustrate the approach of data-driven GUI updates and how they are mediated by the model layer. The same sequence would apply in general to any Model->widget update in the framework.

Customization and Nodal Configuration

GUI customization is a way for the MCT Framework to define GUIs for general use but to have modifications that can be loaded at launch time that override the basic definitions. These *customizations* can be associated with individuals or they can be more institutional.

At present the customization is limited to foreground and background colors, and fonts, but there is no limitation in general as to which GUI attributes could be customized.

Note that the GUI customization mentioned here is Nodal, meaning that it precedes the parsing of the GUI by one step and works on the Node defined in the parse tree. Run time customization would/might be a different animal.

GUI Nodal Customization

The customization process is simple:

At launch time the customizations are read/parsed

When the GUI file is read the customizations previously read are swapped at the nodal level.

When the GUI is rendered the customizations are applied.

Customization File

A simple example of MCT nodal GUI customization (filename = FrameworkComponentConfig.xml) is shown in Figure 42:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- $Id: FrameworkComponentConfig.xml,v 1.1 2008/02/09 07:26:44 jhodges Exp $ -->
<Configurations>
  <Configuration id="MCTTabPanel" visible="true">  (1)
    <Font name="tahoma" size="12" style="plain"/>  (2)
 (3) <BGColor>
      <Color>
        <IntColor alpha="255" blue="0" green="255" red="0"/>
      </Color>
    </BGColor>
 (4) <FGColor>
      <Color>
        <IntColor alpha="255" blue="255" green="0" red="0"/>
      </Color>
    </FGColor>
  </Configuration>
</Configurations>
```

Figure 42: An example of nodal GUI customization.

This file represents a simple schema that is comprised of an arbitrary number of Configuration (at **1**) elements, each of which references an id from the GUI file and any elements within that GUI component that are supported by the MCTSwing schema. In the example above only one GUI element is *customized*: MCTTabPanel: the font is changed (at **2**, from whatever it was) to 12 pt plain tahoma, the background color is changed (at **3**) from whatever it was to green, and the foreground color is changed (at **4**) from whatever it was to blue.

Customization File Parsing

The customization file, FrameworkComponentConfig.xml, is parsed in FrameworkLauncher. initializeModelManager () with a call to

ConfigurationManager.config(). The path to the configuration file is stored in mct.properties and is retrieved by initializeModelManager.

The ConfigurationManager.config method reads the configuration file and creates a new instance of Configuration based on the Document returned. The Configuration returned has all of the attributes and elements from the configuration file.

Customization

Configurations are actually applied while the GUI file is being parsed. At present only a few attributes, those associated with JComponent, are configurable, so the only place where configurations are assigned is in parserHelper.parseComponent. In this location a call is made to ConfigurationManager.configure based on the node being parsed, and the attributes in the configuration are swapped into the Node prior to parsing.

This same approach could be applied to any attributes and a copy of the configure call could be placed before the node parse in every GUI widget parser.

Widget Model Validation

As an information-based approach MCT-based declarative descripts can be validated by the Semantics Manager (or other point of origin) at the point in the startup sequence where they are loaded. At run time modifications to model values must be validated by the Validation service. These topics are address in their respective chapters.

Widget Composition and Aggregation

Composition of widgets is a design functionality and is discussed below and in the chapters relating to rule processing and composition.

Required Baseline Model Components and Representation Components

Central to the MCT environment is the notion of User Object as a UE-centric marriage of data model and UI model. The concept is that a user may view a data model in different ways but they all represent the data model. In MCT a component visualization is called a Representation Component, and a Model Component intended for viewing, or to be more precise a Representable Component, can have multiple Representations. The component library is constructed from model components and representations.

Baseline Model and View Role Types

The backbone for constructing populating user interfaces in MCT is the Model Role. Components are constructed from a library of Component types built on top of Model and View Role types, or component library. The library is not about the Component types, as there is no such thing, but about the domain-specific Role types. So in effect the Component library is really a Role library.

The baseline Role set satisfies the use cases defined for the Component Toolkit and provides a starting palette for the construction of new/domain-specific roles. The purpose of the baseline Role set is to provide a controllable user experience but to not otherwise limit the construction of new roles. The baseline role types and their requirements are detailed in Table 12:

Core Role	Required Feature
Housing Roles	• Container containing Housings or Items • 4 Types: Frames, Dialogs, Panels, Tabbed Panes • 5 areas: Title, Control, Content, Directory, Inspection, Menubar • Tabbed control in control area • Composition target • Controllable layout
FrameHousingRole	• Reveal control area widget • Status widget • Version number widget
Primary Window Control Area	• Menu support as toolbar • No window-closing 'X'
PanelHousingRole	• Collapse/expand widget • Reveal control area widget • Containment: L1 -> L2 -> L3 -> L4 -> …(flesh out functionality)
CollectionModelRole	• Filtering (available filters attribute) • Filter appears in Housing control area
TableViewRole	• Contain collections (representation for collections) • User objects (and Headings) can be column-based or row-based • Custom cell renderers • Row or column formatting
InspectorViewRole	• Any selected item • Inspection type configurable (different inspection reps)
TooltipViewRole	• Rollover inspection (only if tooltip is a rep) • Configurable
TimelineViewRole, TimespanViewRole, TimepointViewRole	• Timespan has a start time, an end time, and a duration • Timespan has a time scale • Timespans can contain timespans
PlotViewRole	• Axes configurable • Moveable limit lines • Legends
UserEnvironmentViewRole	• Multiple representations • Telemetry items
TelemetryGroupModelRole	• Can contain telemetry groups, telemetry parameter collections (multiple PUI), telemetry parameters (single PUI)
EvaluatorModelRole	• Algorithm applied to a component through composition • Source and target (can be same) • Can be sequenced
EventModelRole	• Threshold notification • Logging required • Notification required (to Entity) • Can be generated by limit evaluators • Have a timestamp • Logged based on related component • Multiple field/value support • Should be persisted to keep manageable
ReportViewRole	• Format • Configurable • Can include screen captures of screen regions

ClockViewRole	• Multiple representations • Have time, timezone, description, offset, synchronization
ProcedureViewRole	• Action sequences • Have name, description, number of steps • Have a collection of steps • Procedure step has parents and children, content, timestamp, order • Procedure step content refers to ordered actions by subjects on objects
CommandModelRole CommandViewRole	• Instructions to space-based hardware (mechanisms, devices, systems, subsystems, assemblies) having composition, connection, behavior, function, and use • Documentation • Can be drawn
VideoModelRole *VideoViewRole*	• Special Housing for video • Video controls (forward, backward, start, stop, pause, zoom in, zoom out, pan, …)
AudioModelRole *AudioViewRole*	• Multiple channel support • Switchable tracks • Audio controls (forward, backward, start, stop, pause, filtering, …)
BrowserViewRole	• Include browser window • Controllable events • Supports hyperlinks • Supports locally-supported packages • Supports security • Controllable port access
WorkflowModelRole *WorkflowViewRole*	• Graphical support (for procedures for example) • Nodes and links • Annotations on nodes and links

Table 9: Required baseline components in library.

This table represents a baseline component set. Given that general widget, design, and layout support is required in MCT, other components can be created through composition. Italicized items have not been made requirements in any known document but appear mandatory.

Application Design and Layout

The ability to construct MCT application interfaces rests solely with the ability to perform composition on existing or new components, but it also requires several capabilities that extend beyond simple composition. These capabilities are summarized as a design IDE in 16 requirements:

- The IDE is an integrated part of the MCT runtime environment so design templates that are mapped to models are populated with data.
- The IDE supports user access through identity-level roles.
- The IDE supports layout control and management.

- The IDE has a palette of composable component templates that matches the graphical/interactive components supported by the MCT framework (widgets and View role types).
- The IDE palette provides extendability by including user-created components.
- The IDE has a property inspector for components.
- The IDE is able to load and save project files in XML format.
- The IDE supports the enforcement of a set of UE-defined rules governing the construction of application interfaces.
- The IDE supports the mapping of GUI component instances to their Model role counterparts.
- The IDE acquires, if possible, information model capabilities from the associated ontology server.
- The IDE can perform model-based validation.
- The IDE has a rules editor/generator.
- The IDE can perform rule-based validation.
- The IDE can perform rule-based composition.
- The IDE has an action mapping capability.
- The IDE has a workflow interface that enables construction of multiple windows and dialogs and to simulate transfer of control whether or not the functionality and models associated with those components have been developed.

These requirements can be associated with 8 design-specific requirements, as depicted in Table 10:

Requirement	Use Cases?	Related Use Cases
UIT49: The design portion of the component toolkit shall support design project management.	Yes	• USER import/open design project into MCT • USER export design project into MCT • USER create design project in MCT • USER edit design project in MCT • USER save design project in MCT • USER delete design project in MCT
UIT50: The design IDE supports layout control and management.	Yes	• Add a display component to the view • Remove a display component from the view • Copy a display component to a new location • Move a display component to a new location

UIT51: The design portion of the component toolkit shall support GUI component editing.	Yes	• Edit component properties
UIT52: The component toolkit shall support model to component mapping in design mode.	Yes	• Map a model to a component
UIT53: The design portion of the component toolkit shall support rule creation in design mode.	Yes	• Create a rule • Remove a rule • Validate rules
UIT54: The design portion of the component toolkit shall support action management in design mode.	Yes	• Create an action • Map an action to a component
UIT55: The design portion of the component toolkit shall support undo/redo histories in design mode.	Yes	• Undo operation(s) • Redo operation(s)
UIT56: The design portion of the component toolkit shall support workflow creation and management in design mode.	Yes	• Create a workflow

Table 10: UI toolkit design-specific requirements and use cases.

The functionality described by these use cases will manifest itself in context-sensitive palette items, menus, and windows. The context will be based on the login role a user has selected. There will be a role hierarchy for using the design capability as defined below:

- **Normal User**: Normal composition in the runtime environment, no layout abilities
- **Integrator**: Normal User + simple layout abilities
- **UE Designer**: Integrator + look and feel control
- **Application Developer**: Integrator + template composition, rule construction, model mapping, action mapping, full layout support
- **Component Developer**: Application Developer + full widget palette
- **UE Developer**: Component Developer + look and feel control

This is not an exhaustive list. A user will be able to switch between design and runtime roles at will.

Design Palette

Added Menus

Added Windows

Design Capability Architecture

Chapter 4 Component Library

Component instances and their associated role instances provide the backbone for developing usage domain interfaces. They are built on top of the structure of the component model but provide reusable, application-level, functionality that is critical to rapid application development. MCT provides a core set of component role types and their associated GUIs in the Component Toolkit. Additionally, the framework supports an user-contributed, extendable, library of domain-specific component roles. Currently the library consists of components suitable for building telemetry applications but the framework supports the addition of new roles/GUIs. The next chapter shows how information models are decoupled from the MCT Framework while also providing information-gathering capabilities to the framework.

Introduction to View Roles and the MCT Component Library

An MCT view is comprised of GUI elements that can display anything one would expect to see in an application's user interface. In MCT everything the user can see and manipulate on the screen is called a user object. Some of these are constructed from MCT GUI widgets (such as buttons), but most are constructed using MCT component instances that have aggregate GUI views and data models in the form of View Roles and Model Roles. The view role provides the user interface and the model role provides the binding to a data model.

MCT View Roles have a guiSpec that defines the root GUI widget container and provides limited access to that structure. As part of an MCT component, the View Role achieves the notion of inheritance through its role ancestry. Role behavior is achieved through actions defined on the role and the interactions of the GUIs organized by the Role.

The combination of MCT components, model roles, and view roles provides a basis for constructing a library of reusable components that extend the foundation/core component roles provided by the Component Toolkit. This foundation is made possible by a hierarchical description of user objects at the role level.

Constraints Limiting Component Design

As an extension of the Component Toolkit the Component Library shares all of the constraints and requirements of the toolkit. The Component Library is comprised of components that extend those of the Component Toolkit and, as such, satisfy the look and feel that the MCT UE team, in conjunction with NASA flight controllers, have determined is optimal for the MCT environment. Nonetheless, the Component Library has its own specific component requirements in terms of providing baseline functional capability that can be used directly in usage domains. These component types represent a body of functional/visual capability that extend the MCT GUI widget set.

User Objects, Representable Components, and Representations

Every visual object in MCT is either a core (i.e., MCT GUI) widget or a User Object. A User Object is a visual representation of something meaningful to a user. User Objects are comprised of a visual rendering and a model. The model is bound to a data model and provides value. The model is also part of a Model Role since it can have a visualization associated with it. An MCT component is minimally defined as a View Role that is possibly bound to a Model Role (and an associated user interface description which defines how the GUI and data model will interact).

A Model Role may have one or more View Roles, meaning that it can be viewed in any number of ways. A View Role, on the other hand, may or may not have a Model Role. A View Role whose guiSpec points to a panel GUI, for example, would not be expected to have a Model Role, but the items organized by the panel would. A View Role, as a visualization, has a 1:1 relationship with a user interface.

Component Library Requirements and Use Cases

The use cases derived for the component library are shown in Table 11:

Requirement	Use Cases?	Related Use Cases

CL1: Users can adjust a view's content area visualization via the view's control area.	Yes	• Entity edit [plot] control panel
CL2: Users can adjust a view's content area visualization via the view's filter area.	Yes	• Entity modify [filter] control controls
CL3: Users can adjust th		•
CL4:		•
CL5:		•
CL6:		•

Table 11: Component library requirements and use cases associated with user objects.

It should be noted that this is a limited subset of use cases derived from the current functional specification and is not intended (for the moment) to illustrate the full scope of component functionality.

Required Baseline Model Components and Representation Components

Central to the MCT environment is the notion of User Object as a UE-centric marriage of data model and UI model. The concept is that a user may view a data model in different ways but they all represent the data model. In MCT a component visualization is called a Representation Component, and a Model Component intended for viewing, or to be more precise a Representable Component, can have multiple Representations. The component library is constructed from model components and representations.

Baseline Model Components

The backbone for constructing populating user interfaces in MCT is the Model Role. Components are constructed from a library of Component types built on top of Model and View Role types, or component library. The library is not about the Component types, as there is no such thing, but about the domain-specific Role types. So in effect the Component library is really a Role library.

Baseline Representation Components

The backbone for constructing user interfaces in MCT is the combination of core UI widgets and pre-defined Representations. Representations are constructed from a library of Representation Component types, or component library.

The component library is comprised of a set of baseline components satisfying the use cases defined above and providing a starting palette for the construction of new components. The purpose of the baseline component library is to provide a controllable user experience but to not otherwise limit the construction of new components. The baseline components and their requirements are detailed in Table 12:

Required Component	Required Feature
Housings	• Container containing Housings or Items • 3 Types: Primary Windows, Dialogs, Panels • 3 areas: Title, Control, Content • Tabbed control in control area
Primary Window Housing	• Reveal control area widget • Status widget • Version number widget

Primary Window Control Area	• Menu support as toolbar • No window-closing 'X'
Panel Housing	• Collapse/expand widget • Reveal control area widget • Containment: L1 -> L2 -> L3 -> L4 -> …(flesh out functionality)
Collections (not a rep)	• Filtering (available filters attribute) • Filter appears in Housing control area
ListRep	• Contain collections (representation for collections) • User objects (and Headings) can be column-based or row-based • Custom cell renderers • Row or column formatting
Inspector	• Any selected item • Inspection type configurable (different inspection reps)
Tooltip (not a rep)	• Rollover inspection (only if tooltip is a rep) • Configurable
Timeline, Timespan, Timepoint	• Timespan has a start time, an end time, and a duration • Timespan has a time scale • Timespans can contain timespans
Plotting	• Axes configurable • Moveable limit lines • Legends
User Environment	• Multiple representations • Telemetry items
Telemetry Group (collection)	• Can contain telemetry groups, telemetry parameter collections (multiple PUI), telemetry parameters (single PUI)
Scratchpad	• Composition target Primary Window • Predefined layout • Inspector support
Evaluator	• Algorithm applied to a component through composition • Source and target (can be same) • Can be sequenced
Event	• Threshold notification • Logging required • Notification required (to Entity) • Can be generated by limit evaluators • Have a timestamp • Logged based on related component • Multiple field/value support • Should be persisted to keep manageable
Report	• Format • Configurable • Can include screen captures of screen regions
Clock	• Multiple representations • Have time, timezone, description, offset, synchronization
Procedure	• Action sequences • Have name, description, number of steps • Have a collection of steps • Procedure step has parents and children, content, timestamp, order • Procedure step content refers to ordered actions by subjects on objects

Command	• Instructions to space-based hardware (mechanisms, devices, systems, subsystems, assemblies) having composition, connection, behavior, function, and use • Documentation • Can be drawn
Video feed	• Special Housing for video • Video controls (forward, backward, start, stop, pause, zoom in, zoom out, pan, …)
Audio feed	• Multiple channel support • Switchable tracks • Audio controls (forward, backward, start, stop, pause, filtering, …)
Browser	• Include browser window • Controllable events • Supports hyperlinks • Supports locally-supported packages • Supports security • Controllable port access
Workflow	• Graphical support (for procedures for example) • Nodes and links • Annotations on nodes and links

Table 12: Required baseline components in library.

This table represents a baseline component set. Given that general widget, design, and layout support is required in MCT, other components can be created through composition. Italicized items have not been made requirements in any known document but appear mandatory.

Representation Instance Library

MCT is supplied with representation components suitable for constructing telemetry applications.

Summary

The component library provides a foundation for constructing applications by providing reference implementations of commonly-used representation components. These components can be tailored to use in a variety of applications.

Chapter 5 Information Semantics Management

MCT is an information model dependent framework. Component and representation conceptual definitions are stored in what is called an ontology server. Internally the MCT framework works with java component implementations. The information semantics manager is responsible for translating ontological information to the MCT framework and vice versa.

Introduction to Information Semantics Management

MCT is an information model approach to building applications. In the recent past application information models were stored in relational databases and interacted with predefined UI applications through transaction management systems. These were very static models and schemata changes were extremely difficult to make, requiring massive work to the database and every layer up from there. More recently information models have been made more transparent between the repository and the client, with the client requesting an object and a similar transaction management system negotiating the interaction. But the clients were still, for the most part, prebuilt, and the data models were still, for the most part, difficult to modify. MCT introduces the notion of a dynamic set of UI components and a dynamic information model. What this means is that the information model should be able to change without breaking the user interface application, and the user interface application should be able to change the local data model without breaking the information model. This approach has dramatic implications on the development of client applications.

The MCT component model has been shown to be a flexible foundation from which this approach can be implemented, and that is a substantial aspect of the system. Just as important, however, is the mechanism that provides access to the information models and maintains them at runtime with respect to the component models and instances that comprise an application. There are many issues, but the primary role of this system, called the Information Semantics Manager (or ISM), is to act as an information broker between the information model repository and the MCT Framework. Its role is not to persist objects from the application[5]. The ISM's role is to guarantee that the models being used are current with respect to the information model and that changes that take place at either end are conveyed to the other. It also has the role of providing access to information models by the MCT Framework.

In MCT conceptual information models take the form of ontologies and are stored in a long term repository called an ontology server. The ontology server can be queried in the same fashion as a database server, only it can provide information about concepts, concept definitions, concept inheritance, concept causality, etc.

Information Model Types

MCT has need of 3 types of information models: (1) an information model defining the structure and behavior of components and roles, which we have referred to as the component model/ontology; (2) information models for component instances, and (3) information models that contain domain conceptual information. MCT relies on the ontology server as a common source for its conceptual models so that diverse sources can update and synchronize their component definitions and so that the baseline definitions can be updated independently of the client applications that use them.

The Information Semantics Management (ISM) subsystem provides access to ontologies by the client application. When an application is initially launched it is the ISM that loads the baseline component model that forms the basis for the construction of component instances that will make up the application's representations. When updates occur in the ontology it is the ISM that is responsible for conveying these changes to the component

[5] That is another task and another aspect of the system. See Chapter 15.

model and application representation components. The ISM is both a control and a translation module.

Constraints and Requirements that Inform ISM Design

The ISM is a complex subsystem because it provides several types of functionality and integrates external functionality with its own. Also, the ISM provides fundamental services to the rest of the MCT framework which makes it a key subsystem. The design of the ISM is informed by 12 constraints:

- The Information Architecture (IA) group, which will provide information models through their ontology server, is using STCE (RDF, RDFS, and OWL-DL) to represent conceptual relationships. MCT must adhere to their content formats. IA also produces an XTCE schema version of STCE.
- Information models are defined using a structured language (RDF, OWL, and XML Schema, see above).
- The Information Architecture group is using a MySQL triple store for their repository.
- NASA Johnson (JSC) is using XTCE to represent its command and control metadata, and MCT must adhere to their format.
- The ISM subsystem has limited state of its own but must maintain intermediate states of all information models.
- As information models are updated the ISM must notify the UserPlatform to update component instances appropriately.
- The ISM manages all information models loaded into the framework.
- Information models can be queried from the ISM.
- The ISM provides integration services for data and metadata with the External Services subsystem.
- Information models can be used to construct components.
- Information models can be used by any subsystem.
- Information models can be exported with component information.

ISM Design Approach

The foundation for the Information Semantics Manager approach is based on three ideas. First, MCT is information based but that the information will not reside in MCT but in another repository. Second, the information will take the form of a conceptual model rather than a structural model. Third, the information models made available with the ISM must be accessible to any service or subsystem in the framework, and changes to the information model should be reflected in MCT components. As a result, the ISM provides functionality to the framework but also provides this functionality through external sources and must manage the information received from those sources in the same manner that one would have a transaction management system manage views rather than to keep going to a database to retrieve relational data. Also, the ISM is intended to provide services to the framework, so it is a type of SOA. The idea is to be able to plug different 3rd-party solutions for ontology querying and management into MCT and still have the

same level of functionality provided to the framework. These ideas are exemplified by the organization suggested in Figure 43:

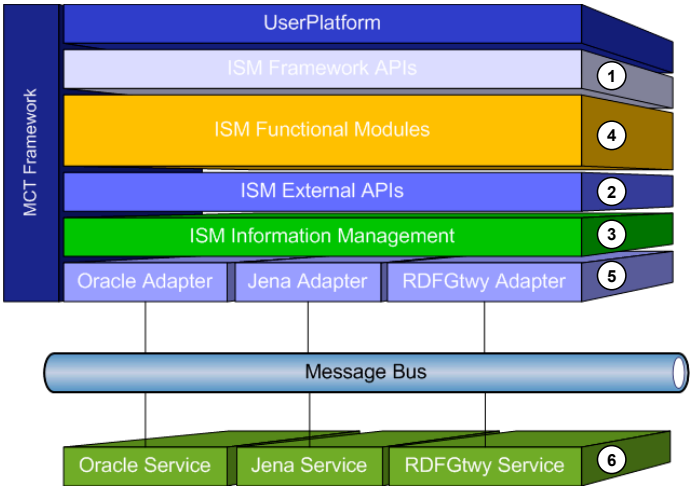


Figure 43: ISM general architecture.

The design approach used to achieve this general set of requirements utilizes interfaces on the MCT framework side (at **1**) to insulate the entire ISM design implementation, and interfaces on the services side (at **2**) to insulate the ISM and MCT framework from the underlying service implementations. Within this design, information management is provided (at **3**), along with implementations of the various information-specific functionalities (at **4**). Different external services are supported through adapters (at **5**) to their actual services (at **6**).

Information Semantics Manager Requirements and Use Cases

The ISM has been associated with the requirements and use cases presented in Table 13:

Required Functionality	Use Cases?	Related Use Cases
ISM1: MCT shall provide an information management system that acts as an intermediary between the framework and an external information store.	No	
ISM2: The information semantics management subsystem shall include a service for determining if a component satisfies a role description	Yes	• ISM check C plays role Role
ISM3: The information semantics manager shall provide a role behavior lookup service that maps behavior names to their ontological specifications.	Yes	• ISM access role Role
ISM4: Direct access to the attributes and behaviors of a component shall be permitted through the use of an abstract content model interface that is application independent	Yes	• SYS access field

ISM5: Information models shall be described using the MCT modeling language of choice	No	
ISM6: The information semantics manager shall include a suite of services to facilitate the integration of external applications, particularly assisting in the integration of their information models.	Yes	• ES change updates ISM model • ENV merges ES and ISM metadata to components
ISM7: The information semantics management subsystem shall support ontology merging based upon configurable policies	Yes	• ISM merge ontology1 and ontology2
ISM8: The information semantics manager shall support the transformation of one ontology into another ontology.	Yes	• ISM update ontology
ISM9: The information semantics manager shall provide a common and accessible declarative knowledge store service.	No	
ISM10: The information semantics manager shall provide declarative system descriptions to other framework subsystems.	No	
ISM11: The information semantics manager shall maintain the component model configuration description.	No	
ISM12: The information semantics manager shall provide conversion support from its native language to a declarative form and back.	Yes	• ISM convert ontology to XML
ISM13: The information semantics manager shall maintain a declarative model of all application components and their relationships to one another.	No	
ISM14: The information semantics manager shall maintain a declarative description of the current system configuration.	No	
ISM15: The core model knowledge stores shall provide a query interface to components that makes it possible to search models' semantic webs.	Yes	• SYS query component model from ISM
ISM16: The information semantics manager shall maintain the active set of role descriptions for use by the system to test component adherence to roles and by components when descriptions are needed to facilitate component interoperability.	No	

Table 13: ISM requirements and use cases.

Information Semantics Manager General Architecture

The Information Semantics Manager (ISM) provides services to the MCT framework while encapsulating its functionality and keeping external services transparently available. The ISM is comprised of four general functional capabilities: (1) knowledge storage and manipulation, (2) distributed knowledge interfacing, (3) MCT component support, and (4) interoperability. The general layering structure associated with these functionalities is shown in Figure 44:

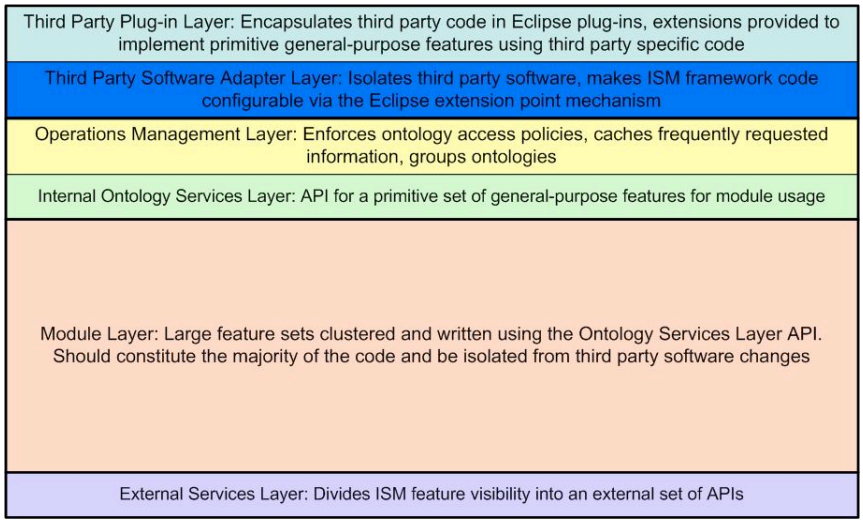


Figure 44: ISM layering structure.

The block structure showing how these functional capabilities (peach layer) are organized by the ISM is shown in Figure 45:

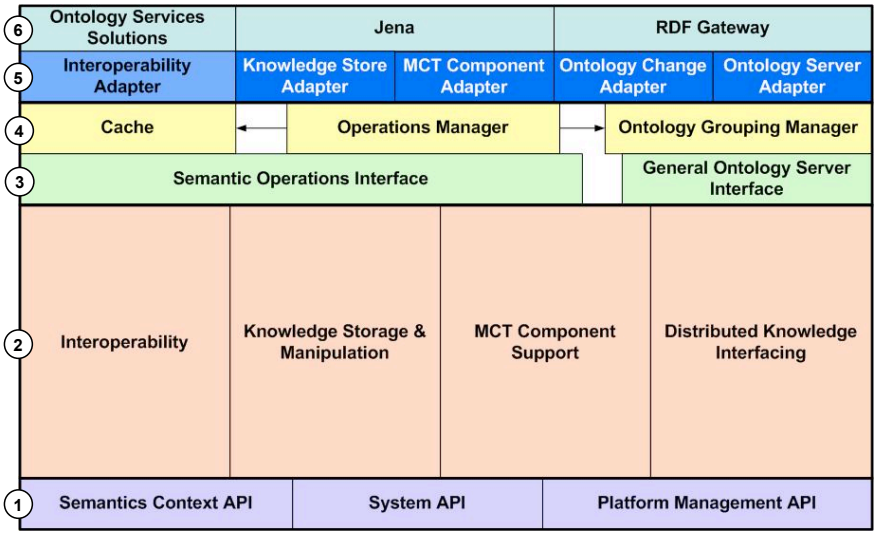


Figure 45: Structure of the information semantics management subsystem.

Six layers are depicted in this diagram. The lavender (or API) layer (at **1**) depicts the public APIs provided by the ISM to the MCT Framework. There are 3 types. The first is the Semantics Context API. This is provided to all of the Framework subsystems so that they can make use of services provided by the ISM. The second is the System API for

interacting with the client host. The third is the Platform Management API which is made available to the User Platform and requires more direct access to ISM services than the other subsystems.

The peach (or Module) layer (at **2**) provides the four larger feature sets associated with the primary functional areas described earlier, clustered and written using the Ontology Services layer API. These four modules constitute the majority of the ISM code base and must be isolated from 3rd party software changes.

- **Interoperability Module:** Responsible for providing vocabulary conflict resolution services to facilitate interoperability between components that use different information models.
- **Knowledge Storage and Manipulation Module:** Manages the local set of domain information models for use by application components.
- **MCT Component Support Module:** Manages the local set of role and component instance descriptions, and offers MCT semantic component services including role matching.
- **Distributed Knowledge Interfacing Module:** Synchronizes the local ontology store with the ontology server through downloading and updating of ontologies.

Each of these modules accesses the underlying functionality through the two interfaces shown in light green Ontology Services layer (at **3**), which provides two APIs (Semantic Operations Interface and General Ontology Server Interface) for a primitive set of general-purpose features for module usage.

These interfaces make use of the underlying ontology services through the yellow Operations Management layer (at **4**): which enforces ontology access policies, caches frequently requested information, and groups ontologies when they are semantically related.

The blue layer (at **5**) is the third party software adapter layer, which isolates third party software and makes ISM framework code configurable via the Eclipse extension point mechanism.

The last layer is the third party plug-in layer (at **6**), which encapsulates third party code in Eclipse plug-ins as extensions provided to implement primitive general-purpose features using third-party specific code. Currently there are 3 services provided: (1) an Ontology Services Solution; Jena, which is an ontology query engine; and RDF Gateway, which is an ontology server. These are implemented (currently) as 5 adapters.

Ontology and Information Management

The outer layers (3-5) of the ISM subsystem figure function similarly to a transaction management system as shown in Figure 46:

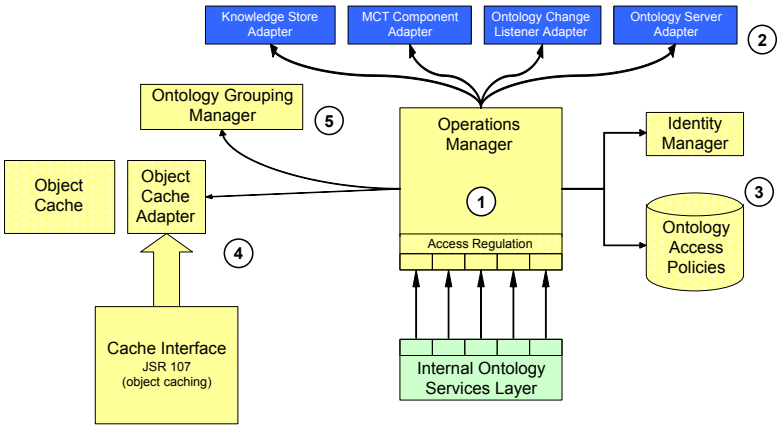


Figure 46: ISM information management.

The operations associated with the different adapters (at **2**) are managed through an Operations Manager (at **1**). The operations manager is accessed through the Ontology Services Layer (green) and protects access to the adapters using access policies that are enforced by the MCT Identity Management subsystem (at **3**). Information retrieved from external services is cached locally (at **4** – this may be refactored). It is possible to group ontologies that are semantically related, and the information management system is responsible for this (at **5**).

ISM Package and Class Structure

The ISM interfaces are shown in Figure 47:

Figure 47: ISM Interfaces.

ISM Deployment

Content to be added.

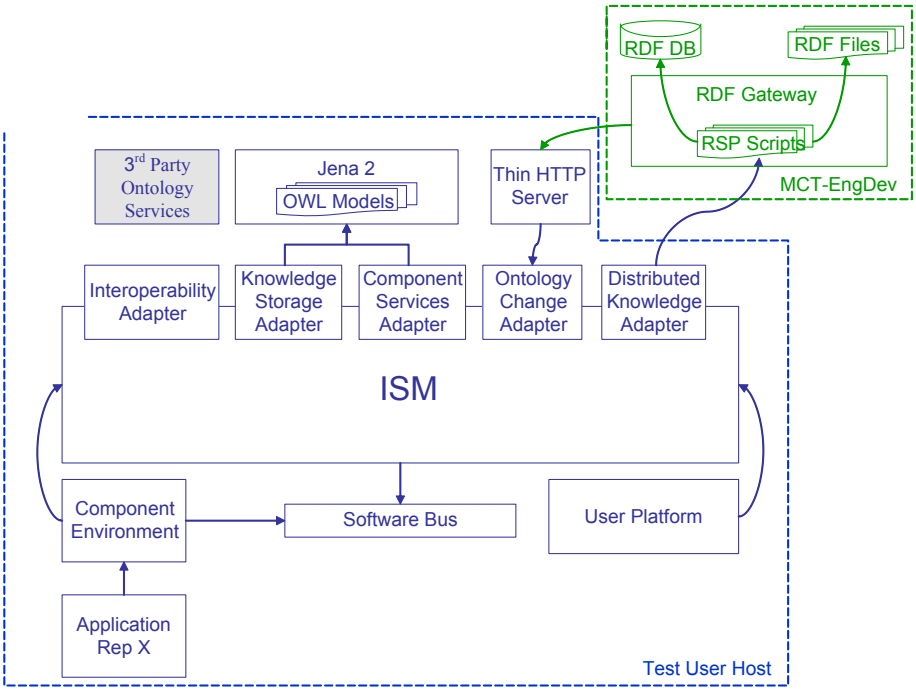


Figure 48: ISM deployment.

ISM Module Decomposition

Content to be added.

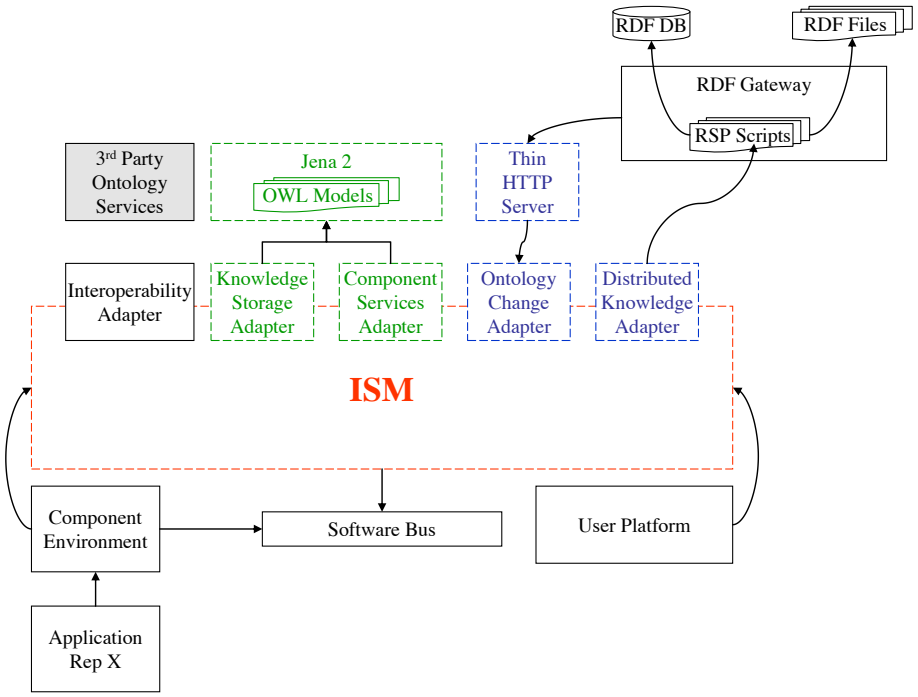


Figure 49: ISM module decomposition.

ISM System Relationships

Content to be added.

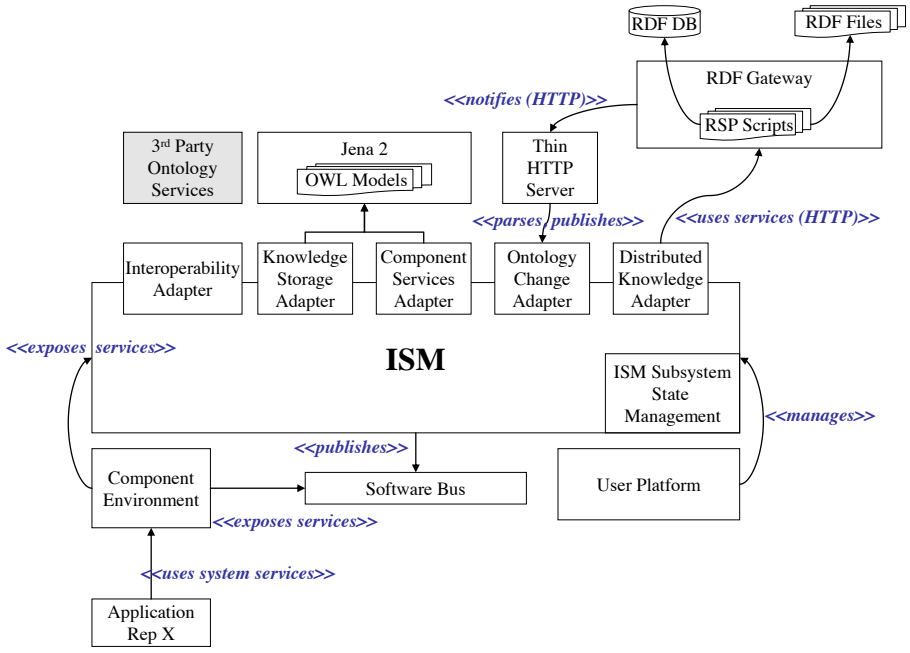


Figure 50: ISM system relationships.

Candidate Ontology Description Languages

Although MCT is currently defining ontologies using RDF and OWL-DL, there is no specific commitment to a particular services language for conveying ontological models or information. There are four languages that are currently vying as standards for defining ontology services:

- **OWL Web Ontology Language for Services (OWL-S):** Original semantics consortium (2002), from CMU and others, based on service profile, service model, and service grounding.
- **Web Service Semantics (WSDL-S):** Initially from the University of Georgia, a WSDL with OWL-S like annotations.
- **Web Service Modeling Ontology (WSMO)**: From Open University (England), a service model based on process language that splits interests from providers and adds mediators.
- **Semantic Web Services Framework (SWSF)**: Academic consortium (Stanford, etc.).

Summary

Content to be added.

Chapter 6 User Platform

The user platform is the MCT framework services and subsystems management hub. It is responsible for all component, service and subsystem lifecycle management: construction, configuration, startup, runtime, and shutdown. It is also responsible for runtime operations, which involves component management, policy management, caching and persistence management, and message management.

Introduction to the User Platform

Every application framework requires an entry point that manages the execution environment. In MCT the UserPlatform is the subsystem responsible for managing the framework and its resources. This includes all lifecycle-specific management of components, subsystems, and services as well as all component-component, component-system, component-subsystem, and component-service interactions.

The UserPlatform provides the subsystems and services the greatest amount of autonomy, because it provides access to each of the subsystems and services to one another through a collection of context facades and delegates. This way the subsystems and services need to know about the UserPlatform, but not about each other.

UserPlatform Design Constraints

The UserPlatform is not a complex subsystem because it provides no functionality of its own. Rather, it is a management hub for all of the services and subsystems in the MCT framework. Nonetheless, its design is informed by 5 constraints:

- Subsystems do not include each others' interfaces
- A single façade is used to provide services and subsystem functionality to all
- The UserPlatform manages all services and subsystem lifecycles
- Multiple startup and shutdown sequence types should be supported
- Multiple environments supporting specialized delegates should be supported

Within the context of these constraints the UserPlatform design is dictated only by the mandatory orderings imposed by the framework startup and shutdown sequences.

User Platform Requirements and Use Cases

The UserPlatform is responsible for a large amount of the functionality in the MCT framework simply because it is responsible for the lifecycle of everything in the framework. The use cases directly related to the user platform are shown in Table 14:

Required Functionality	Use Case?	Related Use Cases
UP1: The lifecycle of an MCT component shall be managed by the User Platform.	No	
UP2: Components shall execute within a managed user platform environment.	No	
UP3: The User Platform shall provide a common framework support layer to support functions that are common to the other layers of the framework.		
UP4: The component execution environment (user platform) shall provide access to the global environment (to all properties including user, session, hardware device, and namespace information, and to services and subsystems managed by the user platform).	Yes	• SYST access UP ENV
UP5: The User Platform shall provide access to services and systems through a one-directional	No	

API (UP -> Foo) through the aforementioned env.		
UP6: An instance of MCT runs in a single VM.	No	
UP7: Components shall be uniquely identifiable though a combination of platform id and unique component id produced using a hierarchical naming convention.	No	
UP8: The system shall dynamically check (at appropriate times) for updates to code and install these updates.	Yes	• USER update MCT • UP update MCT
UP9: The User Platform shall dynamically check, based on policy, for updates to application components and install these updates.		Automated Application Component Updates
UP10: Every component shall have a globally unique ID and semantically-relevant name.	No	
UP11: The User Platform shall provide name resolution mechanisms to discover components from their symbolic names.	Yes	• SYS find object by id • SYS find object by name
UP12: The User Platform subsystem shall aggregate a suite of services provided by different parts of the MCT infrastructure and make these services accessible to all of the components it manages.	Yes	• SYS access service from UP • SYS access subsystem from UP
UP13: The User Platform shall be policy based.	Yes	
UP14: The User Platform will be able to function in both online and offline modes and to re-synch when offline is brought back online.	No	
UP15: The User Platform will provide for a mechanism to interact as a peer with the messaging subsystem.	Yes	• UP register as peer with pub/sub broker • UP unregister as peer with pub/sub broker
UP16: The User Platform will address/resolve issues of concurrency at the component, subsystem and back end levels.	No	
UP17: The User Platform shall support component initialization/reinitialization.	Yes	• UP startup MCT • UP shutdown MCT
UP18: The User Platform shall be configurable.	Yes	• UP configure UP

Table 14: User Platform requirements and use cases.

UserPlatform General Architecture

The internal architectural relationships for the UserPlatform is shown in Figure 51:

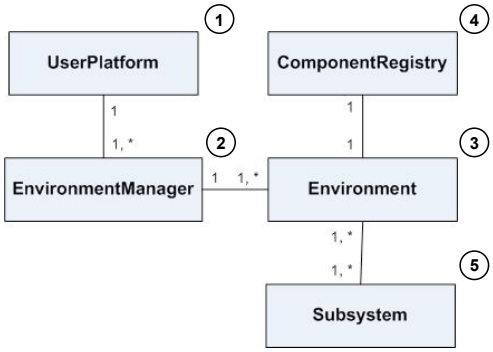


Figure 51: UserPlatform relationships.

Despite the importance of this system, the UserPlatform (at **1**) has a simple structure. It has a single EnvironmentManager (at **2**) that manages Environment instances (at **3**). The Environment manages models through a ComponentRegistry (at **4**), and provides component services and subsystem access to service-related elements in the framework. It achieves this by providing subsystem delegates and operational facades in the environment (at **5**) and the environment is a part of every service-oriented element in the framework.

Looking inside the UserPlatform there are five functional levels: (1) public APIs, (2) framework lifecycle, (3) framework access point, (4) component-based services, and (5) internal component functionality, as shown in Figure 52:

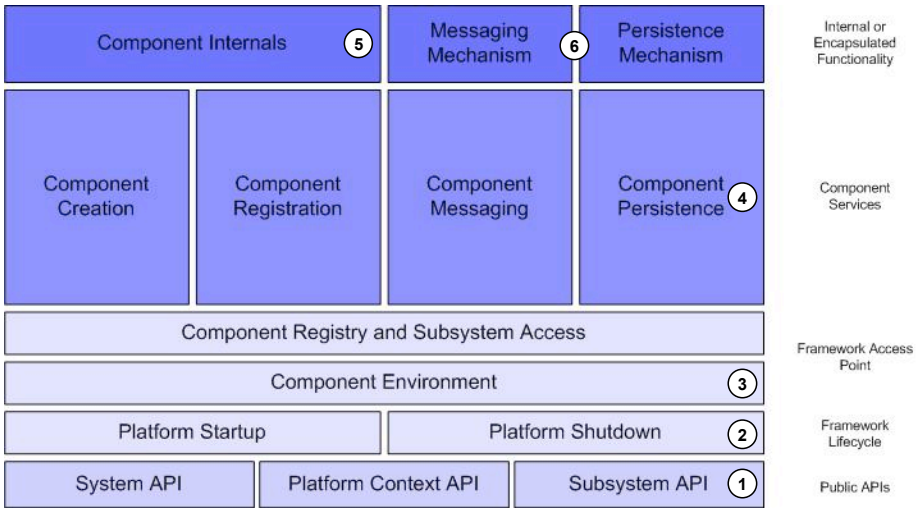


Figure 52: UserPlatform functional levels.

The APIs associated with UserPlatform (at **1**) are divided between system APIs and framework APIs. The framework APIs are the platform context, which provides access to component services, and subsystem APIs which provide access to subsystems. From the UserPlatform instance the startup and shutdown sequences (at **2**) can be initiated. These

control the lifecycle of all component-based services and framework subsystems and modules.

The framework access point (at **3**) is an abstraction layer that provides access to the component registry and related services through a shared environment. There is also an environment manager in case different contexts are required for different subsystems but currently there is only one component environment being used. The component service level (at **4**) represents core services, and consists of component creation, registration, messaging, and persistence.

Below these is the internal implementation of the component model itself (at **5**), as well as implementations of messaging and persistence models (and appropriate adapters, shown at **6**).

The following sections will address the functionality associated with the UserPlatform, starting with the sequences that define the startup and shutdown sequences, followed by a discussion of the component environment, component registry, and the component model itself. Component messaging, data validation, constraint satisfaction, persistence, policy management, and localization are addressed in separate chapters.

Component and Service State

The UserPlatform is responsible for lifecycle management within MCT. As such it is important to identify the states that components and services can have prior to describing the startup and shutdown sequences responsible for moving components and services from state to state.

Component State

Components can take on thirteen states in their complete lifecycle, as shown in Table 15:

State Name	State Description
constructed	A component can be constructed and have no other bindings.
synchronized	A component whose model has been synchronized with the ontological definitions.
instantiated	A component whose instance definition has been loaded.
initialized	A component whose values and data bindings have been loaded.
subscribed	A component that has been added to the pub/sub network.
modified – dirty	A component that has been modified but neither validated nor persisted.
modified – buffered	A component that has been modified and validated.
modified – persisted	A component that has been modified, validated, and persisted.
published	A component that has been modified and published.
unsubscribed	A component that has been removed from the pub/sub network.
unmodified	A component that has been Unmodified.
reinitialized	A component that has been resynchronized or reloaded from persistent storage.
destroyed	A component that is recycled.

Table 15: Component states.

The states in this table are roughly sequenced through the lifecycle. Some states need never be realized. The required states for a runtime component are: constructed, instantiated, initialized, and destroyed. States specific to modification are obviously only applicable to components that can change state. Subscription need not be applicable to a

component if it never joins the component network. Synchronization and reinitialization may never take place and depend on whether the component model changes or the component is used in a context where it hasn't been initialized, respectively.

Service State

Services can be thought of both in terms of component services and subsystem services with slight differences. There are eight states that services can take on, as presented in Table 16:

State Name	State Description
constructed	The service has been instantiated with default values.
configured	The service has been provided with default configuration data.
started	The service has undergone and completed its startup sequence.
published	The service has been made available to the MCT framework.
initialized	The service has had its policies applied to the runtime environment.
unpublished	The service has been removed from the MCT framework.
stopped	The service has undergone and completed its shutdown sequence.
destroyed	The service has been recycled.

Table 16: Service states.

Generally a service's lifecycle includes the constructed, configured, started, stopped, and destroyed states. For MCT services, each is published to the framework so all of these states are applicable to MCT framework services.

UserPlatform Startup Sequence

The startup sequence is responsible for bringing all of the MCT framework services and subsystems online. This involves all creation lifecycle states (initialization). The current (default) system startup sequence is depicted in Figure 53:

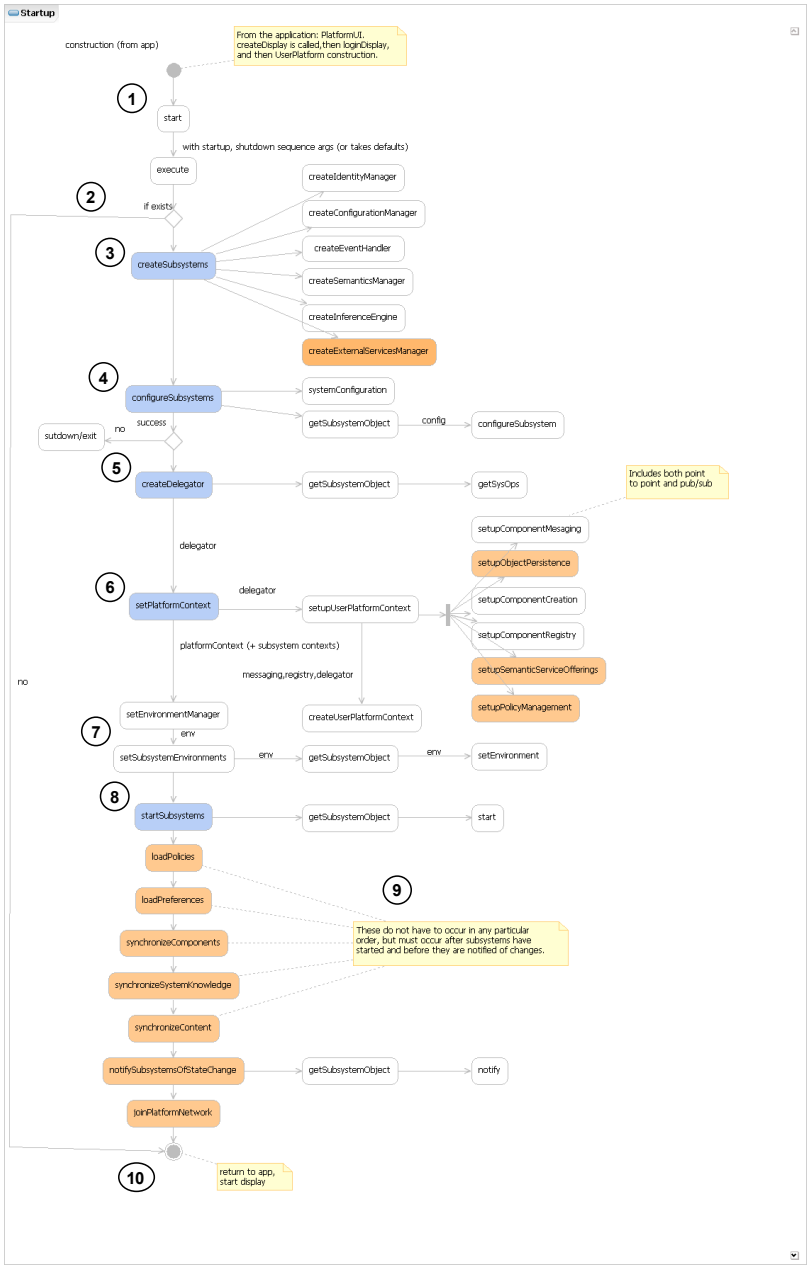


Figure 53: MCT User Platform startup activity diagram.

This figure represents an activity diagram for the UserPlatform startup sequence. The UserPlatform startup is begun when the launching application constructs an instance of the UserPlatform (at **1**). It is possible to create different startup sequences, but the default startup sequence execute() method produces the sequence shown. The first test that is performed is to make sure that UserPlatform construction worked (at **2**). If not the application exits. If it succeeds, the following steps are performed:

- **Create subsystems**
- **Configure subsystems**
- **Create a subsystems delegate**
- **Create platform context (start the platform services)**
- **Create the shared environment**
- **Start subsystems**
- **Load models, policies, preferences, components, and synchronize**
- **Restore the application interface**

Subsystem creation: Subsystems are constructed in a single step (at **3**). Each subsystem is responsible for its own creation, but the UserPlatform initiates it. All subsystems are currently integrated into the UserPlatform though the figure indicates (by orange highlight) that the ExternalServices subsystem isn't currently implemented. The state resulting from system creation is the *constructed* state.

Subsystem configuration: Just after subsystems are created they are configured (at **4**). This is because they must be configured before the subsystem delegate is created or the subsystems are started. Again, each subsystem is responsible for its own configuration. If configuration fails for any subsystem the startup sequence is aborted and the application fails to launch. The state resulting from system configuration is the *configured* state.

Subsystems delegate creation: Each subsystem is responsible for providing a set of methods (sysops) that will provide a façade for the rest of the platform (at **5**). These interfaces are collected into a single delegate at this step.

Create platform context: A general façade must be created (at **6**) that provides access to all services and subsystem functionality. It is called the platform context. To create it the services which form the basis of MCT framework functionality – component creation, registration, persistence, policy handling, messaging, and semantic services are started. These services generally have their functionality defined outside the platform so that they can be used across the framework, but they are managed by the platform. The Semantic Service Offerings service has not been implemented yet and is highlighted. The delegate and the services are used to construct the platform context. After this step the service state is *published*.

Shared environment creation: The platform context and the subsystem contexts together are used to create the shared environment (at **7**). An environment manager allows for the possibility of creating multiple environments but currently there is only one. Once the environment is created it is provided to each of the subsystems so that they can access the published functionality of the other subsystems and the framework services.

Start subsystems: Once the environment has been provided to each of the subsystems they can be started (at **8**). As in creation and configuration, each subsystem is responsible for starting itself. The system state after starting is completed is *started*.

Load policies, preferences, models, components (at **9**): These tasks require that both the ISM and persistence store are online and available (thus the need to follow services and subsystem start steps). At this point policies can be loaded into services and subsystems that will affect the application runtime. Then preferences can be loaded, along with components. After this component synchronization can take place. As with previous steps, it can be seen the most of these steps have not been implemented in the UserPlatform. Those that have been implemented have been done in different ways and need to be migrated. The system state after this step is complete is *initialized*. All component startup states have reached the *subscribed* state by this point.

Restore Application: Once all of the components have been loaded, and the user has been authenticated, the application can be restored. This step involves evaluating user preferences and mission policies (at **10**) along with any steps required to display the application to the user. Steps 9 and 10 are associated with the UI Toolkit among others.

If not clear, the user authentication cannot complete until the framework subsystems have completed their loading.

UserPlatform Shutdown Sequence

The shutdown sequence is responsible for taking all of the MCT framework services and subsystems offline. This involves all destruction lifecycle states. The current (default) system shutdown sequence is depicted in Figure 54:

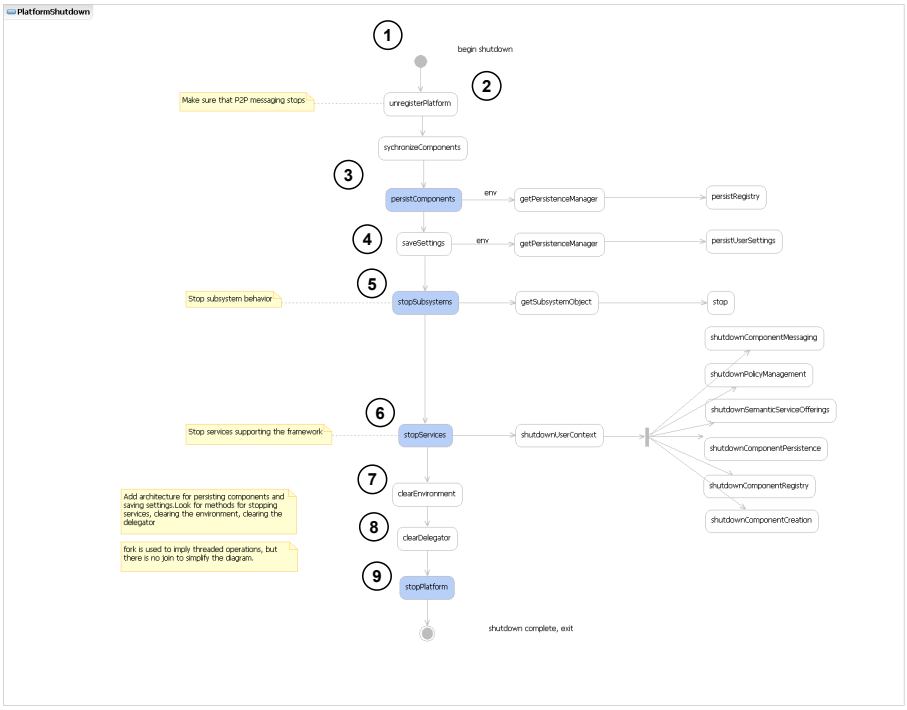


Figure 54: MCT UserPlatform default shutdown as activity diagram.

The UserPlatform shutdown sequence is initiated in a similar way to the startup sequence, through a call to platform.stop(). This will call the execute method on the selected shutdown sequence (at **1**). After that there are 8 main steps in the shutdown sequence:

- **Unregister platform**
- **Synchronize and persist components**
- **Save settings**
- **Stop subsystems and platform context**
- **Stop services**
- **Destroy environment, delegate**
- **Stop platform**

Unregister platform: The most important task is to make sure that the platform can no longer respond to distributed events. As a result it must first be taken out of the client network. After this step the platform is in an *unpublished* state.

Synchronize and persist components: Once the platform is off the network the local components can be synchronized with the ontology and the persistence store, which are still available (at **2**). After this step components have, if necessary, been through both the *synchronized* and *modified – persisted* states.

Save settings: After components are save the remaining settings can be persisted (at **3**).

Stop subsystems: As with the creation of subsystems, each subsystem is responsible for shutting itself down (at **4**). This must be performed before the services can be shutdown because subsystems must stop using the services first. After this step all systems are in the *stopped* state.

Stop framework services: Just as with platform subsystems, services must be shutdown and are responsible for doing so. Of particular note are the component registry, which must be cleared now that all of the components and settings have been persisted, messaging which must clear all queues and unsent messages, and external services which must stop associated servers. After this step all of the services are in the *stopped* or *destroyed* state. Also after this state all components have completed their lifecycle to the *destroyed* state.

Destroy environment and delegate: These are perfunctory steps but must be performed before a new framework can be started.

Stop platform: Once everything else is done the platform can run its own shutdown sequence (as a subsystem). After this step the UserPlatform is in the *stopped* state.

UserPlatform Class Structure

As seen from the discussion of constraints, use cases, and workflows, the UserPlatform manages 11 major functional tasks:

- Component, Service, and Subsystem lifecycle
- Component creation service
- Component registration service
- Component messaging service (Chapter 14)
- Component persistence service (Chapter 15)

- Data validation (Chapter 13)
- Constraint validation (Chapter 11)
- Configuration (Chapter 7)
- Policy management (Chapter 16)
- Updates
- Localization (Chapter 18)

Since many of these functional capabilities are independently designed and implemented (they are somewhat autonomous to the UserPlatform), their design and implementation are provided in other chapters of this document, as stated. The general class architecture of the UserPlatform is itself shown in Figure 55:

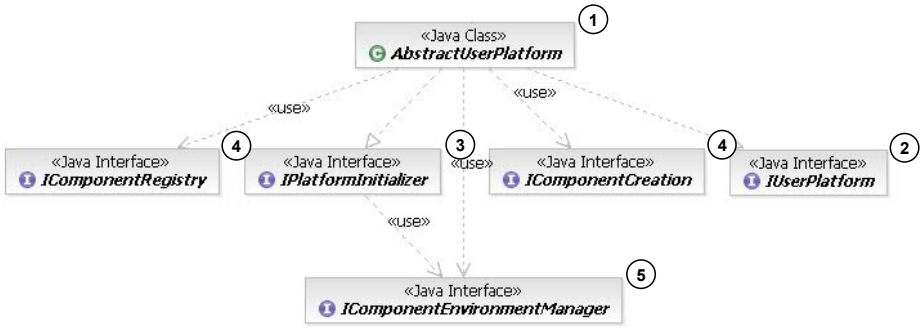


Figure 55: UserPlatform general class structure and interaction.

The AbstractUserPlatform (at **1**) defines the attributes and operations that the UserPlatform instance will override/implement. Specifically it defines the subsystems and their accessor/mutator methods. It implements the IUserPlatform interface (at **2**), which provides access to the Environment. It also implements the IPlatformInitializer interface (at **3**), which defines the accessor/mutator methods for the MCT subsystems the UserPlatform initializes. Since the UserPlatform is responsible for component creation and registration, and environment management, it uses the three interfaces IComponentRegistry, IComponentEnvironmentManager, and IComponentCreation (at **4**).

The relationships of concrete classes in the MCT reference implementation are illustrated in Figure 56:

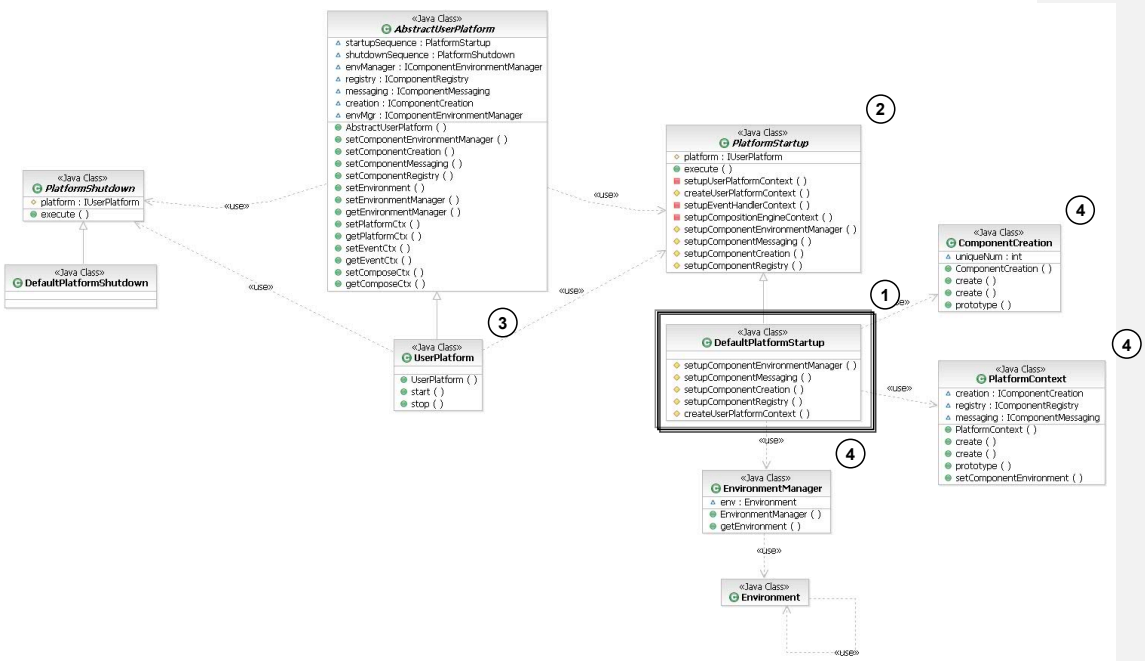


Figure 56: UserPlatform concrete classes.

The central role in the UserPlatform startup is played by DefaultPlatformStartup (at **1**, and highlighted). This class implements the methods that invoke subsystem factories to instantiate (initialize and configure) them. DefaultPlatformStartup extends the abstract class PlatformSetup (at **2**) which defines startup operations. The UserPlatform class (at **3**) defines the operation that calls these methods. DefaultPlatformStartup also makes use of the EnvironmentManager, ComponentCreation, and PlatformContext classes (at **4**).

Functionality Managed by the UserPlatform

Aside from lifecycle management, the UserPlatform is responsible for providing framework access to services and subsystems. This section will present the services the UserPlatform provides as well as how access is provided. There are five services that the UserPlatform provides to subsystems:

- Component creation
- Component registration
- Component messaging
- Component persistence
- Policy management

As can be seen, most of these services are specific to components. Policy management applies to subsystems and services. Each of these services will be discussed below.

Component Creation

The way components are created has changed over the history of MCT. In the beginning they were created procedurally. Later they were implemented in Eclipse plugin.xml files as extensions and parsed into components when the application was launched. More recently they are being created procedurally again. None of these mechanisms follows the original intent, and that is because the original intent was for components to be created from the information models and ontological source and there was no such mechanism in place. A reasonable alternative would have been to represent components using XML and XML Schema and to parse them by the UserPlatform at launch time until the ISM was in place, but that has never been done.

There are two steps in component creation. First, the component model must be acquired because it may have changed since the last application launch. It is from the component model that the basic component architecture can be constructed. Second, given a component architecture, component instances can be read and instantiated. This process has three steps: (a) read the component instance description as a template, (b) register the template in a template registry, and (c) create an instance from the component registry. Component creation refers to this third step.

Component Registration

MCT uses a component model the instances of which are operated upon by sending messages to a particular component instance. When component instance template models are parsed at launch time they are stored into a template registry. At run time component templates are used to instantiate components and they are themselves managed through the registry by unique name or id.

Component Messaging

Component operations are effected through a message-passing mechanism. Every component can have actors assigned to it, where an actor implements the component act method, and the act method calls the component receiveMsg() method. The general messaging approach is also effected through the receiveMsg() method.

Component Persistence

Component persistence is effected during the message-passing mechanism as an access-oriented operation. When a component's model is accessed the persistence manager is notified and the component is updated according to its runtime policies. This topic is addressed in greater detail in Chapter 15.

Policy Management

Policy management is a runtime utility that applies to all component services and subsystems, including the user platform. It should probably be moved from here. It is discussed in greater detail in Chapter 16.

Summary

The UserPlatform is a management subsystem within the MCT platform. It is responsible for making sure that all components, services, and systems flow through their lifecycles in

a predictable manner, for providing access to required functionality, and for generally providing a functional framework for running applications. The following chapters will present the specific functional services and systems provided by the MCT framework and managed by the UserPlatform.

Chapter 7 Configuration Management

An important task of the UserPlatform is subsystem configuration. Configuration involves setting up a subsystem so that it is ready to start. Configuration is best performed using declarative sources so that it can be set up externally to the framework and be modified without rebuilding the framework. Each subsystem is responsible for its own configuration within the guidelines set forth by the ConfigurationManager.

Introduction to Configuration Management

Configuration is the process of making a subsystem ready for startup. It can involve reading variable values and constants, loading files, assigning modules or resources, and assigning subsystem parameters. MCT, having many subsystems, requires a central and uniform configuration management solution.

Configuration management can be viewed three ways. One way is that it is used to select what systems, features, or settings will apply for an application. Another is to load application components. A third is to override initial settings with customized values. In all three cases configuration data is used at launch time and should be kept distinct from the code; to be read and interpreted at when the application is launched. When MCT is initialized a number of systems must be configured along possibly with the application components that are being loaded. As such, a flexible configuration management system is needed.

Constraints on Configuration Manager Design

The ConfigurationManager must satisfy 8 constraints:

- Configuration information is structured using XML Schema
- Configuration information is provided in XML
- Configuration files are validated using the XML Schema
- Configurations can be element and attribute defined
- Configurations can be recursively defined
- Configuration can be used for any subsystem
- Configuration can be used to load components
- Configuration can be used to load user preferences and settings

Configuration Manager Design Considerations

A general question that should be asked is where in the execution workflow should configuration management take place: before constructing java objects or after. If the configuration takes place before java objects are constructed then the configuration file can be written in XML, can potentially be validated against a schema, and can modify the system's document prior to construction. This is a very generic approach. If the configuration takes place after the construction of java objects the configuration file can still be written in XML and potentially be validated against a schema, but now the result of the parse is applied directly to the feature set of the system in question. The tradeoff is that in the former approach the subsystem must be altered to make use of XML during initialization so that there is a document to modify by the configuration management system, but the modifications made to these systems can be generic. For the latter approach to be generic a generic java object must be designed that can work equally well for any subsystem configuration. For these reasons the generic approach is favored for the MCT configuration management system.

Configuration Manager Requirements and Use Cases

Use cases associated with the 10 Configuration Manager requirements are presented in CONFIG8:

Required Functionality	Use Cases?	Related Use Cases
CONFIG1: The central configuration subsystem shall manage configurations for all services and subsystems.	No	
CONFIG2: Each MCT service or subsystem will create its own configuration schema and configuration file.	Yes	• SYST define SYST-Configs-Schema • SYST define SYST-Configs
CONFIG3: Each MCT service or subsystem will be responsible for applying its own configurations.	Yes	• SYST configure SYST
CONFIG4: User objects are configurable.	Yes	• ENV apply configs to components
CONFIG5: The central configuration subsystem shall be invoked by the user platform during platform startup.	No	
CONFIG6: The central configuration subsystem will validate all configuration files.	Yes	• CONFIG validate SYST-Configs using SYST-Configs-Schema
CONFIG7: Configuration schema can be hierarchical but can only include elements and attributes. No other structure is imposed.	No	
CONFIG8: The central configuration subsystem shall be configurable.	Yes	CONFIG config CONFIG

Table 17: Configuration Manager requirements and use cases.

What can be seen from the combination of constraints and use cases is that the configuration manager must be functional, flexible, general, and declarative.

General Configuration Manager Design

The configuration manager is a subsystem the interacts with all services and subsystems through the UserPlatform. To keep it as simple as possible it makes use of declarative models that implement XML schema that are specific to the particular service or subsystem. The configuration manager must be able to read, validate, parse, and manage all configurations. The consideration of its design will thus first address how it interacts with other services and subsystems.

Interaction with Other Services and Subsystems

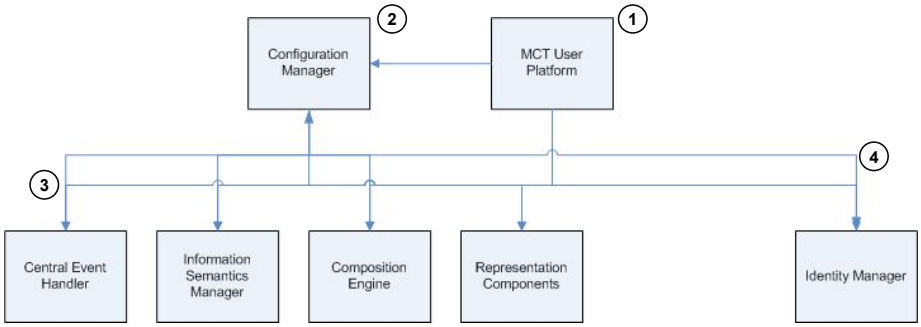


Figure 57: MCT Configuration Manager associations with MCT subsystems.

The User Platform is responsible for creating, starting, configuring, and shutting down MCT subsytems. As a subsystem, the User Platform (shown at **1**) creates, initializes, and configures the Configuration Manager (at **2**). The User Platform also uses the Configuration Manager to read configuration information for other subsystems (shown as the line segment labeled **3**). Each individual subsystem then accesses the configuration information to configure itself (shown as the line segment labeled **4**). The type of information that can be configured, the form it takes, how to create it, and where to place it are topics for the remainder of this document. In the meantime, a closer look at the Configuration Manager will help understand how it works.

An architecture that illustrates this system is illustrated in Figure 58:

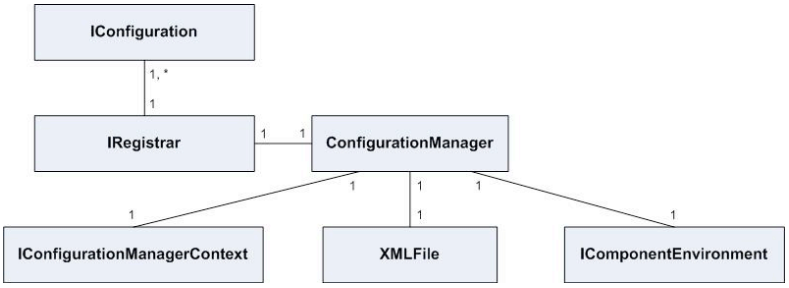


Figure 58: Configuration manager relationship diagram.

The general design of the system is shown in Figure 59:

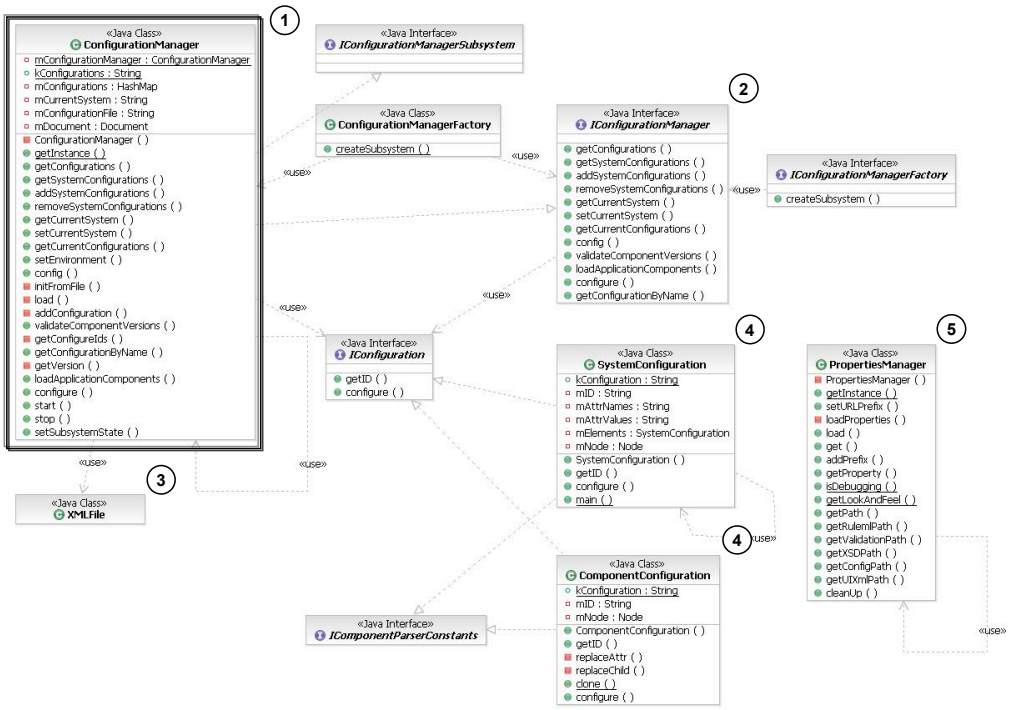


Figure 59: MCT configuration management system.

In this figure, the ConfigurationManager (at **1**) implements an interface (IConfigurationManager, at **2**) that has a number of methods for setting or acquiring an application-specific set of configurations. It also has an init() method that is responsible for loading the configuration file using the utility class XMLFile (at **3**). The configuration file names, along with other runtime properties, are acquired using a PropertiesManager class (at **5**). The IConfigurationManger getConfigurations() method is used to get those configurations associated with a particular application or subsystem. The configure() method dispatches to the appropriate Configuration type (SystemConfiguration or ComponentConfiguration, at **4**) configure() method, which is used to configure subsystems or replace attribute and element information for a particular item with the configuration values, respectively.

The process associated with the configuration is to parse the configuration file into item configurations. These are stored in a local registry. When an item is used by the system, the configuration manager instance is queried to see if that item has a configuration value and, if so, the replacement is made using the associated Configuration found (at **4**).

How Configuration Management Works

The process whereby the configuration manager works is summarized in the diagram shown in Figure 60:

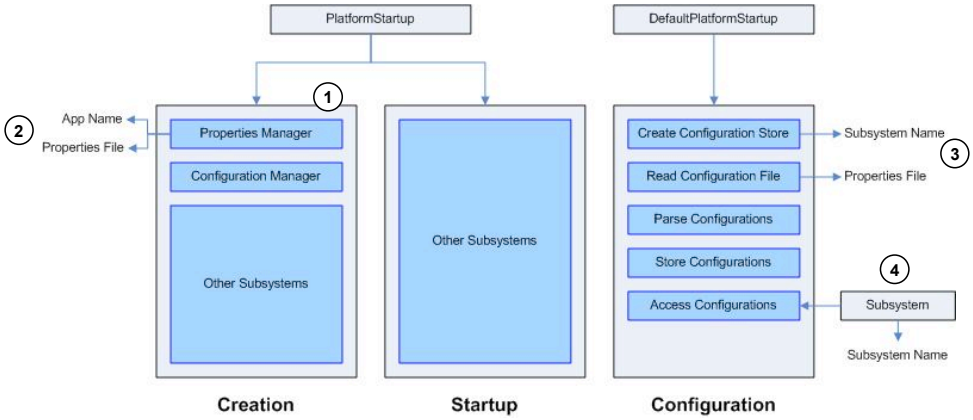


Figure 60: Configuration management workflow.

As can be seen from this figure, there is a component named PropertiesManager (at **1**). This component is used to acquire flat information at platform startup. For example, the path to where the configuration files are located in the deployment directory. The properties can be associated with different applications, so reading the properties file requires an application name (to store the properties) along with a name in the deployment directory for the properties. Currently the view context name is "Module1" and the properties file is called "mct.properties" (at **2**). Initialization of the PropertiesManager is performed in the PlatformStartup createSubsystems() method, and calls initializePropertyMgr(). This method will take the application name and properties file name and load the properties. The PropertiesManager is implemented using a singleton pattern, so it can be accessed from the UserPlatform as required.

PlatformStartup also has a method called systemConfiguration that is used to create the subsystem-specific configuration repository and to read/parse subsystem-specific configurations. This method can be called in PlatformStartup or from DefaultPlatformStartup. If called from the former, then it is called as part of the create[SubsystemName] method. If called from the latter it is called from the configureSubsystems method.

Both initializePropertyMgr and systemConfiguration refer to files using relative paths to the User Platform package, so they make use of a system property to set the URL of relative-path files. With respect to configuration, all configuration files reside in the UserPlatform "config" directory. The names of these files are held in the properties file and referenced through named constants in either DefaultPlatformStartup or PlatformStartup. Currently there are three constants: kEHSystemConfig with a value of path_ehsysconfig_xml, kSMSystemConfig with a value of path_smsysconfig_xml, and kCESystemConfig with a value of path_cesysconfig_xml. As can be seen, these values are assumed to be associated with XML formatted files.

An example of how these files are mapped is shown in Figure 61:

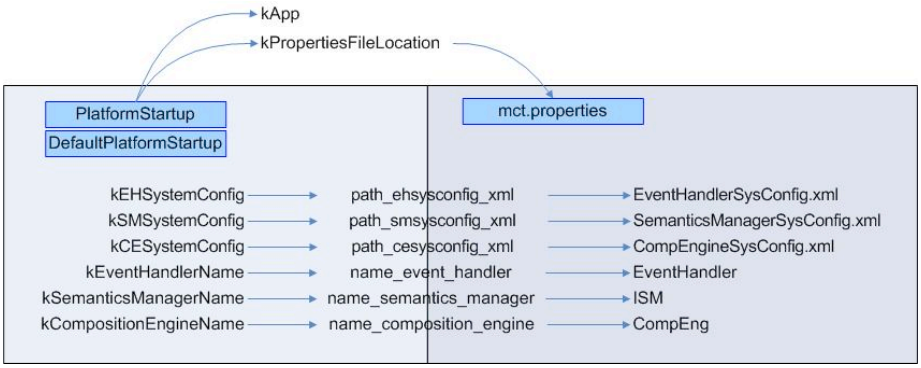


Figure 61: Filename mapping in MCT configuration.

In this figure, the PlatformStartup refers to the kApp and KPropertiesFileLocation named constants to initialize the PropertiesManager. In either PlatformStartup or DefaultPlatformStartup there are six named constants that refer to a subsystem name and its associated configuration file mapping value. In the properties file, the mapping value is associated with the actual filename. This way the properties manager allows the framework to locate files without directly providing filenames.

Configuring MCT Subsystems

The subsystem configuration files contain the content used by the subsystem to configure itself. This information takes one of three forms depending on how the information is to be used:

- System parameter configuration
- Application component loading
- Application component configuration

System Parameter Configuration

The primary intent of the MCT configuration management is to configure subsystems for use in MCT applications. Each subsystem can define its own configuration parameters and can use its own XML schema to define the structure (e.g., of modules or subsubsystems). When parsing these configurations, and because it is desired to have limited cross-system references, the configurations are parsed into a Java structure called a Configuration. This structure is very simple in that it contains a Hashtable of attribute keys and values and it contains a Hashtable of elements that are themselves Configurations and so can contain the same elements. The fact that a Configuration structure exists requires that this structure be available to subsystems.

The contents of a sample system configuration are shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- $Id: SemanticsManagerSysConfig.xml,v 1.1 2006/08/14 22:40:56
jhodges Exp $ -->
<Configurations>
  <Configuration id="ISMModule" mode="SystemComponent">
    <Font name="tahoma" size="12" style="plain"/>
```

```
    <BGColor>
      <Color>
        <IntColor alpha="255" blue="255" green="255" red="255"/>
      </Color>
    </BGColor>
    <FGColor>
      <Color>
        <IntColor alpha="255" blue="255" green="0" red="0"/>
      </Color>
    </FGColor>
  </Configuration>
</Configurations>
```

This file is identified as a system component configuration because the mode configuration attribute is assigned the value of "SystemComponent". The structure associated with the configuration and parse into the Configuration item, is: 2 attributes and 3 elements. The first element has 3 attributes and no elements. The second and third elements have no attributes and 1 element. Each of these elements has 4 attributes and no elements. Since each Configuration item has an id, it can be located and the contents of the Configuration can be parsed by the subsystem.

To access system configuration information, send the Configuration Hashtable to the subsystem during configuration and then locate the item to configure as follows:

```
Hashtable configs ← provided by configuration call
Configuration config = configs.get(idkey);
```

One would then parse this result and assign parameter values accordingly.

Application Component Loading

The User Platform is responsible for controlling which application components are loaded. These components are then registered so that subsystems can access and manipulate them in their proper manner. Since the Configuration Manager is already maintaining configuration information the information for component loading is also maintained by the Configuration Manager. The contents of a sample application component configuration file is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- $Id: SemanticsManagerSysConfig.xml,v 1.1 2006/08/14 22:40:56
jhodges Exp $ -->
<Configurations>
  <Configuration id="JackProxy" mode="ApplicationComponent"/>
</Configurations>
```

In this case, the component being loaded is so identified by the Configuration mode attribute having a value of "Application Component". What happens is that, when the ConfigurationManager loadApplicationComponents() method is called, all items labeled as application components are loaded.

To invoke the application component loading feature you must first access the ConfigurationManager and then invoke the loadApplicationComponents() method:

```
ConfigurationManager configMgr = platform.getConfigurationManager();
configMgr.loadApplicationComponents();
```

Since this operation is only performed by the User Platform it can be called from the platform directly.

Application Component Configuration

In some cases it might be desirable to configure an application component after it has been loaded. In this case it would be easiest, since all application components are in XML/OWL form, to configure right from the XML Document. The Configuration Manager has been designed to maintain the Node information and to swap in the configuration information for the originally-defined information. The contents of a sample application component configuration file is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- $Id: ApplicationComponentConfig.xml,v 1.1 2006/08/14 22:40:56
jhodges Exp $ -->
<Configurations>
  <Configuration id="TrekProxy" mode="ApplicationComponent">
    <Font name="tahoma" size="12" style="plain"/>
    <BGColor>
      <Color>
        <IntColor alpha="255" blue="255" green="255" red="255"/>
      </Color>
    </BGColor>
    <FGColor>
      <Color>
        <IntColor alpha="255" blue="255" green="0" red="0"/>
      </Color>
    </FGColor>
  </Configuration>
</Configurations>
```

As can be seen, this version is almost identical to the application component loading example. The only difference is that in this case there are component modifications identified in the associated Configuration data. When the ConfigurationManager encounters this item it parses the Node information and saves it as Node information into the Hashtable along with the component id and mode. As such, the Configuration has no attributes or elements, but it has an id, a mode, and a node.

To access this information the subsystem would need access to the ConfigurationManager. Then all that is necessary is to call the config() method on any particular Node. The ConfigurationManager will find any configuration associated with the Node's id and replace the content with what is stored in the configuration table:

```
ConfigurationManager configMgr = platform.getConfigurationManager();
configMgr.config(node);
```

This aspect of the configuration process is not yet complete since it is questionable, at present, whether component configuration will be needed. If so then this operation would probably be performed by the Information Semantics Manager (ISM) and then the ISM would need access to the ConfigurationManager.

Chapter 8 Event Handling

The event handling system is intended to capture a wide variety of exceptions and to maintain them for evaluation.

Introduction to Event and Exception Handling Mechanism

The event handling system is used as a central mechanism for capturing, collecting, and analyzing exception events. It must be a centralized mechanism because all of the framework services and subsystems require its functionality. Moreover, it must be the first one started since all of the others might need it during initialization.

- What are the constraints and requirements that inform this framework component's design
- How flexible/autonomous must this framework component be
- What design approaches are feasible, what approach is recommended, and why
- What use cases must be supported by this framework component
- General workflow for this framework component
- Framework component design overview and block diagram
- Appropriate UML to enable development (class diagrams, state diagrams, sequence diagrams, etc.)

Event Handling Requirements and Use Cases

The use cases associated with the Event Handler are provided in Table 18:

Required Functionality	Use Cases?	Related Use Cases
EH1: The central event handling subsystem shall support the logging and auditing of security events.	yes	• EH log security event to file • EH log security event to APPL
EH2: The central event handling subsystem shall have a mechanism for the logging of system failures.	yes	• EH log system failure to file • EH log system failure to APPL
EH3: The central event handling subsystem shall have a mechanism to log application events.	yes	• EH log app event to file • EH log app event to APPL
EH4: The central event handling subsystem shall permit user examination of the system event log.	Yes	• USER view EH event log from file • USER view EH event log from APPL
EH5: The central event handling subsystem shall provide a mechanism to handle events uniformly across the system but to handle them differentially based on event type.	Yes	• EH register handler • Event handler deregisters handler
EH6: The central event handling subsystem will be available to all MCT services and subsystems.	No	
EH7: The central event handling subsystem shall permit the runtime addition of event handlers.	No	
EH8: Events shall be expressed using terms defined in MCT system ontologies and application ontologies.	No	

EH9: The central event handling subsystem shall define and support hierarchical event typing (e.g., severity: event, exception, routine, minor, major, local, and global).	No	
EH10: The central event handling subsystem shall define and support the categorization of events where an event may be in multiple categories (e.g., communication, security, workflow, collaboration, user platform, identity management, information semantics management, persistence, messaging, composition, content management, internationalization, localization, component model, and application).	No	
EH11: The central event handling subsystem shall permit the dynamic addition of event types and categories.	Yes	• EH register event type and category • Event handler deregisters event type and category
EH12: The central event handling subsystem shall support configurable event history sizes.	No	
EH13: The central event handling subsystem shall persist event information by way of the persistence management subsystem.	Yes	• Event handler persists events
EH14: The central event handling subsystem shall include past event retrieval via query.	Yes	• EH retrieve event history by query
EH15: The central event handling subsystem operations shall be policy based (e.g., failure noticing, failure-ignoring).	Yes	• EH handles event by policy
EH16: The central event handling subsystem shall be parameterized with an event handler execution policy. This policy specifies which handlers should service an event, the order of handling, and how/if multiple handlings of single events is performed.	Yes	• EH handles event by policy
EH17: The central event handling subsystem shall support event notification based on configurable attributes (e.g., dialogs, console alarms, email).	No	
EH18: To facilitate the creation of event descriptions, the central event handling subsystem shall permit the dynamic configuration of its event description factory.	Yes	•
EH19: The central event handling subsystem shall provide an event description factory that can be used to generate skeleton event descriptions based on a previously supplied description specification.	No	
EH20: The central event handling subsystem shall log information about components: message, component involvement, associated component roles, and attributes.	Yes	• EH log event
EH21: The central event handling subsystem shall provide a configurable output format that includes timestamp and line numbers where applicable.	Yes	• EH log event

Table 18: Event Handler requirements and use cases.

The event and exception handling mechanism is intended to handle runtime failures in a consistent and flexible manner. There are three aspects to this system, as shown in Figure 62:

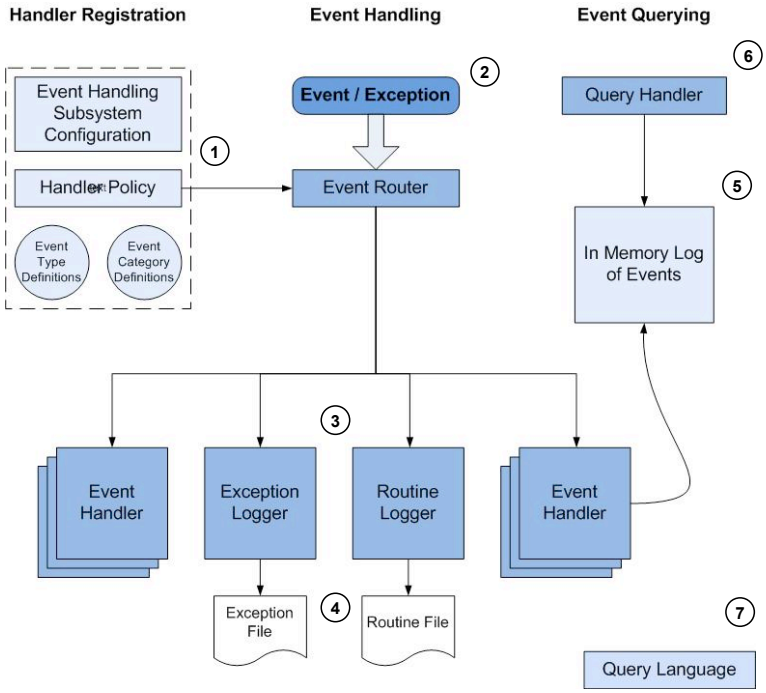


Figure 62: Event/Exception handling system

The central event handler is the subsystem executive. The **registration** phase occurs at launch time when the subsystem is initialized, and configured with event type definitions, category definitions, and handling policies (at **1**). Event **logging** occurs at runtime and controls what events and exceptions (at **2**) are logged, where, and what information is logged (at **3**). These can take the form of in-memory or log-file repositories (at **4**, **5**). Finally, a **querying** component allows the repositories to be queried, either during runtime (of the in-memory log) or after (of the log files), at **6**, according to a query language (at **7**).

A relationship diagram that illustrates the architecture of this system is shown in Figure 63:

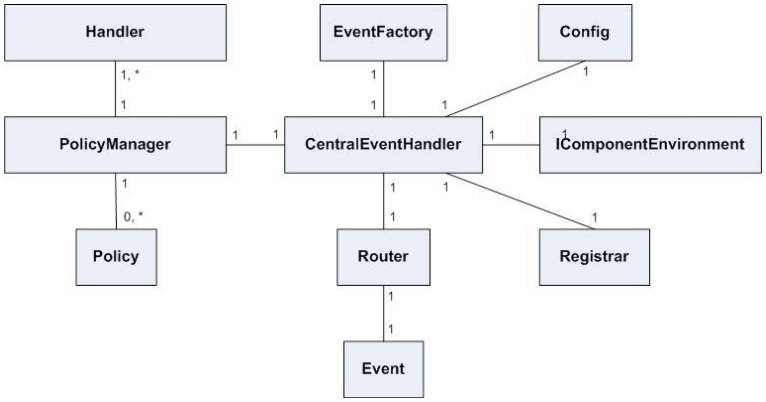


Figure 63: Event handler relationship diagram.

The initialization and configuration of the event handler is performed through the UserPlatform and is described in that section of this document. The workflows for the logging and querying components are shown in Figure 64:

Figure 64: Event and exception handling subsystem logging and querying workflows.

A class diagram depicting the components that comprise the event handling subsystem is shown in Figure 65:

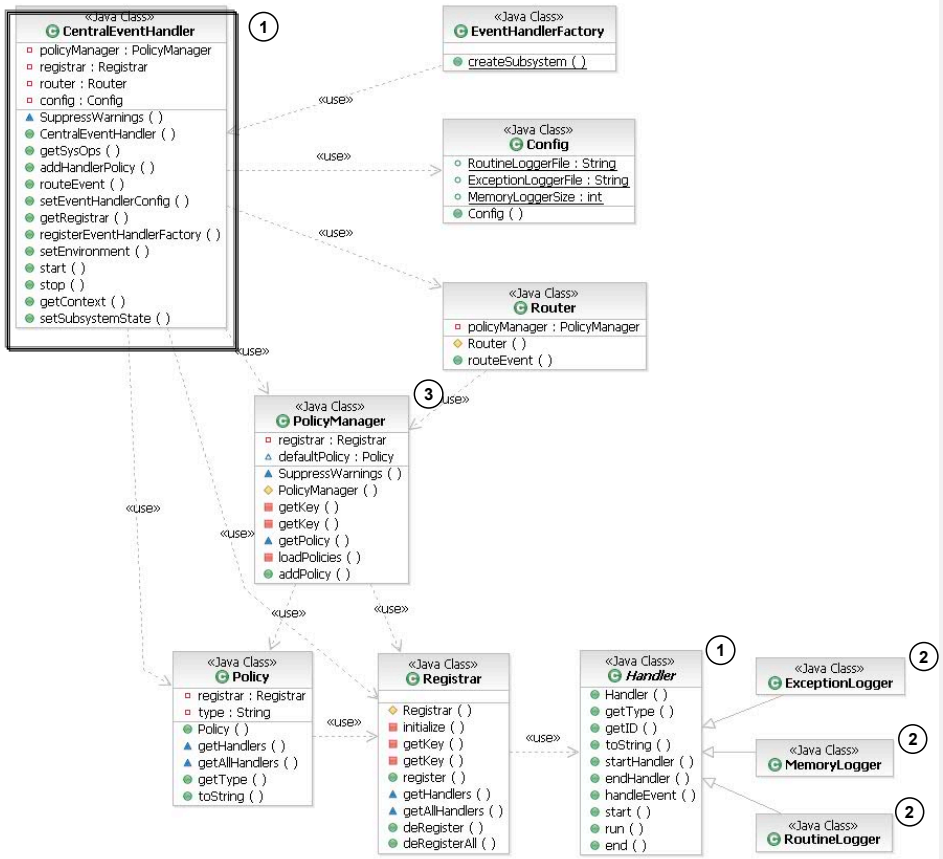


Figure 65: Event/Exception handling class diagram.

Chapter 9 Identity Management

MCT allows users to perform operations on a variety of information sources. MCT controls access to these operations through identity management. MCT leverages existing identity information for operating in its deployment environment, as well as manages its own information about users and security policies for operation within MCT.

Introduction to Authentication and Identity Management Mechanism

Identity management deals with the issues involved in controlling user access to resources, as well as allowing users to operate in an environment with minimal system interference. The goal of the Identity Management subsystem is to provide a robust and extensible security infrastructure that is transparent (for the most part) to the user. Identity management is critical aspect of MCT because mission control applications are potentially used by a broad audience having different capabilities and responsibilities.

The Identity Management Subsystem has four primary functions:

- **Authentication** – In order to gain access to MCT users must verify their identity as a valid user of the system. Verification is determined by the user or system providing information about the user that is known only by the user.
- **User Authorization –** Actions in MCT must have permission to be performed. Authorization can be determined in two ways. First, using traditional access control policies, statically defined properties on an action (read, write, execute) to be used by a certain domain (user, group, admin). Second, by a policy that dynamically compares sets of rules based on properties of the action being performed and the user or system elements performing the action.
- **User Requisitioning/Management** – Identity information needed for operation is managed in the Identity Manager. User identity information is collected and stored in the system to facilitate identity-based operations like authentication and authorization. Access to this information needs to be controlled by the subsystem.
- **Component Access** – A component has access to specific components of the application

Constraints to the Identity Manager Design

fdf

- The External Security Policy and Environment is a factor in design and implementation
- Identity Manager is responsible for all security operations performed in the framework
- Identity Manager is easily extendable and configurable
- MCT has unique identity data that must be managed by the framework
- Performance is a concern when determining levels of security

fdf

Identity Manager Requirements and Use Cases

The use cases associated with the Identity Manager are shown in Table 19:

Required Functionality	Use Cases?	Related Use Cases
ID1: The Identity Management subsystem shall provide Identity Management services within the MCT framework. The external operating	No	

environment shall provide MCT with security and identity information required for operation within external environment.		
ID2: The system shall support user authentication.	No	
ID3: The Identity Management subsystem shall interoperate with the external authentication mechanism such that users login once and gain access to all appropriate resources without further authentication.	Yes	• Single Sign-On Authentication
ID4: The Identity Management subsystem shall ensure that a user has sufficient privileges to access data or executable resources.	Yes	• USER Role invoke operation
ID5: The Identity Management subsystem shall include the ability to grant access to resources based on the current role of a user when the request was made.	Yes	• USER Role invoke operation
ID6: The Identity Management subsystem shall provide the same level of security as the information source.	No	
ID7: The Identity Management subsystem shall provide security that is in compliance with the restrictions imposed by ITAR.	No	
ID8: The Identity Management subsystem shall have a mechanism for defining MCT user security privileges.	No	
ID9: Users will have policy-based and configurable/assignable rights.	Yes	• ID use policy to assign rights to USER
ID10: The Identity Management subsystem shall have a mechanism for managing MCT users.	Yes	• ID manage users
ID11: Each user identity has its own root collection of user objects called a user environment.	Yes	• UP provide access to User Environment
ID12: The Identity Management subsystem will manage user environments.	No	
ID13: The Identity Management subsystem will control access to and content of user environments.	Yes	• UP provide access to User Environment
ID14: The Identity Management subsystem operations shall be policy based.	Yes	• ID operate using policy
ID15: The Identity Management subsystem operations shall support the persistence of users (e.g., user environments, preferences).	Yes	• ID persist user env
ID16: Each user will have at least one user environment.	No	
ID17: The userid will be used to identify the user environment to open upon authentication.	No	
ID18: The Identity Management subsystem shall provide user information to components and other subsystems.	Yes	• ID provides information about USER through ENV to components
ID19: The Identity Management subsystem shall manage security information for external services.	Yes	• ID provides ES with authentication information

Table 19: Identity Manager requirements and use cases.

Identity Manager General Design

The general structure of the identity management subsystem is shown in Figure 66:

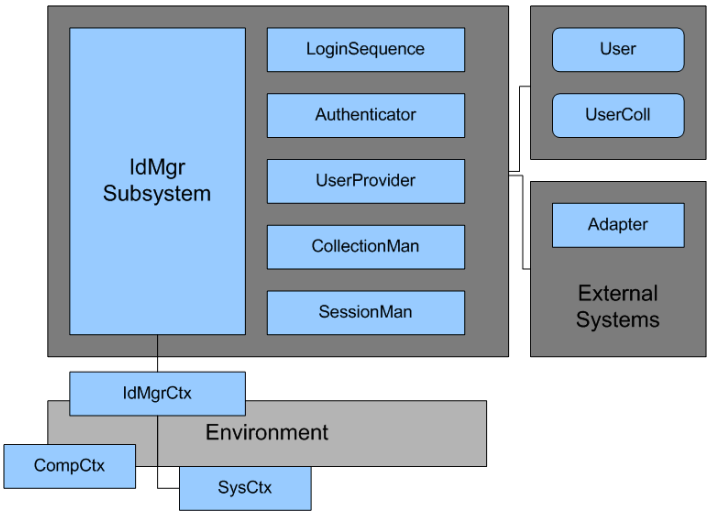


Figure 66: Identity management subsystem.

An architecture diagram illustrating the core relationships in the Identity Manager is shown as

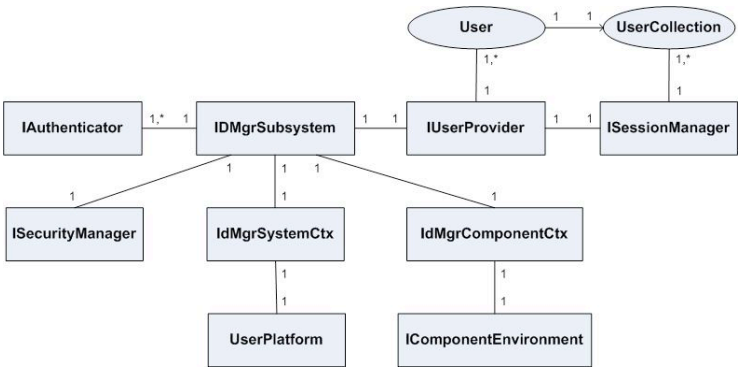


Figure 67:

The workflow associated with how the identity management mechanism functions is shown in Figure 68:

Figure 68: Identity management mechanism workflow.

The general structure of the identity management subsystem is shown in Figure 69:

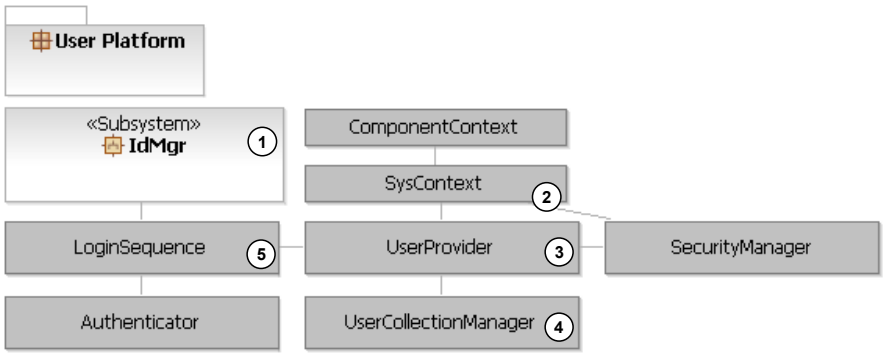


Figure 69: Identity management subsystem.

The diagram shows the general relations between modules in the subsystem. The SecurityManager (at **1**) provides services to the SysContext (at **2**) as well as uses information provided from the UserProvider (at **3**). The UserProvider provides information services to the contexts and the UserCollectionmanager (at **4**) is a delegate of the UserProvider. The LoginSequence (at **5**) manages authentication and initiates operations in the UserProvider.

The Identity Manager subsystem follows the standard API definition and architecture of MCT User Platform subsystems. It conforms to the startup and shutdown sequences and provides system operations APIs for use with the component environment and other subsystems.

Besides the standard subsystem architecture, the Identity manager is divided into 5 distinct operational modules. Each module performs key identity management functions through a well defined API. Leveraging the plug-in architecture provided by the runtime each module has an API whose implementation can be changed at runtime. This means that execution environment implementations can be swapped in and out depending on the identity policies of the executing environment. Modules are also extendable, meaning that the key operations (authentication, user requisitioning) can be extended easily by adding new functionality and configuring the ID Manager to recognize the functionality.

Authenticator

Jack Hodges 8/13/07 7:50 AM
Comment [1]: Can we talk about these configurations are performed?

The authenticator performs all authentication operations for the system. It can be configured to perform different types of authentication. The authenticator manages input between the user and the authentication mechanism. It is also responsible to manage external environment level authentication during start up. The authenticator can be configured to perform a number of authentication requests upon start up of MCT. The authenticator can also be invoked during other system operations if needed. The authenticator collects authentication information for use by other modules in user component creation.

User Provider

The user provider manages access to the User Component. It also facilitates the construction of the User component. The User Provider begins the process of user creation and manages the incoming authentication information as well as interacts with other identity stores to aggregate user information for inclusion in the user component. The user provider then manages subsystem and component access to the user component, allowing appropriate access through interface definitions. The User provider also manages user component derequisitioning which includes persisting the User Component and deallocating identity resources.

User Collection Manager

The user collection manager operates in conjunction with the user provider. It is tasked with managing the life cycle of the User Collection. On login it restores the collection definitions from MCT and makes it available to the user provider and the user component. During normal MCT operations the manager manages access to the collection. Most importantly the User Collection manager manages the persistence of changes to the User Collection. The User can create and delete components in their User Collection and the ID manager is responsible for persisting these changes.

Security Manager

This module handles all authorization operations for MCT. It uses information from the User component to determine whether parts of the system (component or user) have permission to perform actions or access resources. It works in conjunction with the policy manager to evaluate security policies defined by MCT to determine access controls for objects in the system. The security manager also manages session and authorization information and tokens.

System and Component Context

This module consists of interfaces that are to provide external systems with access to common identity manager functionality. These interfaces isolate the identity manager into a set of functions for use by the rest of MCT. Other subsystems do not have direct access to any of the module described above. Instead the subsystems may go through the system operations interface to perform controlled operations. The component Context is similar to the System context except that it is an interface provided to components through the Component Environment. The Component context has far fewer capabilities than the System context, providing basic user information retrieval services and minimal authorization capabilities.

Detailed Identity Management Subsystem Design

Below is a class diagram of the current design of the identity management subsystem. It shows the interfaces for the main modules as well as how they are referenced by the subsystem implementation, see Figure 70:

Figure 70: Identity manager class diagram.

This diagram provides an overview of modules mapped to interfaces. Interfaces perform approximations of module functionality. The actual methods of interfaces are subject to change upon further refinement of Identity manager functionality. Mechanisms for switching and extending implementations at runtime are omitted as they are part of the greater external configuration environment. (OSGI Plug-in mechanism)

The IdMgrSystemContext interacts with modules through IdMgrSubsystem. In some descriptions and sequences this connection is omitted for simplicity. The above is only the barebones module mappings, modules may contain a number of “helper” classes that abstract out functionality specific to implementation. In these cases a number of relations between modules may be present that are not represented in the diagram.

A class diagram depicting the components involved in identity management reference implementation is shown in Figure 71:

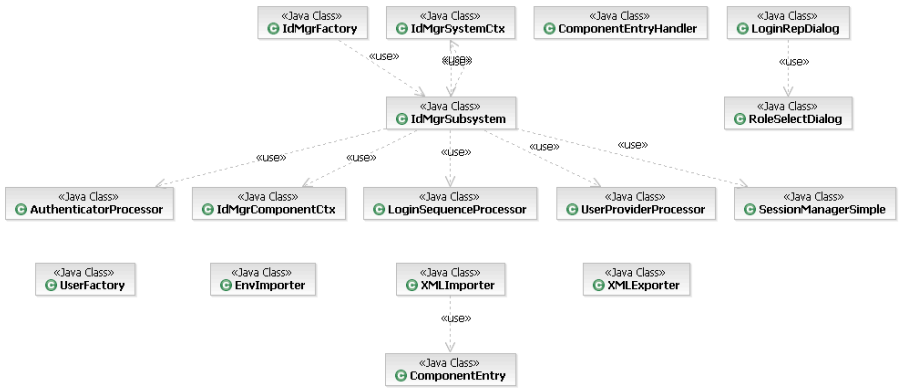


Figure 71: Identify management subsystem class diagram.

The reference implementation differs from figure 45 in that it did not have the Security Manager interface nor did it include the authorization module. The fourth tier of components(and ComponentEntryHandler) is implementation specific as a simple XML based persistence mechanism was used to persist user information. The persistence management subsystem will be leveraged to manage persisted user information.

This diagram also shows the tiered relationship between the subsystem and its modules. The top tier IdMgrSystemCtx is what the external MCT framework interacts with when dealing with the identity manager, the LoginRep is also what a user will interact with during authentication. All other functionality is independent of the framework and should be transparent to the user.

Operation Sequences

The following sequence diagrams provide a detailed look at interactions between modules and the identity management subsystem in performing some basic identity management operations.

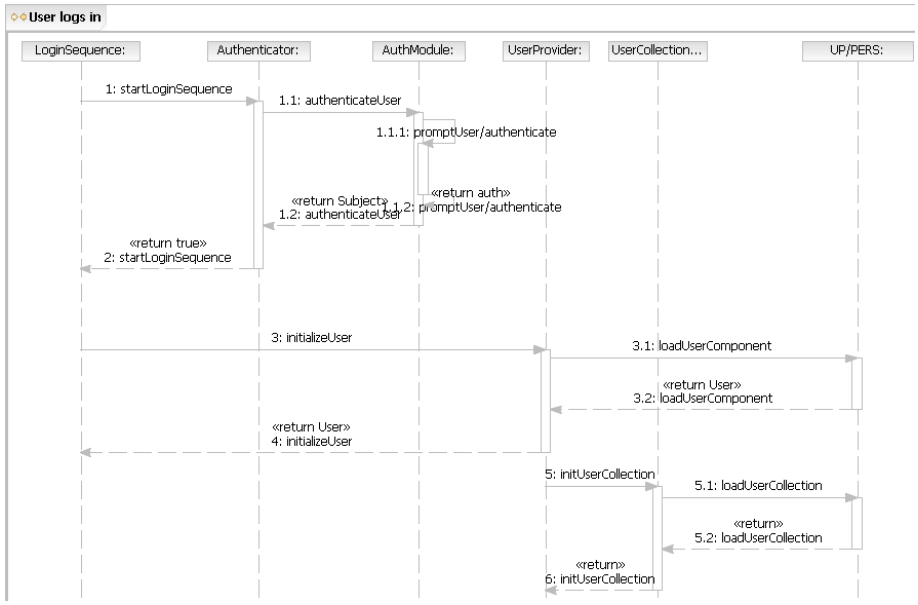


Figure 72: Login sequence sequence diagram.

Dsdd

Authentication

The LoginSequence module provides a sequence of operations to follow to initiate the authentication and configuration of the user component. It is necessary in a capacity similar to the User Platform start up sequence. The LoginSequence triggers the Authenticator module to load its authentication extensions. It authenticates with the external environment or by asking the user for input. A number of extensions can be included to authenticate with different authentication sources. Each extension returns the results of the authentication along with information about the authentication subject from the authentication source.

After authentication is successful the LoginSequence initiates User component requisitioning. The UserProvider then requests the persisted user component from the persistence manager. After the user is loaded from persistence and the recently acquired authentication information is stored in the component the UserProvider delegates User Environment collection requisitioning to the UserCollectionManager. This manager interacts with the persistence manager to requisition the User's root collection. Which is a collection of all components the user can access. This collection's content is based on user role (which the user selected during authentication). The second part of the User Environemnt collection is the User Collection. This is a subcollection of components that have been created by the user.

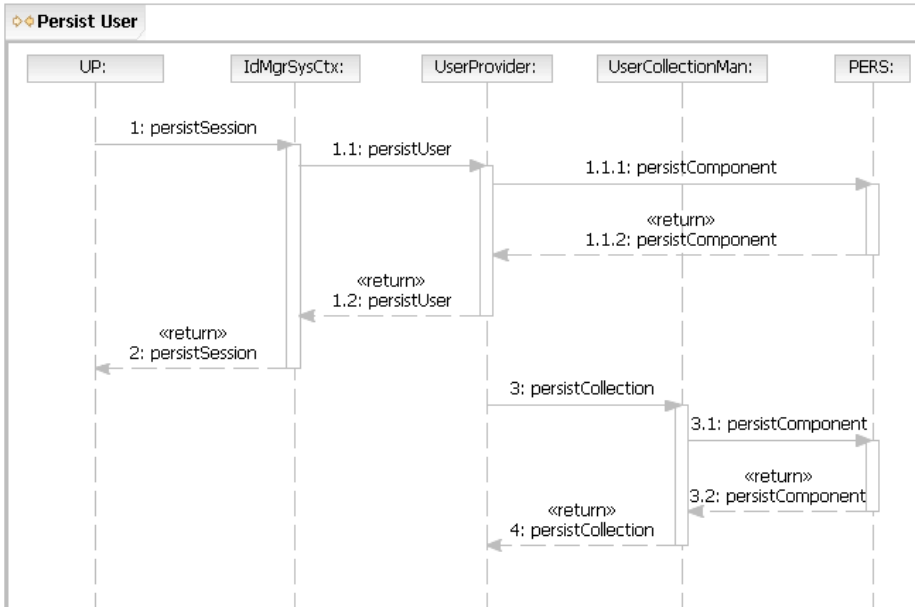


Figure 73: User persistence sequence diagram.

User Management and Persistence

The IdMgrSysCtx provides an interface for persisting the User and User Collections. This interface will be triggered by actions that policy dictates will cause a persist action to occur. These will usually involve changes to the User collection or user component. Users will be able to create and delete new collections within their user collection and each of these changes will trigger a persistence calls.

The context delegates persistence to the UserProvider which persists the user with the persistence manager. The UserCollectionManager then persists the User collection.

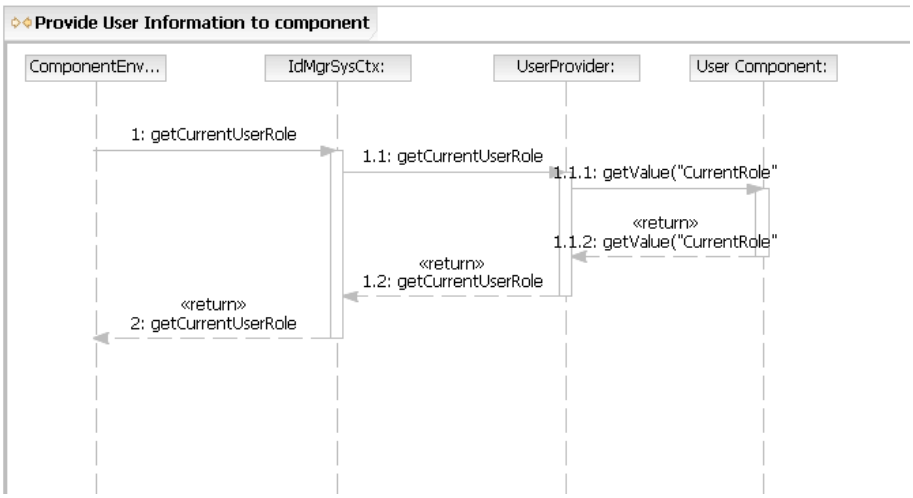


Figure 74: Component user information request

Information Services

This scenario involves a component requesting the current user's role. An example of a use of this operation could be to display the current role being played by the user in the title bar of a window representation. The component does not have permission to access the User Component itself but can use the context to query for basic user information that has been decided by policy to be accessible to components.

The component makes the call to the ComponentEnvironement IdMgrComponentContext which delegates the call to the IdMgrSysCtx uses the interface to the UserProvider to request the user's current role. The UserProvider then has direct access to the UserComponent on the component model .getValue() level and can retrieve the requested information and provide it to the context which then provides it to the component environment for use by the component.

Summary

The Identity Management Subsystem provides authentication, authorization, and user management services to the User Platform. Each of the services is extendable and configurable at runtime. The subsystem is designed to be easily configurable because of the variety of runtime environments it may be required to run in. The Identity Management Subsystem manages all MCT specific data related to the user including a user component as well as user environment collections. Components and other subsystems may request information about the user from the System Context through the User Platform and component environment. Finally, policy-based authorization is performed through this subsystem in conjunction with the MCT Policy Management subsystem.

Chapter 10 Rule Engine

Every application performs a myriad of evaluations and associated decision making. In many cases the logic associated with a workflow can be managed based on context at the time the evaluation is to be made. Complex decisions can make code very difficult to read, and aren't set up well to be reused or modified without significant effort. A rule engine provides a central/generalized processing mechanism for evaluating situations (context) and producing many of the same kinds of decision-making as an embedded approach, but it is much more flexible. The MCT framework has a need for a generalized logic engine that can be configured outside of the framework and used across functionally-disparate subsystems and services. As a result a rule engine subsystem has been implemented.

Introduction to Rule-Based Processing

The use of rule-based processing in any context is associated with three concerns: (1) performance, (2) flexibility, and (3) code maintainability. With respect to performance, rule-based processing can only be effectively used in contexts where massive application of rules is unnecessary because such systems are inherently slower than embedded code. They are appropriate, however, for user interface applications.

Second, with respect to flexibility, separating out business logic from the components which must adhere to it allows for manipulation and maintenance of said logics independently of the components themselves. In cases where a single codebase must support a large number of products this type of approach can be very effective.

Finally, the most specialized and brittle code in any software application is the business logic. When the number of component types supported by the application increases, generally the maintenance staff increases in some linear fashion simply to maintain these logics. Moreover, because they are generally procedurally implemented, and are specific to the object they are related to, any change to these logics will require rebuilding and recertifying the component (and possibly a group of components if the logic is similar across them). A rule engine can be extremely effective in such cases because the logic is extracted from the component implementation, so an increase in the number of component types has no impact on software maintainability. In contexts where maintenance staffing needs to be kept to a minimum this approach might differentiate success from failure. MCT satisfies the first concern and can benefit greatly from the latter two concerns, and so rule-based processing has been embraced in the architecture.

Rule-based processing can be thought of along several dimensions, but the essence of rule-based processing is to uniformly apply a logic to a set of conditions and to determine whether or not an action or event leads to a contradiction in viable state in the system. In the MCT context, rule-based processing is used to test conditions and to vote yes or no.

Surely the application of conditional logic is ubiquitous in programming, but it is too difficult to write conditional logic to handle real-time conditional evaluation because each possible scenario must be taken into account within code that must be updated and managed (and rebuilt and possibly recertified). Where rules can come into play is that different rules can be written to apply to different contexts and they can compete or be aggregated based on how they satisfy the current runtime context – thus allowing for contextual conditional logic application without impacting the underlying codebase. These logics can be sequenced, or chained together, to mirror or simulate the way we induce, deduce, explain, plan, and experiment with the world around us. The mechanism enables us to take the information we have at hand and to explain what happened or problem solve from it.

To illustrate how inferencing works, and just how ubiquitous it is in everyday activities, if we want to go out for lunch with a group of people, we have to decide where we are going, how to get there, who will drive, who will ride with whom, etc. This is a simple planning scenario. Some of these steps are sequential while others aren't. The very fact that anyone can easily come up with an ordered sequence of states for this scenario illustrates both the reality and power of knowledge, experience, and inference for people. When we break this planning scenario into segments, such as the decision segment, the driving segment, the parking segment, etc, there will be variables used in the segments which may be shared across segments. When the plan is created, it must accommodate the variables and the sharing. Since planning is basically a process that takes knowledge and deduces outcomes that can be sequenced, it can be simulated with a forward-chaining, or

deductive, logic because it begins with initial conditions (states describing the world as we know it) and uses the steps in the logic chain to produce new states. These states in turn enable new inferences to be made, and so forth, resulting in a plan that meets the original requirements. If, in our example, one car's worth of people get to the restaurant and we are missing others, we might engage in an abductive (explanation based) form of reasoning to try to induce what might have happened to our friends. This approach can also be implemented using chaining, but uses a backward chaining algorithm to move backward through logical relations to a point of knowledge or knowledge violation. Chaining algorithms are ones where the logic is discretely represented with a form of rule, which is a knowledge-based condition/action pair.

In fact all applications use inferencing all the time, in the form of procedural logics, but these logics are pre-defined and brittle in that any change in the logic requires rebuilding the application and thus makes the application fragile to changes in the logic. Because procedural logics are difficult to write they are also difficult to maintain and this adds additional cost to the maintenance of procedural logics.

Rationale for Inferencing in MCT

For a system like MCT, which is overwhelmingly biased towards declarative representations and information models, a rule engine is a logical choice for performing logic-based operations if it can be made to perform well if applied carefully. The following are nine reasons why/how such an approach could benefit the MCT design.

- **Uniformity:** Other aspects of the system are declaratively-based so it makes sense for the logic processing mechanism to be declaratively based as well.
- **Discrete Organization:** Logic can be organized in discrete locations (and discrete pieces) and moved around easily without adversely affecting the codebase.
- **Processing Centrality:** A single/uniform process model can be used for logic evaluation.
- **Semantic Clustering:** Logics can be ordered or clustered in meaningful ways and later easily found, read, and updated.
- **Interoperability:** The same logics can be applied in disparate contexts without the need to reproduce the logic in multiple locations in the codebase.
- **Autonomy:** Declarative logics can be designed and tested outside the application, supporting scenarios where logic needs to be changed dynamically and without rebuilding the codebase.
- **Metalogic:** Supports scenarios where metalogic needs to be applied to existing logics (e.g., repeat until stable, run in parallel, repeat once).
- **Runtime Logic Choice**: Supports runtime choice of which logic to apply.
- **Complex Logics**: Supports scenarios where there are complex dynamic interactions in logic (i.e., inferencing, reasoning, deduction).

Of course, there are also drawbacks to this kind of approach. Unlike classic procedural logic, declaratively defined rules introduce a new nomenclature and need a mechanism for testing outside the application because it is possible to introduce infinite recursion and looping conditions (e.g., stack overflows). Also, as the data object representation (models) get more complex, the logic becomes more complex and adversely affects performance. Finally, as the number of rules increases performance is degrades, requiring semantic

bundling/switching. These drawbacks assume that the evaluation function is based on equality. If the evaluation functions shifts to partial matching things get even worse. So for our purposes it will be assumed that the evaluation function is based on value equality.

Rule-based processing can be helpful in many scenarios within the MCT framework. For example, it can be used to perform component composition, which is a fundamental feature of the MCT design. It can also be used to perform graphical enablements and disablements for related graphical components. To illustrate, if an item is selected, it may require that another item become enabled, or disabled. This can be enforced by a rule engine. Another way inferencing can be used in MCT is to check data dependencies. If a user makes a change to a value, the rule engine can check to make sure that the value is (after validating the data type and value itself) consistent with other data it shares a dependency relationship with. There are other scenarios in which rule-based processing can be applied, in particular policy management. In the past, it was impossible to apply general policies to running software because there are too many competing events and interpretations and the cost of injecting a rule engine in the process would be very high. It is possible, if carefully applied, that a rule engine could be used to dynamically apply policy to a running system. This is one area where rule-based processing in MCT might run into performance issues, but not in terms of the implementation or application of the rule engine and rules but in terms of the granularity of evaluations that are expected of it.

With respect to possible penalties for using a rule engine, the good/important thing about each of these scenarios is that they are used infrequently and sequentially. Constraint satisfaction could be used heavily in a system where editing is taking place, but that is a very discrete scenario and serial in nature so performance could be kept reasonable. The other examples are infrequently used and should not pose a performance burden on the application but provide great flexibility to the framework.

Rule-Based Processing Approach in MCT

Logic processing in MCT is forward, or deductive, because we always begin from a stable information state and perturb the system in some way. For example, in composition we start from a state where all components are in non-composition-changing states. When a drag event is initiated by the user this perturbation provides the new information required to active/enable a rule chain. From there we simply check to see if what we have done leads to a consistent state or to an inconsistent/violated state. If the state is consistent we do something/nothing as intended. If the state is inconsistent we inform the user, back out any temporary state changes, and continue. Again, in the case of composition, when we drop the dragged component over another component, we are either able to compose the two, in which case we do, or we cannot, in which case we do nothing (or indicate to the user that we cannot).

Constraints to Rule Engine Design

A rule engine is a very simple concept with a wide variety of applicable implementation approaches. The rule engine used in MCT must provide for a variety of inferencing types the extent of which might not be known at present. There are six constraints that inform this design:

- **Pattern Directed State Based Approach**: The implementation approach should be knowledge based, meaning that it will be based on the states of a system and the dependencies that system exhibits between its states. This provides a bounded-world reasoning boundary.

- **General Applicability:** The design must apply equally well to all services and subsystems that might require such a model.
- **Uniform Processing Mechanism:** How the engine works should not change depending on how the engine is used; it should be a uniform approach.
- **Rules Language:** The language used to describe rules should be general enough to describe complex relationships, including function application, on MCT components.
- **Declarative Semantics**: The semantics of rules should be separate from the processing of rules.
- **Policy Support**: The engine should support policies at the level of conflict set ordering and rule execution.

Rule Engine Requirements and Use Cases

The use cases identified with generic rule engine functionality are presented in Table 20:

Required Functionality	Uses Cases?	Related Use Cases
RE1: Rules shall be reusable artifacts.	No	
RE2: CallBacks shall be reusable code artifacts.	No	
RE3: The system shall provide a language to express component-based logic as rules	No	
RE4: The rule engine shall permit the execution of application code in accordance with its active rules.	Yes	• COMP execute function
RE5: The rule language syntax shall support the Integrator friendly expression of logics.	No	
RE6: The rule language shall include the +, -, x, / operations.	Yes	• COMP field value + • COMP field value – • COMP field value x • COMP field value /
RE7: The rule language shall include the fundamental set of comparator operations including <, >, =, >=, <=, !=.	Yes	• COMP field value < • COMP field value > • COMP field value >= • COMP field value <= • COMP field value !=
RE8: The rule language shall permit variable reification such that an abstract variable name can be grounded with a specific value where this value is used in subsequent portions of the expression.	No	
RE9: The rule language shall include the AND, OR, and NOT logical operators.	Yes	• ATOM OR ATOM • ATOM AND ATOM • ATOM NOT ATOM
RE10: The system shall separate rule definitions from the rule engine such that the rule engine can support alternate policy languages	No	
RE11: The rule engine shall include a mechanism to select which rules are executed when multiple policies are satisfied.	Yes	• RE select rule execution
RE12: The rule engine shall permit the parameterization of selection strategies to enforce when multiple policies are satisfied.	Yes	• RE selection parameterization

RE13: The rule policy language and rule engine shall support the selection of rules to execute based upon the generality/specificity of the roles that are matched within a policy expression.	Yes	• RE order rules
RE14: The rule engine shall support the configuration of how many rules to execute.	Yes	• RE select rule execution strategy
RE15: The rule engine and policy language shall support the association of a priority with policy expressions.	Yes	• RE rule ordering strategy
RE16: The rule engine will be policy based.	Yes	• RE is policy based
RE17: The rule engine will support rule sequentiality	No	• Rule Sequencing
RE18: The rule engine will support single pass execution	No	• Single Pass Rule Execution
RE19: The rule engine will support run to stability	No	• Stability Rule Execution
RE20: Rules will support kb dependencies	No	• Knowledge Base Dependencies
RE21: The rule engine will only load/execute semantically-appropriate rules (KB swapping)	No	• KB Specificity Execution

Table 20: Rule Engine requirements and use cases.

These use cases decompose to those relating to rule language expressiveness and flexibility, flexibility of the engine and knowledge bases themselves, and to ease of use. The last of these is difficult to quantify at the subsystem level and will be relegated to an application external to the framework. The first will be addressed in a succeeding section. The flexibility of the engine and knowledge bases are related to how the subsystem works within the framework so these will be addressed first.

Design Overview and Framework Integration

An important requirement of the use of a rule engine in any system is to guarantee the greatest degree of autonomy from the rest of the logic. After all, the intent of using a rule-based system is to remove brittle logic from the system, so to couple the rule engine to the rest of the system only serves to defeat the original intention. Most rule-based systems are integrated into the application they are a part of because the application is an inferencing application. Not so with MCT. So in MCT the rule engine must be designed, implemented, and applied as though it were a library and a set of library APIs. It must perform its functionality in its own environment with a very minimal of interaction with the rest of the framework.

The rule engine subsystem is divided into a layering architecture similar to other MCT framework subsystems, and is shown in Figure 75:

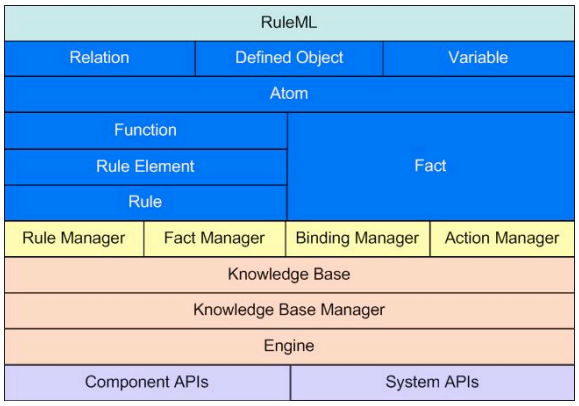


Figure 75: Rule engine layers.

There are essentially four layers in this system. The interaction with the MCT framework occurs at the most general, or API, layer in the system, depicted above in light blue. The core functionality is implemented by the modules that implement these interfaces, namely the rule engine, knowledge base manager, and knowledge base, denoted in blue. The rule engine and knowledge base make use of four managers that keep track of the items needed during inferencing. Most notable are the rules and facts that comprise the core knowledge aspects of the subsystem. Also, during inferencing a binding manager keeps track of the variable bindings within and across rules, and an action manager keeps track of what actions have been taken and is useful for avoiding loops. Finally, the core aspects of the system are the interfaces and classes relating specifically to these items; namely facts, rules, rule elements, functions, and the atoms, relations, variables and independent objects that comprise them, all shown in lavender.

The APIs that are exposed to the UserPlatform and thus to the component environment and other subsystems are:

- **addFact**: This is used by drag and drop listeners to add a fact to an inferencing list.
- **setKB**: This is used to set to active knowledge base.
- **triggerEngine**: This is used to tell the engine to see if inferencing can be triggered by any new or existing facts.

These operations are defined in RuleEngineContext, which implements IRuleEngineContext defined in the gov.nasa.arc.mct.mctcore.contexts.comp package.

The associations the Rule Engine has with the MCT User Platform subsystem is depicted in Figure 76:

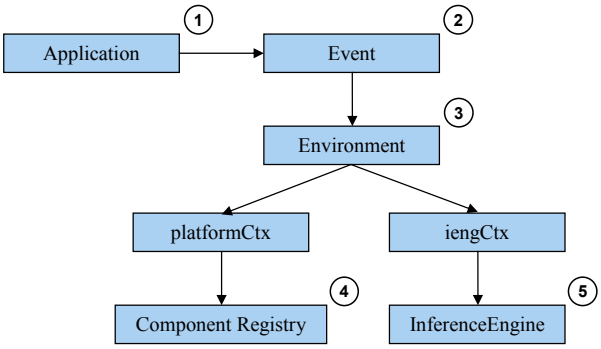


Figure 76: MCT Rule Engine association with MCT User Platform.

When an application drag/drop operation is performed by the user (at **2**) the associated source and target components are found from the component registry by way of the User Platform Environment (at **3**) and platform context (at **4**). The source and target components are then provided to the Rule Engine (at **5**) through its RuleeEngineContext (rengCtx) interface. The User Platform is responsible for starting up and shutting down these services and related subsystems as well as for providing access across services and systems while maintaining service and system autonomy.

Rule Representation and RuleML

One mechanism being developed to address the rule representation language requirements is the Rule Markup Language (or RuleML). This is an XML-based language for representing semantic constraints of arbitrary complexity and intended for use with the Semantic Web. Using a language such as RuleML, inter-object logics can be defined in a constraints/rules file, parsed during application initialization, and then the rules can be cycled through as an event-handling mechanism. Cycling through the rules requires a rule engine (such as Jess or JBoss rules) and a means to translate the XML-based RuleML rules into the Java format required by the rule engine. The RuleML structure will be presented first, followed by the translation into a Java-based rule syntax and then the processing mechanism whereby rules are applied at runtime.

RuleML Structure and Capabilities

The general structure of MCT's modified RuleML schema is shown in Figure 77:

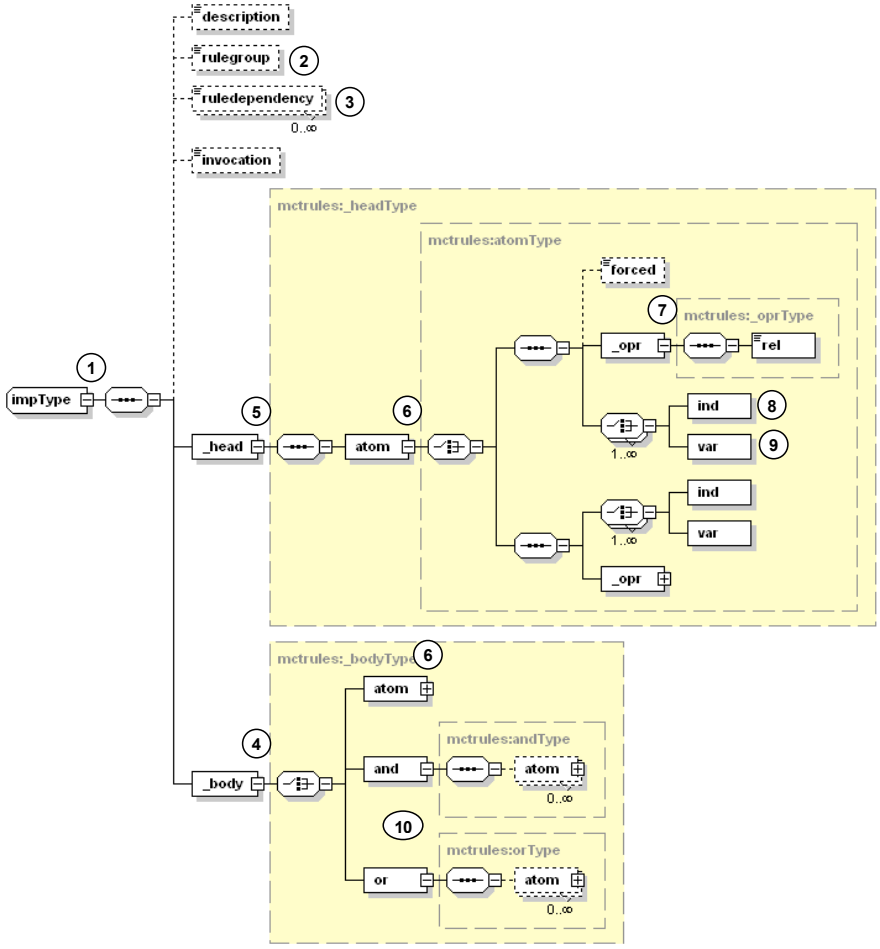


Figure 77: RuleML XML Schema structure for implications.

This is a standard XML Schema diagram. The arcs represent compositional relations, so the item labeled impType (at **1**) is the root of the schema. It is comprised of a sequence (the box with what looks like an ellipsis in it) consisting of a description, a rulegroup, zero or more rule dependencies (represented by "0..oo"), an invocation type, a head and a body. Dotted lines on composition arcs represent that these elements are optional. If they are present, elements must appear in the order specified. The first four items have a small icon in the upper left that signifies that these elements are implemented as strings. The description should be obvious. The rulegroup (at **2**) is as previously described, and identifies the semantic rule grouping that this rule is associated with. Rule dependencies (at 3) identify other rule groups that this rule is associated with.

The content of the rule is described by its head (at **5**) and body (at **4**). Think of the body as the 'if' portion of the rule, and the head as the 'then' part of the rule, where the rule is defined as follows:

```
If [body], then [head]
```

The rule engine checks known states against the if part (often called an antecedent) of a rule. If it matches the rule engine can produce the then part (often called the consequence) of a rule. As can be seen in the figure, the body (at **4**) can itself be decomposed into a choice of (denoted by a funny switch-looking icon) an 'atom' element, an 'and' element, or an 'or' element (at **10**). That is, the body is allowed to hold complex logics.

Normally the head is also allowed to contain complex antecedent logics. In the current case, the head is only allowed to hold a single atom (at **6**), which itself is comprised of a choice between two expression formats (prefix and postfix). The prefix notation (shown at **7**-**9**) is comprised of an operator (written as a string-represented relation, at **7**) and a sequence of operands that can either be independent arguments (defined components or literals, at **8**) or variables (at **9**). The reason that head is only allowed a single atom is that it is a very difficult task to explain why a rule violation occurred if there are many consequences of a rule. It is easy to explain if there is only one. So it is better to have many rules with single consequences (from an operational, user-centered point of view) than to have a few rules with many consequences. The exception to this 'rule' is graphic rules, which this model accommodates through the use of functions.

Relations (as shown at **7**) can be any defined function. Currently there are two types of functions defined in this system: (1) validation functions, and (2) callback functions. Validation functions return Boolean values. Callback functions return no values. It is thoroughly possible to create functions that produce new facts for further inferencing, but they are not applicable to the composition mechanism

RuleML Representation

An example showing how RuleML is used to represent constraints, for a single rule, is illustrated in Figure 78:

```
<?xml version="1.0" encoding="utf-8"?>
<rulebase direction="bidirectional"
          xmlns=http://gov.nasa.arc.mct.issruleelements                    (1)
          xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
          xsi:schemaLocation="schemas\RuleML.xsd">
  <!-- Titan Monitoring Composition Rules -->
  <imp>
    <rulegroup>TitanComposition</rulegroup>                                (2)
    <invocation>Chaining</invocation>                                      (3)
    <_head>
      <atom>
        <_opr>
          <rel>compose</rel>                                   (6)
    (4) </_opr>
        <var name="?source1"/>                                 (7)
        <var name="?target1"/>
      </atom>
    </_head>
    <_body>
      <and>
        <atom>
          <_opr>
            <rel>validateRolePredicate</rel>
          </_opr>
          <var name="?source1" type="Elements"/>
          <ind name="ComposableRole"/>                                     (8)
    (5) </atom>
        <atom>
          <_opr>
            <rel>validateRolePredicate</rel>
          </_opr>
          <var name="?target1" type="!Compose"/>
          <ind name="InspectorRole"/>
        </atom>
      </and>
    </_body>
  </imp>
</rulebase>
```

Figure 78: An MCT rule represented with MCT RuleML.

The rulebase is comprised of any number of implications or facts wrapped inside a banner (at **1**) that describes the locations of namespaces used in the file, including the RuleML schema itself. Several features are worth noting in the single rule provided. First, each rule has a semantic group (at **2**). The semantic group identifies which knowledge base this rule will be loaded with. Each rule can also have an invocation declaration (at **3**). In this case the type is 'chaining' which is the default case for normal inferencing. Next, the rule is divided into its head (at **4**) and body (at **5**) constituents. The head represents what to do when the rule is satisfied and the body represents the conditions under which the rule will be satisfied. Each head or body can contain atoms, which represents a relationship between two or more arguments. The first atom example is in the head portion, where the relation (at **6**) is 'compose' and the arguments (at **7**) are both variables. The second atom example is in the body portion, where the relation is 'validateRolePredicate' and the arguments are a variable and a constant value (at **8**) 'ComposableRole'.

RuleML Parsing to Java Rules

RuleML forms cannot be processed directly, so they must first be parsed into Java rules. The Java-based rule engine can interpret them. Since the rules do not change with time,

and can be loaded and unloaded at launch time, or runtime, and since MCT doesn't have a large number of rules to begin with, this extra processing shouldn't force a noticeable performance loss. The MCT rule engine parses directly from RuleML into its knowledge base of rules. Other engines, such as Jess and JBoss Rules, must be translated from RuleML into their own native formats.

Parsing is initiated from the KnowledgeBase class but makes primary use of the Importer class found in the rules package. The Importer class uses DOM to parse the rules using the classes found in the ruleml package. When parsing is complete all rules have been parsed and exist as facts or implications in the current knowledge base.

Rule-Based Processing

Once rules have been parsed from RuleML into the rule engine's knowledge base, the rules can be processed. The MCT rules engine utilizes a forward-chaining (FC) algorithm to perform its analysis because it is a deductive model. That is, the engine is trying to start from some state change and see whether the conclusions violate any other rules or states.

Forward Chaining

In the FC approach, the algorithm starts from a set of known facts, and matches those to the condition/if parts of available rules. Those rules that satisfy the current knowledge/fact base are collected, sorted, and executed in order, leading to predictions. The general mechanism is shown in Figure 79:

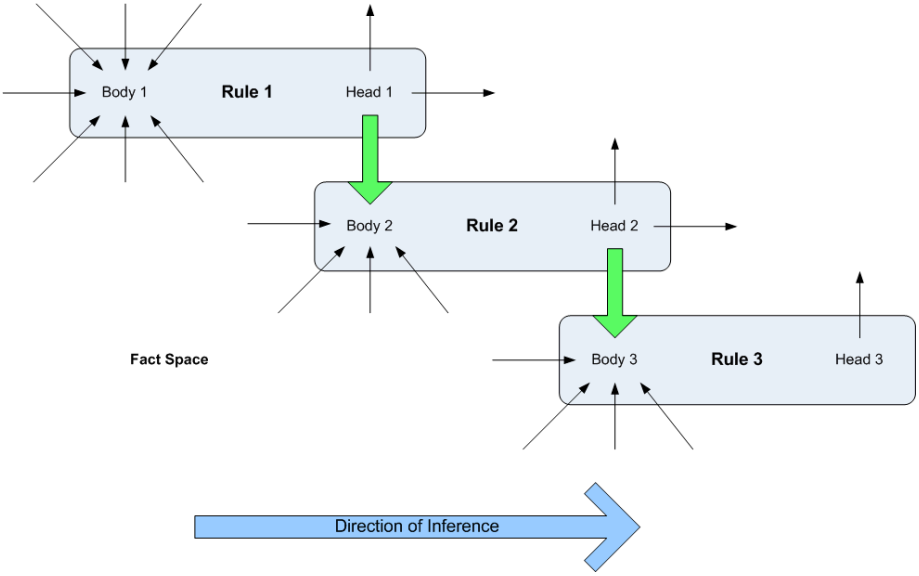


Figure 79: Forward chaining approach.

Consider the three rules (Rule1…Rule3) in the above figure as representing all of the rules available for a particular semantic group and knowledge base. Each rule is comprised of a body portion and a head portion. The body represents the conditions required to activate,

while the head portion represents the new actions to be taken. Considering Rule 1, the arrows pointing toward Body 1 represent comparisons to facts in the fact space (i.e., all of the facts available to the rule engine). Likewise, the arrows pointing away from Head 1 represent new facts produced when Rule 1 is fired. Considering Rule 2, if the facts created by Rule 1, along with any other facts that exist in the fact space, enable it, then Rule 1 is said to have enabled Rule 2. Since the postcondition of one rule helps to satisfy the preconditions of another rule, the first rule is said to have implied the second rule. The processing direction, from Rule 1 toward Rule 3, is forward, or deductive, and that is why it is called a forward chaining algorithm.

Rule Engine Reference Implementation

An architecture that can implement the requisite rule engine functionality is illustred in Figure 80:

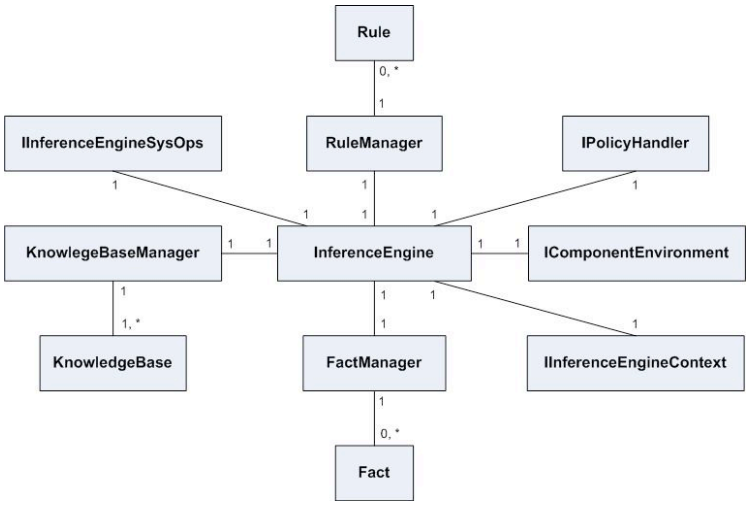


Figure 80: Rule engine relationship diagram.

The top-level design of the Rule Engine system is comprised of the RuleEngine, KnowledgeBaseManager, KnowledgeBase, RuleManager, and FactManager, as shown in Figure 81:

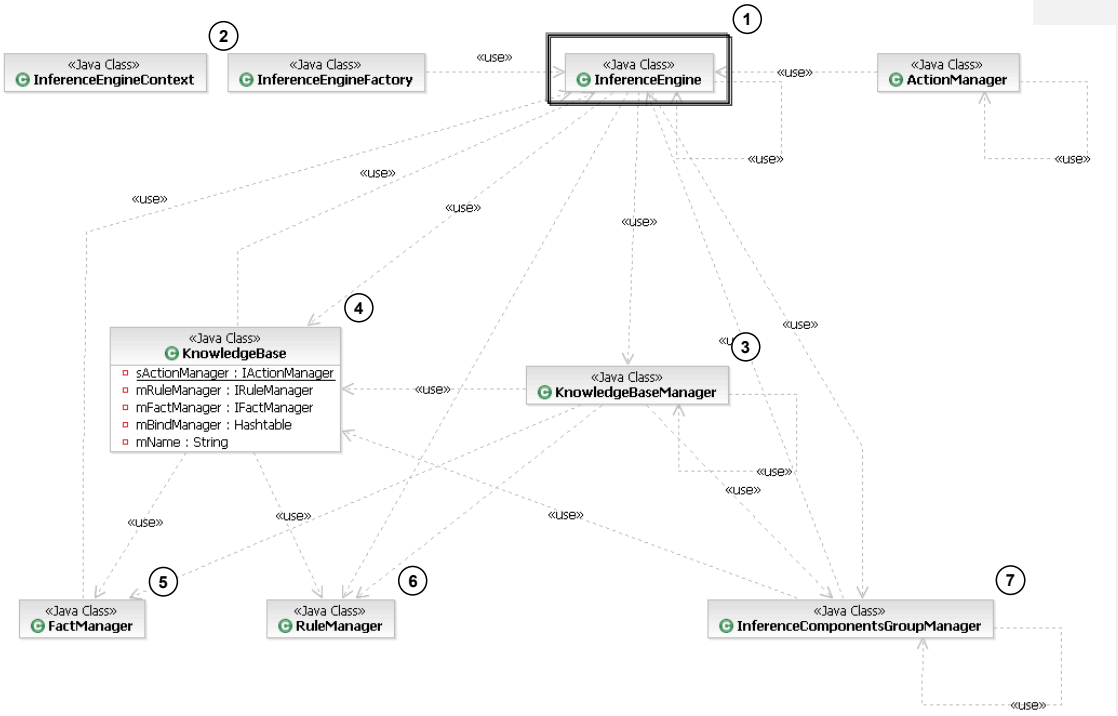


Figure 81: MCT rule engine subsystem.

In this figure, the RuleEngine (at **1**) implements an interface (IRuleEngine, not shown) that is responsible, as an MCT subsystem implementation, for startup and shutdown functions, as well as configuration management. The subsystem has a context (at **2**) which exposes functionality at the platform level to other services and subsystems. This functionality is a façade to the functionality underlying the subsystem but is intended to provide general-use functionality of the subsystem. The rule engine can make use of any number of knowledge bases and so there is a KnowledgeBaseManager (at **3**) that is used to access a particular KnowledgeBase (at **4**). KnowledgeBases are comprised of (among other things) a FactManager (at **5**) and a RuleManager (at **6**). At any given time the RuleEngine will be associated with a specific (or current) knowledge base, and hence the facts, rules (etc) that are associated with that knowledge base. Each knowledge base keeps track of those components that are associated with it through the InferenceComponentsGroupManager (at **7**). The purpose of multiple knowledge bases is to guarantee semantic homogeneity between rules loaded for inferencing at any given time. This reduces the number of rules that are loaded and helps improve performance. This capability comes at a cost, and that is that rules must be associated with a semantic group. If a component is associated with multiple semantic groups the system must be capable of joining the knowledge bases at run time. The overall approach is quite flexible.

RuleEngine

KnowledgeBaseManager

KnowledgeBase

ActionManager

FactManager

RuleManager

Rule Engine Packages

There are five packages associated with the rule engine that have not been discussed:

- **Rules**: Organizes the classes that define the types of rules that can be constructed in the subsystem.
- **Element**: Organizes the classes that define the types of rule elements that can be constructed in the subsystem.
- **Function**: Organizes the classes that define the types of functions that can be used in the subsystem.
- **Fact**: Organizes the classes that define the types of facts that can be constructed in the subsystem.
- **RuleML**: Organizes the classes that implement the RuleML language.

The Rules Package

The rules package is used to define types of rules that can be constructed and processed using the rule engine. It is also used to:

- **Import**: Parse rules from RuleML
- **Export**: Write rules back to XML
- **Construct**: Uses a factory and interface to construct the correct rule type

The package currently supports an abstract rule called a BinaryRule. A binary rule is a rule that is either satisfied or not satisfied; there is no partial satisfaction. The MCT binary rule manages its current state and firing history. There are four classes that extend BinaryRule:

- **InclusiveRule**: If two propositions exist saying that A implies B and a implies b, then a = A implies that b = B.
- **ExclusiveRule**: If two propositions exist saying that A implies B and a implies b, then a = A implies that b != B.
- **MutualInclusiveValuesRule**: two propositions exist saying that A implies B and a implies b, then a = A implies that b = B, and b = B implies that a = A.
- **MutualExclusiveValuesRule**: two propositions exist saying that A implies B and a implies b, then a = A implies that b != B, and b = B implies that a != A

The rules package is shown in Figure 82:

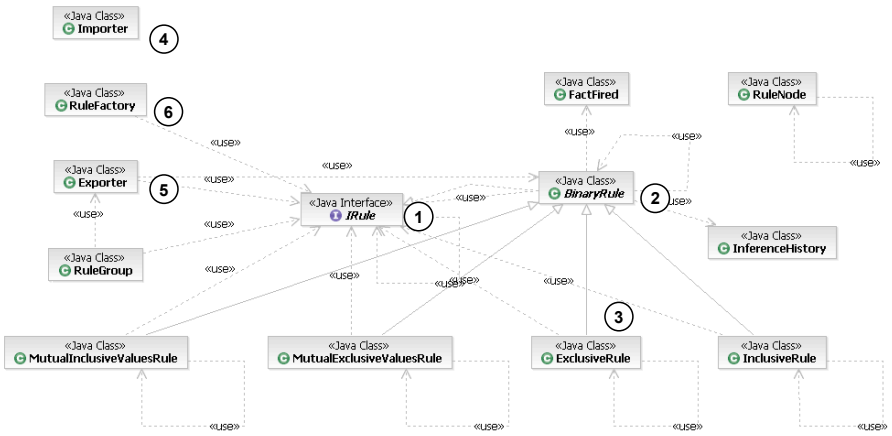


Figure 82: The rules package.

The IRule interface (at **1**) defines the baseline functionality for the package, while BinaryRule (at **2**) is the abstract class that defines shared functionality for the package. InclusiveRule, ExclusiveRule, MutualInclusiveValueRule, and MutualExclusiveValueRule implement BinaryRule (at **3**). The package is also responsible for initially acquiring rules using an Importer (at **4**), and for exporting rules with an Exporter (at **5**).

The Rule Element Package

Rules are comprised of rule elements. The element package defines the type of rule elements that can be part of MCT rules, along with a factory for creating them. There is an abstract RuleElement that provides the basic rule element functionality, and five classes that extend RuleElement:

- **ComplexRuleElement**: A rule element that has more than one element. This extends RuleElement.
- **ComplexAndRuleElement**: A rule element where the elements are conjunctive. This extends ComplexRuleElement.
- **ComplexOrRuleElement**: A rule element where the elements are disjunctive. This extends ComplexRuleElement.
- **ValidateRuleElement**: A rule element that answers a Boolean question and extends RuleElement.
- **CallBackRuleElement**: A rule element that performs an operation on a class and extends RuleElement.

The rule element package is shown in Figure 83:

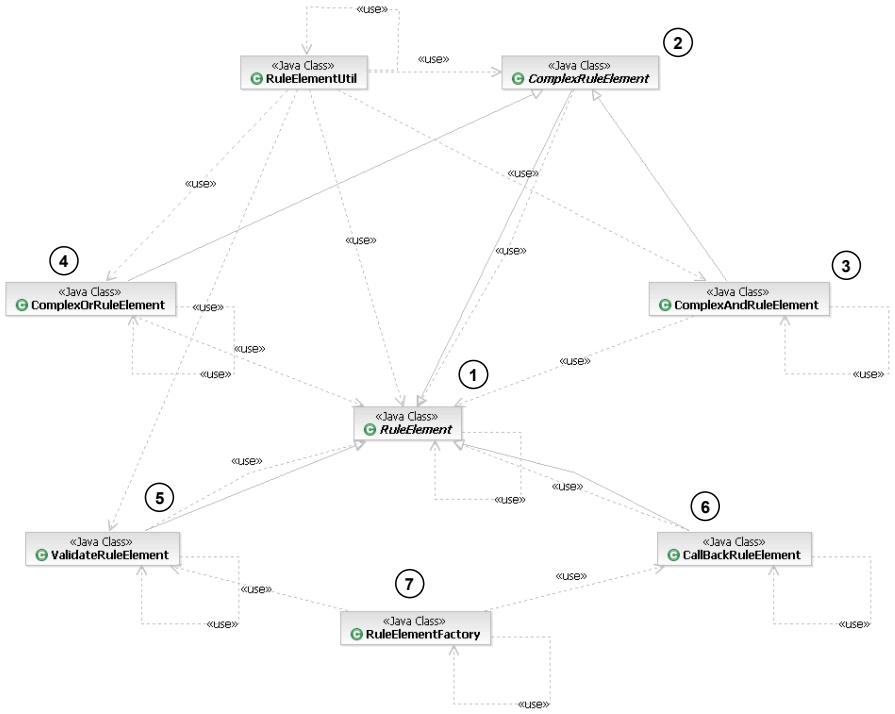


Figure 83: The rule element package.

RuleElement (at **1**) forms the abstract foundation for the package. ValidateRuleElement, CallBackRuleElement, and ComplexRuleElement extend the abstract class (at **2**). ComplexOrRuleElement and ComplexAndRuleElement both extend ComplexRuleElement (at **3**). The class factory is used to construct validate and callback rule elements.

The Function Package

The function package provides the ability to apply functions within rules, on rule operands. Functions instantiate rule elements, so they are constructed when rule elements are constructed.

There are two types of [abstract] functions; those that are CallBackFunctions and perform some operation on a class instance, and there are ValidateFunctions that return a Boolean value. Currently the only types of CallBack functions are associated with composition. ValidateFunctions are the type that compare component values or check some component property.

The function package is depicted in Figure 84:

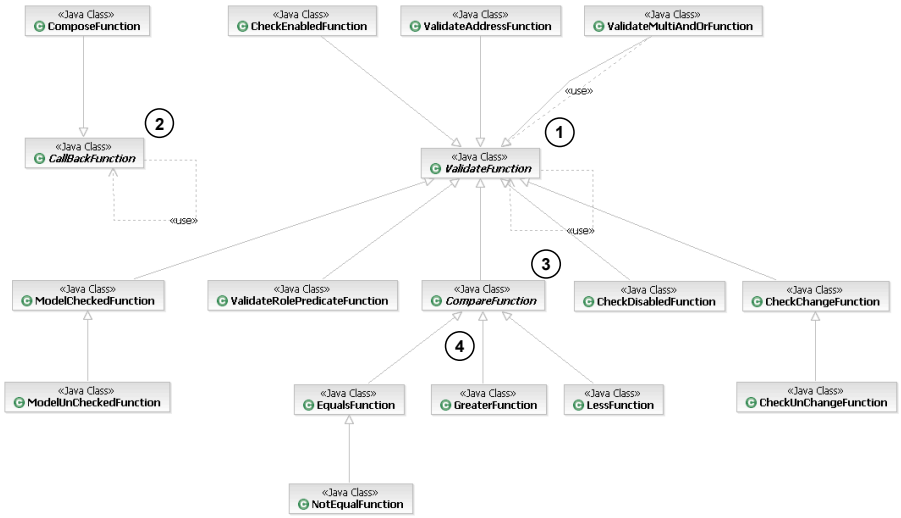


Figure 84: The function package.

The class diagram in the figure illustrates the two abstract classes, Validate Function (at **1**) and CallBackFunction (at **2**) which form the backbone of function support in the rule engine. ValidateRolePredicateFunction is an extension of ValidateFunction that is used for composition. Two function types, CheckEnabledFunction and CheckDisabledFunction, are used for user interface component dependencies. Two other function types, CheckChangeFunction and CheckUnChangeFunction are used for constraint satisfaction. CompareFunction (at **3**) is a specialization of ValidateFunction used for comparisons. Three CompareFunction subclasses are EqualsFunction, GreaterFunction, and LessFunction (at **4**). A NotEqualFunction extends EqualsFunction.

Function Library Extension

It should be noted that this is not a complete function library but simply a start. Other functions can be added that meet additional requirements for various inferencing tasks. The process whereby new functions are added is straight forward:

- **Create Class:** Create the class as an extension to an existing class. Provide a label that will match to the factory and to the RuleML. This will be either a CallBack type class or a Validate type class.
- **Update RuleElement Factory:** Add the new class to the factory with the label used to match against. Add the class to the correct portion of the factory: callback or validate.

The Fact Package

Facts form the 'other' foundation of a rule-based processing system. In MCT facts are really describing system (component) states, which limits the kinds of evaluations to viable system states since inferencing will start from and return to a collection of component states. The fact package defines the kinds of component states that can be described and follows the structure and organization of the rule element package.

The fact package is depicted in Figure 85:

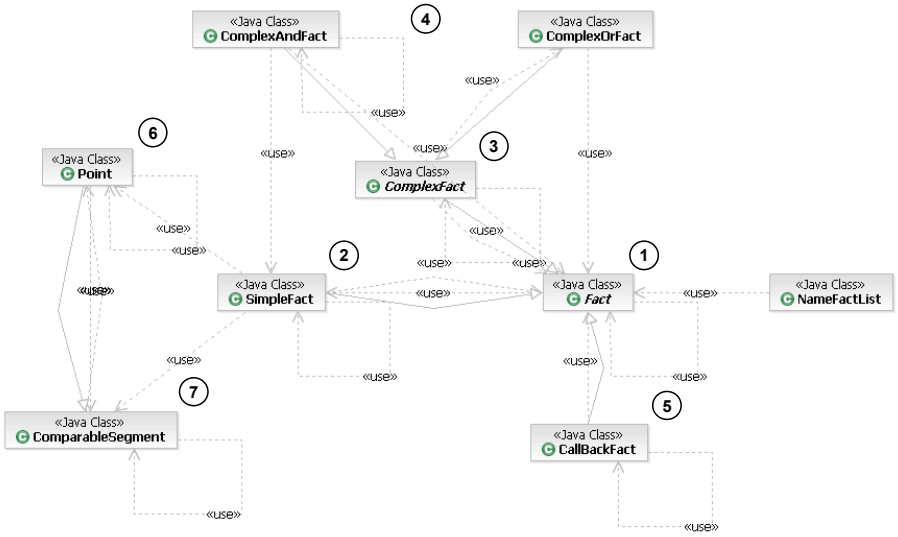


Figure 85: The fact package.

The figure illustrates that the abstract Fact class (at **1**) provides the baseline functionality for the package, with SimpleFact (at **2**), ComplexFact (at **3**), and CallBackFact (at **5**) providing the baseline implementations. ComplextOrFact and ComplexAndFact extend ComplexFact (at **4**). SimpleFact instances make use of Point and ComparableSegment classes for the purpose of comparisons.

The RuleML Package

The ruleml package is used to implement the layer that handles ruleml and implements the associated java as rules, rule elements, functions, and facts. It is primarily an import and export package.

The ruleml package is shown in Figure 86:

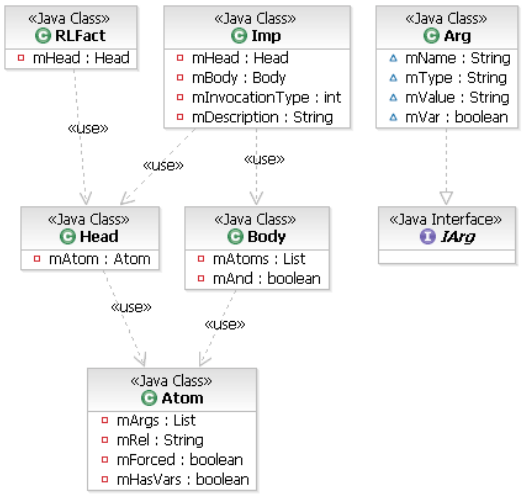


Figure 86: The ruleml package.

The figure illustrates that the java structure mirrors the RuleML structure, with the Imp class using elements from the Head and Body classes, which in turn use Atoms. Atoms make use of instances of the Arg class, which is used to represent both variables and independent object instances, and relations.

Summary

The rule engine packages provide the baseline functionality to perform a variety of inferencing tasks based on a forward-chaining rule and fact algorithm. The capability can make use of state changes in the system, for example with constraint satisfaction. The capability can also test component states or evaluate functions during execution, which are needed for using inferencing to manage UI states, or composition, respectively. The subsystem is extendable at the rule type, rule element type, fact type, and function type level while providing for interoperability of these types in different inferencing contexts.

Chapter 11 Constraint Validation

One use of a rule engine is to support inter-object constraint satisfaction. Currently there are no MCT applications that use this feature but it will be described as it is supported by the current rule engine with a few modifications to the component model.

Introduction to Constraint Validation

Most applications involve a user making changes to object values. In some case, those changed values can be considered without any consideration for other component values because they are semantically decoupled. For those occasions where component values semantically depend on other component values another approach must be implemented. That is, there are often dependencies on the data presented in one UI component or page and others. When a user types an IP address into a text field, not only must that IP address be a valid string, and lie within a particular range (xxx.yyy.zzz.www), it may also have to be on a particular subnet. Just as importantly, if a fixed IP address is selected in one portion of a page, other values may have to change on that page or others. For example, if we choose to use Auto IP detection, then we won't need to ask the user to set a DNS server address, since it would be inappropriate. In a user interface where the user can type in the DNS server address, and did so, a conflict would arise.

Inter-object dependencies must be validated, or satisfied, just as importantly as object or type/value validations. The difference is that there is now a context, and the previous object values that could be evaluated on their own merit can no longer be evaluated in the same way. What *is* required is a constraint validation mechanism and a set of rules for validating constraints when new and conflicting data is identified.

In some cases, constraints can be satisfied by enabling/disabling UI components, but components which are text fields would be problematic to constrain in such a way, so an alternate constraint validation mechanism is indicated.

Constraint Representation and RuleML

One mechanism being developed to address the constraint satisfaction issue is the Rule Markup Language (or RuleML, presented in 0). This is an XML-based language for representing semantic constraints of arbitrary complexity and intended for use with the Semantic Web. Using a language such as RuleML, inter-object constraints can be defined in a constraints/rules file, parsed during application initialization, and then the rules can be cycled through as an event-handling mechanism to validate the constraints prior to rendering a new page or, more importantly, saving component values.

An example showing how RuleML is used to represent constraints, for a single rule, is illustrated in Figure 87:

```
<?xml version="1.0" encoding="utf-8"?>
<rulebase direction="bidirectional"
          xmlns=http://www.efi.com/tsruleelements
          xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
          xsi:schemaLocation="http://www.mct.org/mctruleelements
                              \schemas\RuleML.xsd">
  <!-- EditGroupPropertiesDialog Rules -->
  <imp>
    <!-- * If anything on the UsersAndGroups edit group dialog changes,
         * Then call enableOKButton in UsersAndGroupsEditGroupDialogManager scope. -->
    <rulegroup>EditGroupProperties</rulegroup>
    <invocation>Page</invocation>
    <_head>
      <atom>
        <_opr>
          <rel>Configure.EditGroupProperty.enableOKButton</rel>
        </_opr>
      </atom>
    </_head>
    <_body>
      <atom>
        <_opr>
          <rel>checkChange</rel>
        </_opr>
        <ind>BUF::UserGroup.GroupDescription</ind>
        <!--ind>BUF::UserGroup.GroupPrivileges.GroupPrivilege(0).Value</ind-->
      </atom>
    </_body>
  </imp>
</rulebase>
```

Figure 87: A rule/constraint represented with MCT RuleML.

Constraint Validation

In MCT, what is sought is a situation where the set of rules that describe a condition are violated by the current/available user-supplied information or by states produced by rules and user-supplied changes. The available information is comprised of the values in the various Java models associated with the application contents. That is, the combined set of conditions *should* evaluate to true but doesn't. In this respect, the complexity of rule-based processing is really not needed in MCT constraint validation, because a predictive model isn't currently part of the design requirements. A predictive model *could* be employed, e.g. to fill in values on one page once values on another page are selected, but that is a separate matter for discussion.

Once violated rules are identified (i.e., a special form of exception), associated with them should be a (localized) display string that can be rendered in a dialog.

Chapter 12 Composition

Component composition is a foundational attribute of the MCT application framework. Composition means to aggregate functionally-equivalent representations into a viable, joined, representation that displays both. The MCT rule engine has been adapted to support component composition.

Introduction to Composition

A key functionality in MCT applications is the ability to visualize a component in different ways and to compose new visualizations from existing visualizations. The mechanism whereby components can be aggregated or disaggregated in MCT involves the rule engine and composition policies written as rules.

There are different approaches one can employ to perform composition. First, one could do it procedurally. In this case events of a proper type (such as drag and drop) would be recognized and the associated actions would check the source and target object types and do the appropriate thing. This approach would be brittle and the associated logic would be highly coupled to representations which can change dynamically.

Another approach is to use an rule engine and to construct the logic in rules that reside in a non-compiled form. This provides a number of benefits. First, the rules, which embody the logic for performing composition, are all in one place, making modifications most easy to effect. Second, the rules are decoupled from the logic of the engine. Third, the rules are very discrete and easy to locate, unlike embedded logics. Fourth, the rules can be changed at load time, or even run time, without rebuilding the subsystem. Finally, the rules can adapt to the dynamics of the representation components they are working with.

Composition Requirements and Use Cases

The use cases distinguished from those referring to the rule engine as specific to composition functionality are shown in Table 21:

Required Functionality	Use Cases?	Related Use Cases
CMPS1: User drag and drop capability is limited only by composition policies.	Yes	• USER drag COMP to COMP
CMPS2: Compositors shall be reusable code artifacts.	No	
CMPS3: The composition policy language shall include a built-in set of functions that make writing policies easier. (e.g., roleSatisfied and compose).	No	
CMPS4: In determining the semantics of a composition, the composition engine shall use as inputs the source representation, target representation, UI-managed environment information, user initiation action, and the rich MCT environment information offered to components.	No	
CMPS5: The composition language shall permit role satisfaction testing of components.	Yes?	• COMP satisfies Role
CMPS6: The system shall support the dynamic composition of components during application execution	No	
CMPS7: Composition is governed by composition policies.	Yes	• Entity compose Components
CMPS8: Users can create and modify compositions of user objects.	Yes	• Entity compose Components
CMPS9: Component de-composition of user	Yes	• Entity remove user

objects shall be possible.		object from container
CMPS10: User object composition and decomposition can be effected via user interface controls (menus, right clicking, keyboard shortcuts, composition).	Yes	• Entity compose with drag/drop • Entity compose with select/arrow keys

Table 21: Composition requirements and use cases.

Composition Policies

Several policies have been defined as being functional requirements for the composition capability in MCT. These policies are provided in Table 22:

Policy Description	Policy in Rule Notation
Adding a component of a specific kind to a collection restricted to components of a different kind, with role coercion.	If source satisfies role1 and target satisfies role2 then compose source to target and cast to role2.
Adding a component of any kind to a collection restricted to components of a specific kind, with role coercion.	If source satisfies role1 and target satisfies role1 then compose source to target.
Adding a predicted object to a specific composition.	If source has no model but has an implied activity role, and if target has an activity role, then compose source with target as activity.

Table 22: Composition policies.

There appear to be other composition policies defined, but they appear to be specializations, for the most part, of these policies.

How Composition with the Rule Engine Works

The process whereby the composition works is summarized below:

1) Select a knowledge base (currently performed through configuration)

2) Create component drop listeners for the affected components

3) Create component !Compose actors for the affected components

4) Identify the affected components and roles

5) Construct a new rule that includes the affected components and roles

Selecting a Knowledge Base

The rule engine can swap knowledge bases based on the semantic grouping desired. This mechanism was introduced to limit the number of rules that are active at any given time. The current MCT application is a telemetry application and the entire user interface is represented in a single main window. As such, there is neither a need to swap knowledge bases nor is there a need to select one at run time. For these reasons the selection has been done during rule engine subsystem configuration.

There is a file in the ruleengine package (*gov.nasa.arc.mct.ruleengine*) rules directory that keeps track of semantic group name and knowledge base name mappings. Currently this is called ISSTelemetryRuleMappings.txt. The content of the file maps a semantic name to a path to where the rules for that semantic group can be found (within the package). Currently the content of this file is as shown below:

```
TitanComposition      path_rule_titan_telemetry_xml
```

Thus any rule that is related to the Titan display should use the TitanComposition rule group (or semantic grouping). During configuration the ISSTelemetryRuleMappings.txt file is read and the results are placed into the RuleResManager.

In an application where windows are being swapped on a regular basis, it would make sense to swap the current knowledge base. For such cases, the RuleEngineContext has a method, called setKB(), to change the current knowledge base to a new semantic group. It would be invoked as follows:

```
mEnvironment.iengCtx().setKB([RuleGroupName]);
```

Everything else should work the same way as being described herein.

Create Component Drop Listeners

The manner in which the rule engine is invoked is through the event listeners attached to various operations. Generic drop listeners are not possible in MCT since components are not currently related, but the same listeners can be applied by implementing a listener that points to the abstract definition. This is the case in the example provided below:

```
public class WorrisomeDropListener extends AbstractCompositionDropListener {
    public WorrisomeDropListener(IComponent rep, IComponentEnvironment env) {
        super(rep, env);
    }
}
```

Of course, this means that any drop listener using this approach would be using the same drop functionality. Let's take a look at the AbstractCompositionDropListener and see what the impact of such an act would be. The only method of import is the drop() method:

```
public final void drop(DropTargetEvent event) {
    ICompositionContext context = (ICompositionContext) event.data;

    event.data = null;
    context.setTargetRepID(mRepresentation.getUniqueID());

    IComponent sourceRep = mEnvironment.platformCtx().findByID(context.getSourceRepID());

    mEnvironment.iengCtx().addFact(sourceRep.getName());
    mEnvironment.iengCtx().addFact(mRepresentation.getName());
    mEnvironment.iengCtx().triggerEngine(mRepresentation);
}
```

As you can see, there is nothing harmful or overly specialized in this approach whatsoever, so it is safe to use this approach for all composition (or for any inferencing). The result is that the first two steps in using the rule engine for composition are almost done for you.

Create Component !Compose Actors

Any form of composition will require some action to be performed as a consequence of an actionable composition event. If you look at a typical composition rule head then you will see what is going on:

```
<imp>
  <rulegroup>TitanComposition</rulegroup>
  <invocation>Chaining</invocation>
  <_head>
    <atom>
      <_opr>
        <rel>compose</rel>
      </_opr>
      <var name="?source1"/>
      <var name="?target1"/>
    </atom>
  </_head>
  <_body>
    <and>
      <atom>
        <_opr>
          <rel>validateRolePredicate</rel>
        </_opr>
        <var name="?source1" type="Elements"/>
        <ind name="ComposableRole"/>
      </atom>
      <atom>
        <_opr>
          <rel>validateRolePredicate</rel>
        </_opr>
        <var name="?target1" type="!Compose"/>
        <ind name="InspectorRole"/>
      </atom>
    </and>
  </_body>
</imp>
```

The first two lines identify the rulegroup this rule belongs to and the type of invocation, neither of which is needed for the current discussion.

The head portion of this rule states that the operator is 'compose' and that the arguments are both variables, named ?source1 and ?target1, respectively.[6] What this means is that if this particular rule is triggered (I'll get to that later), and selected, then a function called 'compose' will be invoked one these two arguments. I do not want to go into a lot of engineering detail here, but the 'compose' function is called a callback function and it is implemented in the gov.nasa.arc.mct.ie.rules.function package as ComposeFunction. The method that is ultimately invoked when the above rule is satisfied is the doCallBack() method:

[6] Variables use a prefix '?' by convention. This is checked by the MCT rule engine.

```
public ArrayList doCallBack() {
    IUserPlatformContext ctx    = RuleEngine.getInstance().getEnvironment().platformCtx();
    ArrayList            params = new ArrayList();

    for (int index = 0; index < mParams.size(); index++) {
        Arg param = (Arg) mParams.get(index);

        if (param.isVar())
            params.add(resolve(param));
        else
            params.add(param);
    }

    IComponent source = (IComponent) ctx.findByName((String) params.get(0));
    IComponent target = (IComponent) ctx.findByName((String) params.get(1));

    target.receiveMsg("!Compose", source);
    return(params);
}
```

Almost everything in this method can be ignored except the highlighted lines. That is because everything else is associated with tracking the arguments internally to the rule engine. The source and target are retrieved from the component registry using the names provided in the rule (sans question mark character), but by now they have been resolved as variables and matched to the appropriate component names. The effect is to call the receiveMsg method called !Compose on the target component with the source component as its other argument.

As you can see, if the !Compose method is defined on the target, the rule engine actually has no knowledge whatsoever of the actual behavior in that method, only that it is supposed to invoke it.

The !Compose method is defined by the relationships in the declarative representation of the component. Below is part of one such definition for the ScratchpadPanel:

```
<component name="ScratchpadPanel">
  <field name="!open">
    <default-value
      fail-on-error="false"
      remove-inherited-values="false">
      <actor-value class="gov.nasa.arc.mct.reps.scratchpad.ScratchpadPanelOpenActor"/>
    </default-value>
  </field>
  <field name="LeftPanelElement"/>
  <field name="InspectorRep"/>
  <field name="!Compose">
    <default-value
      fail-on-error="false"
      remove-inherited-values="false">
      <actor-value class="gov.nasa.arc.mct.reps.scratchpad.InsertNewWorrisomeActor"/>
    </default-value>
  </field>
  <field name="!Update">
    <default-value
      fail-on-error="false"
      remove-inherited-values="false">
```

```
      <actor-value class="gov.nasa.arc.mct.reps.scratchpad.UpdateScratchpadActor"/>
    </default-value>
  </field>
  <roleref role-name="RepresentationRole"/>
  <field name="!AttachModel">
    <default-value
      fail-on-error="false"
      remove-inherited-values="false">
      <actor-value class="gov.nasa.arc.mct.reps.common.SetModelActor"/>
    </default-value>
  </field>
  <roleref role-name="ComposableRole"/>
</component>
```

As you can see, the !Compose field maps to the InserNewWorrisomeActor class in ov.nasa.arc.mct.reps.scratchpad. That act method in that class is defined below:

```
public Object act(IComponent recipient, IComponentEnvironment env,
                  String index, Object... args) throws MCException {
    IComponent model = (IComponent)recipient.getValue("Model");
    model.addValue("Elements", (IComponent) args[0]);
    return(null);
}
```

So the value sent in to act upon, the recipient, has its model retrieved and then adds the other arguments sent in (the source from the calling function) to a predefined field named “Elements”. As you can see, this is a bit brittle. What would happen, for example, if the declarative definition of the drop slot name were to change. Now the code would be a hostage to that change. One approach to alleviate this would be to define a ‘drop field’ field in the declarative representation and to use the value bound to that field in the addValue call above. But that isn’t extremely important right now.

Identify Affected Components and Roles

Before a new rule can be completely written you need to identify the components associated with the source and target by their required roles and attributes. Since every composition rule is defined by conditions used to ‘trigger’ (i.e., enable) it, your rule will not trigger unless the conditions you define are actually met by components in the application. It isn’t enough to start dragging things around and expect them to compose magically (although that might be nice – there are drawbacks). So let’s use the deployment/resources/module1 package. In this package we have an XML file that defines all of the components needed for the view context. When the framework and view context is launched all of these definitions will be loaded into the component and role registries. Any of the elements that are actually ‘used’ will become available for rule-based processing.

Let’s say we want to drag SGANT temperature data into an editable telemetry group housing. What objects are available form the model role definitions of the XML files:

The TelemetryGroup component definition:

```
Fields:

Attributes - Elements
Behaviors - !Init
```

Roles:

ComposableRole

We need some attribute that defines this component somewhat uniquely, and we need a role to satisfy. We select the Elements field and the ComposableRole as they fit our requirements.

From the reps plugin.xml we can get the SGANT_InspectorRep component definition:

Fields:

```
Attributes - TemperatureGroupRep, VerticalScaleMin, VerticalScaleMax
Behaviors - !Init, !GetElements, !Compose
```

Roles:

```
RepresentationRole, InspectorRole
```

In this case the attribute we use is the fact that there exists a !Compose operation defined, since that clearly allows for composability, and the InspectorRole will have to do for the role requirement, barring something better. So the chosen items are highlighted in bold.

Construct New Rule

Rule construction, as long as you are sticking, for the time-being, to role-satisfaction composition rules, is really easy. You take a sample rule like the one provided above, and again below and copy in the values you found in the last step. Here is how you go about it (use the rule provided below as a template):

1) Change the variable names (defensive, variable names should be unique in the rules file). So, for example change ?source1 to ?sourcejbh23, and ?target1 to ?targetjbh23. The name is arbitrary other than the requirement that it be unique in the rules file. Notice that this will result in 4 edits.

2) Take the item found as the drag item in the last section and plug it into the 'source' section of the rule body. Specifically, the name of the field required to identify this component, and the name of the role that this component is required to satisfy. The latter is because this rule is based on the validateRolePredicate validation function.

3) Take the item found as the drop item in the last section and plug its values into the 'target' section of the rule body. Specifically, the name of the field required to identify this component, and the name of the role that this component must satisfy.

That's it, all you have to do is to add this rule to the rules file in the rules directory for the gov.nasa.arc.mct.ruleengine package and run the application:

```
<imp>
  <_head>
    <atom>
      <_opr>
        <rel>compose</rel>
      </_opr>
        <var name="?sourcejbh23"/>
        <var name="?targetjbh23"/>
    </atom>
  </_head>
  <_body>
```

```
    <and>
      <atom>
        <_opr>
          <rel>validateRolePredicate</rel>
        </_opr>
        <var name="?sourcejbh23" type="Elements"/>
        <ind name="ComposableRole"/>
      </atom>
      <atom>
        <_opr>
          <rel>validateRolePredicate</rel>
        </_opr>
        <var name="?targetjbh23" type="!Compose"/>
        <ind name="InspectorRole"/>
      </atom>
    </and>
  </_body>
</imp>
```

The trace resulting from the run should show whether the rule was triggered (not that again), what the resulting action should have been. Of course, if the composition worked you will see the results in the user interface.

OK, what does it mean to trigger a rule? Simply put triggering means that all of the antecedents for a rule are satisfied. In a complex system this doesn't guarantee that a rule will be fired, only that it is active. There are other tests that need to be performed before a rule can be invoked. For one thing, has this rule already been fired by the same set of circumstances? If so, it would signal a cyclic action and that would not be good, so the rule engine has to trap such occurrences and not be suckered into action. Second, does this rule produce any effect? If not, why bother with it? In composition this isn't an issue, but for chaining systems in general, a rule might try to produce only new information which is already available, and that would not be worth the effort. Beyond this there are the issues of how to order viable rules and, beyond that, how to execute them. The bottom line is that if the composition circumstances trigger a rule, the current system should ultimately enable the composition and it should appear.

As more components come on line it will be easier to add composability because there will be more to work with.

Chapter 13 Data Validation

Data validation is important when there is data that gets loaded into a system where the data integrity with the system is critical or where data is modified by a user. In such cases it is important to have a mechanism that can be applied uniformly and yet be flexible enough to change validation criteria without modifying the associated codebase. The MCT data validation mechanism meets these requirements and manages the validation of any loaded or changed data.

Introduction to Data Type and Value Validation

Data validation is important whenever a system undergoes some change in the state of runtime data values. The state might be at launch time when there is no state to an initial, but consistent and coherent, state. The state might be at run time when an object is modified in a manner that could produce inconsistent values. At such times the framework is responsible for guaranteeing the integrity of data values. In MCT data loading and modification are associated with model components, and the UserPlatform manages all messages to model components as well as their lifecycle, data validation must be an element of the UserPlatform.

Validation takes two forms in MCT: (1) type validation and (2) constraint validation. Type validation refers to validating that a value as a type, that it is within a valid range, or has a particular default value, or a particular cardinality. Type validation refers to a property of an object, so it is considered context-free. That is, it doesn't rely on the types or values of other objects. On the other hand, constraint validation refers to value constraints imposed on multiple objects by the context of their interaction. If one of the objects takes on a value which is incompatible with other objects in the same dependency group, then a violation is flagged. This chapter will address only type validation. Constraint validation is addressed in the next chapter (Chapter 11).

Constraints to Data Validation Mechanism Design

The design of the data validation mechanism is driven by the following 5 constraints and requirements:

- **Transparency**: The data validation mechanism should be transparent to operations on model components.
- **Integration**: The data validation mechanism must integrate with the UserPlatform.
- **Flexibility**: Changes to validation should be possible without modifying the source code.
- **Modularity**: It should be possible to add new validators that encapsulate specific functionality but also support validator inheritance.
- **Generality**: It should be possible to use the same set of validators for any MCT framework element.

These requirements and constraints limit the types of approach that can be employed in designing a data validation mechanism. In particular, the flexibility, modularity, and generality requirements combined suggest a declarative model for defining validator to model mappings. This approach allows validators to be defined on a per-component basis while still providing a generalized mechanism for handling validation.

It is recommended that, for the short term, declarative validation mappings adhere to an XML Schema approach, and that, later, this model be put into an ontology and loaded at launch time, if possible.

Types of Data Validation

Data validation can be very general or very specific to an organization's needs. An example of a general validator might be a simple check against a data type, such as an

integer or string. A validator can just as easily check against minimum and maximum, or even range values. In some cases one might wish to validate complex field types, such as IP addresses or email addresses. Below is a short list of possible data validators:

- **StringValidator**: Generally a string validation is used to distinguish the value from numeric or Boolean values. In some cases the intention is to specialize the string more carefully, for example to limit the number of characters (min and max) that can be used in the field. Such validators are often used in server names, and password fields.
- **AlphanumericValidator**: In some applications it is important to allow only alphanumeric characters to be input by users. One obvious example is passwords or email addresses. In such cases what is sought is a general way of representing alphanumerics that enables the user to specialize how alphanumeric is interpreted at the attribute level. Some of the characteristics of an alphanumeric validator are: is it a pure alphanumeric or just alpha, can it be empty, does it match special characters and, if so, which ones.
- **IncrementValidator**: When widgets like spinners are used, sometimes they do not adequately enforce their own rules. In such cases having an increment validator allows the developer to enforce the rules at the widget. For example, if a user writes a value that is between two legal values the widget can be forced to take a legal value in the direction of their choosing.
- **IPAddressValidator**: Checks to make sure that all of the values in an IPv4 address are legal. IPv6 would be much more complicated to validate.
- **EMailAddressValidator**: Checks to make sure that the baseline email address follows established conventions.
- **GeneralNumericValidator**: Checks values for numeric entries. Generally includes both min, max, min_inclusive, and max_inclusive values.

Data Type, Value, and Range Validation

The MCT component model makes JVM-based data validation impossible because the names, types, and cardinality of objects are embedded in component fields and facets, and these may not be known until runtime (or they may change at runtime). As such, the validation mechanism must be bound to the component instances.

Data type validation is performed at three stages in the MCT model: (1) during generation when the XML files are validated against the XSD, (2) at load time when the UI representation components are first added into the registry, and (3) at runtime when change events are handled.

Generate-Time Validation

When the components are initially created, they must satisfy the constraints imposed by the OWL or XML Schema (XSD) that defines the models and their data elements. This level of validation essentially certifies that every element in the model follows the model definition and that the data elements included therein follow the type and range definitions of the model. This level of certification is static in that it is a creation phase of validation only.

Runtime Validation

There are two types of runtime validation, at: (1) initialization, and (2) when values undergo modification.

Initial Data Type and Value Validation

When representation components are initially loaded into the environment it is possible to run a validation test on newly acquired values. This is a sanity check in case they were accidentally stored improperly or someone modified the validation files (or the data attributes) between the time they were created and when the application was launched. This way it is ensured that all the data that is displayed is valid.

Modified Value Validation

By far the most prevalent validation type is required when component values are modified.

The difference between these two run-time validation types is that in the initialization case there is no value stored, so as long as the value is valid it is fine to save it immediately. In the case of modified value validation, the modified value must be validated and, if successful, replaces the earlier value. In both cases the same mechanism is employed.

Data Validation Requirements and Use Cases

The use cases identified with the Data Validation functionality are presented in Table 23:

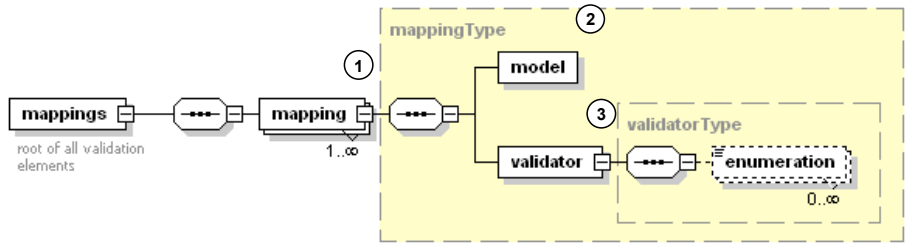
Required Functionality	Use Cases?	Related Use Cases
VALD1: UI widget parameter modifications are verifiable.	Yes	• Entity changes UI widget value
VALD2: Changed component field values shall undergo a validation before values are assigned.	Yes	• Entity changes Component value
VALD3: Validation shall include data type checking for strings, Booleans, and numbers.	Yes	• Entity changes Component string, Boolean, or number value
VALD4: String type checking shall include min and max number of characters.	Yes	• Entity changes string Component value that has a minimum and maximum number of characters
VALD5: String type checking shall include variations of alphanumeric strings.	Yes	• Entity changes alphanumeric Component value
VALD6: Integer type checking shall include min, max, default and increment checks.	Yes	• Entity changes integer range Component value
VALD7: Number type checking shall include min, min_inclusive, max, and max_inclusive tests.	Yes	• Entity changes numeric range Component value
VALD8: Specialty validators shall be supported (e.g., email addresses and ip addresses).	Yes	• Entity changes address Component value
VALD9: Declaration of component field validations shall be kept separate from the code that	Yes	• SYS loads validations

implements them.	
VALD10: The central component validation mechanism shall support component field data type adherence.	No
VALD11: The central component validation mechanism shall support component field cardinality adherence.	No
VALD12: The central component validation mechanism shall enforce the adherence of field semantics.	No

Table 23: Data Validation requirements and use cases.

Validation Schema

MCT currently uses an XML Schema to define a minimalist structure around validation entries. The schema used for performing data validation is shown in Figure 88:



Model attributes: (4)
name [xs:string]

Validator attributes: (5)
min_length [xs:integer]
max_length [xs:integer]
type [xs:string, in StringValidator, IntegerValidator, IncrementedIntegerValidator, IPAddressValidator, HexStringValidator, BooleanValidator, AlphaNumericStringValidator]
isORAlpha [xs:Boolean]
isORMatching [xs:Boolean]
isNullable [xs:Boolean]
isORSpecial [xs:Boolean]
min_inclusive [xs:number]
max_inclusive [xs:number]
increment [xs:integer]
isORNullable [xs:Boolean]
min_value [xs:number]
max_value [xs:number]

Figure 88: Data validation schema and attributes.

The schema has a root/organizational node called mappings which is comprised of any number of mapping elements (at **1**). A mapping element has both a model reference (at **2**) and a validator (at **3**). The validator element serves to identify which validator type will be

used and what its attributes will be. The model element has a single attribute representing both the name of the component and the field associated with this validator (at **4**). The validator has a (mandatory) type attribute which is assigned the name of the validator being used. It also has several possible attributes (at **5**) based on the various validator types available.

An example entry in validation.xml is provided below:

```
<mapping>
   <model name="Login.Password"/>
   <validator type="StringValidator" min_len="3" max_len="15"/>
</mapping>
```

In this example the model element identifies the component the validator will be associated with, in this case the model name is “Login” and the field name is “Password”. The validator element provides the type and values, as attributes, that are used to validate this model. In this case, the values are validated using StringValidator, and the string must be between 3 and 15 characters in length.

Data Validation Workflow

Data type, value, range, and cardinality is processed at two times during the existence of an MCT application interface: (1) during initialization, and (2) when a field changes value. In both cases, a call to receiveMsg will be made. The call must be intercepted by the various validation mechanisms before the component value can be set. Whether this is done in an aspect-oriented fashion or whether it is a part of the Component code is relatively unimportant.

The processing outline for the initialization mode is:

- Retrieve the component model associated with the event
- Test the data type against that expected
- Test the value against that expected
- Set data validation to true/false
- Assign the value

When actions are performed in the interface that require data requests or updates, a new data/value validation must be performed. The processing mechanism for this mode is similar to the one for initial data validation:

- Retrieve the component model associated with the event
- Test the data type against that expected
- Test the value against that expected
- Set data validation to true/false
- Associate a dialog string for invalid data
- After user provides a data value, run check as above
- Assign the value

Validation Aspects

The validation process has three steps: (1) creation of the declarative validations file, validation.xml, (2) runtime parsing of validation.xml into validators mapped to components, and (3) evaluation of values against a validator when values change. In all three steps a validator must exist for every entry in validation.xml.

Validation File

As mentioned previously a validation file will exist that maps model elements to validation types. This file will conform to an information model or an XML schema, and will reside in the deployment/resources/modeule package. The validation code is part of the User Platform as a component service.

Validators

A validator is a java component that is invoked when objects are initialized or modified and is intended to be a general functional model for objects of a specific type. That is, they are both general, and configurable. Validators are managed by a validation manager. Each validator, as noted in the schema above, may have attributes which are parsed from validation.xml at load time and which are then matched against object values at load time or run time. Since the approach isn't based on java bean structure, it can be applied to the MCT component model. Each validator also has a validate method that implements the logic which it embodies.

Validators are implemented in a way that they can extend functionality, so it is possible to use a form of inheritance when testing an object value. For example, if a validator is set on an object as an AlphaNumericValidator an attribute for StringValidatorEx can be placed in the validator and, assuming that the validation succeeds for AlphaNumericValidator it would then be passed to the StringValidatorEx to check the attribute values associated with it that aren't a part of AlphaNumericValidator.

There are instances where data validators cannot be used to test runtime changes in the user interface. These are where multiple values must be input before a validation test can be made. Most notable in this group are validation of password changes, which generally require the user to input the password twice. These validations are first passed through a validator to make certain that the first value is valid, and when the second value is provided it is checked to see if it is the same as the first.

Validator Types and Creation

Validators exist in *platform.comp.validators* and are based on the org.exolab.castor validation project. There are several validators defined already:

- AlphaNumericStringValidator
- BooleanValidatorExtension
- DateTimeValidator
- DoubleValidatorExtension
- EmailAddressValidator
- FloatValidatorExtension
- HexStringValidator

- IncrementedFloatValidator
- IncrementedIntegerValidator
- IntegerValidatorExtension
- LongValidatorExtension
- StringValidatorExtension

Each validator defines a set of members that are associated with the attributes in the XML file, a constructor that takes those attributes, and a validate method that takes the object to validate and a ValidationContext but which is not used. The validate method uses a data object (in the case of AlphaNumericStringValidator an AlphaNumeric class, defined in platform.utils) that has a predicate isValid() to test whether an object value is valid. The remainder of the validate function is used to determine what string to return (and to localize it) depending on the type of validation failure.

Model Validation Parsing

The validation file, MCTValidations.xml, is parsed in FrameworkLauncher.launch() with a call to parseValidation that is in the FrameworkModelMgr class. The path to the validation file is stored in mct.properties and is stored in FrameworkModelMgr by the FrameworkLauncher.initializeModelManager method. Both initializeModelManger and launch are called, consecutively, by the constructor.

The parseValidation method uses XMLFile to read the XML file into a Document and then passes the Document to the ValidatorResolver class, where the mappings are parsed. Somewhere down the call sequence each of the mappings is parsed and added into a table, referenced by the model id. The sequence is shown in Figure 89:

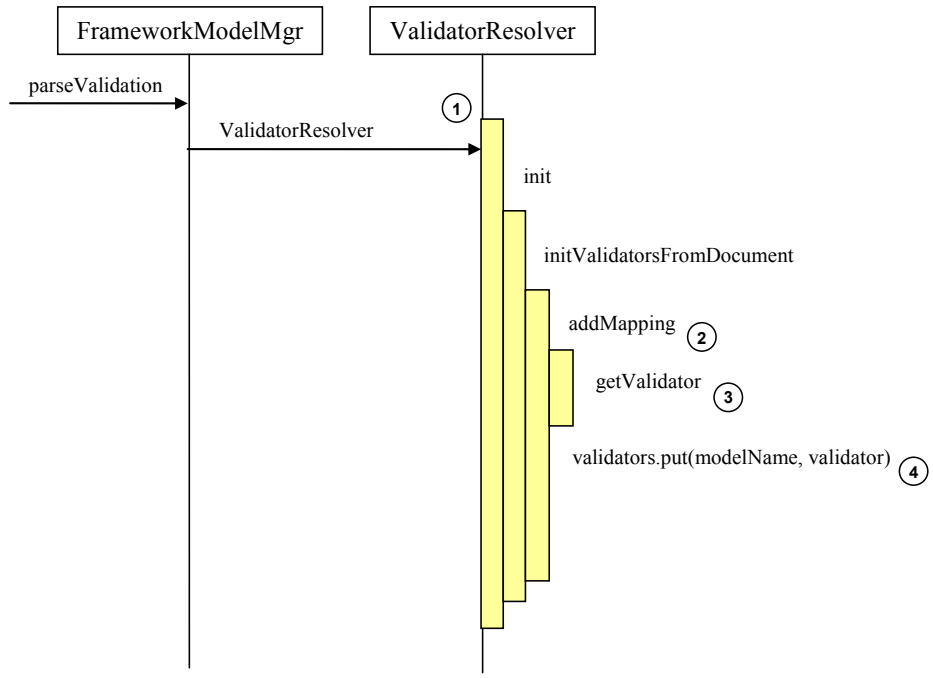


Figure 89: Validation parsing workflow.

Model Validator Assignment

Validation assignment is part of the parsing process. In the FrameworkModelMgr call to parseValidatin the ValidatorResolver is instantiated (at **1** above). For each mapping in the parsed file a call to addMapping (at **2**) is made, and this results in storage of the model/validator (the validator is derefenced at **3**) pair in a validator registry (at **4**).

Runtime Validation

Validation is invoked when a model is initialized or updated. For example when a property change event occurs. The model type implements PropertyChangeListener and calls setValue on the model, and this in turn calls validate. The validate method is defined at the model level. The validate method requires that the validator is defined, which is done by calling setValidator in the model creation method. As you may recall, the model is created in FrameworkModelMgr during the validator parsing workflow.

Chapter 14 Messaging

MCT as a framework must interact with enterprise applications and services. It must also provide the ability for components and MCT clients to interact across network boundaries. MCT is not only a consumer of messaged but is also a provider of messages. Thus MCT can be seen as a networking peer in a network of service-oriented peers. The aim of providing a messaging interface or layer in MCT is to achieve these goals while insulating MCT from the construction of code specific to any networking interface or protocol. It is also the intention of this design that the MCT messaging layer interact with the CSI framework.

Introduction to Messaging

The MCT messaging subsystem is responsible for providing access to and from MCT to the outside world. This could take the form of messaging to MCT components on other clients, or general messaging to enterprise applications using disparate interfaces and protocols. The messaging subsystem is responsible for providing transparent interfaces to whatever capabilities are provided, or needed.

Four general types of communication are needed by and used in MCT: (1) synchronous and asynchronous data communication with 3rd-party enterprise applications such as ISP or the CSI Framework, (2) synchronous data serialization and storage for persistence, (3) synchronous and asynchronous messaging between components in a publish and subscribe paradigm, and (4) synchronous and asynchronous messaging between platform instances in a peer-to-peer paradigm.

Messaging Requirements and Use Cases

The use cases identified for the Messaging subsystem are shown in Table 24:

Use Case	Use Cases?	Related Use Cases
COM1: MCT shall provide a central Messaging layer that will enable MCT to communicate through an abstract communication interface that is application independent.	No	
COM2: The central Messaging layer shall be configurable.	Yes	• COM configure COM
COM3: The central Messaging layer shall be policy based (e.g., message durability, transience, recent).	Yes	• COM policy based
COM4: The central Messaging layer will be available to services and subsystems through the shared platform environment.	Yes	• SYS access COM
COM5: The central Messaging layer shall provide a base set of common communication adapters that can be used by specific proxy component implementations.	No	
COM6: A network of peer-to-peer execution environments shall serve as the layer between component communication and the underlying network topology.	No	
COM7: User Platforms shall be able to communicate with other user platforms on the network.	Yes?	• UP update
COM8: There shall be a distinction between local and remote communication from the point of view of the components.	Yes	Eleven use cases have been identified among the 17 Messaging-specific

		requirements, as detailed below: • COMP message COMP
COM9: The central Messaging layer shall include a publish and subscribe communication mechanism.	Yes	• COMP subscribe to COMP field • COMP publish field value
COM10: The following are requirements of any system used to implement publish/subscribe for MCT: subscription durability (messages delivered while the subcriber is unavailable are delivered when it becomes available again), subscription transience (a subscriber only receives those messages that are delivered while it is available), and subscription recency (a subscriber receives the last N messages that were delivered while the subscriber was unavailable).	No	
COM11: Components shall be able to subscribe to messages by their content, where content is defined according to a publish/subscribe language/logic.	Yes	• COMP subscribe to COMP field
COM12: Components shall be able to subscribe to messages by their type, where type is defined according to a publish/subscribe language/logic.	Yes	• COMP subscribe to COMP type
COM13: Publishers shall be able to specify a lease for a message.	Yes	• COMP publish by lease
COM14: Components shall have the ability to communicate directly with other components.	Yes	• COMP message COMP
COM15: The central Messaging layer shall provide a synchronous communication mechanism for component to component communication.	Yes	• COMP message COMP
COM16: The central Messaging layer shall provide an asynchronous communication mechanism for component to component communication.	Yes	• COMP message COMP
COM17: Publish and Subscribe language will satisfy the requirements defined in table COM1.	Yes	• COMP subscribe to COMP field • COMP publish COMP field
COM18: Component to component communication shall be loosely coupled.	Yes	• COMP message COMP
COM19: Changes to component state shall be propagated to all components with registered interest, particularly active representations visualizing that state.	Yes	• COMP message COMP

Table 24: Messaging requirements and use cases.

Messaging

Messaging is a mechanism for objects or services to interact, generally across the network. MCT is a distributed application environment, so it makes use of messaging at various levels and in various ways. Currently MCT uses messaging in 4 ways:

- **Subsystem Interaction:** When a subsystem needs another subsystem to update based on change that has taken place the pub/sub mechanism can be used to inform subsystems to update.
- **Component Interaction:** This is the nominal use of pub/sub to update components that are following changes to other components.
- **External Services Integration:** This is an implicit use of messaging through a common message bus. Examples include persistence, data access, and information access. These are illustrated in Figure 90:
- **Client Interaction:** This is where an MCT client joins an MCT client network and uses pub/sub to specify client updates.

The general architectural assumption is that the message bus and related utilities will be provided by an external source.

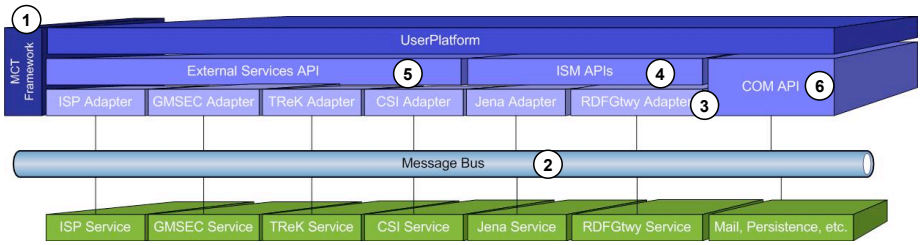


Figure 90: Use of a messaging bus for external services.

The figure illustrates how external services that themselves make use of a common message bus (at **2**) can be hooked up to the MCT ISM API (at **4**) and ExternalServices API (at **5**) using adapters (at **3**). The intent is to provide access to external services without binding them to the MCT architecture. This requires general-purpose APIs for both the ISM and ExternalServices subsystems but also provides for a plug-and-play approach to integrating such services with MCT.

General Messaging and the CSI Framework

MCT is designed around its own messaging APIs so that messaging mechanisms can be tested prior to a uniform messaging framework becoming available. The general structure of the MCT messaging APIs is shown in Figure 91:

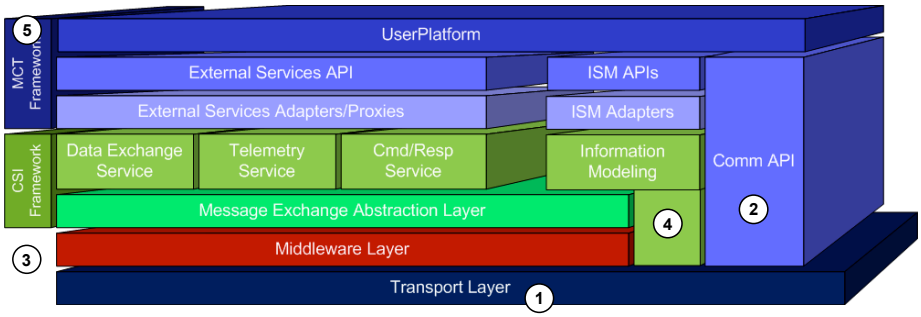


Figure 91: MCT messaging architecture.

This figure illustrates an extended view of the messaging architecture from the last figure. In particular, it shows the transport layer (at **1**) and how the CSI Framework (at **3**) is expected to connect to the transport and middleware networking layers. The CSI Framework has its own notion of services and information modeling (at **4**), unlike the MCT design, connects directly to the transport layer. In this figure, the MCT messaging API (at **2**) also has direct access to the transport layer, but the CSI Framework is expected to provide pass-through capability to do so. Everything at this level or above in the figure is part of the MCT framework (at **5**).

Messaging Implementation

The messaging layer is divided into two types: (1) pub/sub messaging and (2) external services. As previously mentioned, pub/sub messaging is not part of the ExternalServices subsystem but, rather, a part of the general messaging API, but it serves the ISM APIs as well as component pub/sub requirements. ExternalServices APIs are intended to provide adapters to specific 3rd-party applications. Web services are expected to be folded into the ExternalServices layer.

An architectural diagram showing the functional capabilities for both pub/sub and p2p messaging is illustrated in Figure 92:

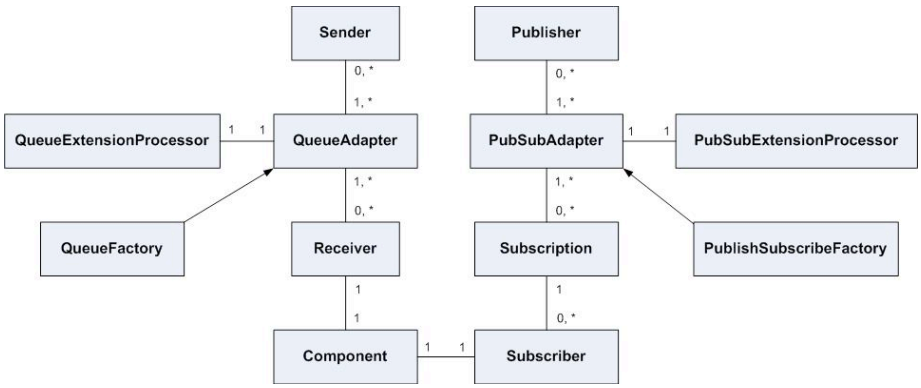


Figure 92: Messaging layer relationship diagram.

Messaging APIs

Dsdsd

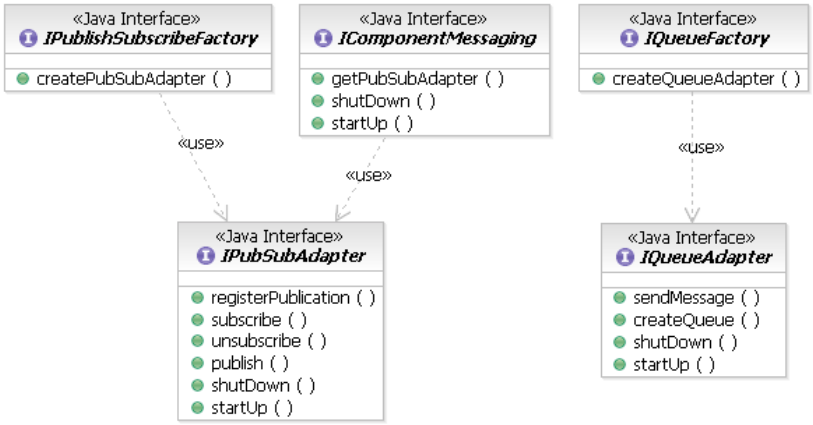


Figure 93: General messaging API.

Dsdsd

Figure 94: Basic messaging service.

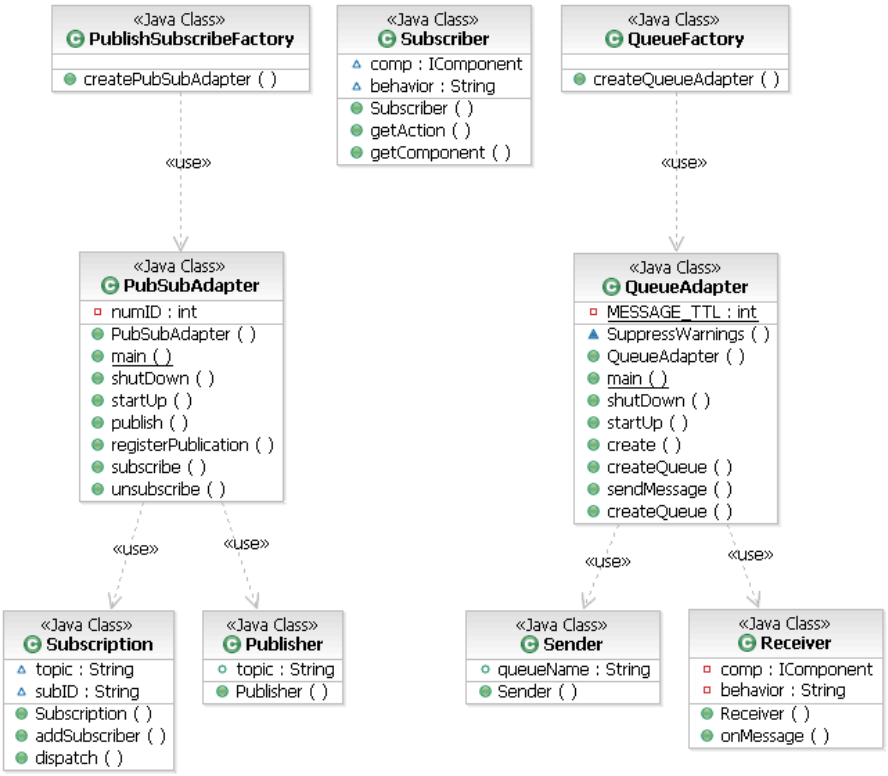


Figure 95: Standalone component messaging implementation.

Sdsd

Figure 96: Mantaray component messaging implementation.

Chapter 15 Persistence

Persistence is associated with caching or buffering runtime (volatile) information to long-term (non-volatile) store. Persistence is used to handle crash recovery or session management, to support customization, to reload an environment, or to speed up performance. A persistence mechanism needs to be implemented programmatically at the point objects are being manipulated, and in the case of MCT that is in the UserPlatform template registry.

Introduction to Persistence Management

MCT has many needs associated with short ←→ long term data storage. Its ontological information must be stored outside of the framework in order to achieve maximum interoperability. This includes any conceptual definitions (rules, configuration schema, policy schema, component modeling, domain models). MCT must also have the ability to save information that isn't conceptual/definitional but, rather, the values that are bound to the conceptual definition at some moment in time. This can include a user's session id, preference information, component layout and attributes, configuration files, and maybe even runtime data in some cases. The former are considered resources because they generally do not change – their source is external to the framework. The latter, on the other hand, originate within the framework and thus must be managed by the framework. The management of volatile data to non-volatile data in MCT defines the role of persistence.

Persistence Mechanism Dependencies

The persistence of information is dependent on 5 systems and mechanisms:

- **UserPlatform:** The platform is responsible for all lifecycle management in MCT and so the persistence mechanism must be part of the UserPlatform.
- **Information Semantics Manager:** The ISM manages ontological models in several states of instantiation. The persistence mechanism must interact with the ISM to guarantee that values are synchronized and that, at the very least, models are synchronized.
- **Policy Management:** The manner in which the persistence mechanism works should be runtime managed according to [possibly interacting] policies for determining how/when to apply the mechanism to particular types of data.
- **Data Management:** MCT works with large amounts of data and a mechanism is needed for managing the local cache that interacts with the persistence mechanism.
- **Persistence Management System:** The MCT persistence management mechanism must interact with any number of external systems for persisting information.

The amount of dependency precludes the persistence mechanism from being implemented as a standalone subsystem, mostly because of its intimate relationship with the component model and user platform. Every attempt should be made to isolate its behavior from the rest of the framework.

What to Persist

These framework dependencies are somewhat independent of what is being persisted, but it is essential to identify what types of information need to be persisted:

- **User:** There are four things that must be persisted for the user. First is the user's session, in case there are multiple users or in case the system fails, so that buffered and long-term data can be restored. Second, user preferences must be persisted, enabling the user to configure the general appearance and behavior of

their environment. Third, the user environment itself must be persisted. Finally, components the user has manipulated must be persisted.

- **Services and Subsystems:** For framework elements and files that can be modified at runtime should be persisted. In some cases this might include configuration and policy files.
- **Application UI State:** MCT is a framework for defining and executing application interfaces. As a user makes use of the environment they will add or move items, aggregate items, define filters, etc., all of which need to be saved for future sessions, or not, based on the user's wishes.
- **Events:** MCT maintains event logs and these have both a runtime and a persisted state. MCT must be able to acquire persisted events in the same manner that the non-persisted events are.
- **Buffered Telemetry Data:** MCT buffers telemetry data for display, but the ODRC data repository is not equipped to provide data between the beginning of a session and 15 minutes prior. As such, MCT must persist its realtime data, and be able to retrieve it, and be able to synchronize with ODRC data, to provide a near seamless data record. This persisted data is cleared at the end of a session since it will all be archived to ODRC.

Persistence Management Constraints

The persistence mechanism cannot operate in a vacuum. It has 13 constraints on its design:

- **Workflow:** The mechanism must intercept receiveMsg calls and dispatch appropriately in a transparent manner to framework services and subsystems.
- **Encapsulation;** The mechanism must be encapsulated with respect to both the user interface and to the underlying persistence implementation.
- **Cache:** The mechanism must work in concert with a cache management system.
- **Policy managed:** The mechanism must support polices for cache and long-term stores.
- **Underlying mechanism:** The mechanism should be transparent to how persistence is implemented, and multiple persistence mechanisms should be supported.
- **Component support:** The mechanism must work with MCT components.
- **Transaction support:** It may be necessary to support multiple/dependent transactions so that rollback can occur if all transactions don't commit.
- **SQL support:** It may be necessary to support SQL-type queries, depending on the underlying mechanism.
- **Unique identifiers:** The system will make use of unique identifiers.
- **Cursor support:** In cases where large amounts of data might be involved, for example with a User's environment, the use of cursors should be supported for paging.
- **Proxy support:** In cases where the retrieved information is large, and in particular where cursors may be used, light-weight proxies should be supported to retrieve list data and then the heavy-weight data can be retrieved on demand.

- **Multiple connections:** MCT is designed to be a distributed architecture, so the persistence mechanism should support multiple simultaneous connections.
- **Record support:** Most information isn't organized as single field/value pairs but multiple, semantically-related pairs. Whether or not stored in record form, applications generally need information in record form and this should be an aspect of the persistence mechanism.

Any summary?

Persistence Management Use Cases

Need more use cases for persistence management but the ones that have been identified to date are shown in Table 25:

Required Functionality	Use Cases?	Related Use Cases
PRST1: MCT shall provide a central Persistence Management subsystem for persisting component data (and caching) that provides session-level capabilities.	No	
PRST2: The central Persistence Management subsystem shall be configurable.	Yes	• PERST configure PERST
PRST3: The central Persistence Management subsystem shall be policy based.	Yes	• SYS operate on Object
PRST4: The central Persistence Management subsystem will be available to services and subsystems through the shared platform environment.	Yes	• SYS access PERST
PRST5: The central Persistence Management subsystem shall be able to persist to multiple formats.	No	
PRST6: The central Persistence Management subsystem shall be transparent to data management services.	No	
PRST7: The central Persistence Management subsystem shall support object caching.	No	
PRST8: The central Persistence Management subsystem shall support standard storage operations as defined in table PRST1 (e.g., get, put, update, delete).	Yes	• PERST get Object • PERST save Object • PERST update Object • PERST delete Object
PRST9: The central Persistence Management subsystem shall support mechanisms to query the persistence storage.	Yes	• PERST get Object with Query
PRST10: The central Persistence Management subsystem shall support cursors in the case of large data set retrieval.	Yes	• PERST get large Object
PRST11: The central Persistence Management subsystem shall persist model and representation components at the point of update.	Yes	• SYS modify Object
PRST12: The central Persistence Management subsystem shall persist all user object specific entities during creation: model mappings, action mappings, rules, and validations.	Yes	• PERST save Object
PRST13: The central Persistence Management	Yes	• UP restore MCT

subsystem shall support system state restoration.		
PRST14: The central Persistence Management subsystem shall support system state buffering.	No	
PRST15: User objects can be persisted via user interface controls (menus, right clicking, keyboard shortcuts).	Yes	• USER save entity with keyboard strokes

Table 25: Persistence Management use cases.

Persistence Management System Design

The following design has been adapted from that provided by Scott Ambler in his paper titled *The Design of a Robust Persistence Layer for Relational Databases*. In this paper Scott focuses on relational databases, what design approaches are feasible, what approach is recommended, and why, but the general design is adequate for any type of persisted data.

Given the design constraints and requirements discussed above, a suitable approach is presented in Figure 97:

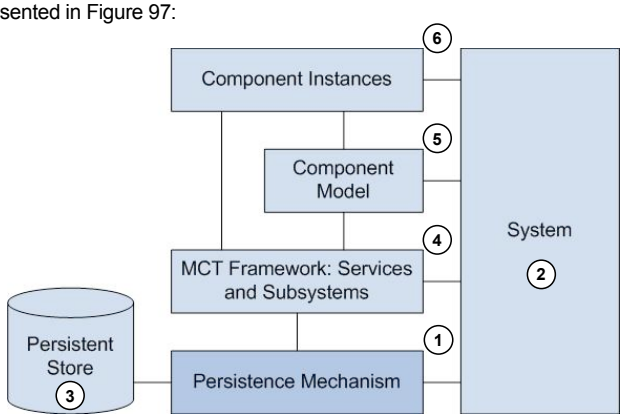


Figure 97: General persistence management design interactions.

In this figure, the persistence mechanism (at **1**) is encapsulated from the persistence store (at **3**), system (at **2**), framework (at **4**), component model implementation (at **5**), and component instances (at **6)**.

An architectural diagram illustrating the relationships necessary to implement the persistence layer in MCT is shown in Figure 98:

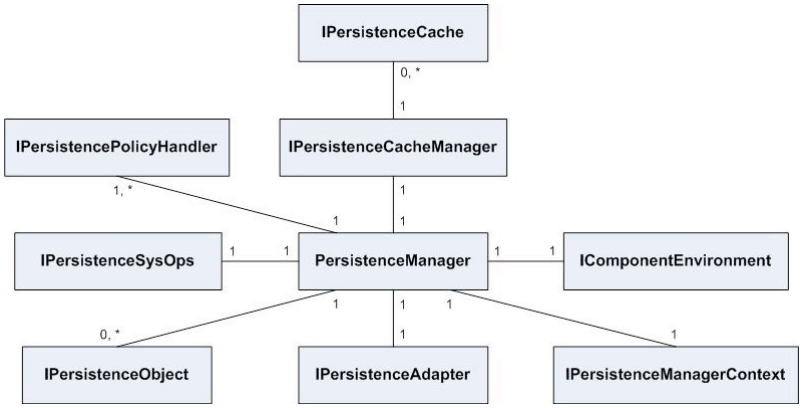


Figure 98: Persistence relationship diagram.

A design that implements the persistence mechanism depicted above is shown in Figure 99:

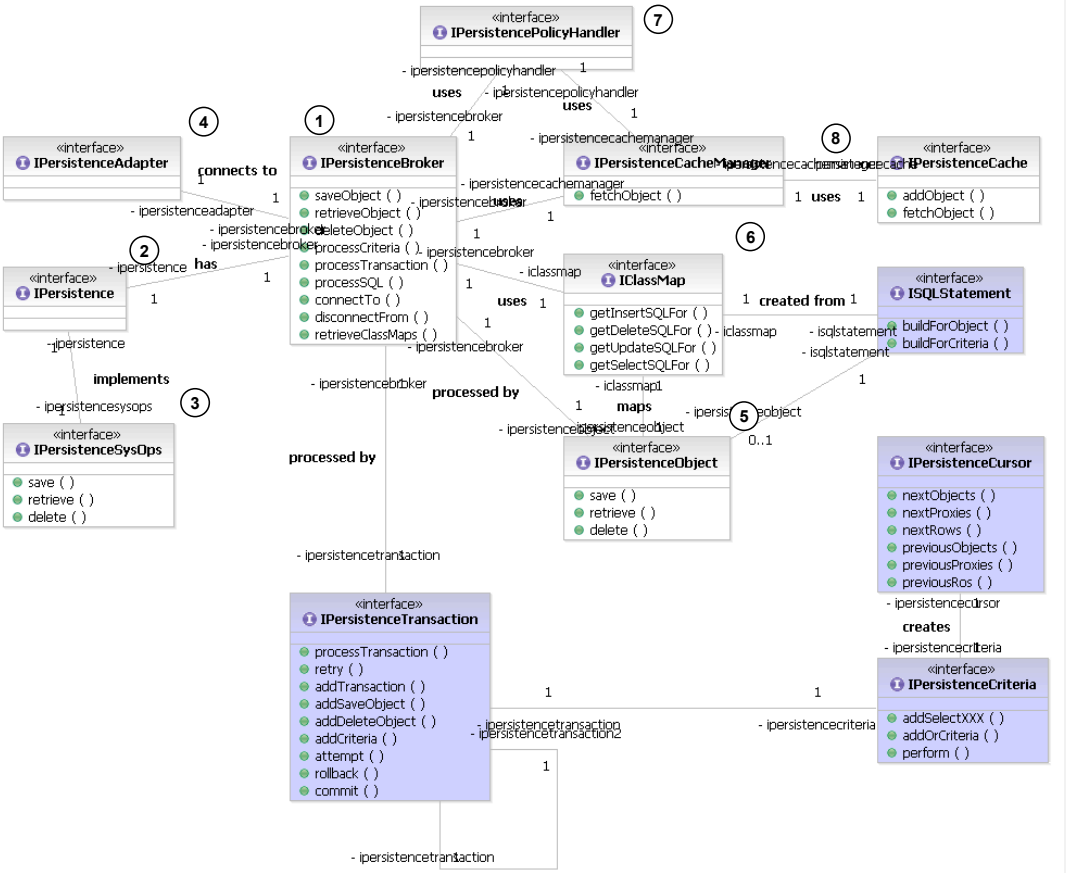


Figure 99: Persistence mechanism.

The design presented above is incomplete, but addresses most of the stated concerns for the persistence mechanism. The operational hub of the mechanism is the PersistenceBroker (at **1**), though it is the PersistenceObject (at **5**) which provides the interface to the MCT framework (through PersistenceSysOps, at **3**). The PersistenceBroker sets up the connection to the persistence store through a PersistenceAdapter (at **4**). The mechanism whereby components are persisted requires conversion of the component fields (etc.) to a form conducive for export. This is performed by the ClassMap classes (at **6**). When operations to retrieve or save are invoked on PersistentObjects the PersistencePolicyHandler (at 7) is invoked to determine whether to use the persistence mechanism or the cache mechanism (or both). If the latter, then the PersistenceCacheManager (at **8**) fetches the component value from the PersistenceCache.

Persistence Broker: Maintains connections to persistence mechanisms, such as flat files or relational databases. Handles the communication between the object application and

the persistence mechanisms. The Persistence Broker is managed by the general Persistence interface, processes Persistence Objects, and processes Persistence Transactions.

Persistent Object: This class encapsulates the behavior needed to make single component instances persistent and is the class that framework classes inherit from to become persistent. Persistent Object instances are processed by the Persistence Broker, map to instances of Class Map, and are related to SQL Statements.

Class Map: This is used to map classes and attributes to the repository. In the case of MCT, this is an important class that must parse the component structure and assign field names to the repository. Class Map has relations with Persistence Object, creates SQL Statements, and is used by the Persistence Broker.

Persistence Cache Manager: Manages access to the component cache. Interacts with Persistence manager to determine when and where to retrieve component values (or save them). The cache will have a configurable size. Once filled there will be a replacement policy.

Cache Management Policies

When considering how to manage persistence caching/buffering, the following five policies should be addressed:

- **LRU (Least Recently Used):** Discards the least recently used items first. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards the least recently used item. If a probabilistic scheme that almost always discards one of the least recently used items is sufficient, the Pseudo-LRU algorithm can be used which only needs one bit per cache item to work. For items that aren't in RAM the **LRU** (least recently used) policy is generally used. "*More efficient caches compute use frequency against the size of the stored contents, as well as the latencies and throughputs for both the cache and the backing store. While this works well for larger amounts of data, long latencies, and slow throughputs, such as experienced with a hard drive and the Internet, it's not efficient to use this for cached main memory (RAM).*"[7]
- **Write-Through:** Every time the cache is accessed the new value is persisted.
- **Write-Back:** Every time a value is expunged from the cache its value is persisted back to the main store. Keep track of items that have been changed with a 'dirty' flag and, when it is time to write back to the main store dirty items will be written. Write-back can be triggered other ways, among them manually.
- **Least Frequently Used:** Counts how often an item is needed. Those that are used least often are discarded first.
- **Adaptive Replacement Cache:** Improves on LRU by constantly balancing between recency and frequency.

In selecting a policy, cost should be considered. Items that are expensive to obtain (e.g., take a long time to retrieve) is an example. Size should also be considered. If items have different sizes, one may want to discard a large one to store several smaller ones. Finally, time should be considered. Some caches keep information that expires (e.g. a news cache, a DNS cache, or a web browser cache). The computer may discard items because

[7] Wikipedia under caching algorithms.

they are expired. Depending on the size of the cache no further caching algorithm to discard items may be necessary.

Persistence Policy Handler: Imposes defined policies on the Persistence Broker and Persistence Cache Manager.

Persistence Policy Types

Using a policy manager the persistence mechanism can be configured to persist different data objects according to their own policy. Five example persistence policies are identified below:

- **Immediate**: Every time an object is modified
- **Aggregate**: When semantic group changes, persist the group
- **Constant Value**: When some value is reached
- **Trigger**: When some event is created
- **Buffer Size**: Persist when the cache/buffer reaches some threshold (e.g., 10 MB)

The relationship between the PersistenceAdapter and the Persistence Implementation is illustrated in Figure 100:

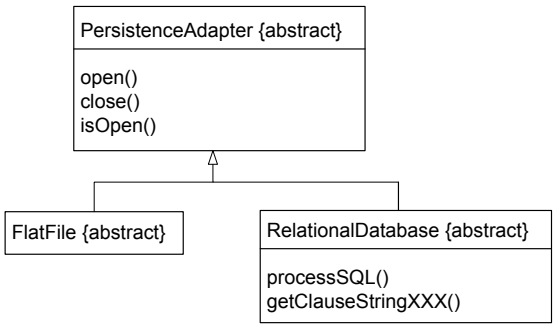


Figure 100: Persistence adapters and persistence implementations.

Class Map: This is used to map classes and attributes to the repository. In the case of MCT, this is an important class that must parse the component structure and assign field names to the repository. Class Map has relations with Persistence Object, creates SQL Statements, and is used by the Persistence Broker.

Persistent Transactions: This class encapsulates the behavior needed to support transactions, both flat and nested, in persistence mechanisms. For cases where there might be several, dependent, persistence operations, a set of transactions might be required. In this case a transaction policy (e.g., rollback unless commit on all succeeds). Presumably this would not be needed in a flat file persistence mechanism. The PersistenceTransaction class interacts directly with the PersistenceBroker and PersistenceCriteria classes (and is recusive).

Persistence Cursor: This class represents a mapping to a standard database cursor, for viewing collections of records rather than the entire collection. It provides the ability to page forward and backward and is configurable. Initially we will not need to implement this

class, but it will be needed earlier than, say, persistence criteria (but after relational support).

Persistence Criteria: A class hierarchy that encapsulates the access to flat files, relational databases, etc. For relational databases this hierarchy wraps complex class libraries, such as JDBC, protecting MCT from changes to the class libraries. This functionality is for saving, retrieving, or deleting several objects at once, and is used to create complex queries.

At first this can be stubbed if we do not want to perform complex persistence operations.

PersistenceCriteria interacts with PersistenceCursor and PersistenceTransaction.

Persistence Workflow

Object persistence is managed at the point of operation on components. When component values are required the persistence management system must be invoked to determine where to acquire the value from. When component values are modified the persistence management system must be invoked to store the value. Thus any action on a component potentially invokes the persistence management system, and the operation point is the single receiveMsg call since this is the only component access mechanism.

Chapter 16 Policy Management

Policy management means that any service or subsystem functionality can be tailored to satisfy specific runtime requirements/parameters. Unlike configuration management, which only applies to a system when it is initially launched, policies can be runtime dependent on the context of an operation's application.

Introduction to Policy Management

The runtime behavior of every MCT service and subsystem is determined by one or more policies. A policy is simply a set of conditions to match and a set of assignments to make when the conditions are met. Policies differ from configurations in that configurations are assignments that apply for the application session, whereas policies are applied dynamically. More than one [possible conflicting] policy can be enabled at once, thus complicating how they are created, managed, and executed.

Traditionally policies are effected through logic associated with an application's behavior. MCT decouples logic between the framework and application layers, and the application layer is described declaratively, so policies need to be managed and be described declaratively, to fit into this paradigm. Also, due to the ubiquitous nature of policies, policy management, and policy handling, this functionality has been made a centralized mechanism of the MCT Framework.

Like every other MCT functionality, the policy management system must be configurable. This means that, for any particular service or subsystem, the type and breadth of the policy must be configurable.

Examples of policies include what level of malleability to apply to components, when/how to persist components, when/how to use local store vs full retrieval, what type of cache management algorithm to use, what rule selection strategy to use, what rule execution strategy to use, or how much data to cache before persisting. Clearly each of these applies differently in different contexts, so the behavior is dynamic in nature.

Constraints to the Policy Management Design

There are 6 drivers in selecting a policy management approach:

- **Language**: Being a declarative description, a language must be selected that is suitably flexible that it can be used to describe any service or subsystem and the logic needed to describe how the policy works
- **Types/Levels**: Although fine-grained policies provide greater control they come at a huge cost in performance. As a result, it is best to use an approach where policies can be applied uniformly for a group of individuals in the same manner. Thus the policy is defined for a level and any group that satisfies that definition gets the same policy applied.
- **Interpretation**: As a declarative approach, there must be a standardized manner for reading and parsing policy descriptions into the run-time environment.
- **Management**: Policies should be managed by MCT as distinct with the service or subsystem they relate to, so that when it comes time to select a policy fewer policies can be reviewed.
- **Disambiguation**: As a central approach, and since multiple policies might be in effect at any given time, a mechanism is needed that can disambiguate which policies are active and how to apply them. This may involve selecting among conflicting or competing policies, recognizing that one policy subsumes another, or recognizing that two policies are compatible with one another (e.g., they operate on different attributes and have no dependencies on one another).

- **Handlers**: Once a set of policies have been evaluated based on a context service or subsystem specific handles must apply the policy in the same manner that the configuration mechanism would; attribute values are assigned and the originally-requested operation is performed.

In essence, a policy management system is a general filter through which all operations are funneled. A single mechanism is used to disambiguate policies and a flexible language is used to describe the policies so that they can be applied across the variety of contexts.

Policy Management Requirements and Use Cases

The following 7 use cases have been identified with 13 policy management requirements, as shown in Table 26:

Required Functionality	Use Cases?	Related Use Cases
PLCY1: The MCT Framework will provide a central policy management system for reading, validating, disambiguating, and assigning service and subsystem attributes at run time.	No	
PLCY2: The central Policy Management subsystem shall be configurable.	Yes	• POLCY is configurable
PLCY3: The central Policy Management subsystem shall be policy based.	Yes	• SYS apply operation
PLCY4: The central Policy Management subsystem will be available to services and subsystems through the shared platform environment.	Yes	• SYS access POLCY
PLCY5: The central Policy Management subsystem shall support policy levels (application, mission, role, component) for services and subsystems.	No	
PLCY6: Policies will use a declarative language.	No	
PLCY7: Each service and subsystem will be responsible for defining its own policies that satisfy the policy language.	No	
PLCY8: Policy files may be stored externally to MCT.	No	
PLCY9: Service and subsystem policy files will be validated.	Yes	• POLCY load policies
PLCY10: The central Policy Management subsystem will store policies by service and subsystem.	Yes	• POLCY store policies
PLCY11: The central Policy Management subsystem shall be context sensitive (component, operation type, system bindings, etc.).	Yes	• POLCY store policies
PLCY12: The central Policy Management subsystem shall provide a common mechanism for disambiguating, ordering, and selecting policies.	Yes	• POLCY disambiguate, order, select policy for COMP, ACT
PLCY13: Each service or subsystem will provide its own policy handlers.	Yes	• SYS handle policy
PLCY14: Component service and subsystem	Yes	• SYS select policy

attributes and behaviors can select policy level control (e.g., persistence is immediate).	enforcement type

Table 26: Policy Management requirements and use cases.

General Policy Management Design

Policy Management Approach

Policy management is a process of applying logic at runtime to match conditions for an operation. This enables the framework to be very versatile because different logics can be applied to different contexts. The drawback is that to do this in enterprise code makes the associated code very brittle, difficult to maintain, and difficult to extend. An approach that can decouple the application of logic in this manner without the associated drawbacks is to use a rule engine, which will cycle through all available policy rules and identify which one to apply as it matches the rules to the operational context (i.e., the operation operand, type, and arguments). The drawback to using a rule engine is that as the number of rules increases, or the rule complexity increases, performance suffers).

MCT already makes use of a rule engine in other capacities, so using a rule engine in the context of policy management is a reasonable choice. It remains to be seen whether it will be an appropriate choice, but the architecture should be set up in such a way (if possible) to allow another engine to be used instead of a rule engine.

Policy Language

A policy language must apply equally well to any MCT service or subsystem, and it must allow the description of general logic operations (e.g., inequalities, algebra). The RuleML language has been defined for the purpose of describing inter-agent behavior and is capable of describing general multivariate logics. Moreover, RuleML is already being used elsewhere in the MCT framework. Thus it makes a good choice for describing policies.

Policy Levels

Policy management can be a costly operation because it can be so finely grained that it is applied, potentially, to every operation on every service or subsystem. As such, policy application should be configurable at different levels of applicability. The MCT policy management approach defines 5 policy application levels:

- **System**: These are policies that are applied at the system level.
- **Application**: These are policies that are applied for a particular application.
- **Role or Group**: These are policies that are applied to a particular component role or identity group.
- **Component**: These are policies that are applied to specific components.
- **None**: These are policies that aren't applied to any components. They are noops.

It is unlikely that policy management at the component level will be practical, but the system is being designed to accommodate this operational level so that it can be tested one way or another.

Policy Scope

Policies apply to a particular semantic group or cluster. This enables them to be loaded or unloaded in smaller groups to improve performance. The semantic clustering is generally associated with information types, such as telemetry groups. When an operation is identified with a particular item in a group, the policies related to the group and operation are loaded and disambiguated.

Policy Representation

Competing Policies – Conflict Resolution

Policy Management Workflow

Policy Management System Design

Given the design constraints and requirements discussed above, a suitable approach is the use of an instance of the rule engine presented earlier and reproduced in Figure 101:

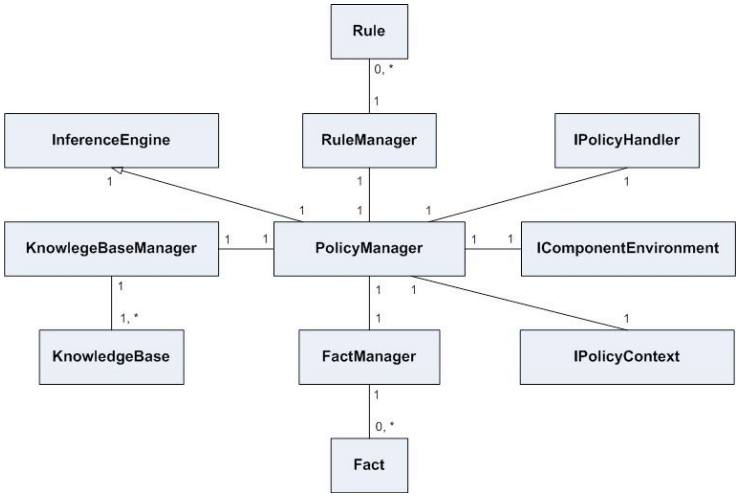


Figure 101: Policy management relationship diagram.

Policy Workflow

Policies are invoked at the point of operation application on a component, much the same as persistence, except that policies can apply to any service or subsystem while persistence is only applied to components. In terms of component applicability, policies are applied at the receiveMsg call in the same manner as the persistence management system. They also apply prior to persistence since they might apply to whether an object is persisted. In terms of subsystems, the implementation of policy management must be controlled internally by the policy handler.

Chapter 17 External Services

MCT can make use of any number of 3rd-party applications. In most cases these applications are providers of data and MCT is a consumer of the data. The external services subsystem is responsible for providing a transparent, integrated environment for working with 3rd-party applications. Not only must it provide appropriate mechanisms for setting up and configuring a service, but it must provide the means to acquire the metadata associated with a service. All of this must be performed in a manner that renders it transparent to the MCT framework.

Introduction to External Services

MCT is a data-oriented as much as it is an information-oriented architecture. Large amounts of data from disparate sources must be displayed and interacted with in complex ways. It is essential to the flexibility and performance of MCT applications that external data sources and applications be provided to the MCT framework in a unified and transparent manner. The external services subsystem is responsible for providing this interface.

Generally speaking external services can be bi-directional. In most cases the 3rd-party application is a provider of data and information about it, and MCT is the consumer. In some cases it is the other way around, and MCT is the provider of data and information about it, and the 3rd-party application is the consumer. In essence, the External Services layer must provide both client and server capabilities.

It is also unknown in advance what communications protocols will be used by the 3rd-party applications. In theory they could use anything (e.g., direct socket connections, CORBA, JMS, FTP, HTTP, SOAP, XMLRPC, etc.). In addition the transport format used by the 3rd-party application could vary widely (e.g., XML, RDF, text, binary, streams, HTML, etc.). Finally, along with the 3rd-party application will come metadata associated with the application and the data stream, and this information must be conveyed to MCT. In essence the External Services layer is a big sieve using service and format adapters. What comes in may be highly diverse but what gets in must be highly coherent.

Constraints on the External Services Subsystem Design

The External Services subsystem is informed by several externally-defined constraints:

- Command and Control format: NASA command and control is currently based on a set of ASCII files called 'standard out'. It is anticipated that this format will be converted to the XTCE XML/XML Schema format in Constellation.
- The CSI communications backbone communicates using CSI data exchange packets which are a combination of binary and XTCE content.
- The NASA telemetry service ISP uses a push technology to send blocks of id/value pairs. Both push and pull technologies should thus be supported.
- Any number of formats should thus be supported.
- Different technologies will use synchronous or asynchronous communications, so both should be supported.

External Services Requirements and Use Cases

The requirements that have been identified with the External Services functionality, to data, are shown in Table 27:

Required Functionality	Use Cases?	Related Use Cases
ES1: MCT shall provide an External Services subsystem that will integrate 3rd party data sources to MCT componentry.	No	
ES2: The External Services	Yes	• ES configure ES

subsystem shall be configurable.		
ES3: The External Services subsystem shall be policy based.	Yes	• ES handlePolicy on COMP
ES4: The External Services subsystem shall be available to services and subsystems through the shared platform environment.	Yes	• SYS access ES
ES5: The External Services subsystem shall isolate 3rd party sources through a common API.	No	
ES6: The External Services subsystem shall provide the means to discover and access 3rd party source metadata.	Yes	• ES discovers EXT metadata • ES publishes EXT metadata
ES7: The External Services subsystem shall provide 3rd parties the ability to discover component services.	Yes	• EXT ask MCT for service description
ES8: The External Services subsystem shall provide component export services.	Yes	• ES export COMP
ES9: The External Services subsystem shall offer a set of metadata attributes and behaviors that are applicable across all wrapped applications. This metadata shall conform to the semantic description language used by the information semantics manager.	Yes	• ES asks for EXT metadata • SYS asks ES for EXT metadata
ES10: The External Services subsystem shall provide a loose coupling between a service adapter and an associated Model Role attribute.	No	
ES11: Service adapters shall enforce the secure interaction with external applications in cooperation with the identity management and security subsystems.	Yes	**ES6:** ES discovers EXT metadata **Description:** The External Services subsystem shall provide the means to discover and access 3rd party source metadata. **Scope:** ES **Primary Actor:** ES **Stakeholders:** EXT **Preconditions:** ES initialized, EXT is started **Triggers:** ES requests EXT metadata **Postconditions:** ES has EXT metadata **Success Scenario:** ▪ ES requests EXT metadata

	▪ ES has EXT metadata **Failure Scenarios:** EXT has no metadata, EXT has no API for providing metadata **ES6:** ES publishes EXT metadata **Description:** The External Services subsystem shall provide the means to discover and access 3rd party source metadata. **Scope:** ES **Primary Actor:** ES **Stakeholders:** EXT, UP, SYS **Preconditions:** ES, UP, and SYS initialized, EXT is started **Triggers:** SYS wants access to EXT metadata **Postconditions:** SYS has access to EXT metadata **Success Scenario:** ▪ SYS wants to access EXT metadata ▪ SYS invoked ES request for EXT metadata through ES context ▪ ES returns EXT metadata to SYS ▪ SYS has access to EXT metadata **Failure Scenarios:** ES doesn't have access to EXT metadata, EXT has no metadata **ES7:** EXT ask MCT for service description **Description:** The External Services subsystem shall provide 3rd parties the ability to discover component services. **Scope:** ES **Primary Actor:** ES **Stakeholders:** MCT, POLICY **Preconditions:** UP initialized **Triggers:** EXT queries ES for MCT services

	Postconditions: MCT provides service description **Success Scenario:** ▪ ES queries ES for service description ▪ MCT checks POLICY for query ▪ MCT provides service description **Failure Scenarios:** **ES8:** ES export COMP **Description:** The External Services subsystem shall provide component export services. **Scope:** ES **Primary Actor:** ES **Stakeholders:** POLICY, COMP **Preconditions:** UP initialized. EXT configured, loaded, and started **Triggers:** EXT queries for COMP **Postconditions:** ENV returns response to query **Success Scenario:** ▪ ENV uses ID component context to query for User information from user component **Failure Scenarios:** **ES9:** ES asks for EXT metadata **Description:** The External Services subsystem shall offer a set of metadata attributes and behaviors that are applicable across all wrapped applications. This metadata shall conform to the semantic description language used by the information semantics manager. **Scope:** ES **Primary Actor:** ES **Stakeholders:** POLICY,

		COMP **Preconditions:** UP initialized. EXT configured, loaded, and started **Triggers:** EXT queries for COMP **Postconditions:** ENV returns response to query **Success Scenario:** ▪ ENV uses ID component context to query for User information from user component **Failure Scenarios:** **ES9:** SYS asks ES for EXT metadata **Description:** The External Services subsystem shall offer a set of metadata attributes and behaviors that are applicable across all wrapped applications. This metadata shall conform to the semantic description language used by the information semantics manager. **Scope:** ES **Primary Actor:** ES **Stakeholders:** POLICY, COMP **Preconditions:** UP initialized. EXT configured, loaded, and started **Triggers:** EXT queries for COMP **Postconditions:** ENV returns response to query **Success Scenario:** ▪ ENV uses ID component context to query for User information from user component **Failure Scenarios:** • ES ask EXT for QoS • ES provide QoS • ES write data to COMP
ES12: External application service adapters shall hide the network protocol used to connect to the application from components using	Yes	

adapters.		
ES13: The External Services subsystem shall provide a common management point for all 3rd party services.		• ES load EXT • ES start EXT • ES stop EXT • ES unload EXT • ES get EXT
ES14: The External Services subsystem shall configure 3rd party services using messaging to work with the Messaging subsystem.	Part of ES2	
ES15: The External Services subsystem shall support synchronous and asynchronous communication between a component and the application it wraps. Asynchronous components will be threaded?	Yes	• EXT get data from SERVICE • EXT provide data to SERVICE • EXT get data from COMP
ES16: The External Services subsystem shall support push and pull external applications.	Yes	• EXT subscribe data • EXT publish data
ES17: It shall be possible to query a service adapter for the status of a pending asynchronous request.	Yes	• ES get EXT status
ES18: The External Services subsystem shall include the implementation of a mechanism for batching requests according to a parameterizable policy.	Yes	• ES batches requests
ES19: A batching policy shall exist that allows external application communications with a component to be scheduled at specific instances in time	Yes	• ES schedules batched requests
ES20: A policy shall exist that makes possible the batching of component communications with its wrapped application according to the number of queued requests.	Yes	• ES adjusts batches via POLICY
ES21: It shall be possible for the External Services subsystem to notify the User Platform of any changes in state.	Yes	• ES notifies UP of state changes

Table 27: External Services requirements and use cases.

General External Services Subsystem Design

The general approach to the ExternalServices subsystem design is shown in Figure 102:

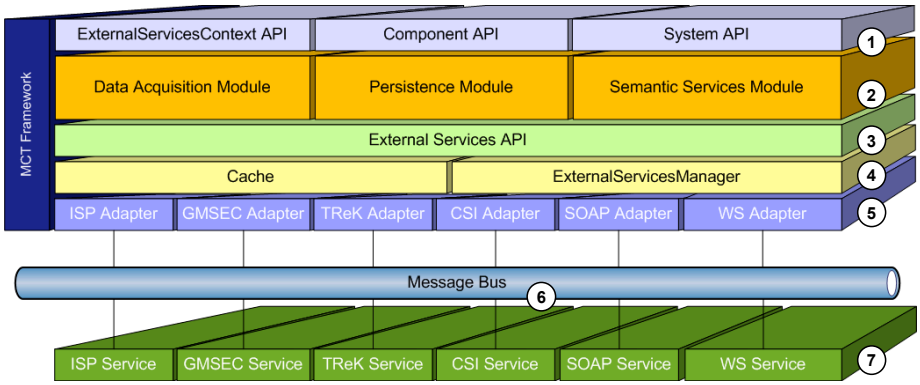


Figure 102: ExternalServices architecture.

The approach provided here is to create a subsystem that can communicate with external services, either to access data or to provide data. For all of these capabilities, APIs are provided to the MCT framework in the form of subsystem, component, or system APIs (at **1**). The functional modules are the concrete classes that implement the messaging aspects associated with the respective functionality: data acquisition, persistence, or semantic services (at **2**). Each of these service types implements a common external services API (at **3**). Information that is acquired through an external service is cached locally, and all external services are managed using the ExternalServicesManager (at **4**). Finally, any number of adapters translate services to the external services API (at **5**). These adapters make use of the common message bus (at **6**) that provides access to the data services themselves (at **7**).

Metadata Support

An important aspect of the ExternalServices API is that the individual characteristics of a particular service are lost when it becomes part of the MCT framework. As a result, metadata associated with the service must be encapsulated in the ExternalServicesManager so that traffic can be appropriately routed and so that the service can be kept up to date. This mechanism is similar to how the Information Semantics Manager works, and the structure is being kept as similar as possible to the ISM in case the two subsystems are merged at some point.

ExternalServices APIs

All external services use a common API in MCT, with different services walled off from MCT by service adapters that comply with the service API on one side and with the MCT API on the other. The general external services subsystem is shown in Figure 103:

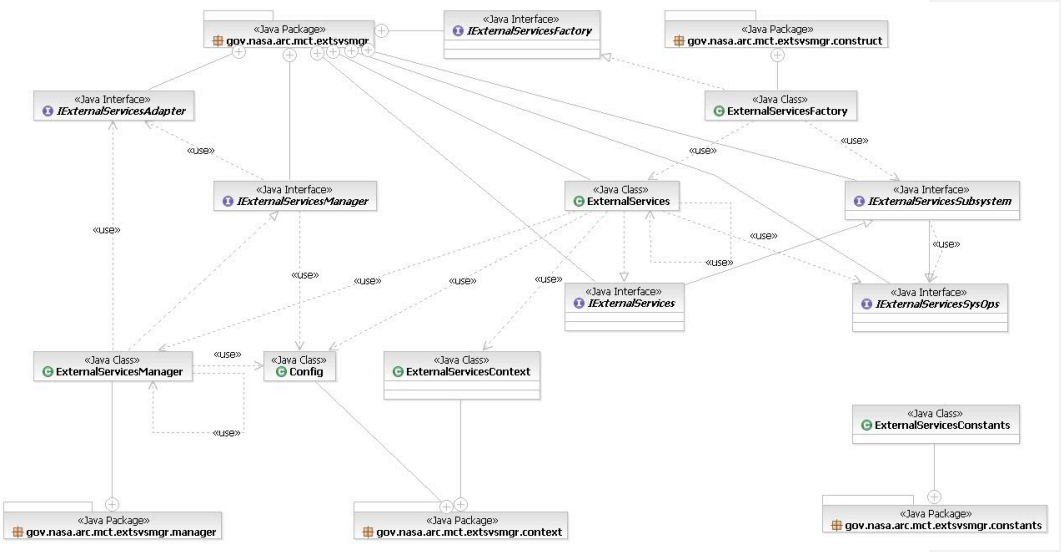


Figure 103: External Services subsystem architecture.

The External Services interfaces are further expanded in Figure 104:

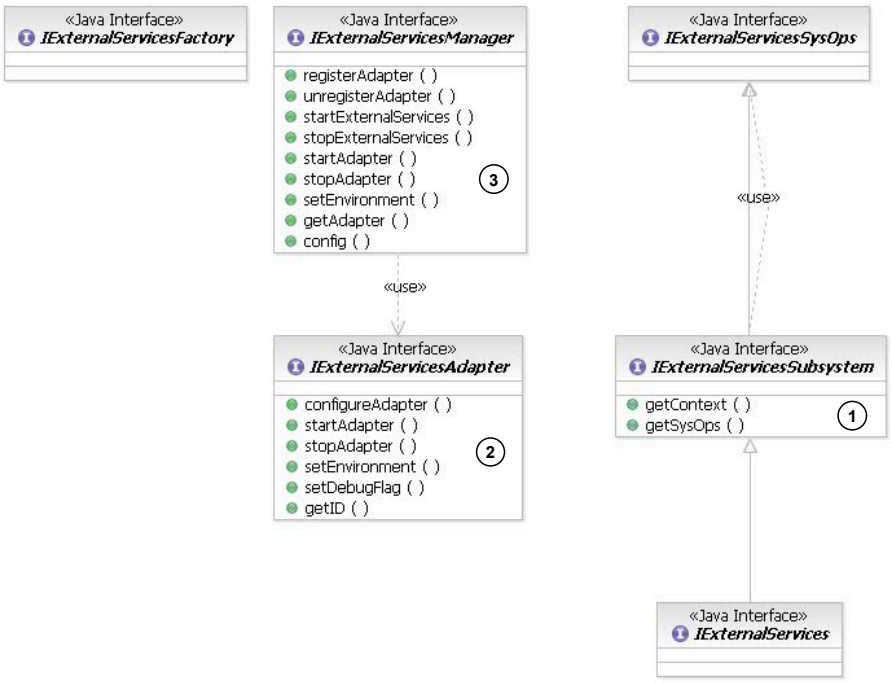


Figure 104: External Sevices subsystem interfaces.

Dsd

In addition to those specified above, individual service adapters will need to implement the following operational capability:

- getData(IDataId id) – get data with given Id (synchronous)
- getStatus(IDataId id) – get status with given Id (synchronous)
- getMessage(IDataId id) – get message with given Id (synchronous)
- getError() – get error information (synchronous)
- getState() – get state information (synchronous)
- receiveData(IDataId id) - get data with given ID (asynchronous callback)
- receiveStatus(IDataId id) - get status with given ID (asynchronous callback)
- receiveMessage(IDataId id) - get message with given ID (asynchronous callback)
- receiveError() - get error information (asynchronous callback)
- receiveState() - get state information (asynchronous callback)
- sendData(Object) – send data to the external service
- sendData(Vector<Object>) – send a vector of data to the external service

- configureData(Vector<Object>) – configure the external service with the given data

Adaptor States

External Services adapters can take on states that can be monitored to provide information about how they are working. The state family describes the preconditions, transition events, and post conditions for adaptor states. To traverse into a subsequent state, all actions must complete successfully, otherwise the adaptor remains in its current state. A state diagram representing the states of External Services adapters is presented in Figure 105:

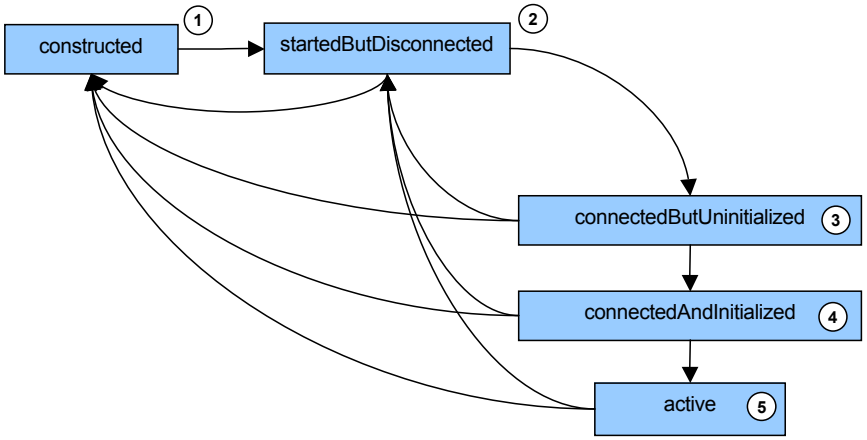


Figure 105: External Services adapter state diagram.

Five states are represented for each adapter in this system: constructed (at **1**), startedButDisconnected (at **2**), connectedButUninitialized (at **3**), connectedAndInitialized (at **4**), and active (at **5**). The normal state transitions are the ones the move from left to right or top to bottom. Abnormal transitions move from right to left or bottom to top.

The table describing the state transitions associated with this diagram is presented as

Initial State	Transition Event	Post State
constructed	ES starts adapter	startedButDisconnected (1)
startedButDisconnected	ES issues connect to adapter	connectedButUninitialized
connectedButUnitialized	ES initializes adapter	connectedAndInitialized
connectedAndInitialized	ES requests data	active

active	ES requests stop data	connectedAndInitialized	
connectedButUninitialized	Network disconnect event	startedButDisconnected	2
connectedAndInitialized	Network disconnect event	startedButDisconnected	2
active	Network disconnect event	startedButDisconnected	2
startedButDisconnected	ES issues stop to adapter	constructed	3
connectedButUninitialized	ES issues stop to adapter	constructed	3
connectedAndInitialized	ES issues stop to adapter	constructed	3
active	ES issues stop to adapter	constructed	3

Table 28: External Services adapter transition table.

When the External Services (ES) subsystem starts an adapter (at **1**), adapter configurations are applied. In any case where a network disconnect occurs (at **2**) it is considered an error. In any case where the ES issues a stop action on the adapter (at **3**) it is also considered an error unless the system is undergoing a normal shutdown.

ISP Telemetry Adapter

At present real-time telemetry data is acquired using the ISP service using the ISPresso java interface. The API is presented in Figure 106:

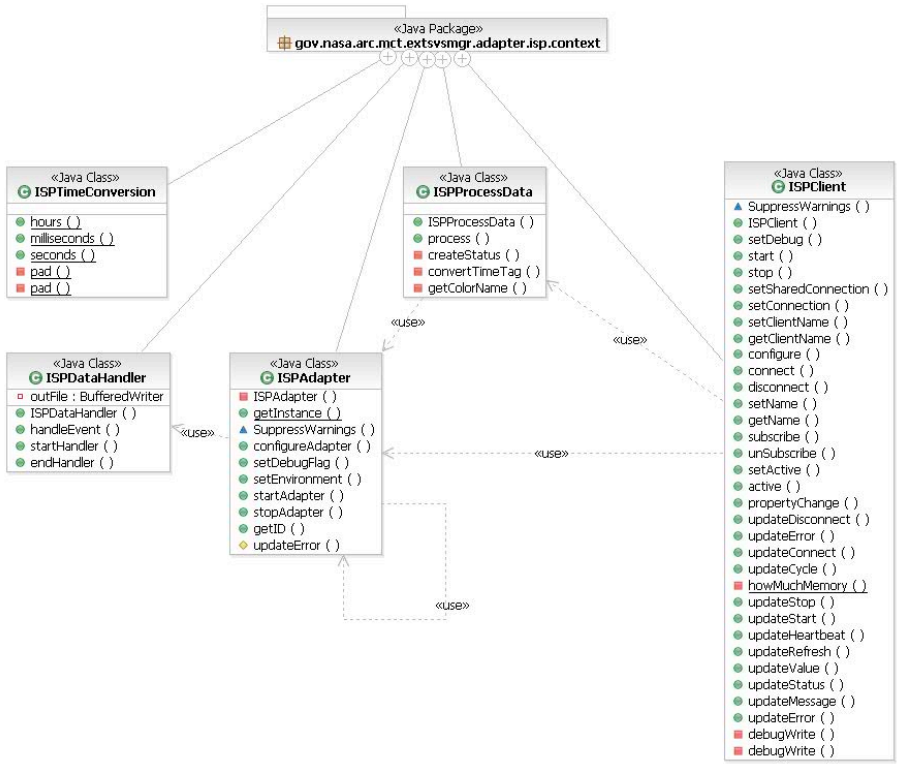


Figure 106: MCT ISP adapter and related architecture.

Data Model, Component, Representation Binding

An important aspect of any application framework is the mechanism whereby visual widgetry is bound to values, updated, manipulated, and validated. In a component model approach it is essential that the data acquisition and the data binding be autonomous from the user interface components used to render it. The MCT approach to resolving this issue is shown in Figure 107:

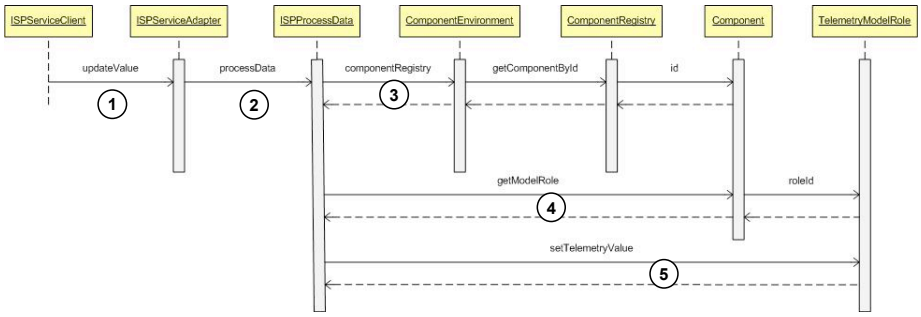


Figure 107: Data binding in MCT.

The ExternalServices subsystem has many service adapters that take the form such as the ISPServiceAdapter that connects to an ISPServiceClient. Since ISP is a push service, an updateValue operation (at **1**) is applied to the service adapter. This in turn results in a call to processData in the ISPProcessData class (at **2**). This class interacts with the ComponentEnvironment to get the ComponentRegistry and retrieve the appropriate Component(s) (at **3**). Once the component is acquired its Model role instance is retrieved (at **4**), whereupon the value assignment can take place (at **5**). The value takes the form, in the case of telemetry, of a TelemetrySamplePoint, which is a structured object. GUI instances are bound to attributes of the Model role instance and thus update when it changes.

Chapter 18 Localization

Localization is the mechanism of selecting and displaying the correct locale, font, and translation for a particular audience. Often the administration of a localization policy is not a one size fits all matter. For example, distributed data sources may already encoded in a particular way and cannot be controlled from the current application. In this case a mixed encoding may be the best one can achieve. In cases where multiple encodings are displayed it is imperative to display them correctly.

Introduction to Localization and Internationalization

MCT is not currently designed to accommodate language or ADA compliance. The discussion below is intended to address localization issues in MCT.

Generally speaking a localization approach is necessary for any application in which people from different countries or cultures might be using the application and especially in cases where those applications are displayed in local fonts other than roman. Although this is generally more an issue for commercial software where deployment could be in any number of countries, it also applies to applications which are broadly deployed but are under central (or distributed) control.

Localization and Internationalization

Currently MCT localization strings are implemented on the client. Java applications generally support Unicode double-byte encoding at the interface, and UTF-8 encoding at the interfaces. Since this has become a standard in localization and encoding it is recommended that this approach be applied in MCT.

The bulk of internationalization rests with the structure of strings such as dates and times, and symbols such as units symbols, which aren't themselves the string content. Both of these problems must be addressed by the MCT approach. Those items, such as timezones, dates, and times are handled through existing Java libraries. Currencies and addresses are currently missing in MCT applications so they do not need to be handled at present.

General Localization Approach

There are several approaches that could be used to localize MCT applications. Eclipse provides support for localizing strings but assumes that those strings are hardcoded into the java code, which is not the case in MCT, so this approach might not work. Another approach is to map incoming strings through a java resource bundle as the objects are being populated. This approach is transparent to the source of the strings and is quite general but requires that the representation code be modified to reference the resource bundle at the time a field is populated with the data value.

Another issue is where to store string translations. Since the strings being used in MCT applications might be used in different applications or even different contexts within the NASA family, it is reasonable to construct a central and searchable string translations repository that can be employed at build time to produce a resource bundle appropriate for a particular application. This application will be discussed in greater detail in a separate document. It may even be appropriate to install these strings along with related ontological information in the information model.

String Translation

There is a single class associated directly with localization translation in MCT: LocalizationResource. There is an associated ResourceManager whose primary purpose is to select a resource based on the selected locale, and then to translate from the string key to the localized string value. The resources themselves are in files named: MCTString_[lang]_[spec], where "lang" is a locale such as "en" and "spec" is a locale specialization such as "us". These files are produced at build time.

LocalizationResource is initialized in UserPlatform. There are two ways that LocalizationResource can be applied. One is during component parsing, in which case the workflow looks as shown in Figure 108:

```
UserPlatform:

  LocalizationResource mLocalizationResource;

Initialize Component:

  NamedNodeMap nnp = node.getAttributes();
  Node         nd  = nnp.getNamedItem("text");

  mLocalizationResource = LocalizationResource.getInstance();

  if (nd != null)
    comp.setText(mLocalizationResource.getString(nd.getNodeValue().trim()));

  nd = nnp.getNamedItem("name");

  if (nd != null)
    comp.setName(mLocalizationResource.getString(nd.getNodeValue().trim()));
```

Figure 108: String localization construction in MCT.

The example provided above illustrates the parsing of the content for a component (comp) from XML. Since MCT uses an information model this would have to be adapted to parse from another language such as RDF. In this case, the UserPlatform instantiates the LocalizationResource object (a singleton). The individual components access this object and set their string components to the localized versions based on the search key (the nominal English value). When the component is rendered, the localized value is displayed.

An alternative approach is to use the LocalizationResource in the RepresentationComponent directly. In this case the code looks as shown in Figure 109:

```
UserPlatform:

  LocalizationResource mLocalizationResource;

Initialize Component:

  NamedNodeMap nnp = node.getAttributes();
  Node         nd  = nnp.getNamedItem("text");

  mLocalizationResource = LocalizationResource.getInstance();

  if (nd != null)
    comp.setText(mLocalizationResource.getString(nd.getNodeValue().trim()));

  nd = nnp.getNamedItem("name");

  if (nd != null)
    comp.setName(mLocalizationResource.getString(nd.getNodeValue().trim()));
```

Figure 109: String localization in MCT representation components.

Chapter 19 Packaging and Deployment

Packaging and deployment refer to how MCT is made available in release form.

Introduction to Packaging and Deployment

Copy from Dennis' document for now. Should we include documentation on the build procedures as well? Probably.

- What is the function of this framework component
- Why is this framework component needed
- What are the constraints and requirements that inform this framework component's design
- Where does this framework component fit into the larger MCT functional picture
- How flexible/autonomous must this framework component be
- What design approaches are feasible, what approach is recommended, and why
- What use cases must be supported by this framework component
- General workflow for this framework component
- Framework component design overview and block diagram
- Appropriate UML to enable development (class diagrams, state diagrams, sequence diagrams, etc.)

Packaging and Deployment Use Cases

The packaging and deployment of MCT has been identified with the following 10 use cases:

Required Functionality	Use Cases?	Related Use Cases
DPLY1: A process and corresponding integration mechanism shall exist to migrate existing standard Eclipse applications to a fully compatible set of MCT components offering the same application functionality, provided the Eclipse Application Migration Requirements are met		
DPLY2: The suite of tools supporting MCT component development and integration shall be bundled as an Eclipse product and released as features using the Update Manager.		
DPLY3: The suite of tools supporting the runtime administration of the MCT system shall be bundled as an Eclipse product and released as features using the Update Manager.		
DPLY4: The MCT core components shall be bundled as an Eclipse product and released as features using the Update Manager.		
DPLY5: An Eclipse update site shall be maintained to publish all MCT products.		
DPLY6: The core set of MCT system frameworks shall be bundled as an Eclipse product and		

released as features using the Update Manager.	
DPLY7: The MCT SDK shall aggregate and offer the MCT Development Tool Suite, Core Components, and Frameworks plug-ins as one product.	
DPLY8: A process and corresponding integration mechanism shall exist to migrate existing standard Eclipse applications to a fully compatible set of MCT components offering the same application functionality, provided the Eclipse Application Migration Requirements are met	
DPLY9: The suite of tools supporting MCT component development and integration shall be bundled as an Eclipse product and released as features using the Update Manager.	
DPLY10: The suite of tools supporting the runtime administration of the MCT system shall be bundled as an Eclipse product and released as features using the Update Manager.	

Table 29: Packaging and Deployment use cases.

Appendix A Framework Use Cases

In the context of MCT a use case represents a contract between entities about an action to be performed and generally how it will be performed. As a contract, it can be used to determine whether software designed and implemented satisfies requirements, so use cases are often used as a measure of software requirements compliance.

A use case has five primary constituents: (1) actors, (2) an action to perform, (3) preconditions, (4) postconditions, and (5) scenarios. Actors represent the entities that are involved in the action. If a user is involved with an action the user is always an actor. The action is what task is being performed. The preconditions are those states on the actors that must be met before the action can be performed. The postconditions are those states on the actors that are assured if the action is performed. The preconditions and postconditions are always defined for the primary, or success, scenario. The scenarios describe the sequence of events leading from the preconditions to the postconditions under various circumstances. There is always a primary or success scenario which is the expected behavior. There can be any number of alternate or failure scenarios.

Combined, the scenarios for a particular use case completely define the behavior associated with an action, and they describe action sequences at a high enough level that they can be used to inform the architectural design of packages and classes. In these respects they are an excellent design tool.

MCT Actors

The sections below articulate in greater depth the use cases defined for the various MCT services and subsystems. Significant to this exercise is the definition of actors. There is a near 1:1 mapping between services/subsystems and actors, but there are often locally-defined stakeholder within use cases that describe items that are more 'atomic' than systems, such as the component being acted upon or the action being performed. In the context of this document, actors will always be capitalized because it is easier to read what is an actor if it stands out.

- **COMP**: Component model
- **UP:** User platform system
- **RE:** Rule engine system
- **VALID:** Validation service
- **POLCY:** Policy management service
- **PERST:** Persistence management service
- **ISM:** Information semantics management system
- **HANDL:** Event handler system
- **ES:** External services system
- **EXT:** External service
- **ID:** Identify management system

- **CONFIG:** Configuration management system
- **REG:** Component registry service
- **UIMGR:** User interface manager
- **ENV:** Environment
- **CLIB:** Component library
- **COM:** Communications/messaging system
- **ONT:** Triple store
- **ONTQ:** Ontology query engine
- **PMS:** Persistence management store
- **SYS:** An arbitrary MCT system
- **BE:** Behavior entity
- **ENTITY:** Any agent with privileges that can interact with a component
- **DT:** Design tool
- **UIT:** UI Toolkit
- **USER:** MCT user

MCT Use Case Structure

The sections below articulate in greater depth the use cases defined for the various MCT services and subsystems. The basic structure of the use case has been expanded to include some additional attributes:

- **Description:** This is a prose description of the requirement.
- **Scope:** The primary system 'hosting' the action.
- **Primary Actor:** Generally, the initiating entity.
- **Stakeholders:** Participating entities, mostly systems.
- **Preconditions:** States that must be met on participants to enable the action.
- **Trigger:** The event that is the catalyst for the action to take place.
- **Postconditions:** Participant states that are guaranteed after the action is performed in the success scenario.
- **Primary/Success Scenario:** The sequence of behaviors leading from preconditions to postconditions for the expected action.
- **Secondary/Failure Scenarios:** The sequence of behaviors leading from alternate preconditions to alternate postconditions for action 'failures'.

It should be clear that the scope, primary actor, and stakeholders simply clarify roles in the previous actors descriptor, and that trigger is simply clarifying between preconditions and the first behavior of the action sequence, so together this set of descriptors allows those who wish to read use cases in a bit more detail can understand them without reading the sequences themselves. To that end the increased number of descriptors has value.

MCT Framework Use Cases

A total of 223 use cases have so far been identified for the services and systems that comprise MCT. These are defined for the 266 requirements. There is not a 1:1 mapping between use cases and requirements. In many cases there are several use cases per requirement, and in many cases there are no use cases for a requirement.

Subsystem or Mechanism	Actor Name	Use Cases	Requirements
Component Model	COMP	24	25
UI Toolkit	UIT	61	25
Component Library	CL	2	2
Information Semantics Manager	ISM	7	17
User Platform	UP	11	27
Configuration Manager	CONFIG	5	9
Event Handler	EH	16	19
Identity Manager	ID	11	19
Rule Engine	RE	24	22
Composition	CMPS	7	10
Constraint Satisfaction	CONST	0	0
Validation	VALID	9	12
Messaging	COM	12	20
Persistence Manager	PERST	12	14
Policy Manager	POLCY	8	23
External Services Manager	ES	14	22
Total		223	266

Table 30: MCT Framework use cases.

Component Model Use Cases

Currently there are 24 use cases dedicated to the 25 requirements associated with the Component Model.

CM1: Message/System changes Component Value

Description: A component shall be able to hold state, including references to other components.
Scope: COMP
Primary Actor: SYS
Stakeholders: UP, Component C, COMP assignment method ASSN, COMP message method MESG, C assignment operation AO (as bang string), Behavior Entity BE
Preconditions: UP has been initialized, C exists, SYS exists, SYS has reference to C
Trigger: SYS calls MESG on C with values
Postconditions: C has new value
Primary/Success Scenario:
- SYS calls MESG on C with values
- COMP maps AO to BE implementing it for C
- Invoke the BE
- BE calls ASSN
- C value changes

Secondary/Failure Scenarios: C assignment fails, C assignment throws exception
Notes: System version doesn't throw exception

CM2: Message/System retrieves Component Value

Description: Component functional roles and constituents shall be examinable (access to structure, behavior, and values) at runtime.
Scope: COMP
Primary Actor: SYS
Stakeholders: UP, Component C, COMP retrieval method RETR, COMP message method MESG, C retrieval operation RO (as bang string), Behavior Entity BE
Preconditions: UP has been initialized, C exists, SYS exists, SYS has reference to C
Trigger: SYS calls MESG with C
Postconditions: C value is available
Primary/Success Scenario:
- SYS calls MESG on C with values
- COMP maps RO to BE implementing it for C
- Invoke the BE
- BE calls RETR
- C value is available

Secondary/Failure Scenarios: C retrieval fails, C retrieval throws exception
Notes: System version doesn't throw exception

CM3: Message/System to Component

Description: The component state shall be understood as a mapping from names to values. (Reference through component structure)
Scope: COMP
Primary Actor: SYS
Stakeholders: UP, Component C, COMP operation method OPER, COMP message method MESG, C operation OP (as bang string), Behavior Entity BE
Preconditions: UP has been initialized, C exists, SYS exists, SYS has reference to C
Trigger: SYS calls MESG with C and names and values
Postconditions: Operation based on message, and message arguments, is applied to C
Primary/Success Scenario:
- SYS calls MESG on C with names and values

- COMP maps OP to BE implementing it for C
- Invoke the BE
- BE calls OPER
- Operation based on message, and message arguments, is applied to C

Secondary/Failure Scenarios: Operation on C fails, Operation on C throws exception
Notes: System version doesn't throw exception

CM4: Message add Component attribute name

Description: A component shall be dynamically extendable through the addition of functional roles.
Scope: COMP
Primary Actor: SYS
Stakeholders: UP, Component C, COMP addition method ADD, COMP message method MESG, C add operation AO (as bang string), Behavior Entity BE
Preconditions: UP has been initialized, C exists, SYS exists, SYS has reference to C
Trigger: SYS calls MESG with C and attribute name
Postconditions: C has name attribute
Primary/Success Scenario:

- SYS calls MESG on C with attribute name
- COMP maps AO to BE implementing it for C
- Invoke the BE
- BE calls ADD
- C has name attribute

Secondary/Failure Scenarios: Add name operation fails, Add name operation throws exception
Notes: System version doesn't throw exception

CM5: Message add Component attribute value

Description: A component shall be dynamically extendable through the addition of functional roles.
Scope: COMP
Primary Actor: SYS
Stakeholders: UP, Component C, COMP assignment method ASSN, COMP message method MESG, C assignment operation AO (as bang string), Behavior Entity BE
Preconditions: UP has been initialized, C exists, SYS exists, SYS has reference to C
Trigger: SYS calls MESG with C and attribute name and value
Postconditions: C has name attribute and value
Primary/Success Scenario:

- SYS calls MESG on C with attribute name and value
- COMP maps AO to BE implementing it for C
- Invoke the BE
- BE calls ASSN
- C has name attribute and value

Secondary/Failure Scenarios: Assignment operation on name fails, Assignment operation on name throws exception
Notes: System version doesn't throw exception

CM6: Message/System annotate Component

Description: The annotation of component state shall be possible.
Scope: COMP

Primary Actor: SYS
Stakeholders: UP, Component C, COMP add method ADD, COMP message method MESG, C add operation AO (as bang string), Behavior Entity BE
Preconditions: UP has been initialized, C exists, SYS exists, SYS has reference to C
Trigger: SYS calls MESG with message, attribute and facet or note name on C
Postconditions: C attribute or facet is annotated
Primary/Success Scenario:
- SYS calls MESG on C with attribute, facet, note name
- COMP maps AO to BE implementing it for C
- Invoke the BE
- BE calls ADD
- C attribute or facet is annotated

Secondary/Failure Scenarios: Add facet or note operation fails, Add facet or note operation throws exception
Notes: System version doesn't throw exception

CM7: Message/System add Component type (specialization of annotation for type field)

Description: Component values shall be typed through annotation (e.g., other components, behavior actors, primitive programming language types, or object programming language [reference] types).
Scope: COMP
Primary Actor: SYS
Stakeholders: UP, Component C, COMP add method ADD, COMP message method MESG, C add operation AO (as bang string), Behavior Entity BE
Preconditions: UP has been initialized, C exists, SYS exists, SYS has reference to C
Trigger: SYS calls MESG with message, type attribute and value on C
Postconditions: C has type attribute and value
Primary/Success Scenario:
- SYS calls MESG with message, type attribute and value on C
- COMP maps AO to BE implementing it for C
- Invoke the BE
- BE calls ADD
- C has type attribute and value

Secondary/Failure Scenarios: Add name operation fails, Add name operation throws exception
Notes: System version doesn't throw exception

CM8: Message/System change Component value

Description: Component values shall be typed through annotation (e.g., other components, behavior actors, primitive programming language types, or object programming language [reference] types).
Scope: COMP
Primary Actor: SYS
Stakeholders: UP, Component C, COMP add method ASSN, COMP message method MESG, C assignment operation AO (as bang string), Behavior Entity BE
Preconditions: UP has been initialized, C exists, SYS exists, SYS has reference to C
Trigger: SYS calls MESG with message, attribute name and value on C
Postconditions: C has new value for attribute name
Primary/Success Scenario:
- SYS calls MESG with message, attribute name and value on C
- COMP maps AO to BE implementing it for C

- Invoke the BE
- BE calls ASSN
- C has new value for attribute name

Secondary/Failure Scenarios: C assignment fails, C assignment throws exception
Notes: System version doesn't throw exception

CM9: Message/System remove Component attribute or behavior

Description: The state of a component shall be dynamically restricted by removing functional roles.
Scope: COMP
Primary Actor: SYS
Stakeholders: UP, Component C, COMP removal method REM, COMP message method MESG, C removal operation RO (as bang string), Behavior Entity BE
Preconditions: UP has been initialized, C exists, SYS exists, SYS has reference to C
Trigger: SYS calls MESG with remove message, attribute, facet, or note name on C
Postconditions: C no longer has attribute, facet, or note attribute
Primary/Success Scenario:

- SYST calls MESG with remove message, and attribute, facet or note name on C
- COMP maps RO to BE implementing it for C
- Invoke the BE
- BE calls REM
- C no longer has attribute, facet, or note attribute

Secondary/Failure Scenarios: Remove name operation fails, Add name operation throws exception
Notes: System version doesn't throw exception

CM10: Component plays Role

Description: Named role predicates shall be used to define sets of attributes and behaviors that describe capabilities that a component may offer.
Scope: COMP
Primary Actor: COMP
Stakeholders: REG, UP, COMP role predicate method PR, Component C, Role ROLE
Preconditions: UP has been initialized, C and ROLE exist in REG, COMP has reference to C and ROLE
Trigger: UP calls PM on C with ROLE
Postconditions: C has ROLE attributes and behaviors verified
Primary/Success Scenario:

- UP calls PM on C with ROLE
- Iterate through ROLE attributes and behaviors
- Verify that C has each
- C has ROLE attributes and behaviors verified

Secondary/Failure Scenarios: C assignment fails, C assignment throws exception
Notes: System version doesn't throw exception

CM11: Component constructed with Role (sees Ancestors)

Description: Component roles shall inherit attributes/behaviors from their parent roles.
Primary Actor: UP (creation)
Stakeholders: REG, UP, Component C, Role ROLE
Preconditions: UP is being initialized, ROLE exists in REG, UP has reference to ROLE
Trigger: UP construction of C

Postconditions: C has ROLE (and ancestors) attributes and behaviors
Primary/Success Scenario:
- UP construction of C
- Recursively (upward) iterate through ROLE attributes and behaviors
- Clone/Copy attributes and behaviors and add to C (using another use case)
- C has ROLE (and ancestors) attributes and behaviors

Secondary/Failure Scenarios: C assignment fails, C assignment throws exception
Notes: System version doesn't throw exception

CM12: System constructs Component

Requirement: The construction of context-specific implementations of the component shall be provided (factory).
Scope: COMP
Primary Actor: UP (component creation)
Stakeholders: UP, COMP creation method CREAT, Component C, COMP factory FCTY
Preconditions: UP has been initialized
Trigger: UP begins construction of C
Postconditions: Correct baseline component C is constructed
Primary/Success Scenario:
- UP (component creation) begins construction of C
- UP calls CREAT
- CREAT passes to FCTY to determine baseline type
- Correct baseline component type for C is constructed
- Remaining construction tasks are completed
- Correct baseline component C is constructed

Secondary/Failure Scenarios: FCTY construction fails, C construction task fails

CM13: System loads Component

Description:
Scope: COMP
Primary Actor: UP
Stakeholders: REG, UP, ISM, Component C
Preconditions: UP is being initialized, ISM has been initialized
Trigger: UP begins component loading startup phase
Postconditions: C is initialized
Primary/Success Scenario:
- UP begins component loading startup phase
- UP (component creation) reads component description
- UP parses description → C construction
- UP adds C to registry
- C is initialized

Secondary/Failure Scenarios: C description read fails, C description parse fails, C addition to registry fails, C construction fails

CM14: System saves Component (user version)

Description: The system shall support a fundamental set of operations on component roles including retrieval (find), adding a role or roles (add), and removing a role or roles (remove).
Scope: COMP
Primary Actor: PERST

Stakeholders: REG, UP, Component C, ISM, PERST
Preconditions: UP has been initialized, C exists, PERST has a target repository online, PERST has a save method SAV
Trigger: USER selects a save operation (on C)
Postconditions: C is saved
Primary/Success Scenario:
- USER selects a save operation (on C)
- UP calls SAV on C
- PERST serializes C to declarative representation
- PERST saves C to repository
- C is saved

Secondary/Failure Scenarios: User selection fails, SAV fails, C serialization fails, C persistence fails

CM15: System updates/synchronizes Component to peers

Description: It shall be possible for components to be developed independently (outside the MCT IDE), based on existing components, for inclusion into the registry without developing new java code. (declarative design capability.
Scope: COMP
Primary Actor: UP
Stakeholders: REG, UP, Component C, ISM, PERST, POLCY
Preconditions: UP has been initialized, C exists, ISM online, PERST online, POLCY online
Trigger: ISM or PERST identify that C is out of sync with other peers
Postconditions: C is up to date/synchronized with other peers
Primary/Success Scenario:
- ISM notifies the C is out of sync with other peers using POLCY
- Most recent version is accessed
- A difference engine is applied to local and peer versions using POLCY
- Differences are mitigated in the local version
- Result of operation is stored in REG
- PERST saves result of operation
- C is up to data/synchronized with other peers

Secondary/Failure Scenarios: Timing doesn't allow synchronization, Looping updates cause instability, Most recent version cannot be determined, difference engine fails to work, failure to store changes in REG, failure to PERST

CM16: System constructs Component from Prototype

Description: Components shall be able to use other components as prototypes such that behaviors and attributes from the prototype are transferred to the component being constructed.
Scope: COMP
Primary Actor: UP (creation)
Stakeholders: REG, UP, Component C, Prototype PROT
Preconditions: UP has been initialized up to component loading phase, PROT exists in REG
Trigger: UP begins C construction
Postconditions: Component is constructed with PROT attributes and behaviors
Primary/Success Scenario:
- UP begins C construction
- UP (component creation) parses C from declarative description

- COMP construction passes to factory to determine baseline type
- COMP creates baseline C
- UP identifies PROT to construct from
- UP looks up PROT in REG
- UP clones PROT attributes and behaviors and adds to C
- Component is constructed with PROT attributes and behaviors

Secondary/Failure Scenarios: C construction fails, PROT attribute/behavior assignment to C fails, UP unable to identify PROT, UP unable to retrieve PROT from REG, UP unable to clone PROT attributes or behaviors

CM17: System creates Component from Prototype at runtime

Description: Components shall be able to use other components as prototypes such that behaviors and attributes from the prototype are transferred to the component being constructed.
Scope: COMP
Primary Actor: UP (creation)
Stakeholders: CONFIG, REG, UP, Subsystem SYS, Component C, Prototype PROT
Preconditions: UP has been started, REG and ENV have been initialized, PROT has been created and is available through REG
Trigger: SYS calls prototype creation method in ENV to create C using PROT
Postconditions: C is constructed that has PROT attributes and behaviors available in REG
Primary/Success Scenario:

- UP calls REG to create C
- REG calls COMP to create blank Component C
- REG iterates through attributes and behaviors of PROT and uses COMP assignment to update C
- Reference to PROT is created in C
- C with PROT attributes and behaviors is available in REG through ENV

Secondary/Failure Scenarios: C does not have PROT attributes and behaviors

CM18: System verifies Component prototype compliance

Description: A child component shall have access to its parent (or prototype parent) that was used during the child's extension or creation (for compliance).
Scope: COMP
Primary Actor: UP
Stakeholders: REG, UP, Component C, Prototype PROT
Preconditions: UP has been initialized, C exists, PROT exists
Trigger: UP verifies C satisfying PROT
Postconditions: C verified to satisfy PROT
Primary/Success Scenario:

- UP calls verify C satisfying PROT
- C is looked up in REG
- PROT is looked up in REG
- Iterate through PROT attributes and behaviors
- Verify that C has all attributes and behaviors
- C verified to satisfy PROT

Secondary/Failure Scenarios: C lookup fails, PROT lookup fails, C fails to verify

CM19: System creates Component from Prototype with Restrictions

Description: It shall be possible to label the roles of a component as not being prototyped during child component creation or extension.
Scope: COMP
Primary Actor: UP (creation)
Stakeholders: REG, UP, Component C, Prototype PROT
Preconditions: UP has been initialized up to component loading phase, PROT exists in REG, PROT has restrictions (defined in config or defined in PROT)
Trigger: UP begins C construction
Postconditions: Component is constructed with PROT non-restricted attributes and behaviors
Primary/Success Scenario:
- UP begins C construction
- UP (component creation) parses C from declarative description
- COMP construction passes to factory to determine baseline type
- COMP creates baseline C
- UP identifies PROT to construct from
- UP looks up PROT in REG
- UP clones PROT non-restricted attributes and behaviors and adds to C
- Component is constructed with PROT attributes and behaviors

Secondary/Failure Scenarios: C construction fails, PROT attribute/behavior assignment to C fails, UP unable to identify PROT, UP unable to retrieve PROT from REG, UP unable to clone PROT attributes or behaviors

CM20: System creates GUI-based Component

Decription: The system shall provide base code that facilitates the wrapping of GUI widgets with
Scope: COMP
Primary Actor: UP
Stakeholders: REG, UP, ISM, Component C
Preconditions: UP is being initialized, declarative C representation exists, ISM exists
Trigger: UP loads C into REG
Postconditions: C has GUI elements
Primary/Success Scenario:
- UP loads C into REG
- C declarative representation is parsed
- GUI description is parsed into MCT GUI core objects (hierarchical)
- GUI objects are bound to C
- C is added to REG
- C has GUI elements

Secondary/Failure Scenarios: C parse fails, GUI parse fails, GUI binding fails, C addition to REG fails

CM21: System verifies Component value

Description: Component values shall be verified/validated when changes are made.
Scope: COMP
Primary Actor: VALID
Stakeholders: REG, UP, VALID, System SYST, Component C, COMP message method MESS, COMP assignment method ASS
Preconditions: UP has been initialized, C exists, VALID exists

Trigger: SYST calls MESS with assignment message on C
Postconditions: C value change is validated
Primary/Success Scenario:
- SYST calls MESS with assignment message, attribute, and value on C
- C is looked up in REG
- Message arguments, C are sent to ASS
- ASS is executed ← could start use case scenario here
- VALID is informed of C assignment
- VALID looks up C attribute validator
- VALID applies validator to C, attribute, value
- C value change is validated

Secondary/Failure Scenarios: C lookup fails, C assignment fails, C validator cannot be found, C validation fails

CM22: System localizes Component strings

Description: Error! Reference source not found.
Scope: COMP
Primary Actor: LOCL
Stakeholders: REG, UP, LOCL, Locale LOC, Application APPL, Component C
Preconditions: UP has been initialized, C exists, LOCL approach exists
Trigger: APPL renders C
Postconditions: C GUI is localized for rendering
Primary/Success Scenario:
- APPL renders C
- C is looked up in REG
- C GUI is accessed
- C GUI model mapping is accessed
- LOC is accessed from APPL properties
- LOCL is applied to GUI using LOC
- C GUI is localized for rendering

Secondary/Failure Scenarios: C assignment fails, C assignment throws exception

CM23: System supports accessibility requirements

Description: Components will support accessibility requirements as set forth by NASA policy (e.g., 508B).
Scope: COMP
Primary Actor: MCT APPL
Stakeholders: REG, UP, APPL, Accessibility Requirement ACCREQ
Preconditions: UP has been initialized, APPL is running
Trigger: ACCREQ is invoked on APPL
Postconditions: ACCREQ is appropriately applied in APPL
Primary/Success Scenario:
- ACCREQ is invoked on APPL
- APPL environment responds to ACCREQ
- ACCREQ is appropriately applied in APPL

Secondary/Failure Scenarios: APPL environment has no support for ACCRREQ

UI Toolkit Use Cases

Sixtyone use cases (forty shown below) have been identified from the 25 UI Toolkit requirements, as shown below:

UIT1: User compose user object from template objects

Description: Users can design new user objects by extending existing user object types.
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, UIMGR**,** User Object UO
Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO
Trigger: ENTITY select UO
Postconditions: UO and related C are selected
Primary/Success Scenario:

- ENTITY select UO
- UO looked up by UIMGR
- UO mapping from UI to C returned
- UO selection is set
- UI selection is set
- C selection is set
- UO and related C are selected

Secondary/Failure Scenarios: UO lookup fails, UO mapping to UI fails, UI mapping to C fails, UO selection setting fails, UI selection setting fails, C selection setting fails

UIT2: MCT supports widget functionality – implemented with Swing widget

Description: MCT shall provide a transparency layer from MCT widgets, roles, and components to widget set, role, and component implementations (there will be a 1:1 mapping from MCT widget to implementation, but not a 1:1 mapping from MCT widget to a particular implementation.
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, UIMGR**,** User Object UO
Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO
Trigger: ENTITY select UO
Postconditions: UO and related C are selected
Primary/Success Scenario:

- ENTITY select UO
- UO looked up by UIMGR
- UO mapping from UI to C returned
- UO selection is set
- UI selection is set
- C selection is set
- UO and related C are selected

Secondary/Failure Scenarios: UO lookup fails, UO mapping to UI fails, UI mapping to C fails, UO selection setting fails, UI selection setting fails, C selection setting fails

UIT3: Entity select user object

Description: All user objects shall be selectable.
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, REG, UIMGR**,** User Object Listeners UOL, User Object UO, Component C

Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO, REG has reference to C
Trigger: ENTITY selected UO
Postconditions: UO UI and related C are selected
Primary/Success Scenario:
- ENTITY selected UO
- UO mapping to UI, C looked up by UIMGR
- UO selection is set
- UO notifies listeners that it is selected
- UO UI and related C are selected

Secondary/Failure Scenarios: UO lookup fails, UO mapping to UI fails, UI mapping to C fails, UO selection setting fails, UI selection setting fails, C selection setting fails, UO rendering fails

UIT4: Entity select menu option (also Entity right click on User Object)

Description: Users can choose actions on all user objects
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, REG, UIMGR, User Object UO, Component C, MenuItem MI, Operation OP
Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO, REG has reference to C, OP is defined on C
Trigger: ENTITY select MI on UO
Postconditions: OP is applied to UO
Primary/Success Scenario:
- ENTITY select MI on UO
- UO looked up by UIMGR
- UO mapping from UI to C returned
- MI mapping to OP is looked up
- OP is applied to C
- OP is applied to UO

Secondary/Failure Scenarios: UO lookup fails, UO mapping to UI fails, UI mapping to C fails, MI doesn't have OP for UO, OP has no mapping to C, OP application to C fails

UIT5: Entity edit [plot] control panel

Description: Users can adjust a view's content area visualization via the view's control area.
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, REG, UIMGR, User Housing Object UHO, Housing component H, UHO control area CNTL, UHO content area CONT, Edit operation EOP
Preconditions: UP has been initialized, UHO exists, ENTITY exists, UIMGR has reference to UHO, REG has reference to H, CNTL, and CONT, EOP is defined on CNTL, ENTITY has permission to apply EOP to UHO
Trigger: ENTITY apply EOP on UHO CNTL attribute
Postconditions: EOP is applied to UHO CONT
Primary/Success Scenario:
- ENTITY apply EOP on UHO CNTL attribute
- UHO looked up by UIMGR
- CNTL looked up by UIMGR
- CONT looked up by UIMGR

- EOP mapping to CNTL attribute looked up
- EOP (mapping) applied to CNTL attribute
- EOP is applied to UHO CONT

Secondary/Failure Scenarios: UHO lookup fails, UHO mapping to CNTL fails, UI mapping to CONT fails, EOP mapping to CNTL attribute fails, EHO application fails

UIT6: Entity modify [filter] control controls

Description: Users can adjust a view's content area visualization via the view's filter area.
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, REG, UIMGR**,** User Housing Object UHO, Housing component H, UHO filter control area FCNTL, UHO content area CONT, Edit operation EOP
Preconditions: UP has been initialized, UHO exists, ENTITY exists, UIMGR has reference to UHO, REG has reference to H, FCNTL, and CONT, EOP is defined on CNTL, ENTITY has permission to apply EOP to UHO
Trigger: ENTITY apply EOP on UHO FCNTL attribute
Postconditions: EOP is applied to UHO CONT
Primary/Success Scenario:

- ENTITY apply EOP on UHO FCNTL attribute
- UHO looked up by UIMGR
- CNTL looked up by UIMGR
- CONT looked up by UIMGR
- EOP mapping to FCNTL attribute looked up
- EOP (mapping) applied to FCNTL attribute
- EOP is applied to UHO CONT

Secondary/Failure Scenarios: UHO lookup fails, UHO mapping to FCNTL fails, UI mapping to CONT fails, EOP mapping to FCNTL attribute fails, EHO application fails

UIT7: Entity edit user object

Description: Selected user object properties shall be configurable.
Scope: UIT
Primary Actor: ENTITY
Stakeholders: COMP, UP, ENV, REG, UIMGR, POLCY, User Object UO
Preconditions: UP has been initialized, UO exists in ENV, UIMGR has reference to UO
Trigger: ENTITY modifies selected UO UI/model attributes
Postconditions: UO and related UI/model attributes are modified
Primary/Success Scenario:

- ENTITY modifies selected UO UI/model attributes
- UO looked up by UIMGR
- UO mapping from UI to C returned
- Selected attribute changes made (sent to VALID)
- UO selection is set
- UI selection is set
- C selection is set
- UO and related UI/model attributes are modified

Secondary/Failure Scenarios: UO lookup fails, UO mapping to UI fails, UI mapping to C fails, VALID fails on attributes, UO selection setting fails, UI selection setting fails, C selection setting fails

UIT8: Entity render user object

Description: User objects shall have at least one visualization.
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, UIMGR**,** User Object UO
Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO
Trigger: ENTITY select UO
Postconditions: UO and related C are selected
Primary/Success Scenario:

- ENTITY select UO
- UO looked up by UIMGR
- UO mapping from UI to C returned
- UO selection is set
- UI selection is set
- C selection is set
- UO and related C are selected

Secondary/Failure Scenarios: UO lookup fails, UO mapping to UI fails, UI mapping to C fails, UO selection setting fails, UI selection setting fails, C selection setting fails

UIT9: Entity render user object preferred visualization

Description: User objects shall have a preferred visualization.
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, UIMGR**,** User Object UO
Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO
Trigger: ENTITY select UO
Postconditions: UO and related C are selected
Primary/Success Scenario:

- ENTITY select UO
- UO looked up by UIMGR
- UO mapping from UI to C returned
- UO selection is set
- UI selection is set
- C selection is set
- UO and related C are selected

Secondary/Failure Scenarios: UO lookup fails, UO mapping to UI fails, UI mapping to C fails, UO selection setting fails, UI selection setting fails, C selection setting fails

UIT10: Entity delete user object

Description: User objects can be created and controlled via user interface controls (e.g., menus, right clicking, keyboard shortcuts, composition).
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, UIMGR**,** User Object UO
Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO, Delete Action accessible/permitted to user
Trigger: ENTITY select delete instance on UO
Postconditions: UO is deleted and references to UO removed from REG

Primary/Success Scenario:
- ENTITY select delete instance on UO
- Delete operation mapped to ENV
- UO looked up by UIMGR
- UO mapping from UI to C returned
- Delete on C called
- UO removed from screen
- UP calls REG to dereference COMP
- PERST deletes component from repository
- ID USERMAN dereferences component from UE
- UP calls COMP destruction method
- UO is deleted and reference to UO removed from REG

Secondary/Failure Scenarios: UO lookup fails, UO mapping to UI fails, UI mapping to C fails, UO screen removal fails, PERST deletion fails, ID dereference fails, COMP deletion fails

UIT11: Entity select user object displays inspector

Description: User objects shall support high-level interactions such as selection, cut, copy, paste, and inspection as enumerated in table UIT3.
Scope: CLIB
Primary Actor: ENTITY
Stakeholders: UP, UIMGR**,** User Object UO, Inspector INSP
Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO, INSP exists
Trigger: ENTITY select UO
Postconditions: UO inspector representation is displayed in inspector
Primary/Success Scenario:
- ENTITY select UO
- UO looked up by UIMGR
- UO mapping from UI to C returned
- C representations attribute values returned
- C inspector representation found
- C inspector representation created and initialized via component creation and provided to REG
- UP passes reference to new component to UIMGR
- UO inspector representation is displayed in inspector

Secondary/Failure Scenarios: UO lookup fails, UO mapping to UI fails, UI mapping to C fails, Inspector representation identification fails, inspector creation fails, view is not opened in the right place, view isn't opened at all

UIT12: User move cursor

Description: User objects shall support low-level interactions such as mouse and keyboard events, shortcuts.
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, UIMGR**,** User Object UO
Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO
Trigger: ENTITY move cursor
Postconditions: Appropriate action taken depending on location of cursor
Primary/Success Scenario:

- ENTITY move cursor
- Cursor position detected as motion event
- UO position listener notified of event
- UO actor executed
- Appropriate action taken on UO based on cursor position

Secondary/Failure Scenarios: UO has no position listener, UO has no position-specific actor, UO position-specific actor fails

UIT13: Entity enables a representation

Description: GUI widgets shall support nominal GUI widget properties: enablement/disablement, visibility, borders, layout management, accessibility, localization.
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, UIMGR, User Object UO, User Object Enablement Widget (radio button, checkbox, …) UOEW
Preconditions: UP has been initialized, UO exists, ENTITY exists, UOEW exists
Trigger: ENTITY select UOEW
Postconditions: UO items enabled
Primary/Success Scenario:

- ENTITY select UOEW
- UO gui looked up by UIMGR
- UO mapping from UOEW to related items returned
- Selection logic applied to related items
- UO items enabled

Secondary/Failure Scenarios: UO gui lookup fails, logic application fails

UIT14: Get Parent Representation

Description: User objects can be hierarchical with respect to GUI containment (i.e., it maps 1:1 to the GUI containment hierarchy).
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, UIMGR, User Object UO
Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO
Trigger: ENTITY operate on UO
Postconditions: Operation is applied to UI ancestry
Primary/Success Scenario:

- ENTITY operate on UO
- UO looked up by UIMGR
- UO mapping from UI to C returned
- Operation on UO UI is applied
- Operation is applied to UI ancestry

Secondary/Failure Scenarios: UO lookup fails, UO mapping to UI fails, UI mapping to C fails, UO operation application fails

UIT15: Copy this representation

Description: User objects shall support copying/cloning.
Scope: UIT
Primary Actor: ENTITY

Stakeholders: UP, UIMGR**,** User Object UO
Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO
Trigger: ENTITY copy UO
Postconditions: UO copy exists
Primary/Success Scenario:
- ENTITY copy UO
- Local UO copy exists but not persisted
- UO copy exists

Secondary/Failure Scenarios: UO copy fails

UIT16: Entity assigns a model component to a representation

Description: View role may be bound to a model role component.
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, UIMGR**,** User Object UO, UO widget model UOWM, Model Component MC
Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO, MC exists
Trigger: ENTITY assign MC to UOWM
Postconditions: UOWM is bound to MC
Primary/Success Scenario:
- ENTITY assign MC to UOWM
- UO looked up by UIMGR
- UO mapping from UI to C returned
- UOWM is identified
- UIWM is assigned reference to MC
- UOWM is bound to MC

Secondary/Failure Scenarios: UO lookup fails, UO mapping to UI fails, UI mapping to C fails, UO has no UOWM, UOWM binding to MC fails

UIT17: Entity render representation

Description: View roles may be asked to rerender.
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, UIMGR**,** User Object UO
Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO
Trigger: ENTITY select rerender UO
Postconditions: UO is rendered
Primary/Success Scenario:
- ENTITY select rerender UO
- UO looked up by UIMGR
- UO mapping from UI to C returned
- UO UI is set to dirty or otherwise told to redraw
- UO is rendered

Secondary/Failure Scenarios: UO lookup fails, UO mapping to UI fails, UI mapping to C fails, UO selection setting fails, UI redraw fails

UIT18: Entity update representation

Description: Roles may be asked to update.
Scope: UIT
Primary Actor: ENTITY
Stakeholders: UP, UIMGR**,** User Object UO
Preconditions: UP has been initialized, UO exists, ENTITY exists, UIMGR has reference to UO
Trigger: ENTITY select update UO
Postconditions: UO models are updated
Primary/Success Scenario:

- ENTITY select update UO
- UO looked up by UIMGR
- UO mapping from UI to C returned
- Iterate through UI models and have each update
- UO modesl are updated

Secondary/Failure Scenarios: UO lookup fails, UO mapping to UI fails, UI mapping to C fails, UI update fails

UIT19: USER import/open design project into MCT DT

Description: The design portion of the component toolkit shall support design project management.
Scope: UIT
Primary Actor: USER
Stakeholders: USER, DT, UP, UIT, File Menu FM, Design Project DP
Preconditions: Opening DT screen is viewable, FM Open menu item exists, DP exists
Trigger: USER selects FM->Import/Open project
Postconditions: DP is open in DT
Primary/Success Scenario:

- USER selects FM->Import/Open project
- DT displays a file selection browser to USER
- USER navigates to/selects DP in file system navigator and selects submit action
- DT imports DP into current project
- DP is open in DT

Secondary/Failure Scenarios: DT unable to import DP

UIT20: USER export design project into MCT

Description: The design portion of the component toolkit shall support design project management.
Scope: UIT
Primary Actor: USER
Stakeholders: USER, DT, UP, UIT, File Menu FM, Design Project DP
Preconditions: The DT is open
Trigger: USER selects FM->Export/Close project
Postconditions: DP is not open in DT, DP exported to desired location and name
Primary/Success Scenario:

- USER selects FM->Export Project from DT File menu
- DT displays a file selection browser to USER
- USER navigates to and selects path and project name in file system navigator and selects submit action
- DT exports project file to desired directory and project name

- DT closes DP
- DP is not open in DT, DP exported to desired location and name

Secondary/Failure Scenarios: Desired file cannot be written by DT

UIT21: USER create design project in MCT

Description: The design portion of the component toolkit shall support design project management.
Scope: UIT
Primary Actor: USER
Stakeholders: REG, UP, File Menu FM
Preconditions: The DT is open
Trigger: USER selects FM->New from the DT file menu
Postconditions: A new project has been created and is ready for editing
Primary/Success Scenario:

- USER selects selects FM->New from the DT file menu
- DT displays project creation wizard to USER
- USER selects type of project (e.g., new, from existing) from creation wizard and fills in appropriate information
- DT creates necessary files
- DT opens project into browser and creates a blank design canvas for the project
- A new project has been created and is ready for editing

Secondary/Failure Scenarios: DT unable to create new project files, DT unable to open project in browser, DT unable to create new canvas

UIT22: USER edit design project in MCT

Description: The design portion of the component toolkit shall support design project management.
Scope: UIT
Primary Actor: USER
Stakeholders: REG, UP, Edit Menu EM
Preconditions: The DT is open
Trigger: USER selects EM->Copy menu item
Postconditions: A design item has been copied into buffer
Primary/Success Scenario:

- USER selects EM->Copy menu item
- DT copies currently-selected component
- A design item has been copied into buffer

Secondary/Failure Scenarios: DT unable to copy design component
Similar Use Cases: USER selects EM->Paste menu item, USER selects FM->Properties, USER changes design item properties, USER performs normal design operations

UIT23: USER save design project in MCT

Description: The design portion of the component toolkit shall support design project management.
Scope: UIT
Primary Actor: USER
Stakeholders: REG, UP, File Menu FM
Preconditions: The DT is open
Trigger: USER selects FM->Save menu item

Postconditions: The design project has been saved and all design items have been exported

Primary/Success Scenario:

- USER selects FM->Save menu item
- DT performs an update on the project file contents
- DT iterates through all design componets and saves to project location
- The design project has been saved and all design items have been exported

Secondary/Failure Scenarios: DT unable to save design project, DT unable to export design components

UIT24: USER delete design project in MCT

Description: The design portion of the component toolkit shall support design project management.
Scope: UIT
Primary Actor: USER
Stakeholders: REG, UP, File Menu FM
Preconditions: The DT is open
Trigger: USER selects FM->Delete menu item
Postconditions: The design project has been closed without saving from DT and all files removed from the file system
Primary/Success Scenario:

- USER selects FM->Delete menu item
- DT closes the project without saving
- DT removes project files from file system
- The design project has been closed without saving from DT and all files removed from the file system

Secondary/Failure Scenarios: DT unable to remove files from the file system

UIT25: Add a display component to the view

Description: The design IDE supports layout control and management.
Scope: UIT
Primary Actor: USER
Stakeholders: USER, DT, UIT, UP
Preconditions: An open project with modified files is open in USER's DT
Trigger: USER selects an item from the DT component palette
Postconditions: The component is displayed in the DT project view
Primary/Success Scenario:

- USER selects an item from the DT component palette
- USER places item onto the DT project view
- DT validates operation and opens the component inspector
- USER fills in applicable content in the component inspector
- DT propagates information to other viewers
- UIT performs applicable rendering
- The component is displayed in the DT project view

Secondary/Failure Scenarios: USER unable to select item from palette, USER unable to place selected item in project view, USER unable to fill in content to inspector, UIT unable to render

UIT26: Remove a display component from the view

Description: The design IDE supports layout control and management.

Scope: UIT
Primary Actor: USER
Stakeholders: USER, DT, Design Item DI
Preconditions: An open project with modified files is open in USER's DT
Trigger: USER selects delete for DI
Postconditions: DI is removed from the DT project view but remains in the palette
Primary/Success Scenario:

- USER selects delete for DI
- DT removes DI from canvas
- DT clears inspector
- DT deletes DI
- DT lays out remaining items
- DI is removed from the DT project view but remains in the palette

Secondary/Failure Scenarios: DT unable to remove DI from canvas, DT unable to delete DI, DT unable to lay out remaining items
Similar Use Cases: USER selects EM->Delete item

UIT27: Copy a display component to a new location

Description: The design IDE supports layout control and management.
Scope: UIT
Primary Actor: USER
Stakeholders: USER, DT, UIT, Design Item DI, Edit Menu EM
Preconditions: An open project with selected DI
Trigger: USER selects EM->Copy, EM->Paste menu item
Postconditions: The component is displayed in multiple locations in the DT project view
Primary/Success Scenario:

- USER selects EM->Copy menu item on selected DI
- DT copies the selected DI into buffer
- USER moves cursor to new location in DT project view
- USER selects EM->Paste menu item
- DT pastes buffered DI copy to new location
- DT adds DI copy to new layout
- DT propagates information to other viewers
- UIT performs applicable rendering
- The component is displayed multiple locations in the DT project view

Secondary/Failure Scenarios: DT unable to copy DI to buffer, DT unable to paste DI to new location, DT unable to add copied DI to new location layout, DT unable to propagate new item to other viewers
Similar Use Cases: USER drag/drop DI, USER rc DI/USER rc DI

UIT28: Move a display component to a new location

Description: The design IDE supports layout control and management.
Scope: UIT
Primary Actor: USER
Stakeholders: USER, DT, UIT, Design Item DI, Edit Menu EM
Preconditions: An open project with selected DI
Trigger: USER selects EM->Cut, EM->Paste menu items
Postconditions: The component is displayed in a new location in the DT project view
Primary/Success Scenario:

- USER selects EM->Cut menu item on selected DI
- DT copies the selected DI into buffer

- DT removes DI from current layout/location
- USER moves cursor to new location in DT project view
- USER selects EM->Paste menu item
- DT pastes buffered DI copy to new location
- DT adds DI copy to new layout
- DT propagates information to other viewers
- UIT performs applicable rendering
- The component is displayed multiple locations in the DT project view

Secondary/Failure Scenarios: DT unable to copy DI to buffer, DT unable to remove DI from first layout, DT unable to paste DI to new location, DT unable to add copied DI to new location layout, DT unable to propagate new item to other viewers

Similar Use Cases: USER cntl drag/drop DI, USER rc DI/USER rc DI

UIT29: Edit component properties

Description: The design portion of the component toolkit shall support GUI component editing.

Scope: UIT

Primary Actor: USER

Stakeholders: USER, DT, UIT, UP

Preconditions: An open project with open component view is open in USER's DT

Trigger: USER modifies location or properties of ADT component in project view or USER modifies attributes in ADT component inspector

Postconditions: The modifications have been rendered in the project view and propagated to the appropriate views

Primary/Success Scenario:

- USER modifies location or properties of ADT component in project view or USER modifies attributes in ADT component inspector
- DT propagates content changes to listening components
- UIT performs applicable rendering
- The modifications have been rendered in the project view and propagated to the appropriate views

Secondary/Failure Scenarios: USER unable to modify selected item, changes don't propagate, UIT unable to render

UIT30: Map a model to a component

Description: The component toolkit shall support model to component mapping in design mode.

Scope: UIT

Primary Actor: USER

Stakeholders: USER, DT, UIT, UP, UI Widget UIW, Model Mapping View MMV, Model Component MC

Preconditions: An open project with open component view is open in USER's DT, UI widget inspector is open, MMV is open

Trigger: USER selects MC to map a UIW field to

Postconditions: The UIW field is bound to MC

Primary/Success Scenario:

- USER selects MC to map a UIW field to
- DT assigns MC to UIW field
- The UIW field is bound to MC

Secondary/Failure Scenarios: No MC exists, DT fails at binding MC to UIW field

UIT31: Create a rule

Description: The design portion of the component toolkit shall support rule creation in design mode.
Scope: UIT
Primary Actor: USER
Stakeholders: USER, DT, UIT, UP
Preconditions: An open project is in USER's DT
Trigger: USER selects Create Rule from DT Rules menu
Postconditions: The new rule has been associated with the project
Primary/Success Scenario:

- USER selects Create Rule from DT Rules menu
- DT opens the ADT rule generator
- USER creates rules in visual environment and modifies in rule inspector
- DT validates rule (and rule against rules) using RV
- DT shows errors to USER in the DT console
- DT places validated rules into the appropriate grouping
- The new rule has been associated with the project

Secondary/Failure Scenarios: Create Rule operation fails, rule editing fails, rule validation fails, rule not placed into appropriate grouping

UIT32: Remove a rule

Description: The design portion of the component toolkit shall support rule creation in design mode.
Scope: UIT
Primary Actor: USER
Stakeholders: USER, DT, UIT, UP
Preconditions: An open project is in USER's DT
Trigger: USER selects Remove Rule from DT Rules menu
Postconditions: The new rule has been removed from the project
Primary/Success Scenario:

- USER selects Remove Rule from DT Rules menu
- The new rule has been removed from the project

Secondary/Failure Scenarios: Remove Rule operation fails, rule removal fails

UIT33: Validate rules

Description: The design portion of the component toolkit shall support rule creation in design mode.
Scope: UIT
Primary Actor: USER
Stakeholders: USER, DT, UIT, UP
Preconditions: An open project is in USER's DT
Trigger: USER selects Validate Rules from DT Rules menu
Postconditions: The validation result is displayed in the DT console
Primary/Success Scenario:

- USER selects Validate Rules from DT Rules menu
- DT calls rule validator to validate all project rules or selected rule grouping
- The validation result is displayed in the DT console

Secondary/Failure Scenarios: Remove Rule operation fails, rule removal fails

UIT34: Create an action

Description: The design portion of the component toolkit shall support action management in design mode.
Scope: UIT
Primary Actor: USER
Stakeholders: USER, DT, UP, Design Component DC, Model Mapping View MMV
Preconditions: An open project is in USER's DT
Trigger: USER selects Create Action from DT Actions menu
Postconditions: The new action has been associated with DC
Primary/Success Scenario:
- USER selects DC
- USER selects Create Action from DT Actions menu
- DT opens the DT action generator
- USER selects model from MMV to bind with action
- DT creates action for DC
- The new action has been associated with DC

Secondary/Failure Scenarios: Create Action operation fails, action editing fails

UIT35: Map an action to a component

Description: The design portion of the component toolkit shall support action management in design mode.
Scope: UIT
Primary Actor: USER
Stakeholders: USER, DT, UIT, UP, UI Widget UIW, Action Mapping View AMV, Model Component MC, Model Inspector MI
Preconditions: An open project with open component view is open in USER's DT, UI widget inspector is open, AMV is open
Trigger: USER selects MC actor to map a UIW to
Postconditions: The UIW action is bound to MC actor
Primary/Success Scenario:
- USER selects UIW
- USER selects UIW action from MI
- USER selects MC actor from AMV
- DT assigns MC actor to UIW action
- The UIW action is bound to MC actor

Secondary/Failure Scenarios: No MC actor exists, DT fails at binding MC actor to UIW action

UIT36: Undo operation(s)

Description: The design portion of the component toolkit shall support undo/redo histories in design mode.
Scope: UIT
Primary Actor: USER
Stakeholders: REG, UP, APPL, DT Edit Menu EM, DT History DTH
Preconditions: UP has been initialized, DT is open
Trigger: USER selects EM->Undo
Postconditions: Last operation is undone
Primary/Success Scenario:
- USER selects EM->Undo
- DTH retrieves previous state

- DTH loads previous state
- Last operation is undone

Secondary/Failure Scenarios: No states in DTH, Previous state is corrupted, Previous state cannot be loaded

UIT37: Redo operation(s)

Description: The design portion of the component toolkit shall support undo/redo histories in design mode.
Scope: UIT
Primary Actor: USER
Stakeholders: REG, UP, APPL, DT Edit Menu EM, DT History DTH
Preconditions: UP has been initialized, DT is open
Trigger: USER selects EM->Redo
Postconditions: Last operation is redone
Primary/Success Scenario:

- USER selects EM->Redo
- DTH retrieves next state
- DTH loads next state
- Last operation is redone

Secondary/Failure Scenarios: No states in DTH, Next state is corrupted, Next state cannot be loaded

UIT38: Create a workflow

Description: The design portion of the component toolkit shall support workflow creation and management in design mode.
Scope: UIT
Primary Actor: USER
Stakeholders: USER, DT, UP, File Menu FM, Workflow View WV
Preconditions: An open project is in USER's DT
Trigger: USER selects FM->Create Workflow
Postconditions: The new workflow has been associated with the project
Primary/Success Scenario:

- USER selects FM->Create Workflow
- DT opens the WV
- DT adds workflow palette items to palette
- USER drags/drops workflow items to WV
- DT validates workflow additions as added
- The new workflow has been associated with the project

Secondary/Failure Scenarios: Create Workflow operation fails, workflow editing fails, workflow addition validation fails

Component Library Use Cases

Two use cases have been identified from the 2 Component Library requirements, as shown below:

CL1: Entity edit [plot] control panel

Description: Users can adjust a view's content area visualization via the view's control area.
Scope: CLIB

Primary Actor: ENTITY
Stakeholders: UP, REG, UIMGR, User Housing Object UHO, Housing component H, UHO control area CNTL, UHO content area CONT, Edit operation EOP
Preconditions: UP has been initialized, UHO exists, ENTITY exists, UIMGR has reference to UHO, REG has reference to H, CNTL, and CONT, EOP is defined on CNTL, ENTITY has permission to apply EOP to UHO
Trigger: ENTITY apply EOP on UHO CNTL attribute
Postconditions: EOP is applied to UHO CONT
Primary/Success Scenario:
- ENTITY apply EOP on UHO CNTL attribute
- UHO looked up by UIMGR
- CNTL looked up by UIMGR
- CONT looked up by UIMGR
- EOP mapping to CNTL attribute looked up
- EOP (mapping) applied to CNTL attribute
- EOP is applied to UHO CONT

Secondary/Failure Scenarios: UHO lookup fails, UHO mapping to CNTL fails, UI mapping to CONT fails, EOP mapping to CNTL attribute fails, EHO application fails

CL2: Entity modify [filter] control controls

Description: Users can adjust a view's content area visualization via the view's filter area.
Scope: CLIB
Primary Actor: ENTITY
Stakeholders: UP, REG, UIMGR, User Housing Object UHO, Housing component H, UHO filter control area FCNTL, UHO content area CONT, Edit operation EOP
Preconditions: UP has been initialized, UHO exists, ENTITY exists, UIMGR has reference to UHO, REG has reference to H, FCNTL, and CONT, EOP is defined on CNTL, ENTITY has permission to apply EOP to UHO
Trigger: ENTITY apply EOP on UHO FCNTL attribute
Postconditions: EOP is applied to UHO CONT
Primary/Success Scenario:
- ENTITY apply EOP on UHO FCNTL attribute
- UHO looked up by UIMGR
- CNTL looked up by UIMGR
- CONT looked up by UIMGR
- EOP mapping to FCNTL attribute looked up
- EOP (mapping) applied to FCNTL attribute
- EOP is applied to UHO CONT

Secondary/Failure Scenarios: UHO lookup fails, UHO mapping to FCNTL fails, UI mapping to CONT fails, EOP mapping to FCNTL attribute fails, EHO application fails

Information Semantics Manager Use Cases

Seven use cases have been identified from the 17 Information Semantics Manager requirements, as shown below:

ISM1: ISM check C plays role Role

Description: The information semantics management subsystem shall include a service for determining if a component satisfies a role description
Scope: UP
Primary Actor: UP

Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

ISM2: ISM merge ontology1 and ontology2

Description: The information semantics management subsystem shall support ontology merging based upon configurable policies.
Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

ISM3: ISM check C plays role Role

Description: Error! Reference source not found.
Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

ISM4: ENV merges ES and ISM metadata to components

Description: Error! Reference source not found.

Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

ISM5: ISM update ontology

Description: Error! Reference source not found.
Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

ISM6: SYS query component model from ISM

Description: The core model knowledge stores shall provide a query interface to components that makes it possible to search models' semantic webs.
Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

User Platform Use Cases

Eleven use cases have been identified from the 26 User Platform requirements, as shown below:

UP1: SYST access UP ENV

Description: The component execution environment (user platform) shall provide access to the global environment (to all properties including user, session, hardware device, and namespace information, and to services and subsystems managed by the user platform).
Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:
- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

UP2: USER update MCT

Description: The system shall dynamically check (at appropriate times) for updates to code and install these updates.
Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:
- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

UP3: SYS find object by name

Description: The User Platform shall provide name resolution mechanisms to discover components from their symbolic names.
Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates

Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:
- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

UP4: SYS access service from UP [UP register as peer with pub/sub broker]

Description: The User Platform subsystem shall aggregate a suite of services provided by different parts of the MCT infrastructure and make these services accessible to all of the components it manages.
Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:
- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

UP5: UP register as peer with pub/sub broker

Description: The User Platform will provide for a mechanism to interact as a peer with the messaging subsystem.
Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:
- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

UP6: UP startup MCT

Description: The User Platform shall support component initialization/reinitialization.

Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

Configuration Manager Use Cases

Four use cases have been identified from the 9 Configuration Manager requirements, as shown below:

CNFG1: SYST configure SYST

Description: Each MCT service or subsystem will be responsible for applying its own configurations.
Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

CNFG2: ENV apply configs to components

Description: User objects are configurable.
Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction

- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

CNFG3: CONFIG validate SYST-Configs using SYST-Configs-Schema

Description: The central configuration subsystem will validate all configuration files.
Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

CNFG4: SYST define SYST-Configs-Schema

Description: Each MCT service or subsystem will create its own configuration schema and configuration file.
Scope: UP
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

Event Handler Use Cases

Ten use cases have been identified among the 19 Event Handler specific requirements, as detailed below:

EH1: EH log security event to file

Description: The central event handling subsystem shall support the logging and auditing of security events.
Scope: HANDL
Primary Actor: SYS
Stakeholders: UP, POLCY, Security event E

Preconditions: UP has been initialized
Trigger: SYS invokes E
Postconditions: Security event log includes security event information
Primary/Success Scenario:

- SYS invokes E
- HANDL informed of E
- HANDL dispatches security event to POLCY
- POLCY determines correct policy for E
- POLCY is applied by HANDL handler
- HANDL logs E to log file
- Security event log includes E information

Secondary/Failure Scenarios: HANDL doesn't receive notification of E, Improper policy is identified, Policy not applied to E, HANDL exception thrown

EH2: EH log system failure to file

Description: The central event handling subsystem shall have a mechanism for the logging of system failures.
Scope: HANDL
Primary Actor: SYS
Stakeholders: UP, POLCY, System event E
Preconditions: UP has been initialized
Trigger: SYS invokes E
Postconditions: System event log includes system event information
Primary/Success Scenario:

- SYS invokes ACT E
- HANDL informed of E
- HANDL dispatches E to POLCY
- POLCY determines correct policy for E
- POLCY is applied by HANDL handler
- HANDL logs E to log file
- System event log includes security event information

Secondary/Failure Scenarios: HANDL doesn't receive notification of E, Improper policy is identified, Policy not applied to E, HANDL exception thrown

EH3: EH log app event to file

Description: The central event handling subsystem shall have a mechanism to log application events.
Scope: HANDL
Primary Actor: SYS
Stakeholders: UP, POLCY, Event E
Preconditions: UP has been initialized, E exists
Trigger: SYS invokes E
Postconditions: Application event log includes system event information
Primary/Success Scenario:

- SYS invokes E
- HANDL informed of E
- HANDL creates application event
- HANDL dispatches E to POLCY
- POLCY determines correct policy for E
- POLCY is applied by HANDL handler
- HANDL logs E to log file

- Application event log includes security event information

Secondary/Failure Scenarios: HANDL doesn't receive notification of E, Improper policy is identified, Policy not applied to E, HANDL exception thrown

EH4: USER view EH event log from file

Description: The central event handling subsystem shall permit user examination of the system event log.
Scope: HANDL
Primary Actor: HANDL
Stakeholders: UP, POLCY, Event E
Preconditions: UP has been initialized, E exists
Trigger: USER views E in event log EL
Postconditions: NA
Primary/Success Scenario:

- USER views E in event log EL

Secondary/Failure Scenarios: EL cannot be opened

EH5: EH register handler

Description: The central event handling subsystem shall provide a mechanism to handle events uniformly across the system but to handle them differentially based on event type.
Scope: HANDL
Primary Actor: HANDL
Stakeholders: UP, POLCY, Component C, System SYS, Handler H
Preconditions: UP has been initialized, C exists, H exists
Trigger: UP instantiates C
Postconditions: C is registered with H
Primary/Success Scenario:

- UP instantiates H
- SYS notifies HANDL of C
- HANDL registers C for event type and category
- C is registered with H

Secondary/Failure Scenarios: HANDL exception thrown

EH6: Event handler deregisters handler

Description: The central event handling subsystem shall provide a mechanism to handle events uniformly across the system but to handle them differentially based on event type.
Scope: HANDL
Primary Actor: HANDL
Stakeholders: REG, UP, System SYS
Preconditions: UP has been initialized, Component C
Trigger: C is removed from SYS
Postconditions: Handler removed from registry
Primary/Success Scenario:

- C is removed from REG
- SYS notifies HANDL of C
- HANDL deregisters all handlers for C

Secondary/Failure Scenarios: HANDL exception thrown

EH7: EH register event type and category

Description: The central event handling subsystem shall permit the dynamic addition of event types and categories.
Scope: HANDL
Primary Actor: HANDL
Stakeholders: UP, POLCY, Component C, System SYS, Handler H
Preconditions: UP has been initialized, C exists, H exists
Trigger: UP instantiates C
Postconditions: C is registered with H
Primary/Success Scenario:
- UP instantiates C
- SYS notifies HANDL of C
- HANDL registers C for event type and category
- C is registered with H

Secondary/Failure Scenarios: HANDL exception thrown

EH8: Event handler deregisters event type and category

Description: The central event handling subsystem shall permit the dynamic addition of event types and categories.
Scope: HANDL
Primary Actor: HANDL
Stakeholders: REG, UP, System SYS, Component C
Preconditions: UP has been initialized, C exists
Trigger: C is removed from REG
Postconditions: Event type and/or category removed from HANDL registry
Primary/Success Scenario:
- C is removed from REG
- SYS notifies HANDL of C
- HANDL deregisters event type and category
- Event type and/or category removed from HANDL registry

Secondary/Failure Scenarios: HANDL exception thrown

EH9: Event handler persists events

Description: The central event handling subsystem shall persist event information by way of the persistence management subsystem.
Scope: HANDL
Primary Actor: HANDL
Stakeholders: UP, SYS, PERST, Event E
Preconditions: UP has been initialized, E exists
Trigger: HANDL event queue has reached limit
Postconditions: Event is persisted
Primary/Success Scenario:
- HANDL event queue has reached limit
- HANDL notifies PERST with E
- PERST persists E
- E is persisted

Secondary/Failure Scenarios: Persistence with PERST fails, HANDL exception thrown

EH10: EH retrieve event history by query

Description: The central event handling subsystem shall include past event retrieval via query.
Scope: HANDL
Primary Actor: HANDL
Stakeholders: UP, SYS, PERST, Event E, Event Query EQ
Preconditions: UP has been initialized, E exists
Trigger: SYS invokes EQ
Postconditions: HANDL returns event to SYS
Primary/Success Scenario:

- SYS invokes EQ
- HANDL determines whether current or past event
- If necessary, HANDL retrieves event from persisted store using PERST
- HANDL processes EQ
- HANDL returns event to SYS

Secondary/Failure Scenarios: HANDL query handler fails, PERST retrieval fails, EQ processing fails, HANDL exception thrown

EH11: EH handles Component action failure

Description: The central event handling subsystem operations shall be policy based (e.g., failure noticing, failure-ignoring).
Scope: HANDL
Primary Actor: HANDL
Stakeholders: REG, UP, POLCY, SYS, Component C
Preconditions: UP has been initialized, C exists, Actor ACT
Trigger: SYS invokes ACT on C unsuccessfully
Postconditions: Appropriate action taken on failure
Primary/Success Scenario:

- SYS invokes ACT on C unsuccessfully
- Exception thrown against ACT, C
- HANDL informed of exception
- HANDL dispatches to POLCY
- POLCY determines correct policy for ACT, C
- POLCY is applied by HANDL handler
- Appropriate action taken on failure

Secondary/Failure Scenarios: HANDL doesn't receive exception, Improper policy is identified, Policy not applied to ACT, C, HANDL exception thrown

EH12: EH handles event by policy

Description: The central event handling subsystem shall be parameterized with an event handler execution policy. This policy specifies which handlers should service an event, the order of handling, and how/if multiple handlings of single events is performed.
Scope: HANDL
Primary Actor: HANDL
Stakeholders: UP, SYS, POLCY, Event E, Policy P
Preconditions: UP has been initialized, E and P exist
Trigger: SYS invokes E
Postconditions: E is handled according to P
Primary/Success Scenario:

- SYS invokes E
- HANDL informed of E
- HANDL dispatches to POLCY
- POLCY determines correct P for E
- P is applied to E by HANDL handler
- E is handled according to P

Secondary/Failure Scenarios: HANDL exception thrown

EH13: EH is configurable

Description: To facilitate the creation of event descriptions, the central event handling subsystem shall permit the dynamic configuration of its event description factory.
Scope: HANDL
Primary Actor: HANDL
Stakeholders: UP, SYS
Preconditions: UP is being initialized, HANDL has configuration schema and valid configurations file
Trigger: UP invokes configuration on HANDL
Postconditions: HANDL is configured
Primary/Success Scenario:

- UP invokes configuration on HANDL
- HANDL iterates through provided configuration
- HANDL applies configuration parameters to internal members
- HANDL is configured

Secondary/Failure Scenarios: HANDL unable to apply configuration parameter values, HANDL exception thrown

Identity Manager Use Cases

Thirteen use cases have been identified among the 17 Identity Manager specific requirements, as detailed below:

ID1: ID initialization

Description: The Identity Management subsystem shall ensure that a user has sufficient privileges to access data or executable resources**.**
Scope: ID
Primary Actor: USER
Stakeholders: UP, CONFIG, External Environment (EX)
Preconditions: EX is running and configured to allow access to MCT
Triggers: MCT is launched by USER in EX
Postconditions: MCT is configured using external identity information. MCT has access to EX system resources(e.g. permissions to use file system, network interfaces)
Success Scenario:

- MCT is launched by USER in EX
- MCT is granted permission to execute and is given permission to system resources by EX OS
- MCT initializes UP which initializes ID
- ID queries EX for identity and configuration information
- EX provides ID/UP with identity information and authorizations necessary for MCT to execute on EX(in addition to startup/execution permissions)
- MCT continues startup

Failure Scenarios: MCT does not have permissions to access system resources, non executable; USER does not have correct permissions.

ID2: USER Role invoke operation

Description: The Identity Management subsystem shall include the ability to grant access to resources based on the current role of a user when the request was made
Scope: IDMGR
Primary Actor: USER
Stakeholders: UP, AUTH, CONFIG
Preconditions: UP initialized, start up sequence activated, IDMgr initialized, USER exists in system, CONFIG has relevant authentication module information
Triggers: UP triggers IDMGR to begin authentication sequence during startup
Postconditions: USER is authenticated, IDMGR/UP startup continue
Success Scenario:

- USER launches MCT Application
- UP begins IDMGR Startup Sequence
- IDMGR Startup Sequence triggers AUTH
- AUTH loads authentication modules specified by CONFIG
- AUTH queries External Environment or USER for authentication information
- AUTH checks auth information and authenticates user
- AUTH allows USER to configure information requested by modules (ie role selection)
- USER is authenticated, IDMGR/UP startup continue

Failure Scenarios: USER fails authentication 3 times, system exits

ID3: Single Sign-On Authentication

Description: The Identity Management subsystem shall interoperate with the external authentication mechanism such that users login once and gain access to all appropriate resources without further authentication.
Scope: IDMGR
Primary Actor: USER
Stakeholders: UP, AUTH, CONFIG, EX (external environment), External Source
Preconditions: UP initialized, ID initialized, USER Authenticated
Triggers: System performs an action A1 that requires authentication with an external source
Postconditions: Action A1 is performed
Success Scenario:

- Action A1 is performed by system
- ID queries EX and/or CONFIG for authentication information needed to perform A1
- EX returns requires authentication information
- A1 is authenticated with external source
- A1 is executed

Failure scenario: Do not have authentication to perform A1, A1 fails

ID4: ID use policy to assign rights to USER

Description: Users will have policy-based and configurable/assignable rights.
Scope: ID
Primary Actor: USER
Stakeholders: UP, ID, role R1, Action A1, POLICY

Preconditions: UP initialized, ID initialized, USER Authenticated, USER Role is R1
Triggers: System performs an action A1 that requires authorization granted by R1
Postconditions: Action A1 is performed
Success Scenario:

- A1 is performed
- UP performs authorization check using ID
- ID uses POLICY to check authorization for USER
- POLICY matches R1 security policy to permissions for A1
- POLICY returns authorization to ID
- UP allows A1 to be performed and not be spaced out all weird

Failure scenario: R1 does not have permission to perform A1

ID5: Create User Enviroment

Description: Each user identity has its own root collection of user objects called a user environment.
Scope: ID
Primary Actor: USER
Stakeholders: UP, AUTH, CONFIG, ENV
Preconditions: UP initialized. Start up sequence activated, IDMgr available, USER authenticated
Triggers: USER authenticates and IDMGR start sequence begins user requisitioning to USERMAN
Postconditions: User environment is made available through ENV
Success Scenario:

- ID builds User component with Identity information (from AUTH modules, and configured identity stores)
- User permissions located in configured location loaded into user component for authorization requests
- ID looks up USER's User environment collection in PERS and REG and builds collection components
- User Environment is added to the user component so that it can be made available to MCT through ENV

Failure Scenarios: USER does not exist

ID6: UP provide access to User Environment

Description: The Identity Management subsystem will control access to and content of user environments.
Scope: ID
Primary Actor: USER
Stakeholders: UP, AUTH, CONFIG, ENV
Preconditions: UP initialized. Start up sequence activated, IDMgr available, USER authenticated
Triggers: USER authenticates with MCT
Postconditions: User environment is made available through ENV
Success Scenario:

- ID looks up user environment in PERS and REG
- ID Builds user according to APP CONFIG and identity information
- ID loads root collection for USER role
- ID loads USER's personal collection from PERS
- ID provides interface to user environment through environment

Failure scenario: User environment collections cannot be loaded

ID7: User Environment access

Description:
Scope: ID
Primary Actor: USER
Stakeholders: UP, ID, COMP, REG, User Env Component C1, User Env Rep Comp REP1
Preconditions: UP initialized. User is logged into MCT. User Environment requisitioned and available through ID
Triggers: USER logs in and defualt user environment rep is triggered to be displayed
Postconditions: USER's environment rep is displayed with correct user collections
Success Scenario:

- USER logs in and default user environment rep is triggered to be displayed
- After log in APP is configured to open REP1
- UP uses COMP and REG to instantiate a User Environment Rep Component, REP1
- UP queries ID for User Environment C1 providing correct user information
- ID provides reference to C1(including collections)
- UP passes C1 reference to COMP
- COMP sets the Model of REP1 to C1
- REP1 is displayed
- USER's environment rep is displayed with correct user collections

Failure Scenario: incorrect C1 provided, REP1 is not displayed

ID8: ID manage users

Description: The Identity Management subsystem shall have a mechanism for managing MCT users.
Scope: ID
Primary Actor: HANDL
Stakeholders: REG, UP, POLCY, Component C, System SYS
Preconditions: UP has been initialized, C exists, Actor ACT
Trigger: SYS invokes ACT on C unsuccessfully
Postconditions: Appropriate action taken on failure
Primary/Success Scenario:

- SYS invokes ACT on C unsuccessfully
- Exception thrown against ACT, C
- HANDL informed of exception
- HANDL dispatches to POLCY
- POLCY determines correct policy for ACT, C
- POLCY is applied by HANDL handler
- Appropriate action taken on failure

Secondary/Failure Scenarios: HANDL doesn't receive exception, Improper policy is identified, Policy not applied to ACT, C, HANDL exception thrown

ID9: ID operate using policy

Description: The Identity Management subsystem operations shall be policy based.
Scope: ID
Primary Actor: HANDL
Stakeholders: UP, POLCY
Preconditions: UP is in start up sequence
Trigger: ID begins start up sequence

Postconditions: ID is configured with appropriate policies
Primary/Success Scenario:
- UP uses POLCY to determine startup of ID
- ID uses POLCY to determine initial configuration and to set policies for initial ID startup actions
- ID keeps reference of currently active policies in POLCY
- ID references policies during various points of startup and execution

Secondary/Failure Scenarios: ID policies do not exist, ID does not start correctly

ID10: ID persist user env

Description: The Identity Management subsystem operations shall support the persistence of users (e.g., user environments, preferences).
Scope: ID
Primary Actor: USER
Stakeholders: UP, PERS, ID
Preconditions: UP initialized. USER is logged into MCT.
Triggers: USER creates a new telemetry group
Postconditions: USER's environment is persisted with new telemetry group added
Success Scenario:
- USER creates a new collection in their User Environment collection
- COMP alerts UP of change
- UP uses policy to determine whether to persist change
- ID is alerted by UP to persist user information
- ID uses PERS to persist user environment information
- USER's environment is persisted with new telemetry group added

Failure Scenarios: User environment change is not persisted

ID11: Defining Security Policies

Description:
Scope: ID
Primary Actor: COMP
Stakeholders: UP, COMP, POLCY, Component C1, Action ACT1, Permission PERM1, Relevant Policy POL1 (related to ACT1 on Component C1 using PERM1), USER
Preconditions: UP initialized, USER is logged in, User component contains user permission PERM1, POLCY configured with Access control policy POL1.
Triggers: C1 performs ACT1
Postconditions: Execution is returned to C1
Success Scenario:
- C1 performs ACT1
- COMP message call is intercepted by Access Control mechanism
- COMP calls ID to perform access control policy decision
- ID looks up POL1 in POLCY for ACT1 using permission PERM1
- POLCY decides whether ACT1 is allowed with permission P1
- ID returns permission decision
- COMP executes ACT1

  Alt. COMP does not allow ACT1 to execute

Failure Scenarios: User collection does not exist

ID12: ID provides information about USER through ENV to components

Description: The Identity Management subsystem shall provide user information to components and other subsystems.
Scope: ID
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, User Component UC1, USER
Preconditions: UP initialized. USER is logged in, User component contains user information, ID available through ENV
Triggers: C1 queries ENV for USER information
Postconditions: ENV returns response to query
Success Scenario:
- ENV uses ID component context to query for USER information from user component
- ID delegates request to USERMAN
- USERMAN accesses UC1 through REG (something only the USERMAN has permission to do) and queries for user information
- USERMAN returns information to ENV for use by component

Failure Scenarios: User collection does not existNote: This could be for access to user environment should not be allowed to be accessed through the REG to just any component or subsystem

ID13: ID provides ES with authentication information

Description: The Identity Management subsystem shall manage security information for external services.
Scope: ID
Primary Actor: USER
Stakeholders: UP, AUTH, CONFIG, ENV
Preconditions: MCT/UP running, CONFIG contains pointer to identity information for external services.
Triggers: External service performs action that requires authentication
Postconditions: External service finishes action
Success Scenario:
- EXT service queries ID for authentication information
- ID uses CONFIG to locate AUTH information
- ID caches auth information for later use
- ID either 1. Provides authentication information to service (http request requiring basic auth?), or 2. Authenticates with service and passes execution to ext service?

Failure Scenarios: USER collection does not exist

Messaging Use Cases

Eleven use cases have been identified among the 17 Messaging-specific requirements, as detailed below:

COM1: COMP message COMP

Description: There shall be a distinction between local and remote communication from the point of view of the components.
Scope: COM
Primary Actor: COMP

Stakeholders: UP, ID, ENV, REG, Component C1, Component C2 (local or remote), USER
Preconditions: UP initialized, USER is logged in, C1 and C2 are initilialized correctly, COM is initialized correctly, has network access
Triggers: C1 wants to send message to C2
Postconditions: C2 receives message
Success Scenario:

- C1 gets referenceto C2 through REG (through naming convention)
- C1 accesses messaging ops through ENV
- ENV uses COM to send message to C2
- COM sends message through configured message bus/messaging protocol or local REG mechanism
- COM delivers message to COM of C2 (may be the same COM)
- C2 gets message thorugh appropriate message receiving actor

Failure Scenarios: C2 does not receive message, message is lost

COM2: COMP subscribe to COMP field

Description: The central Messaging layer shall include a publish and subscribe communication mechanism.
Scope: COM
Primary Actor: COMP
Stakeholders: UP, COM, ENV, Subsystem S1, Lease L1
Preconditions: UP initialized, COM initialized, publication lease L1 created for UP, subscription to L1 made by subsystem S1
Trigger: UP uses sysops to publish message M1 through L1 using COM
Postconditions: S1 receives publication from UP
Success Scenario:

- UP uses sysops to publish message M1 through L1 using COM
- COM gets publish call and publishes M1 to message bus on L1
- COM's subscription mechanism receives message and delegates it to all subscribers to L1
- S1 receives publication from UP

Failure Scenarios: S1 does not receive M1

COM3: COM policy based

Description: Error! Reference source not found.
Scope: COM
Primary Actor: COM
Stakeholders: UP, POLCY
Preconditions: UP is in start up sequence
Trigger: COM begins start up sequence
Postconditions: COM is configured with appropriate policies
Primary/Success Scenario:

- UP uses POLCY to determine startup of COM
- COM uses POLCY to determine initial configuration and to set policies for initial COM startup actions
- COM keeps reference of currently active policies in POLCY
- COM references policies during various points of startup and execution

Secondary/Failure Scenarios: COM policies do not exist, COM does not start correctly component or subsystem

COM4: COMP subscribe to COMP field

Description: Components shall be able to subscribe to messages by their content, where content is defined according to a publish/subscribe language/logic.
Scope: COM
Primary Actor: COMP
Stakeholders: UP, COM, ENV, Component C1, Component C2, Field of C1: F1
Preconditions: UP initialized. COM initialized, C1 and C2 created, C1:F1 contains value
Triggers: C2 wants to subscribe to C1:F1
Postconditions: C2 receives updates on the value of C1:F1
Success Scenario:

- C2 wants to subscribe to C1:F1
- C2 uses COM sysops available through ENV to register "interest" in C1:F1
- COM creates a subscription to C1:F1 and registers C2 as a recipient with appropriate actor for receiving messages on C2
- When C1:F1 is updated it publishes the change through COM
- C2 receives the change message from COM and UP, and performs registered behavior

Failure Scenarios: Subscrption is not made, C1:F1 does not publish changes, C2 does not receive value change

COM5: COMP subscribe to COMP type

Description: Components shall be able to subscribe to messages by their type, where type is defined according to a publish/subscribe language/logic.
Scope: COM
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, Message type T1, message type T1 lease L1
Preconditions: UP initialized, COM inialized, L1 created for type T1, C1 subscribed to L1 through COM subscription mechanism
Triggers: Message of type T1 is published by COM
Postconditions: C1 receives message of type T1
Success Scenario:

- Message of type T1 is published by COM
- COM subscription mechanism receives T1
- COM delegates message to C1 for processing
- C1 receives message of type T1

Failure Scenarios: C1 does not receive message, C1 receives message of wrong type

COM6: COMP publish by lease

Description: Publishers shall be able to specify a lease for a message.
Scope: COM
Primary Actor: COMP
Stakeholders: UP, COM, ENV, Component C1, Component C2 , Lease L1
Preconditions: UP initialized. COM initialized, C1 and C2 created, C2 has subscription to lease L1
Triggers: C1 uses sysops in ENV to COM api to publish message
Postconditions: C2 receives message from C1
Success Scenario:

- C1 uses sysops in ENV to COM api to publish message
- C1 uses publish message operation in COM to publish message M1 through lease L1

- COM looks up L1 and publishes M1 onto the messaging layer
- COM receives publication and looks for subscribers to L1
- C2 is a subscriber to L1 so it is sent M1
- C2 receives message from C1

Failure Scenarios: C2 does not receive M1

COM7: COMP message COMP

Description: Components shall have the ability to communicate directly with other components.
Scope: COM
Primary Actor: COM
Stakeholders: UP, ENV, REG, Component C1, Component C2
Preconditions: UP initialized, COM initialized, C1 and C2 created and initialized
Triggers: C1 wants to send message to C2
Postconditions: C2 performs action based on message from C1
Success Scenario:

- C1 gets reference to C2 through REG
- C1 acceses COM ops through ENV
- C1 uses send message op to send message to C2
- COM passes message onto message bus to C2
- C2 receives message from COM

Failure Scenarios: C2 does not receive message

COM8: COMP message COMP

Description: The central Messaging layer shall provide a synchronous communication mechanism for component to component communication.
Scope: COM
Primary Actor: COMP
Stakeholders: UP, ENV, REG, Component C1, Component C2
Preconditions: UP initialized, COM initialized, C1 and C2 created and initialized
Triggers: C1 wants to send message to C2 and receive a response
Postconditions: C2 sends response to message from C1
Success Scenario:

- C1 wants to send message to C2 and receive a response
- C1 gets reference to C2 through REG
- C1 acceses COM ops through ENV
- C1 uses send message op to send message to C2
- COM passes message onto message bus to C2
- C2 receives message from COM
- C2 acesses COM ops through ENV and sends response message to C1
- COM receives message and delegates it to C1
- C2 sends response to message from C1

Failure Scenarios: C2 does not receive message, C2 does not send response, C1 does not receive response

COM9: COMP message COMP

Description: Component to component communication shall be loosely coupled.
Scope: COM
Primary Actor: COM
Stakeholders: UP, ENV, REG, Component C1, Component C2

Preconditions: UP initialized, COM initialized, C1 and C2 created and initialized
Triggers: C1 wants to send message to C2
Postconditions: C2 performs action based on message from C1
Success Scenario:

- C1 wants to send message to C2
- C1 gets reference to C2 through REG
- C1 acceses COM ops through ENV
- C1 uses send message op to send message to C2
- COM passes message onto message bus to C2
- C2 receives message from COM
- C2 performs action based on message from C1

Failure Scenarios: C2 does not receive message

COM10: UP update

Description: Error! Reference source not found.
Scope: COM
Primary Actor: COM
Stakeholders: UP on one client UP1, UP on another client UP2, HANDL, POLCY
Preconditions: UP1 and UP2 initialized, COM initialized with working network connection, UP1 and UP2 subscribed to platform changes lease
Triggers: UP1 generates an update event
Postconditions: UP2 receives update command
Success Scenario:

- UP1 generates an update event
- UP1 update event goes to HANDL
- HANDL uses POLCY to determine whether to propagate update
- HANDL accesses Com sysops to send update message on the update lease
- COM publishes UP1 change to message bus
- UP2 COM receives update and delegates update message to UP2
- UP2 receives update command

Failure Scenarios: UP2 does not receive update message, UP1 update event does not trigger a message

COM11: COMP message COMP

Description: Changes to component state shall be propagated to all components with registered interest, particularly active representations visualizing that state
Scope: COM
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, Component C2
Preconditions: UP initialized, C2 has registered interest (subscribed to) changes in C1
Triggers: C1 generates a state changed event
Postconditions: C2 receives C1 change message, acts accordingly
Success Scenario:

- C1 generatges a state changed event
- HANDL uses policy to decide whether to propagate the change
- HANDL and UP use Com sysops to access publish subscribe mechanism
- COM publishes component state change message on message bus
- COM gets message and delegates to component subscribed to C1 changes
- C2 receives C1 change message, acts accordingly

Failure Scenarios: C2 does not receive change event message

External Services Use Cases

ES3: ES handlePolicy on COMP

Description: The External Services subsystem shall be policy based.
Scope: ES
Primary Actor: ES
Stakeholders: COMP, POLICY
Preconditions: UP initialized. EXT configured, loaded, and started
Triggers: COMP invokes access operation on EXT
Postconditions: POLICY returns access permission for COMP based on operation
Success Scenario:
- COMP invokes access operation on EXT
- POLICY is queries for permissions of COMP for operation on EXT
- POLICY returns access code for COMP to operation on EXT

Failure Scenarios: POLICY does not return an access code for COMP operation on EXT

ES4: SYS access ES

Description: The External Services subsystem shall be available to services and subsystems through the shared platform environment.
Scope: ES
Primary Actor: SYS
Stakeholders: ES, UP
Preconditions: UP initialized
Triggers: SYS wants access to ES
Postconditions: SYS has access to ES
Success Scenario:
- SYS wants to access ES
- SYS gets access to ES context from UP

Failure Scenarios: ES not started

ES5: ES discovers EXT metadata

Description: The External Services subsystem shall provide the means to discover and access 3rd party source metadata.
Scope: ES
Primary Actor: ES
Stakeholders: EXT
Preconditions: ES initialized, EXT is started
Triggers: ES requests EXT metadata
Postconditions: ES has EXT metadata
Success Scenario:
- ES requests EXT metadata
- ES has EXT metadata

Failure Scenarios: EXT has no metadata, EXT has no API for providing metadata

ES6: ES publishes EXT metadata

Description: The External Services subsystem shall provide the means to discover and access 3rd party source metadata.
Scope: ES
Primary Actor: ES
Stakeholders: EXT, UP, SYS

Preconditions: ES, UP, and SYS initialized, EXT is started
Triggers: SYS wants access to EXT metadata
Postconditions: SYS has access to EXT metadata
Success Scenario:
- SYS wants to access EXT metadata
- SYS invoked ES request for EXT metadata through ES context
- ES returns EXT metadata to SYS
- SYS has access to EXT metadata

Failure Scenarios: ES doesn't have access to EXT metadata, EXT has no metadata

ES7: EXT ask MCT for service description

Description: The External Services subsystem shall provide 3rd parties the ability to discover component services.
Scope: ES
Primary Actor: ES
Stakeholders: MCT, POLICY
Preconditions: UP initialized
Triggers: EXT queries ES for MCT services
Postconditions: MCT provides service description
Success Scenario:
- ES queries ES for service description
- MCT checks POLICY for query
- MCT provides service description

Failure Scenarios:

ES8: ES export COMP

Description: The External Services subsystem shall provide component export services.
Scope: ES
Primary Actor: ES
Stakeholders: POLICY, COMP
Preconditions: UP initialized. EXT configured, loaded, and started
Triggers: EXT queries for COMP
Postconditions: ENV returns response to query
Success Scenario:
- ENV uses ID component context to query for User information from user component

Failure Scenarios:

ES9: ES asks for EXT metadata

Description: The External Services subsystem shall offer a set of metadata attributes and behaviors that are applicable across all wrapped applications. This metadata shall conform to the semantic description language used by the information semantics manager.
Scope: ES
Primary Actor: ES
Stakeholders: POLICY, COMP
Preconditions: UP initialized. EXT configured, loaded, and started
Triggers: EXT queries for COMP
Postconditions: ENV returns response to query
Success Scenario:

- ENV uses ID component context to query for User information from user component

Failure Scenarios:

ES10: SYS asks ES for EXT metadata

Description: The External Services subsystem shall offer a set of metadata attributes and behaviors that are applicable across all wrapped applications. This metadata shall conform to the semantic description language used by the information semantics manager.
Scope: ES
Primary Actor: ES
Stakeholders: POLICY, COMP
Preconditions: UP initialized. EXT configured, loaded, and started
Triggers: EXT queries for COMP
Postconditions: ENV returns response to query
Success Scenario:

- ENV uses ID component context to query for User information from user component

Failure Scenarios:

ES11: ES ask EXT for QoS

Description: Service adapters shall enforce the secure interaction with external applications in cooperation with the identity management and security subsystems.
Scope: ES
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, User Component UC1
Preconditions: UP initialized. User is logged in, User component contains user information, ID available through ENV
Triggers: C1 queries ENV for user information
Postconditions: ENV returns response to query
Success Scenario:

- ENV uses ID component context to query for User information from user component
- ID delegates request to USERMAN
- USERMAN accesses UC1 through REG(something only the USERMAN has permission to do) and queries for user information
- USERMAN returns information to ENV for use by component

Failure Scenarios: User collection does not existNote: This could be for access to user environment should not be allowed to be accessed through the REG to just any component or subsystem

ES12: ES write data to COMP

Description: External application service adapters shall hide the network protocol used to connect to the application from components using adapters.
Scope: ES
Primary Actor: EXT
Stakeholders: UP, COMP, REG
Preconditions: UP initialized. EXT is configured, loaded and started
Triggers: EXT has new data for COMP
Postconditions: COMP is updated with data

Success Scenario:
- EXT has new data for COMP
- ENV uses COMP id to query for COMP instance
- EXT applies update api on COMP to update COMP data
- COMP is updated with data

Failure Scenarios: COMP does not accept data update

ES13: ES load EXT

Description: The External Services subsystem shall provide a common management point for all 3rd party services.
Scope: ES
Primary Actor: ES
Stakeholders: UP, SYS, EXT
Preconditions: UP initialized. EXT is not loaded
Triggers: SYS wants to load EXT
Postconditions: EXT is loaded into memory
Success Scenario:
- SYS wants to load EXT
- SYS gets EXT id from ES context
- SYS executes EXT load API from ES context
- EXT is loaded

Failure Scenarios: EXT id is invalid

ES13: ES start EXT

Description: The External Services subsystem shall provide a common management point for all 3rd party services.
Scope: ES
Primary Actor: ES
Stakeholders: UP, SYS, EXT
Preconditions: UP initialized. EXT is loaded
Triggers: SYS wants to start EXT
Postconditions: EXT is started
Success Scenario:
- SYS wants to start EXT
- SYS gets EXT id from ES context
- SYS executes start api for EXT using ES context
- EXT is started

Failure Scenarios: EXT id is invalid

ES13: ES stop EXT

Description: The External Services subsystem shall provide a common management point for all 3rd party services.
Scope: ES
Primary Actor: ES
Stakeholders: UP, SYS, EXT
Preconditions: UP initialized. EXT started
Triggers: SYS wants to stop EXT
Postconditions: EXT is stopped
Success Scenario:
- SYS wants to stop EXT

- SYS gets EXT id from ES through ES context
- SYS executes EXT stop api through ES context
- EXT is stopped

Failure Scenarios: EXT id is invalid

ES13: ES unload EXT

Description: The External Services subsystem shall provide a common management point for all 3rd party services.
Scope: ES
Primary Actor: ES
Stakeholders: SYS, EXT, UP
Preconditions: UP initialized, EXT loaded but stopped
Triggers: SYS wants to unload EXT
Postconditions: EXT is unloaded from memory
Success Scenario:

- SYS wants to unload EXT
- SYS gets EXT id from ES context
- SYS executes unload api from ES context
- EXT is unloaded

Failure Scenarios: EXT id is invalid

ES13: ES get EXT

Description: The External Services subsystem shall provide a common management point for all 3rd party services.
Scope: ES
Primary Actor: ES
Stakeholders: COMP, EXT, UP
Preconditions: UP initialized, EXT loaded
Triggers: COMP wants to perform action on EXT
Postconditions: COMP gets EXT reference from ES context
Success Scenario:

- COMP wants to perform action on EXT
- COMP retrieves EXT reference from ES context
- ES provides EXT access to COMP through its context

Failure Scenarios: EXT reference id is invalid

ES15: EXT get data from SERVICE

Description: The External Services subsystem shall support synchronous and asynchronous communication between a component and the application it wraps.
Scope: ES
Primary Actor: EXT
Stakeholders: ES, EXT, SERVICE
Preconditions: UP initialized. EXT configured, loaded and started
Triggers: EXT wants data from SERVICE
Postconditions: EXT receives data from SERVICE
Success Scenario:

- EXT wants data from SERVICE
- EXT requests data from SERVICE or requests SERVICE updates
- SERVICE provides data updates
- EXT receives data from SERVICE

Failure Scenarios: SERVICE is not available, invalid api

ES15: EXT provide data to SERVICE

Description: The External Services subsystem shall support synchronous and asynchronous communication between a component and the application it wraps.
Scope: ES
Primary Actor: EXT
Stakeholders: ES, EXT, SERVICE
Preconditions: UP initialized. EXT configured, loaded and started
Triggers: EXT has new data for SERVICE
Postconditions: EXT provides data to SERVICE
Success Scenario:
- EXT has new data for SERVICE
- EXT posts data to SERVICE using SERVICE api
- EXT provides data to SERVICE

Failure Scenarios: SERVICE is not available, invalid api

ES15: EXT get data from COMP

Description: The External Services subsystem shall support synchronous and asynchronous communication between a component and the application it wraps.
Scope: ES
Primary Actor: EXT
Stakeholders: ES, COMP, REG
Preconditions: UP initialized. EXT configured, loaded and started
Triggers: EXT wants data from COMP
Postconditions: EXT receives data from COMP
Success Scenario:
- EXT wants data from COMP
- EXT gets COMP reference from REG
- EXT requests data from COMP
- EXT receives data from COMP

Failure Scenarios: SERVICE is not available, invalid api

ES16: EXT subscribe data

Description: The External Services subsystem shall support push and pull external applications.
Scope: ES
Primary Actor: ES
Stakeholders: EXT, SERVICE
Preconditions: EXT is configured to retrieve data
Triggers: EXT provides subscription to SERVICE api using subscription api
Postconditions: EXT is subscribed to data through SERVICE
Success Scenario:
- EXT provides subscription to SERVICE api using subscription api
- EXT is subscribed to data through SERVICE according to subscription api

Failure Scenarios: Subscription to SERVICE fails

ES16: EXT publish data

Description: The External Services subsystem shall support push and pull external applications.

Scope: ES
Primary Actor: EXT
Stakeholders: COMP, SERVICE, ES
Preconditions: UP initialized. EXT configured for publish to SERVICE, loaded, and started, SERVICE available
Triggers: COMP is updated
Postconditions: EXT publishes COMP data to SERVICE
Success Scenario:
- COMP is updated
- ES configures EXT with data
- SERVICE configured to subscribe to COMP through ES and EXT
- EXT publishes COMP data to SERVICE

Failure Scenarios: SERVICE not subscribed to COMP through ES/EXT

ES17: ES get EXT status

Description: It shall be possible to query a service adapter for the status of a pending asynchronous request
Scope: ES
Primary Actor: ES
Stakeholders: COMP, EXT, UP
Preconditions: UP initialized
Triggers: COMP queries EXT for status information
Postconditions: COMP receives status information
Success Scenario:
- COMP queries EXT for status information
- ES looks up EXT status information
- ES provides EXT status information to COMP through context api
- COMP receives status information from ES

Failure Scenarios: EXT does not have status information

ES18: ES batches requests

Description: The External Services subsystem shall include the implementation of a mechanism for batching requests according to a parameterizable policy.
Scope: ES
Primary Actor: ES
Stakeholders: UP, COMP, EXT, POLICY
Preconditions: UP initialized
Triggers: COMP makes batch request to EXT
Postconditions: Request is batched
Success Scenario:
- COMP gets EXT id through ES context
- COMP makes batch request to EXT
- ES checks POLICY for COMP, EXT and request
- ES batches request to EXT
- Request is batched

Failure Scenarios: Batch policies not available from POLICY

ES19: ES schedules batched requests

Description: A batching policy shall exist that allows external application communications with a component to be scheduled at specific instances in time

Scope: ES
Primary Actor: ES
Stakeholders: UP, COMP, EXT, POLICY
Preconditions: UP initialized
Triggers: Batch request made to EXT
Postconditions: EXT executes batches requests at scheduled time
Success Scenario:

- Batch request made to EXT
- COMP gets EXT id through ES context
- ES checks POLICY for COMP, EXT and batch request
- ES batches request
- ES checks POLICY for batch schedule
- EXT executes batched requests at schedule time

Failure Scenarios: Batching policies not available from POLICY

ES20: ES adjusts batches via POLICY

Description: A policy shall exist that makes possible the batching of component communications with its wrapped application according to the number of queued requests.
Scope: ES
Primary Actor: ES
Stakeholders: UP, COMP, EXT
Preconditions: UP initialized
Triggers: Queued requests reach configured threshold
Postconditions: Requests are executed
Success Scenario:

- Queued requests reach configured threshold
- COMP gets EXT id through ES context
- COMP makes N number of requests to EXT
- ES checks POLICY for COMP and EXT
- Requests are executed

Failure Scenarios: User collection does not existNote: This could be for access to user environment should not be allowed to be accessed through the REG to just any component or subsystem

ES21: ES notifies UP of state changes

Description: It shall be possible for the External Services subsystem to notify the User Platform of any changes in state.
Scope: ES
Primary Actor: ES, UP
Stakeholders: UP
Preconditions: UP initialized, ES initialized, UP subscribed to ES state changes
Triggers: ES undergoes state change
Postconditions: UP is aware of ES state
Success Scenario:

- ES undergoes state change
- ES publishes state change
- UP is notified of ES state change
- UP is aware of ES state

Failure Scenarios: UP is not subscribed to ES state changes

Rule Engine Use Cases

RE1: COMP execute function

Description: The rule engine shall permit the execution of application code in accordance with its active rules.
Scope: IE
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, User Component UC1
Preconditions: UP initialized. User is logged in, User component contains user information, ID available through ENV
Triggers: C1 queries ENV for user information
Postconditions: ENV returns response to query
Success Scenario:

- ENV uses ID component context to query for User information from user component
- ID delegates request to USERMAN
- USERMAN accesses UC1 through REG(something only the USERMAN has permission to do) and queries for user information
- USERMAN returns information to ENV for use by component

Failure Scenarios: User collection does not existNote: This could be for access to user environment should not be allowed to be accessed through the REG to just any component or subsystem

RE2: COMP field value +

Description: The rule language shall include the +, -, x, / operations.
Scope: IE
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, User Component UC1
Preconditions: UP initialized. User is logged in, User component contains user information, ID available through ENV
Triggers: C1 queries ENV for user information
Postconditions: ENV returns response to query
Success Scenario:

- ENV uses ID component context to query for User information from user component
- ID delegates request to USERMAN
- USERMAN accesses UC1 through REG(something only the USERMAN has permission to do) and queries for user information
- USERMAN returns information to ENV for use by component

Failure Scenarios: User collection does not existNote: This could be for access to user environment should not be allowed to be accessed through the REG to just any component or subsystem

RE3: COMP field value <

Description: The rule language shall include the fundamental set of comparator operations including <, >, =, >=, <=, !=.
Scope: IE
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, User Component UC1

Preconditions: UP initialized. User is logged in, User component contains user information, ID available through ENV
Triggers: C1 queries ENV for user information
Postconditions: ENV returns response to query
Success Scenario:

- ENV uses ID component context to query for User information from user component
- ID delegates request to USERMAN
- USERMAN accesses UC1 through REG(something only the USERMAN has permission to do) and queries for user information
- USERMAN returns information to ENV for use by component

Failure Scenarios: User collection does not existNote: This could be for access to user environment should not be allowed to be accessed through the REG to just any component or subsystem

RE4: ATOM OR ATOM

Description: The rule language shall include the AND, OR, and NOT logical operators.
Scope: IE
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, User Component UC1
Preconditions: UP initialized. User is logged in, User component contains user information, ID available through ENV
Triggers: C1 queries ENV for user information
Postconditions: ENV returns response to query
Success Scenario:

- ENV uses ID component context to query for User information from user component
- ID delegates request to USERMAN
- USERMAN accesses UC1 through REG(something only the USERMAN has permission to do) and queries for user information
- USERMAN returns information to ENV for use by component

Failure Scenarios: User collection does not existNote: This could be for access to user environment should not be allowed to be accessed through the REG to just any component or subsystem

RE5: RE select rule execution

Description: The rule engine shall include a mechanism to select which rules are executed when multiple policies are satisfied.
Scope: IE
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, User Component UC1
Preconditions: UP initialized. User is logged in, User component contains user information, ID available through ENV
Triggers: C1 queries ENV for user information
Postconditions: ENV returns response to query
Success Scenario:

- ENV uses ID component context to query for User information from user component
- ID delegates request to USERMAN
- USERMAN accesses UC1 through REG(something only the USERMAN has permission to do) and queries for user information

- USERMAN returns information to ENV for use by component

Failure Scenarios: User collection does not existNote: This could be for access to user environment should not be allowed to be accessed through the REG to just any component or subsystem

RE6: RE selection parameterization

Description: The rule engine shall permit the parameterization of selection strategies to enforce when multiple policies are satisfied.
Scope: RE
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, User Component UC1
Preconditions: UP initialized. User is logged in, User component contains user information, ID available through ENV
Triggers: C1 queries ENV for user information
Postconditions: ENV returns response to query
Success Scenario:

- ENV uses ID component context to query for User information from user component
- ID delegates request to USERMAN
- USERMAN accesses UC1 through REG(something only the USERMAN has permission to do) and queries for user information
- USERMAN returns information to ENV for use by component

Failure Scenarios: User collection does not existNote: This could be for access to user environment should not be allowed to be accessed through the REG to just any component or subsystem

RE7: RE order rules

Description: The rule policy language and rule engine shall support the selection of rules to execute based upon the generality/specificity of the roles that are matched within a policy expression.
Scope: RE
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, User Component UC1
Preconditions: UP initialized. User is logged in, User component contains user information, ID available through ENV
Triggers: C1 queries ENV for user information
Postconditions: ENV returns response to query
Success Scenario:

- ENV uses ID component context to query for User information from user component
- ID delegates request to USERMAN
- USERMAN accesses UC1 through REG(something only the USERMAN has permission to do) and queries for user information
- USERMAN returns information to ENV for use by component

Failure Scenarios: User collection does not existNote: This could be for access to user environment should not be allowed to be accessed through the REG to just any component or subsystem

RE8: RE select rule execution strategy

Description: The rule engine shall support the configuration of how many rules to execute.
Scope: RE
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, User Component UC1
Preconditions: UP initialized. User is logged in, User component contains user information, ID available through ENV
Triggers: C1 queries ENV for user information
Postconditions: ENV returns response to query
Success Scenario:

- ENV uses ID component context to query for User information from user component
- ID delegates request to USERMAN
- USERMAN accesses UC1 through REG(something only the USERMAN has permission to do) and queries for user information
- USERMAN returns information to ENV for use by component

Failure Scenarios: User collection does not existNote: This could be for access to user environment should not be allowed to be accessed through the REG to just any component or subsystem

RE9: RE rule ordering strategy

Description: The rule engine and policy language shall support the association of a priority with policy expressions.
Scope: RE
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, User Component UC1
Preconditions: UP initialized. User is logged in, User component contains user information, ID available through ENV
Triggers: C1 queries ENV for user information
Postconditions: ENV returns response to query
Success Scenario:

- ENV uses ID component context to query for User information from user component
- ID delegates request to USERMAN
- USERMAN accesses UC1 through REG(something only the USERMAN has permission to do) and queries for user information
- USERMAN returns information to ENV for use by component

Failure Scenarios: User collection does not existNote: This could be for access to user environment should not be allowed to be accessed through the REG to just any component or subsystem

RE10: RE is policy based

Description: The rule engine will be policy based.
Scope: RE
Primary Actor: COMP
Stakeholders: UP, ID, Component C1, User Component UC1
Preconditions: UP initialized. User is logged in, User component contains user information, ID available through ENV
Triggers: C1 queries ENV for user information

Postconditions: ENV returns response to query
Success Scenario:

- ENV uses ID component context to query for User information from user component
- ID delegates request to USERMAN
- USERMAN accesses UC1 through REG(something only the USERMAN has permission to do) and queries for user information
- USERMAN returns information to ENV for use by component

Failure Scenarios: User collection does not existNote: This could be for access to user environment should not be allowed to be accessed through the REG to just any component or subsystem

Composition Use Cases

CMPS1: USER drag COMP to COMP

Description: User drag and drop capability is limited only by composition policies.
Scope: CMPS
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

CMPS2: COMP satisfies Role

Description: The composition language shall permit role satisfaction testing of components.
Scope: CMPS
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

CMPS3: Entity compose Components

Description: Composition is governed by composition policies.
Scope: CMPS
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

CMPS4: Entity compose Components

Description: Users can create and modify compositions of user objects.
Scope: CMPS
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

CMPS5: Entity remove user object from container

Description: Component de-composition of user objects shall be possible.
Scope: CMPS
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

CMPS6: Entity compose with drag/drop

Description: User object composition and decomposition can be effected via user interface controls (menus, right clicking, keyboard shortcuts, composition).
Scope: CMPS
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

Constraint Validation Use Cases

CNST1: Fdf

Description:
Scope: CONST
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

Validation Use Cases

VALD1: Entity changes UI widget value

Description: UI widget parameter modifications are verifiable.
Scope: VALID
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

VALD2: Entity changes Component value

Description: Changed component field values shall undergo a validation before values are assigned.
Scope: VALID
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

VALD3: Entity changes Component string, Boolean, or number value

Description: Validation shall include data type checking for strings, Booleans, and numbers.
Scope: VALID
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved

Success Scenario:
- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

VALD4: Entity changes string Component value that has a minimum and maximum number of characters

Description: String type checking shall include min and max number of characters.
Scope: VALID
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:
- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

VALD5: Entity changes alphanumeric Component value

Description: String type checking shall include variations of alphanumeric strings.
Scope: VALID
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:
- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

VALD6: Entity changes integer range Component value

Description: Integer type checking shall include min, max, default and increment checks.
Scope: VALID
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

VALD7: Entity changes numeric range Component value

Description: Number type checking shall include min, min_inclusive, max, and max_inclusive tests.
Scope: VALID
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

VALD8: Entity changes address Component value

Description: Specialty validators shall be supported (e.g., email addresses and ip addresses).
Scope: VALID
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C

- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

VALD9: SYS loads validations

Description: Declaration of component field validations shall be kept separate from the code that implements them.
Scope: VALID
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

Persistence Management Use Cases

dsd

PRST1: PERST get Object

Description: The central Persistence Management subsystem shall support standard storage operations as defined in table PRST1 (e.g., get, put, update, delete).
Scope: PERST
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

PRST2: PERST save Object

Description: The central Persistence Management subsystem shall support standard storage operations as defined in table PRST1 (e.g., get, put, update, delete).
Scope: PERST
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

PRST3: PERST update Object

Description: The central Persistence Management subsystem shall support standard storage operations as defined in table PRST1 (e.g., get, put, update, delete).
Scope: PERST
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.
Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:

- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

PRST4: PERST delete Object

Description: The central Persistence Management subsystem shall support standard storage operations as defined in table PRST1 (e.g., get, put, update, delete).
Scope: PERST
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Component C
Preconditions: MCT/UP running, C is available.

Triggers: SYS get C
Postconditions: C is retrieved
Success Scenario:
- SYS get C
- Dispatch to POLCY
- POLCY resolves PERST policies on C
- POLCY dispatches policy to PERST
- PERST handles receive for C
- PERST returns C
- C is retrieved

Failure Scenarios: Dispatch to POLCY fails, POLCY fails, dispatch to PERST fails, PERST fails

PRST5: PERST get Object with Query

Description: The central Persistence Management subsystem shall support mechanisms to query the persistence storage.
Scope: PERST
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Repository R, Query Q, Persistence Management System PMS
Preconditions: MCT/UP running, PMS is running, R is available.
Triggers: PERST query Q to R through PMS
Postconditions: R result to Q is returned to PERST
Success Scenario:
- PERST query Q to R through PMS
- Q converted to appropriate PMS query language
- PERST submit converted query through PMS submission to R
- PERST receive Q response from R through PMS
- R result to Q is returned to PERST

Failure Scenarios: Q conversion fails, PMS submission fails

PRST6: PERST get large Object

Description: The central Persistence Management subsystem shall support cursors in the case of large data set retrieval.
Scope: PERST
Primary Actor: SYS
Stakeholders: UP, ENV, POLCY, Repository R, Query Q, Persistence Management System PMS
Preconditions: MCT/UP running, PMS is running, R is available.
Triggers: PERST query Q to R through PMS
Postconditions: R result to Q is returned to PERST with cursor information
Success Scenario:
- PERST query Q to R through PMS
- Q converted to appropriate PMS query language
- PERST submit converted query through PMS submission to R
- PERST receive Q response from R through PMS
- R result to Q is returned to PERST with cursor information

Failure Scenarios: Q conversion fails, PMS submission fails

PRST7: SYS operate on Object

Description: Error! Reference source not found.
Scope: PERST
Primary Actor: SYS
Stakeholders: REG, UP, POLCY, Component C, SYS
Preconditions: UP has been initialized, C exists, Actor ACT
Trigger: SYS invokes ACT on C
Postconditions: Appropriate persistence action taken on C
Primary/Success Scenario:

- SYS invokes ACT on C
- ACT dispatched to POLCY on way to PERST
- POLCY determines correct policy for ACT, C
- policy is applied by PERST handler
- Appropriate persistence action taken on C

Secondary/Failure Scenarios: Improper policy is identified, Policy not applied to ACT, C, PERST exception thrown

PRST8: SYS access PERST

Description: Error! Reference source not found.
Scope: PERST
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

PRST9: SYS modify Object

Description: The central Persistence Management subsystem shall persist model and representation components at the point of update.
Scope: PERST
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

PRST10: SYS save Object

Description: The central Persistence Management subsystem shall persist all user object specific entities during creation: model mappings, action mappings, rules, and validations.
Scope: PERST
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

PRST11: UP restore MCT

Description: The central Persistence Management subsystem shall support system state restoration.
Scope: PERST
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

PRST12: USER save entity with keyboard strokes

Description: User objects can be persisted via user interface controls (menus, right clicking, keyboard shortcuts).
Scope: PERST
Primary Actor: UP
Stakeholders: UP, ENV, ENVMGR, delegates
Preconditions: UP is being initialized, all services and subsystems have delegates
Trigger: UP begins ENV construction phase in startup
Postconditions: Services and subsystems have access to ENV
Primary/Success Scenario:

- UP begins ENV construction phase in startup
- ENV constructed with service members and subsystem delegates
- ENV added to ENVMGR
- Subsystems given ENV after construction
- Services and subsystems have access to ENV

Secondary/Failure Scenarios: ENV construction fails, Subsystem delegates fail, ENV management fails, ENV not assigned to services or subsystems

Policy Management Use Cases

Seven use cases have been identified among the 13 Policy Manager specific requirements, as detailed below:

PLCY1: SYS apply operation

Description: The central Policy Management subsystem shall be policy based.
Scope: POLCY
Primary Actor: SYS
Stakeholders: UP, Component C
Preconditions: UP has been initialized, C exists, Actor ACT
Trigger: SYS invokes ACT on C
Postconditions: Policy-appropriate action taken on C
Primary/Success Scenario:

- SYS invokes ACT on C
- Listener defined on C dispatches to POLCY
- POLCY looks up policies for ACT, C
- POLCY determines correct policy for ACT, C
- POLCY is applied by appropriate handler
- Policy-appropriate action taken on C

Secondary/Failure Scenarios: POLCY doesn't receive ACT content, Improper policy is identified, Policy not applied to ACT, C, POLCY exception thrown

PLCY2: POLCY is configurable

Description: Error! Reference source not found.
Scope: POLCY
Primary Actor: SYS
Stakeholders: UP, Component C
Preconditions: UP has been initialized, C exists, Actor ACT
Trigger: SYS invokes ACT on C
Postconditions: Policy-appropriate action taken on C
Primary/Success Scenario:

- SYS invokes ACT on C
- Listener defined on C dispatches to POLCY
- POLCY looks up policies for ACT, C
- POLCY determines correct policy for ACT, C
- POLCY is applied by appropriate handler
- Policy-appropriate action taken on C

Secondary/Failure Scenarios: POLCY doesn't receive ACT content, Improper policy is identified, Policy not applied to ACT, C, POLCY exception thrown

PLCY3: POLCY load policies

Description: Service and subsystem policy files will be validated.
Scope: POLCY
Primary Actor: SYS
Stakeholders: UP, Component C
Preconditions: UP has been initialized, C exists, Actor ACT
Trigger: SYS invokes ACT on C
Postconditions: Policy-appropriate action taken on C
Primary/Success Scenario:

- SYS invokes ACT on C
- Listener defined on C dispatches to POLCY
- POLCY looks up policies for ACT, C
- POLCY determines correct policy for ACT, C
- POLCY is applied by appropriate handler
- Policy-appropriate action taken on C

Secondary/Failure Scenarios: POLCY doesn't receive ACT content, Improper policy is identified, Policy not applied to ACT, C, POLCY exception thrown

PLCY4: POLCY store policies

Description: The central Policy Management subsystem shall be context sensitive (component, operation type, system bindings, etc.).
Scope: POLCY
Primary Actor: SYS
Stakeholders: UP, Component C
Preconditions: UP has been initialized, C exists, Actor ACT
Trigger: SYS invokes ACT on C
Postconditions: Policy-appropriate action taken on C
Primary/Success Scenario:

- SYS invokes ACT on C
- Listener defined on C dispatches to POLCY
- POLCY looks up policies for ACT, C
- POLCY determines correct policy for ACT, C
- POLCY is applied by appropriate handler
- Policy-appropriate action taken on C

Secondary/Failure Scenarios: POLCY doesn't receive ACT content, Improper policy is identified, Policy not applied to ACT, C, POLCY exception thrown

PLCY5: POLCY disambiguate, order, select policy for COMP, ACT

Description: The central Policy Management subsystem shall provide a common mechanism for disambiguating, ordering, and selecting policies.
Scope: POLCY
Primary Actor: SYS
Stakeholders: UP, Component C
Preconditions: UP has been initialized, C exists, Actor ACT
Trigger: SYS invokes ACT on C
Postconditions: Policy-appropriate action taken on C
Primary/Success Scenario:

- SYS invokes ACT on C
- Listener defined on C dispatches to POLCY

- POLCY looks up policies for ACT, C
- POLCY determines correct policy for ACT, C
- POLCY is applied by appropriate handler
- Policy-appropriate action taken on C

Secondary/Failure Scenarios: POLCY doesn't receive ACT content, Improper policy is identified, Policy not applied to ACT, C, POLCY exception thrown

PLCY6: SYS handle policy

Description: Each service or subsystem will provide its own policy handlers.
Scope: POLCY
Primary Actor: SYS
Stakeholders: UP, Component C
Preconditions: UP has been initialized, C exists, Actor ACT
Trigger: SYS invokes ACT on C
Postconditions: Policy-appropriate action taken on C
Primary/Success Scenario:

- SYS invokes ACT on C
- Listener defined on C dispatches to POLCY
- POLCY looks up policies for ACT, C
- POLCY determines correct policy for ACT, C
- POLCY is applied by appropriate handler
- Policy-appropriate action taken on C

Secondary/Failure Scenarios: POLCY doesn't receive ACT content, Improper policy is identified, Policy not applied to ACT, C, POLCY exception thrown

PLCY7: SYS select policy enforcement type

Description: Component service and subsystem attributes and behaviors can select policy level control (e.g., persistence is immediate).
Scope: POLCY
Primary Actor: SYS
Stakeholders: UP, Component C
Preconditions: UP has been initialized, C exists, Actor ACT
Trigger: SYS invokes ACT on C
Postconditions: Policy-appropriate action taken on C
Primary/Success Scenario:

- SYS invokes ACT on C
- Listener defined on C dispatches to POLCY
- POLCY looks up policies for ACT, C
- POLCY determines correct policy for ACT, C
- POLCY is applied by appropriate handler
- Policy-appropriate action taken on C

Secondary/Failure Scenarios: POLCY doesn't receive ACT content, Improper policy is identified, Policy not applied to ACT, C, POLCY exception thrown

Appendix B Glossary

Domain Model: A conceptual model for a particular information domain.

Actor: A behavioral building block, implements act method.

Application Generic: A more fundamental, or shared functionality that can be applied across applications.

Application Specific: A more specialized functionality that is not sharable across applications.

Aspect Oriented: An approach to performing logic-based operations that doesn't depend on a particular system and thus can be implemented outside of a system and apply across a framework. Logging and tracing are classic examples of aspect-oriented programming. They are triggered locally, and may have local handling mechanisms, but the general functionality is defined globally.

Baseline Component Functionality: Components and widgets that are required to construct a user interface using MCT.

Brittleness: Software which is specialized and requires modification and recompilation for any modifications.

Build Time: This denotes the integration that takes place when an application is being integrated and packaged for deployment. The use is contrasted to compile time since, in some systems, attributes can be made available at build time or launch time but are not as dynamic as those that cannot be acquired until run time.

Compile Time: This denotes events or changes that require source code modification and hence recompilation.

Component Access: The ability to see a component's fields, facets, notes, and their values.

Component Malleability: The ability to modify a component's fields, facets, notes, and associated values.

Component Model: A conceptual model for MCT components.

Composition: The aggregation of model components through user action, such as drag and drop.

Conceptual Model: The aggregate of conceptual classes, attributes, and relations that represent the structure and function of a particular body of knowledge.

Core Component Functionality: The functionality of the Component Model which enables the construction of adaptive components and simple message-passing capability.

Core Widget: An MCT GUI widget, to distinguish it from an externally-defined widget such as a Swing or SWT widget.

Decoupled: Systems that are designed and constructed to not depend on one another. Changing one has no effect on the other, and vice versa.

Data Model: A conceptual model for a particular information domain.

Domain Model: A conceptual model for a particular information domain.

Facade: An interface that presents part of a functional implementation but not all of it; that which is necessary.

Failover: Allowing some degree of network fault tolerance by shifting load to other resources when a targeted resource fails.

GUI Specific: Pertaining to operations on GUI elements and their model dependencies.

Inferencing: To apply a logic to a set of conditions and infer or deduce an outcome based those conditions being met. These logics can be sequenced, or chained together, to mirror or simulate the way we induce, deduce, explain, plan, and experiment with the world around us.

Lifecycle: The states of a system from creation to destruction. A system that is responsible for component lifecycle is responsible for all the states a component can take during application operation.

Launch Time: This marks the beginning, and duration, of the MCT framework startup sequence.

Load Time: Same as launch time, when the framework is initialized.

Message Passing: A generic mechanism for implementing behavior where the component is provided a message and the message is interpreted at run time.

Model Based: In this context, component model based, which is an approach that has a simple, central model that is mapped to an implementation and thus decouples the generic description from the implementation.

Model Component: A component that has a direct mapping to a domain model value (or values).

M/VC: An architecture that is roughly broken into Model, View, and Controller elements, where the View and Controller elements are more tightly coupled than a Model 2 architecture.

Ontology: A mechanism for defining and describing general conceptual models, their relationships, and their behaviors.

Part: A component whose function contributes to the function of another component.

Plug and Play: A mechanism allowing the addition or removal of external components to be recognized by the environment without formally changing the underlying implementation.

Policy: A contextually-appropriate logical description that can be interpreted/evaluated at run time. For example, for an operation on a component both the operation type, the component type, the applicable fields, the field values, etc. might play an important part in defining how the operation is performed.

Prototype: A predefined component that already satisfies a particular set of roles. A prototype can be used to construct any component. A template is generally used to construct representation components.

RDF/RDFS/OWL: Semantic web languages, all declarative, based on subject-verb-object triples.

Real Time: Operations execute in the amount of time measured on the clock. Most often operation for GUI interfaces is measured in terms of the user's reaction time. Thus, for a GUI, real time operation would mean that all operations are executed while the user is working, without the user having to slow down to wait for the system.

Representation Component: Also known as a User Object, is a combination of a user interface and a model component.

Role: A unit of functional equivalence comprising a set of attributes and behaviors and used to construct component functionality.

Rule Engine: A processing mechanism whose purpose is to support inferencing.

Run Time: This denotes events or operations that are performed dynamically after the framework and application have generally completed the startup sequence.

Semantic Web: Using conceptual models based on RDF/RDFS/OWL to represent content.

Service Oriented: This denotes a system whose operation is transparent to the user of the system through a published or discoverable API. That is, the system is defined by its service and not its implementation.

Structural Model: The aggregate of attributes that represent the structure of a particular body of knowledge.

Template: A partly uninstantiated component that serves as a starting point for constructing component instances.

Triple Store: A repository of RDF/OWL triples.

Use Case: A high-level description of a single behavior or action of a system, which creates a contract between preconditions required for the action and the postconditions that define the success of that action. A use case doesn't define how the behavior is implemented.

Validation Model: A conceptual model for describing and validating component type, value, and range.

Domain Model: A conceptual model for a particular information domain.

Appendix C References

dkdkd

Tools Used in MCT Design and Implementation

Spec: MS Word

Diagrams: Visio, Rational Software Architect 6

UML: Rational Software Architect 6

Planning: MS Project (none yet)

Development: XML Spy, Eclipse 3.2, Jakarta Ant 1.6, CVS

Links and Reference Documents

RuleML Related

- http://www.dfki.uni-kl.de/ruleml/
- Extensible Rule Markup Language (XRML): http://xrml.kaist.ac.kr
- Object Constraint Language (OCL): http://www.csci.csusb.edu/dick/samples/ocl.html
- Bowers, S. and Delcambre, L., "Representing and Transforming Model-Based Information", Oregon Graduate Institute.
- Boley, H., Tabet, S, and Wagner, G., "Design Rationale of RuleML: A Markup Lanaguage for Semantic Web Rules": http://www.di.ufpe.br/~compint/aulas-IAS/artigos/BolyTabetWagnerRuleML.html

Validation

- Validation with Java and XML Schema: http://www.javaworld.com/javaworld/jw-09-2000/jw-0908-validation_p.html

Links to MCT Documents

MCT Forum on Wiki: Index

MCT Java Style Guidelines: JavaStyleConventions

MCT Development Plan: