openInstaller Developer Tools Detailed Design Document v0.2 Sept. 10, 2007

Vadiraj Deshpande

Document History

Draft version	Date	Author	Summary of changes
0.1	August 27th	Vadiraj Deshpande	First draft version
0.2	September 10th	Vadiraj Deshpande	Added new features and updated few TODOs

Table of Contents

Introduction

openInstaller Framework[1] is a generic framework for developing installers mainly targeted for middleware products and product stacks such as Java ES. The openInstaller Developer Tools is a sub project, which aims at simplifying overall installer development experience and is targeted for product teams and installer developers who want to build installers on top of openInstaller Framework.

The scope of the current document is to define a detailed design of the openInstaller Developer Tools (a.k.a., openInstaller IDE) based upon the use cases derived.[2] The design of each use cases (and thus features of the tool) contain technical information.

Document conventions

Throughout the document the following conventions apply.

1. When there is a mention of the word 'Netbeans', it represents both Netbeans IDE and Netbeans platform.
2. When referring to the word 'user' or 'tool user', its assumed that both the words refer to Installer Developer.

Design Details

1 Feature : Tool distribution and deployments

The tool is distributed in two types : as Netbeans IDE plug-in and as a stand-alone Java Desktop application.

1.1 Plug-in version

The users of Netbeans IDE can connect to the update center (which hosts the plug-in) and can download and install the tool in their IDE along with other existing and built-in plug-ins. This tool will add a new global menu for all the tool specific operations and will add the support for creating new installer project type(s) among other features. Installer Developers will be using these two features (mainly) to build installers using the tool.

1.2 Stand alone version

The stand-alone version of the tool will be available for download from the same external website (openInstaller.dev.java.net). The stand-alone application can be distributed as a zip (bundle) archive, as JNLP hosted/downloadable application or as a product bundle installer created using openInstaller itself! The zip bundle requires the developer to unzip it in an empty directory. By default, the stand-alone will be a single instance application. That means, one user can run only one instance of the tool. We do not think that there could be requirements to run multiple instances of the tool. This behavior is because of the underlying Netbeans platform.

The stand-alone application bundles only required modules from the whole Netbeans platform and Netbeans IDE cluster[1]. (For example, the application does not need any J2EE modules) Hence it is expected to have lesser footprint than these two combined. The stand-alone application is branded as required, while the plug-in is not. The branding includes a special splash screen, About box, wizard background, custom icons and main window title among few other customizations.

1.3 Update Center

The update center will be hosted on the external site (openInstaller.dev.java.net) and the tool modules are pushed regularly. There can be a possibility that there would be multiple update centers for multiple versions of Netbeans IDE and J2SE. This tool bundles the openInstaller runtime from the framework, so that installer developers have to only download the tool to get started with Installer Development on openInstaller Installer Framework.

1.4 Tool Updates

The plug-in can be automatically (or manually) updated using the same update center that is used to download it. The updates are posted onto the same update center regularly based on the future development schedule. The update experience is similar to other module updates in the Netbeans IDE. The update center for the tool can be configured when the tool is downloaded and installed for the first time. User can also manually setup and configure update center, if required.

1 In Netbeans world, a cluster is a bundle of related modules which work as monolithic application. Existing cluster examples are, Netbeans IDE, Netbeans Enterprise Pack and Netbeans Mobility Pack.

The stand alone application can also be updated using the in built update capabilities of the openInstaller in case, we decide to install the stand-alone version of the tool using openInstaller itself. Otherwise, the update experience would be same as that of the plug-in.

2 Feature : User experience of the tool

2.1 Plug-in version

The plug-in version adds a new global menu in the existing Netbeans IDE which houses the important operations pertaining to creating and updating installers. Also, a new custom project support is added to the existing IDE. A new update center is also added and registered as part of the plug-in installation. This update center will be used to update the plug-in after its installation in the IDE. We might plan to add a new toolbar and some buttons for few features on it, but its TBD as of now. For most of the part, the plug-in interaction and operates like any other Netbeans plug-in general interaction and has the similar user experience.

2.2 Stand alone version

Welcome screen

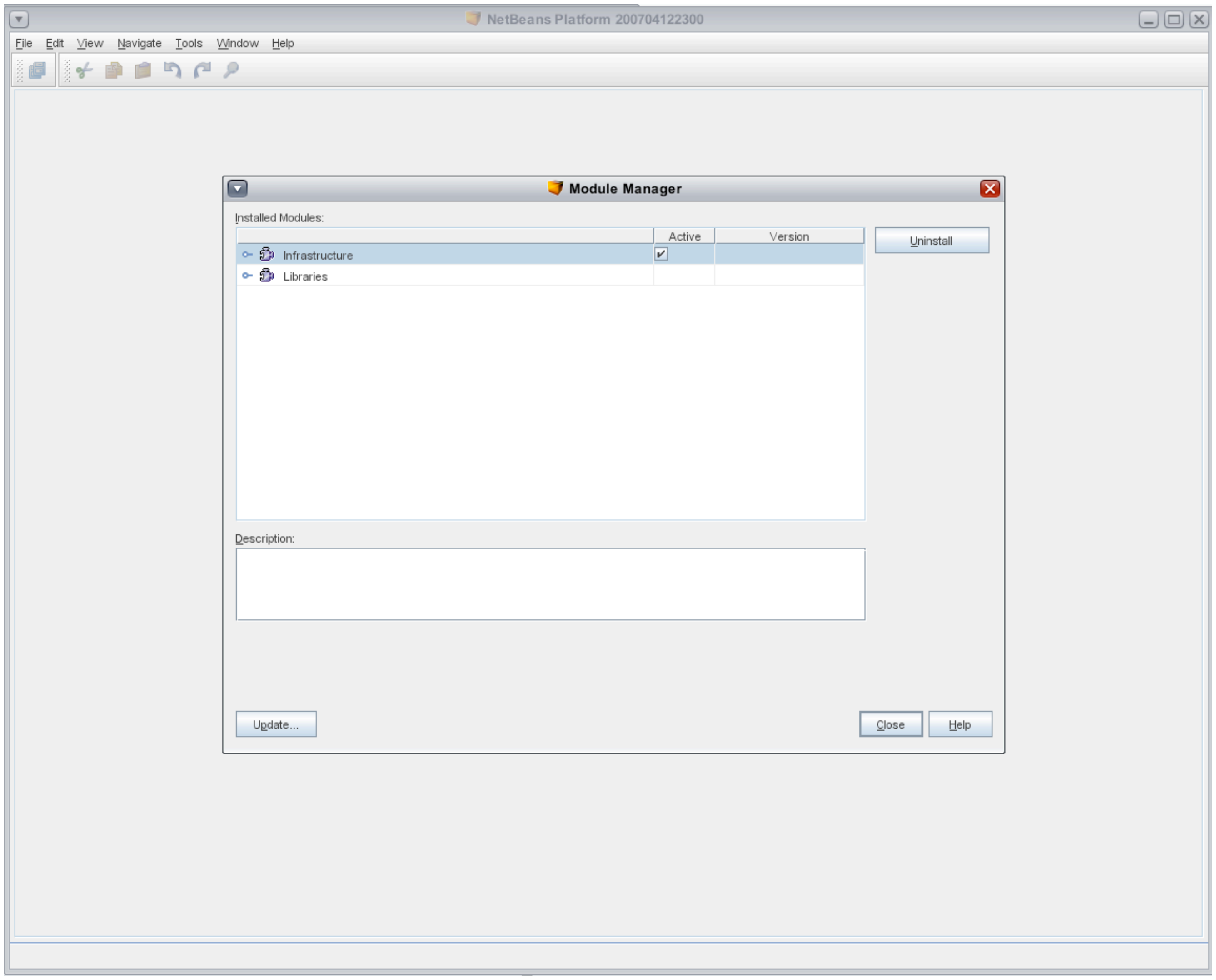
When the stand alone version is executed, the tool will display a screen similar to the Netbeans IDE welcome screen, with some content. The Welcome screen is planned to have following items:

1. The screen is divided into three sections : Projects section, Help section and Project news section.
2. The Projects section has and has links to the recently opened Installer Projects (if any).
3. The Help section as links to the online tutorials, guides and project wiki pages.
4. And finally the Project news section will have news and articles hosted on the project website. (RSS syndication)

This screen can be turned off using a setting, else this screen will always popup when the tool is started.

Application appearance

The tool will have a similar appearance as of the Netbeans IDE with notable exceptions. It does not have all the menus from the IDE. It does not have all the windows from the IDE. The following image (Fig. 1) shows the bare minimum Netbeans platform as a reference. The stand alone tool will have additional menus and functionality added to this platform.



Installed Modules:

	Active	Version
Infrastructure	✓	
Libraries		

Fig.1 Netbeans platform

3 Feature : Create/Edit Installer Projects

Once installed, the tool enables creating and editing Installer Projects. These 'Installer Project's are custom Netbeans projects designed to house all the necessary installer files together and adds the support for adding and editing those files.

Installer Developer can create a new Installer Project in several ways. The welcome screen will have an option to create a new installer project. The global menu will have one menu item and the context menu of the Project window will have a menu item.

3.1 Creation of Installer Project

All the above option will open the 'New Project' wizard. This wizard will allow the developer to choose the openInstaller Installer Project type in the first wizard panel. On the second panel, the developer is required to select/choose the location for the new project and a name for the project. In the 'Expert' mode, the developer has options of customizing the project layout by specifying the sub folders inside the project folder. The developer will also get to specify other advanced options here. (TBD).

The tool will validate all the user inputs on this 'New Project' wizard and displays the validation error messages, if any on the wizard itself. Netbeans wizards are capable of displaying in line validation messages at the bottom without much extra effort and at the same time, they have the capabilities to disable wizard navigation until the error is corrected. These validation include, valid path for the installer project (i.e., the user entered path should exists before), Apart from semantic validations, the actual validation include checking for the write permission of the user specified folder for the project and checking for sufficient disk space to create the project. The actual validations are done on the finish panel of the wizard by marking that panel as 'Validating Panel'. (see API documentation)

3.2 Files created during the new installer project creation

When the wizard terminates with proper user inputs, it creates a new installer project in the specified location with the specified name. By default, the newly created installer project will have the following folders/files already created under the project folder:

1. installerProject (folder)
2. installerProject/installerProject.properties (file)

   The above two are used as 'markers' by Netbeans platform, while identification of Installer Project during opening.
3. metadata (folder)
4. metadata/descriptor (folder)
5. metadata/model (folder)
6. metadata/view (folder)
7. metadata/view/splash.jpg (file)
8. metadata/view/splash.properties (file)
9. metadata/templates (folder)
10. metadata/pagesequence.xml (file)
11. metadata/pagesequence.properties (file)
12. scripts (folder)
13. scripts/unix (folder)
14. scripts/windows (folder)
15. scripts/unix/product-installer.sh (file)
16. scripts/unix/product-uninstaller.sh (file)
17. scripts/unix/install.properties (file)
18. scripts/windows/product-installer.vbs (file)

19. scripts/windows/product-uninstaller.vbs (file)
20. scripts/windows/install.properties. (file)

All of the above, expect the first and second (marker items) can be configurable in 'Expert' mode. The installer project records these paths and filenames in 'installerProject.properties file to allow customization. The installer project refers these files using the 'installerProject.properties' file.

This customization takes precedence if there is a strong push towards it.

3.3 Installer Project types

There are at least two main types of installer projects available for creation. One is a blank project which does not have any components and another, a sample installer project which has few sample components and enable user to quickly prototype his/her installer based on these components.

We plan to use the sample installer project from the openInstaller source base and bundle the built version with the tool. We will maintain a separate copy in the tools source base for this sample project.

Installer project, once created, supports creation and editing of many files of different types that constitute the installer. The creation and editing of the files can be using several menus under different options.

TODO --- bring in the platform dependent and platform independent project types here.

3.4 Storing the changes

All the changes for the installer project files are stored onto the disk automatically or manually (when clicking on 'Save' button or using the keyboard accelerator). The project metadata (mainly the contents of installerProject.properties file) are also updated as required.

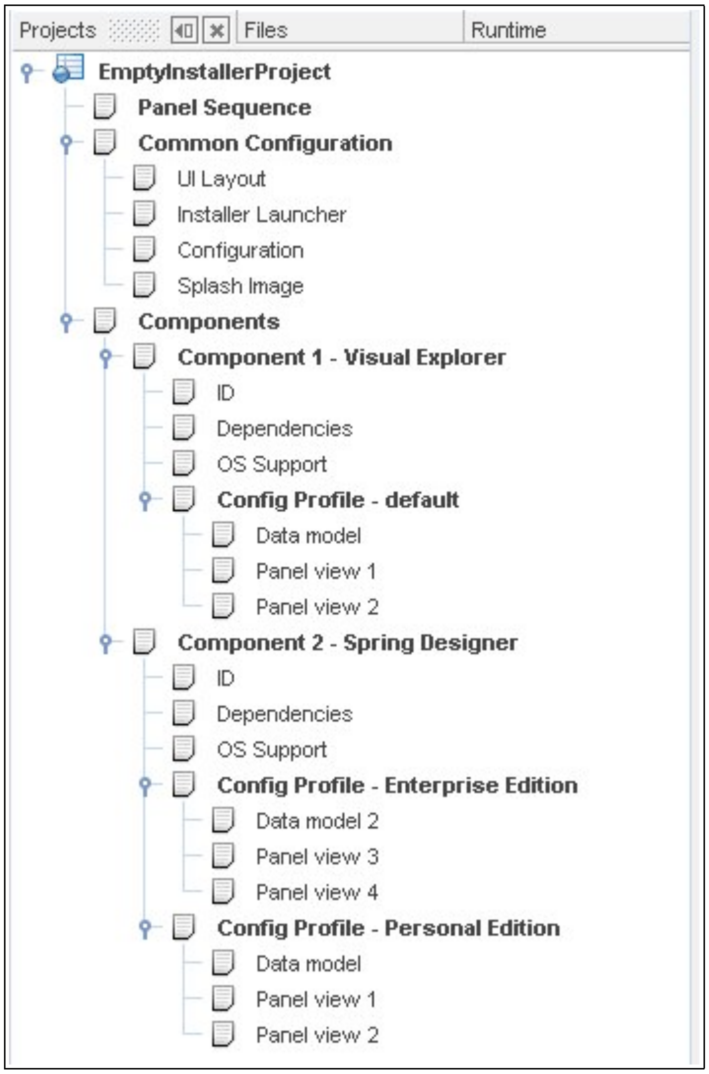
3.5 Opening the existing Installer Project

Netbeans recognizes custom projects by two markers : 1. A folder which is usually used to store project metadata and 2. A project file, which again usually used to store the project metadata contents. This is just a convention and we can have multiple markers here. The Installer Project uses a marker folder called 'installerProject' and a marker file called 'installerProject.properties' under that folder for recognition.

If these marker folder and/or files are lost, there is no in-built way to regenerate them and Netbeans cannot recognize Installer Project without these folder and file in place.

While displaying the files and folders in 'Open Project' dialog, Netbeans examines each of the folder and tries to detect the known projects. If there are any known projects, it marks them with a special icon to show them that the folders are in fact Netbeans recognized projects and users can select and open them.

Each Installer Project will have a logical view in the 'Project' window, which displays the project files in a tree with nodes for each of them. The tree nodes themselves are grouped in different levels. The following figure (Figure 2) shows the current status of the project logical view.



```
EmptyInstallerProject
    Panel Sequence
    Common Configuration
        UI Layout
        Installer Launcher
        Configuration
        Splash Image
    Components
        Component 1 - Visual Explorer
            ID
            Dependencies
            OS Support
            Config Profile - default
                Data model
                Panel view 1
                Panel view 2
        Component 2 - Spring Designer
            ID
            Dependencies
            OS Support
            Config Profile - Enterprise Edition
                Data model 2
                Panel view 3
                Panel view 4
            Config Profile - Personal Edition
                Data model
                Panel view 1
                Panel view 2
```

Figure 2 : Proposed (by UI experts) Project logical view

Project Logical View

The project logical view shows the project contents in a more intuitive way. There are context menus attached to some of the tree nodes that allow Installer Developer to perform project related tasks.

There are two top nodes under the project node. **Components** node holds (or groups) information about individual components in the installer project. It has context sensitive tasks to create new components and edit existing ones. The other top node is **Common** node, which includes the files that belong to the installer and not to any component. For example, panel sequence files (also called page sequence) which stores the order of installer panels. These two top nodes will have several sub nodes.

The **Components** node will have a sub node for each component in the installer. Generally components are top level entities in the installer. Each component is described by its identification information (viz., name, version etc..), its install bits (also called as installation units) and its dependent components. Optionally each component also has install-time configuration information. All this information is displayed in a group under the tree node named after the component.

The **Common** node will have sub nodes for each common installer files. Splash image for the installer, common configuration information, panel sequence files and installer launcher script are grouped here. Optionally there can be sub nodes representing the advanced features of the installer like, installer layout files. These layout files will help in customization of installer layout properties.

Each component node will have sub nodes which represents the component contents as shown in the following figure Fig. 3.

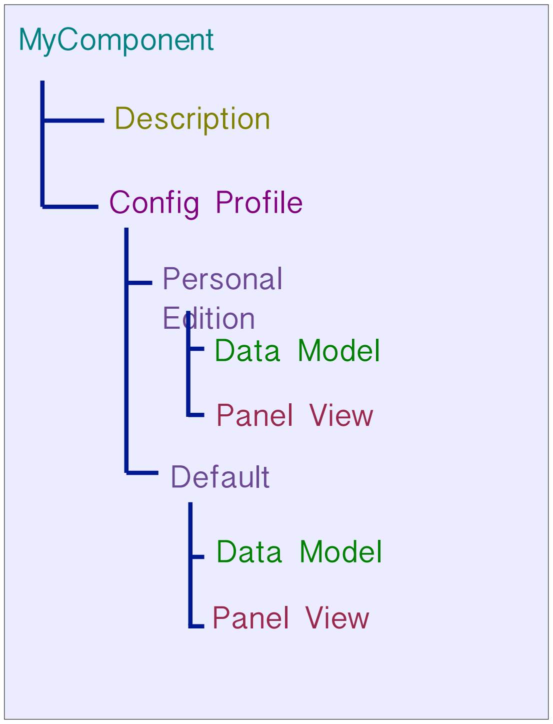


Figure 3. Component sub nodes.

The top node in this figure represents the component node for the component itself ('MyComponent' in this case). The 'Description' node contains the component identification information, its install unit information and its dependency information.

Config Profiles

The 'ConfigProfile' node contains the configuration information for the component collectively. More than one configuration profiles can be created as required. Each 'ConfigProfile' represents a logical group of configuration, which is based on the same set of install units (the actual bits that this component is made up of). Each 'ConfigProfile' can contain different configuration information for the same component to configure the component in different ways as required. For example is Sun Java Application Server 8.X. This component has two flavors : Personal Edition (PE) and Enterprise Edition (EE). Both contains the same install units but they differ in the way they are configured and the configuration information is different for these two flavors. This 'ConfigProfile' notion enables this feature. Note that this feature, is in fact provided by this openInstaller Developer Tool only. This feature is not provided by the underlying openInstaller

Framework. So this information about 'ConfigProfile's are stored in the installer metadata and not in any installer files.

For every component created or included in the tool, there will always be a **default** 'ConfigProfile' which holds the configuration information.

Each 'ConfigProfile' holds configuration information in 'data models' and 'panel views'. A **data model** is a file (represented as a sub tree node under the ConfigProfile node) which holds the configuration entities. These entities describe the configuration data and its properties. The Panel View or just **View** describes the page view of the installer. These views are mapped to a single data model or multiple data models. Each UI field on the page can be mapped to a configuration data in the data model. For more information please refer the Developer workflow document here[3].

Pop up menus

There will be pop up menus for many tree nodes and the pop up menus in all cases contain the context sensitive menu items.

Project node menu

The top project node will have the following as menu items.

1. Project related menus
    1. Close project
    2. Delete project
    3. Rename project
    4. 'Set as Main' project
    5. Save project
2. Project level menus
    1. Create new component.
    2. Create new files. (configuration information files, common installer files)
    3. Import files (from other installer projects)
    4. Include files (from other installer projects)
    5. Build the installer
    6. Verify the installer (Runs the integrity check)
    7. Test run the installer

Components Node menu

The **Components** node will have the following as menu items.

1. Project level menus
    1. Create new component.
    2. Create new files. (configuration information files, common installer files)
    3. Import files (from other installer projects)

    4. Include files (from other installer projects)
2. All components related tasks
    1. View Dependency tree for the installer.
    2. Verify Installer (TBD, do we need this here?)

Component Node menu

Each component will have the following as menu items.

1. Edit component (will allow editing of identification, dependency and install unit information)
2. New Data Model
3. New Panel View
4. Copy component
5. Paste component
6. Delete Component

Config Profile Node Menu

The 'Config Profile' will have the following menu items.

1. Create Config Profile
2. Delete Config Profile

Config Profiles Node Menu

Each 'Config Profile' node will have the following menu items.

1. Rename
2. Delete

Data Model Node Menu

Each Data Model node will have the following menu items.

1. Edit
2. Rename
3. Delete
4. Create View

View Node Menu

Each View node will have the following menu items.

1. Edit
2. Rename

3. Delete
4. View Mockup (Preview panel)

Common Node Menu

The 'Common' Node will have the following menu items.

1. New Data Model
2. New View
3. Edit Panel Sequence
4. Edit Panel Sequence Properties
5. Edit Install layout (advanced option)
6. Edit Install Launcher (advanced option)

4 Feature : Create New / Edit Component

Every Installer Project supports creating new components and editing them. The creation of the components is possible using more than one option. This is done for convenience. In all cases, a wizard will guide the user and at the end of wizard completion, the new component is created and it is opened in the visual editor for further editing.

Installer developer can create a new component either using global menu or the tool bar button for 'New File' option. Installer developer can also create new component using the pop up menu item on the project node or 'Components' node.

4.1 Create New Component

The wizard used in creation of a new component will have several panels. On the first panel, the basic identification details of the component viz., component name, component version, vendor information and the component description will be recorded. (see Appendix chapter on how wizard API is extended for this tool)

The following screen shot shows this first panel as per the current implementation.

Identification wizard panel

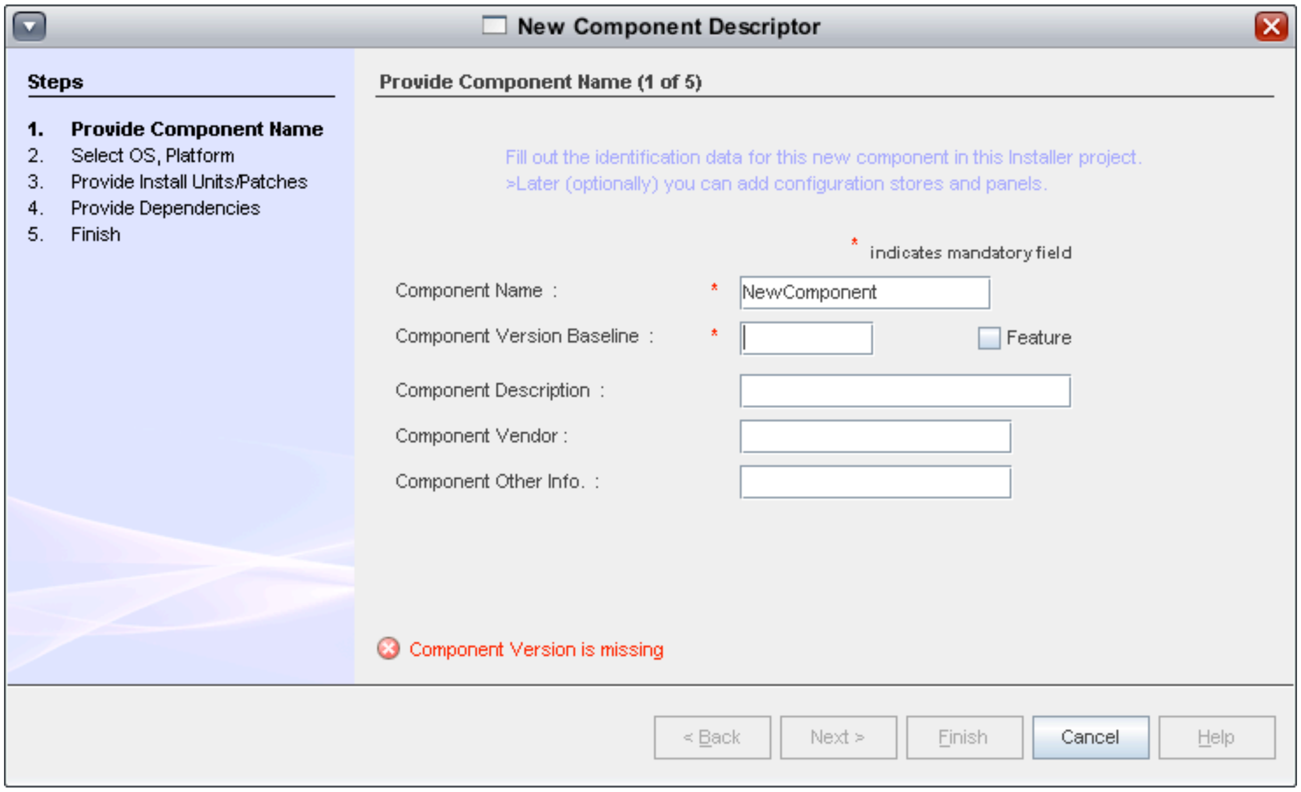
Netbeans wizard provides in place validation and this aspect is shown in the screen shot below in figure 4.. Apart from this 'in-place' validation, the wizard API also provides the functionality validation that would get triggered when the user clicks on the 'Next' button on the wizard panel. Both types of validations will be used to validate the user data appropriately.

Following will be some of the validation checks (both 'in-place' and functional)

1. In an Installer project, there can be no two components with the same name (i.e., **the component name is unique to the installer project**).
2. Component name and version cannot be empty.

3. The description is a free form string.
4. All fields accept alpha-numeric characters, but few special characters.

The 'Next' button will be disabled until all validation checks are passed. (see Appendix chapter on how wizard API is extended for this tool)


New Component Descriptor

Steps

1. **Provide Component Name**
2. Select OS, Platform
3. Provide Install Units/Patches
4. Provide Dependencies
5. Finish

Provide Component Name (1 of 5)

Fill out the identification data for this new component in this Installer project.
>Later (optionally) you can add configuration stores and panels.

* indicates mandatory field

Component Name :	*	NewComponent
Component Version Baseline :	*	
Component Description :		
Component Vendor :		
Component Other Info. :		

Feature: ☐

Component Version is missing

Figure 4. First wizard panel of New component wizard.

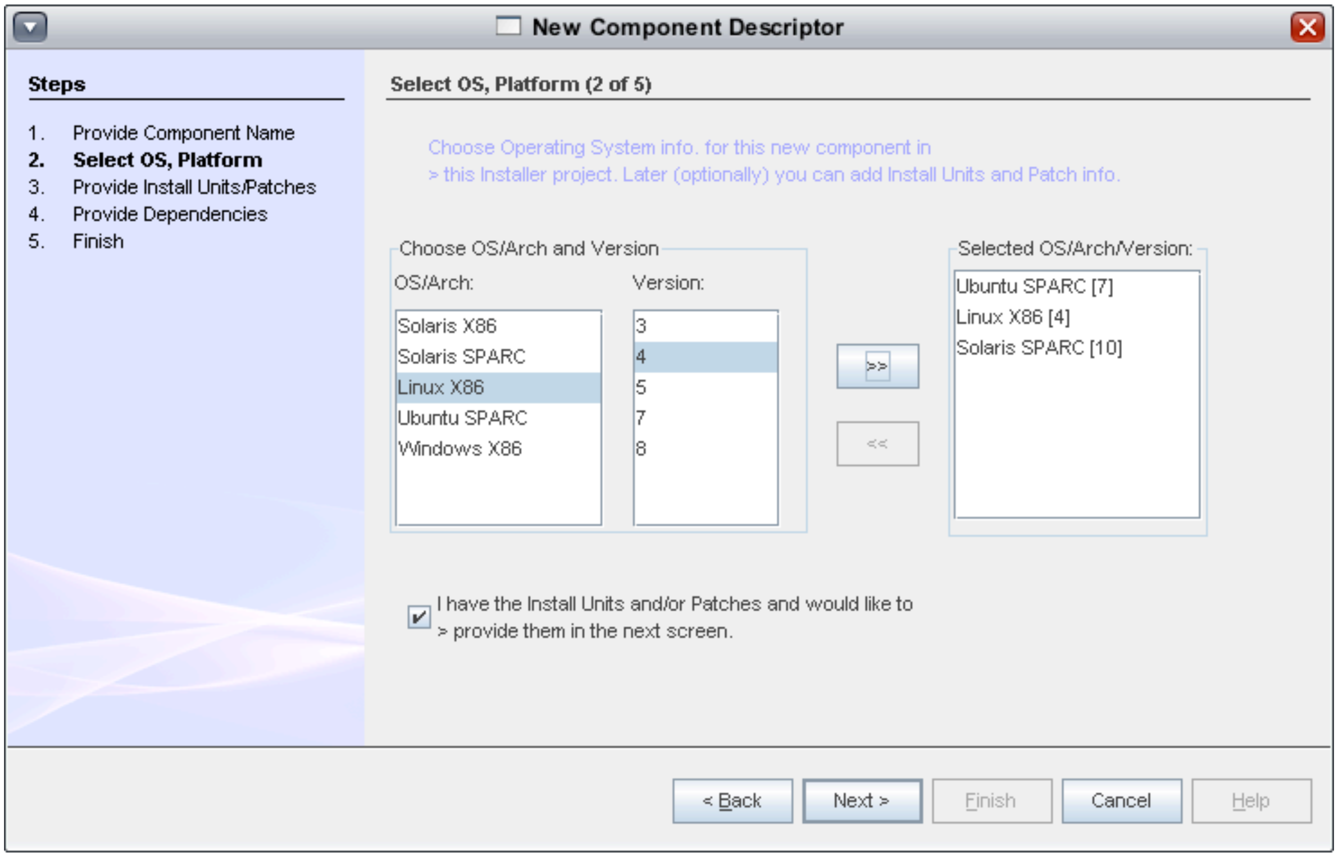
OS Info. Panel

This panel allows the user to select operating systems that the installer should support. This is done by selecting operating system name, its version and the architecture.

The figure 5 shown above has the screen shot of the current implementation for this OS info wizard panel. In this, the first list on the left contains the list of operating systems and architectures that the underlying openInstaller Installer Framework supports for building installers. The list in the middle shows the supported versions of the selected operating system on the left list. Finally the right most list collects the user selections of the OS, its version and the architecture.

The list of supported operating systems, their versions and architectures are derived from a fixed list stored in Platform.java (which belongs to the openInstaller Framework source code). The only validation check on this panel is 'user should select at least **one** combination of OS, version and architecture that the current component is intended to support.

The checkbox at the bottom enables user to decide when they want to provide the actual install units information for each of the OS.version and architecture combination. If they select this option (i.e., they select the checkbox) then, on the next panel, users can enter information about the install units, else they can enter this information in future after the component is created. This option helps the user to quickly create a new component in the installer with minimal data at hand. Later, user can edit the component information anytime by opening it in the editor.


New Component Descriptor
Steps
1. Provide Component Name
2. Select OS, Platform
3. Provide Install Units/Patches
4. Provide Dependencies
5. Finish
Select OS, Platform (2 of 5)
Choose OS/Arch and Version
OS/Arch:
Version:
Selected OS/Arch/Version:
>>
<<
< Back
Next >
Finish
Cancel
Help

Choose Operating System info. for this new component in
> this Installer project. Later (optionally) you can add Install Units and Patch info.

OS/Arch:	Version:
Solaris X86	3
Solaris SPARC	4
Linux X86	5
Ubuntu SPARC	7
Windows X86	8

Selected OS/Arch/Version:
Ubuntu SPARC [7]
Linux X86 [4]
Solaris SPARC [10]

[✓] I have the Install Units and/or Patches and would like to
> provide them in the next screen.

Figure 5. Second panel of the New Component wizard

Install Unit Panel

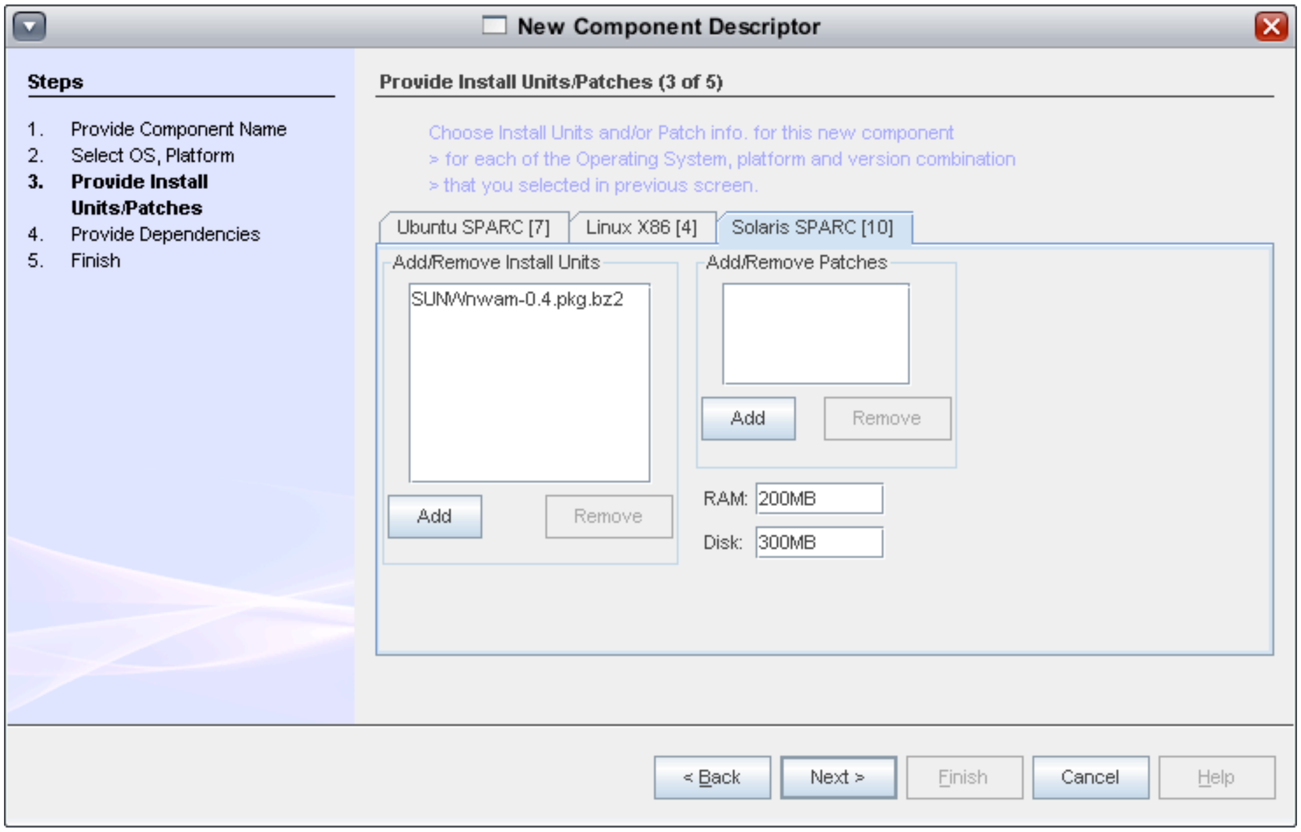
This panel is the third panel in the new component wizard. This panel allows the user to provide information about install units for each of the OS,version and architecture combination.

On this panel(shown below), user can select the install units in a separate tabs for each of the OS, version and architecture combination. Platform specific checks are made here to identify the install unit and read its metadata, if its possible. For example, for solaris, the user selects a SVR4 package for addition, the tool can read the 'pkginfo' file to get the attributes of the package automatically.

The following are the validation checks on this panel.

1. Make sure user selects at least one install unit per OS, version and architecture combination
2. The RAM and DISK fields should not be empty and should contain a valid data.

(TBD – are these hard rules? We can relax these, if we want provide flexibility to the user)


New Component Descriptor

Steps

1. Provide Component Name
2. Select OS, Platform
3. **Provide Install Units/Patches**
4. Provide Dependencies
5. Finish

Provide Install Units/Patches (3 of 5)

Choose Install Units and/or Patch info. for this new component
> for each of the Operating System, platform and version combination
> that you selected in previous screen.

Ubuntu SPARC [7] | Linux X86 [4] | Solaris SPARC [10]

Add/Remove Install Units

SUNWnwam-0.4.pkg.bz2

Add/Remove Patches

RAM: 200MB

Disk: 300MB

Figure 6. Third panel in New component wizard.

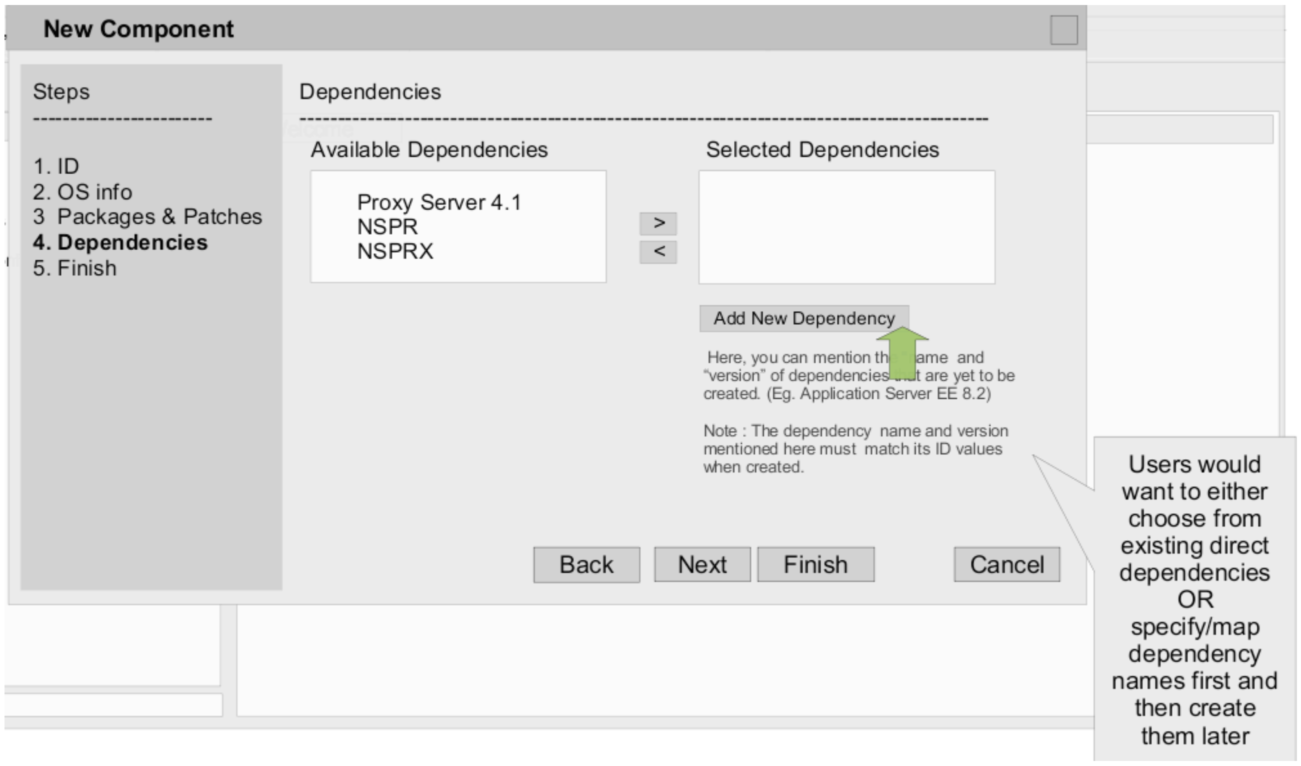
Dependency Panel

This panel allows the user to add new dependencies for this new component. The UI experts recommendation is we should avoid errors in the user input and hence, adding a new dependency by typing its name and version has been removed. As user is allowed to choose a new dependency for the new component which exists already in the project, we can provide a list to the user to select it rather than allowing the user to type it. The following figure 7, shows the screen shot of the current UI design mockup screen. Please note that the current implementation of the wizard panels is subject to change in future.

In the panel below, the list on the left contains the list of components from the current installer project. These components include the imported components too. On the right, the list will collect the user selections. Note that the 'Add New Dependency' button has been removed.

There can be no validation checks for this panel or can be one to check whether user has selected at least one dependency. This is TBD.

TODO – handle the JDK component dependency separately here.



Here, you can mention th ame and "version" of dependencies t are yet to be created. (Eg. Application Server EE 8.2)

Note : The dependency name and version mentioned here must match its ID values when created.

Users would want to either choose from existing direct dependencies OR specify/map dependency names first and then create them later

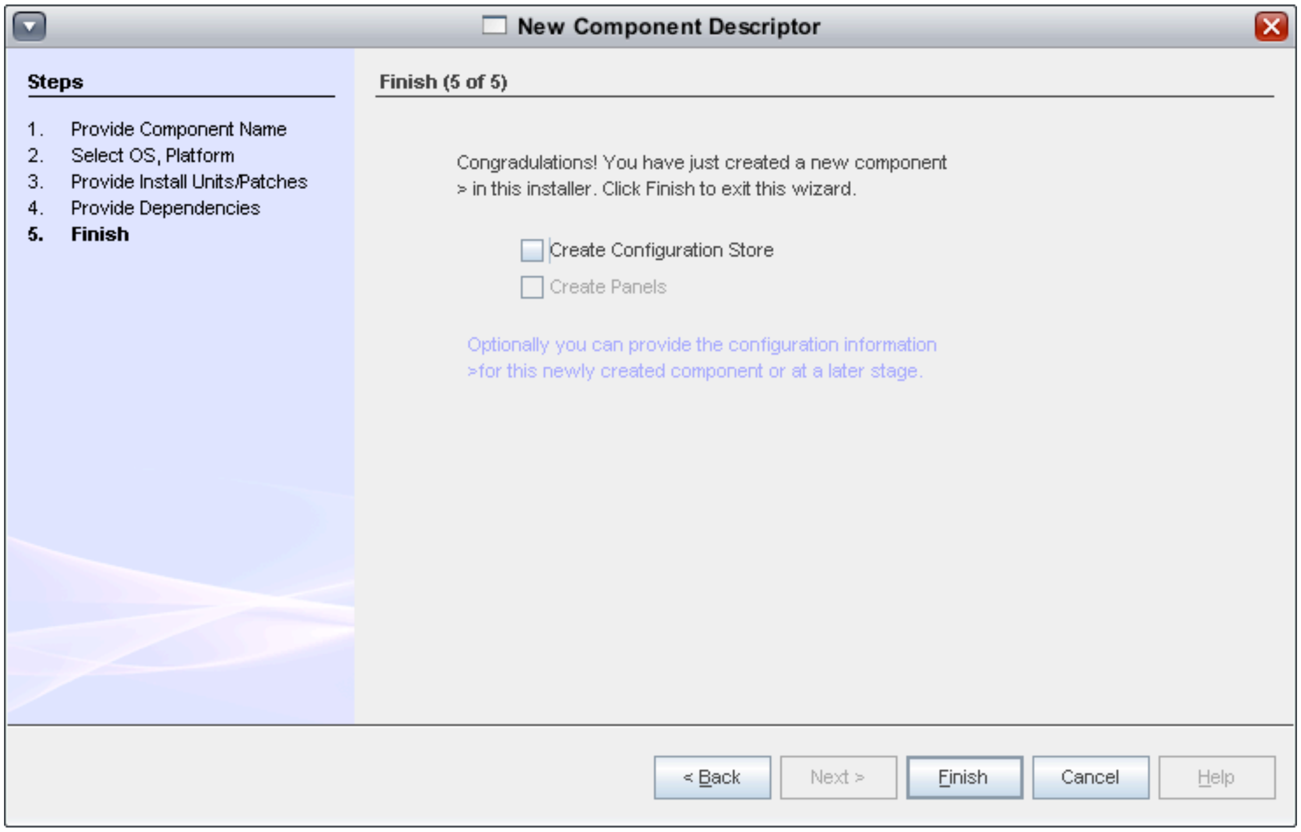
Figure 7. Fourth panel in New component wizard.

Finish Panel

This is the last panel in the new component wizard. This shows a completion message to the user that the new component would be created. It also provides optional features to create a new data model and/or new panel view for the new component just created. These optional features will automatically start the new data model wizard and new panel view wizard (one after another) when the new component wizard completes. Please note that this is 'nice to have' and if there is a pressing demand for this, we can implement this fully. This can help the user to follow a definite path in creation of new installer.

The following figure 8 shows the screen shots of the current implementation of this final panel. There are two optional features represented by two check boxes. The second check box is disabled because, when creating a panel view openInstaller requires that there should be a data model to back it up. This check box will be enabled when the user selects the first option of creating data model.

After the New component wizard completes successfully, the component descriptor file is created on the disk with all the user inputs included. The component file is also opened automatically in the editor for further editing. The project logical view gets refreshed to show the newly created component under the **Components** tree node.


New Component Descriptor

Steps

1. Provide Component Name
2. Select OS, Platform
3. Provide Install Units/Patches
4. Provide Dependencies
5. **Finish**

Finish (5 of 5)

Congradulations! You have just created a new component
> in this installer. Click Finish to exit this wizard.

☐ Create Configuration Store

☐ Create Panels

Optionally you can provide the configuration information
>for this newly created component or at a later stage.

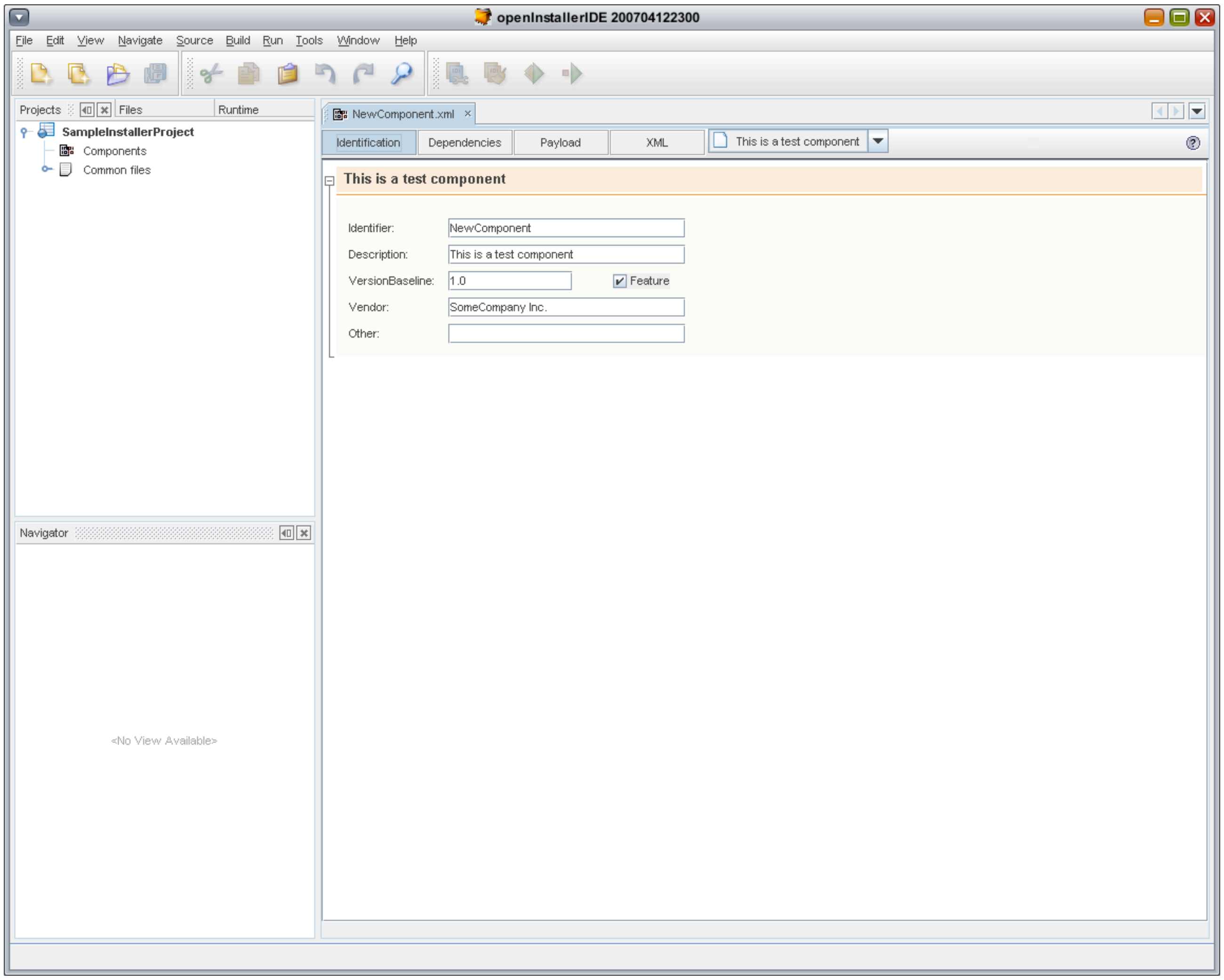
Figure 8. Fifth and final panel of New component wizard.

4.2 Editing the components

Editing of the component is done using a customized editor. This customized editor is implemented using XML multiview API of Netbeans IDE. For more information on this API, please refer to the appendix section on the same. For all XML files, this type of customized editor will be implemented in the tool.

The component editor is split into different groups. (see Figure 9 below) There's 'Identification' group, there's 'Dependencies' group and there's 'Payload' group. Each group allow editing of the respective sections in the underlying descriptor XML. The current version of the component editor is as shown in the following figure 10.

The identification group allows editing of the component identification information such as component name, its version, its vendor and so on. As described in the Appendix section for XML Multiview API, the valid-ness of the user data entered on these editors are always checked against the XML schema constraints. Also using API we can easily implement the custom validations for the user entered data using in-line validation scheme for all the required fields. XML multiview API is quick enough to show the errors on the focus lost event of each of the field for which the in-line validation is written.



This is a test component

Identifier: NewComponent

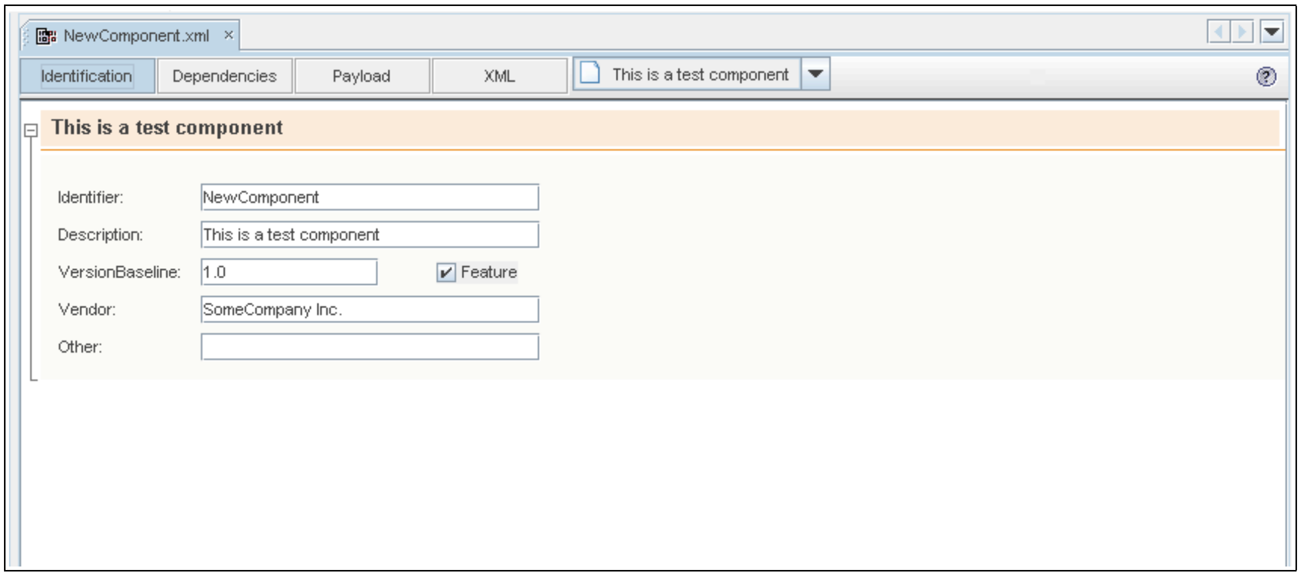
Description: This is a test component

VersionBaseline: 1.0 ✓ Feature

Vendor: SomeCompany Inc.

Other:

Figure 9 : Component Editor



This is a test component

Identifier: NewComponent

Description: This is a test component

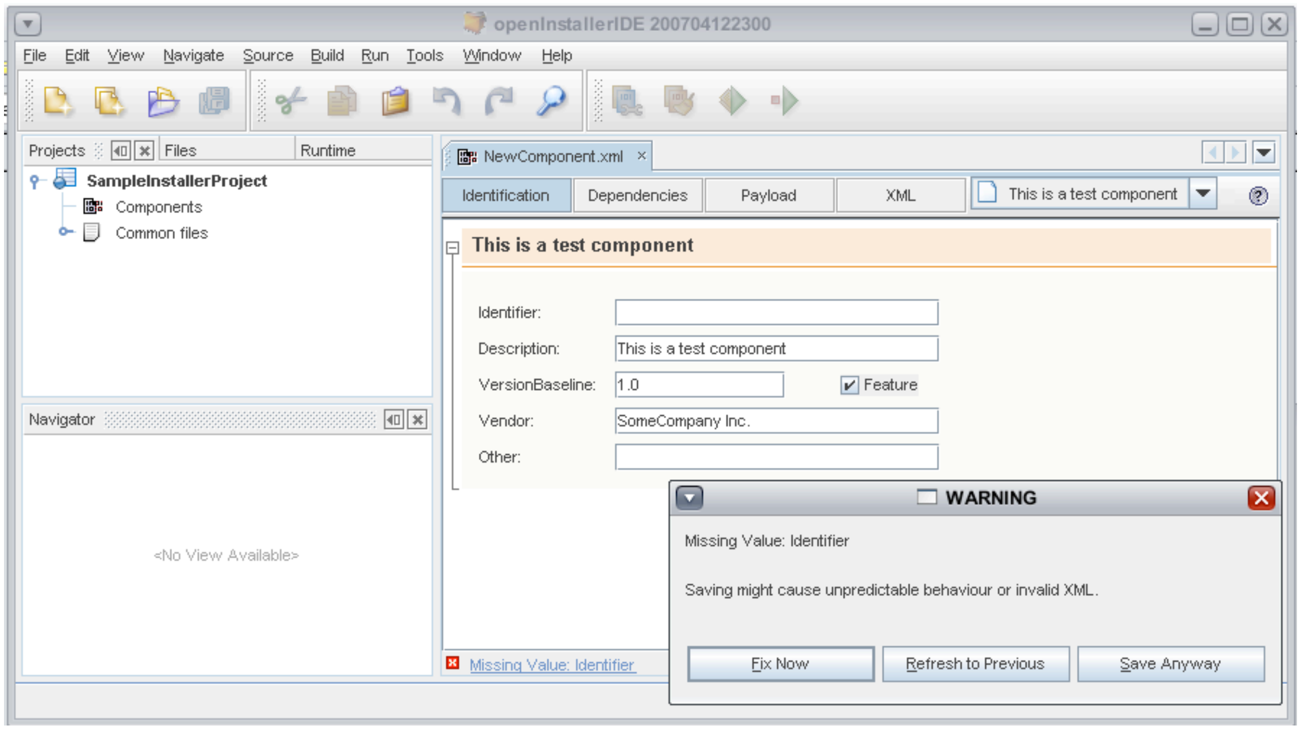
VersionBaseline: 1.0 ✓ Feature

Vendor: SomeCompany Inc.

Other:

Figure 10 : Component Identification Editor

The following figure 11, shows how in-line validation in action as per the current implementation.



This is a test component

Identifier:
Description: This is a test component
VersionBaseline: 1.0 [✓ Feature]
Vendor: SomeCompany Inc.
Other:

WARNING

Missing Value: Identifier

Saving might cause unpredictable behaviour or invalid XML.

Missing Value: Identifier

Figure 11 : In-line validation showing the error message in the status bar and a popup dialog.

The above figure 11, shows how the errors are handled in default way. (provided by the API) we need to extend the API and provide custom messages for each of the custom validation that we implement using the API. The error dialog is automatic and it pops up when the user goes away (focus lost) from the editor. This will prevent the user making any more errors and he cannot continue without fixing this error. This kind of error notification is used for mandatory fields such as component name as shown in the above figure 11.

The next group allows 'Dependency editing' for the component. The current implementation shows the component dependencies in a graphical way as shown in the figure 12 below. This is done using the Netbeans Visual Library 2.0 API. For more information on this API please refer to the Appendix section on the same.

The dependencies can be added or removed from the editor and can be dragged from the project view on the left and dropped into the editor area to create a new dependency. We can edit the dependencies by directly double clicking on them. This will open the component editor for that dependency.

The third group allows editing of the payload section in the component descriptor. This has the information about the Operating system, architecture support and individual OS related Install units that contain the product binaries. This editor part is shown in the following figure 13 below.

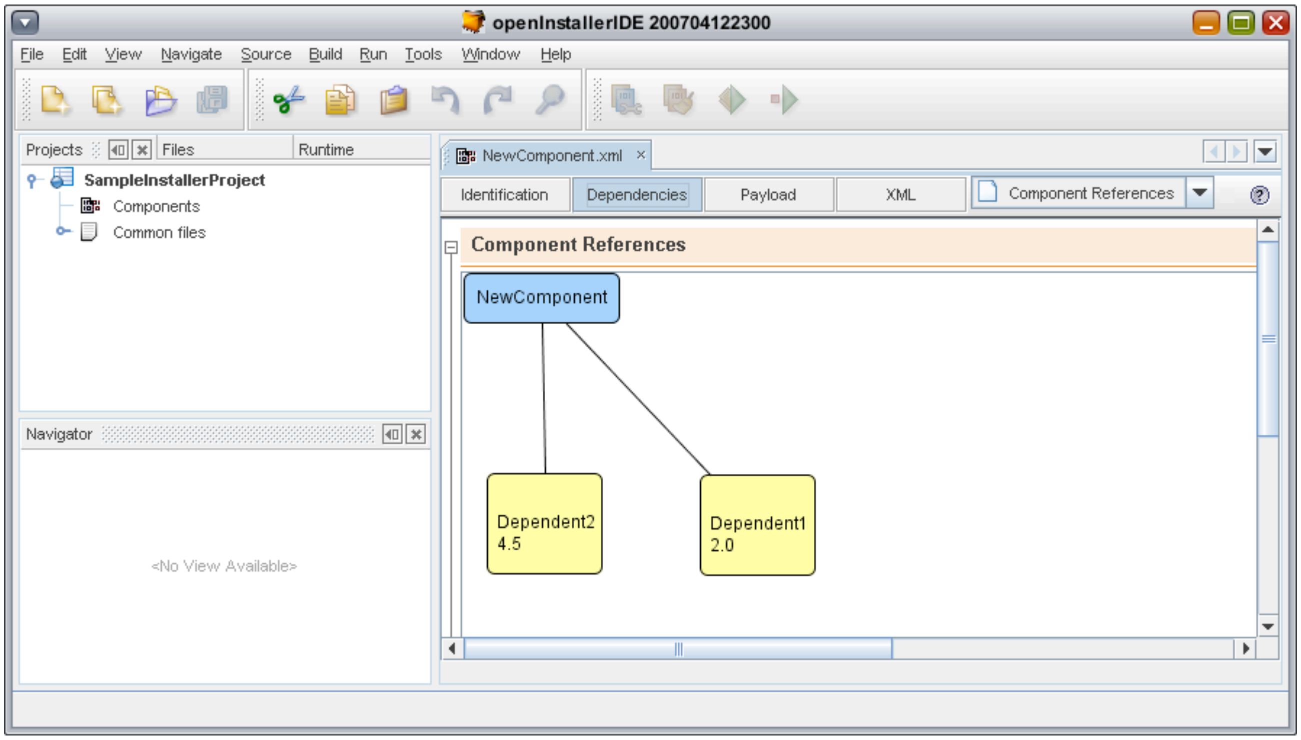
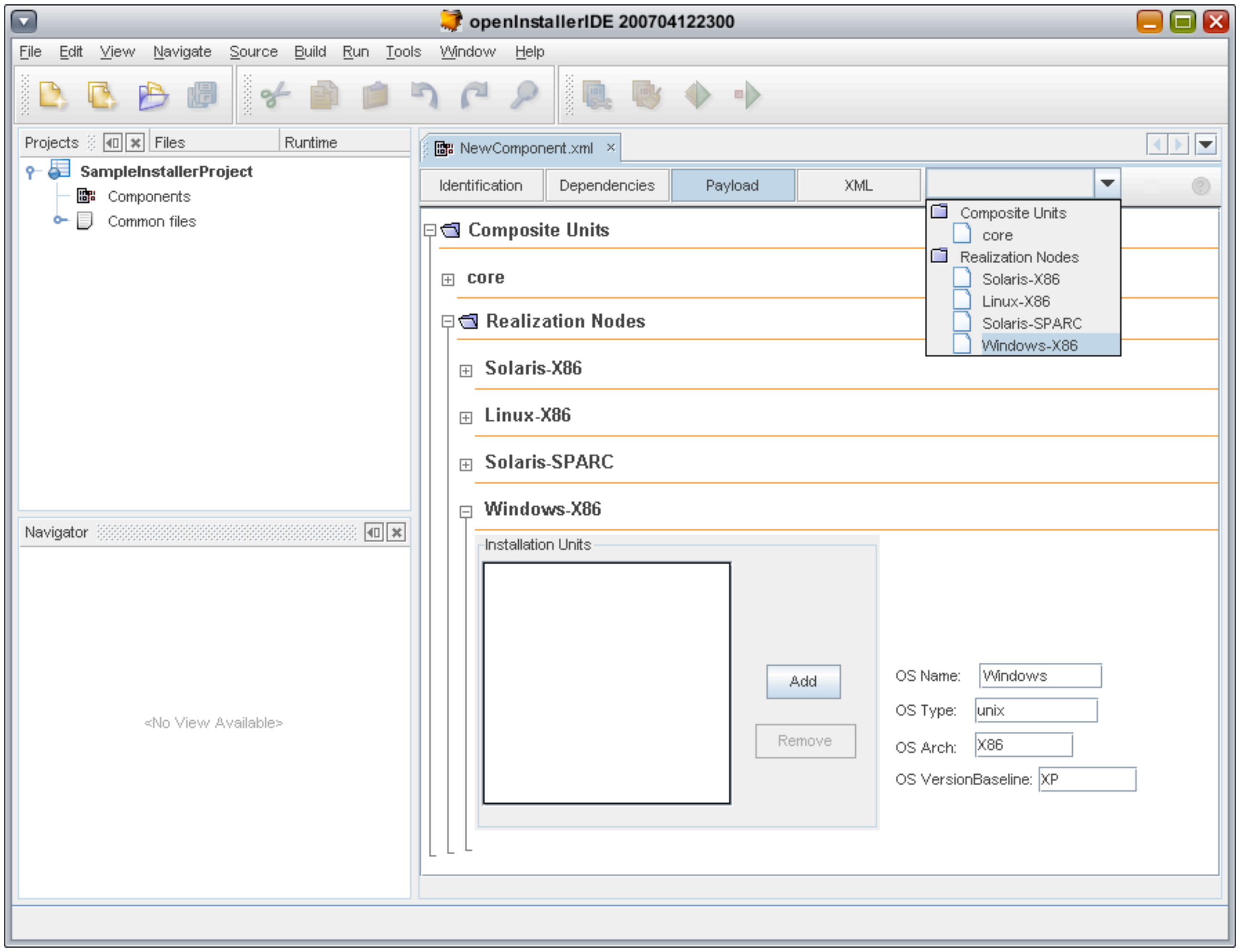


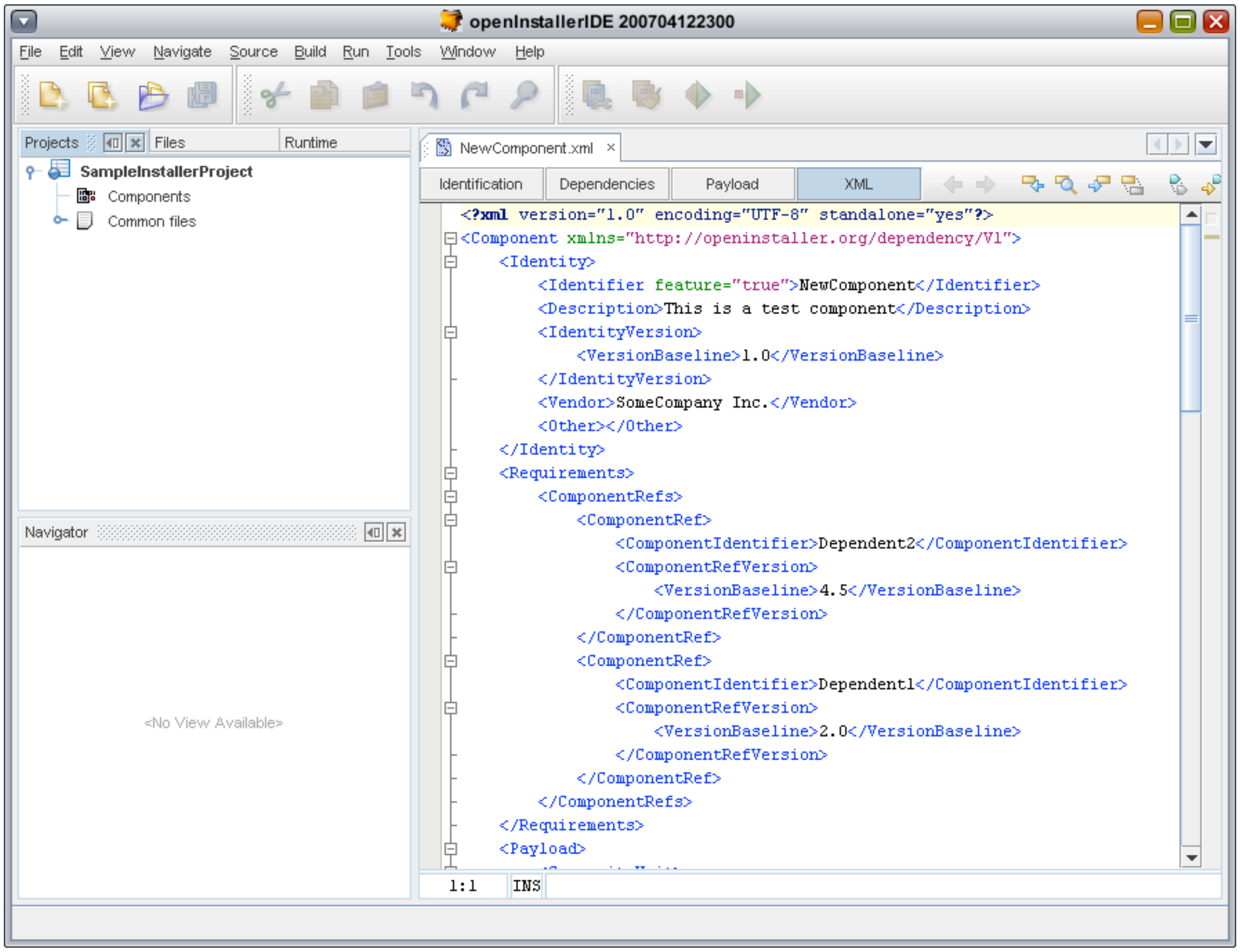
Figure 12 : Component editor showing dependencies



OS Name: Windows
OS Type: unix
OS Arch: X86
OS VersionBaseline: XP

Figure 13 : Component editor showing Payload section

Finally each XML multiview editor comes by default with the file view which allows raw file editing. For component editor, this view is as shown in the figure 14 below:



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Component xmlns="http://openinstaller.org/dependency/V1">
    <Identity>
        <Identifier feature="true">NewComponent</Identifier>
        <Description>This is a test component</Description>
        <IdentityVersion>
            <VersionBaseline>1.0</VersionBaseline>
        </IdentityVersion>
        <Vendor>SomeCompany Inc.</Vendor>
        <Other></Other>
    </Identity>
    <Requirements>
        <ComponentRefs>
            <ComponentRef>
                <ComponentIdentifier>Dependent2</ComponentIdentifier>
                <ComponentRefVersion>
                    <VersionBaseline>4.5</VersionBaseline>
                </ComponentRefVersion>
            </ComponentRef>
            <ComponentRef>
                <ComponentIdentifier>Dependent1</ComponentIdentifier>
                <ComponentRefVersion>
                    <VersionBaseline>2.0</VersionBaseline>
                </ComponentRefVersion>
            </ComponentRef>
        </ComponentRefs>
    </Requirements>
    <Payload>
```

Figure 14 : Component editor showing the default XML file editor

This file view editor gets synchronized automatically with the visual editors for each XML section.

5 Feature : Three click Installer

5.1 Introduction

This feature describes how to generate an Installer based on openInstaller for any Java based stand alone Netbeans project from within Netbeans IDE itself. This is a quick way to automatically create deployable installers for Netbeans projects.

5.2 Why we need it

The Netbeans IDE and platform support al most all aspects of stand alone application development except creating an installer distro for the application. This is a win win opportunity for both openInstaller and Netbeans community.

5.3 How it works

Its simple extension to the developer tools. The netbeans plug in designed for developer tools extend little to support this feature.

In this, a global option, in the form of a tool bar button and a menu item in the 'Build' global menu is added. Also a context-sensitive pop up menu item is added in the pop up menus of supported Netbeans projects. All options invoke the same function : Generate installer.

Initially, we will support **Netbeans module suite** applications. (ant based or otherwise), who generate a jar based or zip based release bundle as part of their build process inside the IDE. Later the support is extended for other types of projects.

The developer tools plug in when invoked through one of the options stated above, examines the selected Netbeans project metadata and attempts to gather information about the project. The information gathered include (but not limited to), project name, location of the built release bundle. If these information is not available or the developer tool plug in could not determine them for some reasons, the information is gathered with the help of user.

The developer tools plug in presents a wizard that collects answers to simple questions as follows:

1. On the first screen the wizard collects the project identification information such as its name and version. It also helps users to provide license text on the wizard panel. There can be one more question on the path to place the generated installer (TODO), but can be same as release folder of this Netbeans project.
2. If the developer tool not able to detect the release bundle for some reason, the second wizard panel will be provided to collect and record the path of the same. If this data is already collected automatically then this panel is not shown and the first panel exits the wizard.
3. On completing the wizard, the developer tools create the required files that make up the installer and bundles everything and creates the distributable zip archive. What actually happens here is as follows:

   This option creates an empty Installer Project on the disk and creates and updates the required files and calls the assembly process to create the installer. The outcome will be a zip archive with all the necessary openInstaller runtime included in it. This archive will be self ready to install the Netbeans platform based installer on any system.
4. Currently Netbeans IDE provides ways of generating a zip archive for the module suite within the IDE itself. We need to hook it up with generating the installer after the zip archive is prepared using the IDE in built functions. This requires defining a new dependency on the 'Netbeans Module Projects' API (which is non-public and hence we need to depend upon implementation version).
5. User/Netbeans developer will have to host this zip archive as a downloadable bundle (or any other means of getting the download bundle) and anyone who wants to install the application, he needs to download, unzip and launch the installer.

6 Feature : Assembling Installer

This feature takes all contents of installer project and creates a distributable installer. This is usually the final phase in writing an installer. This feature gets called after the 'Integrity check' feature when its implemented and included in the tool.

6.1 Inputs assumed for this feature

This feature needs an Installer Project already created and available in the Netbeans IDE. The installer project should be complete in all respects in the sense that it should have all the files with all relevant data filled in. The 'Integrity check' feature, when included in the tool will check for any validation/functional errors in the installer files and will report to the user about any errors. So the assembly process assumes that the installer project as a whole is in valid state to build the installer.

6.2 How it works

The assembly is being written as an **ant** build script for two reasons :

1. Its efficient to call/maintain from the Netbeans IDE.
2. It can be integrated easily to the existing module suite build script to combine the zip archive generation with the assembly process. This is for the 3 click generate installer feature.
3. It can be easily scriptable outside the IDE if required in future.

The build script reads the project properties (installerProject.properties) to find out the project layout details. We would like to maintain this interface for the assembly because, in future, we would be adding support for installer project customization and that includes changing some parts of installer project layout. But all the project layout changes are recorded in the project properties file. So its safe to assume that this file contains the actual layout settings.

The assembly at a high level creates what is known as the 'media layout' on the disk. This media layout will be the final installer image layout. (this is how the user sees the installer when they unzip the installer bundle). We initially decided to follow what openInstaller sample product installers have. Later we might want to change this. The build script copies/creates layout folders and files as required and prepares the media layout.

When the assembly is done, what we will have is the distributable installer bundle.

6.3 Adding JNLP to assembly

When we add the JNLP support for the installer project (the feature that makes generation of JNLP descriptor for installer so that it can be used to distribute the product installer through web) the assembly process will include the generation of the JNLP descriptor in the build script. User will just choose whether they also want to generate JNLP descriptor when they choose to generate installer.

7 Feature : Installer Integrity check

This feature will allow validation and integrity of the installer that would be generated out of an Installer project. There would be various checks (in terms of rules, for example) that are executed on the installer project files to determine at any point of the development time, whether the installer (to be generated) is valid or it has errors.

7.1 How it works

These checks take off where the in-line validation (attached to the new file wizards and file editors) stop. The in-line validations basically catch the syntax errors and usually are driven by the XML schema constraints. But the integrity checks look for semantics and other features that make up the installer. One good example is : Looking for missing files when they are specified somewhere else, like a view file refers to a data model file but the data model file isn't exist in the installer project. Another example would be to look for valid data paths.

The mechanism to define and execute checks is still TBD. Mostly it would be like an Expert system and rule engine based, with rules defined for each checks. The integration of such mechanism to Netbeans is also still TBD.

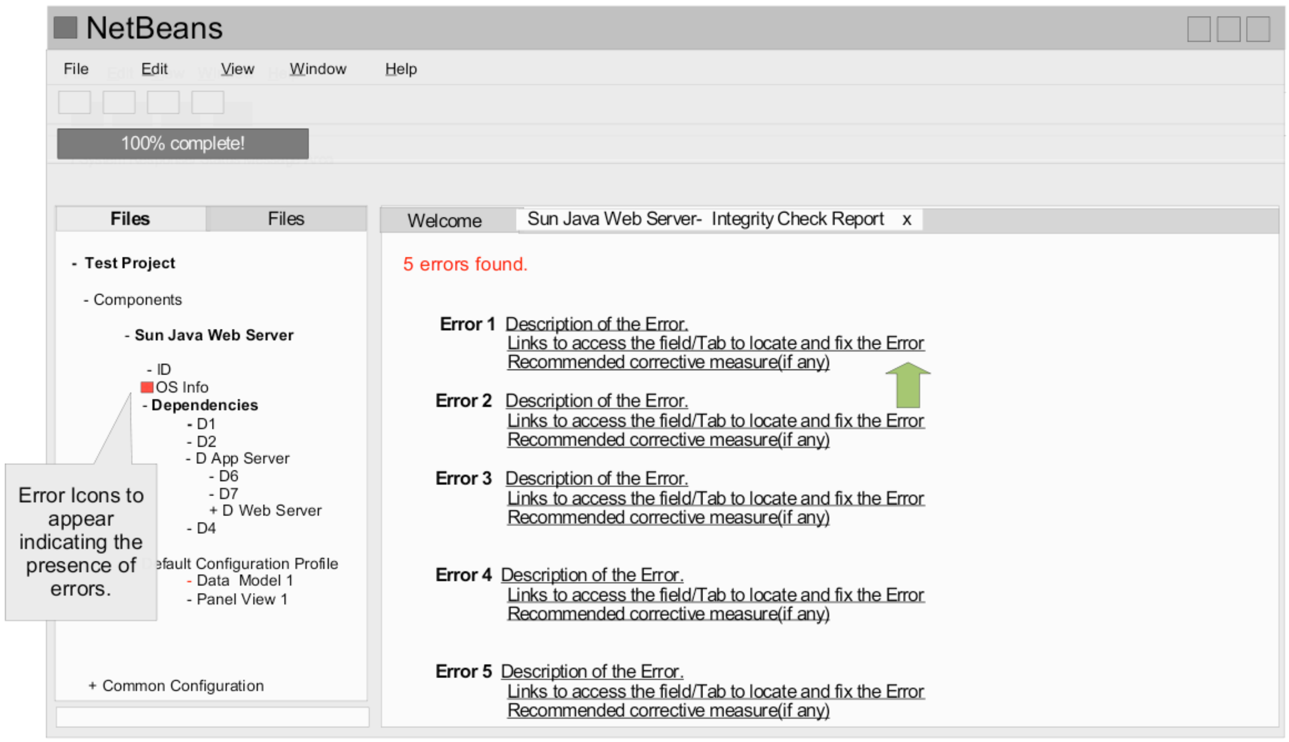
7.2 Current state

The UI expert recommendation for how this integrity check display errors is shown below.

We can display the errors (with clickable links, as shown in the below UI design, figure 15) in the Netbeans output window to align with how Netbeans IDE does display compilation errors for Java files.

All the clickable links when clicked, open up the required files in the editors possibly highlighting the error location/field and may provide a hint to correct the error where its possible to include the hint.

Installer developer can use this integrity check to verify the installer they are building at any time during their development cycle and correct/rectify the errors as they go forward. If there are errors reported in the final integrity verification before the generation of the installer, then the installer generation will be abandoned and user will be presented with the error display and help them to correct/rectify the errors. Unless there are errors found, the installer generation would continue to generate the installer.



5 errors found.

Error 1 Description of the Error.
Links to access the field/Tab to locate and fix the Error
Recommended corrective measure(if any)

Error 2 Description of the Error.
Links to access the field/Tab to locate and fix the Error
Recommended corrective measure(if any)

Error 3 Description of the Error.
Links to access the field/Tab to locate and fix the Error
Recommended corrective measure(if any)

Error 4 Description of the Error.
Links to access the field/Tab to locate and fix the Error
Recommended corrective measure(if any)

Error 5 Description of the Error.
Links to access the field/Tab to locate and fix the Error
Recommended corrective measure(if any)

Figure 15 : Installer Integrity check results UI design

Appendix

1 Netbeans Wizards in this project

This chapter has information about using Netbeans wizards in this openInstaller Developer Tools project. The Netbeans wizards have sophisticated APIs that helps to create wizards very quickly. There are lot of useful methods for controlling the flow of the wizard panels and providing validation. As the wizards are used in many places, it makes sense to abstract the API in a single place and customize.

There is a delegation mechanism used to abstract the common functionality into base classes. These classes include common functions like creation of the files using the user input data, opening of the newly created file in the editor and refreshing the project logical view to show the newly created file under the project node as required.

The base classes are included in a separate helper Netbeans module. These classes implement the basic validation mechanism using the Netbeans Wizard API and provide a simple interface to the New file creation wizards in the project. All New file creation wizards (for example, new component wizard, new data model wizard etc) extend these base classes.

2 XML Multiview API in this project

This project uses this API to implement various editors for different types of XML files. The API is used in implementing many editors in J2EE cluster of Netbeans IDE. Examples of these are the editor for web.xml, editor for ejb xml and editor for WSDL files.

This API allows to build a highly customizable and robust editors for any well defined schema or DTD based XML files. Although this API is extensively used in J2EE cluster, the API is not yet declared as public. This XML Multiview API is currently defined as 'friendly' API[5]. There is an issue to make this API public (Issue No. 107858), but there are not enough votes to resolve this issue at this time. In future it is expected that, this API would become public. For the tool, we think that the current features and sufficient to implement the editors required for all XML files in the installer project.

The API provides robust two way synchronization of the data between the visual editor and the file view. It also provides APIs for handling validation errors based on the schema constraints. There is a standard error notification and correction mechanism in-built into the API. At any point of time, the XML file wont be saved to disk with semantic errors, always the XML file is checked for its well-formedness and validness based on the schema before the user changes are saved.

Initially when we started looking at this API, there were very few help and documentation

5 In Netbeans, there are three different types of APIs depending on the access type. Some are **public** and any module developer can use them freely and those API come with assurance that all future revisions to the APIs are compatible. Some are defined as **friendly** APIs, which of course module developers can use them but however, the future modifications to the APIs are not guaranteed to be compatible. And finally there are some APIs which are **private** and no external module developer can use or depend on them.

available. Now there is some considerable help in my blog and in Geertjan's blog.

3 Netbeans Visual Library 2.0 in the project

This API is a public API (refer the footnote in previous section for what this means) in Netbeans 6.0 IDE. This API allows to create any shape or type of graphs and tree structures easily in Java. The API also provides easy ways of adding motion, selection, zooming and animations to the graph/tree elements.

This API is extensively used in many Netbeans platform based applications and as well as in Visual Web Development cluster which is part of Netbeans Enterprise cluster distribution and allows creation of page flows for JSF pages in a web application.

As this API is not yet made available for Netbeans 5.5.x, we are carrying our own copy of the API along with our module suite.

Glossary

This section lists out (with definitions and explanations) the important terms used in the document.

1. openInstaller : is an Installer framework that help to build installers for individual products or group of products or product stacks. It is an open source project on the java.net and part of the glashfish community. The home page is here : www.openInstaller.org
2. Component : A single product that is included in the installer based on openInstaller.
3. Descriptor : The file which stores component information like its name, version, OS support and install units.
4. Model (a.k.a. Data model): The file which stores the configuration variables information for a component.
5. View (a.k.a. Panel or page view): The file which stores the installer panel information for a component. This view is connected to a model file and the panel fields map to individual configuration variables.
6. Page sequence : The file which stores the installer panel sequence (or order) and has the information about when to show/skip the panels depending upon conditions.
7. Launcher Scripts : Platform specific binaries which run the installer.
8. Installer Layout settings : A set of files which store the customization made to the installer wizard layout.
9. User preferences : A set of files that store the user preferences for the installer. This file currently also stores the license information for the installer.

1 www.openInstaller.org
2 Developer Tools Use case Document : http://wiki.glassfish.java.net/attach/OpenInstallerDeveloperTools/oiDev.ToolsUsecaseDocument.odt
3 https://openinstaller.dev.java.net/files/documents/6533/57967/openInstaller_Developer_Workflow_V1.0.odt