Guidelines for Contributing Eclipse User Assistance to the DTP Project

v 1.2

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

Table of Contents

TABLE OF CONTENTS **2**

VERSIONS **4**

OVERVIEW **5**

CURRENT DTP 1.0 DOCUMENTATION STRUCTURE **5**

PROJECT ARCHITECTURE DECISIONS **5**

- CAPABILITIES 6
- COMPONENT HIERARCHY 6
- ARCHITECTURAL SPECIFICATIONS 7
  - *Download/Installation Packages* *7*
  - *Features* *7*
  - *Capabilities (Categories)/Activity Enablement* *7*

USER ASSISTANCE PLUG-IN STRUCTURE **8**

- PROJECT-LEVEL PLUG-INS 8
- COMPONENT-LEVEL PLUG-INS 9
- USER ASSISTANCE FEATURE IDS AND PLUG-IN IDS 10

WELCOME PAGE (INTRO) **10**

- PROJECT-LEVEL INTRO PLUG-IN 10

HELP CONTENT **11**

- PROJECT-LEVEL DOC PLUG-IN 11
- TOC CONTRIBUTIONS 11
- ANCHORS AND LINKS IN TOC.XML 12
- DITA MAPS 12

CONTEXT-SENSITIVE HELP **13**

- CONTEXT-SENSITIVE HELP IMPLEMENTATION 13
- RULES FOR HELPKEY INTERFACE CLASSES 17
- TEAM COLLABORATION AND RESPONSIBILITIES 18
  - *Development Team Responsibilities* *18*
  - *Documentation Team Responsibilities* *19*
- JAVA SOURCE FILE CONVENTIONS 19
- PROPERTIES FILE CONVENTIONS 20
  - *File Names* *20*
  - *File Content Format* *20*
  - *Context IDs File Example* *21*
  - *Search Expressions File Example* *21*
- DOCUMENTATION TEAM WORKFLOW 21
  - *Defining Help Context IDs* *22*
  - *Defining Help Search Expressions* *23*

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

Context-Sensitive Help Plug-in Structure ... 25
CHEAT SHEETS ... 26
Project-level Cheat Sheet Plug-in ... 26
Cheat Sheet Contributions ... 26
USER ASSISTANCE DELIVERY FORMATS ... 28

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

Versions

Version	Primary Author(s)	Description of Version	Date Completed
1.0	Emily Kapner, Maria Brownstein, Dave Resch	Initial Draft	2/28/07
1.1	Dave Resch	Updated content based on feedback at EclipseCon	3/20/07
1.2	Dave Resch	Updated context-sensitive help info, other revisions	4/16/07

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

Overview

This document outlines the recommended approach for contributing Eclipse user assistance content to the Data Tools Platform (DTP) project. User assistance content includes Welcome page content, help (documentation) content, cheat sheets, and context-sensitive help files, usually delivered in the form of plug-ins.

The purpose of these guidelines is to create an approach that is both consistent and scalable, and that allows the DTP project team to include or exclude help content contributions, based on capabilities (categories/activities, usually defined by DTP UI components).

Current DTP 1.0 Documentation Structure

The DTP 1.0 documentation was not architected for scalability. The current DTP 1.0 user documentation includes:

- A single plug-in containing help content for Connectivity and SQL Development topics,
- An intro plug-in that extends the SDK universal intro contributing DTP Welcome content,
- A plug-in for API documentation, and
- A context-sensitive help plug-in for DTP 1.0 BIRT that bridges DTP to ODA.

For future releases, it would be best to architect the user assistance contributions so that they align with DTP capabilities. This will allow the DTP project team to create different installation packages, each of which includes only the applicable set of corresponding help content-contributing plug-ins. This document identifies how to implement this architecture.

Project Architecture Decisions

Several project-level architecture decisions will affect the design and implementation of Eclipse user assistance plug-ins, particularly the division of content across multiple plug-ins, and the coordination of plug-in IDs.

Those project-level architecture decisions must be made early in the release planning stage, and they must be communicated clearly to both Development teams and Documentation teams, ideally in the form of architectural specifications.

Note: Adequate architectural specifications are critical for the design and implementation of user assistance contributions, and to ensure that Documentation teams can meet release schedules.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

Capabilities

Capabilities (categories/activities) can be used to enable or disable the plug-ins that contribute functionality and/or UI components, effectively "filtering" the user interface to reduce complexity and enhance usability.

Capabilities can be switched on and off by:

- Associating a "trigger" with a user action (e.g., changing the focus in a view or perspective, selecting a menu item, etc.), so that capabilities can be switched programmatically through user interaction
- Defining a preference page and implementing a class to populate that preference page, allowing capabilities to be accessed directly by the user through the Preferences dialog

Note: The Preference dialog need not expose all capabilities to the user.

The project should design capabilities, and define their associations with features and plug-ins to provide appropriate UI filtering.

Capabilities should be contributed by UI plug-ins, so declaring and implementing capabilities is a responsibility of UI plug-in developers.

Component Hierarchy

Ultimately, the project architecture should be able to be expressed as a hierarchy of "components." For example:

```
DTP project
    Installation Package A
        Feature 1
            Plug-in 1.a
            Plug-in 1.b
            ...
        Feature 2
            Plug-in 2.a
            Plug-in 2.b
            ...
        ...
    ...
```

This hierarchy will facilitate an appropriate feature and plug-in architecture for both project UI components and user assistance contributions.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

Architectural Specifications

To facilitate good user assistance architecture and design, the following questions must be answered by the project architectural specifications.

Download/Installation Packages

- Will the project provide multiple installation packages? (e.g., subsets of the complete project deliverables)
- Will any installation packages be defined at a level higher than features?
  - If so, how will the relationships between multiple features in an installation package be expressed? (e.g., dependency, colocation affinity, “nested” features, etc.)

Features

- What are the features to be delivered?
- What is the ID of each feature to be delivered?
- What functionality and UI components will be included in each feature?
- What is the ID of each plug-in that contributes the functionality and/or UI components in each feature?
- Will any features be “nested?” (e.g., features that include other features)
  - If so, how many top-level features will be delivered?
  - What is the feature ID of each top-level feature?
  - What are the dependencies between top-level features?
  - How many subfeatures will be included in each top-level feature?
  - What is the feature ID of each subfeature in each top-level feature?

Note: These questions may iterate to as many levels as features are nested.

Capabilities (Categories)/Activity Enablement

- How will capabilities correlate to features?
  - Which features will be associated with each capability (or activity)?
- How will capabilities correlate to the plug-ins that contribute functionality and/or UI components?
  - Which plug-ins will be associated with each capability (or activity)?

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

User Assistance Plug-in Structure

Each user assistance component should be delivered as a separate plug-in. These components include, but are not limited to, the following:

- Welcome page (intro) extensions
- Help content
- Context-sensitive help
- Cheat sheets

Contributing different types of user assistance content in separate plug-ins allows more flexibility in defining the content architecture (i.e., plug-ins that contribute content at the appropriate level in the project's component hierarchy).

The following sections cover guidelines for each of each type of user assistance component.

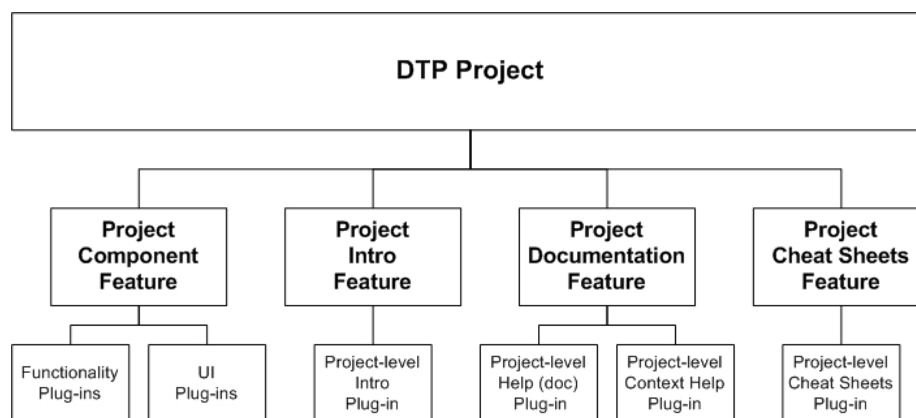
Project-Level Plug-ins

All installation packages should include a project-level plug-in for each relevant type of user assistance contribution. The project-level plug-ins will refer to platform extension points, which are defined by either the Eclipse platform or a DTP platform (or framework) component.

Project-level plug-ins will declare additional extension points or anchors, to which lower-level (component) plug-ins contribute. Thus, each project-level user assistance plug-in will contribute the "framework" for all other project-specific content contributions.

By including project-level plug-ins in every installation package, we ensure that the project's user assistance content will appear consistently in Eclipse presentation mechanisms, regardless of which particular project subsets or features are installed (or enabled) at any time.

The following diagram shows the project-level user assistance architecture:



Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

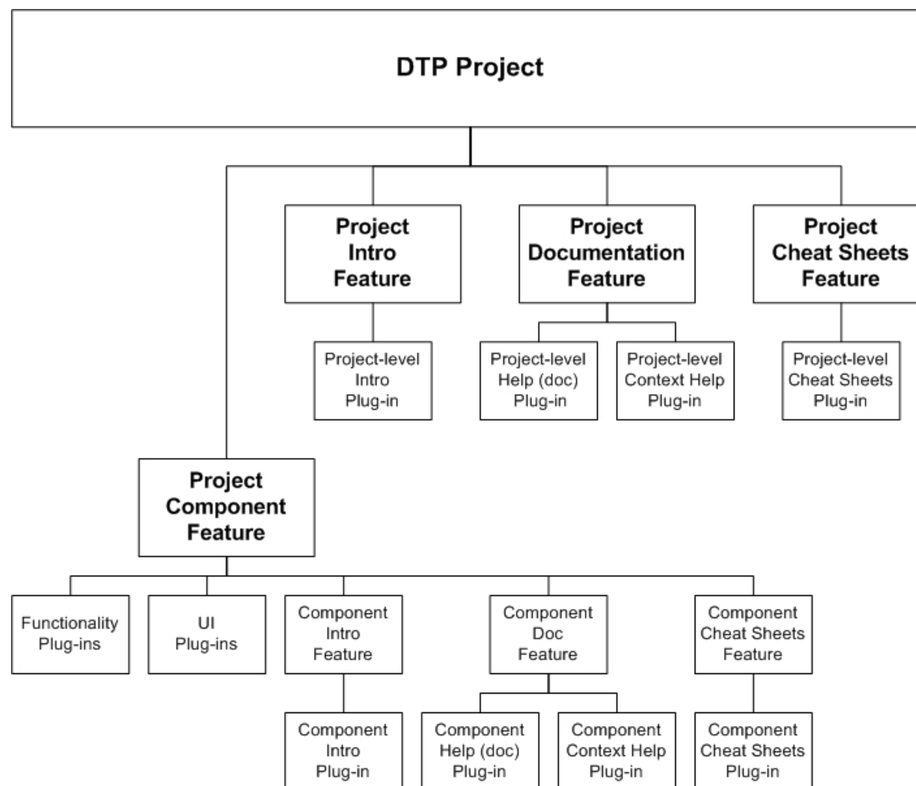
Component-Level Plug-ins

Each component-level installation package should include a component-level plug-in for each relevant type of user assistance contribution. Component-level user assistance plug-ins will be packaged in component-level user assistance features.

Component-level user assistance plug-ins will refer to extension points (or anchors), which are defined by one or more of the following:

- The Eclipse platform
- A DTP platform (or framework) component
- A project-level user assistance plug-in

The following diagram shows the combined project- and component-level user assistance architecture, and how the component-level user assistance architecture is related to component-level features of the project:



Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

User Assistance Feature IDs and Plug-in IDs

To take advantage of capabilities, user assistance features and plug-ins should follow the naming conventions of their corresponding project features and UI plug-ins.

For example:

- Project feature ID org.eclipse.datatools.*feature_name*
- Documentation feature ID org.eclipse.datatools.*feature_name*.doc
- UI plug-in ID org.eclipse.datatools.*plugin_name*
- Documentation plug-in ID org.eclipse.datatools.*plugin_name*.doc

This allows both documentation components and their corresponding functional (or UI) components to be matched to a single activityPatternBinding expression that specifies "org.eclipse.datatools.*feature_name**" or "org.eclipse.datatools.*plugin_name**".

Extending from the examples above, plug-ins that contribute different types of user assistance content should conform to the following ID conventions:

- Documentation (main help) content: org.eclipse.datatools.*plugin_name*.**doc**
- Context-sensitive help: org.eclipse.datatools.*plugin_name*.**doc.contexts**
- Welcome page (intro) content: org.eclipse.datatools.*plugin_name*.**intro**
- Cheat sheets: org.eclipse.datatools.*plugin_name*.**cheatsheets**

Welcome Page (Intro)

Extensions to the Eclipse platform or SDK universal intro should be contributed by one or more separate plug-ins, based on their relationship to DTP project features, component features, and UI plug-ins.

Project-level Intro Plug-in

All installation packages should include a project-level intro contribution plug-in, which points to the anchors defined by the universal intro, and declares additional anchors for lower-level (component or subproject) intro plug-ins. Thus, the project-level intro plug-in contributes the "framework" for all other DTP project intro content contributions.

By including the project-level intro plug-in in every installation package, we ensure that the project's intro content will be presented consistently in the Eclipse Welcome pages, regardless of which particular project subsets or features are installed (or enabled) at any time.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

Help Content

Based on the mapping of documentation (doc) plug-ins to project components and capabilities, each DTP component (or top-level project feature) should have a separate doc plug-in.

For example, Connectivity should have a doc plug-in, SQL Development should have a doc plug-in, etc.

Note: A major project component (top-level feature) might have more than one doc plug-in, but each major project component will have at least one doc plug-in.

Project-level Doc Plug-in

All installation packages should include a common, project-level doc plug-in, which declares anchors (content extension points) for the lower-level doc plug-ins. Thus, the project-level doc plug-in contributes the “framework” for all DTP project documentation presented on the Eclipse bookshelf (Help window TOC).

By including the project-level doc plug-in in every installation package, we ensure that the DTP project documentation will be presented consistently in the Eclipse help system, regardless of which particular project subsets or features are installed (or enabled) at any time.

TOC Contributions

All doc plug-ins must declare an extension to **org.eclipse.help.toc** in their plug-in manifests (plugin.xml files) to identify their TOC contributions.

Only the project-level doc plug-in will declare its TOC contribution to be primary (i.e., appearing at the highest level in the Help window TOC), so the TOC contributions of all lower-level (component) doc plug-ins will appear only within the hierarchy of the project-level TOC.

The project-level doc plug-in makes this declaration in the plugin.xml file:

```
<extension point="org.eclipse.help.toc">
   <toc file="toc.xml"
      primary="true"/>
   </extension>
```

All other doc plug-ins must declare their TOC contributions as follows:

```
<extension point="org.eclipse.help.toc">
   <toc file="toc.xml"
      primary="false"/>
   </extension>
```

or

```
<extension point="org.eclipse.help.toc">
   <toc file="toc.xml"/>
   </extension>
```

Note: The default is `primary="false"`.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

Anchors and Links in toc.xml

The project-level doc plug-in will declare anchors in its toc.xml file, to which the lower-level doc plug-ins will link.

When rendered by the Eclipse help system, the TOC contributions of lower-level doc plug-ins are merged into the project-level TOC. The anchor location that each lower-level doc plug-in links to will determine the location of its TOC contribution within the project-level TOC.

Note: If multiple TOC contributions link to the same anchor, the order in which the contributions are presented is indeterminate (it could vary from time to time). Therefore, each anchor should be targeted by only one lower-level TOC contribution.

Anchors declared in the project-level doc plug-in's toc.xml file look like this:

```
<toc label="Data Tools Platform">
   <topic href="overview.html" label="Project-level Overview Content">
   <anchor id="feature1_contrib"/>
   <anchor id="feature2_contrib"/>
   ...
</toc>
```

A link is declared by using the *link_to* attribute on the <toc> element in the lower-level doc plug-in's toc.xml file:

```
<toc label="DTP Feature 1 Name"
   link_to="../org.eclipse.datatools.doc/toc.xml#feature1_contrib">
   <topic href="feature1_content.html" label="Feature 1 Content"/>
   ...
</toc>
```

DITA Maps

Project-level doc plug-ins can be defined by a DITA map (ditamap file), in which anchors are defined using <anchor> elements. For example:

```
<map id="org.eclipse.datatools.doc">
   <topicref href="some_DTP_content.xml"/>
   <topicref href="more_DTP_content.xml">
      <anchor id="feature1_contrib"/>
   </topicref>
   <anchor id="feature2_contrib"/>
   <anchor id="feature3_contrib">
      <anchor id="DTP_contrib_A"/>
      <anchor id="DTP_contrib_B"/>
   </anchor>
   <anchor id="feature4_contrib"/>
</map>
```

Note: In terms of the map structure, an <anchor> element is valid anywhere that a <topicref> element is. Also, as shown above, <anchor> elements can be nested the same way as <topicref> elements to create hierarchical structures.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

When DITA is used for lower-level doc plug-ins, links to the project-level TOC will be defined using the *anchorref* attribute on the <map> element in the ditamap:

```
<map id="org.eclipse.datatools.blahblah.doc"
   anchorref="../org.eclipse.datatools.doc/toc.xml#feature1_contrib">
   <topicref href="DTP_content.xml"/>
   ...
</map>
```

Note: The value of the *anchorref* attribute must be the output plug-in-relative path (in the Eclipse installation), from the lower-level doc plug-in generated by the ditamap to the <anchor> element ID, in the toc.xml file of the project-level doc plug-in.

DITA maps that use *anchorref* can also use <anchor> elements to implement multi-tiered doc plug-ins, which might be necessary to deliver content partitioned appropriately for the project's component architecture.

Context-Sensitive Help

Context-sensitive help infrastructure (i.e., mapping help contexts to help topics) will be contributed by dedicated plug-ins. Those plug-ins will contribute context manifests (contexts.xml files) that point to the content contributed by corresponding doc plug-ins (the core help content), but they will not contribute any content themselves.

Context-sensitive help plug-ins will "mirror" the architecture of content-contributing (doc) plug-ins. For each doc plug-in that contributes topics used for context-sensitive help, a corresponding context-sensitive help plug-in will be delivered. In general, help content contributions and context-sensitive help contributions will always be "in sync."

Each context-sensitive help plug-in will be named to match its corresponding doc plug-in. For example:

- org.eclipse.datatools.connectivity.doc
- org.eclipse.datatools.connectivity.doc.contexts

Context-Sensitive Help Implementation

In the UI code, editors and other controls will implement methods of IContextProvider to handle context-sensitive help requests.

```
public int getContextChangeMask() {
       return SELECTION;
}
public IContext getContext(Object target) {
       return HelpSystem.getContext(getContextId(getHelpKey(target)));
}
public String getSearchExpression(Object target) {
       return getSearchExpression(getHelpKey(target));
}
```

Rather than declaring help context IDs directly, UI controls will use "help keys."

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

Similar to externalized messages, helpKey constants will be declared as *public static final String* in an interface class. For more information, see Rules for helpKey Interface Classes.

Each helpKey constant will be used to reference the actual help context ID and a context-specific help search expression (defined in .properties files).

To support the abstraction from help context IDs to helpKey constants, UI controls will implement a getHelpKey method, identify the appropriate .properties files for help context IDs and search expressions, and implement static methods for getContextId and getSearchExpression.

Note: This abstraction from help context IDs to helpKey constants is supported by a documentation build system at Sybase. This is simply our design / implementation to enable efficiency of a single helpKey to support both context sensitive help and search expressions. This particular implementation does not have to be adopted by non-Sybase contributed code. The final Eclipse plugin contributions will still be able co-exist for DTP.

getHelpKey

```
public static String getHelpKey(Object target) {
        if (target instanceof Control) {
            for (Control control = (Control)target; control != null;
            control = control.getParent()) {
                Object contextId = control.getData("org.eclipse.ui.help");
                //$NON-NLS-1$
                if (contextId != null && contextId instanceof String) {
                    return (String)contextId;
                }
            }
        }
        return null;
}
```

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

getContextId

```
public static ResourceBundle _contextIdResourceBundle = null;
public static String getContextId(String helpKey) {
        if (helpKey == null) {
            return null;
        }
            try {
                if (_contextIdResourceBundle == null) {
                _contextIdResourceBundle =
            ResourceBundle.getBundle("ContextIds_ResourceBundle"); //$NON-NLS-1$
                }
                String bundleString = _contextIdResourceBundle.getString(helpKey);
                    return bundleString;
            } catch (MissingResourceException e) {
                return null;
            }
}
```

getSearchExpression

```
public static ResourceBundle _searchExpressionResourceBundle = null;
public static String getSearchExpression(String helpKey) {
        if (helpKey == null) {
            return null;
        }
            try{
                if (_searchExpressionResourceBundle == null) {
                    _searchExpressionResourceBundle =
        ResourceBundle.getBundle("SearchExpressions_ResourceBundle");
        //$NON-NLS-1$
                }
                String bundleString =
        _searchExpressionResourceBundle.getString(helpKey);
                return bundleString;
            } catch (MissingResourceException e) {
                return null;
            }
}
```

Each helpKey constant will be associated with two string values:

- The actual help context ID
- A help search expression

Key-value pairs will be defined in two .properties files (one for context IDs and one for search expressions).

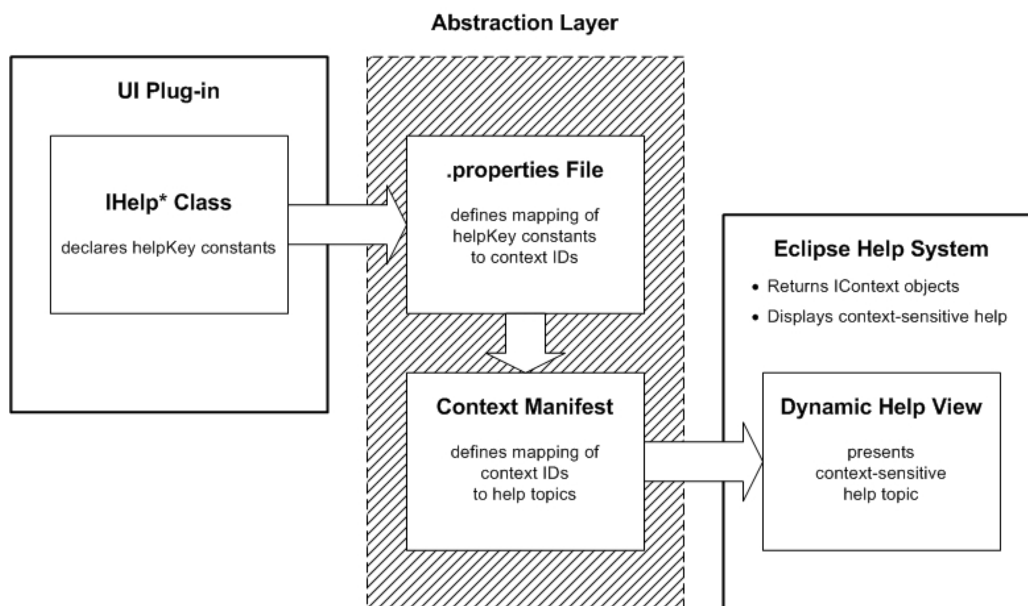
Though meaningless to the Eclipse help system, the helpKey constants provide an abstraction from the actual help context IDs in the UI code, with the following benefits:

- Simplified handling of help context IDs and context-specific search expressions within the UI code.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

- Development teams can associate new helpKey constants with UI controls, without necessitating that corresponding help context IDs or help search expressions exist.
- Documentation teams can define and modify help context IDs, and the mapping from helpKey constants to context IDs, without necessitating any change in the UI code.
- Documentation teams can define and modify context-specific help search expressions, and the mapping from helpKey constants to search expressions, without necessitating any change in the UI code.
- Generating the Eclipse infrastructure for context-sensitive help (context manifests) can be somewhat automated.

The following diagram shows how helpKey constants provide an abstraction from help context IDs.

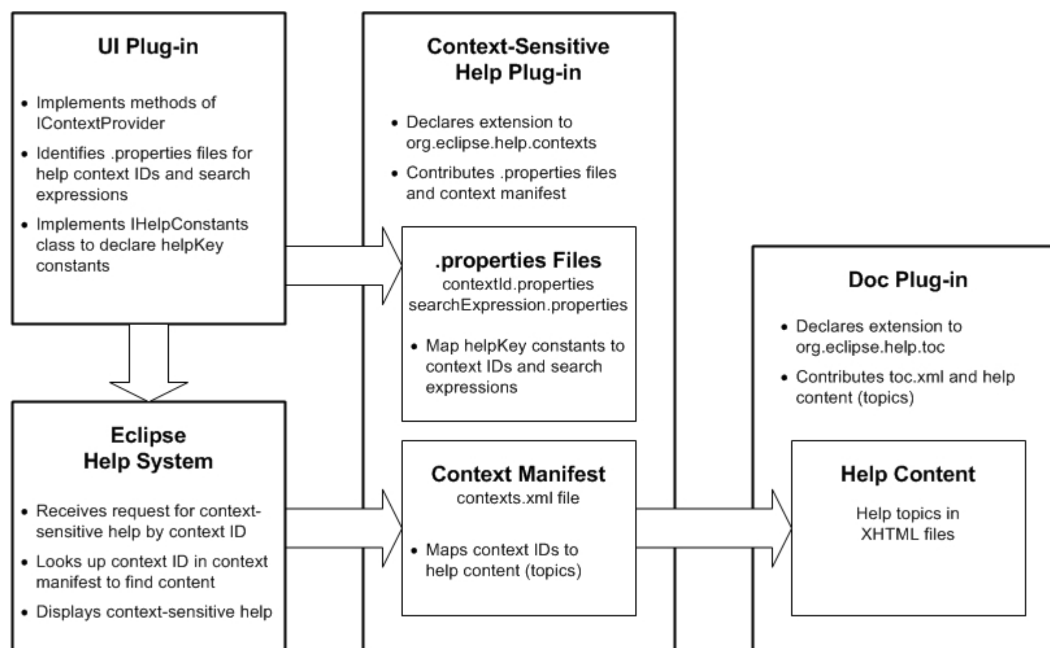


Context-sensitive help plug-ins will contribute the abstraction layer shown above. That includes:

- One or more sets of .properties files (for context IDs and search expressions)
- One or more Eclipse context manifests (contexts.xml files)

The following diagram shows how UI plug-ins will interact with context-sensitive help plug-ins and the Eclipse help system to provide dynamic, context-sensitive help for users.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0



Rules for helpKey Interface Classes

Documentation teams will depend on .java source code files as the original and definitive source of all helpKey constant strings.

Therefore, Development teams that implement the interface classes to declare helpKey constants must adhere to the following rules for the .java file format:

- Package declaration — Each .java file must declare a package that identifies exactly one unique UI plug-in ID. For example:

  ```
  package org.eclipse.datatools.connectivity.UIplugin;
  ```

  where *UIplugin* is any conventional name for a UI plug-in.

- Class declaration — Each .java file must contain exactly one interface class declaration, in the form:

  ```
  public interface IYourHelpKeyClassName
  ```

  where *IYourHelpKeyClassName* is any conventional name for an interface class.

- Constant declarations:
  - Each .java file must declare all of the helpKey constants for exactly one UI plug-in.
  - Each helpKey constant must be declared as *public static final String*, in the form:

    ```
    public static final String HELP_KEY_STRING = "";
    ```

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

where *HELP_KEY_STRING* is any literal character string used for a helpKey constant in a UI editor or control.

- Help context comments:
  - All help context comments must be in standard Javadoc format. For example:

```
/**
 * Used for logging preference and logging view.
 */
```

  - Each help context comment must be preceded by at least one blank line.
  - Each help context comment must immediately precede its associated *public static final String* declaration, with no blank lines between the comment and the declaration. For example:

```
  /**
   * Used for logging preference and logging view.
   */
public static final String HELP_LOGGING_CONTEXT = "";
```

  - Any help context comment may occupy multiple lines.
  - Each help context comment must provide sufficient information for the Doc team to determine the exact location of the associated editor or control within the UI.

Note: No comments (of any type) that precede the class declaration will be parsed, and no implementation comments (/*...*/ and //) will be parsed anywhere in the file.

Team Collaboration and Responsibilities

Since the helpKey constants (defined and owned by Dev teams) are independent of the help context IDs (defined and owned by Doc teams), coordinating UI plug-ins with doc plug-ins will require ongoing collaboration, and it will impose responsibilities on both Dev teams and Doc teams.

Development Team Responsibilities

Development teams will be responsible to:

- Implement methods of IContextProvider for all appropriate UI controls.
- For each UI plug-in, implement an interface class that declares helpKey constants, and then communicate a list of all helpKey constants to Doc teams in a timely manner:
  - Lists of helpKey constants will be provided in the form of .java source code files for the interface classes that declare helpKey constants.
  - Each .java file provided to a Doc team must have a unique file name. To ensure file name uniqueness, Dev teams must prepend the original .java file name with the UI plug-in ID. For example:

    org.eclipse.datatools.connectivity.UIplugin_IHelpContext.java

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

- All .java files provided as helpKey lists must be annotated with Javadoc "help context comments" that provide a cue as to where to find each associated UI control. For more information, see Rules for helpKey Interface Classes.
- The .java files will be sent to a designated Doc team contact at the beginning of each project, when helpKey constants are initially defined, and then subsequently upon any change.
- When any change occurs, Dev teams will also be responsible to provide a "diffs" file that calls out all changes since the previous helpKey list was sent.

Documentation Team Responsibilities

Documentation teams will be responsible to:

- Define the help contexts, and the help context ID strings and context-specific search expressions for each help context.
- Create and maintain the .properties files for context IDs and search expressions, based on helpKey lists (.java source code files) communicated from Dev teams.
- Create and maintain the context-sensitive help plug-ins that contribute those files.

Note: Defining help contexts, and the corresponding context IDs and search expressions, and creating and maintaining the .properties files will require human work processes.

Java Source File Conventions

Java source code files for the interface classes that declare helpKey constants will be the original and definitive source of all helpKey constant strings.

Those files are flat ASCII text files, each of which contains exactly one *public interface* class declaration. For example:

```
public interface IHelpContext
{
...
}
```

The name of the interface class (IHelpContext in the example above) will match the name of the file (e.g., IHelpContext.java).

Note: The actual class names and file names will be determined by each UI component Dev team.

Within the *public interface* class declaration, there will be a series of *public static final String* declarations, each of which defines a literal helpKey constant string. For example:

```
public static final String HELP_LOGGING_CONTEXT = "";
```

The literal helpKey constant string (HELP_LOGGING_CONTEXT in the example above) is the sequence of non-blank characters following the *String* keyword.

Each *public static final String* declaration will be accompanied by a Javadoc comment that describes the location of its associated control in the user interface. Such help context comments will immediately precede each *public static final String* declaration. For example:

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

```
   /**
    * Used for logging preference and logging view.
    */
public static final String HELP_LOGGING_CONTEXT = "";
```

Each help context comment should be sufficient for the Doc team, which is responsible for defining help contexts, context IDs, and search expressions, to:

- Determine the help context associated with each helpKey constant
- Define an appropriate context ID string for each help context
- Point to an appropriate topic to provide the context-sensitive help content
- Define an appropriate context-specific help search string for each help context

In the event that a help context comment is not sufficient for this purpose, the Doc team author (or IA) will request that the Dev team expand or modify the comment so that it meets this requirement.

Each Dev team will be responsible to meet such requests, and provide an updated .java source code file (along with a diffs file) in a timely manner.

Properties File Conventions

Each helpKey constant will be associated with two other string values, the actual help context ID and a context-specific help search expression. Those key-value pairs will be defined in two .properties files, one for context IDs and one for search expressions.

File Names

Each context-sensitive help plug-in could contribute pairs of .properties files for several UI plug-ins. Therefore, all .properties files must have unique names.

For ease of maintenance, each .properties file name should correspond with the UI plug-in that it serves. For example:

- org.eclipse.datatools.connectivity.UIplugin.contextIds.properties
- org.eclipse.datatools.connectivity.UIplugin.searchExpressions.properties

File Content Format

All .properties files are flat ASCII text files that conform to the requirements of the java.util.Properties.load method:

http://java.sun.com/j2se/1.5.0/docs/api/java/util/Properties.html#load(java.io.InputStream)

The following additional rules apply to the content of the helpKey .properties files:

- All strings (helpKey constants, context IDs, and search expressions) are case sensitive.
- Each helpKey constant must be represented *exactly* as it appears in the .java source code file.
- The help context ID strings must **not** include any blank spaces or period (.) characters. For more information, see Defining Help Context IDs.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

- Search expressions must conform to the rules for Apache Lucene query parser syntax: http://lucene.apache.org/java/docs/queryparsersyntax.html

Note: The terms (keywords) in search expressions are not case sensitive, but the Boolean operators (e.g., AND, NOT, OR) must be all upper case.

Context IDs File Example

The following example shows an excerpt of a .properties file that associates helpKey constants with help context ID strings.

```
# Lines that start with either the '#' or '!' character are comments.
! This is a comment, but # is more conventional.
BLAH_BLAH_HELPKEY = help_blah_blah_context
BLAH_BLAH_HELPKEY2 = help_blah_blah2_context
...
```

In the example above, the first helpKey constant is:

```
BLAH_BLAH_HELPKEY
```

The first help context ID string is:

```
help_blah_blah_context
```

Search Expressions File Example

The following example shows an excerpt of a .properties file that associates helpKey constants with help search expressions.

```
#
BLAH_BLAH_HELPKEY = "data tools" AND "blah blah wizard"
BLAH_BLAH_HELPKEY2 = "data tools" AND ("blah blah editor" OR "wingnut")
...
```

In the example above, the first helpKey constant is:

```
BLAH_BLAH_HELPKEY
```

The first help search expression is:

```
"data tools" AND "blah blah wizard"
```

Documentation Team Workflow

Documentation teams should refer to the project's architectural specifications to define an appropriate architecture (i.e., granularity, content partitioning, plug-in naming, etc.) for doc plug-ins.

With that information, Doc teams can begin the process to create context-sensitive help plug-ins.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

The following list summarizes the overall Doc team workflow to create context-sensitive help plug-ins:

1. Get the helpKey list (.java source code file) for each UI plug-in from the Dev team.
2. Analyze the helpKey lists and their associated UI objects to define help contexts. (For more information, see Defining Help Context IDs.)

   Doc teams must determine whether:

   - helpKey constants alone are sufficient to identify help contexts, or
   - context ID strings must be defined to aggregate groups of helpKey constants into common help contexts

   Note: It may be preferable to aggregate help contexts in the .properties file, instead of defining mapping for multiple contexts to a single topic (e.g., in the context manifest). This is a judgment call for the Doc team lead and the IAs responsible for maintaining the context-sensitive help plug-ins. For more information, see Defining Topics Mapped to Context IDs.
3. Analyze help contexts and help topics to define context-specific help search expressions. (For more information, see Defining Help Search Expressions.)
4. Create two .properties files for each .java file, one for context IDs and one for search expressions. (For more information, see Properties File Conventions.)
5. Based on the results of help context analysis and help content (topic) analysis, define the appropriate mapping of helpKey constants to context IDs and search expressions in the .properties files.
6. Save the .properties files, and check in to source control.

Defining Help Context IDs

Since helpKey constants (defined and owned by Dev teams) are independent of help context IDs, Doc teams are responsible to define the help contexts that the Eclipse help system will use to find context-sensitive help content (topics), and associate each helpKey constant with a help context ID.

Additional considerations for defining help context IDs:

- Each help context ID represents a single "help context," but any help context could occur more than once in a single UI component (plug-in contribution). Therefore, any help context ID can be associated with any number of helpKey constants, but
  - Each helpKey constant must be associated with exactly one help context ID.
- Each help context within a UI component should be associated with a unique help context ID, however
  - Help context IDs need not be unique across all plug-ins. For example, a help context ID could be declared by more than one UI plug-in, if the Doc team determines that the actual help context is the same for multiple UI plug-ins.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

The following list summarizes the tasks necessary to define a help context ID:

1. Open the .java source code file and find the helpKey constant with its associated comment. For example:

```
  /**
   * Used for logging preference and logging view.
   */
public static final String HELP_LOGGING_CONTEXT = "";
```

2. Verify that the comment associated with the helpKey constant is sufficient to identify the corresponding UI control.
3. Verify the plug-in ID of the UI component that declares the helpKey constant (identified in the package declaration at the top of the .java file).
4. If it is necessary to define a separate help context ID (based on help context analysis), rather than using the helpKey constant itself, define a help context ID character string that conforms to the following rules:
   - The string may contain both upper- and lower-case characters.
   - The string must **not** contain any blank spaces or period (.) characters.
   - The string should be sufficiently mnemonic to allow other authors (or IAs) to understand the help context.

   For example:

   ```
   help_connectivity_logging_context
   ```

   Note: While there is no practical limit on the number of characters in a help context ID string, the string should be just long enough to be sufficiently mnemonic, but not longer than necessary for that purpose.
5. Enter the help context ID in the .properties file, following the = character on a single line with the helpKey constant. For example:

   ```
   HELP_LOGGING_CONTEXT = help_connectivity_logging_context
   ```

6. After all context IDs are defined and entered in the .properties file, save the file.

Defining Help Search Expressions

Each helpKey constant (defined and owned by a component Dev team) can be associated with a context-specific help search expression, which the Eclipse help system will use to provide search results in the dynamic help view.

Doc teams are responsible to define the context-specific help search expressions.

Additional considerations for defining help search expressions:

- A context-specific help search expression should be defined for each help context.
- Each help search expression can be associated with multiple help context IDs, but
  - Each helpKey constant must be associated with exactly one help search expression.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

- Each help context within a UI component (plug-in) can be associated with a unique help search expression, however
  - It may be more practical to define a “generic” help search expression at the UI component level, which would serve to limit search results to DTP (or project-specific) content, rather than including content contributed by the Eclipse platform (or other open source projects), or content contributed by other (non-DTP) colocated projects.

Note: For each helpKey constant, the help context should be defined *before* the help search expression is defined.

Search expressions are associated with help contexts. If Doc teams choose to aggregate multiple helpKey constants into a single help context, all of the helpKey constants associated with that help context should be associated with the same search expression. Therefore, the first step is to analyze and define the help contexts.

The following list summarizes the tasks necessary to define a context-specific help search expression:

1. Make sure that a context ID is already defined for the helpKey constant. For more information, see Defining Help Context IDs.
2. Refer to the existing context ID .properties file to find the help context ID for the helpKey constant.
3. If a search expressions .properties file does not exist, create a new .properties file for search expressions.
4. Open the .java source code file and find the helpKey constant with its associated comment. For example:

```
  /**
   * Used for logging preference and logging view.
   */
public static final String HELP_LOGGING_CONTEXT = "";
```

5. Verify that the comment associated with the helpKey constant is sufficient to identify the corresponding UI control.
6. Verify the plug-in ID of the UI component that declares the helpKey constant (identified in the package declaration at the top of the .java file).
7. Define a help search expression that conforms to the following rules:
   - Search terms (keywords) should be enclosed in matching quote characters. If the search term is a multi-word phrase, it must be enclosed in quote characters. For example:

     ```
     "data tools"
     ```

     Note: If two consecutive words appear without quotes in a search expression, the expression is interpreted as if the two words were separate terms, with an implicit OR operator. For example:

     ```
     data tools
     ```

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

would be interpreted as:

```
"data" OR "tools"
```

- Search terms (keywords) are **not** case sensitive, but in general, they should be specified in all lower case.
- Boolean operators (e.g., AND, NOT, OR, etc.) in the search expression must be specified in all upper case. For example:

```
"data tools" AND "conectivity"
```

- Search expression strings can include parentheses to group or order logical expressions. For example:

```
("connectivity" OR "database") AND "data tools"
```

Note: There is no practical limit on the number of characters in a help search expression.

Refer to the Apache Lucene query parser syntax for more information about help search expression rules: http://lucene.apache.org/java/docs/queryparsersyntax.html

8. Enter the help search expression on the same line with the helpKey constant, following the = character. For example:

```
HELP_LOGGING_CONTEXT = "data tools" AND "logging"
```

9. After all search expressions are defined and entered in the .properties file, save the file.

Context-Sensitive Help Plug-in Structure

Context-sensitive help plug-ins will “mirror” the content-contributing doc plug-ins. For each doc plug-in that contributes topics for context-sensitive help, a corresponding context-sensitive help plug-in will be delivered.

Each context-sensitive help plug-in will contribute a context manifest (contexts.xml file) that maps help context IDs only to topics contributed by its corresponding doc plug-in; it will **not** define context ID mapping to topics contributed by any other plug-in.

Each context-sensitive help plug-in will be named to match its corresponding doc plug-in. For example:

- org.eclipse.datatools.connectivity.doc
- org.eclipse.datatools.connectivity.doc.**contexts**

The overall architecture for help content (TOC) contributions and context-sensitive help contributions will always be “in sync.”

Plug-ins that contribute the context-sensitive help infrastructure must declare an extension to **org.eclipse.help.contexts** in their plugin.xml files to identify help context contributions.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

For example:

```
<extension point="org.eclipse.help.contexts">
   <contexts file="dtp_connectivity_contexts.xml"
      plugin="org.eclipse.datatools.connectivity.UIplugin"/>
</extension>
```

Because a context-sensitive help plug-in will contribute properties files, which could be common to several UI plug-ins, as well as a context manifest that could support several UI plug-ins, the “component level” with which a doc plug-in and its corresponding context-sensitive help plug-in are associated must be evaluated carefully.

Doc teams must rely on the project’s architectural specifications and the .java (helpKey constant) source code files provided by Dev teams to determine the appropriate capabilities and features with which to coordinate doc plug-ins and context-sensitive help plug-ins.

Cheat Sheets

Similar to other user assistance plug-ins, cheat sheets should be contributed by separate plug-ins so they can be activated/deactivated based on capabilities associated with UI plug-ins.

Cheat sheet content will be developed using the Cheat Sheet Editor provided in Eclipse 3.3, and cheat sheet plug-ins will be constructed from a plug-in project in the Eclipse PDE.

Project-level Cheat Sheet Plug-in

All installation packages should include a common, project-level cheat sheet plug-in, which declares a project-level cheat sheet category. Thus, the project-level cheat sheet plug-in contributes the “framework” for all DTP project cheat sheets presented in the Eclipse Cheat Sheet Selection dialog.

By including the project-level plug-in in every installation package, we ensure that the DTP project cheat sheets will be presented consistently in Eclipse, regardless of which particular project subsets or features are installed (or enabled) at any time.

Cheat Sheet Contributions

To identify cheat sheet contributions, all cheat sheet plug-ins must declare an extension to **org.eclipse.ui.cheatsheets.cheatSheetContent** in their plug-in manifests (plugin.xml files).

To define a project-level cheat sheet category, the project-level cheat sheet plug-in makes this declaration in the plugin.xml file:

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

```
<extension point="org.eclipse.ui.cheatsheets.cheatSheetContent">
   <category id="org.eclipse.datatools.cheatsheets"
      name="%cheatsheet.category.dtp_project">
   </category>
   ...
</extension>
```

The project-level category ID should be the ID of the DTP project-level plug-in, followed by ".cheatsheets". (This will be the same as the project-level cheat sheet plug-in ID.)

Note: All other DTP cheat sheet plug-ins will reference the DTP project category ID.

The project-level category name ("%cheatsheet.category.dtp_project") is an externalized property value, which is defined in the plugin.properties file of the project-level cheat sheet plug-in. The actual category name should be the name of the DTP project. For example:

```
%cheatsheet.category.dtp_project = Data Tools Platform
```

For each cheat sheet contributed (at any level), the cheat sheet plug-in must include the following in its **org.eclipse.ui.cheatsheets.cheatSheetContent** extension declaration:

```
<extension point="org.eclipse.ui.cheatsheets.cheatSheetContent">
   ...
   <cheatsheet
      category="org.eclipse.datatools.cheatsheets"
      contentFile="$nl$/cheatsheets/cheatsheet_file.xml"
      id="org.eclipse.datatools.DTPcomponent.cheatsheet.task_name"
      name="%task_name.cheatsheetName">
      <description>%task_name.cheatsheetDesc</description>
   </cheatsheet>
   ...
</extension>
```

where:

- *cheatsheet_file.xml* identifies the cheat sheet content file
- *DTPcomponent* identifies the DTP project top-level component
- *task_name* identifies the task that the cheat sheet guides

All cheat sheet content files should be located in a *cheatsheets* subdirectory, under the cheat sheet plug-in root.

The cheat sheet name and description ("%*task_name*.cheatsheetName" and "%*task_name*.cheatsheetDesc") should both reference externalized property values, which are defined in the plugin.properties file of the cheat sheet plug-in.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0

User Assistance Delivery Formats

All user assistance plug-ins will be delivered in features.

All user assistance plug-ins will be delivered as jar files (if feasible), with an appropriate jar manifest.

All human-readable content in plug-in manifests (i.e., in extension declarations) will be externalized using property keys, with the actual content defined in the plugin.properties file. This is an Eclipse best practice to facilitate localization.

Copyright © 2007 Sybase, Inc. Made available under the terms of the EPL V1.0