Parametric Technology Corporation

Creo® View 2.0
Java Toolkit Developer’s Guide

March 2012

Copyright © 2012 Parametric Technology Corporation and/or Its Subsidiary Companies. All Rights Reserved.
User and training guides and related documentation from Parametric Technology Corporation and its subsidiary companies (collectively "PTC") are subject to the copyright laws of the United States and other countries and are provided under a license agreement that restricts copying, disclosure, and use of such documentation. PTC hereby grants to the licensed software user the right to make copies in printed form of this documentation if provided on software media, but only for internal/personal use and in accordance with the license agreement under which the applicable software is licensed. Any copy made shall include the PTC copyright notice and any other proprietary notice provided by PTC. Training materials may not be copied without the express written consent of PTC. This documentation may not be disclosed, transferred, modified, or reduced to any form, including electronic media, or transmitted or made publicly available by any means without the prior written consent of PTC and no authorization is granted to make copies for such purposes.

Information described herein is furnished for general information only, is subject to change without notice, and should not be construed as a warranty or commitment by PTC. PTC assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

The software described in this document is provided under written license agreement, contains valuable trade secrets and proprietary information, and is protected by the copyright laws of the United States and other countries. It may not be copied or distributed in any form or medium, disclosed to third parties, or used in any manner not provided for in the software licenses agreement except with written prior approval from PTC.

UNAUTHORIZED USE OF SOFTWARE OR ITS DOCUMENTATION CAN RESULT IN CIVIL DAMAGES AND CRIMINAL PROSECUTION. PTC regards software piracy as the crime it is, and we view offenders accordingly. We do not tolerate the piracy of PTC software products, and we pursue (both civilly and criminally) those who do so using all legal means available, including public and private surveillance resources. As part of these efforts, PTC uses data monitoring and scouring technologies to obtain and transmit data on users of illegal copies of our software. This data collection is not performed on users of legally licensed software from PTC and its authorized distributors. If you are using an illegal copy of our software and do not consent to the collection and transmission of such data (including to the United States), cease using the illegal version, and contact PTC to obtain a legally licensed copy.

Important Copyright, Trademark, Patent, and Licensing Information:
See the About Box, or copyright notice, of your PTC software.

UNITED STATES GOVERNMENT RESTRICTED RIGHTS LEGEND
This document and the software described herein are Commercial Computer Documentation and Software, pursuant to FAR 12.212(a)-(b) (OCT,Äô95) or DFARS 227.7202-1(a) and 227.7202-3(a) (JUN,Äô95), and are provided to the US Government under a limited commercial license only. For procurements predating the above clauses, use, duplication, or disclosure by the Government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 (OCT,Äô88) or Commercial Computer Software-Restricted Rights at FAR 52.227-19(c)(1)-(2) (JUN,Äô87), as applicable. 01012012

Parametric Technology Corporation, 140 Kendrick Street, Needham, MA 02494 USA

Contents

About This Guide

Purpose vi
Related Documentation vi
Conventions vi
Documentation for PTC Products vii
Technical Support vii
Comments viii
Third-Party Products viii
Code Examples viii

Chapter 1: Fundamentals

Introduction to Creo View Toolkit 1-2
Installation Requirements 1-2
Prerequisites for Installation 1-2
System Requirements 1-3
Installing Creo View Toolkit 1-3
Configuration of Creo View Java Toolkit 1-5
About the User Interface 1-5
GUI Mode 1-5
Non-GUI Mode 1-7
Navigating Creo View 1-8
Overview of Creo View Data Structures 1-8

Chapter 2: Creo View Java Toolkit

Overview of the Java APIs 2-3
Class Types 2-3
Creo View-related Interfaces 2-4
Observers and Action Listeners 2-5
Utilities 2-6
Creating Applications Using Creo View Java Toolkit 2-7
Importing Packages 2-7
Application Hierarchy 2-7
Deployment 2-10

World Object 2-10
Support for Creo Parametric Files 2-12
Product Structure 2-13
Creating the Product Structure 2-13
Visiting the Product Structure 2-18
Modifying the Product Structure 2-20
Component Operations 2-21
View Operations 2-25
View Orientations 2-29
Screen Capture 2-30
Selection Operations 2-31
Using the Java Application 2-31
In the 3D Viewer 2-31
Accessing Combined View States 2-34
Annotations 2-37
Product Manufacturing Information (PMI) for Markup Objects 2-38
Circle Operations 2-41
Point Operations 2-42
Lines of a Drawing Leader 2-43
Geometrical Data 2-46
Manipulation Modes 2-48
Accessing Viewable Files 2-48
Accessing Properties 2-50
Layer Operations 2-51
Uploading and Downloading Files from a Webserver 2-51

Chapter 3: Sample Applications

Installing Sample Applications 3-2
Running the Sample Applications Using Source Files 3-2
Running the Sample Applications Using the Pre-compiled Jar File 3-3
Details of Sample Applications 3-3

Index **Index-1**

About This Guide

This section contains information about the contents of this Developer's guide and the conventions used.

Topic	Page
Purpose	vi
Related Documentation	vi
Documentation for PTC Products	vii
Technical Support	vii
Comments	viii

Purpose

The *Creo View Java Toolkit Developer's Guide* describes how to use Creo View Java Toolkit, the Java customization toolkit for Creo View from PTC (Parametric Technology Corporation). Creo View Java Toolkit provides customers and third-parties the ability to expand Creo View capabilities by writing Java code and seamlessly integrating the resulting application into Creo View.

This manual introduces Creo View Java Toolkit, its features, and the techniques and background knowledge users require to use it.

Related Documentation

The documentation for Creo View Java Toolkit includes:

- Online reference documentation—Describes Creo View Java Toolkit function syntax. The reference documentation is available at `<creo_view_api_loadpoint>/documentation/java/`

Conventions

The following table lists conventions and terms used throughout this book.

Convention	Description
UPPERCASE	Creo View-type menu name (for example, PART).
Boldface	Windows-type menu name or menu or dialog box option (for example, **View**), or utility (for example, **promonitor**). Function names also appear in boldface font.
`Monospace (Courier)`	File names and code samples appear `in courier font`.
SMALLCAPS	Key names appear in smallcaps (for example, ENTER).
Emphasis	Important information appears in *italics*. Italic font also indicates function arguments.

Convention	Description
Choose	Highlight a menu option by placing the pointer on the option and pressing the left mouse button.
Select	A synonym for “choose” as above, Select also describes the actions of selecting elements on a model and checking boxes.

- Important information that should not be overlooked appears in notes like this.

  Note: All references to mouse clicks assume use of a right-handed mouse.

Documentation for PTC Products

You can access PTC documentation using the following resources:

- Product CD -- All relevant PTC documentation is included on the CD set.
- Reference Documents Web Site -- All books are available from the Reference Documents link of the PTC Web site at http://www.ptc.com/appserver/cs/doc/refdoc.jsp. On the Web site, choose the product or document type.

  A Service Contract Number (SCN) is required to access the PTC documentation from the Reference Documents Web site. For more information on SCNs, see Technical Support at:

  http://www.ptc.com/support/index.htm

Technical Support

Contact PTC Technical Support via the PTC Web site, phone, fax, or e-mail if you encounter problems using Creo View Web Toolkit or the product documentation.

For complete details, refer to "Contacting Technical Support" in the *PTC Customer Service Guide*. This guide can be found on the PTC Web site at:

http://www.ptc.com/support/cs_guide/cs_guide.pdf

You must have a Service Contract Number (SCN) before you can receive technical support. If you do not have an SCN, contact PTC Maintenance Department using the instructions found in your *PTC Customer Service Guide* under "Contacting Your Maintenance Support Representative".

Comments

PTC welcomes your suggestions and comments on its documentation. You can submit your feedback through the online survey form at the following URL:

http://www.ptc.com/go/wc_pubs_feedback

Third-Party Products

Examples in this guide referencing third-party products are intended for demonstration purposes only. For additional information about third-party products, contact individual product vendors.

Code Examples

Some code examples in this guide have been reformatted for presentation purposes and, therefore, may contain hidden editing characters (such as tabs and end-of-line characters) and extraneous spaces. If you cut and paste code from this manual, check for these characters and remove them before attempting to use the example in your application.

1

Fundamentals

This chapter describes the basic concepts and installation and configuration of Creo View Java Toolkit.

Topic	Page
Introduction to Creo View Toolkit	1 - 2
Installation Requirements	1 - 2
Installing Creo View Toolkit	1 - 3
Configuration of Creo View Java Toolkit	1 - 5
About the User Interface	1 - 5

Introduction to Creo View Toolkit

Creo View Toolkits enable you to embed and control Creo View technology within Web pages and other applications to share product data with customers through interactive portals.

You can use Creo View Toolkit to:

- Embed Creo View tools and data in a custom in-house application.
- Use Creo View tools and data in an extended enterprise Web portal.
- Embed Creo View tools in a custom environment and distribute that application.
- Provide custom integrations of Creo View into Web and Java applications developed by a software services company.

The types of customizations available for Creo View are:

- Creo View Web Toolkit—JavaScript (Web page) API
- Creo View Java Toolkit—Java (embedding in a Java application) API
- Creo View Office Toolkit—Visual Basic APIs

Installation Requirements

This section describes the prerequisites and system requirements for installing Creo View Java Toolkit.

For information on platform support support, refer to the Creo View Toolkit Software matrix at http://www.ptc.com/appserver/cs/doc/refdoc.jsp .

Prerequisites for Installation

Before you install Creo View Java Toolkit:

1. Install JDK 1.6.0.0.
2. Install Java Runtime Environment (JRE) version 1.6.0.0.
3. Set the `JAVA_HOME` environment variable to point to the location of JDK 1.6.0.0.

System Requirements

A software matrix on the PTC Web site lists the combinations of platforms, operating systems, and third-party products that are certified for use with Creo View Toolkit on Windows. To obtain a copy of the latest software matrix, go to:

http://www.ptc.com/appserver/cs/doc/refdoc.jsp

You are directed to the PTC Online Support Web page for reference documents. For your document search criteria, select your product from the Product list, select the current release from the Release list, and select Software Matrices from the Document Type list.

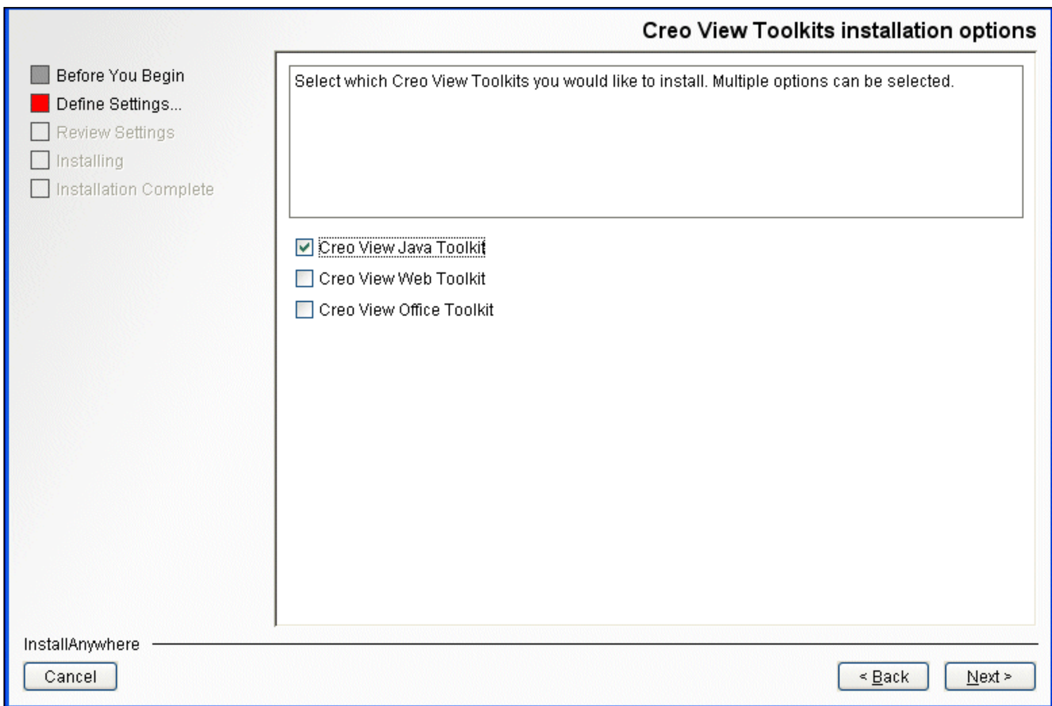
Installing Creo View Toolkit

Remember the following points during the installation:

- Click **Previous** at anytime during the installation process to revise the information that you have provided.
- Click **Cancel** at anytime to stop the installation. You are prompted for confirmation.

Perform the installation as follows:

1. Insert the Creo View Toolkit CD-ROM. If autorun is enabled for your CD-ROM drive, the *setup.vbs* file starts automatically. Otherwise, start Windows Explorer, browse to the CD-ROM drive, and double-click the icon for *setup.vbs*. The **Select Language** dialog box opens.
2. Select the required language and click **OK**. The **Before You Begin** window appears.
3. Review the information and click **Next**. The **PTC Customer License Agreement** window appears.
4. Click **I Accept the License Agreement Terms and Conditions** to proceed with the installation.
5. Click **Next**. The **Select Directory** window appears.
6. Click **Browse** to specify the location for the installation. You are prompted for confirmation if you want to create a new directory.
7. Click **Yes**. The **Creo View Toolkit installation options** window appears.



8. Select the Creo View Toolkit to be installed:
   - Creo View Java Toolkit—Java APIs for Creo View
   - Creo View Web Toolkit—Web APIs for Creo View
   - Creo View Office Toolkit—Visual Basic APIs for Creo View

   Note: You should install only the Toolkit that you have purchased.
9. Click **Next**. The **Review Settings** window appears.
10. Click **Install** to start the installation process. When complete, the **Installation Complete** window appears.
11. Click **Done**.

After the installation is complete, the following directories are created in the installation folder:

- redist
- java
- documentation
- installer
- demodata

Configuration of Creo View Java Toolkit

After installing Creo View Java Toolkit, install the Creo View client as follows:

1. Browse to <creo_view_api_loadpoint>/redist, where, creo_view_api_loadpoint is the location where you have installed Creo View Java Toolkit. The following files are available:
   - CreoView_32.exe—Creo View client installer for a 32-bit platform.
   - CreoView_64.exe—Creo View client installer for a 64-bit platform.
2. Run CreoView_32.exe to install the latest version of the Creo View client available with Creo View Java Toolkit. Run CreoView_64.exe on a 64-bit platform.

   Note: Creo View Java Toolkit is not supported with the Creo View Express client.

About the User Interface

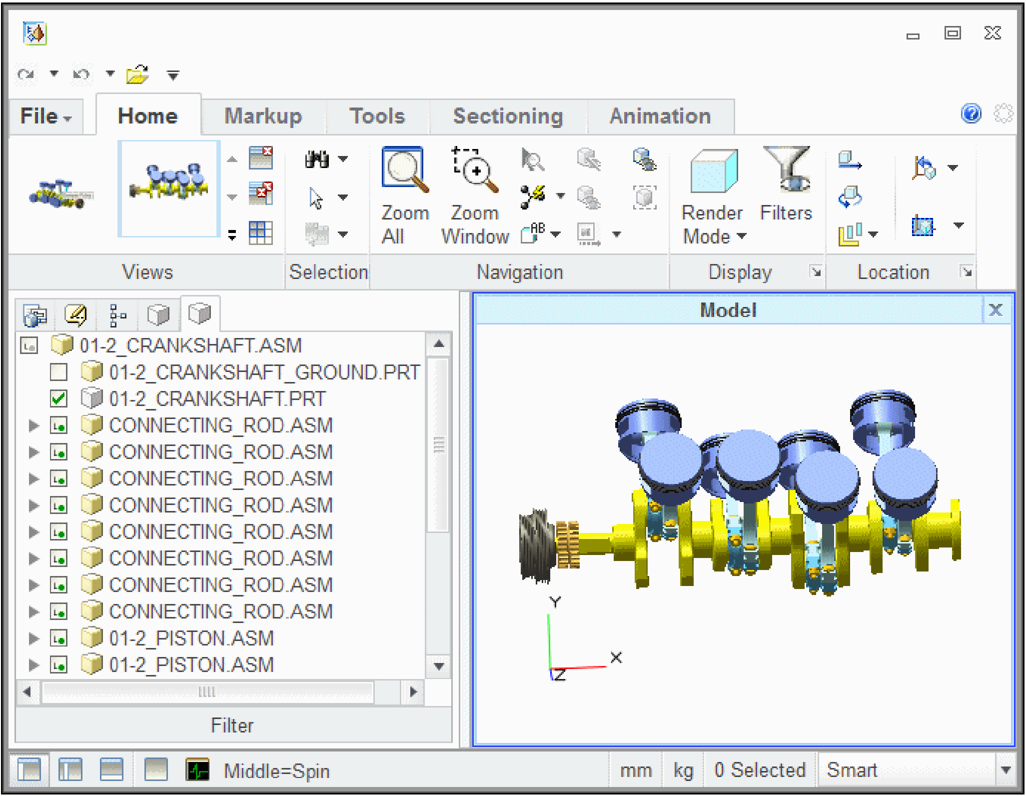
You can launch the Creo View client using the Creo View Toolkit applications in:

- GUI mode—Graphics User Interface mode
- Non-GUI mode—Non-graphics User Interface mode where the Graphical User Interface for the Creo View client is not shown.

This section describes these modes in detail. It also describes how you can customize the user interface using the API applications.

GUI Mode

When the application starts Creo View in this mode, the Viewing or Graphics area is displayed along with the Creo View User Interface (UI) as shown in the following figure.



The UI consists of:

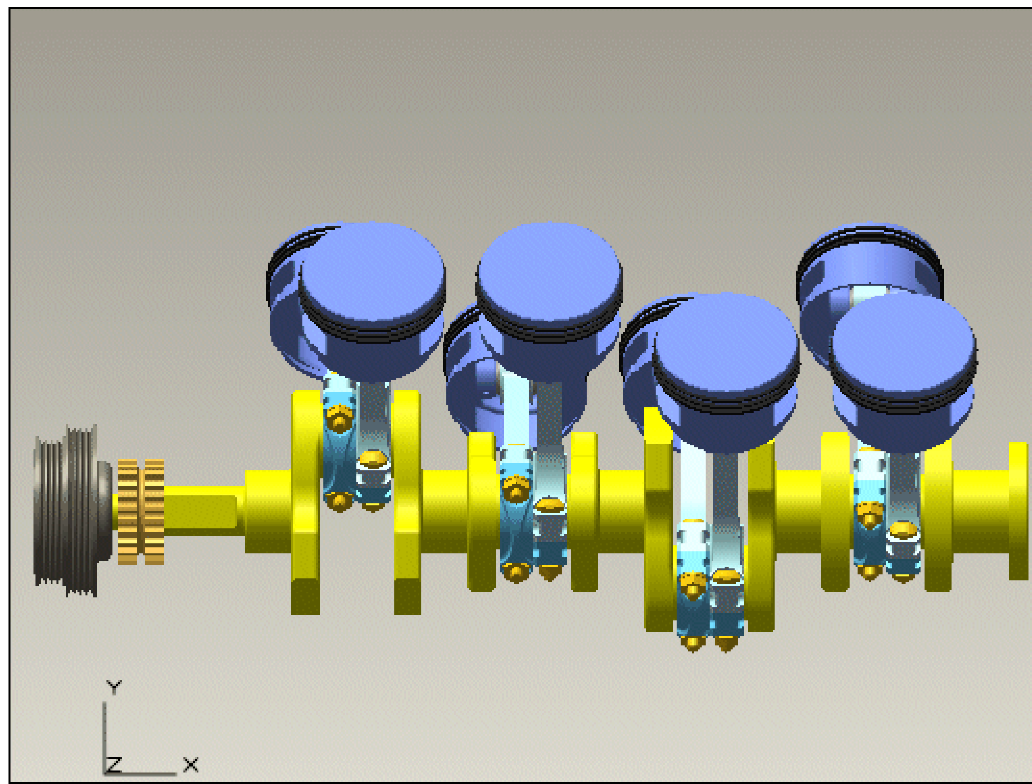
- **Top-level Cascading Menu**—Contains basic commands for using Creo View.
- **Ribbons**—Contain command groups.
- **Panels**—Display information about the product structure, such as files and annotations, as well as attributes.
- **Viewing Area**—The window where 3D models, drawings, and other files are displayed.
- **Status Bar**—Displays information about the current view, along with selection and units settings.

For more information on the Creo View user interface, refer to the online help available with your Creo View installation.

In the GUI mode, you can leverage the complete functionality that the UI offers and manipulate the loaded data beyond the control of your Creo View Toolkit application.

Non-GUI Mode

When the application starts Creo View in this mode, only the Viewing or Graphics area is displayed as shown in the following figure and the Creo View UI is not available.



In this mode, you can control the data displayed in the Viewing area only through the Creo View Toolkit method calls. The methods that change the properties of the data in the Viewing area for a particular session or instance are available only in the non-GUI mode, for example, methods that manipulate:

- Background color of the Viewing area
- Properties of specific instances, such as color, transparency, and so on

Navigating Creo View

The Navigation tools that let you switch between different navigation modes are available in both GUI and non-GUI modes. You can use a combination of mouse and keyboard controls to switch between zoom, pan, and spin operations.

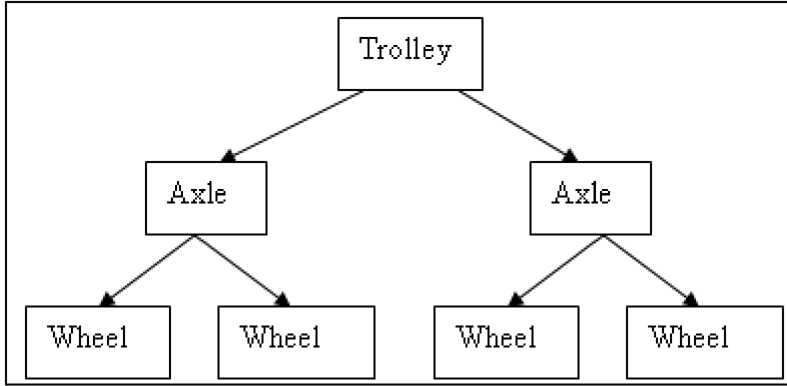
Note: For more information on the navigation controls, refer to the Creo View online help.

Overview of Creo View Data Structures

At the top level, the Creo View kernel contains a **World**. This is created and populated when you load a `.pvs` or `.ed` file. A world contains a **Structure**. This structure represents the hierarchy of parts (also known as components) in that world. For example, consider a structure made up of 3 parts:

- Trolley
- Axle
- Wheel

These can be pulled into an assembly referred to as a **Tree**, and which represents the real trolley with 2 axles and 4 wheels.

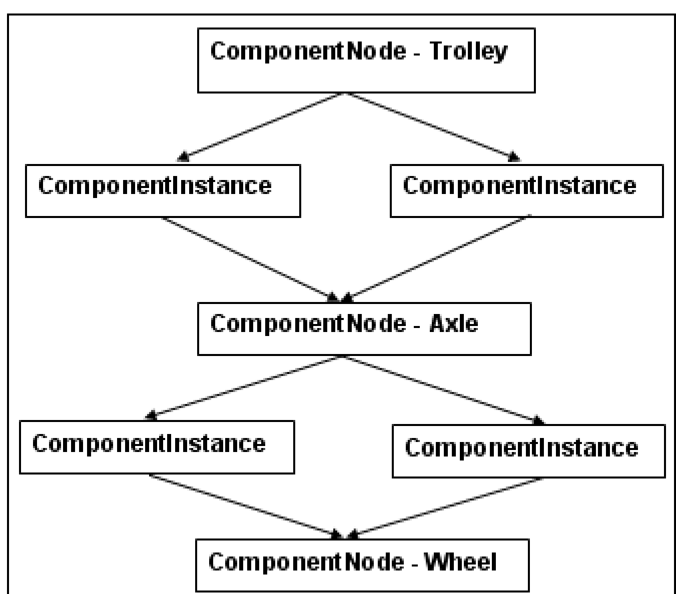


However, the 2 axles are identical to each other as are the wheels except for their locations. As each wheel may have a lot of shared information, we ensure that this information is stored in memory and build this assembly up from components. The component acts as a storage vessel in a file structure. It is used to hold properties which store or reference the actual data or other subcomponents.

Components are of several types:

- **ComponentNode—**Data items that contain all the shared information. These need to be referenced by a ComponentInstance before they are created in the tree. To create children of these nodes the ComponentProxy or ComponentInstances should be added as children.
- **ComponentProxy—**Used to reference children that are contained in a seperate `.pvs` file.
- **ComponentInstance—**References a ComponentNode or ComponentProxy and creates an instance of that ComponentNode in the Tree. Properties of the ComponentNode, such as location or color, can be overridden in a ComponentInstance.

A ComponentInstance creates an instance of a ComponentNode or ComponentProxy when it is instantiated in the Tree. Therefore, the assembly above can be represented as follows:



Each ComponentNode contains the same information for all instances of that component, that is, the properties and files names of secondary content such as 3D shapes, drawings, images, and so on. This hierarchy is grouped under a structure. A structure can be associated with a `.pvs` file and represents the contents of that file.

To realize the fully expanded instance structure, the structure is expanded to create a Tree that contains instances as follows:

Methods are provided on the instance to access information. However, these methods retrieve the relevant information from the ComponentInstance or ComponentNode. The information is therefore, stored only where necessary. However, information may be overridden at each level, for example, each wheel could be given a different name or have a unique property. Information which is only relevant at the Instance level such as the 3D position of the instance is stored at that level.

When you want to look at some data in the World in the view window, you must first create a scene. Different types of scenes are as follows:

- **ShapeScene**—A shape scene has a list of parts that are loaded and visible in it. It also has a selection list.
- **DrawingScene**—A drawing scene
- **ImageScene—**An image scene
- **DocumentScene—**A document scene

Having created a scene, you cannot see it. To be able to see any data you need a **View**. A view adds view point information to the scene, for example, the orientation for the parts in a scene. One scene can have multiple views. As views share the scene, if you unload a part in the scene, all the views would see that part disappear.

The different types of views are:

- **ShapeView—**Specifies a shape view.
- **DrawingView—**Specifies a drawing view.
- **ImageView—**Specifies an image view.
- **DocumentView—**Specifies a document view.

2

Creo View Java Toolkit

Creo View Java Toolkit provides a Java programming interface to Creo View, allowing you to interact with the Creo View client within your own Java applications.

Topic	Page
Overview of the Java APIs	2 - 3
Creating Applications Using Creo View Java Toolkit	2 - 7
World Object	2 - 10
Product Structure	2 - 13
Component Operations	2 - 21
View Operations	2 - 25
View Orientations	2 - 29
Screen Capture	2 - 30
Selection Operations	2 - 31
Accessing Combined View States	2 - 34
Annotations	2 - 37
Product Manufacturing Information (PMI) for Markup Objects	2 - 38
Circle Operations	2 - 41
Point Operations	2 - 42
Lines of a Drawing Leader	2 - 43
Geometrical Data	2 - 37

Manipulation Modes	2 - 48
Accessing Viewable Files	2 - 48
Accessing Properties	2 - 50
Layer Operations	2 - 51
Uploading and Downloading Files from a Webserver	2 - 51

Overview of the Java APIs

Class Types

The Creo View Java Toolkit is made up of a number of classes in many packages.

The main class types are:

- **Creo View-related Interfaces**—Contain unique methods and attributes that are directly related to the function in Creo View.
- **Observer and ActionListener Classes**—Enable you to specify callbacks that are executed only if certain events in Creo View occur. These classes are:
  - TreeObserver
  - ShapeSceneObserver
  - ViewObserver
  - SelectionObserver
- **Utility Classes**—Contain static methods used to initialize certain Creo View Java Toolkit objects. These classes are:
  - DPoint3D
  - DVec3
  - FBox
  - FMat33

Creo View-related Interfaces

Initialization

Use the **Get** or **Create** APIs or the constructors to initialize an object of this class.

Attributes

Attributes within Creo View-related objects are not directly accessible, but can be accessed through **Get** and **Set** methods. These methods are of the following types:

```
Attribute name: int XYZ
Methods:    int GetXYZ();
            void SetXYZ (int i);
```

Some attributes that have been designated as read only can be accessed only by the **Get** method.

Methods

You must first initialize the Creo View-related object. For example,

```
kernel = myActor.getKernel();
//Get EmbeddedControl from Kernel
embeddedControl = kernel.GetEmbeddedControl();
//Create World using EmbeddedControl
theWorld = embeddedControl.CreateWorld("pvkernel");
```

Exceptions

Almost every Creo View Java Toolkit method throws one or more of the following exceptions:

- `MessageTimeoutException`
- `ActorShutdownException`
- `InvalidActorException`
- `ConnectionLostException`

Surround the methods you use with a `try-catch` block to handle any exceptions that are generated.

Observers and Action Listeners

Use `Observers` and `ActionListeners` to assign programmed reactions to events that occur within Creo View. Creo View Java Toolkit defines a set of action listener interfaces that can be implemented, enabling Creo View to call your Java application when specific events occur. These interfaces are designed to respond to events from action sources in Creo View. Examples of action sources include the world, tree object, user-interface commands, shapesource, shapescene, view, and structure.

Initialization

You must derive child classes from the Observer classes in order to react to callbacks issued by the Action source.

```
TreeObserver treeObserver = myActor.getTreeObserver();
tree.RegisterObserver(treeObserver.GetObjectId(),
                      treeObserver.GetOwner());

class MyTreeObserver extends TreeObserver
{
    protected void OnBeginUpdate()
    {
    }
        protected void OnEndUpdate()
    {
    }

    protected void OnInstanceCreate(Instance instance, Instance parent,
                                    String name)
    {
    }
    protected void OnInstanceRemove( Instance instance, Instance parent)
    {
    }

    protected void OnInstanceName(Instance instance)
    {
    }

    protected void OnInstanceLocation(Instance instance)
    {
    }

    public String GetObjectClass()
    {
      return "pvapps::javatestapp::MyTreeObserver";
    }
}
```

Attributes

Action listeners or Observers do not have any accessible attributes.

Methods

Action listeners or Observers have callback methods that notify their inheriting class of actions and events appropriate to the class they are observing. For example, the TreeObserver has the following methods:

- `void OnBeginUpdate();`
- `void OnEndUpdate();`
- `void OnInstanceCreate(Instance instance, Instance parent, String name);`
- `void OnInstanceRemove( Instance instance, Instance parent);`
- `void OnInstanceName(Instance instance);`
- `void OnInstanceLocation(Instance instance);`

Exceptions

You must include exception-handling code inside the callbacks if you want to respond to exceptions.

Utilities

These classes represent points, vectors, bounding boxes, and other data types used by Creo View Java Toolkit.

Initialization

Use the following constructors to initialize an object of this class. For example,

```
DPoint3D dp3 = new DPoint3D();
DPoint3D dp3 = new DPoint3D(double a, double b, double c)
DPoint3D dp3 = new DPoint3D(DPoint3d a);
```

Attributes

Attributes for the utility classes are not directly accessible, and must be accessed using **Get** and **Set** methods.

Methods

The methods for the Utility classes are only used for getting and setting attributes. For example,

```
double DPoint3D.Get(int i)
double DPoint3D.Set(int i, double val)
double DPoint3D.Set(DPoint3D point)
```


Creating Applications Using Creo View Java Toolkit

The following sections describe the process of creating applications:

- Importing Packages
- Application Hierarchy

Importing Packages

To use the Creo View code in your application, you must import the necessary packages. Import each class or package with a statement similar to:

```
For the package pvkapp (all classes):

import com.ptc.pview.pvkapp.*;
```

Application Hierarchy

The object-oriented structure of Creo View Java Toolkit requires a certain hierarchy and order of object creation when you start a Creo View Java Toolkit application. To create an application:

1. Include the `pview.jar` file available at `<creo_view_api_loadpoint>\java\jar` in the dependency package of an Integrated Development Environment (IDE) or your classpath.

   Where, `creo_view_api_loadpoint` is the location where you have installed Creo View Java Toolkit.

2. Initialize the `PviewInit` object as follows
   `pview = new PviewInit();`

3. Use `PviewInit.Start` to start the internal communication layer of Creo View.

4. Create your own class which extends the `ManagedObjects` class.
5. Use the method **GetDistinguishedObject** to get an object of type `kernel`. For example,

```
public Kernel getKernel()
{
    try
    {
      return (Kernel)GetDistinguishedObject(Kernel.CLASS_NAME,
              "pvkernel");
    }
    catch (Throwable x)
    {
    System.out.println("Exception getting the
                        GetDistinguishedObject()");
    }
    return null;
}
```

6. Use the kernel object to instantiate the following:
   - `Kernel.EmbeddedControl` object—Provides an interface to Creo View for creating the World object, opening `.pvs` or `.ed` files.
   - `EmbeddedControl.World` object—Tree which is the real path to create an application accessing 3D contents.
   - `Kernel.Window` object—Used to associate a panel in the Java GUI to the Creo View window.
   - `Kernel.RemoteIf` object—This is to notify Creo View of the status of requested uploads or downloads of files.
7. You can use these objects and the Creo View Java Toolkit methods to create your Java application.
8. Use `PviewInit.GetClientVersion` to get the version of Creo View installed on your machine.
9. Close the application using `PviewInit.Stop()`.

Use the sample applications provided with your installation as a template to create your application. Refer to the chapter, “Sample Applications” for more information on the sample applications provided with your installation.

For more information on the application hierarchy, refer to the file, `ExamplesUtilities.java`, available with your installation at `<creo_view_api_loadpoint>\java\src`.

Example: Application Hierarchy

```
synchronized public boolean runPView()
{
    if(pview.IsPviewInstalled() == false)
        return false;
    if(pview.Start("webserver") == false)
        return false;

    try
    {
        myActor = new MyActor();
        kernel = myActor.getKernel();

        theWindow = kernel.GetWindow();
        embeddedControl = kernel.GetEmbeddedControl();
        PVWindow pvWindow = new PVWindow(theWindow, panel);

        theWorld = embeddedControl.CreateWorld("pvkernel");
        theWorld.SetControlActor("pview");

        addWindowListener( new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                pviewShutdown();
                System.exit(0);
            }
        });

        structure = theWorld.CreateStructure();
        if (structure != null)
        {
            tree = theWorld.CreateTree();
            TreeObserver treeObserver = myActor.getTreeObserver();
            tree.RegisterObserver(treeObserver.GetObjectId(),
                                  treeObserver.GetOwner());
            theWorld.SetParentWindow(pvWindow.GetWindow());
            embeddedControl.SetAutoLoad("auto");
            AsyncEventCB initEvent = myActor.getAsyncEvent("Initialise");
            embeddedControl.Initialise(initEvent.GetAsyncEventIf());
            myActor.listenForEvents();
        }
        else
            return false;
       }
       }
       catch (Throwable x)
       {
         x.printStackTrace();
       }
```

Creo View Java Toolkit

```
      return true;
}

public void pviewShutdown()
{
  pview.Stop();
}
```

Deployment

The Java applications are distributed as compiled `.class` or `.jar` files.

World Object

The Creo View `World` object (contained in the class `com.ptc.pview.pvkapp.World`) is the highest-level object in Creo View Java Toolkit. Any program that accesses data from Creo View must first obtain a handle to this session object before accessing more specific data.

The `World` object contains methods to perform the following operations:

- Access Creo View viewable and structure files.
- Interactive selection of items within the graphics windows.

Methods Introduced:

- **EmbeddedControl.Initialise**
- **EmbeddedControl.CreateWorld**
- **EmbeddedControl.GetEmbedWindow**
- **EmbeddedControl.URLOpen**
- **World.SetControlActor**
- **World.SetParentWindow**
- **World.RemoveContent**

The method **EmbeddedControl.Initialise** initializes Creo View.

The method **EmbeddedControl.CreateWorld** creates a Creo View World with the specified name. Pass the value of the input parameter *worldName* as `pvkernel`.

The method **EmbeddedControl.GetEmbedWindow** returns the handle to the top level window used by Creo View.

The method **EmbeddedControl.URLOpen** loads the specified viewable (`.ol`) file or the specified structure (`.pvs` or `.ed`) file. The input arguments are:

- *sourceUrl*—Specifies the path or URL of the file to be loaded. The supported file types are:
  - Viewable (`.ol`)—Files that represent the 3D model graphics of the components of an assembly.
  - Structure (`.ed`)—Pre-ProductView 9.0 files that contain product structure, component position, orientation, and metadata (part- and assembly-level parameter) information.
  - Structure (`.pvs`)—ProductView 9.0 structure files that contain product structure, component position, and orientation information.
  - Structure Package (`.pvs`/`.edz`)—Compressed version of the structure files.
  - Creo Parametric files. For more information refer to the section "Support for Creo Parametric Files."
  - Image or Drawing file —Image files of format (`.gif`, `.jpg`, `.gif`, `.bmp`) and drawing files of format (`.dwg`, `.dxf`, `.plt`).
  - Portable Document Format (PDF) files—Adobe Acrobat PDF files.
- *markupUrl*—Specifies the URL to the annotation `.etb` file. A `.etb` file lists the annotation sets for a `.pvs` or `.ed` file. It is modified every time an annotation set is created, renamed, or modified. Pass the URL to the `.etb` file when running Creo View against a Web server.
- *modifyMarkupUrl*—Specifies the URL where files will be posted when running Creo View against a Web server, for example, `http://localhost/web/file_upload.jsp?path=F:/tomcat/webapps/ROOT/web/demodata/Crank/&`. The URL can point to a `.jsp`, `.php`, or `.asp` file that will handle the requests to a Web server.

The method **World.SetControlActor** specifies whether the Java application launches Creo View in the GUI or non-GUI mode. The valid values of the input parameter *controlType* are:

- `thumbnail`—Specifies that Creo View is launched in the non-GUI mode.

- `pview`—Specifies that Creo View is launched in the GUI mode. This option displays the tabs that contain data related to the assembly or component. These tabs are:
  - **Files—**Displays the list of files referenced in the loaded assembly.
  - **Annotation Sets—**Displays the annotation sets and groups referenced in the loaded assembly

The method **World.SetParentWindow** sets the parent window to embedd Creo View.

The method **World.RemoveContent** removes the loaded data from the Creo View world. This enables you to load new data without restarting Creo View.

Support for Creo Parametric Files

Creo View Java Toolkit can load the following Creo Parametric format files using the method **EmbeddedControl.URLOpen**:

Creo Parametric File	Type
Part	`.prt`
Assembly	`.asm`
Drawing	`.drw`
Format	`.frm`
Layout	`.lay`
Diagram	`.dgm`
Report	`.rep`
Section	`.sec`

Note: You can load Creo Parametric assembly files (`.asm`) if they reside on the local file system. Loading of assembly files over the https:// or http:// protocol is not supported.

To be able to open a Creo Parametric file in Creo View, set the following configuration options when you save them in Creo Parametric:

- save_model_display—Specify one of the following values:
  - `shaded_lod`
  - `shaded_low`
  - `shaded_high`

- save_drawing_picture_file —Specify as `both`.
- sketcher_save_preview_image —Specify as `yes`.

The Creo Parametric format (`.frm`), layout (`.lay`), diagram (`.dgm`), report (`.rep`), and section (`.sec`) files are treated similar to drawing files in Creo View. You can navigate through the sheets of the Creo Parametric drawing file (`.drw`) using the **Page Up** and **Page Down** keys on the keyboard.

Product Structure

The product structure displays a hierarchical view of the contents of the `.pvs` file. Each node in the product structure tree represents a component or subassembly in the structure.

This section describes the methods provided to create a product structure at runtime. It also describes the methods provided to load a product structure in Creo View.

Creating the Product Structure

You must perform the following steps to create a product structure:

- Create a structure that holds the hierarchy together.
- Add a root node to the structure.
- Create a component. The component acts as a storage vessel or a directory in a file structure. It is used to hold properties, which store or reference the actual data, or other subcomponents. Components can be contained within assemblies.
- Add subcomponents to a root or parent component which returns a `ComponentInstance`.
- Place the component instance at the desired location.

Refer to the section, “Overview of Creo View Data Structures” in the chapter “Fundamentals”, while working with the methods described in this section.

Methods Introduced:

- **World.CreateStructure**
- **Structure.GetRoot**
- **Structure.SetRoot**
- **Structure.CreateComponentNode**
- **ComponentNode.SetShapeSource**
- **ComponentNode.AddComponentNode**
- **ComponentNode.AddComponent**
- **ComponentInstance.SetLocation**
- **ComponentInstance.GetLocation**

The method **World.CreateStructure** provides the ability to create a product structure at runtime.

The method **Structure.GetRoot** returns the root node of the structure tree. Use the method **Structure.SetRoot** to set the root node.

Note: You can set the root node only once for a product structure and once set, it cannot be changed.

The method **Structure.CreateComponentNode** enables you to create a new component node in the product structure. Specify the name of the node to be created as the input parameter for this method. The input parameter *type* is not currently used; pass `0` as the value to this parameter.

The method **ComponentNode.SetShapeSource** enables you to set the shapesource. A shapesource specifies a source CAD file for a component to be viewed with the 3D Model viewer. The input parameters for this method are:

- *fileSource*—Specifies the URL of the component (`.ol` file).
- *xMin*, *yMin*, *zMin*, *xMax*, *yMax*, *zMax*—Specifies the location of a shapesource in 3D space.

Use the method **ComponentNode.AddComponentNode** to add a new component node to the product structure tree. This method returns a component instance.

ComponentNode.AddComponent adds a new component to the component node. This method returns the component instance of the node.

The method **ComponentInstance.SetLocation** enables you to specify the orientation and rotation of the child component relative to the coordinate system.

The method **ComponentInstance.GetLocation** returns the location of the component instance in the structure.

Sample Code

The following sample code shows you the use of the methods described above. For more information, refer to the chapter, "Sample Applications".

```
ComponentNode cn1, cn2, cn3, cn4;
ComponentInstance ci1, ci2, ci3;

private boolean  PopulateStructure(Structure s)
{
   try
   {
      cn1 = s.CreateComponentNode("Wheel Assembly", (byte)'a');
      System.out.println("TestActor.OnInitialise(): "
            +"Lets create the 1st componentNode....");

      if(cn1 == null)
      {
         System.out.println("TestActor.OnInitialise(): "
               +"Could not get a ComponentNode interface.");
         pviewShutdown();
         return false;
      }

      s.SetRoot(cn1);

      cn1 = s.GetRoot();

      System.out.println("TestActor.OnInitialise(): "
            +"Lets create a 2nd componentNode....");

      cn2 = s.CreateComponentNode("Wheel 1", (byte)'a');
      if(cn2 == null)
      {
         System.out.println("\n TestActor.OnInitialise(): "
               +"Could not get a second ComponentNode interface.");
         pviewShutdown();
         return false;
      }

      cn2.SetShapeSource(PART_ROOT+"/wheels_smaller.ol", 0, 0, 0, 1, 1,
                                                                  1);
```

Creo View Java Toolkit

```
System.out.println("TestActor.OnInitialise(): "
      +"Lets create a 3rd componentNode....");

cn3 = s.CreateComponentNode("Axle", (byte)'a');
if(cn3 == null)
{
   System.out.println("TestActor.OnInitialise(): "
         +"Could not get a third ComponentNode interface.");
   pviewShutdown();
   return false;
}
cn3.SetShapeSource(PART_ROOT+"/wheels_smaller_2.ol", 0, 0, 0, 1,
                                                           1, 1);

System.out.println("TestActor.OnInitialise(): "
      +"Lets create a 4th componentNode....");

cn4 = s.CreateComponentNode("Wheel 2", (byte)'a');
if(cn4 == null)
{
   System.out.println("TestActor.OnInitialise(): "
         +"Could not get a fourth ComponentNode interface.");
   pviewShutdown();
   return false;
}
cn4.SetShapeSource(PART_ROOT+"/wheels_smaller_3.ol", 0, 0, 0, 1,
                                                           1, 1);
cn4.AddProperty("testCnProperty", "Properties", "110");


ci1 = cn1.AddComponentNode(cn2, "cn2");
if(ci1 == null)
{
   System.out.println("TestActor.OnInitialise(): "
         +"Could not get add component node as a child.");
   pviewShutdown();
   return false;
}
FMat33 w1FMat = new FMat33(1.0f,0.0f,0.0f,0.0f,1.0f,0.0f,0.0f,
                                                     0.0f,1.0f);
DPoint3D w1DPoint = new DPoint3D(0, 0, 0.01);
Location w1Location = new Location(w1FMat, w1DPoint);

ci1.SetName("Wheel 1");

ci2 = cn1.AddComponentNode(cn3, "Cn3");
if(ci2 == null)
{
   System.out.println("TestActor.OnInitialise(): "
         +"Could not get add component node as a child.");
```

```
         pviewShutdown();
         return false;
      }

      FMat33 axFMat = new FMat33(1.0f,0.0f,0.0f,0.0f,1.0f,0.0f,0.0f,
                                                     0.0f,1.0f);
      DPoint3D axDPoint = new DPoint3D(0, 0, -0.0115);
      Location axLocation = new Location(axFMat, axDPoint);

      ci2.SetLocation(axLocation);
      ci2.SetName("Axle");
      ci2.AddProperty("testCiProperty1", "Properties", "10");
      ci2.AddProperty("testCiProperty2", "PROE Parameters", "120");

      ci3 = cn1.AddComponentNode(cn4, "Cn4");
      if(ci3 == null)
      {
         System.out.println("TestActor.OnInitialise(): "
               +"Could not get add component node as a child.");
         pviewShutdown();
         return false;
      }

      FMat33 w2FMat = new FMat33(1.0f,0.0f,0.0f,0.0f,1.0f,0.0f,0.0f,
                                                    0.0f,-1.0f);
      DPoint3D w2DPoint = new DPoint3D(0, 0, -0.0230);
      Location w2Location = new Location(w2FMat, w2DPoint);



      ci3.SetLocation (w2Location);
      ci3.SetName("Wheel 2");
      ci3.AddProperty("testCiProperty1", "Properties", "20");
      ci3.AddProperty("testCiProperty2", "PROE Parameters", "120");

   }
   catch (java.lang.Exception e)
   {
      System.out.println("Got Exception " + e);
      e.printStackTrace();
      return false;
   }
   return true;
}
```

Visiting the Product Structure

The methods described in this section enable you to traverse the instances in the Creo View tree. These methods are provided on the instance to access the information relevant to a specified instance in the structure tree.

Methods Introduced:

- **Instance.Visit**
- **Instance.GetParent**
- **Instance.GetFirstChild**
- **Instance.GetNextSibling**
- **Instance.GetPrevSibling**
- **Instance.GetComponentInstance**
- **Tree.GetNumInstances**
- **Structure.GetNumComponents**
- **Instance.AddUserData**
- **Instance.GetUserData**
- **Instance.RemoveUserData**

The method **Instance.Visit** visits the instance and its siblings in the product structure. This visit method has the following input parameters:

- *visitor*—An instance visitor of type `com.ptc.pview.pvkapp.InstanceVisitor`.
- *type*—Specify 0 to visit instances by depth or 1 to visit the instances by breadth.

The following sample code shows you how to create an instance visitor.

```
class MyInstanceVisitorEvents
extends  com.ptc.pview.pvkapp.InstanceVisitorEvents
{

   ArrayList list = new ArrayList();

   public MyInstanceVisitorEvents()
   {
      super();
   }
```

```
   public boolean Visit(Instance inst, Instance parent, int depth)
   {
      list.add(inst);
      return true;
   }
}

   public InstanceVisitor getInstanceVisitor(InstanceVisitorEvents ive)
   {

      InstanceVisitor_impl iv = new InstanceVisitor_impl();
      iv.SetEventHandler(ive);
      ManageObject(iv);
      return iv;
   }

MyInstanceVisitorEvents ive = new MyInstanceVisitorEvents ();
InstanceVisitor instVisitor = myActor.getInstanceVisitor(ive);
tree.Visit(instVisitor, 0);
```


The method **Instance.GetParent** returns the parent of the specified instance in the product structure tree.

The method **Instance.GetFirstChild** returns the first child instance of the specified instance.

The method **Instance.GetNextSibling** returns the next sibling of the specified instance in the assembly, while the method **Instance.GetPrevSibling** returns the previous sibling. The methods return null if no siblings exist.

The method **Instance.GetComponentInstance** returns the component instance from which the specified instance is instantiated in the structure tree.

The method **Tree.GetNumInstances** returns the number of instances used in the specified assembly, while the method **Structure.GetNumComponents** returns the number of components referencing these instances.

The method **Instance.AddUserData** allows you to add user data to the instance by ID. The input parameters for this method are:

- *id*—Specifies the ID of the user data.
- *data*—Specifies the reference data for the user object.

The method **Instance.GetUserData** returns the user data specified by the user ID. Specify the ID of the user object as the input parameter for this method. This ID must be generated by the user.

The method **Instance.RemoveUserData** removes the user data associated with the specified ID.

Modifying the Product Structure

Methods Introduced:

- **ComponentNode.RemoveChild**
- **Structure.RemoveComponent**
- **Structure.RemoveAllComponents**
- **Structure.RemoveComponentNode**

Use the method **ComponentNode.RemoveChild** to delete the specified component instance from the component node. The method returns true if the component is removed.

The method **Structure.RemoveComponent** removes a specified component from the structure, while the method **Structure.RemoveAllComponents** removes all the components from the structure tree. The method **Structure.RemoveComponentNode** removes the specified component node from the tree.

These **Remove*** methods only modify the structure tree. However, the ShapeSource attached to the components is still visible in the user interface. To remove the components or nodes completely, use the `TreeObserver` class to detect the components being removed from the tree and use the callback **TreeObserver.OnInstanceRemove** to remove the instances from the ShapeView.

Refer to the example "`BuildAssemblyExample.java`" provided with the Creo View Java Toolkit installation for sample code using the methods described in this section.

Component Operations

Methods Introduced:

- **ShapeScene.GetShapeInstance**
- **ShapeScene.RemoveShapeInstance**
- **ShapeScene.RemoveAllShapeInstances**
- **ShapeInstance.SetVisibility**
- **ShapeScene.AllSetVisibility**
- **ShapeScene.IsolateSelected**
- **ShapeScene.SelectedSetVisibility**
- **ShapeScene.ShowOnlySelected**
- **ShapeScene.ShowAll**
- **ShapeInstance.GetLocation**
- **ShapeInstance.SetLocation**
- **ShapeInstance.RestoreLocation**
- **ShapeScene.RestoreAllLocations**
- **ShapeInstance.IsHighlighted**
- **ShapeInstance.SetHighlight**
- **ShapeInstance.GetColor**
- **ShapeInstance.SetColor**
- **ShapeInstance.GetFirstLayer**
- **ShapeInstance.GetNextLayer**
- **ShapeInstance.HasTransparencyOverride**
- **ShapeInstance.GetTransparencyOverride**
- **ShapeInstance.OverrideTransparency**
- **ShapeInstance.RestoreTransparency**

After you load a `.pvs` structure using the method **EmbeddedControl.URLOpen**, you can hide or unload a specified component.

The method **ShapeScene.GetShapeInstance** provides the shape instance in the scene that corresponds to the specified instance. The following sample code shows you how to access the handle to the specified shape instance.

```
scene = theWorld.GetFirstShapeScene();
ShapeInstance_holder shapeInstanceHolder = new
ShapeInstance_holder();
Instance inst = ci1.GetInstance();
scene.GetShapeInstance(inst, shapeInstanceHolder);

if(shapeInstanceHolder != null)
{
       boolean vis = shapeInstanceHolder.value.IsVisible();
       shapeInstanceHolder.value.SetVisibility(false);
       vis = shapeInstanceHolder.value.IsVisible();
}
```

Create an instance visitor of type `com.ptc.pview.pvkapp.InstanceVisitor`. Call the instance using the method **Instance.Visit**. There will be a callback on the `InstanceVisitor` for every instance and its children. For every callback, check if the instance has an associated `ShapeInstance`. If it has a `ShapeInstance` then use the method **RemoveShapeInstance** with that instance to unload the component from the 3D Viewer. Refer to the following code sample for more details.

```
class MyInstanceVisitorEvents
extends  com.ptc.pview.pvkapp.InstanceVisitorEvents
{

   ArrayList list = new ArrayList();

   public MyInstanceVisitorEvents()
   {
      super();

   }

   public boolean Visit(Instance inst, Instance parent, int depth)
   {
      list.add(inst);
      return true;
   }
}

   public InstanceVisitor getInstanceVisitor(InstanceVisitorEvents ive)
   {

      InstanceVisitor_impl iv = new InstanceVisitor_impl();
      iv.SetEventHandler(ive);
      ManageObject(iv);
      return iv;
   }
```

```
MyInstanceVisitorEvents ive = new MyInstanceVisitorEvents ();
InstanceVisitor instVisitor = myActor.getInstanceVisitor(ive);
tree.Visit(instVisitor, 0);
```

Use the method **ShapeInstance.RemoveAllShapeInstances** to unload all the components from the 3D viewer.

Use the method **ShapeInstance.SetVisibility** to show or hide a specified component in the 3D viewer. Use the method **ShapeScene.AllSetVisibility** to show or hide all the components of the product structure from the 3D view.

The method **ShapeScene.IsolateSelected** shows only the selected instances in the view and hides all the other components.

The method **ShapeScene.ShowOnlySelected** makes the specified instances visible in the 3D view. To show all the instances in the 3D view, use the method **ShapeScene.ShowAll**.

The method **ShapeInstance.GetLocation** returns the location of the specified component in xyz coordinates. Use the method **ShapeInstance.SetLocation** to set the location of the specified component instance.

The method **ShapeInstance.RestoreLocation** restores the specified instance to its original location. Use the method **ShapeScene.RestoreAllLocations** to restore all instances in the product structure to their original locations.

The method **ShapeInstance.IsHighlighted** returns whether the selected components appear in a highlight color, surrounded by a bounding box. Use the method **ShapeInstance.SetHighlight** to set the highlight for the specified instances.

The method **ShapeInstance.GetColor** returns the color of the specified component instance. Use the method **ShapeInstance.SetColor** to set the color.

The method **ShapeInstance.GetFirstLayer** returns the first layer associated with the specified component instance.

The method **ShapeInstance.GetNextLayer** returns the next layer associated with the specified component instance.

Use the method **ShapeInstance.OverrideTransparency** to specify a value for the transparency of the shape instance. Specify a value betwen 0 to 1, where 0 specifies transparent and 1 specifies opaque. This method overrides the default transparency value.

The method **ShapeInstance.HasTransparencyOverride** determines if the default transparency value has been overriden. If the transparency has been overridden, use the method **ShapeInstance.GetTransparencyOverride** to access the transparency value that has been set. Use the method **ShapeInstance.RestoreTransparency** to restore the default transparency value of the instance.

The following sample code shows the use of some of the methods described in this section:

```
private void addInstanceAndChildren (ShapeScene scene, Instance i,
              ArrayList list) throws java.lang.Exception
{
       i = i.GetFirstChild();
       while (i != null)
       {
              ShapeInstance si = scene.CreateShapeInstance(i);
              si.SetVisibility(false);
              if (si.IsVisible())
              {
                     list.add(i);
                     addInstanceAndChildren (scene, i, list);
              }
              i = i.GetNextSibling();
       }
}
```

View Operations

The methods described in this section enable you to specify the preferences for the 3D view.

Methods Introduced:

- **ShapeView.GetMode**
- **ShapeView.SetMode**
- **ShapeView.SetProjection**
- **ShapeView.IsPerspective**
- **ShapeView.GetOrthographicProjection**
- **ShapeView.SetOrthographicProjection**
- **ShapeView.GetPerspectiveProjection**
- **ShapeView.SetPerspectiveProjection**
- **ShapeView.SetProjectionLocking**
- **ShapeView.GetLocation**
- **ShapeView.SetLocation**
- **ShapeView.SetMinNearMaxFarClip**
- **ShapeView.ZoomAll**
- **ShapeView.ZoomSelected**
- **ShapeView.SetRenderMode**
- **ShapeView.SetBackgroundColor**
- **ShapeView.GetWireframeBackgroundColor**
- **ShapeView.SetWireframeBackgroundColor**
- **ShapeView.SetGradientBackgroundColor**
- **ShapeView.GetTopBackgroundColor**
- **ShapeView.GetBottomBackgroundColor**
- **ShapeView.SaveImage**
- **DrawingView.SaveImage**
- **ImageView.SaveImage**

- **ShapeInstance::OverrideModelAnnotationColor**
- **ShapeInstance::IsModelAnnotationColorOverridden**
- **ShapeInstance::GetModelAnnotationColorOverride**
- **ShapeInstance::RestoreModelAnnotationColor**

The method **ShapeView.SetProjection** enables you to specify either the orthographic or perspective viewing mode and the values to create a camera. The input parameters of this method are:

- *left*, *right, top*, *bottom*, *depth*—Specifies the values for the camera.
- *orthographic*—The value true specifies the orthographic viewing mode, else perspective viewing mode.

The viewing mode determines how the objects will look on screen and is as follows:

- **perspective**—Allows you to perceive depth and distance, as objects would appear in reality.
- **orthographic—**Allows you to view objects without any perspective effects.

The method **ShapeView.IsPerspective** specifies whether the viewing mode is perspective for the current view.

The method **ShapeView.GetOrthographicProjection** returns the width of the view in meters for the orthographic mode. Use the method **ShapeView.SetOrthographicProjection** to set the orthographic width. You can specify this width in meters depending on the size of the loaded model.

The method **ShapeView.GetPerspectiveProjection** returns the value of the Horizontal Field of View (HFOV) which represents the angle of the view area, as if seen through a camera. Use the method **ShapeView.SetPerspectiveProjection** to set the value of the HFOV for the perspective viewing mode. You can calculate this value based on the size of the object and its distance from the viewpoint.

The method **ShapeView.SetProjectionLocking** locks or unlocks the horizontal and vertical projection.

The method **ShapeView.GetLocation** returns the location of the graphics view. It returns a 4X4 matrix of the location. This matrix specifies the X, Y, and Z positions for translation and orientation of the view. Use the method **ShapeView.SetLocation** to set the location of the view.

The method **ShapeView.SetMinNearMaxFarClip** sets the minimum and maximum values for the near and far clip planes, respectively. The clip plane is the cut-off point beyond which objects are not shown because they are either too near, or too far away from the view point.

The method **ShapeView.ZoomAll** adjusts the magnification so that all components are displayed in the view.

The method **ShapeView.ZoomSelected** magnifies the view to show the selected component in more detail.

The method **ShapeView.SetRenderMode** specifies the rendering style of the assembly. The valid values for the input parameter *renderMode* are:

- `Shaded(0)`—Displays the shaded model in the 3D view. This is the default rendering mode.
- `Wireframe(1)`—Displays the model in wireframe mode.
- `HLR(2)`—Displays the model in Hidden Line Removal (HLR) mode.
- `Mesh(4)`—Displays the model in mesh render mode.

The method **ShapeView.SetBackgroundColor** specifies the background color of the view window when shaded rendering is enabled.

The method **ShapeView.GetWireframeBackgroundColor** returns the background color of the view window when wireframe rendering is enabled. Use the method **ShapeView.SetWireframeBackgroundColor** to set the background colour.

The method **ShapeView.SetGradientBackgroundColor** specifies the gradient background color for the shaded view. The gradient color appears at the bottom of the background, while the shaded background color appears at the top, with a gradient progression from one color to the next between the two.

This method is available only in the non-GUI mode. In the GUI mode, the user can change the background color using the available user interface options.

The methods **ShapeView.GetTopBackgroundColor** and **ShapeView.GetBottomBackgroundColor** return the background color appearing at the top and bottom of the view, respectively.

The method **ShapeView.SaveImage** saves the view to a `.bmp` file. You can specify the filename, height, width, and resolution of the image file.

The method **DrawingView.SaveImage** saves the view to an image format file. You can specify the filename, height, width, and resolution of the image file.

The method **ImageView.SaveImage** saves the view to an image format file. You can specify the filename, height, width, and resolution of the image file.

Use the method **ShapeInstance.OverrideModelAnnotationColor** to change the color of the specified annotation. Specify a new color as one of the input arguments of this method. This method overrides the default color.

The method **ShapeInstance.IsModelAnnotationColorOverridden** determines if the default color of the annotation has been overriden.

If the color has been overridden, use the method **ShapeInstance.GetModelAnnotationColorOverride** to access the new value of color for the annotation.

Use the method **ShapeInstance.RestoreModelAnnotationColor** to restore the model annotation to its original color.

View Orientations

A view orientation is the angle at which the structure is displayed in the graphics area in Creo View. The methods described in this section enable you to set and observe orientations that are added, deleted, or modified.

Methods Introduced:

- **Orientations.RegisterObserver**
- **World.GetOrientations**
- **Orientations.SetOrientation**
- **OrientationsObserver.OnBeginUpdate**
- **OrientationsObserver.OnEndUpdate**
- **OrientationsObserver.OnOrientationAdd**
- **OrientationsObserver.OnOrientationDeleted**
- **OrientationsObserver.OnOrientationUpdated**
- **OrientationsObserver.OnDefaultOrientChanged**
- **Orientations.UnregisterObserver**

Use the method **Orientations.RegisterObserver** to register an instance of `com.ptc.pview.pvapi.OrientationsObserver`. Use this instance to receive callbacks on the list of available CAD and user defined orientations.

The method **World.GetOrientations** returns a list of available CAD and user defined orientations.

Use the method **Orientations.SetOrientation** to set an orientation to a specified shapeview. The valid values are:

- ORIENTATION_DEFAULT—Specifies the default orientations available with the Creo View installation.
- ORIENTATION_USER_DEFINED—Specifies a user-defined orientation created through the Creo View UI.
- ORIENTATION_CAD—Specifies the orientation defined in the Creo Parametric model.

The method **Orientations.SetOrientation** method takes a shapeview as its first argument. If you pass null as the value for this argument, the orientation is applied to the current active view.

The following are the orientation callback methods which you override in a class that inherits from `com.ptc.pview.pvapi.OrientationsObserver`.

The method of type **OrientationObserver.OnBeginUpdate** indicates the start of a list of specified orientations.

The method of type **OrientationsObserver.OnOrientationAdd** is called when a new view orientation is added to the list of global orientations.

The method of type **OrientationsObserver.OnOrientationDelete** is called when a view orientation has been deleted from the list of global orientations.

The method of type **OrientationsObserver.OnOrientationUpdated** is called when the property of an existing view orientation is changed.

The method of type **OrientationsObserver.OnDefaultOrientChanged** is called when the default orientation of the shapeview or active view is changed.

Screen Capture

Method Introduced:

- **EmbeddedControl.CaptureScreen**

The method **EmbeddedControl.CaptureScreen** captures a snapshot of the current screen and and saves it as an image to a file on disk. You can specify the height and width of the image. The supported image formats are `GIF`, `JPG`, `TIFF`, and `BMP`.

Selection Operations

Using the Java Application

The methods introduced in this section enable the Java application to select a component in the 3D view window. You can also use these methods to select multiple objects in the 3D view from the Java application.

Methods Introduced:

- **ShapeScene.GetSelectionController**
- **SelectionController.InsertItem**
- **SelectionController.RemoveItem**
- **SelectionController.ClearSelection**
- **ShapeInstance.SetSelected**
- **ShapeScene.AllSetSelected**

The method **ShapeScene.GetSelectionController** instantiates the selection object.

The method **SelectionController.InsertItem** adds an item to the list of selected items, while the method **SelectionController.RemoveItem** removes an item from the list of selected items.

The method **SelectionController.ClearSelection** clears the selection buffer of all the selected items.

The method **ShapeInstance.SetSelected** selects the specified instance in the 3D View. Use the method **ShapeScene.AllSelected** to select all the instances in the 3D view.

In the 3D Viewer

The methods described in this section enable you to select a component or multiple components in the 3D window and notify the calling Java application of the selected component, for example, using callback methods.

Selection of Instances

Methods Introduced:

- **SelectionObserver.OnBeginUpdate**
- **SelectionObserver.OnInsertItems**
- **SelectionObserver.OnRemoveItems**
- **SelectionObserver.OnClearSelection**
- **SelectionObserver.OnEndUpdate**

Instantiate a selection observer as follows:

```
public SelectionObserver getSelectionObserver()
{
     SelectionObserver so = new MySelectionObserver();
     ManageObject(so);
     return so;
}
```

Call the selection callback methods using this object. The method of type **SelectionObserver.OnBeginUpdate** indicates the start of a list of selected instances. This method is called first.

The method of type **SelectionObserver.OnInsertItems** is called when one or more instances have been added to the selection list. Pass an array of instances of type `com.ptc.pview.pvkapp.Instance[]` that have been added to the selection list as the input parameter for this method.

The method of type **SelectionObserver.OnRemoveItems** is called when one or more instances have been removed from the selection list. Pass an array of instances of type `com.ptc.pview.pvkapp.Instance[]` that have been removed from the selection list as the input parameter of this method.

The method of type **SelectionObserver.OnClearSelection** is called when all the items have been removed from the selection list, that is, selection of all the instances in the view are cleared.

The method of type **SelectionObserver.OnEndUpdate** is called to indicate the end of a sequence of changes to the selection list.

Selection of Items

Methods Introduced:

- **SelectionController.RegisterSelectionItemObserver**
- **SelectionItem.GetInstance**
- **SelectionItem.GetModelAnnotationID**
- **SelectionObserver.OnInsertItems**
- **SelectionObserver.OnRemoveItems**

The object `com.ptc.pview.pvkapp.SelectionItem` represents an individual selection in the 3D view. Use the method **SelectionController.RegisterSelectionItemObserver** to monitor the selection item events.

The method **SelectionItem.GetInstance** returns the selected instance and the method **SelectionItem.GetModelAnnotationID** returns the id of the annotations, if present in the instance.

The method of type **SelectionObserver.OnInsertItems** is called when one or more items have been added to the selection list. Pass an array of selection items of type `com.ptc.pview.pvkapp.SelectionItem[]` that have been added to the selection list as the input parameter for this method.

The method of type **SelectionObserver.OnRemoveItems** is called when one or more items have been removed from the selection list. Pass an array of selection items of type `com.ptc.pview.pvkapp.SelectionItem[]` that have been removed from the selection list as the input parameter of this method.

Accessing Combined View States

Combined views are used to switch between customized display states of the models. The methods described in this section enable you to access the view states of the models.

Methods Introduced:

- **ComponentNode.Visit**
- **ViewStateVisitorEvents.Visit**
- **ViewStateSource.GetType**
- **ViewStateSource.GetName**
- **ShapeView.SetViewState**

The method **ComponentNode.Visit** visits the component nodes in the structure to access the view states attached to the component nodes. This method takes a `com.ptc.pview.pvkapp.ViewStateVisitor` as the input argument. The following code shows you how to initialize a `ViewStateVisitor`:

```
//Inherit from the ViewStateVisitorEvents class

class MyViewStateVisitorEvents extends ViewStateVisitorEvents
         {

             public MyViewStateVisitorEvents()
             {
                 super();
             }

             public boolean Visit(com.ptc.pview.pvkapp.ViewStateSource
                                     viewStateSource)
             {
                  try
                  {
                       System.out.println("     - ViewState name: "
                                         +viewStateSource.GetName());
                       System.out.println("     - ViewState type: "
                                         +viewStateSource.GetType());
                       System.out.println("     -------");
                       listOfViewStates.add(viewStateSource);
                  }
                  catch (Throwable x)
                  {
                       System.out.println("Exception in
                                          MyViewStateVisitorEvents()");
                  }
```

```
                return true;
            }
        };


//Instantiate a VisitorEvents
        class MyInstanceVisitorEvents extends InstanceVisitorEvents
        {
            ArrayList<Instance> list = new ArrayList<Instance>();
            public MyInstanceVisitorEvents()
            {
                super();
            }
            public boolean Visit(Instance inst, Instance parent, int
                                  depth)
            {
                try
                {
                     MyViewStateVisitorEvents vsve = new
                                 MyViewStateVisitorEvents ();
                     ViewStateVisitor viewStateVisitor =
                                 pviewActor.getViewStateVisitor(vsve);
                     inst.GetComponentNode().Visit(viewStateVisitor);
                }
                catch (Throwable x)
                {
                     System.out.println("Exception in
                                    MyInstanceVisitorEvents()");
                }
                return true;
            }
        };

        class MyViewStateVisitorEvents extends ViewStateVisitorEvents
        {

            public MyViewStateVisitorEvents()
            {
                super();
            }
            public boolean Visit(com.ptc.pview.pvkapp.ViewStateSource
                                 viewStateSource)
            {
                 try
                 {
                      System.out.println("     - ViewState name: "
                                      +viewStateSource.GetName());
                      System.out.println("     - ViewState type: "
                                      +viewStateSource.GetType());
                      System.out.println("     -------");
                      listOfViewStates.add(viewStateSource);
```

Creo View Java Toolkit

```
                }
                catch (Throwable x)
                {
                     System.out.println("Exception in
                                          MyViewStateVisitorEvents()");
                }
                return true;
           }
       };

/* Visiting the viewStates */

 {      Orientations orientation;
        MyInstanceVisitorEvents ive;
        InstanceVisitor instVisitor;
        Instance root;

        orientation = theWorld.GetOrientations();

      MyOrientationsObserver orientObs = new MyOrientationsObserver();
      pviewActor.ManageObject(orientObs);

      orientation.RegisterObserver(orientObs.GetObjectId(),
           orientObs.GetOwner());

        tree = theWorld.GetTree();
       ive = new MyInstanceVisitorEvents ();
       instVisitor = pviewActor.getInstanceVisitor(ive);
       root = tree.GetRoot();

       root.Visit(instVisitor, 0);

       populateList ();
 }
```

The methods **ViewStateSource.GetName** and **ViewStateSource.GetType** return the name and the type of the view state associated with the model. The valid type of view states are:

- `VIEW_STATE`—Specifies a regular view state.
- `EXPLODE_STATE`—Specifies an exploded view state.
- `ALTERNATE_REPRESENTATION`—Specifies an alternate representation.
- `VIEW_STATE_SECTION_CUT`—Specifies a section cut.

Use the method **ShapeView.SetViewState** to apply that view state to the current active view.

Annotations

The methods in this section provide the ability to view, create, rename, and delete annotations for Creo View structure files. You can create, add, or rename annotations in a view and save them as an Annotation Set, which is stored along with the assembly. You can annotate only `.ed` and `.pvs` files.

Methods Introduced:

- **ShapeScene.CreateAnnotation**
- **ShapeScene.SaveAnnotation**
- **ShapeScene.ApplyAnnotations**
- **ShapeScene.AddAnnotation**
- **ShapeScene.RemoveAnnotation**
- **EmbeddedControl.LoadAnnotationSet**

There are two **CreateAnnotation** methods which take different arguments.

The first **CreateAnnotation (AsyncEventIf asyncEvent, ShapeView View, String name, String author, String tel, String email, String description)** method creates a new annotation set for the Creo View structure file. When you create a new annotation set, you can enter information such as the name of the annotation set, author, telephone number, e-mail address, and any other relevant comments.

After you add the annotation to an annotation set, you can save the annotation set using the method **SaveAnnotation**.

The method **ApplyAnnotations** applies an existing annotation set to the shape view.

The second **CreateAnnotation(string type)** method returns an annotation object of the specified type. You can directly interact with these object types. The following object types are supported:

- `Point`—Displays a point at the specified location in the shape view.
- `Circle`—Displays a circle at the specified location in the shape view.
- `Leaderline`—Displays a leaderline at the specified location in the shape view.

The annotations created by the **CreateAnnotation** method are put into the shape view by the **AddAnnotation** method. The annotations are now displayed in the shape view. If you change the properties of the annotations, the shape view window is dynmically updated.

The method **RemoveAnnotation** deletes the specified annotation from the shape view.

The method **EmbeddedControl.LoadAnnotationSet** loads the specified annotation set in the graphics view. Specify the name of the annotation as the input parameter for this method.

Product Manufacturing Information (PMI) for Markup Objects

The methods in this section provide the ability to access Product Manufacturing Information (PMI) for markup objects. The PMI data includes annotations, such as, 3D notes, dimensions, surface finish notes and symbols, general and weld symbols, geometric and dimensional tolerances (GD&T), set datum tags, and datum targets.

Methods Introduced:

- **Markup.GetFirstProperty**
- **Markup.GetNextProperty**
- **Markup.GetMarkupType**
- **MarkupType**
- **Markup.GetName**
- **Markup.GetProperty**
- **PMIProperty.GetName**
- **PMIProperty.GetValueType**
- **PMIPropertyValueType**
- **PMIProperty.GetSymbolValue**
- **PMIProperty.GetFloatValue**

- **PMIProperty.GetDoubleValue**
- **PMIProperty.GetInt8Value**
- **PMIProperty.GetInt16Value**
- **PMIProperty.GetInt32Value**
- **PMIProperty.GetBoolValue**

The method **Markup.GetFirstProperty** returns the first property of the markup object. Use the method **Markup.GetNextProperty** to get the next property of the markup object.

The method **Markup.GetMarkupType** returns the type of markup object. The enumerated class **MarkupType** is used to identify the type of markup and it has the following values:

- UNKNOWN
- NOTE
- BALLOON
- SET_DATUM
- SYMBOL
- SURFACE_FINISH
- SURFACE_FINISH_MACHINED
- SURFACE_FINISH_UNMACHINED
- TOLERANCE
- DIMENSION
- DATUM_PLANE_LABEL
- DATUM_CURVE_LABEL
- DATUM_POINT_LABEL
- DATUM_AXIS_LABEL
- DATUM_CSYS_LABEL
- DATUM_COSMETIC_LABEL

Use the method **Markup.GetName** to get the name of the markup object.

The method **Markup.GetProperty** returns the property associated with the markup object. The method takes as input the property name, which is a unique identifier for the associated piece of markup.

The method **PMIProperty.GetName** returns the name of the Product Manufacturing Information (PMI) property for the markup object.

The method **PMIProperty.GetValueType** returns the value of data type for the specified PMI property. The enumerated class **PMIPropertyValueType** is used to identify the value of data type. The enumerated class has the following values:

- `SYMBOL_VALUE`
- `FLOAT_VALUE`
- `DOUBLE_VALUE`
- `INT8_VALUE`
- `INT16_VALUE`
- `INT32_VALUE`
- `BOOL_VALUE`

The method **PMIProperty.GetSymbolValue** returns the symbol value for the specified PMI property using the parameter `symbolValue_holder`. The method returns `True` if the PMI property is a symbol value.

The method **PMIProperty.GetFloatValue** returns the float value for the specified PMI property using the parameter `floatValue_holder`. The method returns `True` if the PMI property is a float value.

The method **PMIProperty.GetDoubleValue** returns the double value for the specified PMI property using the parameter `doubleValue_holder`. The method returns `True` if the PMI property is a double value.

The method **PMIProperty.GetInt8Value** returns the 8-bit integer value for the specified PMI property using the parameter `intValue_holder`. The method returns `True` if the PMI property is a 8-bit integer.

The method **PMIProperty.GetInt16Value** returns the 16-bit integer value for the specified PMI property using the parameter `intValue_holder`. The method returns `True` if the PMI property is a 16-bit integer value.

The method **PMIProperty.GetInt32Value** returns the 32-bit integer value for the specified PMI property using the parameter `intValue_holder`. The method returns `True` if the PMI property is a 32-bit integer value.

The method **PMIProperty.GetBoolValue** returns the boolean value for the specified PMI property using the parameter `boolValue_holder`. The method returns `True` if the PMI property is a boolean value.

Circle Operations

Methods Introduced:

- **Circle.GetLocation**
- **Circle.SetLocation**
- **Circle.GetDiameter**
- **Circle.SetDiameter**
- **Circle.GetColor**
- **Circle.SetColor**

The method **Circle.GetLocation** returns the location of the circle. Use the method **Circle.SetLocation** to set the location of the circle.

The method **Circle.GetDiamter** returns the diameter of the specified circle. Use the method **Circle.SetDiameter** to set the diamter of the circle.

The method **Circle.GetColor** returns the color of the specified circle. Use the method **Circle.SetColor** to set the color of the circle.

Point Operations

Methods Introduced:

- **Point.GetPointType**
- **Point.SetPointType**
- **Point.GetPosition**
- **Point.SetPosition**
- **Point.GetColor**
- **Point.SetColor**

The method **Point.GetPointType** returns the type of a point. Use the function Point.SetPointType to set the point type. The following table lists the enumerated values and types of points:

Enumerated Values for Point Types	Description
SHAPE_CROSS	Creates a cross shaped point.
SHAPE_DOT	Creates a large dot shaped point.
SHAPE_STAR	Creates a star shaped point.
SHAPE_SMALL_DOT	Creates a small dot shaped point.

The method **Point.GetPosition** returns the position of a point. Use the method **Point.SetPosition** to set the position of the point.

The method **Point.GetColor** returns the color of the specified point. Use the method **Point.SetColor** to set the color of the point.

Lines of a Drawing Leader

Methods Introduced:

- **LeaderLine.GetArrowType**
- **LeaderLine.SetArrowType**
- **LeaderLine.GetPoints**
- **LeaderLine.SetPoints**
- **LeaderLine.GetLineWidth**
- **LeaderLine.SetLineWidth**
- **LeaderLine.GetLineStipple**
- **LeaderLine.SetLineStipple**
- **LeaderLine.GetLineColor**
- **LeaderLine.SetLineColor**

The method **LeaderLine.GetArrowType** returns the type of arrow used by a given leader line. Use the method **LeaderLine.SetArrowType** to set the arrow type.

The following table lists the enumerated values and types of arrows:

Enumerated Values for Arrow Types	Arrow Types
HEAD_NONE_TAIL_NONE	
HEAD_ARROW_TAIL_ARROW	
HEAD_NONE_TAIL_ARROW	

Enumerated Values for Arrow Types	Arrow Types
HEAD_ARROW_TAIL_NONE	
HEAD_NONE_TAIL_ROUND	
HEAD_ROUND_TAIL_NONE	
HEAD_ROUND_TAIL_ARROW	
HEAD_ROUND_TAIL_ROUND	
HEAD_ARROW_TAIL_ROUND	

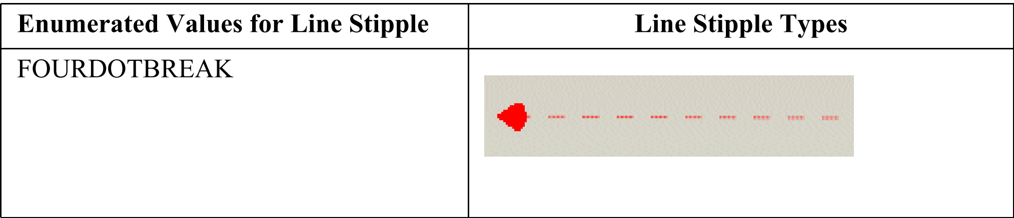
The method **LeaderLine.GetPoints** returns all the points used by a leader line. Use the method **LeaderLine.SetPoints** to set these points.

The method **LeaderLine.GetLineWidth** returns the width of a specified leader line. Use the method **LeaderLine.SetLineWidth** to set the width of the leader line.

The method **LeaderLine.GetLineStipple** returns the line stipple used by a leader line. Use the method **LeaderLine.SetLineStipple** to assign the line stipple.

The following table lists the enumerated values and types of line stipples. You can create the below mentioned line stipples for all the arrow types.

Enumerated Values for Line Stipple	**Line Stipple Types**
SOLID	
DOTDASH	
DOTDOTDASH	
DOTDOTDOT	
DASHDOTDASH	
DASHDASHDASH	
LONGDASHDOT	

Enumerated Values for Line Stipple	Line Stipple Types
FOURDOTBREAK	

The method **LeaderLine.GetLineColor** returns the color of a specified leader line. Use the method **LeaderLine.SetLineColor** to set the color.

Geometrical Data

The methods described in this section provide access to the geometric objects that is the face, edge, markup (for example geometric tolerance GTOL), and so on of the shape instance.

Methods Introduced:

- **ShapeInstance.GetGeometry**
- **ShapeInstance.Visit**
- **Geometry.GetReferences**
- **GeomReference**
- **GeometryType**
- **Geometry.GetReferrers**
- **Geometry.GetID**
- **Geometry.GetType**

The method **ShapeInstance.GetGeometry** returns the geometry (face, edge, and markup) associated with the shape instance. The method takes as input the Id, which is the unique identifier for the associated piece of geometry.

The method **ShapeInstance.Visit** visits all the geometry associated with the shape instance. This method takes the visitor class **GeometryVisitor** as the input argument to return the associated geometry.

GeometryVisitorEvent.Visit is the base class which should be derived from to receive the geometry associated with a shape instance.

The method **Geometry.GetReferences** returns references to other geometries such as GTOL reference faces and edges when the geometry type is markup. When the geometry type is a face or an edge or a datum (DatumCSys, DatumPlane, DatumAxis, or DatumPoint), the method **Geometry.GetReferrers** returns the model annotations associated with the specified geometry. These methods return the array **GeomReference** that contains the Id and information about the geometry type. The enumerated class **GeometryType** is used to identify the type of geometry. It has the following values:

- `GEOM_EDGE`
- `GEOM_FACE`
- `GEOM_DATUM`
- `GEOM_MARKUP`

The method **Geometry.GetID** returns the ID of the specified geometry using the parameter `ID_holder`. The method returns `True` if the geometry exists.

The method **Geometry.GetType** returns the value of geometry type using the parameter `type_holder`. The enumerated class **GeometryType** is used to get the value of the specified geometry. The method returns `True` if the geometry exists.

Manipulation Modes

Creo View provides translation and rotation modes to manipulate models in the view. The method described in this section provides access to these modes and is applicable only in the non-GUI mode.

Method Introduced:

- **ShapeView.SetMode**

The method **ShapeView.SetMode** enables you to rotate and translate the 3D shapeview in the viewing area. The input parameter *mode* has the following values:

- `translate`—Specifies the option to move selected parts in 3D space, so that parts in an assembly can be visually moved out of their regular position.
- `rotate`—Specifies the option to change the orientation of the selected parts.
- `select`—Specifies the option to quit the translation or rotation mode.

Accessing Viewable Files

All Creo View compatible file types other than `.ol` files, that is, images, drawings, documents, and PDF files present in the `.pvs` or `.ed` file are called Creo View viewable files. The methods in this section enable you to access the viewables in the product structure file.

Methods Introduced:

- **ComponentNode.Visit**
- **ViewableSource.GetName**
- **ViewableSource.GetDisplayName**
- **ViewableSource.GetType**

The method **ComponentNode.Visit** visits the viewable sources in the structure tree. This method takes `ViewableSourceVisitor` as the input argument.

The following code shows you how to initialize a `ViewableSourceVisitor`:

```
//Inherit from the ViewableSourceVisitorEvents class
static class MyViewableSourceVisitorEvents extends
```

```
com.ptc.pview.pvkapp.ViewableSourceVisitorEvents
{
       public MyViewableSourceVisitorEvents()
       {
              super();
       }

       public boolean Visit(com.ptc.pview.pvkapp.ViewableSource
                     viewableSource)
       {
              return true;
       }
}
//Instantiate a VisitorEvents
MyViewableSourceVisitorEvents vse = new MyViewableSourceVisitorEvents();
ViewableSourceVisitor_impl vsv = new ViewableSourceVisitor_impl();
vsv.SetEventHandler(vse);
ManageObject(vsv);

//Visit every component node in your structure to find all the Viewable
sources
myComponentNode1.Visit(vsv);
myComponentNode2.Visit(vsv);
```

The method **ViewableSource.GetName** returns the name of the specified viewable file. The name is the actual file name that is used for storing the data. This could be a temporary file.

The method **ViewableSource.GetDisplayName** returns the display name of the specified viewable file. The display name is the name that should be shown to users.

The method **ViewableSource.GetType** returns an integer specifying the type of the viewable file. The type could be a drawing file, document file, image file, and so on.

Accessing Properties

A property exists within a component and is used to store meta information. This can consist of text information or references to other files or locations that store information. Properties are grouped into any of several property groups. A property group is a classification of properties. A component may contain two or more of the same property as long as they exist in different property groups.

The methods described in this section enable you to access the properties or metadata associated with the selected part.

Methods Introduced:

- **Structure.Visit**
- **PropertyGroupInfo.GetName**
- **PropertyGroupInfo.GetFileSource**
- **PropertyGroupInfo.IsLoaded**
- **ComponentInstance.AddProperty**
- **Instance.GetProperty**
- **ComponentInstance.Visit**
- **PropertiesVisitorEvents.Visit**
- **Property.GetName**
- **Property.GetGroup**
- **Property.GetValue**

The method **Structure.Visit** visits the property groups. This method takes the `PropertyGroupInfoVisitor` as the input argument.

The method **PropertyGroupInfo.GetName** returns the name of the specified property group.

The method **PropertyGroupInfo.GetFileSource** returns the path to the reference file that contains the actual property data.

The method **ComponentInstance.AddProperty** adds a property for the component instance. Specify the name of the property, the property group, and value of the property.

The method **Instance.GetProperty** returns the value of the property specified by the property and the property group.

The method **ComponentInstance.Visit** provides access to the properties and values of the component instance. This method takes the `PropertiesVisitorEvents` as the input argument.

Use the methods defined in the class `com.ptc.pview.pvkapp.Property` to access the name, group, and value of the specified property.

Layer Operations

The methods described in this section enable you to access the properties of layers of 3D models.

Methods Introduced:

- **Layer.GetVisible**
- **Layer.SetVisible**
- **Layer.GetName**

The method **Layer.GetVisible** returns the visibility state of a layer of a 3D model. Use the method **Layer.SetVisible** to set the visibility state of a layer.

The method **Layer.GetName** returns the name of a layer.

Uploading and Downloading Files from a Webserver

Methods Introduced:

- **ProtocolHandlerEvents.DownloadFile**
- **ProtocolHandlerEvents.UploadFile**
- **ProtocolHandlerEvents.UploadData**
- **ProtocolHandlerEvents.CancelDownloadFile**

The `ProtocolHandlerEvents` class handles requests to upload or download Creo View files to and from a web server respectively. To upload or download files from a webserver, derive a class from `com.ptc.pview.pvloader.ProtocolHandlerEvents` and implement all or the required methods in this class.

The sample application `CheckServerPerformanceExample.java` demonstrates the use of the methods described in this section. You can specify the protocol as “`http`” or “`https`” for the method `myActor.GetProtocolHandler(phe, "http", remoteIf).` You can also specify multiple protocols using the format “`http:https`”.

The method of type **ProtocolHandlerEvents.DownloadFile** is used to receive notifications to download Creo View files from the web server. The input parameters of this method are:

- *url*—Specifies the location of the file on the webserver.
- *diskfile*—Specify the name of a file on the local machine into which the contents of the downloaded file are written.
- *handle*—Specifies a unique identifier for the download request.

Similarly the method of type **ProtocolHandlerEvents.UploadFile** is used to receive notifications to upload Creo View files to the web server.

Use the method of type **ProtocolHandlerEvents.UploadData** to receive notifications to upload data to the web server. In this method Creo View constructs the header to upload the data.

Use the method of type **ProtocolHandlerEvents.CancelDownloadFile** to stop the download of the file from the server.

3

Sample Applications

This section describes the sample applications installed with your Creo View Java Toolkit installation.

Topic	Page
Installing Sample Applications	3 - 2
Details of Sample Applications	3 - 3

Installing Sample Applications

When you install Creo View Java Toolkit from the CD-ROM, the Creo View Toolkit loadpoint directory contains all the libraries, example applications, and documentation specific to Creo View Java Toolkit. The following are the directories under `<creo_view_api_loadpoint>`:

Directory	Description
`<creo_view_api_loadpoint>/java/jar`	Contains the `pview.jar` and `pview_examples.jar` files.
`<creo_view_api_loadpoint>/java/src`	Contains sample applications source files.
`<creo_view_api_loadpoint>/demodata`	Contains the data used by the sample applications

This directory also contains the file `readme.txt` that specifies the instructions to setup and run the sample applications.

Running the Sample Applications Using Source Files

The original sample applications files provided with Creo View Java Toolkit are available at `<creo_view_api_loadpoint>/java/src`.

To run the sample applications:

1. Set up a project in an Integrated Development Environment (IDE), for example, Eclipse.
2. Copy the `ExamplesBase.java` and `ExamplesUtilities.java` files into the project in the correct package folder.
3. Include the `<creo_view_api_loadpoint>/java/jar/pview.jar` file in the dependency package. This file is required to run Creo View through a Java interface.
4. **Compile** and **run** the `ExamplesBase.java` sample application. This serves as a basic test to check if Creo View Java Toolkit is installed properly. If the installation is successful, the Creo View client is launched.
5. Include the sample application `.java` file that you want to run into the same package folder.
6. **Compile** and **Run** the Java application.

Running the Sample Applications Using the Pre-compiled Jar File

The file `pview_examples.jar` available at `<creo_view_api_loadpoint>/java/jar` contains the compiled sample application files. To run the sample applications using this file:

- Confirm the path to the Java JRE installation.
- At the command prompt, browse to `<creo_view_api_loadpoint>/java`. For example,

```
cd C:\ptc\creo_view_api\java
```

- Run the sample application, for example, `ConfigureViewableAssemblyExample` by specifying the following command:

```
<full_path_to_jre>\bin\java.exe" -cp
.\jar\pview.jar;.\jar\pview_examples.jar
com.ptc.pview.pvexamples.ConfigureViewableAssemblyEx
ample
```

All the sample applications provided with your installation are included in the file `pview_examples.jar`. To run the other sample applications, replace the name of the sample application with the one that you want to run in the above command.

Details of Sample Applications

The following table provides more details on the sample applications.

Sample Application	Filename	Description
Animate Assembly	`AnimateAssemblyExample.java`	This example shows you how to animate the process of assembling and disassembling an existing product assembly programmatically.

Sample Applications

Sample Application	Filename	Description
Create Annotations	AnnotationExample.java	This example shows you how to create annotations in a model using various options.
Build Assembly	BuildAssemblyExample.java	This example modifies the product structure, loads and unloads components from the scene, and moves (rotates and translates) components, and saves the resulting assembly to disk.
Check Server Performance	CheckServerPerformanceExample.java	This example enables performance check across a LAN and WAN by calculating the time taken for a structure stored on a Web server to load.
Configuration of Assembly	ConfigureAssemblyExample.java	This example configures which .ol file is to be associated with a given component node and saves the resulting assembly to disk.
Reading Properties	ReadPropertyExample.java	This example reads the property of the selected component and activates the Help page or other related documentation for the component.
Create Scene	CreateSceneExample.java	This example shows you how to create your own scene and view.
Read files from Web server	ReadWebserverExample.java	This example shows how to open files that are stored on a Web server.

Sample Application	Filename	Description
Configure Viewable Assembly	`ConfigureViewableAssemblyExample.java`	This example creates a Java user interface that contains all the components in the structure tree in two lists; a hidden components list and a shown components list. You can load or unload components using the Java user interface. The resulting hidden or shown state of the assembly can be saved to disk.
View States	`ViewStateExample.java`	The example lists all the view states and the orientations of the model in the 3D view. It allows you to select and set any of the orientations and view states. In addition, there is a menu option to write the information regarding the view state and orientation to a file.

Index

A

accessing
- combined view states 2-34
- properties 2-50
- viewable files 2-48

annotations 2-37

attributes
- Creo View-related objects 2-4
- of observers or action listeners 2-6
- of utilities 2-6

C

circle operations 2-41

class
- observers and action listeners 2-5
- Product View-related interfaces 2-4
- utilities 2-6

class types 2-3

combined view states 2-34

component operations 2-21

creating applications 2-7
- application hierarchy 2-7
- deployment 2-10
- importing packages 2-7

Creo Parametric files 2-12

Creo View data structures 1-8

Creo View world object 2-10

D

data
- geometrical 2-46

E

exceptions
- Creo View-related objects 2-4
- observers or action listeners 2-6

F

fundamentals 1-2

G

geometrical data 2-46

I

initialize
- Creo View-related objects 2-4
- observers or action listeners 2-5
- utilities 2-6

installation
- configuration 1-5
- prerequisites 1-2
- requirements 1-2

installing Creo View Toolkit 1-3

L

lines of a drawing leader 2-43

M

methods
- of Creo View-related objects 2-4
- of observers or action listeners 2-6
- of utilities 2-7

mode
- rotate 2-48
- translate 2-48

O

observers or action listeners
    description of the class 2-5
operations
    component 2-21
    selection 2-31
    view 2-25
overview
    Creo View data structures 1-8
overview up Java APIs 2-3

P

product manufacturing information (PMI) for
        markup objects 2-38
product structure 2-13
    creating 2-13
    modifying 2-20
    visiting 2-18
Product View-Related interfaces 2-4
properties 2-50
ProtocolHandlerEvents class 2-51

S

screen capture 2-30
selection
    in the 3D viewer 2-31
    using Java application 2-31
support for Creo Parametric files 2-12
system requirements 1-3

U

user interface
    GUI mode 1-5
    non-GUI mode 1-5
utilities 2-6

V

view
    operations 2-25
    orientations 2-29
viewable files 2-48

W

Webserver
    download 2-51
    Upload 2-51