

# Active Workspace Customization

AW004 • 2.4



# Contents

## Part I: Customization overview

<b>Client application history and considerations</b> . . . . .	<b>1-1</b>
<b>Active Workspace user interface</b> . . . . .	<b>2-1</b>
Pattern-based design . . . . .	2-1
Interface overview . . . . .	2-1
Global navigation toolbar . . . . .	2-2
Location . . . . .	2-2
Active Workspace back button . . . . .	2-3
Context control . . . . .	2-4
Global search . . . . .	2-5
Sublocations and primary navigation tabs . . . . .	2-5
Sublocation content . . . . .	2-6
Work area toolbar . . . . .	2-6
Work area header . . . . .	2-7
Primary work area . . . . .	2-7
Navigation command set . . . . .	2-8
Secondary work area . . . . .	2-9
Secondary navigation tabs . . . . .	2-10
Tools and information command set . . . . .	2-10
Tools and information panel . . . . .	2-11
<b>Common coding patterns</b> . . . . .	<b>3-1</b>
Code base consistency . . . . .	3-1
Logging . . . . .	3-1
Resources . . . . .	3-3
API visibility . . . . .	3-4
Message reporting . . . . .	3-5
<b>Active Workspace extensibility</b> . . . . .	<b>4-1</b>
<b>Framework architecture</b> . . . . .	<b>5-1</b>
Model-View-Presenter (MVP) . . . . .	5-1
Dependency injection . . . . .	5-2
Extensibility . . . . .	5-4
Component architecture . . . . .	5-5
Component contracts . . . . .	5-6
Managing components . . . . .	5-8
<b>View model and data binding</b> . . . . .	<b>6-1</b>

Model-View-Presenter (MVP)	6-1
Active Workspace client MVP elements	6-1
Repetitive code patterns	6-2
Data binder	6-2
Data binding framework	6-2
Properties in the view and view model	6-4
Collections in view and view model	6-6
View model	6-6
View	6-8
Presenter	6-11
<b>Service-oriented architecture (SOA)</b>	<b>7-1</b>
SOA goals	7-1
Framework support for REST services	7-2
Programming model	7-2
AsyncCallback implementation best practice	7-3
Code autogeneration and integration into the build system	7-4
Client data model	7-5
Object property policy	7-5
SOA checklist	7-8
<b>Client data model and the meta system</b>	<b>8-1</b>
Client data model	8-1
Programmer-friendly interfaces	8-1
Caching	8-2
Events	8-2
<b>Active Workspace hosting</b>	<b>9-1</b>
<b>FTSIndexer customization</b>	<b>10-1</b>
Overview of indexer customization	10-1
Indexer customization prerequisites	10-1
Further information	10-2

## Part II: Customization examples

<b>Simple examples</b>	<b>11-1</b>
Simple example overview	11-1
Configuring the home page	11-1
Overview of the home page configuration	11-1
Reset the home page	11-2
Protect a tile	11-5
Hide a tile	11-7
Create a new collection	11-8
Add a tile to a collection	11-8
Create a new tile type	11-9
Create a tile template that creates a Part	11-11
Action styles	11-13

Theme index	11-13
Tile sizes	11-13
Provided icons	11-14
Configuring page layout using style sheets	11-15
Introduction to using XML rendering templates (XRT) with Active Workspace	11-15
Considerations for using XRTs in Active Workspace	11-16
Configure the information panel using XRTs	11-17
Active Workspace-specific style sheets	11-18
Modular style sheets	11-19
Working with HTML panels in XRT	11-20
<b>Examples using code scaffolding</b>	<b>12-1</b>
Code scaffolding overview	12-1
Use generateModule to create a new module	12-1
Creating custom themes	12-4
Cascading style sheets (CSS) in Active Workspace	12-4
Custom theme overview	12-6
Theme CSS classes	12-8
Edit the CSS live	12-8
Add a new theme to your module	12-9
Contributing commands	12-9
Command contribution constructs	12-9
Command types	12-11
One-step commands	12-12
AbstractCommandHandler base class	12-12
GIN binding command handlers	12-13
Example: command to launch a web page	12-14
Add a new one-step command to your module	12-16
Type icons	12-17
Type icon overview	12-17
Add a new type icon to your module	12-17
Locations and sublocations	12-18
Location and sublocation overview	12-18
Add a new location or sublocation to your module	12-18
Navigation panel	12-19
Navigation panel overview	12-19
Add a new navigation panel to your module	12-21
Tools and information panel	12-22
Tools and information panel overview	12-22
Add a new tools and information panel to your module	12-24
<b>Using property widgets</b>	<b>13-1</b>
Property widget overview	13-1
Common widget features	13-1
Standard Active Workspace widgets	13-2
StringTextBoxWidget	13-2
StringTextAreaWidget	13-2
LabelWidget	13-2
BooleanCheckBoxWidget	13-2

BooleanRadioButtonWidget . . . . .	13-3
BooleanToggleButtonWidget . . . . .	13-3
IntegerTextBoxWidget . . . . .	13-3
DoubleTextBoxWidget . . . . .	13-3
ObjectLinkPropertyWidget . . . . .	13-3
DateWidget . . . . .	13-4
Property widget examples . . . . .	13-5
Property widget examples overview . . . . .	13-5
Add property widgets . . . . .	13-7
Define the view . . . . .	13-10
Provide data binding to property widgets . . . . .	13-11

## Figures

---

Selecting a user's tile collection . . . . .	11-3
Selecting a tile in a user's tile collection . . . . .	11-4
Repin a tile . . . . .	11-5
Expanding a user's tile collection . . . . .	11-6
Protect a tile . . . . .	11-7
Searching for Active Workspace style sheet preferences in the rich client . . . . .	11-15
Searching for style sheet files in the rich client . . . . .	11-16

## **Part I: Customization overview**





# Chapter 1: Client application history and considerations

Two technology trends revolutionized client applications and application delivery to consumers.

- The use of tablet and touch-based devices.
- The emergence of HTML5 and CSS as a viable cross-platform, cross-browser client platform.

The tablet market introduced new operating systems, such as iOS, Android, Windows 8, as well as new application development technologies and environments. These technologies bring a new focus on usability and a variety of new development paradigms that cross these platforms. Application developers must build the same application using multiple, different technologies, and many are turning to **HTML5** and CSS3.

- Contemporary mobile platforms and browsers support HTML5-based rich applications on desktop platforms and mobile devices.
- The Active Workspace client framework leverages these technologies to deliver superior user experience to you on your platform of choice.

## Usability

The Active Workspace client framework presents Teamcenter and its applications in an intuitive user interface, rather than the traditional interfaces and applications that targeted expert users.

Active Workspace explicitly reduces exposure of complexity and, as a result, presents an uncluttered graphical user interface (GUI) that is engaging and fluid.

Applications typically face challenges in defining user interaction models.

- When an interface is defined for a casual or infrequent user, the experienced user may find the interface slow or inefficient.
- When the interface is optimized for the sophisticated user, the casual user finds it complex and overwhelming.

The goal of Active Workspace is an intuitive interface for all levels of users.

When you customize Active Workspace, keep this in mind as you create your new content.

## Performance

Many factors contribute to application performance, and Active Workspace provides the best possible performance to the end user.

- The Active Workspace framework provides efficient service-oriented architecture (SOA) APIs that use a JavaScript Object Notation (JSON) payload.

- The Active Workspace client is optimized to ensure individual user gestures generate as few server calls as possible—in most cases one or none. This makes the client very tolerant of latency operating over a wide-area network (WAN).
- The Active Workspace client is designed to work with the Teamcenter File Management System (FMS) to bring the same level of efficiency that FMS provides to installed clients in the HTML5 environment.

## Deployment

The Active Workspace client eases deployment of PLM clients by requiring no installation or plug-ins. This reduces cost of ownership, and lets you rapidly deploy new versions of the software as needed.

## Mobile readiness

Active Workspace is touch-friendly to enable mobile platforms. Be certain to test your customizations on a variety of platforms, including tablets and other portable or touch-enabled devices.

## Configurability

Teamcenter is deployed in many industries and configured to meet the needs of specific enterprises.

The Active Workspace user interface is designed to be:

- Sensitive to the needs of different user groups, roles, projects, and programs.
- Configurable, extensible, and modular.

You can deploy only the applications and capabilities that you need. You can configure the graphical user interface (GUI) to meet your needs using the extensibility built into the Teamcenter platform, such as Teamcenter XML rendering templates (XRTs) and the client framework. You can also add new components to the system using the published extension mechanisms and API.

## Common User Experience

In addition to primary clients, Teamcenter has many client integrations. You can bring Teamcenter to users in a familiar application.

Active Workspace components can be embedded in many applications with full support for seamless interoperability. This provides a consistent user interface across integrations and applications and supports users by providing Teamcenter functionality in commonly used applications.

## Technology and GWT

Active Workspace uses HTML5, CSS3 and JavaScript. This lets you build on mature, open standards that are robust, flexible, and that work across all supported platforms.

A goal of Active Workspace is to provide a foundation for a very large enterprise application, and to support well-defined, published contracts with a clear deprecation policy. These and other considerations led to the use of Google Web Toolkit (GWT) as the underlying technology for Active Workspace.

The GWT SDK provides a set of core Java APIs and web-based user interface (UI) widgets. These allow you to write AJAX applications in Java and then compile the source to optimized HTML5, CSS3, and JavaScript that runs across all supported browsers.

At the core of GWT are two fundamental capabilities:

- A cross-language compiler that takes code written in Java and compiles it into an optimized JavaScript code.

The compiler provides hooks so client framework developers can add extension points through a mechanism called *deferred binding*. *Deferred binding* supports advanced use cases by providing autogeneration and code replacement based on considerations such as platform type.

- A development mode that supports live debugging of the Java code while it is running in the browser.



## Chapter 2: Active Workspace user interface

### Pattern-based design

Active Workspace presents a crisp, clear, intuitive user experience. The graphical user interface (GUI) focuses on users' needs and leverages the paradigm shift introduced by touch devices, while still supporting those using a pointing device. Above all, the Active Workspace user interface (UI) provides a consistent user experience across all different application areas.

To achieve these goals, Active Workspace uses a *pattern-based* design, similar to the approach commonly used in software architecture. Based on experience and needs, a set of patterns are defined and refined for various user scenarios. An inventory of patterns is created and guidelines are developed for when to use these patterns.

UI patterns let Active Workspace developers build a compliant UI using common components.

Patterns apply in various categories:

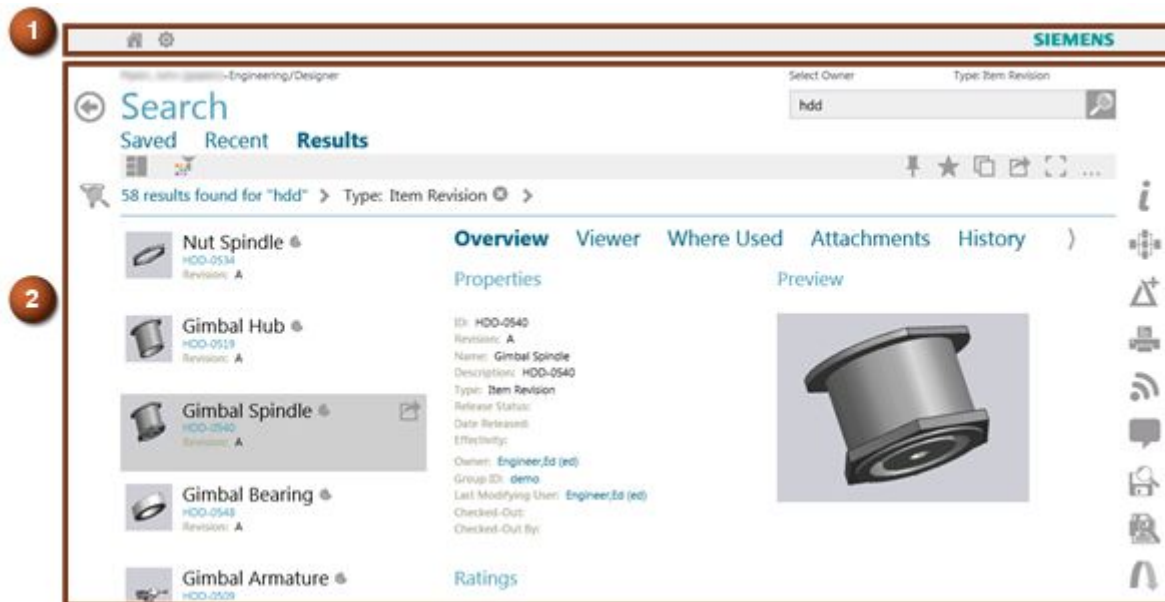
- Some patterns *apply across all applications*, such as the defining layout of the application and the global toolbar.
- Some patterns *interact with data and across multiple domains*, such as patterns for objects shown as titles in a list view and XRT.
- Some patterns are specific to a use case, such as the application landing page (default start page).

#### Note

Typically patterns specific to a use case provide few opportunities for reuse but are still valuable for the application of standard practices to create a consistent user experience.

### Interface overview

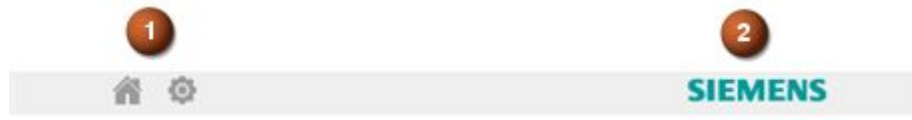
The Active Workspace interface consists of two main areas.



1. *Global navigation toolbar*
2. *Location*

## Global navigation toolbar

The global navigation toolbar is present on all pages of the GUI. It has two areas.



1. *Global navigation buttons* are commands that are common and useful across all the pages in the GUI.
2. The *logo* area displays branding information.

### Note

Below the toolbar, a progress indicator is displayed when Active Workspace communicates with the Teamcenter server.

## Location

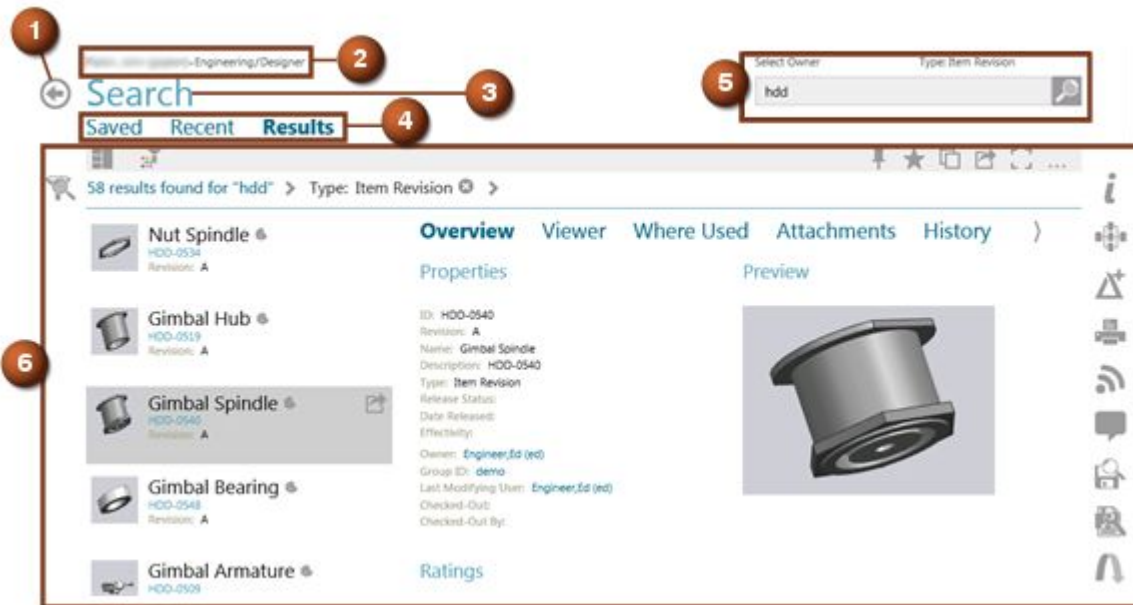
A *location* defines a page that supports a set of closely related functions and workflows. Each location includes the following:

- A title that provides a page name
- A unique **NameToken** identifier

**Note**

If a location allows contributions, this identifier is published.

- One or more sublocations



1. Active Workspace back button
2. *Context control*
3. Name of the current *location*
4. *Primary navigation tabs*
5. *Global search*
6. *Sublocation*

## Active Workspace back button

All locations have an Active Workspace back button that allows users to move to previously visited locations such as **Search**, **My Changes**, or an open object.



The Active Workspace back button differs from a browser back button. A browser back button moves through each URL that was displayed in the browser address bar, while the Active Workspace button moves to the previously visited location but not tabs. The behavior of the Active Workspace back button allows a user to quickly navigate from **My Changes** to a target object and back, irrespective of the intermediate steps they may have taken to look at various tabs of information on the target object.

#### Note

Active Workspace includes the active tab within a location and can include information such as browser address bar URL content. This supports browser refresh and URL sharing with little or no page changes.

## Context control

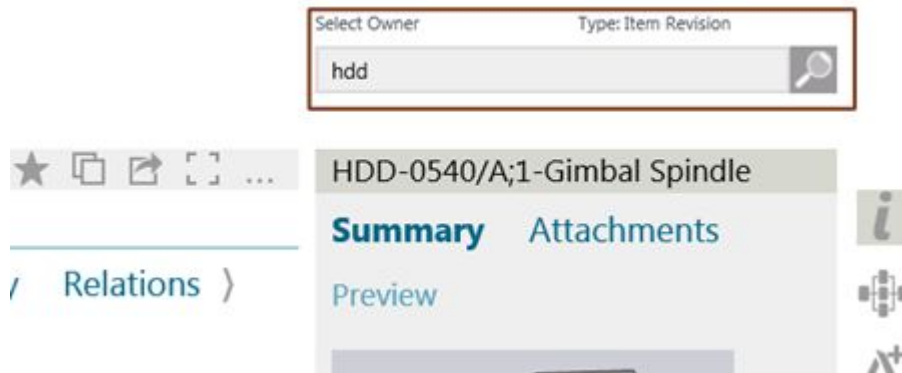
A *context control* is present on every location in the GUI. It allows the user to view their profile, log out, and change context information: current project or program, group and role, and the revision rule for selecting the specific revisions.





## Global search

The search box is present on all locations for full-text searches. The user can enter any search string and perform a search. Performing a search changes the location to the **Search** location and presents the objects that meet the search criteria.



## Sublocations and primary navigation tabs

A *sublocation* defines the content of the location and how it is presented. Sublocation names are presented as the *primary navigation tabs*.



- Users navigate between sublocations by using the *primary navigation tabs*.
- When a location has only one sublocation, the tabs are not displayed.

When the location is a Teamcenter business object (part, document, or change, for example), the sublocation tabs are defined by the object's XML rendering template (XRT).

Each sublocation has a unique URL. The URL can be used to navigate to the sublocation. The component contributing the location and sublocation defines this URL.

### Note

It is a best practice to use a *namespace* for these URL fragments to prevent collisions across contributions from multiple, independent components.

## Sublocation content

Each sublocation has a unique **NameToken** identifier used to contribute its content.

Content can be contributed by either of the following:

- The component that defines the sublocation can contribute content to the sublocation.
- Other components can contribute content to the sublocation using the published identifier.

In either case, each sublocation defines its own content that is unique to that sublocation.



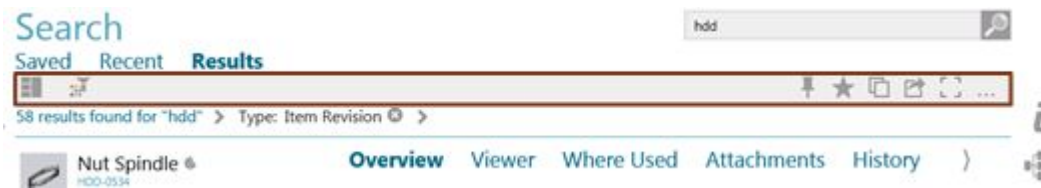
1. Work area toolbar
2. Work area header
3. Primary work area
4. Navigation command set
5. Secondary work area
6. Secondary navigation tabs
7. Tools and information command set
8. Tools and information panel

### Note

This panel may be visible or hidden.

## Work area toolbar

The work area toolbar contains tools that operate on the content in the work area, such as refresh, add to favorites, and pin. The toolbar contains work area displays on the left and work area one-step commands on the right.



*Work area displays* change the display of the content in the work area. For example, the work area displays show the content as a list or table.

*Work area one-step commands* are commands that operate on selections within the work area.

- They do not require input from the user.
- They are aware of context and selection and are enabled and displayed based on those conditions.

Components can contribute to work area displays and work area one-step commands.

- Commands can be reused and contributed to more than one sublocation.
- Command contributions must be constructed, so they are enabled and visible only as needed.
- Each command contribution specifies an ordering priority to express its position. The convention used to specify the ordering priorities includes gaps, so other commands can be introduced between existing commands without changing priorities.

## Work area header

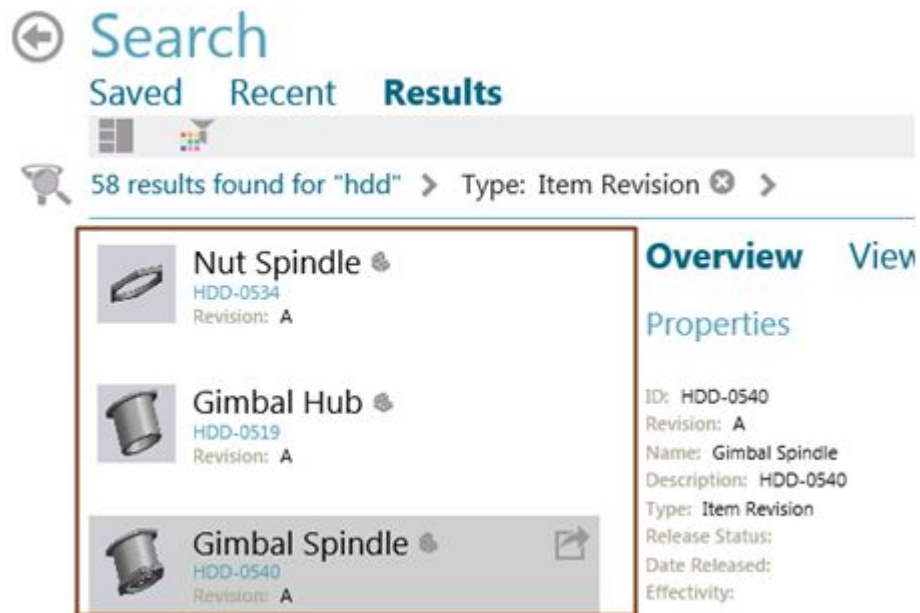
The work area header is displayed immediately below the work area toolbar.



This header is used for summary information such as the number of results a search has found. It also displays the breadcrumb, which is used as an additional means of refining what is displayed.

## Primary work area

The *primary work area* contains the rendered main content for the sublocation.



The primary work area is defined by the sublocation and cannot be replaced by other components.

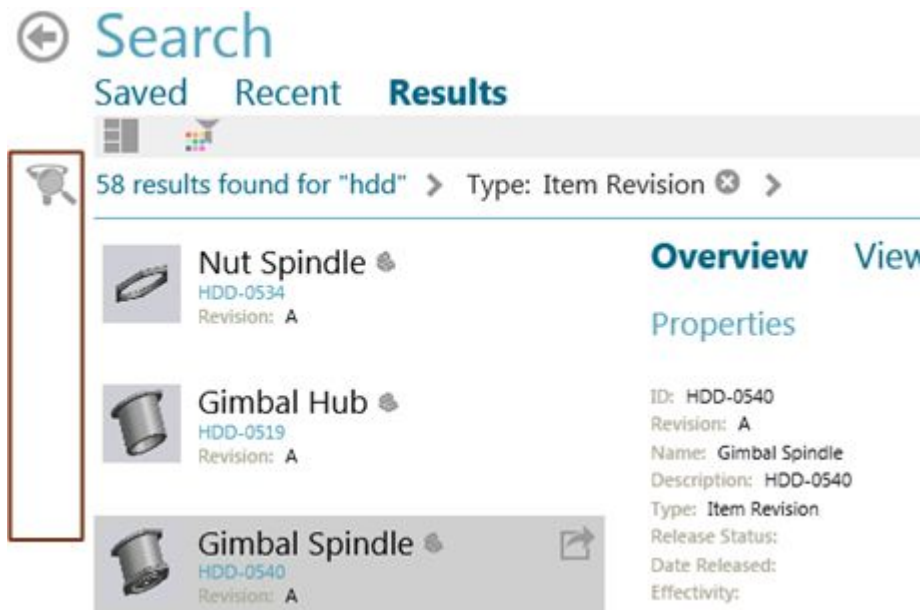
- A component can define how content is rendered in the primary work area.
- Siemens PLM recommends that customizers use the common widgets provided by the framework.

**Note**

Components and work areas can provide extension points that other components can register and contribute to.

## Navigation command set

Commands in the navigation command set apply to content in the primary and secondary work areas or apply filters to the content in these work areas.

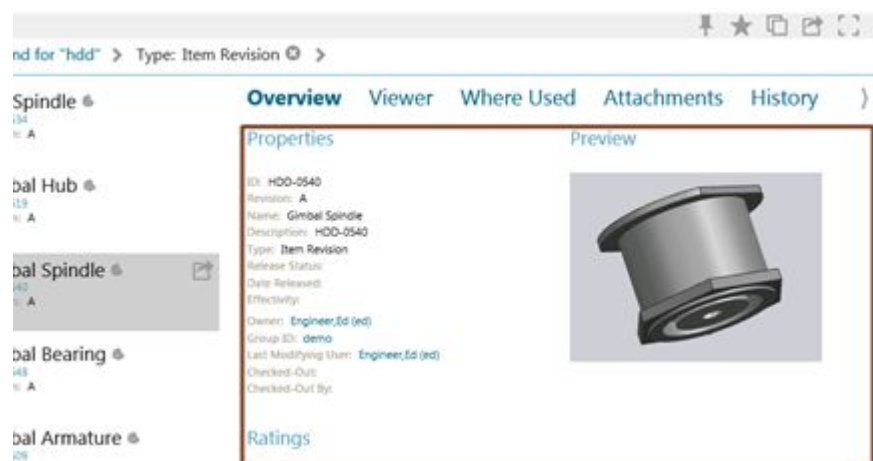


Components can contribute a navigation command to the navigation command set.

- When there are no contributions, the navigation command set is hidden.
- Contributions are registered to the sublocation unique **NameToken** identifier.
- Order priority is supported.

## Secondary work area

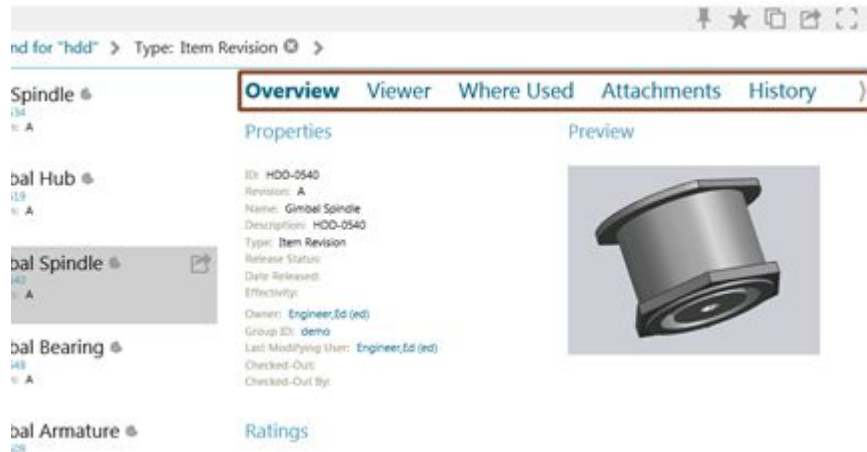
The *secondary work area* typically shows the details of the content selected in the primary work area.



Other components can use the sublocation unique **NameToken** identifier to make contributions. There can be zero or more contributions to a sublocation secondary work area.

## Secondary navigation tabs

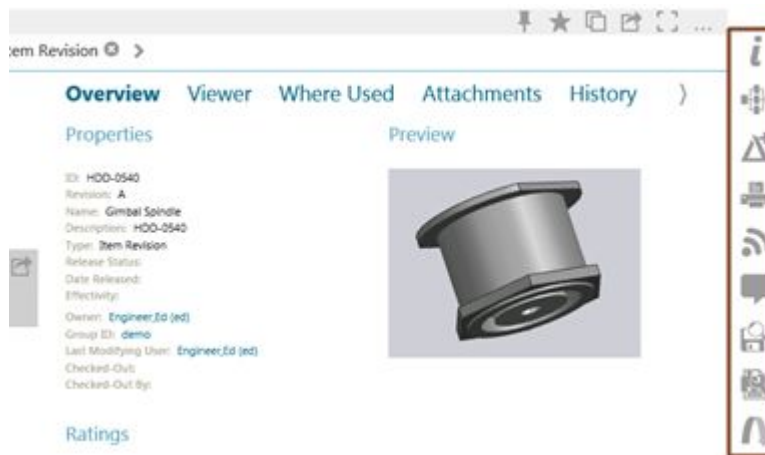
When multiple secondary work areas exist for a sublocation, they are shown as *secondary navigation tabs* in the graphic user interface.



Priority ordering is defined by the contributors. It is a best practice to leave gaps.

## Tools and information command set

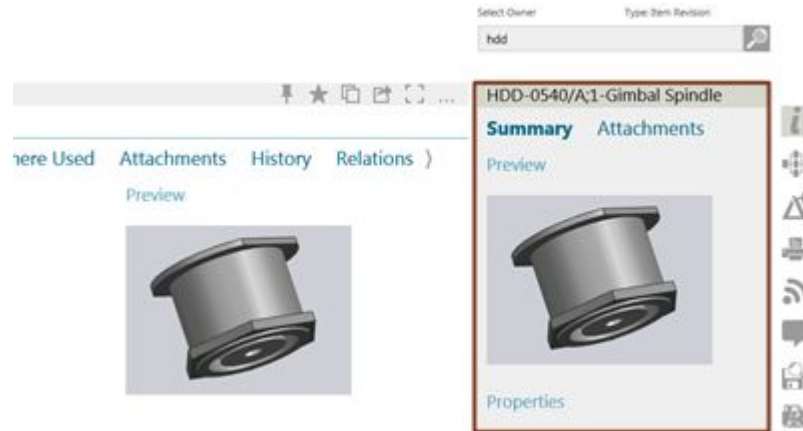
Commands in the tools and information command set operate on the content in the primary or secondary work areas and require user input.



- Components can contribute to the tools and information command set.
- When there are no contributions, the tools and information command set is hidden.
- Contributions are registered against the sublocation unique **NameToken** identifier.
- Order priority is supported.

## Tools and information panel

This panel appears when a button from the tools and information command set is clicked. It provides an area for information to be exchanged with the user.



Unlike one-step commands, many of the tools and information commands require user interaction, such as **Save As or Revise**, **Create Change**, or **Print**.





## Chapter 3: Common coding patterns

### Code base consistency

The Active Workspace client framework provides common utilities and patterns for basic capabilities that all client code leverages.

To ensure consistency across the client code base, follow the best practices for logging, resources, API visibility, and message reporting.

For information about internationalization (I18N), see the following references:

- <http://www.gwtproject.org/doc/latest/DevGuideI18n.html>
- <http://www.gwtproject.org/doc/latest/DevGuideUiBinderI18n.html>

### Logging

*Logging* is the process of recording events in an application to understand how the application runs and to diagnose problems. Logging results in an audit trail that makes it easier to troubleshoot issues encountered by developers and users.

You can log the following events:

- Entering and exiting of important functions. This is verbose, so log with the finest level.
- Important forks in application logic.
- All warnings and error messages.

For reference information about logging, see the following references:

- <http://docs.oracle.com/javase/7/docs/technotes/guides/logging/overview.html>
- <http://www.gwtproject.org/doc/latest/DevGuideLogging.html>

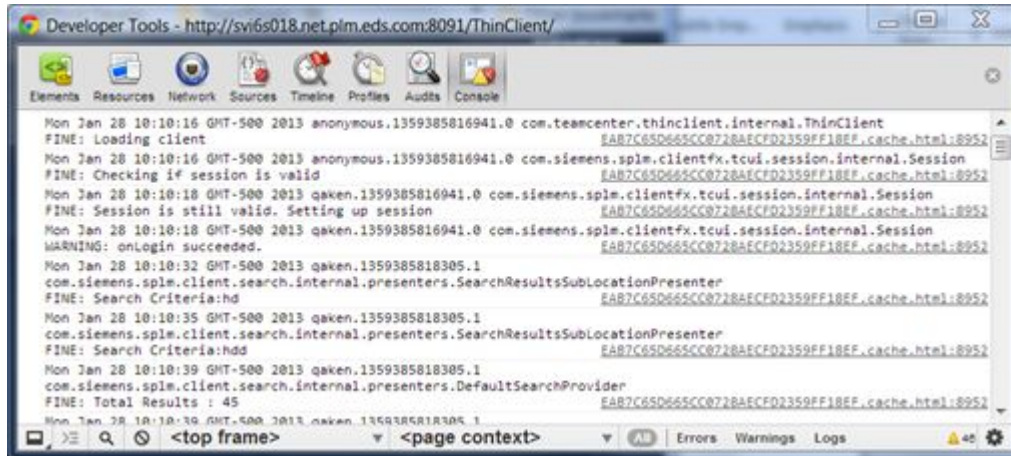
Logging in the client follows the guidelines published in the *GWT Development Guide for Logging*. The top-level application component controls the log levels and handlers for logging.

The relevant entries in the GWT module file, **com.teamcenter.thinclient.ThinClient.gwt.xml**, are:

```
<!-- Logging configuration -->
<!-- To change the default logLevel -->
<set-property name="gwt.logging.logLevel" value="FINE" />
<!-- To enable/disable logging -->
<set-property name="gwt.logging.enabled" value="TRUE" />
<!-- Disable the console handler -->
<set-property name="gwt.logging.consoleHandler" value="ENABLED"/>
<set-property name="gwt.logging.firebugHandler" value="ENABLED" />
<set-property name="gwt.logging.popupHandler" value="DISABLED" />
```

The preceding configuration enables logging and sets the log level to **FINE**. The logging is output to the JavaScript Console of the browser.

The following image shows the logging information in Chrome JavaScript Console.



Log level	Description
<b>SEVERE</b>	Indicates a serious failure.  <b>SEVERE</b> messages describe important events that prevent normal program operation. These should be meaningful to end users and to system administrators.
<b>WARNING</b>	Indicates a potential problem.  <b>WARNING</b> messages describe events of interest to end users or system managers, or which indicate potential problems.
<b>INFO</b>	Informational messages are typically written to the console or its equivalent. Use for significant messages for end users and system administrators.
<b>CONFIG</b>	Provides static configuration information to assist in debugging problems that may be associated with particular configurations. For example, a <b>CONFIG</b> message can include the CPU type, the graphics depth, the GUI look-and-feel, and so forth.
<b>FINE</b>	Provides general information for developers who do not have a specialized interest in the specific subsystem.  Messages generally include minor (recoverable) failures.  Issues indicating potential performance problems are also logged as <b>FINE</b> .
<b>FINER</b>	Logs calls for entering, returning, or throwing an exception.
<b>FINEST</b>	Provides a very detailed tracing message.

**Note**

**FINE**, **FINER**, and **FINEST** are intended for relatively detailed tracing. The output at the three levels can vary between subsystems.

Use **FINEST** for voluminous detailed output, **FINER** for less detailed output, and **FINE** for the fewest and most important messages.

- At run time, users can reduce, but not increase, log levels by adding the **logLevel=<level>** query parameter to the URL, as shown.

```
http://<...>/ThinClient/ThinClient.html?logLevel=SEVERE
```

- If the log level in the **gwt.xml** file is set to **FINE**, you cannot log **FINER** messages by setting the **logLevel** to **FINER** in the URL. The URL parameter can reduce the logging level to log less to the console.

The framework provides an additional logging module in the **com.siemens.splm.clientfx.base** project.

This module is used as part of the framework and enhances logging in the client as follows:

- Manages the **Correlation Id** value. **Correlation Id**:
  - Changes when the history token is changed by navigation to a different page or when a command is run.
  - Is an identifier used to associate log entries that relate to a logical unit of activity, such as the client, middle tier, and server.
- Provides a log formatter that adds the **Correlation Id** value to all log messages output.

## Resources

Resources such as images and text files are commonly used in modules. These resources are brought into the application by the **ClientBundle** functionality provided by GWT.

For information about **ClientBundle** functionality, see

<https://developers.google.com/web-toolkit/doc/latest/DevGuideClientBundle>.

For clients, place all resources in the **resources** subpackage of the GWT module.

- If the package name for your module is *com.siemens.splm.clientfx.xyz*, create a package named *com.siemens.splm.clientfx.xyz.resources* to contain the resources for your GWT module.
- Create separate subpackages under **resources** for **images** and **i18n** resources.
- Add the **resources** folder to the source path in the GWT module's XML file, **XYZ.gwt.xml**.

```
<source path="resources"/>
```

The **ClientBundle** class, which provides the module with the information it needs to access the resources, must be in the GWT module's internal package because the resources are typically private to the GWT module.

## Command icons

Command icons are monochromatic to avoid distracting from the content.

- The style is flat without shadows.
- Typical icons use RGB **939393** to work well with various light and dark backgrounds.
- Image size varies by command type.

Command type	Width and height (pixels)
Global toolbar command	18 x 18
Navigation command	32 x 32
One step command	22 x 22
Display command	22 x 22
Tool command	32 x 32

### Note

Currently, there is no support for different icons per theme color.

## API visibility

Modules publish APIs for other modules to consume.

- All classes in an *internal* package are internal to the GWT module and must not be referenced outside the GWT module.

As a best practice, components must expose only the necessary API in published packages; this is the component contract. The component contract must contain only the limited set of interfaces that the component authors expect consumers of the component to use.

- All classes in the *published* package are published based on the **ApiVisibility** annotation on the class or interface and its methods.

Maturity	Publish scope	Description
Experimental	Internal	The API is experimental and internal to Siemens PLM Development and must not be used by modules that are not authored by Siemens PLM Development.
Mature	Internal	The API is mature and internal to Siemens PLM Development and must not be used by modules that are not authored by Siemens PLM Development.

<b>Maturity</b>	<b>Publish scope</b>	<b>Description</b>
Experimental	Public	<p>The API is experimental and public. It can be used outside the GWT module by any internal, third-party, or customer module.</p> <p>Because the API is still being developed, there is no deprecation and the API can be modified or removed in the next build/version. A replacement may not be provided for the API. Experimental APIs that are removed or modified are listed in release notes.</p> <p><i>Do not</i> use an experimental API for production.</p>
Mature	Public	<p>The API is mature and public. It can be used outside the GWT module by any internal, third-party, or customer module.</p> <p>The API will not change. If changes are needed, the existing API is deprecated and an alternative API is identified. The alternative becomes available in the same version in which the API is deprecated.</p> <p>Release notes specify the time frame for removal of deprecated API. The deprecation is typically two major releases of the client.</p>

## Message reporting

The framework provides classes and functionality to support messages that inform the user that an action has occurred. The messages include the following:

- Error reporting, such as the partial or complete failure of a network call.
- Notification reporting, such as successful modification of an object state for a name change.
- Notification reporting, such as requesting user confirmation before proceeding with a delete operation.

Use the following guide to determine when to use logging, error reporting, or notification reporting.

<b>Report</b>	<b>Audience</b>	<b>Recommendation</b>
Logging	System administrator	No translation or localization is required; this is internal information.
Notification	Client user	The message should provide specific, detailed, and localized information about expected results or should prompt the user for confirmation prior to running some action.

Report	Audience	Recommendation
Error	Client user	<p>The message should provide specific, detailed, and localized information to convey a user-related error, rather than internal details.</p> <p>The preferred form of an error message is:</p> <p><i>object</i> could not be <i>action</i> because <i>reason</i>.</p> <p><i>object</i>, <i>action</i>, and <i>reason</i> should be put into terms that the user would recognize, not data model or process names.</p> <p>The following message does not convey useful information to the client users:</p> <p>Call to SOA XYZ failed with status 3.</p> <p>A more useful message would be:</p> <p><b>Document123</b> could not be <b>checked out</b> because of <b>a network problem</b>.</p> <p>More detailed information about SOA and error codes should be logged. These can also be made available to the user as additional details.</p>

The classes and interfaces provided by the framework for error and notification reporting services are located in the **:com.siemens.splm.clientfx.base.published** package.

### General usage

In the simplest case, error reporting can be accomplished using **IMessageService.error(String method)**.

```

/** Message Service */
@Inject
private IMessageService m_messageService;

// Check to see if this name is already in use.
SearchUtil.getSavedSearches( m_fullTextSearchService, new AsyncCallback<>()
{
    @Override
    public void onFailure( Throwable caught )
    {
        m_messageService.error(SearchMessages.INSTANCE.getSavedSearchesFailed(), caught );
    }
    @Override
    public void onSuccess( List<SavedSearch> result )
    {
        //...
    }
} );

```

The following overrides are also available:

```

public void report( IMessageContext ec, IMessageHandlerComplete cb )

```

## IMessageContext

The **IMessageContext** interface is implemented by the **MessageContext** concrete class and provides context information for the message being broadcast. A **MessageContext** instance contains data for the following content.

Content	Required or optional	Description
<b>severity</b>	optional: error by default	Uses the Java logging level.
<b>message</b>	required	A brief message describes the error.
<b>showModal</b>	optional: nonmodal by default	Treat as a modal dialog, and block other actions until dismissed.
<b>detailedMessage</b>	optional	A detailed message describes the error.
<b>exception</b>	optional	Exception of type <b>Throwable</b> generated when the error occurs.
<b>navigationOptions</b>	optional	Uses <b>list&lt;NavigationOption&gt;</b> to represent button choices presented to the user when the error is broadcast.

To define the **navigationOptions** object for notification or prompt reporting, use the following two sets of keys to configure a dialog box.

- **OkayCancelKeys**  
Display **OK** and **Cancel** buttons in the message dialog box.
- **AbortRetryKeys**  
Display **Abort** and **Retry** buttons in the message dialog box.

For more information, see the example in *IMessageHandlerComplete*.

## IMessageHandlerComplete

The **IMessageHandlerComplete** interface defines a callback invoked when the error dialog is dismissed, such as when the user clicks **OK** or **Cancel**.

This interface requires the implementation of a single method:

```
void handled( IMessageContext notificationContext, Object
selectedNavigationOption );
```

The **IMessageContext** object is passed back with the **NavigationOption** object when the error dialog is dismissed and with any state passed by:

- **reportError( IMessageContext ec**
- **IMessageHandlerComplete cb**
- **Object state**

The following example shows how **IMessageHandlerComplete** can be used:

```
void testErrorContext()
{
    IMessageContext mc = getMessageService().createMessageContext();
```

```

mc.init( Level.SEVERE, "message", false, "detailed info" ); //$NON-NLS-1$ //$NON-NLS-2$

getMessageService().report( mc, new IMessageHandlerComplete()
{
    @Override
    public void errorHandled( IMessageContext mc, Object selectedNavigationOption )
    {
        assert state.equals( "State" ); //$NON-NLS-1$
        assert selectedNavigationOption == null;

        Window.alert( "Error Done" ); //$NON-NLS-1$
    }
} );
}

```

## Prompting

The message service can also be used to prompt the user for confirmation before proceeding with an action.

This type of prompt notification can be accomplished as follows:

```

getMessageService().prompt( mc, new IMessageHandlerComplete()
{
    @Override
    public void notificationHandled( IMessageContext ec, Object selectedNavigationOption,
                                    Object state )
    {
        if( selectedNavigationOption != null )
        {
            NavigationOption navigationOption = (NavigationOption) selectedNavigationOption;
            if( (OkayCancelKeys) navigationOption.getKey() == OkayCancelKeys.OK )
            {
                IOperationsInjector.INSTANCE.getOperationManager().schedule( operation );
            }
        }
    }
} );

```

### Note

You must create the **IMessageContext** interface containing information about the prompt and provide an **IMessageHandlerComplete** callback that lets you take some action based on the user response to the prompt.

The following overrides are available for notify and prompt:

```

/**
 * Reports a notification to the user and prompts them for feedback via navigation
 * options specified in the MessageContext.
 *
 * @param notificationContext - notification context information
 * @param callback - callback to notify when notification has been accepted by user.
 *                  Can be null if no completion status is required.
 */
void prompt( IMessageContext notificationContext, IMessageHandlerComplete callback );

/**
 * Reports an notification to the user. This is used to provide a "toast-like" notification message.
 *
 * @param message - message to display. No options are displayed to user.
 */
void notify( String message );

```



## Chapter 4: Active Workspace extensibility

Active Workspace extensibility provides a modular basis for extending capability within Active Workspace user interface (UI) paradigms. Active Workspace extensibility is implemented on top of **Google's Dependency Injection (GIN)** for GWT and uses a subset of Guice binding language.

*Dependency injection* creates and implements dependent objects and makes it easier to test. Dependent objects get dynamically injected by the framework.

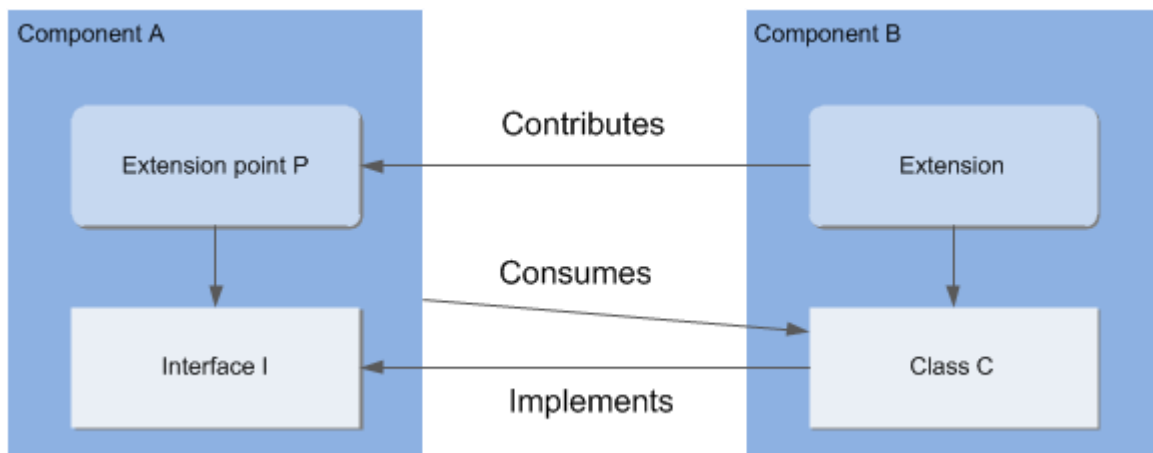
Client extensibility involves:

- *Extension points*

Hook points *declared* by a component for extensibility.

- *Extension*

A *contribution* made by a component to an extension point defined by the same or another component. Think of an electric plug on an electric appliance that plugs in to the electric socket in the wall.



### Helper classes

Various *helper* classes exist for contributing extensions to extension points. The classes are exposed in the published package for a component. Most helper classes follow the form: `[ExtensionPointName]ExtensionPointHelper`. For example, `TypeIconExtensionPointHelper` is a helper utility class used to contribute custom type icons to the `TypeIconModuleRegistry` class.

### Config module

All extensibility is configured in the component **config** module **configure()** method.

#### Note

The **config** module is often a subclass of **AbstractGinModule**.

```

package com.siemens.splm.clientfx.base.internal.config;

import com.google.gwt.inject.client.AbstractGinModule;

/**
 * Base Module Configurations
 */
public class FxBaseGinModule
    extends AbstractGinModule
{
    @Override
    protected void configure()
    {
        //
    }
}

```

The **config** module maps interfaces to implementations for presenters, views, commands, and sublocations.

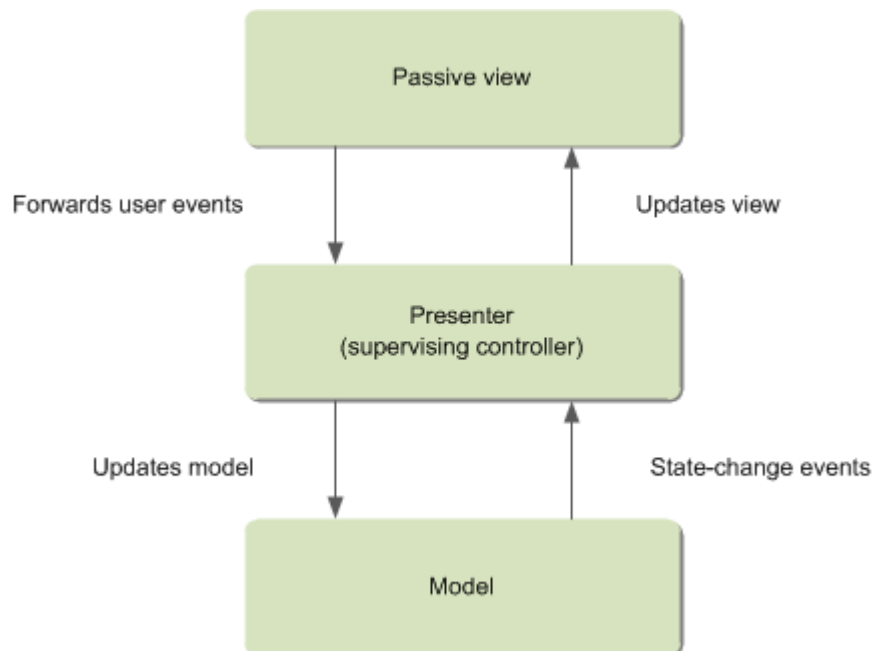
- Dependencies are bound together.
- Extension points are *declared*.
- Extensions are *contributed* to extension points.

## Chapter 5: Framework architecture

### Model-View-Presenter (MVP)

With any large scale user interface (UI) application you need to set up a pattern to provide a workable separation of concerns (SOC). Without this, the code becomes complex and hard to manage and maintain. MVP is one of the many design patterns that are available to help model the UI in an application.

The Active Workspace client framework uses MVP as its core design pattern to handle the UI.



- The *model* encapsulates all the data as well as the methods to query or change the data. It provides notifications when the state in the model changes.
- The *view* contains the UI components that display the content of the model for the user and provides the required user interaction controls.  
There is no business logic within the view. It forwards any user actions from the UI components to the presenter to handle.
- The *presenter* contains all the logic for the application.  
The presenter controls the life cycle of the view and handles the user interaction events from the controls within the view.

A number of frameworks are available that provide an MVP framework on top of GWT.

Articles that provide a detailed insight into the pattern with examples include:

- <http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>
- <http://www.gwtproject.org/articles/mvp-architecture.html>
- <http://www.gwtproject.org/articles/mvp-architecture-2.html>

## Dependency injection

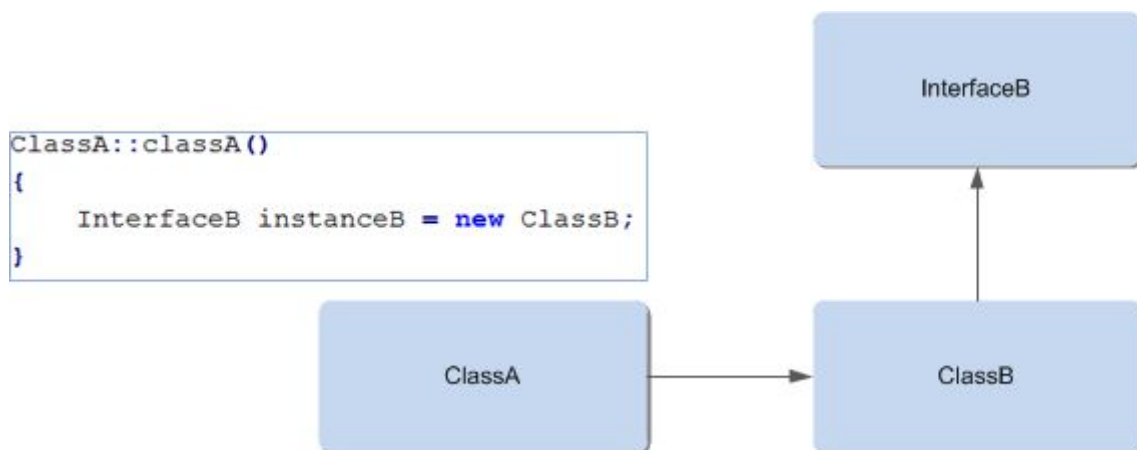
Software is composed of many objects that interact with each other to provide the needed behavior.

Many ways exist for one object to find or use another object:

### Instantiate the dependent object and use it

This results in very tight coupling as it adds dependencies on concrete implementation classes.

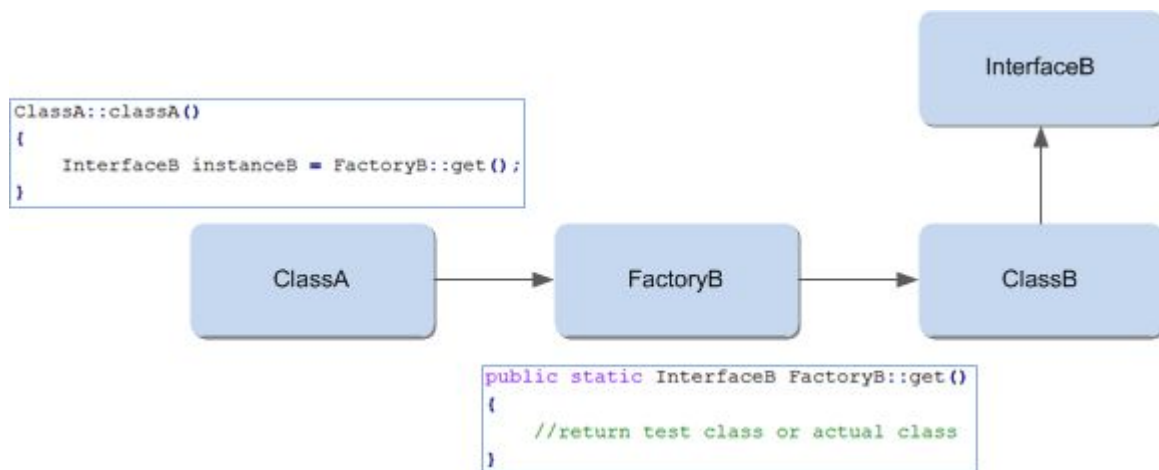
The direct, compile-time dependency on the concrete implementation class means that you cannot test the client code (class A) mocking (substituting in) different states of the concrete implementation class.



### Use a factory to instantiate the dependent object

This approach decouples the client code and implementing class.

- The client code calls on the factory to instantiate the interface.
- The factory provides static methods to get and set mock (substitute) implementations for interfaces.
- A factory is implemented with some boilerplate code.

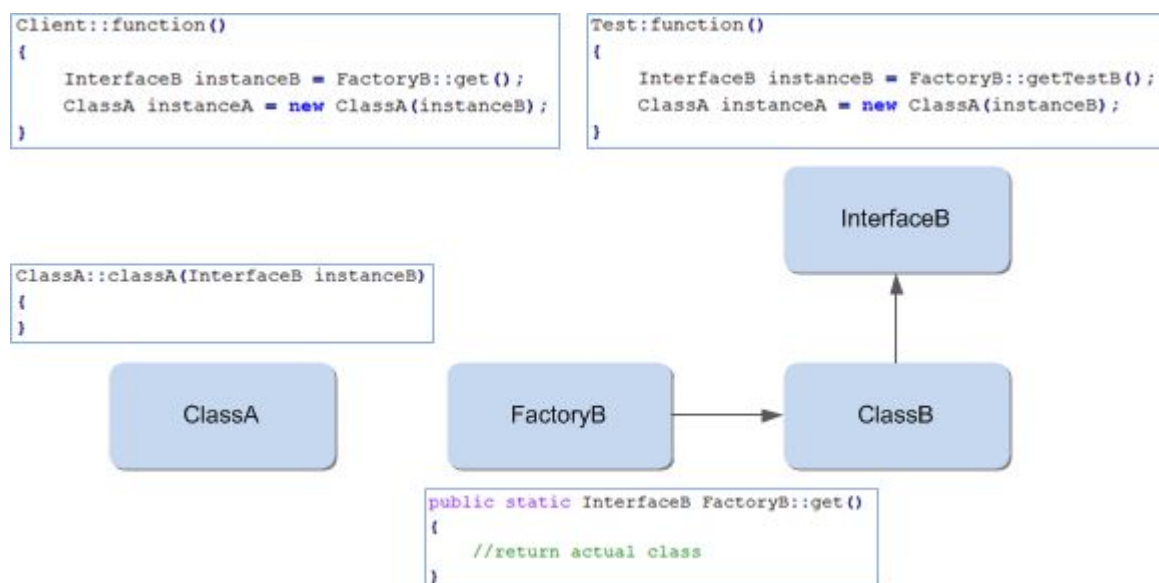
**Note**

When you use a factory to instantiate the dependent object, the code can be unwieldy because the factory must retain mock implementations for testing. Test cases must ensure that they remove the mock in their tear down methods; otherwise, they can impact other test cases

**Manual dependency injection**

With manual dependency injection, the client code does not look up the dependencies. The dependencies are passed into the constructor of the client.

- The client code does not need to know how to look up the factory or instantiate the concrete class.
- The dependencies are moved to the calling code.
- For testing the code, the test module can provide mock, or testing, classes as inputs to the component. This makes the test cases simpler.

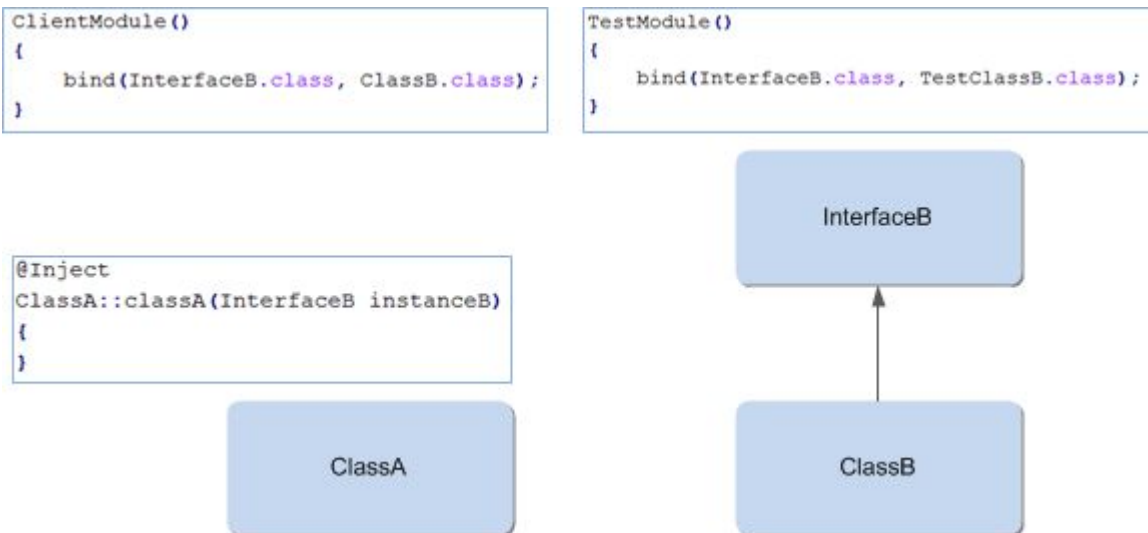


**Note**

The drawback of this approach is that the caller of the client code has to look up the dependencies.

**Automatic dependency injection**

- A configuration file specifies the implementation class to use for an interface.
- The caller of the client or the client code does not have to look up the dependencies.
- Tests can use a different configuration file to get mock instances.

**Note**

Dependency injection frameworks such as Spring and Google Guice, use configuration files (XML, annotations, binding) to define the binding of an implementation to the contract. At run time, the framework uses byte code manipulation to construct and provide the right implementation class to the component that needs it.

GIN (GWT injection (<http://code.google.com/p/google-gin/>)) provides automatic dependency injection for GWT applications by leveraging GWT compile time generator support. The Active Workspace client framework leverages GIN to provide dependency injection.

**Extensibility**

*Extensibility* is a system design principle where the implementation considers future growth. Extensions can be made through the addition of new functionality or through modification of existing functionality. The goal is to support change while minimizing impacts to existing system functions.

The system must be designed to include hooks and mechanisms to expand and enhance the system with anticipated capabilities, without having to make major changes to the system infrastructure.

Teamcenter is a vast system that provides a rich set of functionality to meet customer needs in product lifecycle management.

- Most implementations do not incorporate all provided, out-of-the-box (OOTB) functionality. Based on domain and specific needs, most implementations require only a subset of the provided functionality.
- Many implementations require incorporation of components from third-party developers into the installation.
- When OOTB or third-party components do not provide needed functionality, you can build components and add them to your installation.
- If you need to configure OOTB, third-party, and custom components, the Active Workspace client framework lets you extend and configure components as part of the framework.

The goals for the extensibility in the Active Workspace client include:

- Allow components to define well-bounded extension points.
- Support zero, single, or multiple extensions for extension points.
- Support asynchronous loading of extensions to ensure that the code behind extensions are loaded only when needed.
- Reduce coupling between application and its components.
- Support a deprecation policy for published extension points.
- Allow components to define and attach extensions to their own or dependent component extension points.

**Note**

Extensions are used only if the component is part of the overall application.

The Active Workspace client extensibility framework:

- Is designed to let modules provide extension points or interfaces that the dependent modules can extend. This supports decoupling of the application and its components.
- Is based on the Dependency Injection capabilities provided by Google GIN framework (<http://code.google.com/p/google-gin/wiki/GinTutorial>). GIN is enhanced by the framework to achieve the needs for extensibility in the Active Workspace client.

## Component architecture

The Active Workspace client is built on a component-based vision of a software factory and its development process. The foundation of the factory is built on modularity and the definition of clear, well-defined, managed contracts across the boundaries.

The modularity specification has five levels of granularity:

- Class

At the lowest level is a *class* (that is a Java class in the GWT client) or a resource (for example, image, css, text, and so on).

- Package

A *package*, at the next level, provides a namespace for qualifying the class or resource. It is also a means to group the class or resources that are closely related together.

- Module

A *module* is a group of packages. In the client, this corresponds to the GWT module construct. A module is the smallest deployable artifact in this architecture.

- Component

The *component* provides a set of functionality that satisfies a set of use cases. The functionality can be implemented by using multiple modules that are within the component. The component is the smallest managed software artifact.

- o Components are coarse grained, such as Search and Change Management.
- o Components are managed as independent units of software.
- o Components have versions and are managed individually.
- o Components manage their dependencies explicitly, specifying the components they depend on and the version of those components.
- o Components publish explicit contracts. They have clear contracts and well-publicized and managed deprecation policies.

**Note**

When a new version of a component is released, other components that have dependencies on the new component can update to the new version as long as they only rely on the published contract without any restrictions.

- Solution

A *solution* collects multiple components and is the smallest unit of deployment at an installation. Solutions put together components to solve business problems for a well-defined domain and user community.

## Component contracts

Each component has a well-defined *contract*.

- The contract consists of artifacts and behaviors that the component exposes to satisfy the stories it supports.
- A contract is any semantic that the component exposes, such as APIs, resources, and GUI components.



- Each component can define what comprises its contract based on the component's desired capabilities.

The component contract has two levels of publication:

- Published to Public
- Published within Siemens PL DEV

Contracts tagged as published to **Public** for licensed customers and partners can be used by customers, partners, and other PL developed components and internally within the component publishing the contract. These contracts are the most mature and least likely to change and, therefore, have the longest notice for deprecation.

Contracts tagged as published to **Siemens PL DEV** can be accessed by other components that are developed by Siemens PL DEV and are not available for use by customers and partners. These are either less mature contracts that are not ready to publish or are explicitly held back to manage intellectual property or the cost of maintenance. These typically have a deprecation notice duration that is shorter than the published contracts. However, there is always an agreed upon notice duration that is well publicized and guaranteed to be honored so that other components can rely on it.

Contracts internal to the component that are not published must not be used by any other component, including components built by Siemens PL DEV.

- This ensures that a component can evolve with internal implementation as long as it continues to honor its external facing contracts. This characteristic is key to the evolution of the components and maintains quality.
- This gives customizers the stability needed to continue to deploy newer versions of their extensions and components.

#### Caution

Be sure to stop using deprecated contracts as they can be removed anytime after the deprecation notice period expires.

An API in the Active Workspace client uses Java annotations to express its level of publication. The annotation can be applied to a class or to a method in a class. When it is applied to a class, all methods in the class honor the level of publication unless it is specifically overridden by a method-level annotation.

Every API in a component's *published* package has annotation. Annotation specifies the *maturity* and *visibility* of a given interface or class.

The following code shows an annotation definition:

```
@Target( { ElementType.METHOD, ElementType.TYPE } )
public @interface ApiVisibility
{
    MaturityEnum maturity();

    Scope publishScope();

    /**
     * How mature or stable is the api
     */
    public enum MaturityEnum
    {
```

```

        Experimental, // as the name states; this can change without notice
        Mature, // provides a guaranteed deprecation notice period before removal
        Deprecated // is in the notice period for future removal
    }

    /**
     * what is the intended visibility or access
     */
    public enum Scope
    {
        Hidden, // only available within the component that defines this API
        Internal, // only available to Siemens PL DEV
        Public // Available to customers, 3rd parties etc.
    }
}

```

An example of applying this annotation follows:

```

/**
 * Interface for the End Point configuration information.
 * This is used to hold configuration values for configured
 * connection and authorization data.
 */
@ApiVisibility( maturity = MaturityEnum.Experimental, publishScope = Scope.Internal )
public interface IEndPointDefinition
{

```

## Managing components

A *component* is the unit of management and development that goes through the software life cycle and development process.

- Typically, each component is developed and enhanced in an independently managed process. The component goes through its life cycle and is released when it achieves its goals, and as long as its dependencies are satisfied.
- When a component is released, its version number should be incremented. Its published and previously undeprecated contracts are guaranteed to function in the new version. The component clearly articulates what contracts are deprecated and when they will be removed.
- A *solution* is an assembly of specific compatible versions of components that are released together.

As components evolve, they manage their dependencies and move to newer versions of components they depend on. A component version is incremented if it adds, removes, or deprecates a contract such that form, fit, or function are changed.

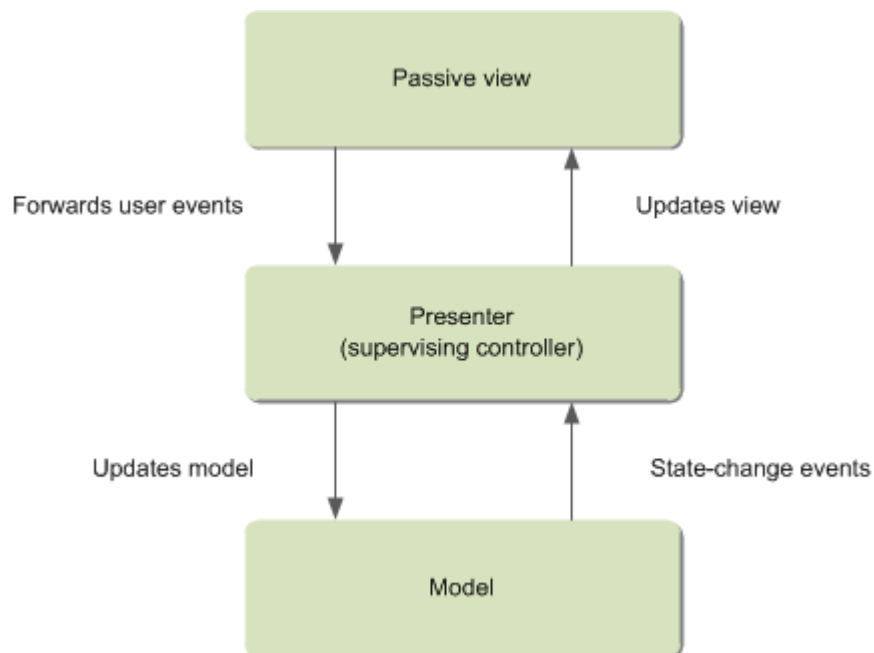
- A published contract does not change, but it is deprecated and a new contract is published if change is necessary.
- Only contracts that have been previously deprecated and the notice period has expired can be removed. Using a new version of the dependency does not, by itself, cause a component to increment its version.

## Chapter 6: View model and data binding

### Model-View-Presenter (MVP)

Model-View-Presenter (MVP) sets up a pattern that provides good separation of concerns (SOC). This allows building large scale user interface (UI) applications. The data binding framework used in the Active Workspace client helps to bind the data in the model with the widgets in the *view*. This reduces the amount of code application developers must write to implement use cases.

- The *view* provides the UI. The view does not have business logic.
- The *model* is the entity graph that the view represents.
- The *presenter* contains the business logic and coordinates the interaction between the view and the model.



The diagram shows the interaction between the different elements of the MVP pattern.

- The presenter manages creating and updating the model and view.
- The view presents the UI and forwards user events, such as click events, to the presenter.
- The model forwards state-change events to the presenter to update the view.

### Active Workspace client MVP elements

The Active Workspace client uses the GWT platform to provide the MVP framework.

The *model* typically consists of client data model objects, such as model objects and model types.

- Client data model objects consist of presenter specific data that are not model objects (JavaBean style objects).
- The model is typically populated by making an SOA call or by client-side logic.
- If the model consists of model objects, the model has to listen for changes in the model objects and update based on user actions.

The *view* consists of widgets for layout and content. Styling is derived from cascading style sheets (CSS), and the content can be static and dynamic.

- Static elements, such as a titles, are defined in the view.
- Dynamic elements, such as objects and object properties, come from the model.
- The presenter initializes the dynamic content and keeps it up to date.
- The view communicates user events and value-change events to the presenter through a **UiHandlers** interface.

The *presenter* is the controller that coordinates interaction between the model and view. It also handles the life cycle of the UI. The business logic is present in the presenter.

## Repetitive code patterns

The following code patterns are repeated in each presenter and view:

- Populate the model from the client data model.
- Create the view based on the model data.
- Listen for state changes in the model.
- Update the view based on model changes.
- Listen for value changes in the view.
- Update the model based on value changes.

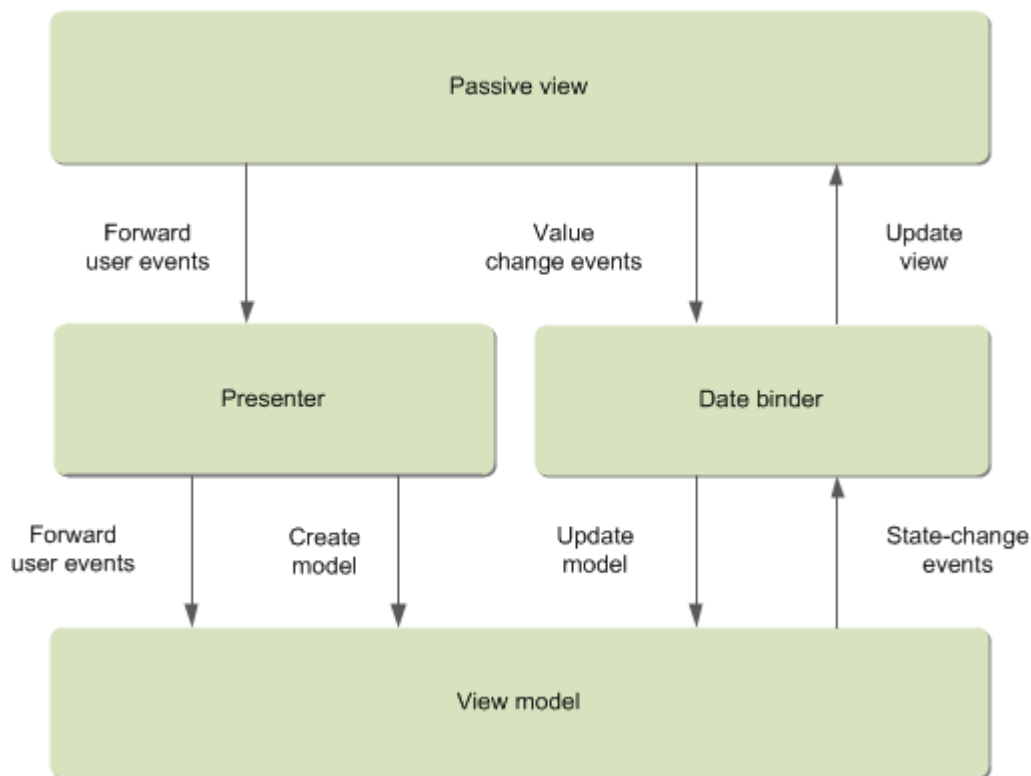
These patterns provide a framework to encapsulate repetitive work and minimize work needed in each presenter, view, and model for presenting the UI, and help make testing simpler.

## Data binder

### Data binding framework

The Active Workspace client data binding framework handles repetitive code patterns for these interactions:

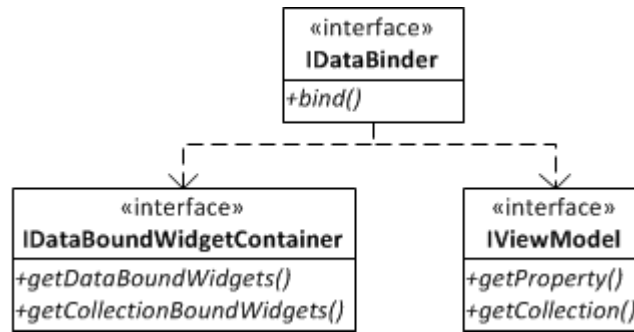
- Initialize the view from the model.
- Listen for state changes in the model.
- Update the view based on model changes.
- Listen for value changes in view.
- Update the model based on value changes.



To assist in testing, the business logic for UI interactions is moved from the presenter to the view model. Test modules can instantiate the view model and perform the testing without having to instantiate and mock up the presenter or view.

This pattern lets the presenter role focus on:

- Creating the model and view at the appropriate time in the UI life cycle.
- Providing the view and model to the data binder to bind when the UI is displayed.
- Unbinding the view and model when the UI is hidden.
- Forward user events from the view to the model.



The view and presenter do not have to generate code for initializing or updating data in the view or model. The data binder does this, and it enforces a contract on the view and view model to achieve the data binding.

- The view model implements the **IViewModel** interface to provide the list of properties and object collections. The data binder must listen for observable state changes in the view model.
- The view implements the **IDataBoundWidgetContainer** interface to provide the list of property and object collection widgets that participate in data binding.

## Properties in the view and view model

The widgets in the view that display properties on objects implement the **IDataBoundWidget** interface.



These widgets render a specific property in the view model. They are bound to a property in the view model by the *bind value* on the widget. The bind value is a string that identifies the property to render.

A corresponding property in the view model must implement the **IViewModelProperty** interface.



This property model object contains the data type, property name, property value and property display name. It also contains state flags for the property such as required, editable, enabled, and so forth, and it specifies the valid list of values for the property.

The data binder binding:

1. Asks the view for a list of data-bound widgets.
2. For each data-bound widget:
  - a. Gets the bind value.
  - b. Asks the view model for the property with that bind value.
  - c. Ensures the data type for the widget and for the **ViewModel** property are identical.
  - d. Becomes a listener for widget value changes.
  - e. Becomes a listener for view model property changes.
  - f. Initializes the widget with the view model property data.

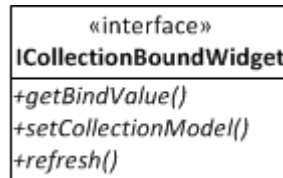
**Note**

If the state changes in the view model property, the widget is updated with the values from the view model property.

If the value changes in the widget, the new value is set in the view model property. Any validation error returned by the view model property is passed on to the widget.

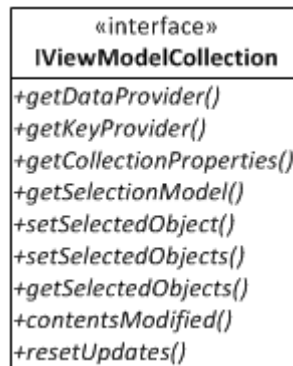
## Collections in view and view model

You may encounter use cases where the view must show properties from a list of objects in the UI, such as in a table, a tree, or a list. The widgets that display list of objects must implement the **ICollectionBoundWidget** interface.



The widgets are bound to a collection in the view model by the bind value on the widget. The bind value is a string that identifies the collection to render.

A corresponding collection in the view model must implement the **IViewModelCollection** interface.



The collection uses the data provider to provide the list of objects that are part of the collection and specifies:

- The properties on the list of objects rendered in the UI, such as columns in a table.
- The selection model to be followed by the view for this collection. This can be no selection, single selection, or multiple selection.

The binding process for collection by the data binder is same as the property binding process.

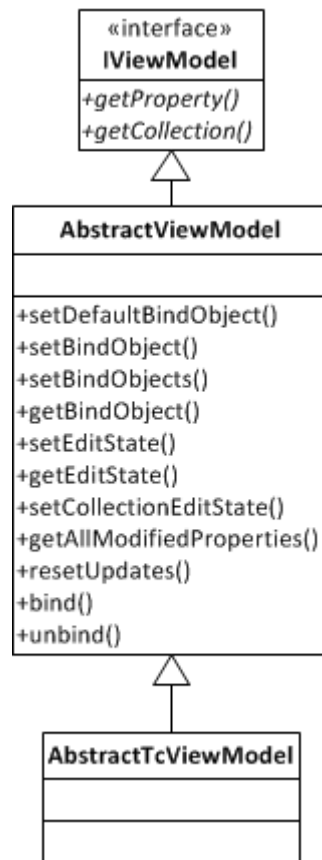
## View model

The client uses view model objects to represent the data model within the client.

### Abstract view model

The framework provides an abstract view model implementation to handle model objects.





The abstract implementation, **AbstractViewModel**:

- Presents properties on the model object as view model properties.
- Allows presentation of the list of model objects as a view model collection.
- Adds itself as a listener for model object modified events to update the view model and to update the view when model objects change in the client data model.

The framework provides the **AbstractTcViewModel** subabstract implementation to handle Teamcenter-specific lists of values and naming patterns.

### Concrete view model

All concrete view models that are displaying Teamcenter model objects must inherit from **AbstractTcViewModel** to incorporate this behavior. When view models get model objects from the server, using SOA or from the presenter, the view models add the model object to the view model by using the **setDefaultBindObject** method on the parent class. The view model can then serve properties on this model object to the view. For example, a widget has a bind value of **object\_name**.

```
setDefaultBindObject(modelObject);
```

In some cases, properties from more than one model object can be presented in the view, such as showing properties from a task and the parent process in the view. The concrete view model can add the other objects that are going to be part of the UI using the **setBindObject** method. The concrete view model must provide the scope and identifier for the model object, such as process, for the view

to identify the object, such as a widget with a bind value of **process::object\_name**. If scope is not provided in the widget bind value, the property is sought in the default bind object.

```
setBindObject("process", processModelObject);
```

#### Note

The properties that must be presented in the view must be loaded from the server before the object is added to the view model. The view model cannot load missing properties. This lets the concrete view model and presenter leverage property policy mechanisms to bulk load the properties from the server and reduces the number of interactions with the server.

The concrete view model can add a named list of objects that are rendered in collection widgets to the view model using the **setBindObjects** method. Each collection is identified by a bind value that must be provided as input to the **setBindObjects** method. The list of properties to be displayed on the list of object must also be provided as input.

```
setBindObjects("set1", modelObjects, properties);
```

The widget that renders this collection specifies a bind value of **set1**.

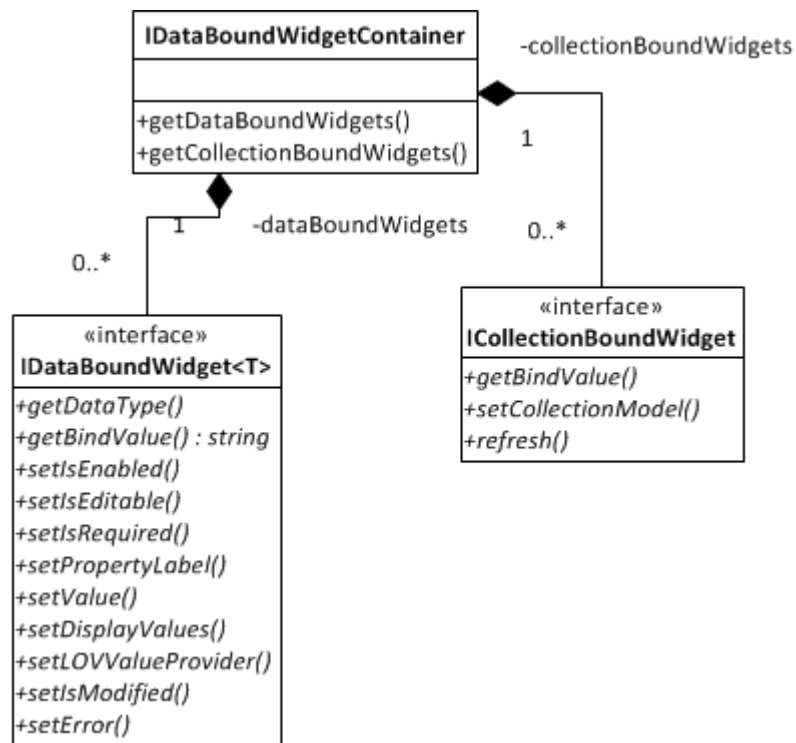
In addition to model object support, the abstract view model provides generic object.

- The concrete view model can leverage these objects to add model data that is not represented by model objects in the client.
- The **GenericViewModelObject**, **GenericViewModelProperty**, and **GenericViewModelCollection** classes are provided for this purpose.

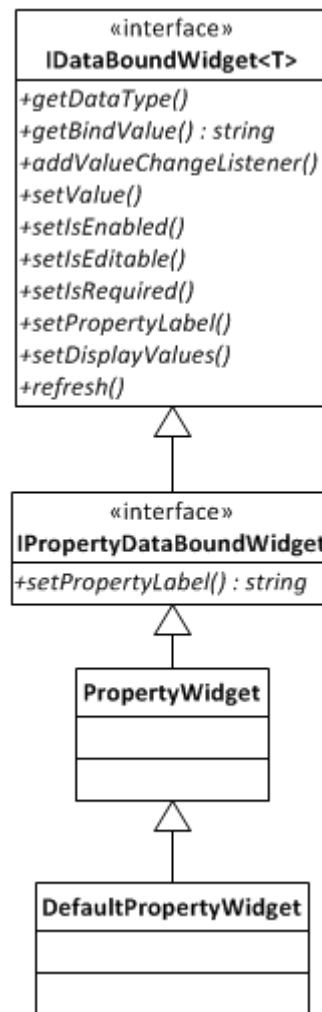
To add an object to the view model, the concrete view model must create an instance of **GenericViewModelObject**, add instances of **GenericViewModelProperty** class to represent the properties on the object, and then add the object to the view model using the **setBindObject** method.

## View

The view consists of a list of widgets for rendering the layout and content. If any widgets are leveraging data binding, the view must implement the **IDataBoundWidgetContainer** interface and provide the list of widgets to the data binder.



The Active Workspace client framework provides a set of data bound widgets as part of the Teamcenter UI layer.



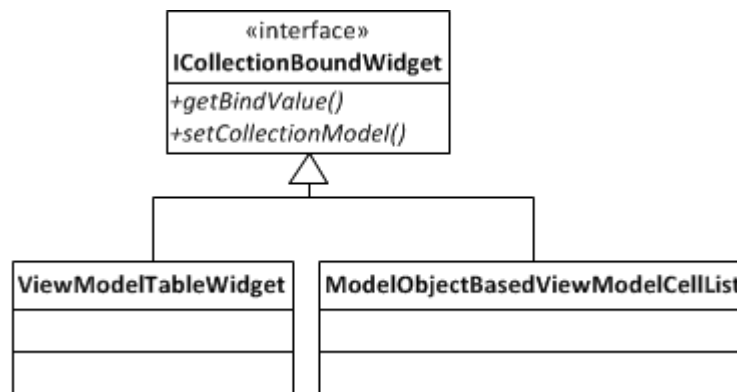
These widgets are described in *Using property widgets*.

- These widgets inherit from **DefaultPropertyWidget** and provide support for Teamcenter semantics for properties such as LOVs, naming patterns, required, and so on.
- They should be used wherever Teamcenter properties are rendered in the UI for consistency in functionality and look and feel. Custom Property widgets can be created by inheriting from **DefaultPropertyWidget**.

**Note**

Ensure that all Teamcenter semantics for properties are honored.

Collection widgets that are provided by the framework Teamcenter UI layer for presenting model objects in a table or a list. The list widget leverages preferences defined on the server for determining the properties to render in the list tile.

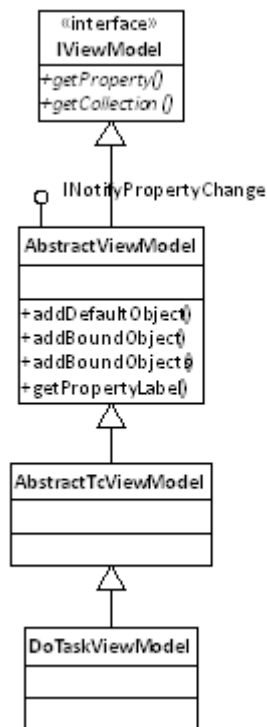


Value change events from the data bound widgets are handled by the data binder. Other events of interest in the widgets in the view, such as button click and scroll end, for example, must be trapped by the view and passed on the presenter by the **UiHandlers** interface.

## Presenter

The *presenter* creates the view model and provides the view and view model to the data binder for binding.

The data binder manages the data flow between the model and view.



```

m_dataBinder = IDataBindInjector.INSTANCE.getDataBinder();
m_dataBinder.bind( getView(), m_viewModel );

```

When the presenter is hidden, the data binder and view model do not have to listen to events for synchronization. The data binder must be unbound when the presenter hidden.

```

if( m_dataBinder != null ) {    m_dataBinder.unbind();    m_dataBinder = null; }

```

For user events that the view communicates to the presenter, the presenter must forward these events to the view model. This ensures the business logic is in the view model, rather than in both the view model and presenter. This lets you test the business logic by testing the interfaces on the view model. The tests do not have to *mock* the presenter or view to do this testing.

## Chapter 7: Service-oriented architecture (SOA)

### SOA goals

The client makes service-oriented architecture (SOA) service calls to the server.

Whether you want to use an existing SOA service or create a new one, SOA services for the Active Workspace client framework must be suitable for your needs.

You must understand the approach taken to make Teamcenter SOA services available to a GWT client and how those services are consumed, and you must understand the principles of the client-server interaction model and related industry best practices.

The goals of SOA support in the Active Workspace client include:

- Allow a native GWT access to Teamcenter SOA services similar in style to GWT - RPC, such as using an asynchronous access pattern and providing interfaces for each service with a factory pattern to hide implementation specifics.
- Integrate into the current development environment and autobuilt from the same Business Modeler IDE templates that build other SOA bindings.
- Hide, from application developers, the details of the marshaling and unmarshaling logic.
- Have a light foot print. Do not require a large download to make a few service calls. Only download what is absolutely needed based on the current location's needs. This is essential when running on resource-constrained platforms such as a smart phone or tablet computer.
- Meet or exceed the performance of the traditional clients.

The Active Workspace client framework follows industry standard patterns in its client-server interaction; each pair of request-and-response activities are entirely independent and unrelated to prior or subsequent communication. The interfaces of SOA operations must be stateless and explicit.

An SOA operation is stand-alone with no reliance on prior or subsequent operations. The individual request is serviced only by the server. There must be no factors outside the scope of the request-response activity that dictate what the operation does. This lets you systematically ensure that the client remains resilient to server reassignment such as when connection to a **tcserver** process is lost.

For the Active Workspace client, you must ensure our operations remain stand-alone, and the client is properly decoupled from the server. This prevents building in implied assumptions across SOA calls or holding server state in the client that is tied to a given server's process.

Stand-alone operations are important for the following reasons:

- They simplify the relationship between client and server; the server fulfills a single request with a response.
  - o State is cleanly managed.

- o No indefinite synchronization of state between client and server.
- They decouple the client from the server and separate tiers and encapsulation of responsibilities, which improves stability and maintainability.
- The client is not indefinitely bound to a **tcserver** process.
  - o The administrator can manage the server pool without being concerned that clients are tied indefinitely to a particular **tcserver** instance.
  - o If a server fails, the client can continue because any SOA call made by the client can be serviced by a new **tcserver** instance.
- Users can return to the client after a period of inactivity and resume their work. There is no reliance on a particular server remaining available.
- Users can copy-and-paste or email the browser URL at any time. After passing the necessary security challenge, users can open the link to get to the same location.
- Operations that can be explicitly validated and tested. There is no additional complexity from influence of various possible prior operations.

The framework of the client and server tiers is designed and enhanced to facilitate this approach to client-server interaction. However, to ensure you use only explicit and stand-alone operations requires that application code you develop for the Active Workspace client must follow the guidelines for the design and selection of SOA operations.

## Framework support for REST services

Teamcenter SOA services are document-style *representational state transfer* (REST)-based services.

- A client calls a service by posting a document with an HTTP request to a service end point and the server responds back with a response document.
- Each service defines its input and output schemas using Business Modeler IDE templates.

Prior to development of the Active Workspace client framework, the supported payload was XML, and the XML schema is compliant to the definitions in the Business Modeler IDE definitions. As part of the introduction of the Active Workspace client framework, SOA supports a *JavaScript Object Notation* (JSON) payload. In this case, the JSON payload is compliant with the same schema definitions except the format is JSON compliant.

## Programming model

Synchronous calls from a client to a SOA service is not recommended. Calls to the server from the GUI thread results in the client interface becoming unresponsive for the duration of the call, giving the impression the application is not functioning. Because the service call is typically made over a WAN, latency is directly felt by the user in terms of unresponsiveness.



The Active Workspace client framework defines a programming model to make it impossible to use this pattern. From a review of the GWT RPC style of services, the adopted model shares the benefits of that approach by leveraging asynchronous patterns.

An autogenerated interface definition is provided for each Teamcenter SOA service. The interface follows an asynchronous pattern.

```
public interface SessionService
{
    public void login( com.teamcenter.services.gwt.core.published._2011_06
                      .Session.Credentials credentials,
                      final AsyncCallback<com.teamcenter.services.gwt.core.published._2011_06
                      .Session.LoginResponse> callback );
    ...
}
```

#### Note

Logon operation input parameters are the input types defined on the service in the Business Modeler IDE. There is an additional last parameter that is the callback object (**AsyncCallback**). The output of the service is void because it is an asynchronous service that has an immediate return.

When this service is called, the application calling the service provides a callback object. The callback object has two methods (**onSuccess** and **onFailure**), which are called based on the outcome of the call to the server. The caller must implement these methods and process the return.

A factory is provided to get a service. An example access pattern is as follows:

```
SessionService service = SessionGinjector.INSTANCE.getSessionService();

service.login(userName, password, "", "", "en_US", "A-JSON-Client",
              new AsyncCallback<LoginResponse>()
{
    @Override
    public void onFailure(Throwable caught)
    {
        //do your error processing here
        logger.log(Level.WARNING, "Login Failed");
    }

    @Override
    public void onSuccess(LoginResponse result)
    {
        // Do your work here
    }
});
```

## AsyncCallback implementation best practice

The **programming model** example shows an anonymous class implementing **AsyncCallback** being passed to the logon request.

This works well where the **onSuccess** and **onFailure** methods do not require any state to be available to finish processing the success response or the failure.

In other cases, the **onSuccess** or **onFailure** methods require a state to continue processing the request. To handle these use cases, the framework provides the following convenience classes that applications can use:

```
public abstract class AsyncCallbackWithContext<Tstate, Tresult>
public abstract class AsyncCallbackWithContextPair<Tstate1, Tstate2, Tresult>
```

These abstract classes, located in the **com.siemens.splm.clientfx.base.published** package:

- Allow state to be passed as part of their construction.
- Override and finalize the **onSuccess** methods from the **AsyncCallback** class, and then delegate to these statements:

```
public abstract void onSuccess( Tstate state, Tresult result );
public abstract void onSuccess( Tstate1 stateObj1, Tstate2 stateObj2, Tresult result );
```

- Pass in the state provided at construction time.

## Code autogeneration and integration into the build system

The interfaces, input/output types, and the implementation and factories to return these implementations are standard code that developers should not have to re-create. To achieve this, the Active Workspace client framework:

- Provides a code generator to generate these interfaces and classes as part of the build system, and packages this using modularity principles. The code generator is similar to the code generators for the other SOA client bindings, such as C++, that many customizers already use.
- Uses the Business Modeler IDE template definitions for the services and produces the boilerplate artifacts needed by the Active Workspace client and packages it up.

To generate GWT client bindings for an SOA service:

1. To add the SOA service operations, follow the instructions in the *Working with services* topic in *Business Modeler IDE*.
2. Enable the GWT code generator & build **your-service-name.autogen.jar** in the Business Modeler IDE.
3. Copy the resultant **JAR** file into the **STAGE/src/your-service-name** directory.
4. Create the following **module.json** file in this directory alongside your **your-service-name.autogen.jar** file.

```
{
  "type": [ "soa" ],
  "gwtModules": [
    "full-package-name-of-your-GWT-module"
  ]
}
```

5. From the command line, run the **gwtcompile.cmd** script to build your SOA module for use in your Active Workspace customization.

## Client data model

When data is returned from the server, it is deserialized into client objects using an autogenerated binding.

The over-the-wire schema for Teamcenter SOA is a typeless schema. When data arrives, it is essentially a collection of properties with no information about the types or related metadata. *Client data model and the meta system* describes the interfaces provided, so these objects can be consumed within the client model-view-presenter design pattern of the Active Workspace client.

## Object property policy

The object property policy dictates the required properties for given object types. It determines the properties that are returned in a SOA response by the server for a given service call.

For the Active Workspace client, this policy is constructed by the client framework, given the property requirements of its application consumers. The policy is tailored to client needs as registered by the presenters.

The policy is included in the SOA request document when a call is made to the server. In this way, the property policy requirements are explicitly stated on the request, and there are no implied stateful assumptions about the active property policy at the server or about which **tcserver** process is handling the request. This explicit and stateless mechanism for specifying client property requirements ensures the SOA request is standalone and is resilient to server reassignment.

This approach is in keeping with the goals to:

- Have the SOA contract to explicitly state in the SOA request the properties the caller requires for the objects returned.
- Have the SOA response explicitly contain the properties set requested for the objects returned.

### **IObjectPropertyPolicyManager**

For the client framework, an **IObjectPropertyPolicyManager** class is designed to centrally manage the client's property requirements.

- The client **IObjectPropertyPolicyManager** class manages the client's property requirements as registered with it by the application.
- It uses an **OR** operator to collect the requirements of the client views and presenters, such as the requirement for a tree view with specific columns.
- To prevent ripple to client application callers, the **IObjectPropertyPolicyManager** class is in the client framework layer. It can be queried by the **IOperationManager** class when an SOA request is to be sent to insert or inject the property policy in the request

The *effective* aggregated needs of the application, as provided by the **IObjectPropertyPolicyManager** class, are used to construct the SOA request by the client framework.

### **Property requirements registered by the application**

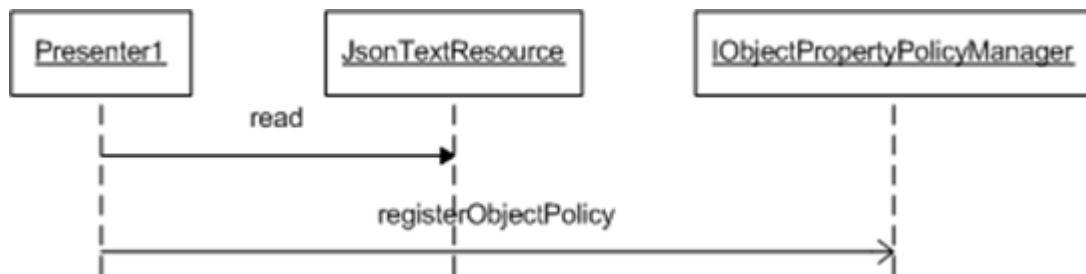
The application presenters must register, update, and unregister their object property policy requirements with the **IObjectPropertyPolicyManager** class as needed. This lets the

**IObjectPropertyPolicyManager** class track the needs of the application and maintain the effective policy. The effective policy is used when constructing the SOA request to explicitly specify the client's property policy requirements for an SOA operation.

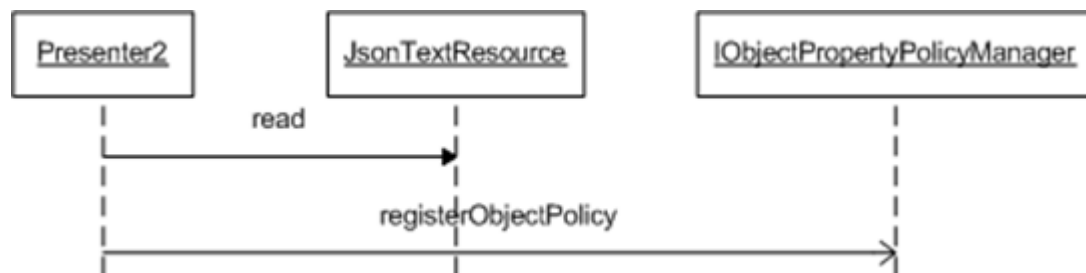
As the requirements of the client change, such as when a new column is added to a table for a given presenter or when a new presenter is created, the effective policy is managed by the **IObjectPropertyPolicyManager** class.

Consider the life cycle from the application's perspective.

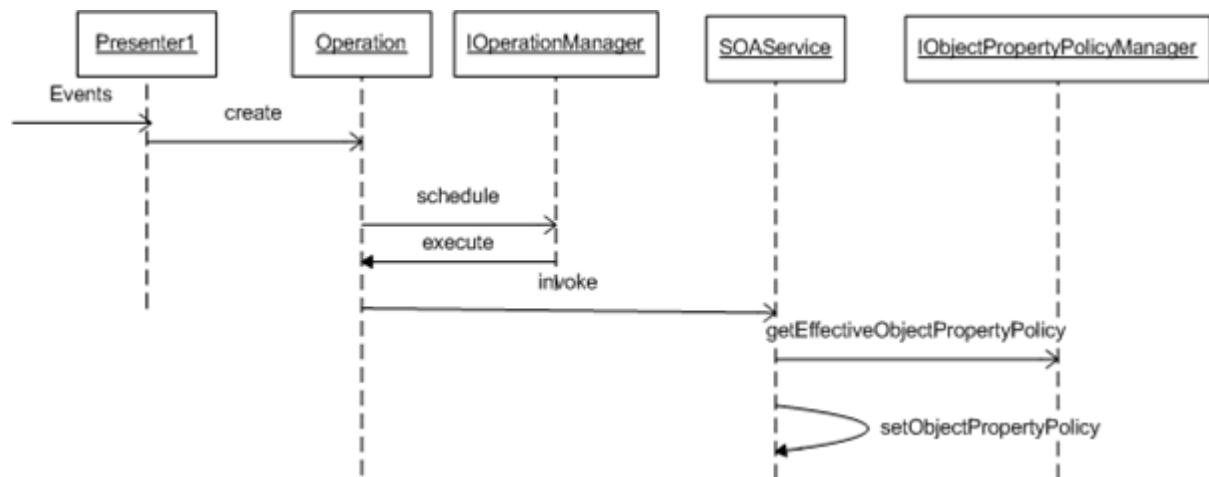
1. **Presenter1** uses **IObjectPropertyPolicyManager** to register **checked\_out** and **object\_name** properties on the workspace object (WSO).



2. **Presenter2** registers **object\_name** and **owning\_user** on the WSO with **IObjectPropertyPolicyManager**.
  - The effective object policy in **IObjectPropertyPolicyManager** now has **checked\_out**, **object\_name** and **owning\_user** properties on the WSO.
  - The **object\_name** property has a **registeredCount** of 2 because it is registered by both **Presenter1** and **Presenter2**.

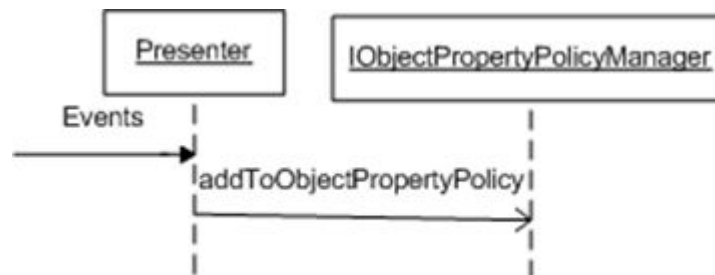


3. **Presenter1** creates an operation and initiates an SOA call.
  - The client framework queries the effective object property policy from the **IObjectPropertyPolicyManager** class and explicitly sets it in the request envelope.
  - The SOA response returned to the client includes the WSO **checked\_out**, **object\_name** and **owning\_user** properties.



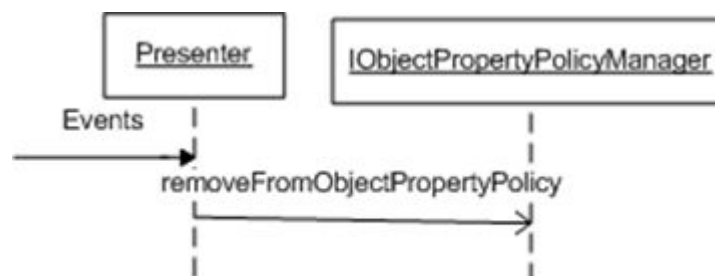
4. **Presenter1** adds the **creation\_date** property to the object property policy it registered.

The effective object policy in **IObjectPropertyPolicyManager** now has **checked\_out**, **object\_name**, **owning\_user**, and **creation\_date** properties on the WSO.



5. **Presenter2** removes the **owning\_user** property from the object property policy it registered.

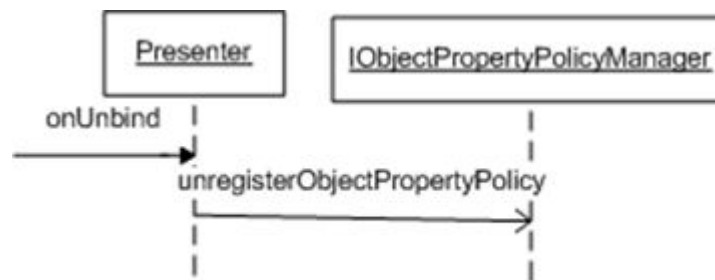
The effective object policy in **IObjectPropertyPolicyManager** now has **checked\_out**, **object\_name**, and **creation\_date** properties on the WSO.



6. **Presenter1** is closed.

It unregisters the object property policy.

The effective object policy in **IObjectPropertyPolicyManager** now has **object\_name** and **creation\_date** properties on the WSO. The **object\_name** is still there because it was also registered by **Presenter2**.



As the needs of the client change, such as when a new column is added to a table for a given presenter or when a new presenter is created, the effective policy is maintained and managed by the **IObjectPropertyPolicyManager** class.

The **IObjectPropertyPolicyManager** class provides **addToObjectPropertyPolicy** APIs for your registered policy.

### Explicit object property policy

For the client framework, the effective object property policy maintained by the **IObjectPropertyPolicyManager** class is an explicit property policy that is included in the SOA request document. The JSON text sent to the server logically has two parts: the request body and the explicit property policy.

When handling an SOA call, the **tcserver** gateway has preexisting capabilities to *push* and *pop* the client state before and after the service call is invoked.

As part of the *pushing* of the client context, the explicit property policy is used to construct a dynamic property policy that is applied by the policy manager for the duration of the server call. When the service response is constructed, as part of *popping* the client context, the dynamic property policy is unregistered with the policy manager.

## SOA checklist

You must ensure operations remain stand-alone, and the client is properly decoupled from the server.

- This prevents building in implied assumptions across SOA calls and prevents holding a server state in the client that is tied to a particular server process.
- This approach also systematically ensures the client remains resilient to server reassignment, such as when connection to a **tcserver** process is lost. The client must not depend on which server services a request.

To use an existing SOA operation or create a new SOA, you must consider the following statements.

The SOA operation must *not*:

- Rely on any prior or subsequent operation.
- Rely on being serviced by a particular **tcserver** instance.
- Rely on any objects in a given **tcserver** instance being present or loaded.

- Load an object or state in the **tcserver** instance with the expectation that it is needed by a subsequent operation.
- Contain any run-time business objects, pointers, tags, or data that is specific to a particular **tcserver** process.
- Return run-time business objects, pointers, tags, or data to the client that are specific to a particular **tcserver** process. An exception is allowed if a run-time business object is used as a container for providing properties back to the client, and that object is never used in a subsequent SOA request.
- Implicitly reference addition data, such as service data, that is not directly related to the request and response document.
- Encourage or result in patterns of client-server chattiness.
- Be overly specialized, resulting in highly restricted usefulness.
- Expose objects to the client unnecessarily. For example, low-level server subobjects that relate to the persistence scheme must not be exposed to the client.

The SOA operation *must*:

- Be entirely stand-alone. There is no state or implied state across calls.
- Only perform operations explicitly specified in the SOA request. Implicit factors do not exist.
- Only exchange data that is explicitly stated in the request and response document.
- Use the standard patterns for error handling, partial errors, normalization, and closure.
- Be coarse-grained and can act on sets of inputs and outputs.
- Be consistent with patterns employed for other SOA operations.
- Be suitable for any client or consumer. The interface is client independent.
- Have an interface that is clearly decoupled from its underlying implementation. The interface does not leak implementation specifics.
- Have a clear contract and purpose and be logically encapsulated.
- Have simple and well-defined interfaces to minimize misinterpretation.
- Exploit entity abstraction to share logical concepts across services.





## Chapter 8: Client data model and the meta system

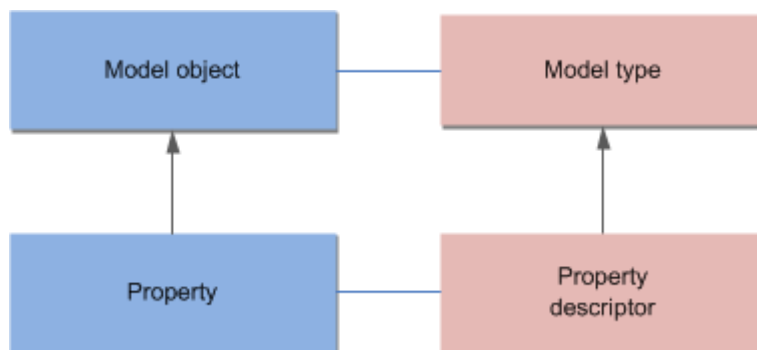
### Client data model

A Model-View-Presenter (MVP) design pattern is implemented for the Active Workspace client framework. The model data that the view model consumes are client objects. These client objects are **ModelObject** objects as defined by the Teamcenter SOA schema. In other words, they are client representations of business objects that form the basis of the client data model.

The client data model has a related *meta definition*.

When data is returned from the server in a service response, it is deserialized into client objects using an autogenerated service binding. The schema for Teamcenter SOA is a typeless schema, so data arrives at the client as a collection of properties with no information about the types. The *meta* system is provided so meta information can let the system make sense of the data returned by the server.

The client data **ModelObject** object has a corresponding **ModelType** object that provides its meta definition. Property meta definition data is provided by a **PropertyDescriptor** object. In this diagram, model data objects are on the left; meta definition objects are on the right.



When data is serialized at the server, the **ModelType** and **PropertyDescriptor** objects are not sent back on that response. This is because there is one instance of the **ModelType** object, the metadata, and many instances of the **ModelObject** object. Serializing the meta information on every call for every object is inefficient. Instead, the client fetches the meta information separately using a dedicated SOA service. This provides the client framework with the flexibility to manage the meta system within the resource constraints of the target runtime environment.

### Programmer-friendly interfaces

The mechanics of the relationship between the model and its associated meta definition is managed by the framework.

A **ModelObject** object has properties, and all properties in the typeless schema are of type string.

- If this schema were to be exposed directly to the application programmer, the programmer would have to manage the lookup of the **PropertyDescriptor** meta definition to determine the type of that property, and then parse the string into the required type before the data is used.

This results in duplicate boilerplate code, which adds little value and increases the complexity of using the framework.

- Instead, the framework client data model provides a convenient API for accessing the **ModelObject** properties and the type and property descriptors.

**Note**

The authoritative source for interface definition is the Javadoc for the release of the Active Workspace client you are using.

The property **getValue** methods use Java Generics to provide type safety as you access the values. For convenience, the interface also provides **getTypeDescription** and **getPropertyDescription** methods for access to meta information.

## Caching

**ModelObject** objects have related **ModelType** objects. Because the **ModelType** objects are meta descriptors, they do not change often for a given Teamcenter deployment and are not expected to change within the same session. Therefore, these objects can be cached for the duration of a session. Such caching reduces the number of calls to the server and improves client performance.

Running in a browser environment brings challenges when caching. Browsers do not allow access to the disk for security reasons. HTML5 supports client-side caching by using the HTML storage API, a key-value database. However, there are limits on how much data can be stored and also differences between browser implementations. In the future, the framework may support some level of storage-backed caching. The Active Workspace client framework provides a simple in-memory cache for meta objects. This cache is used by the SOA response handler to find meta objects as needed and to store meta objects in the in-session cache once they are downloaded. The framework ensures a minimum number of calls are made as this data is incrementally cached.

**ModelObject** objects can also be cached, but these are user-authored objects that can change at any time. The framework currently has a simple in-memory **ModelObject** cache. The goal is to maintain the object while it is needed and to use the object from the cache when it is there. This ability can be used, for example, to follow reference properties in the client, but you do not want to rely on objects being in the cache. Most of these optimizations are invisible to the application programmer and are hidden behind the client data model APIs.

The life cycle of a **ModelObject** object in the client data model is currently managed by the Teamcenter SOA framework, which informs the client framework about changes to objects in SOA responses. It provides a list of the objects that have been modified or deleted by that explicit SOA operation and the properties that were modified. This information is used to update the objects that are cached during the session. The framework SOA handler manages this information and maintains the cache up-to-date.

## Events

Objects can be modified in a session either by the user in the client session or by a server notification of changes on an SOA response. In both cases, any client code using the **ModelObject** object must be made aware of updates to the objects. The Active Workspace client framework provides

an event mechanism to publish, update, and delete events. Handlers can be registered for these events. For example, in an MVP model, these events are used to update the view with the updated object information.

The following code shows how the application can register for event notifications:

```
public class ModifiedObjectHandler
    implements ModelObjectModifiedEvent.Handler
{
    /**
     * Handler Registration for modified object event
     */
    private HandlerRegistration m_handlerRegistration = null;

    public ModifiedObjectHandler()
    {
        m_handlerRegistration = ModelObjectModifiedEvent.register(
            IEventInjector.INSTANCE.getEventBus(), this );
    }

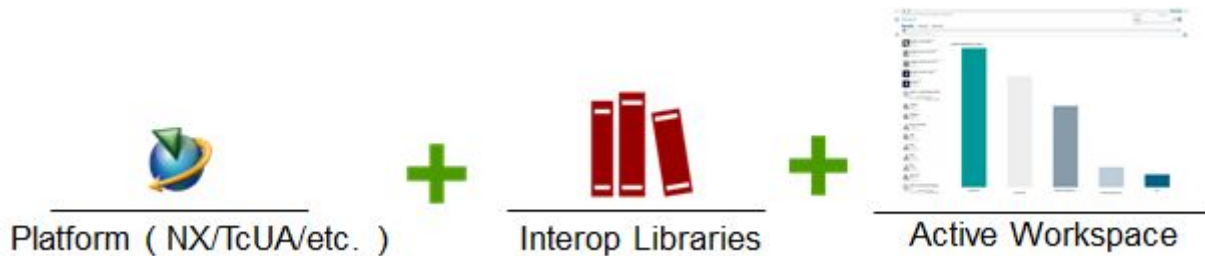
    @Override
    public void onObjectsModified( ModelObjectModifiedEvent e )
    {
        // Handle modified object events here    }
    }
}
```



## Chapter 9: Active Workspace hosting

Active Workspace hosting allows the Active Workspace client experience to be brought into another application.

Examples of Active Workspace hosting can be seen in existing software like NX, Lifecycle Visualization, and the Teamcenter rich client. This functionality can be added to other software platforms as well; the interop libraries allow you to host Active Workspace functionality by utilizing Java, C++, C# (.NET), or JavaScript bindings.



Further information about hosting an Active Workspace session within an application can be found in the Active Workspace hosting programmer's guide, which is available on the GTAC website.



## Chapter 10: FTSTIndexer customization

### Overview of indexer customization

TcFTSTIndexer is a Java application that can execute types, flows, and steps.

TcFTSTIndexer:

- Is an SOA client that connects to Teamcenter to extract data and index the data into Solr.
- Allows modification of any existing steps and flows to meet customer requirements.
- Can be customized to extract external system data and index into Solr.
- Provides utilities that can be used in step customization.

Details of these utilities are in Javadocs available in the `TC_ROOT\TcFTSTIndexer\docs\javadocs` directory.

### Indexer customization prerequisites

- A working TcFTSTIndexer Installation in stand-alone mode.
- A high-level understanding of TcFTSTIndexer architecture.
- An understanding of input and output objects associated with each step in a flow that is being customized.
- An understanding of properties associated with the flow.
- Review sample code for steps in the `TC_ROOT\TcFTSTIndexer\sample` directory.
- Refer to Javadocs for published methods and classes discussed.
- Refer to the `TC_ROOT\TcFTSTIndexer\sample\TcFtsIndexer_sample1.properties` files and the `TC_ROOT\TcFTSTIndexer\conf\TcFtsIndexer_objdata.properties` files for example configurations.
- For the new requirements, create a high-level design of the functionality and:
  - o Create a list of steps with their input and output objects defined. Check if there are existing steps that can be reused.
  - o Chain these steps to create a new flow or modify an existing flow.
  - o Determine if the flow is part of an existing type or a new type.

## Further information

Further information about TcFTSIndexer extensibility can be found in *Active Workspace Deployment*.



## **Part II: Customization examples**



# Chapter 11: Simple examples

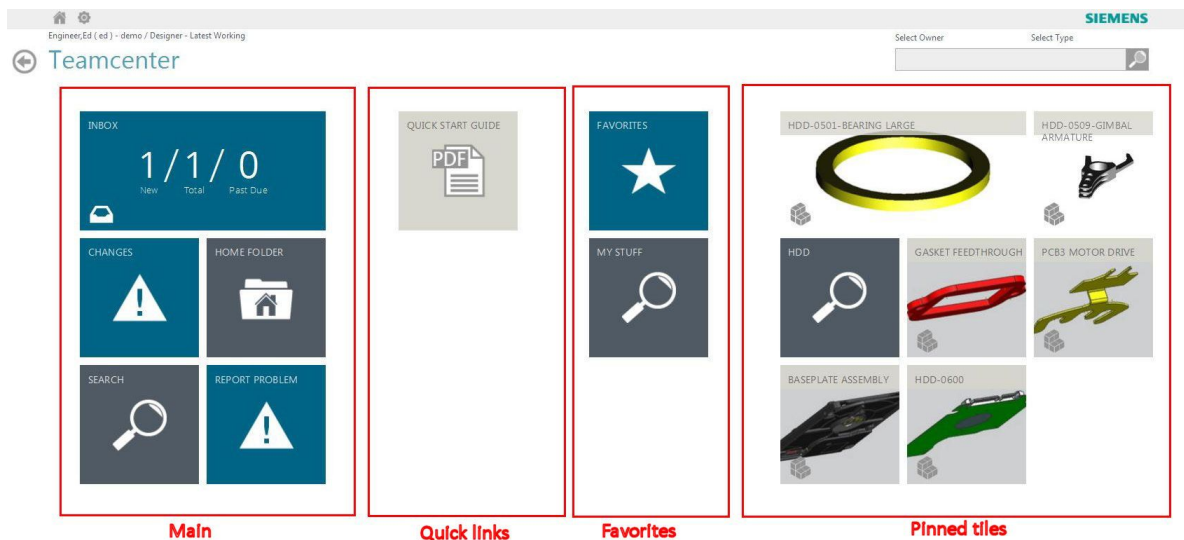
## Simple example overview

These simple customizations require no third-party software. They are performed using the rich client.

## Configuring the home page

### Overview of the home page configuration

The Active Workspace home page is preconfigured to include tiles for the most commonly used features of the client. By default, all users, groups, and roles use the same home page. You can, however, create new configurations for specific groups, roles, and projects. The home page displays the same content regardless of the device used.



When a user logs on to Active Workspace, the home page is based on a combination of the **Site**, **Group**, **Role**, **Project**, and **User** collections. Any tile collections matching the user's current context are combined.

The following objects are used to persist the configuration:

- **Tile**  
Stores tile instances.
- **Tile Template**  
Stores the tile definition.
- **Tile Collection**

Collects tiles together for a given scope (**Group/Role/User/Project**).

- **Tile Relation**

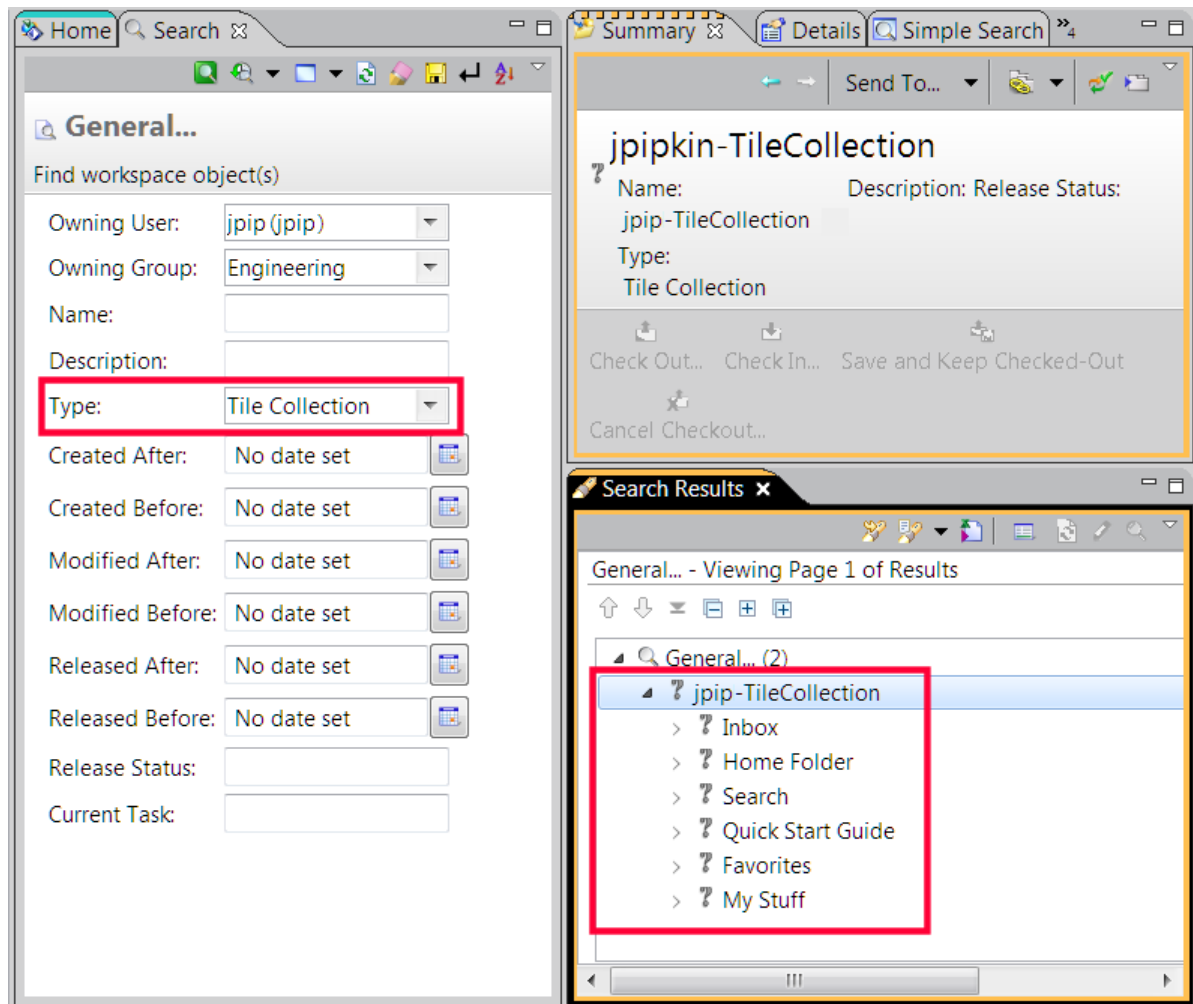
Specifies the relation type that attaches tiles to collections.

## Reset the home page

If a user unpins tiles from the home page, you can reset the home page to the default for the user's group and role. When you use this technique, the user loses all personal home page customizations.

1. Log on to the rich client.
2. Search for the **Tile Collection** object owned by the user.
3. Select the tile collection (for example, *user-name* - **TileCollection**).
4. Delete the collection.

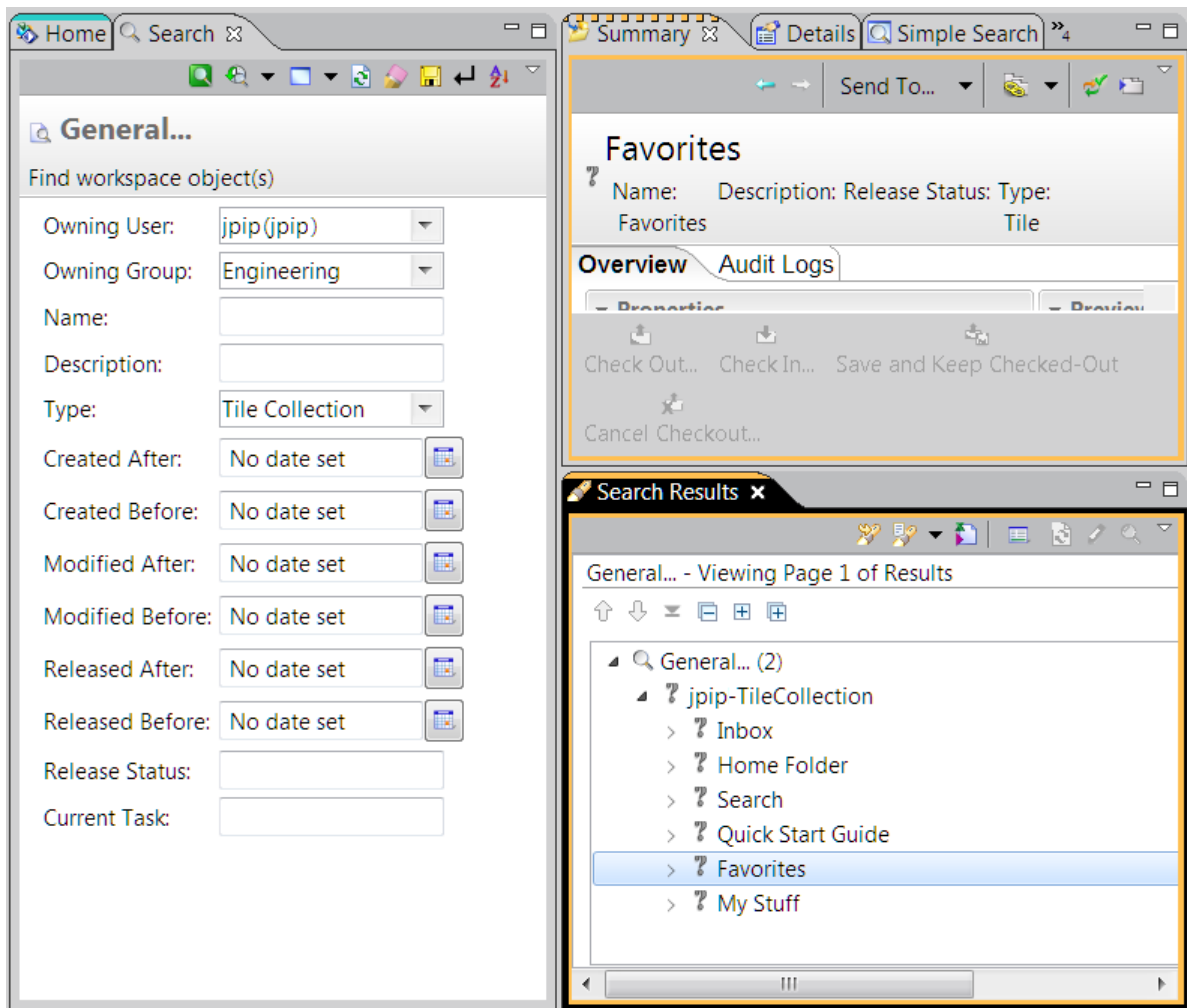
The next time the user logs on to Active Workspace, the tile collection is created for the user based on group and role.



### Selecting a user's tile collection

If instead you want to repin a tile to the user's home page, perform the following steps:

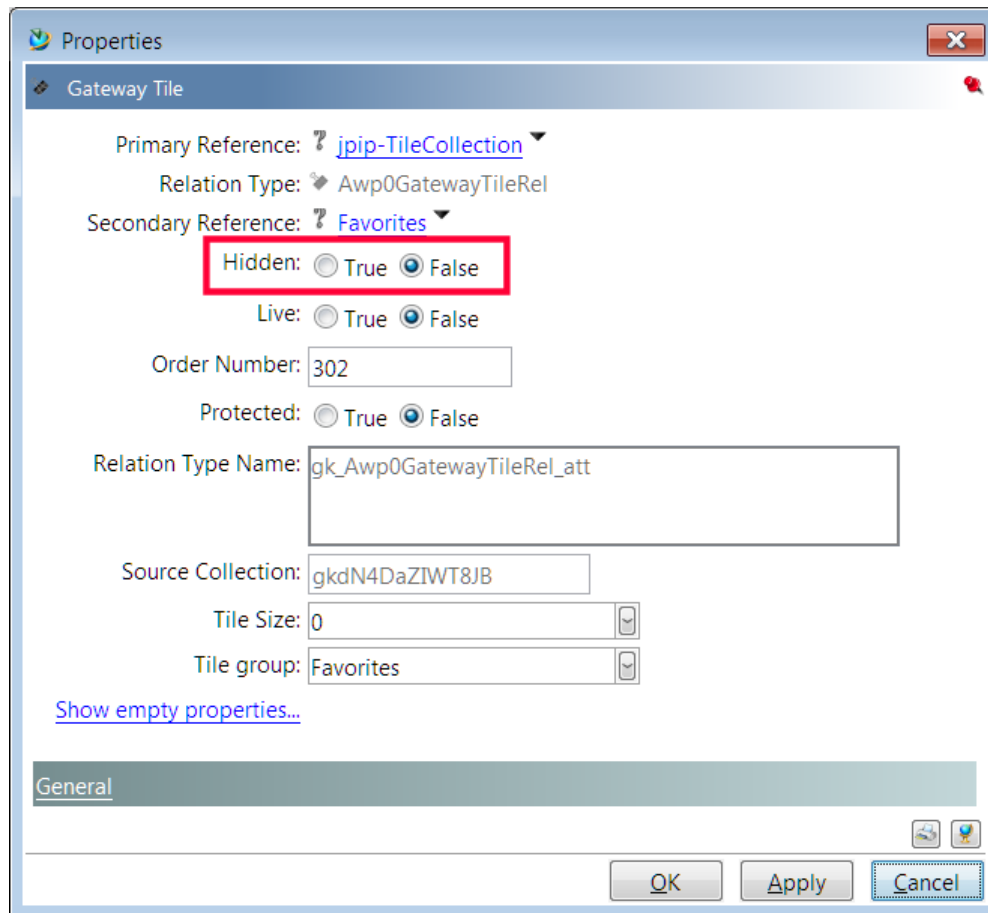
1. Search for the user's tile collection.
2. Select the unpinned tile (for example, **Favorites**).



**Selecting a tile in a user's tile collection**

3. On the menu bar, choose **Edit**→**Properties on Relation**.
4. In the **Hidden** property, select **False** and click **OK**.

The next time the user logs on to Active Workspace, the tile reappears on the home page.

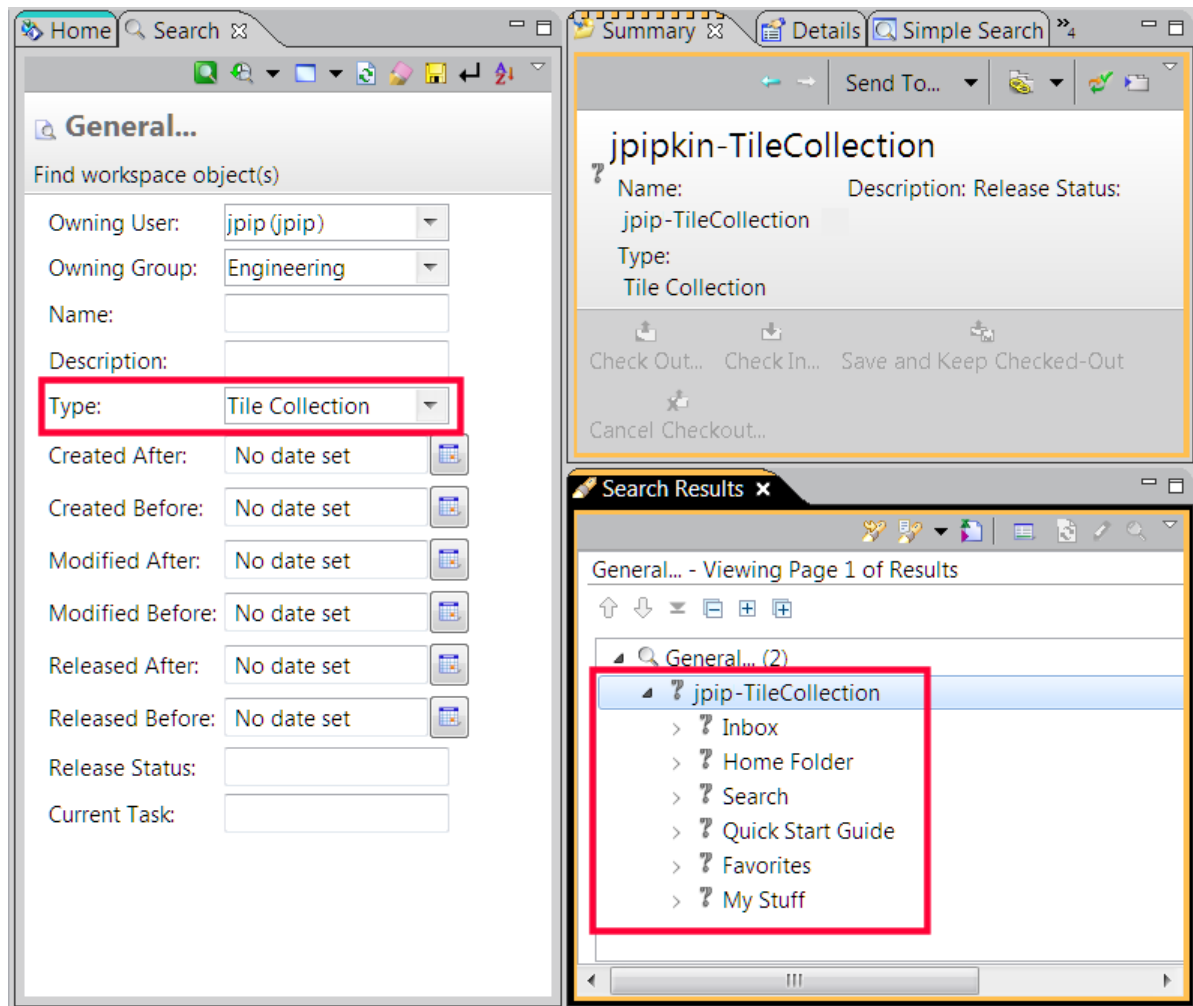


### Repin a tile

### Protect a tile

If you do not want users to be able to unpin a tile from their home page, you can protect it. A protected tile can be moved, resized, or reorganized into a new group, but it cannot be removed from the user's home page. The unpin button is not available for a protected tile.

1. Log on to the rich client.
2. Search for the **Tile Collection** object owned by the user.
3. Expand the tile collection (for example, *username - TileCollection*).

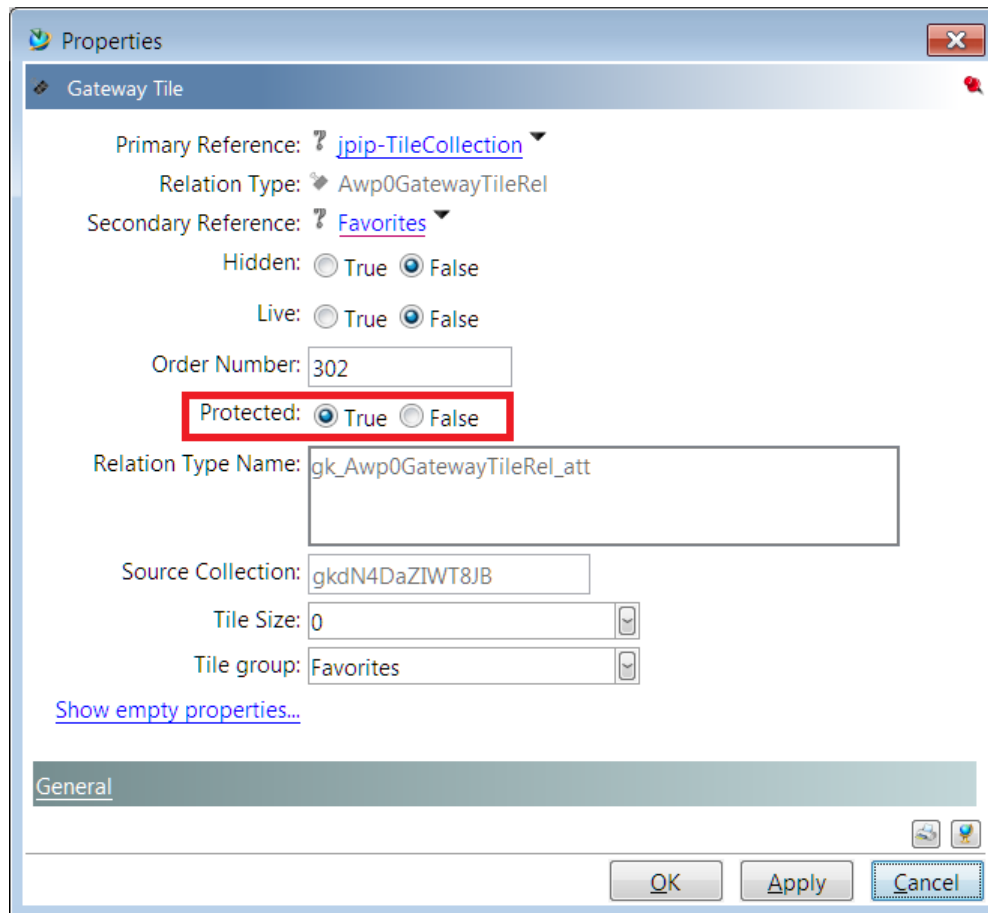


### Expanding a user's tile collection

4. Select the tile that you want to protect (for example, **Favorites**).
5. On the menu bar, choose **Edit**→**Properties on Relation**.
6. In the **Protected** property, select **True** and click **OK**.

The next time the user logs on to Active Workspace, the tile is protected.





Protect a tile

## Hide a tile

To hide a tile, you create a collection specific to a group, for example, and then hide the tile in that collection. This allows you to suppress something from the site collection for a specific group.

1. **Create a collection** for the group for which you want to suppress the tile.
2. In the Teamcenter rich client or thin client, search for the tile and choose **Edit→Copy**. Then, choose **Edit→Paste Special**.
3. Copy the tile you want to suppress and paste it into the newly created collection.
4. In the relations properties dialog box, select **True** for the **Hidden** property.
5. Click **Finish**.

### Note

To unhide the tile, open the tile collection object, right-click the hidden tile in the collection, and choose **Properties on relation**. On the **Hidden** property, select **False**.

## Create a new collection

You can create a collection specific to a group or role. You can then add tiles to the collection so that only users in the specified group or role see the tiles.

1. In the Teamcenter rich client, copy the object for the scope for your collection.  
 To use a group or role, perform an **Admin – Group/Role Membership** search, select **View Properties**, and then click **Copy** to copy the group or role.  
 To use a project, perform a **Project** search, select the project in the **Search Results** panel, and press **Ctrl-C** to copy it.
2. Choose **File→New→Other** and select **Tile Collection**.
3. In the dialog box that appears, specify the following:
  - **Name**  
 Type a recognizable name for your new collection, for example, **Engineering-Group-TileCollection**.
  - **Scope**  
 Paste the object that you copied in step 1.
4. Click **Finish**.

## Add a tile to a collection

1. In the Teamcenter rich client or thin client, perform a **General** search with **Type=Tile** to search for the tile, and then choose **Edit→Copy**.
2. Copy the tile to the clipboard.
3. Paste the tile onto the desired collection.  
 Choose **Edit→Paste Special**.
4. In the relation properties dialog box, specify the following:
  - **Order Number**  
 Enter the position of the tile within its group. The lowest number in a group indicates the position of the group relative to other groups.
  - **Tile Size**  
 Select the size for the tile.

Option	Size
<b>small</b>	1x1
<b>wide</b>	2x1
<b>large</b>	2x2

- **Tile Group**

Specify the group in which the tile should be included. You can choose from the predefined group names or type in a new name.

5. Click **Finish**.

## Create a new tile type

1. In the Teamcenter rich client or thin client, create a new tile template.

Choose **File**→**New**→**Other**, and then select **Tile Template**.

The screenshot shows the 'New Business Object' dialog box with the 'Tile Template Information' tab selected. The form contains the following fields and values:

- Name \***: milford map
- Description**: (empty text area)
- Icon**: idea
- Theme Index**: 2/ Highlight
- Tile Supported Sizes**: A list box showing '0/ Small', '1/ Wide' (selected), and '2/ Large'. Below it is a text input field containing '0, 1'.
- Action**: http://goo.gl/maps/TdQzD
- Action Type**: 1/ External Link
- Content Names**: A list box with three empty entries and '+' and '-' buttons below.
- Icon Source**: (empty dropdown menu)
- Template ID \***: awp0MilfordMap

- **Name** — This value is required.
- Specify the icon to display on the tile using one of two properties.
  - Icon** Enter a value which specifies the base name of the icon.
  - Icon Source** Paste a link to an object that contains the icon.
- **Theme Index** — This value determines tile color style.

- **Tile Supported Sizes** — This can be used to limit the available sizes for a tile type — if blank, all sizes are allowed.
- **Action** — This is run when the tile is clicked.
- **Action Type** — This is the type of action used.
- **Content Names** — This contains the display names if the tile has live data.
- **Template ID** — This is a unique name for the template. Use your solution prefix (**awp0** in this example).

2. Copy the new template to the clipboard.

3. Create a new tile object.

Choose **File**→**New**→**Other**, and then select **Tile**.

New Business Object	
Type	Tile Information
Name*	milford map tile
Description:	
Tile Template ID:	
Action Parameters:	
Tile ID*	Awp0MilfordMapTile
Pinned Object:	▼
Display Name:	milford office
Style:	
Tile Template*	milford map ▼

- **Name** — This value is required.
- Specify the tile template to use.  
**Tile Template ID** This is the ID string of the template (used by automated create, ignore when manually creating).

**Tile Template** Paste the tile template reference here (there should be one on your clipboard from a previous action).

- **Action Parameters** — Place arguments here to append to the template action.
- **Tile ID** — This is a unique name for the tile. Use your solution prefix (**awp0** in this example).
- **Pinned Object** — This is a reference to the Teamcenter object to which this tile refers (used for pinned objects).
- **Display Name** — This the name of the tile as shown in the gateway.

**Note**

The new tile type is now created, but it is not displayed until it is included in a tile collection. You can create a new collection or contribute to an existing one.

## Create a tile template that creates a Part

Normally, after logging on, if a user wishes to create a new Document, Part, or Design, for example, a user must click their **Home** tile, and then click the **Create** command, and then choose which type object they wish to create. If this is a common occurrence, creating a new tile designed specifically for this functionality will save the user time.

Although you could create a new tile template by using the URL generated when you navigate to the create panel, this URL is static, and will not update as needed if there are changes in the Active Workspace server URL, for example.

In this example, you will create from a provided template a tile that presents the user with the **Part** creation dialog box with a single click, and is robust enough to be valid even if the Active Workspace server changes. This template is called **Awp0HomeFolderCreateTemplate**.

1. Use the rich client to create a new tile, but use the provided **Awp0HomeFolderCreateTemplate** tile template instead of creating your own.
2. Fill in the fields as normal, but for the **Action Parameters** field with the name of the type you wish to create.

cmdArg=**Part**

3. Paste the new tile into a tile collection.

**Note**

If you wish to provide a short list of types, you can modify the command argument with additional types, separated by semi-colons.

```
cmdArg=Part;Document;Design
```

## Action styles

Action style	Value	On click
Default	0	Go to provided history token.
External link	1	Open the provided URL in a new window or tab.
Static resource	2	Open the provided static resource from the WAR file. Resource link is relative to the host HTML file.
Command	3	Run the command with the provided ID.

### Note

In all but the default style, the history token is used to provide the name or Location of the URL, resource, or command.

## Theme index

If you choose	The theme uses the
0	Light tile color.
1	Medium tile color.
2	Highlight tile color.
3	<i>Alternate tile color.</i>

### Note























The alternate theme index is not used by any of the provided themes.

## Tile sizes

If you choose	By entering	The tile is
small	0	1 x 1
tall	1	1 x 2
wide	2	2 x 1
large	3	2 x 2

## Provided icons

Dark and light versions of each icon is automatically retrieved as needed. The base name + **\_dark** or **\_light** is retrieved.

	announcement		idea
	change		inbox
	changes		issue
	document		issuepr
	favorites		mail
	favoritesfolder		part
	feedback		pdf
	folder		people
	help		power
	home		requirement
	homefolder		search

### Note

The **\_dark** version of each icon is shown.



## Configuring page layout using style sheets

### Introduction to using XML rendering templates (XRT) with Active Workspace

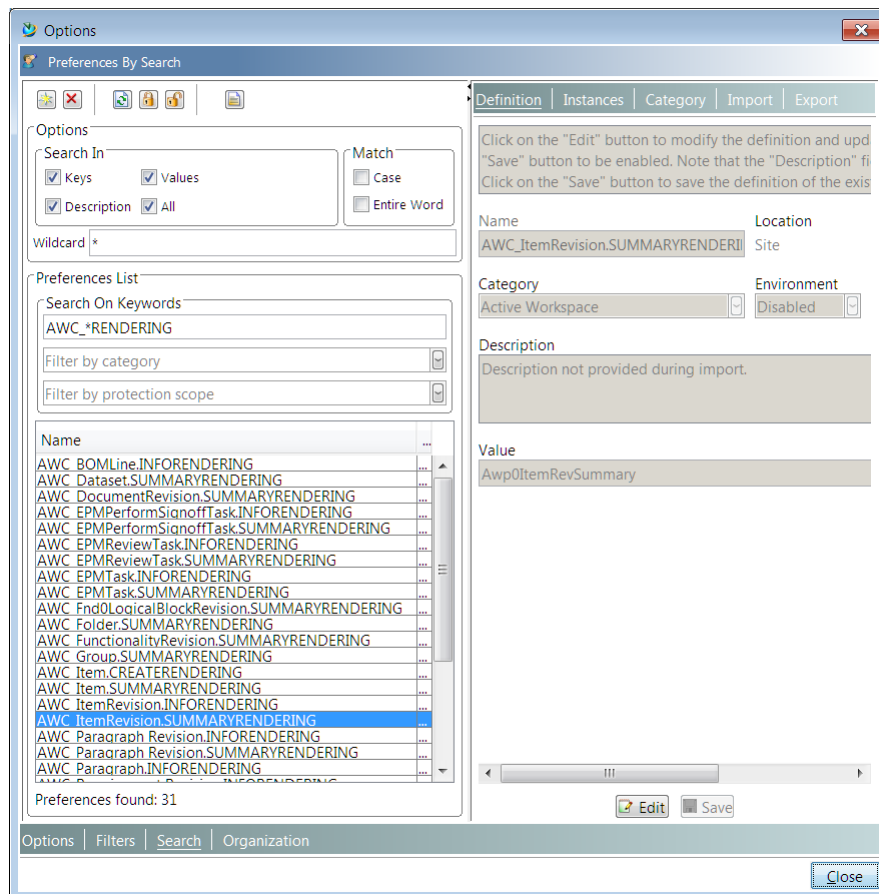
XRT files use the XML format. They are used to configure layout in Teamcenter clients, including Active Workspace, the thin client, and the rich client, based on object type, user group, and role. XRT files are also commonly referred to as style sheets. However, they do not follow either CSS or XSL standards, nor are they intended to perform any transformations.

In Active Workspace, they control areas such as the *secondary work area* and the *tools and information panel*.

You edit XRT files in the rich client. You can locate Active Workspace style sheet preferences in the rich client by choosing **Edit**→**Options**→**Search** and looking for preferences whose names follow this format:

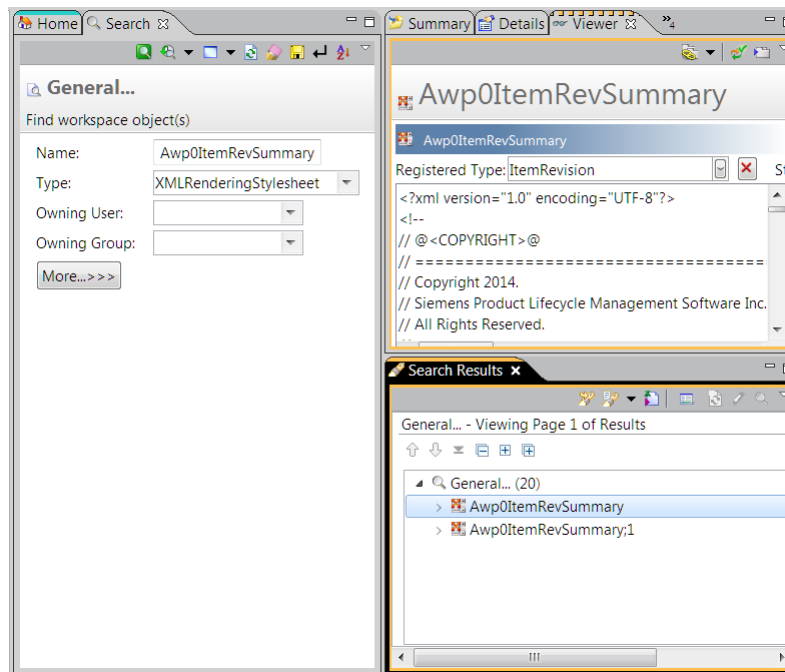
**AWC\_<type-name>.\*RENDERING**

The value of the preference points to a style sheet dataset name.



### Searching for Active Workspace style sheet preferences in the rich client

To edit a style sheet, look up the preference value and then query for the corresponding dataset. You can locate the style sheet files in the rich client by searching for **XMLRenderingStylesheet** files.



Searching for style sheet files in the rich client

**Tip**

For complete instructions about how to work with style sheets, see the topics on using style sheets in *Client Customization* found in the Teamcenter help collection.


## Considerations for using XRTs in Active Workspace

Although Active Workspace uses XRTs like the other clients do, there are some differences:

- Active Workspace XRT rendering preferences have **AWC\_** added to the beginning of the preference name, allowing for the assignment of style sheets that are unique to Active Workspace.
- Layout of Active Workspace XRTs in landscape mode is wide compared to the rich client and thin client and, therefore, requires multiple columns.
- Do not alter the out-of-the-box XRT files for Active Workspace. If you do, the next time you upgrade Active Workspace, the out-of-the-box files are replaced and you lose your changes. Instead, create XRT files with unique names and assign them using the **AWC\_<type-name>.SUMMARYRENDERING** preferences.
- Active Workspace can share XRT files with the rich client for the **Summary** tab. The default summary XRT preference for Active Workspace is **AWC\_ItemRevision.SUMMARYRENDERING**. If you remove this preference, Active Workspace uses the default summary XRT preference used by the rich client and thin client: **ItemRevision.SUMMARYRENDERING**.
- If you do not want to use the header on the overview page, you can remove it from the XRT files used by the **AWC\_<type-name>.SUMMARYRENDERING** preferences.
- Active Workspace supports:

- o Multiple pages with **visibleWhen** tags
  - o Columns
  - o Sections
  - o **objectSet** tables (using tile/list or table modes). Lists in the rich client appear as tiles in Active Workspace.
  - o Labels
  - o Breaks
  - o Separators
- The following are not currently supported in Active Workspace:
    - o Custom rendering hints
    - o **conditions** tag
    - o **GoverningProperty** tag

## Configure the information panel using XRTs

The Active Workspace information panel displays details about the opened object and is accessed by clicking the  button.

### Information panel

You use XRT datasets to configure the layout of the information panel. By default, there is an XRT dataset for **WorkspaceObject** and **ItemRevision** object types. To modify the information displayed in the information panel for other types, you must create an **XMLRenderingStylesheet** dataset, attach an XML file to it, and then create a preference to point to the dataset. The XRT is registered using the **AWC\_<type-name>.INFORENDERING** preference.

1. Create a dataset of type **XMLRenderingStylesheet**.

#### Tip

You can copy an existing XRT dataset and rename it rather than create a new one. Find existing XRT datasets in the rich client by searching for **XMLRenderingStylesheet** dataset types. Then copy an existing XRT dataset by selecting it and choosing **File→Save As**. Make sure you change the named reference file attached to the dataset to point to a unique file name.

2. Attach the XML file to the new dataset as a named reference.

Siemens PLM Software recommends that your XRT be set up to display content in the information panel as follows:

- Limit to one or two pages
- Limit to one column per page
- Use list displays for object sets

Keep in mind the following:

- Keep it simple. Do not make the layout the same as the summary or overview pages.

- Active Workspace supports multiple pages with the **visibleWhen** tag, sections, and **objectSet** tables (use the tile/list mode to fit the narrow display).
  - The XRT used in the user interface is based on the selected object's hierarchy. For example, if you select an **Item** object type, but it does not have an XRT associated with it, the XRT for **AWC\_WorkspaceObject.INFORENDERING** is used because an **Item** is also a **WorkspaceObject**.
3. Use the rich client to create a preference using the following parameters:
    - **Name:** **AWC\_<type-name>.INFORENDERING**, for example, **AWC\_WorkspaceObject.INFORENDERING**.
    - **Value:** Name of the dataset created in step 1.
    - **Scope:** Site preference.

## Active Workspace-specific style sheets

If you want to have different style sheets in Active Workspace than you have in other Teamcenter clients, you can create **AWC\_** preferences in the rich client to tell Active Workspace which style sheet to use. This has no effect on the other clients or on any customer-created style sheets.

1. Create a dataset with a type of **XMLRenderingStylesheet**.
2. Attach the XRT style sheet to the new dataset as a named reference.
3. Use a text editor to edit the style sheet as necessary.
4. Create a preference in the rich client using the following parameters:
  - **Name:**

**AWC\_<type-name>.SUMMARYRENDERING**  
**AWC\_<type-name>.CREATERENDERING**  
**AWC\_<type-name>.INFORENDERING**

For example:

```
AWC_WorkspaceObject.INFORENDERING
```
  - **Value:** Name of the dataset created in step 1.
  - **Scope:** Site preference.

When rendering style sheets, Active Workspace first searches for **AWC\_<type\_name>.[SUMMARYRENDERING, CREATERENDERING, ...]**. If no match is found, it searches for **<type-name>.[SUMMARYRENDERING, CREATERENDERING, ...]**. If it still does not find a match, it continues with the standard lookup mechanism for style sheets.

It is possible to register style sheets to a specific location, sublocation, or object type in Active Workspace.

Use the following format to create the registration preferences:

```
type.location.sublocation.SUMMARYRENDERING
```

- **type** specifies the type of object. PartRevision or DesignRevision, for example.
- **location** specifies the location in the UI. For example, if the context is `com.siemens.splm.clientfx.tcui.xrt.showObjectLocation`, then the location is **showObjectLocation**.
- **sublocation** specifies the sublocation in the UI. for example, if the context is `com.siemens.splm.client.occmgmt:OccurrenceManagementSubLocation` then the sublocation is **OccurrenceManagementSubLocation**.

As with normal registration preferences, the value of this preference is the name of the dataset. When rendering a page, the system will search for the most specific case to the most general.

- ```
type.location.sublocation.SUMMARYRENDERING
```
- ```
type.location.SUMMARYRENDERING
```
- ```
type.SUMMARYRENDERING
```

If none of the above preferences are found for the object, the immediate parent type will be searched in the same manner. This process continues until a match is found.

## Modular style sheets

It is possible to use the `<inject>` tag to refer to another **XMLRenderingStylehseet** dataset that contains a block of XML rendering tags. This XML rendering style sheet would be incomplete on its own, normally containing only a single page or section, but they allow a modular approach to style sheet design and maintenance.

In the example below, a second **XMLRenderingStylehseet** dataset exists with the name **myXRTblock**.

There are two methods of specifying which dataset is to be used.

- Directly, by referring to the name of the **XMLRenderingStylehseet** dataset.

```
<inject type="dataset" src="myXRTblock"/>
```

- Indirectly, by referring to a preference which contains the name of the dataset.

```
<inject type="preference" src="additional_page_contributions"/>
```

Then create the **additional\_page\_contributions** preference, containing the value **myXRTblock**.

If the preference contains multiple values, then each dataset will be located and injected in order.

**Note**

As well formed XML files must have a root node in order to be well-formed, the XRT you inject must be wrapped in a **<subRendering>** element.

```
<subRendering>
  <label text="This text will get injected."/>
</subRendering>
```

Someone leveraging injection must think about the resulting XML file, so that the resulting XRT will be correct. The injection mechanism does not make any assumptions about where it is injecting data.

## Working with HTML panels in XRT

### HTML panel in Active Workspace XML rendering datasets

The **<htmlPanel>** tag supports URL and HTML content to be included in an XML rendering.

- **<htmlPanel>** can be specified as a child tag in **<page>**, **<column>**, and **<section>** tags.

**Note**

The **<htmlPanel>** tag is supported only in Active Workspace XML renderings.

- The URL and HTML content can have the value of properties of the currently selected object introduced into them. This technique is known as *data binding*.

Instead of embedding HTML directly into an XRT, it is possible to use the **<inject>** tag to refer to an HTML dataset instead. There are two methods of specifying which HTML dataset is to be used. The HTML dataset must contain only valid HTML code, with no XML style sheet tags.

In the example below, an HTML dataset exists with the name **myHTMLblock**.

There are two methods of specifying which dataset is to be used.

- Directly, by referring to the name of the HTML dataset.

```
<inject type="dataset" src="myHTMLblock"/>
```

- Indirectly, by referring to a preference which contains the name of the dataset.

```
<inject type="preference" src="additional_page_contributions"/>
```

Then create the **additional\_page\_contributions** preference, containing the value **myHTMLblock**.

If the preference contains multiple values, then each dataset will be located and injected in order.

**Caution**

Uncontrolled JavaScript code included in the HTML panels can be used to exploit a security issue or other network policy violation. System administrators must exercise care to ensure the XML rendering preferences, datasets, and any WAR build changes are monitored and require DBA level access.

## Specifying a URL

The **src** attribute is used to specify the fully qualified URL as the source of the content to display in a new **iframe**.

- The **src** attribute can be complete or can contain references to various properties in the currently selected object.
- You can specify multiple properties.

### Example: simple static URL

To display the contents of the given URL within the **iframe**:

```
<htmlPanel src="https://www.mycorp.com/info"/>
```

### Example: include a property value in a URL

To display the contents of the given URL with the current value of the **item\_id** property used as the ID in the URL query:

```
<htmlPanel src="https://www.mycorp.com/info?id={{selected.properties['item_id']}}"/>
```

If the **item\_id** property for the selected object is **Part 1234**, the final **<iframe>** tag is encoded as:

```
<iframe src="https://www.mycorp.com/info?id=Part%201234"/>
```

#### Note

The resulting URL is made *safe*—all characters are encoded to assure they are valid for a URL.

For example, any space characters in the property value for the object are encoded as **%20** in the final URL.

## Specifying HTML content

Any **CDATA** section included within the **<htmlPanel>** tag is used to set the inner HTML of the panel in the rendering. This means that HTML tags in the **CDATA** section are placed in the panel verbatim.

Any occurrences of properties within double curly brackets (**{{.....}}**) are replaced with the values for those properties in the currently selected object.

### Example: simple CDATA section

In this example, the current **object\_string** value is placed in a level two header (**<H2>**) and the specified **item\_id** value is used to complete the sentence that begins with **The ID is**:

```
<htmlPanel>
  <![CDATA[
    <div>
      <H2>{{selected.properties['object_string'].uiValue}}</H2>
    </div>
    <div>
      The ID is {{selected.properties['item_id'].uiValue}}
    </div>
  ]]>
</htmlPanel>
```

## Data binding

*Data binding* is a way to connect the current property values of an object model to attributes and text in an HTML content view.

- This binding mechanism follows the conventions of [AngularJS \(by Google\)](#).
- A section of HTML to be replaced with some other value is enclosed in double curly brackets, `{{xxxx}}`, with `xxxx` indicating a reference to a property in the current *scope* object.

For more information, see <https://docs.angularjs.org/guide/databinding>.

## Specialized HTML tags

In addition to standard HTML tags such as `<B>`, for bold text, and `<BR>` (to force a line break), XML rendering can use new specialized tags to simplify the work to display and edit Teamcenter data. This new tag reduces the amount of HTML required to accomplish common tasks.

### Note

This new tag is implemented as an AngularJS *directive*. For more information about AngularJS directives, see <https://docs.angularjs.org/guide/directive>.

### <aw-property>

The `<aw-property>` custom tag is used to simplify label and value display for Teamcenter properties. It also handles the editing of these properties when appropriate.

You can use the `<aw-property>` tag to display all supported Teamcenter property types including single and multiple values, images, object sets, and object references.

The `<aw-property>` tag supports these attributes:

- **prop**  
For labels and values, this required string attribute specifies the property to display. This attribute supports data binding value substitution.
- **hint**  
This optional string attribute specifies variations in the way a property is displayed.  
The valid values are:
  - o **label**
  - o **objectlink**
- **modifiable**  
This optional Boolean attribute specifies whether the property can be modified during edit operations. It applies only when the property is otherwise editable.

### <aw-frame>

The `<aw-frame>` custom tag is used to simplify displaying URL contents in an **iframe**. It supports a single **src** attribute.



The **<aw-frame>** attribute inserts HTML structure and CSS styling to correctly display the **iframe** using the full width and height available in the page, column or section in which it is placed.

There are limitations on what can be shown in an **iframe**:

- Not all external URLs can be used within an **<iframe>** tag.  
Some sites detect this tag and prevent their content from being displayed within an **<iframe>** tag. This is a way sites control content display.
- Some browser, site, and network settings prevent some scripts from running if they come from a Location other than the root Location of a page.

This capability, also called *cross-site scripting*, is a potential source of network attack.

For more information, see [http://en.wikipedia.org/wiki/Cross-site\\_scripting](http://en.wikipedia.org/wiki/Cross-site_scripting).

This tag supports the following attribute:

#### **src**

This required attribute specifies the URL to be displayed in the **iframe**.

The **src** attribute supports data binding value substitution.

### Specifying CSS styling

Creators of HTML content are free to specify their styling in their application. However, all existing Active Workspace CSS styling selectors are available for use in HTML content contributed by the **<htmlPanel>** tag. Use these existing styling selectors to save time and ensure UI consistency.

#### **Note**

For a list of all available CSS selector class names and their intended uses, see the **thinclient/styleguide.html** file available in the development environment.



## Chapter 12: Examples using code scaffolding

### Code scaffolding overview

These customizations require changes to the core client application, the **awc.war** file. You will create one or more modules to contain your custom code. After performing these modifications, you must recompile the WAR file using the **gwtCompile** script.

Active Workspace makes use of *Node.js* and *gulp*. Both of these are already part of the Active Workspace client installation, as well as the required node modules and dependencies.

The machine on which the WAR file is built must have:

- Java JDK 7 64-bit.
- A minimum of 16 GB of free physical RAM (unused by other applications) to avoid paging. Siemens PLM Software recommends that the machine used have a minimum of 24 GB.

Use the provided **generateModule** script to add the following customizations to Active Workspace:

- Theme
- One-step command
- Type icon
- Location
- Location command
- Sublocation
- Navigation panel
- Tools and information panel
- Property policy

### Use generateModule to create a new module

The **generateModule** script is located in the Active Workspace **STAGE** directory, **TC\_ROOT\aws2\stage**. The script creates basic boilerplate code for the various module components. It must be run from your new module's directory.

Three main steps create Active Workspace customizations:

- Create the Java and GWT modules.

These contain your customizations.

- Create module components.

For example: sublocations, one-step commands, navigation panels, and so on.

- Create the **kit.json** file.

All of this can be organized within your staging location's source directory.

### Create the Java and GWT modules

1. Create a new directory at **STAGE\src** to contain your custom module.

**STAGE\src\myModule**

2. Change to your new module directory.

```
cd STAGE\src\myModule
```

This and all following steps must be performed from a command line.

3. Run the **generateModule** script.

```
..\..\generateModule
```

The script does not require any arguments. If none are provided, it will prompt you to enter the information it needs.

4. Create the files for a new Java module by following the prompts.

```
Enter type to generate: java
Enter module name (Required): myJavaModuleName
Enter module description: My Java module description
[12:34:56] [Info] Creating java module
[12:34:56] Successfully created java module myJavaModuleName
```

This creates a **src** directory within your module directory and the **module.json** file.

5. Run the **generateModule** script again, this time to create the gwt module information.

```
Enter type to generate: gwt
Enter fully qualified GWT module id (Required): com.mycompany.mymodule.myGWTModuleId
Enter GWT module description: My gwt module description
[12:34:56] [Debug] Creating gwt module myGWTModuleId
[12:34:56] Successfully created gwt module myGWTModuleId
```

This creates the **gwt.xml** file for the module as well as several Java source files under the **src** directory for localization, interface, injector, resources, and so on.

Your new module is ready to be populated with custom Active Workspace components.

### Create module components

Continue to use the **generateModule** script to create locations, type icons, tool and information panels, and so on.

**Caution**

**generateModule** automatically updates the files in your module directory each time you run it. However, once you begin to edit these files to add your custom code, the script may not be able to make the required changes. It is recommended to create all the module pieces that you require before beginning to write your code.

**Create the kit**

The final step to include a custom module into the war build is to create a **kit.json** file. This file tells the war build to include your custom module and add the required `inherits` statement into the primary GWT XML file used for the GWT compile. An existing **kit.json** file can be copied, or the **generateModule** script can be run using the **kit** type. The **kit name** value must be unique within your stage directory.

```
Enter type to generate: kit
Enter kit name {Required}: MyKit
[12:34:56] [Info] Creating kit.json
```

**Note**

Once all your customization code has been written, use the **gwtCompile** script to build your new WAR file.

**The module.json file**

A module file is only compiled and added to the WAR file if it is referred to by a kit file. A module may list other modules as dependents.

The **module.json** file contains the following information:

<b>name</b>	This is the name of the module.
<b>type</b>	This is the type of module. Currently, this value will only be <b>java</b> .
<b>dependencies</b>	This is a list of other module names that are required.

**The kit.json file**

Kit files are discovered by the **gwtCompile** script. They list the modules that are compiled and added to the WAR file.

The **kit.json** file contains the following information:

<b>name</b>	This must be a unique name amongst the other kits in your stage directory.
<b>modules</b>	This is a list of the modules (defined by <b>modules.json</b> ) that are part of this kit.
<b>gwtModulesDeps</b>	This is a list of GWT modules references which are insterted into the primary GWT XML file as <code>inherits</code> statements.

**Upgrading from hand-written or Maven-type modules**

The following steps detail how to upgrade any existing modules that were created from scratch or by using the Maven archetypes.

1. Copy or link your existing customization code under **STAGE/src**.

2. In the directory with your existing **pom.xml** file, create the following **module.json** file. You can use the **artifactId** from your **pom.xml** file for **your-module-name** if you prefer.

```
{
  "name": "your-module-name",
  "type": [
    "java"
  ]
}
```

3. If you need to declare a dependency on another one of your custom modules, you can add a **dependencies** array to declare this dependency.

```
{
  "name": "your-module-name",
  "type": [
    "java"
  ],
  "dependencies": [
    "your-other-module-name"
  ]
}
```

4. Delete the **pom.xml** file.

## Creating custom themes

### Cascading style sheets (CSS) in Active Workspace

The Active Workspace client uses CSS3 to control the layout and styling of the web application. CSS is an industry standard that provides a modern interface consistent across multiple platforms.

#### CSS layers

Active Workspace uses five cascading levels of CSS evaluated in the following sequence:

1. **reset.css** and **base.css**

Specifies appearance of base HTML elements such as heading one (**<H1>**) and line break (**<BR>**) tags. It also specifies common base definitions rules.

2. **layout.css**

Specifies top-level elements such as location, sublocation, and panel classes. Classes should be reused in higher level style sheets.

3. **module\_name.css**

There are many modules with only one CSS file per module. It is located in the **staticresources** folder in each module.

4. **state.css**

Specifies states such as active, hidden, and expanded.

5. **ui-lightTheme.css** and **ui-darkTheme.css**

Only one theme is applied at any given time. The customizer uses **generateModule** to create **ui-custom\_theme\_name.css**.

**Note**

Do not modify provided theme files. Always create a custom theme file for modifications.

During the build phase, all CSS files except themes are concatenated into a single **main.css** file.

The files are concatenated in this order:



**Note**

Theme files are not concatenated. The current theme ID is stored in a cookie and toggled dynamically.

All module files are concatenated between the **layout.css** and **state.css** files, but the order of the modules is not guaranteed. If you have the same tag in different modules, unexpected results may occur.

When CSS classes override each other, the last file takes precedence. The **reset.css** file is overridden by all, and the **state.css** file overrides everything except the theme CSS.

## Location of CSS files in Active Workspace

Framework-level CSS files (**reset.css**, **base.css**, **layout.css**, **state.css**, and theme CSS files) are located in the **fx-ui staticresources** project.

Module CSS files are located in the *module/staticresources* directory for each module.

## CSS naming pattern

CSS files names are the unique names of the module, such as **search.css**, **change.css**, **viewer.css**, and **docMgmt.css**.

CSS class names follow this pattern: **aw-xxx-yyy**.

- **aw**: Active Workspace
- **xxx**: the containing CSS file name, without the .css extension
- **yyy**: the semantic meaningful name

**Note**

Do not use names based on current position (**top**, **LeftHandSide**) or style characteristics (**RedWithPadding**).

Example CSS class names:

```
aw-base-scrollPanel
aw-layout-commandBarHorizontal
aw-search-searchContainer
aw-layout-popup
```

## Styling guide

The styling guide, located in `war-location/Thinclient/styleguide.html`, shows all objects as they should be rendered throughout Active Workspace.

The styling guide is automatically synchronized because it uses the actual application CSS.

## Custom theme overview

Active Workspace supports predefined user interface (UI) *themes* delivered with the framework. This provides a centralized mechanism to control the overall UI look-and-feel across client applications.

Themes control:

- The background color or background image for web pages.
- Text styling for fonts, color, and sizing.

Themes are not intended to control the specific *chrome* (size, location, behaviors) of the UI for widgets. Rather, themes provide a common definition for the overall styling of colors, fonts, and sizing to ensure consistency in look-and-feel across all client UI and widgets.

- Themes are driven from a main CSS definition (per theme).
- Users can change themes dynamically while running the client.



- Theme selection is retained in subsequent sessions.

Each theme is defined in a main cascading style sheet (CSS) file.

This provides a layer of abstraction for the component level CSS, which accesses common CSS class names.

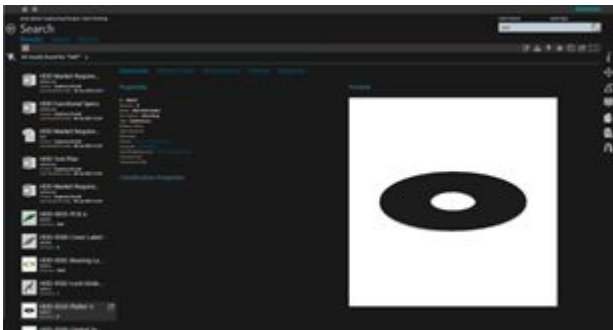
Two themes are provided with Active Workspace:



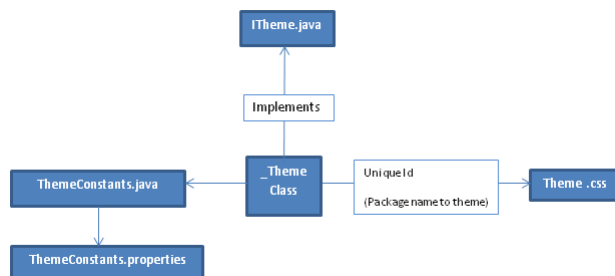
- **ui-lightTheme.css**: low contrast colors



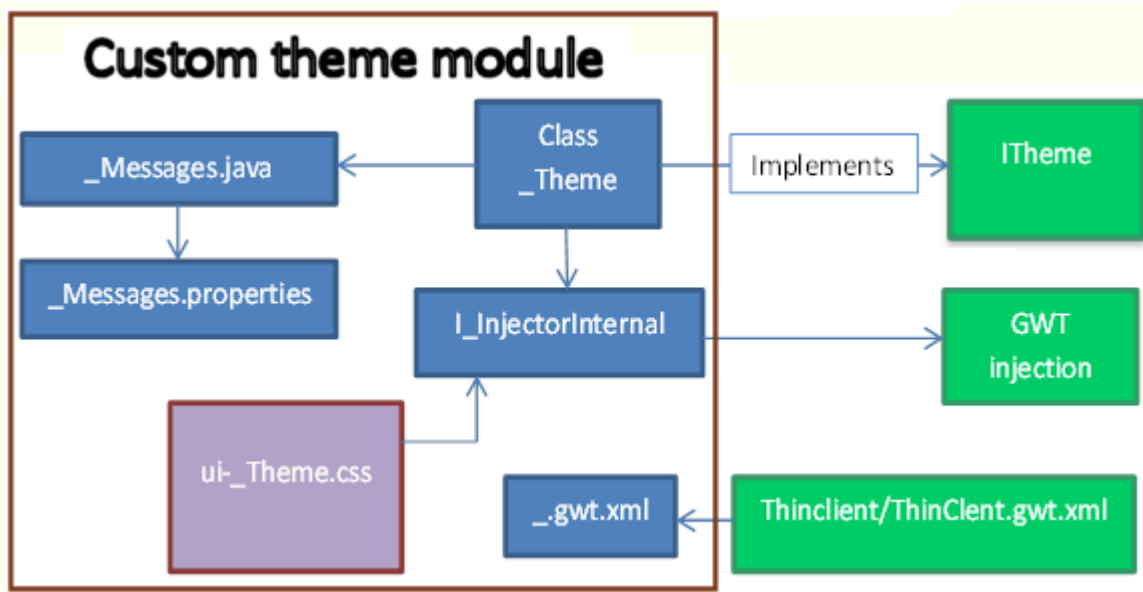
- **ui-darkTheme.css**: high contrast colors



To add a custom theme to Active Workspace, implement the **ITheme** class.



- The **\_Theme** class implements **ITheme**.
- *UniqueId* is the theme package name.
- The displayed string is stored in the theme constants.

**Note**

The **ui-darkTheme.css** and **ui-lightTheme.css** files are located in *war-location/thinclient/*. Refer to these files while creating your **ui-customTheme.css** file.

## Theme CSS classes

The Active Workspace \*.css files contain CSS classes.

```
.aw-base-toolbar,
.aw-layout-LocationTitle {
    color: #55a0b9;
```

These CSS classes control objects in the \*.ui.xml files:

```
<!-- Location Title -->
<g:InlineLabel ui:field="LocationTitle" styleName="aw-layout-LocationTitle"/>
```

**Note**

The **ui-customTheme.css** file is empty when the module is created.

The **ui-darkTheme.css** and **ui-lightTheme.css** files are located in *war-location\thinclient*. Use these files as guides to create your **staticresources\ui-customTheme.css** file.

## Edit the CSS live

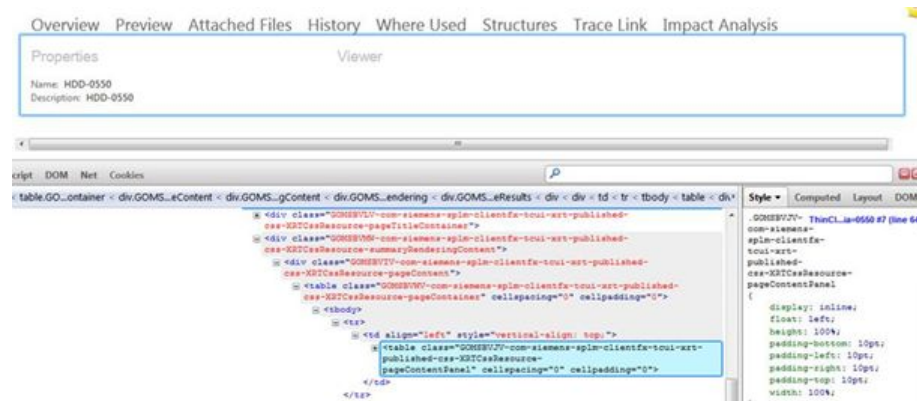
You can use a web browser to view and edit the generated HTML and CSS files. This lets you see how the HTML was constructed and lets you style changes before adding them to the code and without rebuilding and reloading for each change.

**Note**

The example uses the Mozilla Firebug plug-in. Internet Explorer and Google Chrome provide similar tools that can be accessed by pressing F12.

1. Restart the server with your edits in place, and log on.
2. Press F12 to start the web browser style viewer.
3. Select an object in the user interface.

The style viewer highlights the selection in the user interface and shows the supporting HTML and applied style for the selected object.



The style viewer lets you edit either the HTML or the CSS file; the running browser session is updated dynamically.

After your changes are made, they can be copied into the **ui-customTheme.css** file.

## Add a new theme to your module

### Note

If you have not created a module yet, follow the steps to *Use generateModule to create a new module*.

1. Open a command prompt and change to your module directory.

```
cd STAGE\src\myModule
```

2. Run the **generateModule** script, and use the **theme** argument.

```
Enter type to generate: theme
Enter theme name (Required): myThemeName
[12:34:56] [Debug] Creating theme myThemeName
[12:34:56] Successfully created theme myThemeName
```

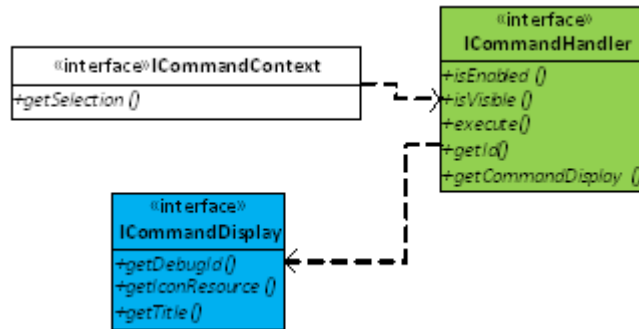
This updates the **gwt.xml** file, and creates a **css** file for your new theme within your module directory.

## Contributing commands

### Command contribution constructs

To contribute commands, you need to understand command-related constructs and command context, display, and handlers.

## Command-related constructs



- **ICommandDisplay**

Provides the visual characteristics, including the image resource and the title, for a command.

These separated visual aspects support alternate visual representations for a command behavior and separates visual concerns from functional concerns.

- **ICommandContext**

The associated application state identifies the target, typically a selected object or data, for the command.

- **ICommandHandler**

Represents the functional aspects and state, whether the command is enabled and visible, and what occurs when the command is called.

## Command display

**ICommandDisplay** represents the visual characteristics of a command instance, including the image resource, text display string, and debug identifier values.

The **ICommandDisplay** information is provided by the **ICommandHandler** reference.

## Command context

**ICommandContext** is the run-time state against which the command operates.

- The current implementation is tied to a selection context, required in most situations. When a nonselection context is required, you can create a specific subclass that ignores or stubs out the selection-oriented portion of the context.
- Changes in the context may initiate state changes in **ICommandHandler**.

**ICommandHandler** observes the command context and can react to changes. When the command handler is sensitive to the current context, it can change state accordingly and become a listener (observer) of the command context.

## Command handler

**ICommandHandler** provides the focus for invocation and setting run context, and tracks the visual state (**Enabled/Visible**).

- The handler is an observable object that lets you monitor state changes on the instance.

- Each **ICommandHandler** has a defined unique command handler.
- **ICommandDisplay** is also accessed from **ICommandHandler**.
- The command handler ID must follow a fully qualified name pattern to maintain unique identifiers.

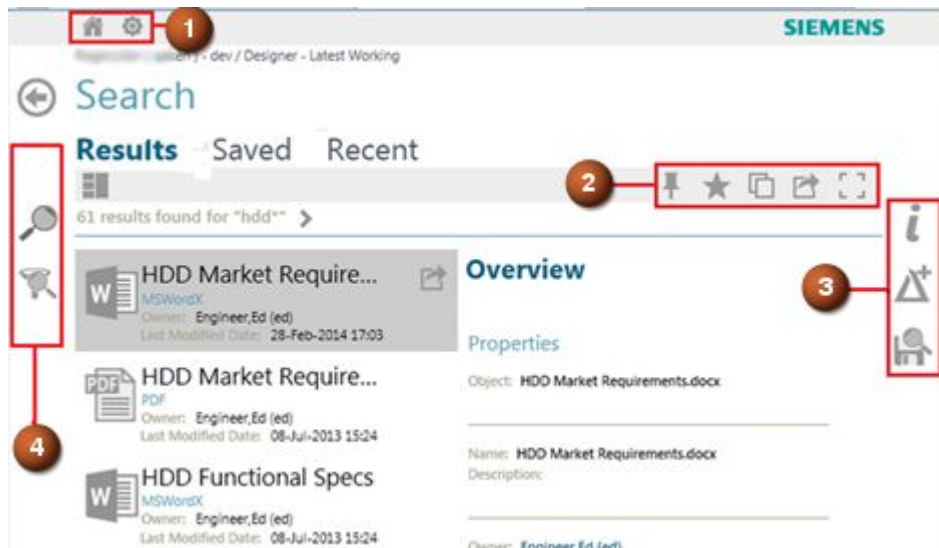
**Note**

Use the package name of the GWT module and a functional name for the particular command to help define a unique ID.

GWT injection (GIN) generates errors if a duplicate is encountered.

## Command types

Commands are classified based on category, such as whether they require user input and where the commands are contributed.



- 1 Global commands These commands appear in the global toolbar. They are present on all pages and are universally applicable.

Examples include **Home** and **Change Theme**.

**Note**

A module can contribute a command in the global toolbar, but this is not recommended. Only commands that apply across all locations and sublocations should appear on the global toolbar.

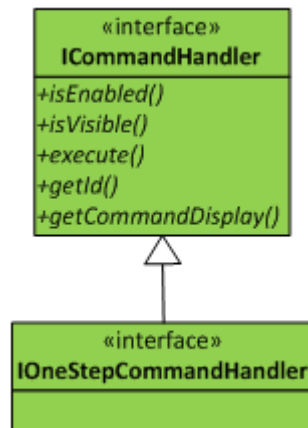
- 2 One-step commands These commands do not require user interaction. Examples include **Open Object** and **Cut Object**.

See *One-step commands* for an example of adding a one-step command.

- |   |                               |                                                                                                                                                                                                                                                                                                                                                                                       |
|---|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | Tool and information commands | <p>These commands require user interaction or inputs to get input values or confirm steps.</p> <p>Examples include <b>Save Search</b> and <b>Perform Do Task</b>.</p> <p>A module can contribute a tool command to a sublocation of another module.</p>                                                                                                                               |
| 4 | Navigation commands           | <p>These commands control the content of the main work area.</p> <p>Examples include search filter and BOM configuration.</p> <p>A module can contribute a navigation command to a sublocation of another module.</p> <p>The data content is controlled by the sublocation (work area) for the other module. It can be used only with the published contract of that sublocation.</p> |

## One-step commands

The **IOneStepCommandHandler** is a marker interface to help classify this type of Active Workspace command.

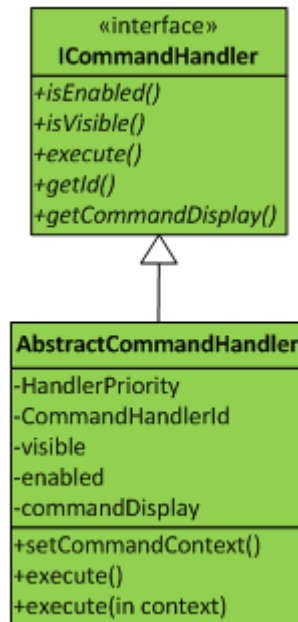


A one-step command is a stand-alone operation that requires no user input or intervention.

## AbstractCommandHandler base class

**AbstractCommandHandler** is a base class for **ICommandHandler** shared implementation.

- This class provides consistent shared logic for all the command handler implementations.
- Most command handler implementations should inherit from this type.



Property value management and access to the declared values are provided by this class.

## GIN binding command handlers

Most command handlers are declared and contributed to the application through GIN (GWT injection) module declarations.

- Every command handler referenced by the configured application must be declared in GIN.
- After a command is declared, it can be bound to one or more containers within GIN.

To help with some of the verbose nature of GIN binding statements, helper functions are provided in **com.siemens.splm.clientfx.ui.commands.published.CommandsExtensionPointHelper**.

### Utility method to register a command handler

```

void com.siemens.splm.clientfx.ui.commands.published
    .CommandsExtensionPointHelper
    .registerCommandHandler(
        GINBinder binder,
        String commandId,
        String commandHandlerId,
        Class<? extends ICommandHandler> commandHandlerClass,
        Class<? extends ICommandDisplay> commandDisplayClass)
  
```

Parameters:

<b>binder</b>	GIN binder
<b>commandId</b>	Identifier of the command for which this handler is defined Identifier of the command handler
<b>commandHandlerID</b>	This can be the same as the command ID if there is only one handler for the command.
<b>commandHandlerClass</b>	Handler class

Display class

### commandDisplayClass

This is the class used by default, unless overridden in a widget.

### Utility method to contribute a command to a command area

```
void com.siemens.splm.clientfx.ui.commands.published
    .CommandsExtensionPointHelper
    .contributeCommandToArea(
        GINBinder binder,
        String subLocationNameToken,
        String commandAreaNameToken,
        Class<? extends Provider<CommandId>> commandIdProviderType,
        int order)
```

Parameters:

<b>binder</b>	GIN binder
<b>subLocatoinNameToken</b>	Sublocation name token
<b>commandAreaNameToken</b>	Command area type, such as <b>OneStep</b>
<b>commandIdProviderType</b>	<b>CommandID</b> provider class
<b>order</b>	Order of command

### Example: command to launch a web page

```
# HelloWorldCommandHandler.java

package com.samples.internal.commands;

import com.google.gwt.user.client.Window;
import com.google.inject.Inject;
import com.google.inject.name.Named;
import com.samples.published.NameTokens;
import com.siemens.splm.clientfx.ui.commands.published.AbstractCommandHandler;
import com.siemens.splm.clientfx.ui.commands.published.ICommandDisplay;

/**
 * HelloWorld command handler
 */
public class HelloWorldCommandHandler
    extends AbstractCommandHandler
{
    /**
     * Constructor
     *
     * @param commandDisplay command display to use for this handler
     * @param presenterProvider provider for presenter
     */
    @Inject
    public HelloWorldCommandHandler(
        @Named( NameTokens.CMD_HelloWorld ) ICommandDisplay commandDisplay )
    {
        super( NameTokens.CMD_HelloWorld, commandDisplay );
    }

    @Override
    public void commandContextChanged()
    {
        /**
         * In this method, you'll want to determine & set the enablement & visibility
         * of your command. This is typically done by querying the selection &
         * making decision based upon it.
         */
    }
}
```



```

        * E.g. Object selectedObject =
        *   getCommandContext().getSelection().getSelectedObject();
        * selectedObject returned here is generally an IModelObject.
        */
        setIsEnabled( true );
        setIsVisible( true );
    }

    @Override
    protected void doExecute()
    {
        /**
         * In this method, you'll typically get the selection & execute the command
         * on it.
         */
        Window.open( "http://www.plm.automation.siemens.com", "_blank", "" );
    }
}

```

# HelloWorldCommandDisplay.java

```

package com.samples.internal.commands;

import com.google.gwt.resources.client.ImageResource;
import com.siemens.splm.clientfx.ui.commands.published.AbstractCommandDisplay;
import com.samples.internal.Resources;
import com.samples.resources.i18n.HelloWorldMessages;

/**
 * Hello World command display
 */
public class HelloWorldCommandDisplay
    extends AbstractCommandDisplay
{
    /**
     * Constructor
     */
    public HelloWorldCommandDisplay()
    {
        super( HelloWorldMessages.INSTANCE.HelloWorldCommandTitle() );
    }

    @Override
    public ImageResource getIconResource()
    {
        return Resources.INSTANCE.getHelloWorldCommandImage();
    }
}

```

#HelloWorldModule.java

```

package com.samples.internal.config;

import com.google.gwt.inject.client.AbstractGinModule;
import com.google.inject.Provider;
import com.siemens.splm.clientfx.ui.commands.published.CommandId;
import com.siemens.splm.clientfx.ui.commands.published.CommandsExtensionPointHelper;

import com.samples.internal.commands.HelloWorldCommandDisplay;
import com.samples.internal.commands.HelloWorldCommandHandler;
import com.samples.published.NameTokens;

/**
 * Gin Module configuration for HelloWorld command

```

```

    */
    public class HelloWorldModule
        extends AbstractGinModule
    {
        @Override
        protected void configure()
        {
            CommandsExtensionPointHelper.registerCommandHandler( binder(),
                NameTokens.CMD_HelloWorld, NameTokens.CMD_HelloWorld,
                HelloWorldCommandHandler.class, HelloWorldCommandDisplay.class );

            // Add the command to the global one step commands
            CommandsExtensionPointHelper.contributeCommandToArea( binder(),
                com.siemens.splm.clientfx.ui.published.NameTokens.GLOBAL_COMMANDS,
                com.siemens.splm.clientfx.ui.published.NameTokens.ONE_STEP_COMMANDS,
                HelloWorldCommandIdProvider.class, 10000 );
        }

        /**
         * Command ID Provider for HelloWorld command
         */
        public static class HelloWorldCommandIdProvider
            implements Provider<CommandId>
        {
            @Override
            public CommandId get()
            {
                return new CommandId( NameTokens.CMD_HelloWorld );
            }
        }
    }
}

```

## Add a new one-step command to your module

### Note

If you have not created a module yet, follow the steps to *Use generateModule to create a new module*.

1. Open a command prompt and change to your module directory.

```
cd STAGE\src\myModule
```

2. Run the **generateModule** script, and use the **command** argument.

```

Enter type to generate: command
Enter one step command name (Required): myOneStepCommandName
[12:34:56] [Debug] Creating command myOneStepCommandName
[12:34:56] Successfully created command myOneStepCommandName

```

This updates the **gwt.xml** file, adds a placeholder icon **png** file, and Java source files for your new one-step command within your module directory.

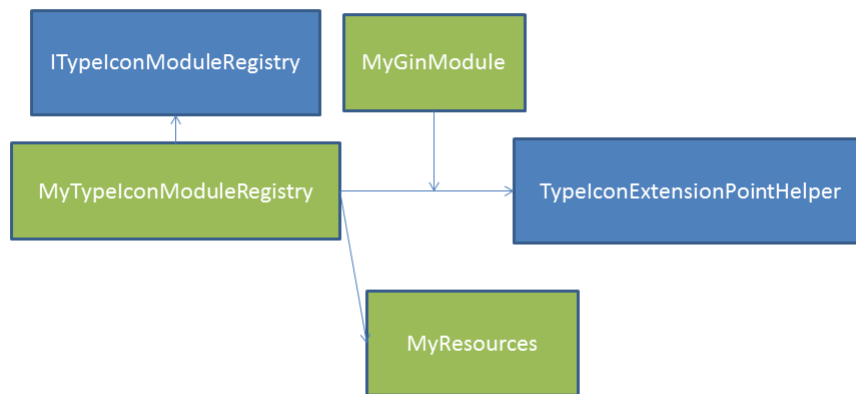
## Type icons

### Type icon overview



The default icon for revision objects is .

This module component allows you to change the icons based on the type of object.



You must define your own **MyTypeIconModuleRegistry**, which implements **ITypeIconModuleRegistry**. Your **MyTypeIconModuleRegistry** uses your own **MyResource** for image files. Your **MyGinModel**, which extends **AbstractGinModule**, registers your **MyTypeIconModuleRegistry** with **TypelconExtensionPointHelper**.

### Add a new type icon to your module

#### Note

If you have not created a module yet, follow the steps to *Use generateModule to create a new module*.

Type icons are all 46 x 46 pixels.

1. Open a command prompt and change to your module directory.

```
cd STAGE\src\myModule
```

2. Run the **generateModule** script, and use the **type** argument.

```
Enter type to generate: type
Enter type name (Required): myTypeIconName
[12:34:56] [Debug] Creating type myTypeIconName
[12:34:56] Successfully created type myTypeIconName
```

This updates the **gwt.xml** file, and creates placeholder **ai** and **png** files, as well as a Java source file for registering your icon to an object type.

## Locations and sublocations

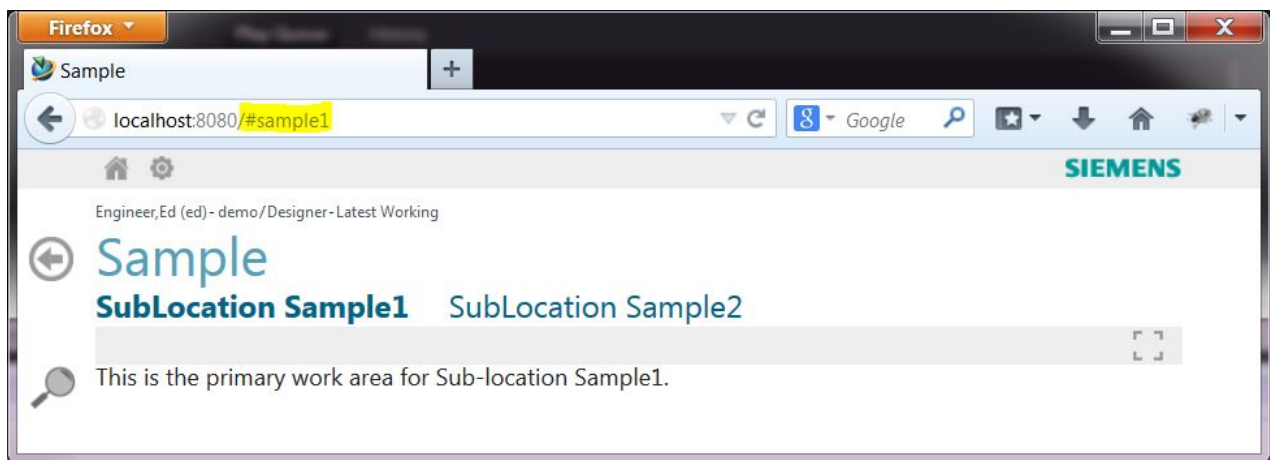
### Location and sublocation overview

After logging on, append one of your sublocation names to your URL.

#### Example

```
http://localhost:8080/#Sample1
```

This takes you to your new location and displays the first sublocation.



### Add a new location or sublocation to your module

#### Note

If you have not created a module yet, follow the steps to *Use generateModule to create a new module*.

#### location

1. Open a command prompt and change to your module directory.

```
cd STAGE\src\myModule
```

2. Run the **generateModule** script, and use the **location** argument.

```
Enter type to generate: location
Enter location name (Required): myLocationName
[12:34:56] [Debug] Creating location myLocationName
[12:34:56] Successfully created location myLocationName
```

This updates the **gwt.xml** file, and creates a Java source file for your new location presenter.

## sublocation

1. Open a command prompt and change to your module directory.

```
cd STAGE\src\myModule
```

2. Run the **generateModule** script, and use the **subLocation** argument.

```
Enter type to generate: subLocation
Enter sub-location name (Required): mySubLocationName
Enter the location to apply this sub-location to. (Required): myLocationName
[12:34:56] [Debug] Creating subLocation mySubLocationName
[12:34:56] Successfully created subLocation mySubLocationName
```

### Note

The argument is *subLocation*, not *sublocation*; it is case-sensitive.

The name of the location to which you apply this sublocation is also case-sensitive.

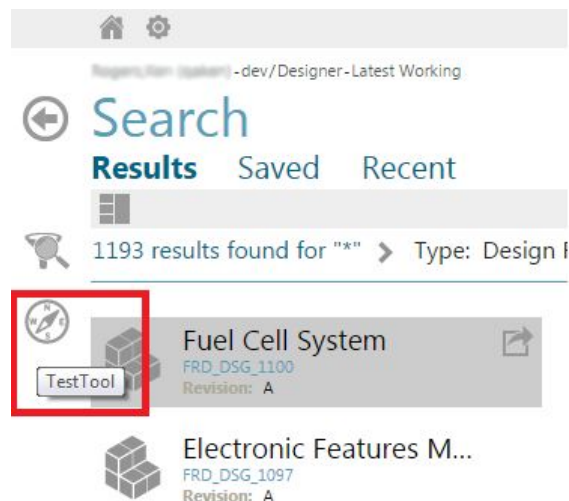
This updates the **gwt.xml** file, and creates a Java source files for your new sublocation and primary work area presenters.

## Navigation panel

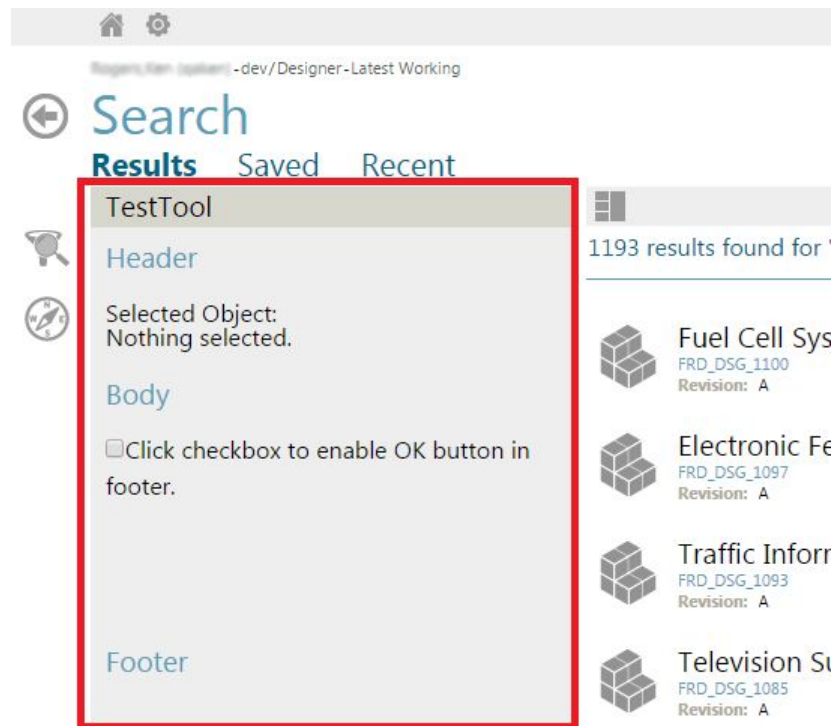
### Navigation panel overview

In the search results, a new command is added in the navigation commands area.

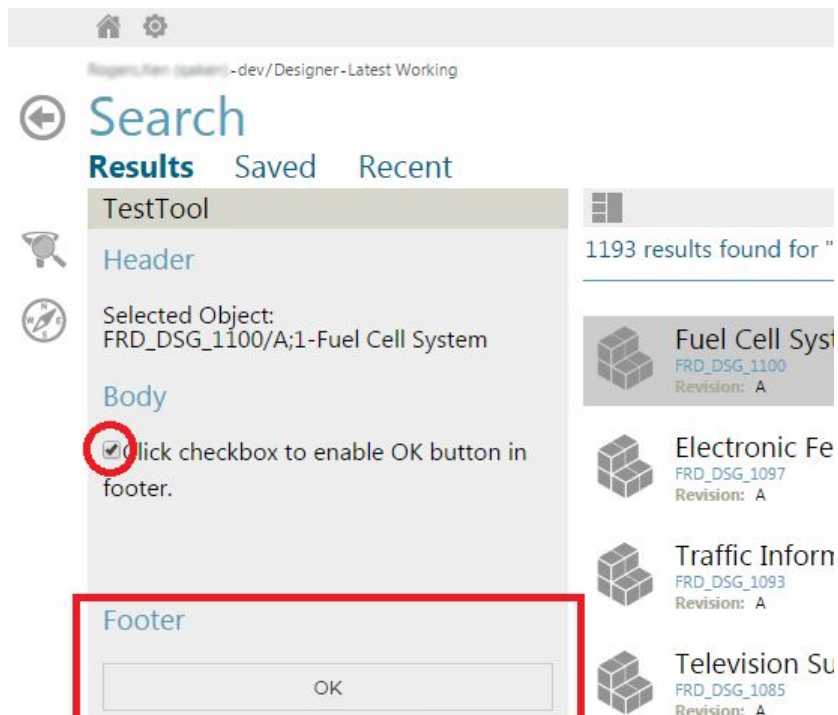
When hovering on the command, the tool name is presented as a tooltip.



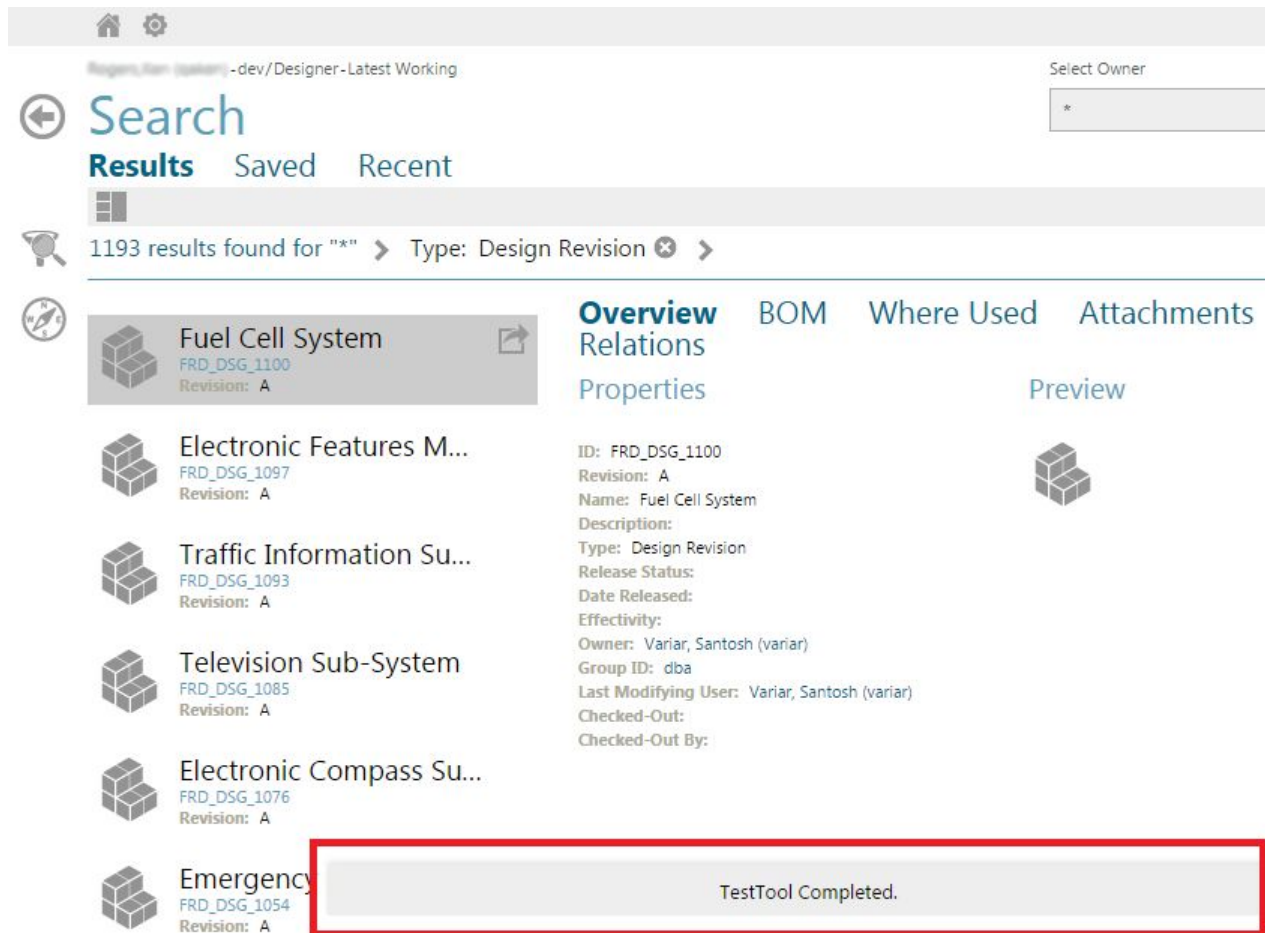
When you click the **TestTool** button a panel displays the title, header, body, and footer sections of the panel. The currently selected object's display name appears as well; if no selection is made, Nothing Selected appears.



When you select the **Body** checkbox, the footer is updated to display the **OK** button.



When you click the **OK** button, the panel hides and a message indicates that the operation is complete. For this example, nothing is actually performed on the selected object.



## Add a new navigation panel to your module

### Note

If you have not created a module yet, follow the steps to *Use generateModule to create a new module*.

1. Open a command prompt and change to your module directory.

```
cd STAGE\src\myModule
```

2. Run the **generateModule** script, and use the **navPanel** argument.

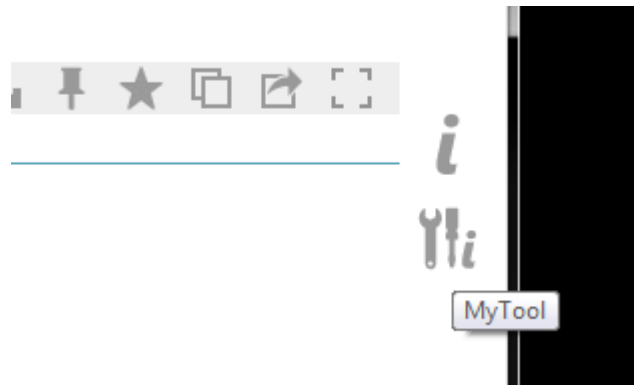
```
Enter type to generate: navPanel
Enter navigation panel name (Required): myNavPanelName
[12:34:56] [Debug] Creating navPanel myNavPanelName
[12:34:56] Successfully created navPanel myNavPanelName
```

This updates the **gwt.xml** file, creates placeholder **ai** and **png** files, as well as a Java source file for view, view model, UI handler, command display and handler, and presenter.

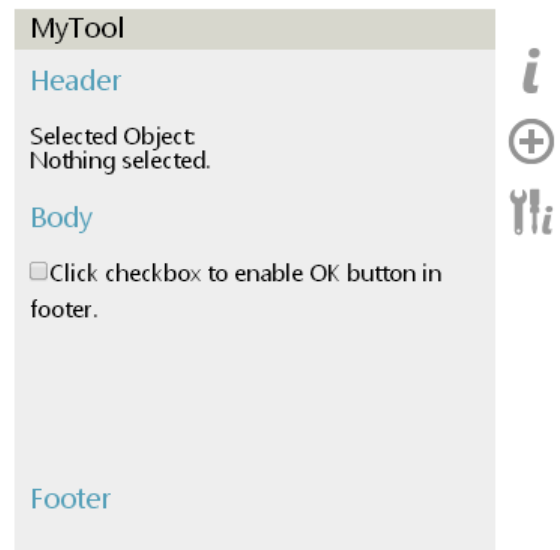
## Tools and information panel

### Tools and information panel overview

In the user's home folder, a new command is added in the tools and information commands area. When hovering on the command, the tool name is displayed as a tooltip.

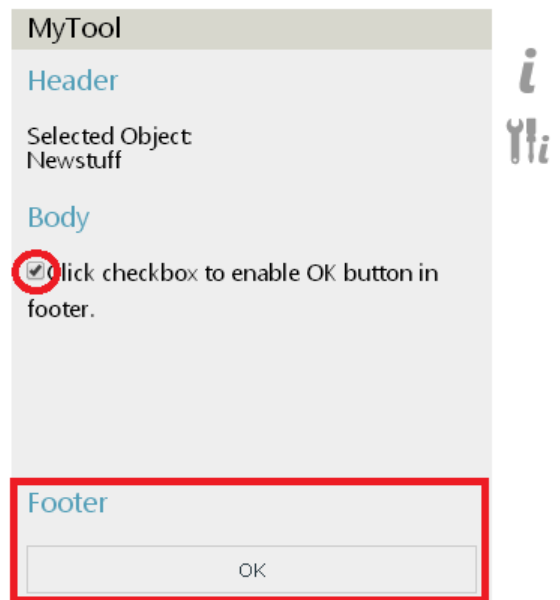


When you click the **MyTool** button, a panel displays the title, header, body, and footer sections of the panel. The currently selected object's display name appears as well; if no selection is made, then `Nothing Selected` appears.

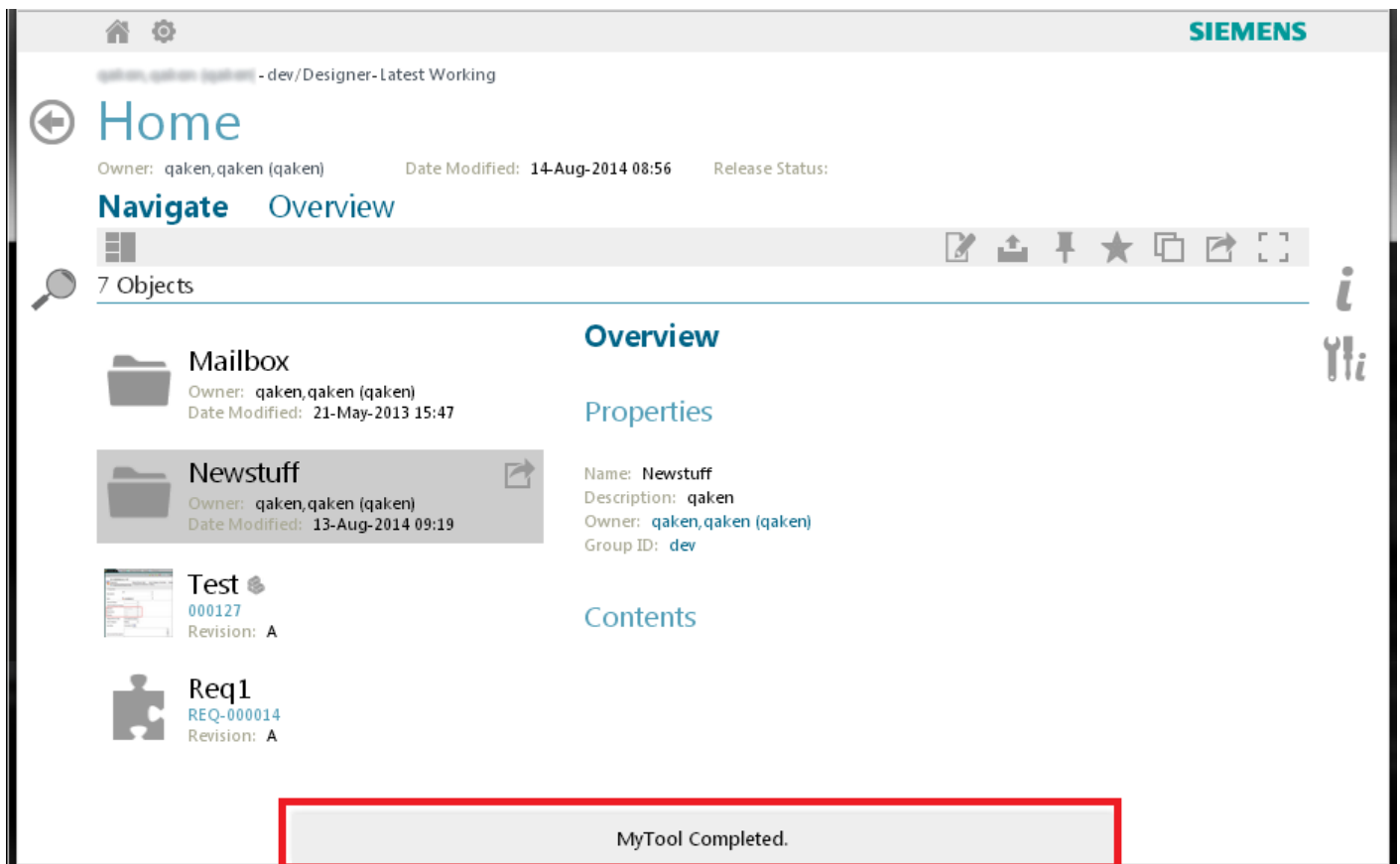




When you select the **Body** checkbox, the footer is updated to display the **OK** button.



When you click the **OK** button, the panel hides and a message indicates that the operation is complete. For this example, nothing is actually performed on the selected object.



## Add a new tools and information panel to your module

### Note

If you have not created a module yet, follow the steps to *Use generateModule to create a new module*.

1. Open a command prompt and change to your module directory.

```
cd STAGE\src\myModule
```

2. Run the **generateModule** script, and use the **theme** argument.

```
Enter type to generate: toolsInfoPanel
Enter tools and info panel name (Required): MyTnIPanelName
[12:34:56] [Debug] Creating toolsInfoPanel MyTnIPanelName
[12:34:56] Successfully created toolsInfoPanel MyTnIPanelName
```

This updates the **gwt.xml** file, creates placeholder **ai** and **png** files, as well as a Java source file for view, view model, UI handler, command display and handler, and presenter.

## Chapter 13: Using property widgets

### Property widget overview

Property widgets are a common set of widgets available for rendering properties of different types in Active Workspace. The following examples assume that you already have a place to put them: a sublocation, a tool and information panel, a navigation panel, etc.

### Common widget features

The set of widgets available for designing the application's input and output interaction have the following components in common.

If the field is...	the user will see...
empty	
required	
modified	
invalid	

- The *property label* is rendered flat and represents the name of the property.

Interface Enum **IPropertyDataBoundWidget.PropertyLabelDisplay** can be used to have different styles of labels

**PropertyLabelDisplay.NO\_PROPERTY\_LABEL** does not display the property label.

**PropertyLabelDisplay.PROPERTY\_LABEL\_AT\_TOP** displays the property label at the top of the property value.

**PropertyLabelDisplay.PROPERTY\_LABEL\_AT\_SIDE** displays the property label at the left side of the property value.

- The *property value* is where the user places input.

This can change depending on the type of widget. The example shows a widget with a text area. As the user enters value to this text area, it turns yellow indicating that the value is modified since displayed. This is supported only if the widgets are data bound with the property in model.

This also supports having a real value and a display value. In most cases, the UI renders the display values.

## Standard Active Workspace widgets

### StringTextBoxWidget

A text box widget capable of showing a string value. Use this to show a property of type string with limited characters.

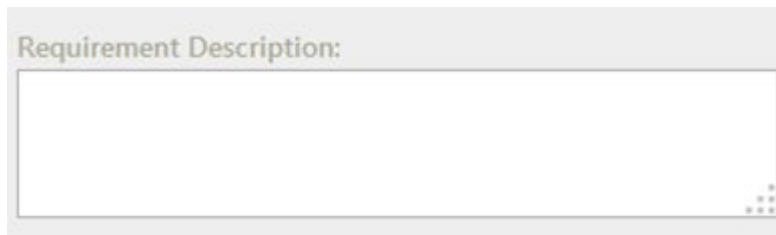


In read-only mode, the widget is displayed as shown.

**Model** : This is the Model Value in read-only mode

### StringTextAreaWidget

A text area widget capable of showing a string value in multiple lines. It is normally used to show property values with larger character limits.



In read-only mode, the widget is displayed as shown.

**Description:** Automobile safety is the study and practice of design, construction, equipment and regulation to minimize the occurrence and consequences of automobile accidents. Road traffic safety more broadly includes roadway design.

### LabelWidget

A label shows a value as label.

**Owner:** qaken,qaken (qaken)

This widget is always read-only.

### BooleanCheckBoxWidget

A Boolean widget capable of showing a value rendered as a check box.



In read only mode, the widget is displayed as shown.

In Teamcenter 9: **Power Steering:** Y

In Teamcenter 10: **Power Steering:** True

## BooleanRadioButtonWidget

A Boolean widget capable of showing a value rendered with radio buttons.

**RadioButton:** ☐ True ☒ False

In read-only mode, the widget is displayed as shown.

In Teamcenter 9: **Alloy Wheels:** N

In Teamcenter 10: **Alloy Wheels:** False

## BooleanToggleButtonWidget

A text box widget capable of showing a string value. Use this to show a property of type string with limited characters.

The toggle button has two states **On** and **Off**. The labels can be customized through the constructor.

On/True

**ToggleButton:** 

Off/False

**ToggleButton:** 

## IntegerTextBoxWidget

A text box widget capable of showing an integer value. It looks exactly like **StringTextBoxWidget**. However, it does not accept non-integer values as input. When the user tries to type in a noninteger value, it deletes the value after validation.

**Seat Capacity:**

## DoubleTextBoxWidget

A text box widget capable of showing a double value. Similar to **IntegerTextBoxWidget**, it also does not accept any nondouble values as input.

**Mileage:**


## ObjectLinkPropertyWidget

A widget capable of showing a Teamcenter object reference, either *External*, *Typed*, or *Untyped*.

Unpopulated

UntypedReference: 

Populated

UntypedReference:  
HDD-0527/A;1-Hard Drive Assembly 

In read-only mode, the widget is displayed as shown. The value is a link that the user can use to navigate to the object.

UntypedReference: [HDD-0527/A;1-Hard Drive Assembly](#)

## DateWidget

A widget capable of showing a Teamcenter date property.

### UI features of the date widget


- Separate date and time selection.


Date:

mm/dd/yy	HH:mm:ss 
----------	------------------------------------------------------------------------------------------------


- The date selector.

Date:

03/04/2015	12:34:56 
------------	----------------------------------------------------------------------------------------------




March 2015




Mo	Tu	We	Th	Fr	Sa	Su
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

- The time selector.



12:30
13:00
13:30
14:00

Date:

03/04/2015	12:34:56 
------------	----------------------------------------------------------------------------------------------

- You can manually enter date and time; the calendar display is kept synchronized.

## Availability

The date widget is available:

- In an XRT panel.

The choice to have a time selector is configurable through code only.

- In an HTML panel.


They can be configured through a style sheet.

```
<page title="HTML Panel">
  <htmlPanel>
    <![CDATA[
      <div class="aw-layout-panelSectionContent">
        <div class="aw-widgets-propertyContainer">
          <aw-property prop="selected.properties['object_name']"></aw-property>
          <aw-property prop="selected.properties['start_date']"></aw-property>
        </div>
      </div>
    ]]>
  </htmlPanel>
</page>
```

- Using UIBinder.

## Property widget examples

### Property widget examples overview

In this example, you open a tool and information panel showing a few COTS widgets. The  button in the tools and information command bar is contributed in our sample. When you click the command, a **Car Comparison Criteria** panel displays the example widgets.

## Prerequisites

A tool and information command is already defined with the **CarComparisonCriteriaCommandHandler** command handler, which appears in a panel in **CarCriteriaCommandPresenter**.

In this example, a reusable features panel is created which can be plugged into a sublocation, or a tool and information panel, and so on. In this example, **FeaturePresenter** is shown as a subpresenter in **CarCriteriaCommandPresenter** presenter. It is injected to **CarCriteriaCommandPresenter**.

The class definition for feature presenter with its view interface in **com.siemens.splm.samples.cotsWidgets.internal.presenters** package follows.

```
/**
 * Car Feature Presenter
 */
public class FeaturePresenter
    extends AbstractPresenterWidget<FeaturePresenter.MyView>
    implements UiHandlers
{
    //constructor

    /**
     * View interface
     */
}
```



```

public interface MyView
    extends View, HasUiHandlers<UiHandlers>
{
}

```

## Add property widgets

### Add imports

To use the provided widgets in the **FeatureView.ui.xml** file, add the following definitions of XML namespace prefixes in the file:

```

xmlns:uiw="urn:import:com.siemens.splm.clientfx.ui.widgets.published"
xmlns:tcuiw="urn:import:com.siemens.splm.clientfx.tcui.widgets.published"

```

To use the localized constants, add:

```

<ui:with
    type="com.siemens.splm.samples.cotsWidgets.resources.il18n.CotsWidgetsConstants"
    field="resil18n" />

```

To use the provided property label display style, import **IPropertyDataBoundWidget**:

```

<ui:import
    field="com.siemens.splm.clientfx.ui.databind.published.view.IPropertyDataBoundWidget
        .PropertyLabelDisplay.*" />

```

### Add widget declaration

Each widget used to build the UI uses the following format:

```

<uiw:{WidgetClass} ui:field="..." propertyLabelDisplay="{PROPERTY_LABEL_AT_TOP}"
    isEditable="..." propertyLabel="..." value="..." bindValue="..."
    isRequired="..." maxLength="..." numberOfCharacters="..." />

```

Properties supported in UI binder declaration:

**{WidgetClass}** – This should be replaced by the widget to be used in UI building.

**ui:field** – The value for this should be a unique name in a panel. If required, this name is used to inject the widget with **@UIField** annotation in the **FeatureView.java** file.

**propertyLabelDisplay** – This property is used to define the property label display style for the widget. The value for this should be **NO\_PROPERTY\_LABEL**, **PROPERTY\_LABEL\_AT\_TOP**, or **PROPERTY\_LABEL\_AT\_SIDE**.

**isEditable** – This property is used to show the widget in editable or read only mode. It takes the values **True** and **False**.

**value** – This property is used set an initial value to the widget. It takes a string value.

**isRequired** – This property is used to configure the widget as required. It takes **True** and **False** values. If the value is set to **True**, a word **Required** appears inside or next to the value field.

**maxLength** – This property is used to set the maximum number of characters that is allowed in the input field. For example, if the **maxLength** is set to 15, the user cannot type a string more than 15 characters long.

**numberOfCharacters** – This property can be used to set the maximum number of characters that is displayed in the value. For example, if **numberOfCharacters** is set to 25, the user cannot see more than the first 25 characters of the string.

**bindValue** – This property is used to set the binding key or constant between a widget and a property. This must be set if the widget needs to be data bound.

### StringTextBoxWidget example

To use this widget with the **model** property of type string, add the following in the **FeatureView.ui.xml** file.

```
<uiw:StringTextBoxWidget ui:field="model"
    propertyLabelDisplay="{PROPERTY_LABEL_AT_TOP}" isEditable="true"
    propertyLabel="{resi18n.Model}" value="" bindValue="model"
    isRequired="true" />
```

**StringTextBoxWidget** is instantiated for the **model** property.

**propertyLabelDisplay="{PROPERTY\_LABEL\_AT\_TOP}"** signifies that the property label appears at the top of the property value.

**isEditable="true"** displays the widget as editable.

**propertyLabel="{resi18n.Model}"** The localized label of the property is provided by the **resi18n** field defined in the XML file.

**value=""** means that no initial value is displayed in the input field.

**isRequired="true"** marks the widget as required.

**bindValue="model"** The binding key or constant is set between the widget and the **model** property.

#### Note

**StringTextBoxWidget** and **LabelWidget** are very similar to **StringTextBoxWidget** and can be declared similarly.

### BooleanRadioBoxWidget example

To use this widget with the **alloyWheels** property of type string, add the following in the **FeatureView.ui.xml** file.

```
<uiw:BooleanRadioBoxWidget ui:field="alloyWheels" isEditable="true"
    propertyLabel="{resi18n.AlloyWheels}" value="False"
    bindValue="alloyWheels" />
```

**BooleanRadioBoxWidget** is instantiated for the **alloyWheels** property.

**propertyLabelDisplay** is not currently supported for Boolean widgets.

**propertyLabel="{resi18n.AlloyWheels}"** The localized label of the property is provided by the **resi18n** field defined in the XML file.

**value** can not be blank for Boolean widgets.

**numberOfCharacters** and **maxLength** are not supported for Boolean widgets.

**bindValue**="alloyWheels" The binding key or constant is set between the widget and the **alloyWheels** property.

**Note**

**BooleanCheckBoxWidget** and **BooleanToggleButtonWidget** are very similar to **BooleanRadioBoxWidget** and can be declared similarly.

### IntegerTextBoxWidget example

To use this widget with the **seatCap** property of type string, add the following in the **FeatureView.ui.xml** file.

```
<uiw:IntegerTextBoxWidget ui:field="seatCap" isEditable="true"
    propertyLabel="{resi18n.SeatCapacity}" value="5" isRequired="false"
    bindValue="seatCap" />
```

**IntegerTextBoxWidget** is instantiated for the **seatCap** property.

**propertyLabelDisplay** is not currently supported for integer widgets.

**propertyLabel**="{resi18n.SeatCapacity}" The localized label of the property is provided by the **resi18n** field defined in the XML file.

**value** can not be blank for integer widgets; it must be set to an integer.

**numberOfCharacters** and **maxLength** are not supported for integer widgets.

**bindValue**="seatCap" The binding key or constant is set between the widget and the **seatCap** property.

### DoubleTextBoxWidget example

To use this widget with the **mileage** property of type string, add the following in the **FeatureView.ui.xml** file.

```
<uiw:DoubleTextBoxWidget ui:field="mileage" isEditable="true"
    propertyLabel="{resi18n.Mileage}" value="14.6" isRequired="false"
    bindValue="mileage" />
```

**DoubleTextBoxWidget** is instantiated for the **mileage** property.

**propertyLabel**="{resi18n.Mileage}" The localized label of the property should be provided by the **resi18** field defined in the XML file.

**value** can not be blank for double widgets. To set a value like 14, specify **value**="14d".

**bindValue**="mileage" The binding key or constant is set between the widget and the **mileage** property.

### ObjectlinkPropertyWidget example

To use this widget with the **engine** property of type string, add the following in the **FeatureView.ui.xml** file.

```
<tcuiw:ObjectlinkPropertyWidget ui:field="engine" isEditable="true"
```

```
propertyLabel="{resi18n.Engine}" bindValue="engine" />
```

**ObjectlinkPropertyWidget** is instantiated for the **engine** property.

**propertyLabel="{resi18n.Engine}"** The localized label of the property is provided by the **resi18** field defined in the XML file.

**propertyLabelDisplay** is not currently supported for object link widgets.

**value**, **numberOfCharacters** and **maxLength** are not supported for object link widgets.

**bindValue="engine"** The binding key or constant is set between the widget and the **engine** property.

## Define the view

To define the view for **FeaturePresenter**, create a **FeatureView** class that uses **UiBinder** to build the UI containing the widgets. To work with the widgets and provide some action on them, inject them with **@UiField** annotation.

The following code shows where **FeatureView** implements **FeaturePresenter.MyView**:

```
/**
 * Car Feature view
 */
public class FeatureView
    extends ViewWithUiHandlers<UiHandlers>
    implements MyView
{
    /**
     * Widget
     */
    private final Widget m_widget;
    /**
     * model property widget
     */
    @UiField
    StringTextBoxWidget model;
    /**
     * alloy wheels property widget
     */
    @UiField
    BooleanRadioBoxWidget alloyWheels;
    /**
     * Ui binder
     */
    public interface Binder
        extends UiBinder<Widget, FeatureView>
    {
        //
    }
    /**
     * Constructor
     *
     * @param binder Ui binder
     */
    @Inject
    public FeatureView( final Binder binder )
    {
        m_widget = binder.createAndBindUi( this );
        initWidget();
    }
}
```

## Provide data binding to property widgets

- To make the widgets in the feature view data bound, **FeaturePresenter.MyView** should extend **IDataBoundWidgetContainer**.

```
/**
 * View interface
 */
public interface MyView
    extends View, HasUiHandlers<UiHandlers>, IDataBoundWidgetContainer
{
}
```

- A member variable is defined in **FeatureView** to list the set of widgets that are data bound:

```
/**
 * List of data bound widgets in the section.
 */
private final List<IDataBoundWidget<?>> m_dataBoundWidgets = new ArrayList<>();

Implement the interface methods in FeatureView:

@Override
public List<IDataBoundWidget<?>> getDataBoundWidgets()
{
    return m_dataBoundWidgets;
}

@Override
public List<ICollectionBoundWidget> getCollectionBoundWidgets()
{
    return null;
}
```

### Note

This list is populated in the constructor of **FeatureView** with the widgets injected with **@UiField** annotation. See the sample code for implementation.

- FeatureViewModel** is the view model for **FeatureView**:

```
/**
 * Feature View Model
 */
public class FeatureViewModel
    extends AbstractTcViewModel
{
    /**
     * model property
     */
    public static final String MODEL = "model"; //$NON-NLS-1$

    /**
     * alloyWheels property
     */
    public static final String ALLOY_WHEELS = "alloyWheels"; //$NON-NLS-1$
}
```

- **bindValue** in **UIBinder**

The widget declaration in **UIBinder** has a property called **bindValue**. This value for this property provides the key to bind the property with a widget. This is mostly the property name; as in a panel, the property name is unique.

```
<uiw:StringTextBoxWidget ui:field="model"
    propertyLabelDisplay="{PROPERTY_LABEL_AT_TOP}" isEditable="true"
    propertyLabel="{res18n.Model}" value="" bindValue="model"
    isrequired="true" />
```

- Binding **view** and **view model**

To bind the view and the view model, the **FeaturePresenter** calls:

```
IDataBindInjector.INSTANCE.getDataBinder().bind( getView(), m_viewModel );
```

- Setting the default binding scope to view model object

In **FeatureViewModel**, a view model object (**viewModelObject**) is instantiated containing the properties corresponding to the view. The binding scope is set to this model object.

```
setBindObject( AbstractViewModel.DEFAULT_DATA_BIND_SCOPE, viewModelObject );
```

- To check if data binding is working properly, enter a value to the **model** property to see the field become yellow as the value is modified.



## **Siemens Industry Software**

### **Headquarters**

Granite Park One  
5800 Granite Parkway  
Suite 600  
Plano, TX 75024  
USA  
+1 972 987 3000

### **Americas**

Granite Park One  
5800 Granite Parkway  
Suite 600  
Plano, TX 75024  
USA  
+1 314 264 8499

### **Europe**

Stephenson House  
Sir William Siemens Square  
Frimley, Camberley  
Surrey, GU16 8QD  
+44 (0) 1276 413200

### **Asia-Pacific**

Suites 4301-4302, 43/F  
AIA Kowloon Tower, Landmark East  
100 How Ming Street  
Kwun Tong, Kowloon  
Hong Kong  
+852 2230 3308

## **About Siemens PLM Software**

Siemens PLM Software, a business unit of the Siemens Industry Automation Division, is a leading global provider of product lifecycle management (PLM) software and services with 7 million licensed seats and 71,000 customers worldwide. Headquartered in Plano, Texas, Siemens PLM Software works collaboratively with companies to deliver open solutions that help them turn more ideas into successful products. For more information on Siemens PLM Software products and services, visit [www.siemens.com/plm](http://www.siemens.com/plm).

© 2015 Siemens Product Lifecycle Management Software Inc. Siemens and the Siemens logo are registered trademarks of Siemens AG. D-Cubed, Femap, Geolus, GO PLM, I-deas, Insight, JT, NX, Parasolid, Solid Edge, Teamcenter, Tecnomatix and Velocity Series are trademarks or registered trademarks of Siemens Product Lifecycle Management Software Inc. or its subsidiaries in the United States and in other countries. All other trademarks, registered trademarks or service marks belong to their respective holders.