NetBeans RCP

By John Kostaras

Open Conference Crete 2012

Agenda

* Introduction
* Module System
  * Lookups
* Window System
  * Explorer Views
  * Nodes
* Action System
* ~~Dialogs and Wizards~~
* ~~Options & Settings~~
* ~~FileSystem & DataSystem~~
* ~~Visual Library~~
* ~~Internationalisation~~
* ~~Help System~~
* Hands on

Prerequisites

* NetBeans 7.2 (http://www.netbeans.org)
* Todo app (http://netbeans.org/community/magazine/code/nb-completeapp.zip)

Introduction

What is NetBeans Rich Client Platform?

* Set of APIs and Libraries
* Framework and Module system
* Generic Desktop Application
* Platform modules + IDE Modules = NetBeans IDE
* Platform modules + your modules = Your Application
* Provides services that almost all Desktop Applications need
* Saves development time
* Allows you to concentrate on the actual business logic
* NetBeans RCP ⇔ Application server for the desktop
* Improves development and maintainability through modularization

Benefits

* Shorter development time
* Consistent User Interface
* Updateable (Update center / Webstart)
* Platform Independence (Swing + platform specific launchers)
* Modularity

Hands-on

TodoRCP

* A todo list application:

"A complete App using NetBeans 5" by Fernando Lozano

http://netbeans.org/download/magazine/01/nb01_completeapp.pdf

http://netbeans.org/community/magazine/code/nb-completeapp.zip

* 1st step: UI prototype
* 2nd step: User interaction & event handling
* 3rd step: Persistence & validation logic

1st Step: UI Prototype

1st Step: UI prototype Requirements

* Tasks should have a priority, so users can focus first on higher-priority tasks;
* Tasks should have a due date, so users can instead focus on tasks which are closer to their deadline;
* There should be visual cues for tasks that are either late or near their deadlines;
* Tasks can be marked as completed, but this doesn't mean they have to be deleted or hidden.

1st Step: UI Prototype Steps to follow

1. Design the tasks list window
   * **File → New Project →** *NetBeans Platform Application* in the *NetBeans Modules*
   * *Modules →* **Add New**

Modularity

Overview

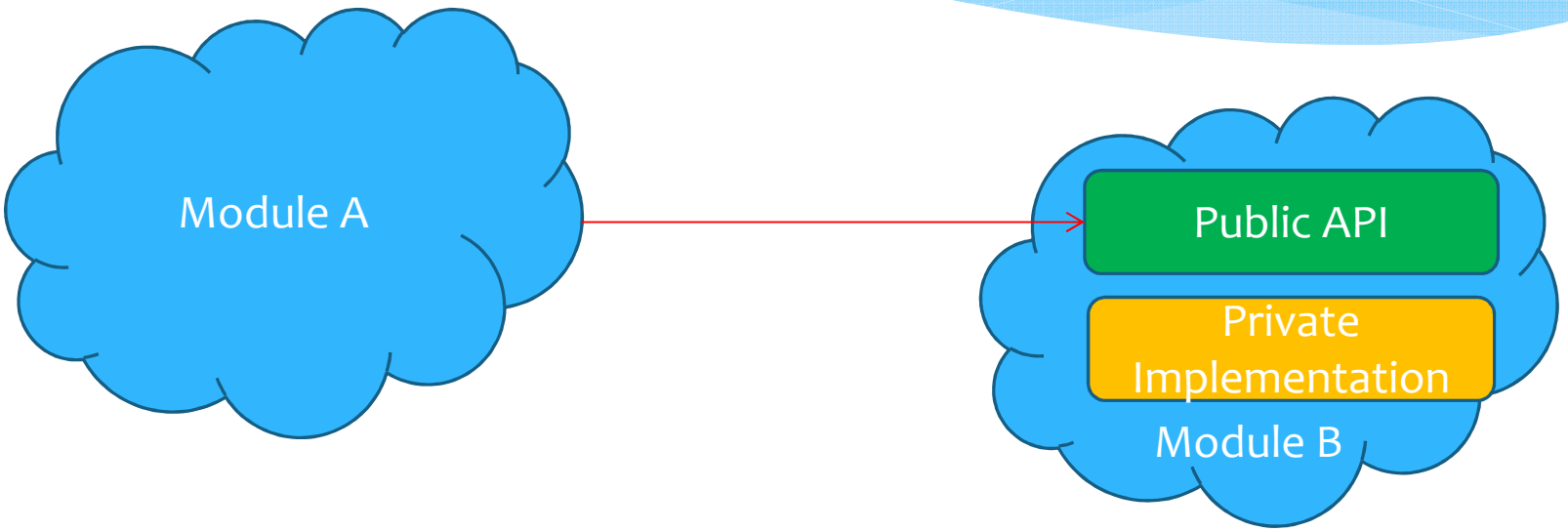
* Modules are an extension of Java packaging
* Based on OSGi
* Life cycle management
* Allow explicit import/export (information hiding)
* Dependency management (No cyclic dependencies)
* Versioning
* Dynamic Service Infrastructure (to reduce direct dependencies)
* Plugins

Overview

Module System

* Recognizes & Loads Components at runtime
* Resolves versions & dependencies
* Adds/removes/refreshes Components
* Provides
  * Information Hiding
  * Infrastructure for loosely coupled communication

Module dependencies



Window API

Overview

* TopComponents ⇔ JPanels
* Default Position (Mode) defined declaratively
* Grouping, docking, sliding, dragging
* Life cycle (opened, showing, closing, closed…)

Window System

* Window System is responsible for display and management of windows
* Typical feature requests:
* Persistent resizing, closing and moving
* Docking and undocking
* Minimizing (sliding)
* Persistence (State, Layout)
* Plugins should be able to add new Windows
* Selection Management

Main Classes

* TopComponent
* Mode
* TopComponentGroup
* WindowManager

TopComponent

* Base unit of the WindowSystem
* Extends JComponent
* Always displayed inside a Mode
* Managed by WindowManager
* Can react on Lifecycle Events

Mode

* A Container inside the Window System
* Can display multiple Windows in a tabbed view
* Created by WindowManager
* Described by a XML file
* Controller of contained TopComponents
* Resizeable

TopComponentGroup

* Windows can be opened as groups
* Typical in Document centric applications (main document + detail views, project views,...)
* Complex logic

WindowManager

* Creates and Manages Modes, and Groups
* Has methods for finding it's components

Hands-on

1st Step: UI Prototype Steps to follow

1. Design the tasks list window
   * New → Window

```
TasksTopComponent.java
TasksTopComponent.form
TasksTopComponentSettings.xml
TasksTopComponentWstcref.xml
```

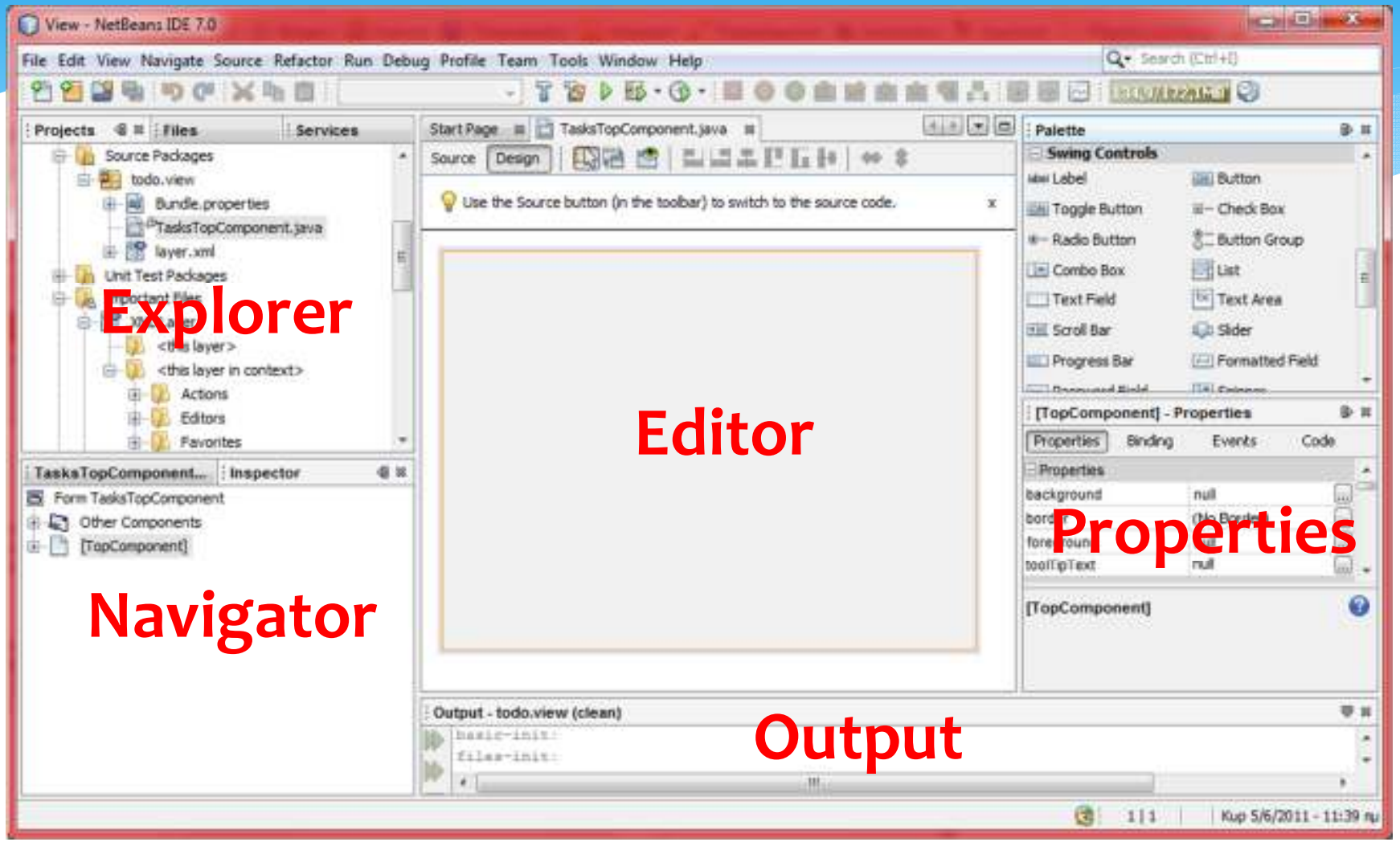
Defines how an instance of TasksTopComponent is created

ties the TasksTopComponent to the particular Mode, and sets it's initial state

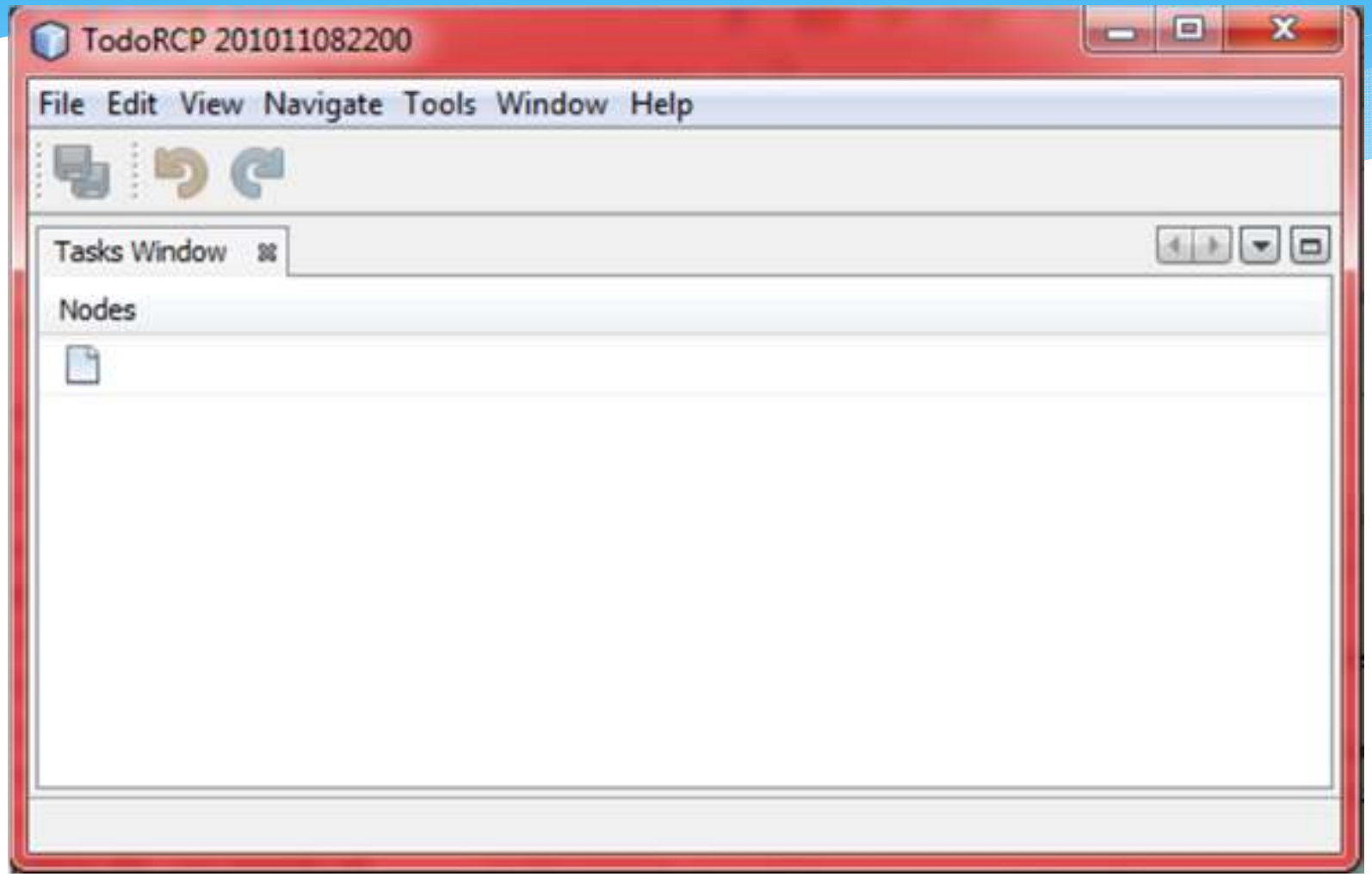
TasksTopComponent

```
@ConvertAsProperties(dtd = "-
//todo.view//Tasks//EN", autostore = false)
@TopComponent.Description(preferredID =
"TasksTopComponent",
//iconBase="SET/PATH/TO/ICON/HERE", persistenceType
= TopComponent.PERSISTENCE_ALWAYS)
@TopComponent.Registration(mode = "editor",
openAtStartup = true)
@ActionID(category = "Window", id =
"todo.view.TasksTopComponent")
@ActionReference(path = "Menu/Window" /*, position =
333 */)
@TopComponent.OpenActionRegistration(displayName =
"#CTL_TasksAction", preferredID =
"TasksTopComponent")
```

Positioning (Modes)



TodoRCP


TodoRCP 201011082200
File Edit View Navigate Tools Window Help
Tasks Window
Nodes

Persistence Modes

Persistence

* Last state of TopComponent is saved (opened/closed) depending on persistence type
* Skeleton Code for persisting form data created by wizard

Persistence Modes

* PERSISTENCE_ALWAYS
* PERSISTENCE_NEVER
* PERSISTENCE_ONLY_OPENED

Lifecycle

`componentOpened()` → open (≠ visible)

`componentShowing()` → visible

`componentActivated()` → has input focus

`componentDeactivated()` → loses input focus

`componentHidden()` → not shown anymore

`componentClosed()` → has been closed

WindowManager

* `findTopComponent(String id)`
* `findMode(String id)`
* `findMode(TopComponent tc)`
* `findTopComponentGroup(String name)`
* `getMainWindow()`
* `getRegistry()`

The Swing way

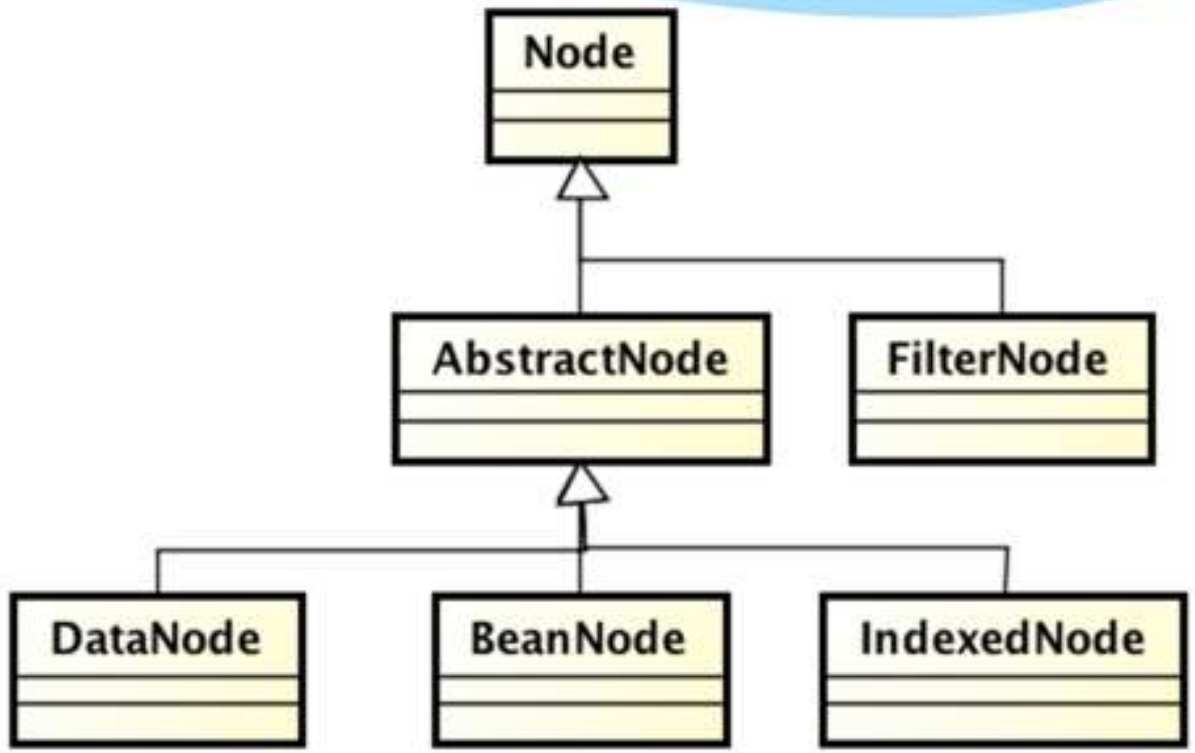
* Different Models for different views
  * TreeModel
  * ListModel
  * ComboBoxModel
* Different Renderers
  * ListCellRenderer
  * TableCellRenderer

The RCP way

* ExplorerManager (Model-Controller)
* Node (Model wrapper)
* Views

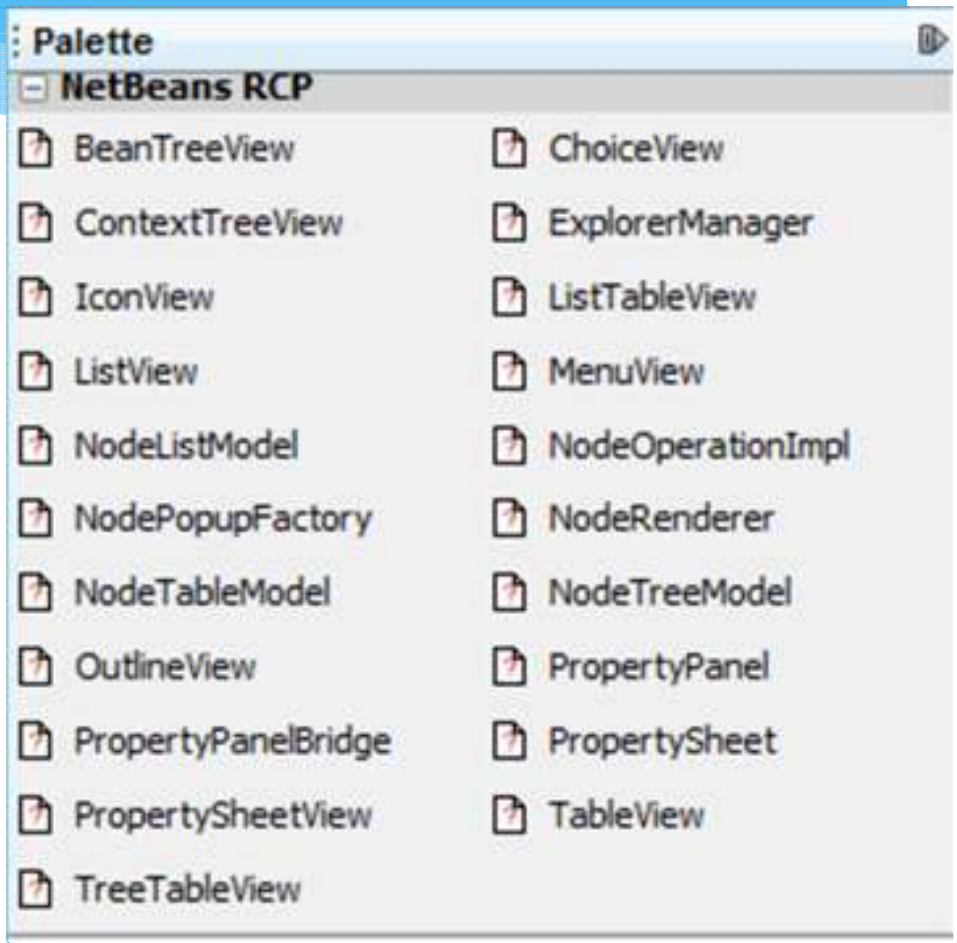
Nodes

* Nodes are the *presentation layer*
* Nodes are Hierarchical
* Nodes ***wrap*** Objects and make them UI friendly



Explorer Views

* BeanTreeView
* ContextTreeView + ListView
* IconView, MenuView
* ChoiceView
* OutlineView
* PropertySheet



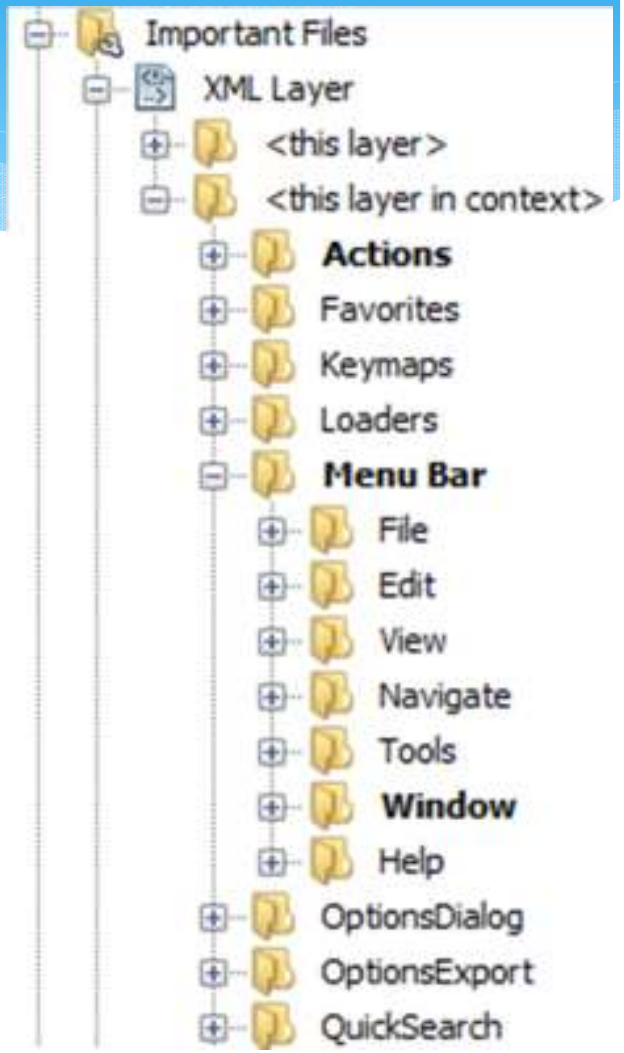
Views' Features

* Most Views are `JScollPanes` with preset View-Component
  * Except `ChoiceView`
* You don't set the Model of a view yourself
  * View searches it's model via the Swing-Component-Hierarchy (`ExplorerManager.Provider`)
* Views use Node-Wrappers for visualising nodes
  * For Optimization Reasons that's important when writing custom renderers.

Hands-on

layer.xml

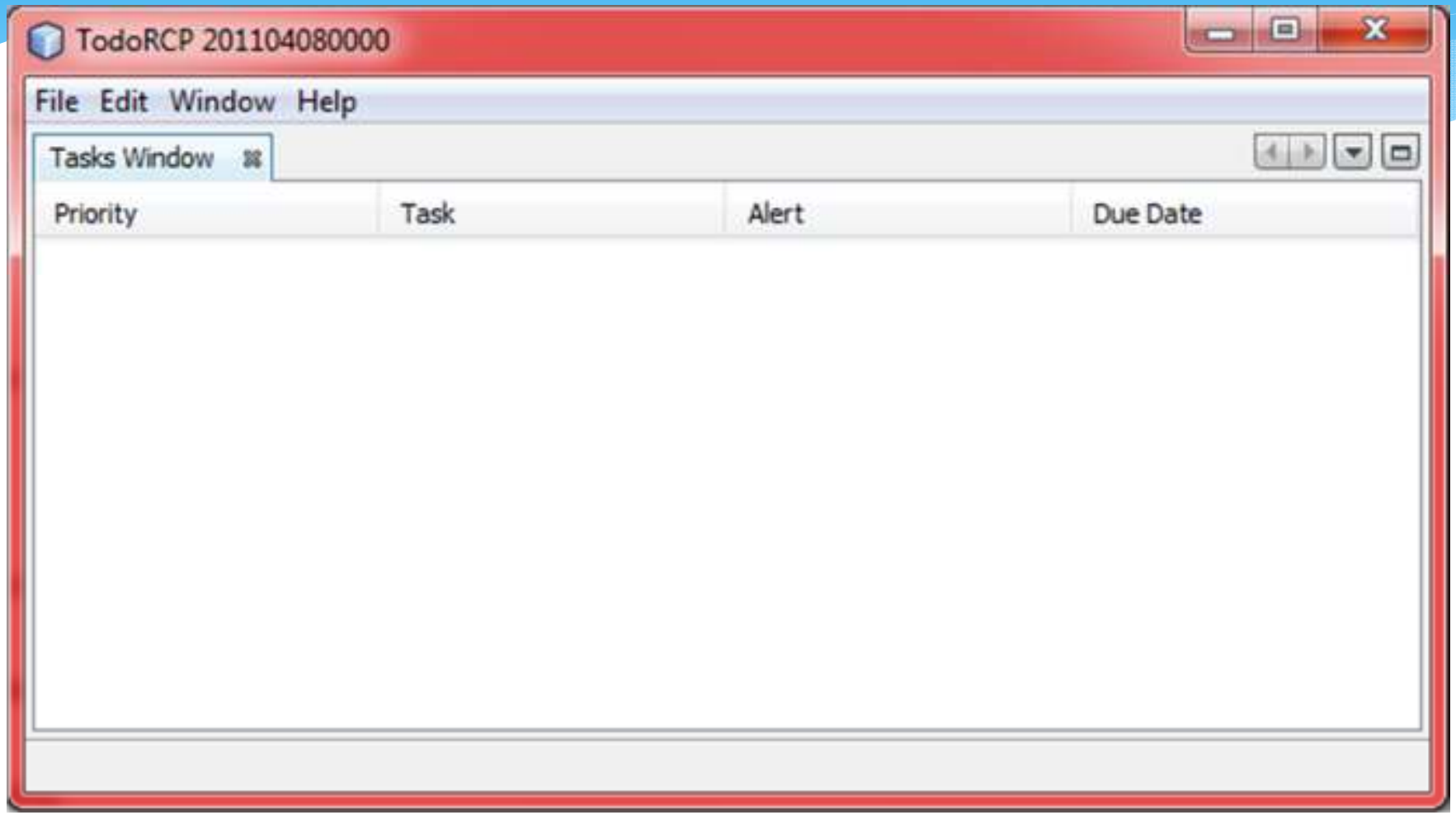
* *Important Files -> XML Layer -> <this layer in context> -> Menu Bar*
* The XML Layer is a file named `layer.xml` and each module has one. NetBeans RCP combines all `layer.xml`s during runtime and creates what is called the *central registry* of the application.



OutlineView

```
OutlineView ov = (OutlineView)outlineView; //Set
the columns of the outline view,
//using the name of the property
//followed by the text to be displayed in the
//column header:
ov.setPropertyColumns("priority", "Priority",
"description", "Task", "alert", "Alert",
"dueDate", "Due Date"); //Hide the root node,
since we only care about //the children:
ov.getOutline().setRootVisible(false);
ov.getOutline().getColumnModel().removeColumn(ov
.getOutline().getColumnModel().getColumn(0));
```

TodoRCP


TodoRCP 201104080000
File Edit Window Help
Tasks Window
Priority
Task
Alert
Due Date

Hands-on
TaskNode

```
package todo.view;
import java.beans.IntrospectionException;
import org.openide.nodes.BeanNode;

class TaskNode extends BeanNode {
  public TaskNode(Object bean) throws
IntrospectionException {
    super(bean);
  }
}
```

Children

* Nodes can have Sub-Nodes
* Sub-Nodes are managed via special Container-Objects: `org.openide.nodes.Children`
* Children is **Container** and Node-**Factory**
* Different implementations for different purposes
* `Children.Keys` (extends `Children.Array`)
  * Asynchronous creation via `ChildFactory`
  * `Children.SortedMap` (extends `Children.Map`)
  * `Children.SortedArray` (extends `Children.Array`)
* Nodes without Sub-Nodes return `Children.LEAF`

ChildFactory

* Finding and creating objects for Sub-Nodes can take a while
  * e.g.: Files on a remote server
* Better to create Children-Objects asynchronously
* `ChildFactory` is a Factory, that can create the data (keys) threadsafely and asynchronously
* Static method `Children.create (ChildFactory, asynchronous)` creates a Children-Container from a ChildFactory
* Beware: Creation of Nodes happens on Event-Dispatcher Thread again (GUI Blocking!) => must be fast!
* Use `ChildFactory` instead of `Children.Keys`
  * Doing this it's easy to switch to asynchronous creation when needed

Hands-on TaskChildFactory

```
package todo.view;
class TaskChildFactory extends ChildFactory<Task>{
  @Override
  protected boolean createKeys(final List<Task>
toPopulate) {
    final GregorianCalendar cal = new
GregorianCalendar(TimeZone.getTimeZone("Europe/Belgium"
));
    cal.set(2012, Calendar.JULY, 2, 10, 00, 00);
    toPopulate.add(new Task(1, "Hotel Reservation", 1,
cal.getTime(), true));
    return true;
  }
...
}
```

Hands-on TaskChildFactory (cont.)

```
...
  @Override
  protected Node createNodeForKey(final Task
key) {
    TaskNode taskNode = null;
    try {
     taskNode = new TaskNode(key);
    } catch (IntrospectionException ex) {
Exceptions.printStackTrace(ex);
    }
    return taskNode;
  }
```

Hands-on ExplorerManager

```
public final class TasksTopComponent extends
TopComponent
implements ExplorerManager.Provider {
  private static final ExplorerManager EM = new
ExplorerManager();
  public TasksTopComponent() { ...
    EM.setRootContext(new
AbstractNode(Children.create(new
TaskChildFactory(), true)));
} ...
@Override public ExplorerManager
getExplorerManager() { return EM; }
```

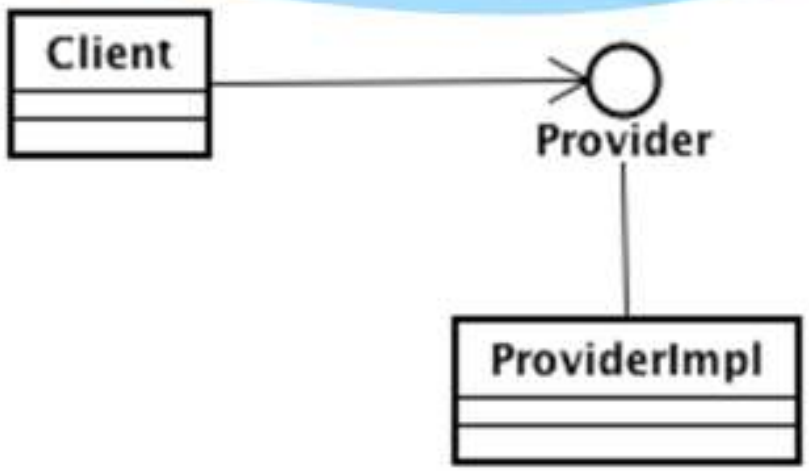
Lookup API

Problem statement

* Modules developed by independent groups or individuals must be able to communicate
  * Extensibility: Third parties can provide (alternative) implementations for services (e.g. algorithms)
  * Dynamics: Services can be added (& replaced) at Runtime
  * Loose Coupling: depend on Interface instead of Implementation

Dependency Reduction

* How does the client find the implementation?
  * Spring uses *dependency injection* or *inversion of control* via its xml files
* Java 6 uses a Query-based approach, the ServiceLoader:




```
ServiceLoader<Provider> serviceLoader =
ServiceLoader.load(Provider.class);
for (Provider provider : serviceLoader) {
return provider; }
```

ServiceLoader

The ServiceLoader has a number of problems:

* it isn't dynamic (you cannot install/uninstall a plugin/service at runtime)
* it does all service loading at startup (as a result it requires longer startup time and more memory usage)
* it cannot be configured; there is a standard constructor and it doesn't support factory methods
* it doesn't allow for ranking, i.e. we cannot choose which service to load first

ServiceProvider

* Netbeans introduces a new way to accomplish loose coupling, the ServiceProvider:

```
@ServiceProvider(service = Provider.class)
public class ProviderImpl implements
Provider { }
```

* Netbeans creates a text file `package.Provider` inside `build/classes/META-INF/services/` folder of the module which contains the fully qualified names of the implementation classes, e.g. `package.ProviderImpl`

Lookup API

* But how does the client find the implementation?
* The client looks in the *default* lookup for the interface.
* The *default* Lookup is a Lookup that evaluates the service declarations in the `META-INF/services` folder.
* It is callable through the `Lookup.getDefault()` method. By asking for a service interface in this way, you receive instances of implementing classes registered in the `META-INF/services` folder.

Lookup API (cont.)

* A lookup is a map with class objects as keys and sets of instances of these class objects as values, i.e.

`Lookup = Map<Class, Set<Class>>`, e.g.

`Map<String, Set<String>>` or

`Map<Provider, Set<Provider>>`

* Netbeans provides a number of methods to access a lookup:

```
Provider provider=Lookup.getDefault().lookup(Provider.class);
provider.aMethod();
```

* or if you have more than one implementations of Provider:

```
Collection <? extends Provider> providers =
Lookup.getDefault().lookupAll(Provider.class);
for (Provider provider : providers) { }
```

ServiceProvider pros

* dynamic, so you can plugin/unplug modules while your application is running
* it doesn't load all services at startup
* allows you to set priorities (with the position attribute), e.g.:

```
@ServiceProvider(service=Provider.cla
ss, position=1)
```

Lookup Listener

```
Lookup services = Lookup.getDefault();
Lookup.Result<Task> result =
        services.lookupResult(Task.class);
result.addLookupListener(new LookupListener() {
@Override
public void resultChanged(LookupEvent event) {
  Lookup.Result result = (Result)
event.getSource();
  Collection c = result.allInstances();
    ...
  }
});
```

Examples of Lookups

* ServiceProvider (default Lookup)
* Each Window (TopComponent) has a Lookup
* Nodes have Lookups

```
taskNode.getLookup().lookup (Task.class);
```

* With a Proxy it's simple to connect the TopComponent-Lookup with the Lookup of Nodes
  * `TopComponent.associateLookup`
* Via ExplorerManager selection of Nodes is controlled (Selection-Management).
  * `TopComponent` implements `ExplorerManager.Provider`

Important Lookups

* Services: *Lookup.getDefault()*
* Selection:

`Utilities.actionsGlobalContext()`

* Lookups of UI Objects:
  * Contain dynamic „Capabilities“

Hands-on

Status Bar

```
@ServiceProvider(service=StatusLineEleme
ntProvider.class, position=1)
public class StatusBar implements
StatusLineElementProvider {
   @Override
   public Component
getStatusLineElement() {
   return new JLabel("Todo - Task List");
   }
}
```

Actions API

Overview

* Based on Swing Actions
* Actions are registered declaratively via annotations
* Global Actions
* Menus, Toolbars, etc.
* Keybindings
* Context sensitive Actions
* Custom Views: Presenter

Swing Actions Framework & NetBeans

* NetBeans Actions API is based on Swing Action Framework
* Actions can simply implement Action or AbstractAction
* NetBeans also supplies some base classes to use:
  * **Standard presenters for menu bar, toolbar and popup menus**
  * **Simple to add Icons & customizable Keybindings**
  * **Simpler asynchronous execution**
  * **Context sensitive Actions**

Action Registry

* Actions are registered in a central registry (Layer file)
* Only one instance is generated
* Can be reused in different contexts (e.g. Menu bar & Toolbar)
* Can be accessed from different Modules
* Because all actions are registered in the same way, users can configure Toolbars
* Actions can be declaratively added in different places
  * Menus
  * Toolbars
    * Context Menus of Explorer
    * **Folders**
    * **File**

Hands-on

Global Action

```
package todo.controller.edit;
@ActionID(category = "Edit",
id = "todo.controller.edit.AddTaskAction")
@ActionRegistration(iconBase =
"todo/controller/edit/add_obj.gif", displayName =
"#CTL_AddTaskAction")
@ActionReferences({
 @ActionReference(path="Menu/Edit", position=10),
 @ActionReference(path="Toolbars/Edit", position=10),
 @ActionReference(path="Shortcuts", name="Insert") })
@Messages("CTL_AddTaskAction=Add Task...")
public final class AddTaskAction implements
ActionListener {
public void actionPerformed(ActionEvent e) { }}
```

Context Action

```
package todo.controller.edit;
@ActionID(category = "Edit",
id = "todo.controller.edit.EditTaskAction")
@ActionRegistration(iconBase =
"todo/controller/edit/configs.gif", displayName =
"#CTL_EditTaskAction")
@ActionReferences({
@ActionReference(path = "Menu/Edit", position = 20),
@ActionReference(path = "Toolbars/Edit", position =
20),
@ActionReference(path = "Shortcuts", name = "O-ENTER")
})
@Messages("CTL_EditTaskAction=Edit Task...")
public final class EditTaskAction implements
ActionListener {
public void actionPerformed(ActionEvent e) { }}
```

Context Action (cont.)

```
public final class EditTaskAction
implements ActionListener {
  private final Task context;
  public EditTaskAction(Task context) {
    this.context = context;
  }
  public void actionPerformed(
     ActionEvent e) {
  }
}
```

Context Action (cont.)

```
public final class MarkAsCompletedTaskAction
implements ActionListener {
  private final List<Task> context;
  public MarkAsCompletedTaskAction(List<Task>
context)  {
    this.context = context;
  }
  public void actionPerformed(ActionEvent e) {
    for (Task task : context) { }
  }
}
```

Wizard

* Registers Action in Layer
* Creates resource bundle entries (CTL_MyAction)
* Registers Icons
* Registers key binding
* Creates default implementation

Key-bindings

* Actions can be invoked by Keyboard-Shortcuts
  * **D- → Ctrl**
  * **O- → Alt**
  * **S- → Shift**
  * **M- → Meta / cmd**

Menus

* The display of an Action can be customized using Presenters @Override:

```
public JMenu getMenuPresenter()
public JMenu getPopupPresenter()
public Component
getToolbarPresenter()
```

Books

* Bourdeau T., Tulach J., Wielenga G., (2007), *Rich Client Programming, Plugging into the NetBeans Platform*, Prentice Hall.
* Petri J. (2009), *NetBeans Platform 6.9 Developer's Guide*, Packt Publishing.
* Bock H. (2012), *The Definite Guide to NetBeans Platform 7*, Apress.
* Myatt A. (2008), *Pro NetBeans IDE 6 Rich Client Platform Edition*, Apress.
* Toulach J. (2008), *Practical API Design Confessions of a Java Framework Architect*, Apress.

References

* TodoRCP
* Netbeans Platform Tutorials
* layerxml
* Geertjan's blog
* Toni Epple's blog
* Netbeans Lookups Explained!
* Anchialas' Java Blog
* NetBeans RCP Recipes

Q&A