# Eclipse IDE


Meeting (22.11.2016)

Robert Müller, Stefan Schmidt, David Schlosser

UNIVERSITÄT SIEGEN

Praktische Informatik

# Agenda

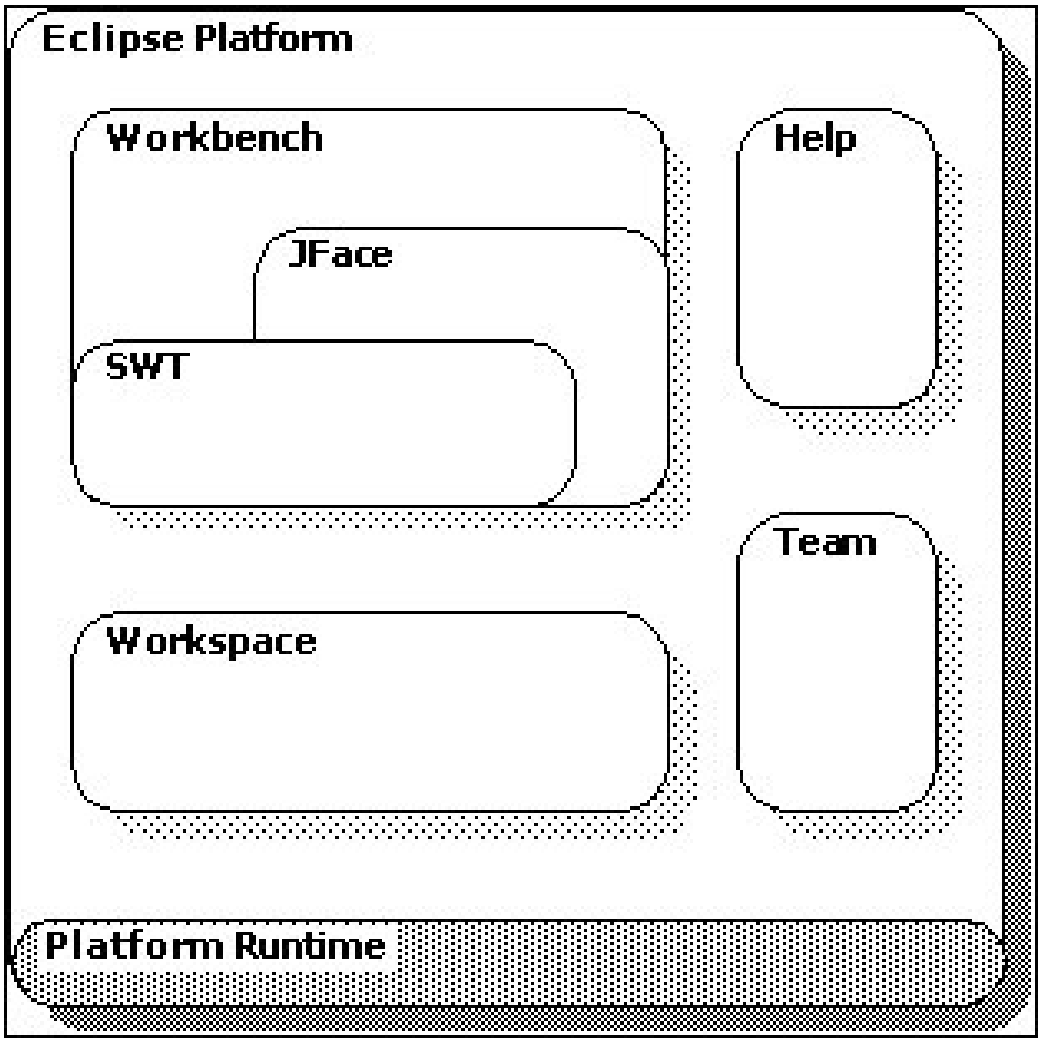


# Teil 1

# Architektur

# Architektur





# Extension Points

- Plugins können an vordefinierten **Extension Points** die Funktionalität von Eclipse erweitern

# Platform runtime

- Definiert Extension Points
- Basiert auf OSGi Framework
- Verwaltet installierte Plugins
- Aktiviert diese, wenn sie benötigt werden

# OSGi-Framework

- Offene, modulare und skalierbare Platform auf Java-Basis
- Ursprung in eingebetteten Systemen
- Nachträgliche Installation von Services zur Laufzeit möglich
- Zugrundeliegendes Service-Modell in Eclipse (Serviceorientierte Architektur, SOA)

# Workbench

- Definiert API, um Ressourcen zu erstellen und zu verwalten
- Ressourcen sind bspw.: Projekte, Dateien, Ordner
- Werden im Dateisystem gespeichert

# Workbench UI

- Extension Points um UI Komponenten hinzuzufügen
- Toolkits **JFace** und **SWT** zur Erstellung von UIs (ähnlich Swing)
  - JFace basiert auf SWT
  - SWT unabhängig von Eclipse

# Help

- Definiert Extension Points um Hilfe und Dokumentationen anzuzeigen

# Team

- Definiert Model für das Management und die Versionierung von Ressourcen

# Debug

- Definiert sprachunabhängiges Model zum Debugging und UI zum Erstellen von Debuggern

# Utilities

- Verschiedene andere Funktionalität:
  - Suchen und Ersetzen
  - XML Konfiguration
  - Zugriff auf Server
  - ...

# JDT

- Java Development Tools
- Funktionen zum Bearbeiten, Anzeigen, Kompilieren, Debuggen und Ausführen von Java Code

# PDE

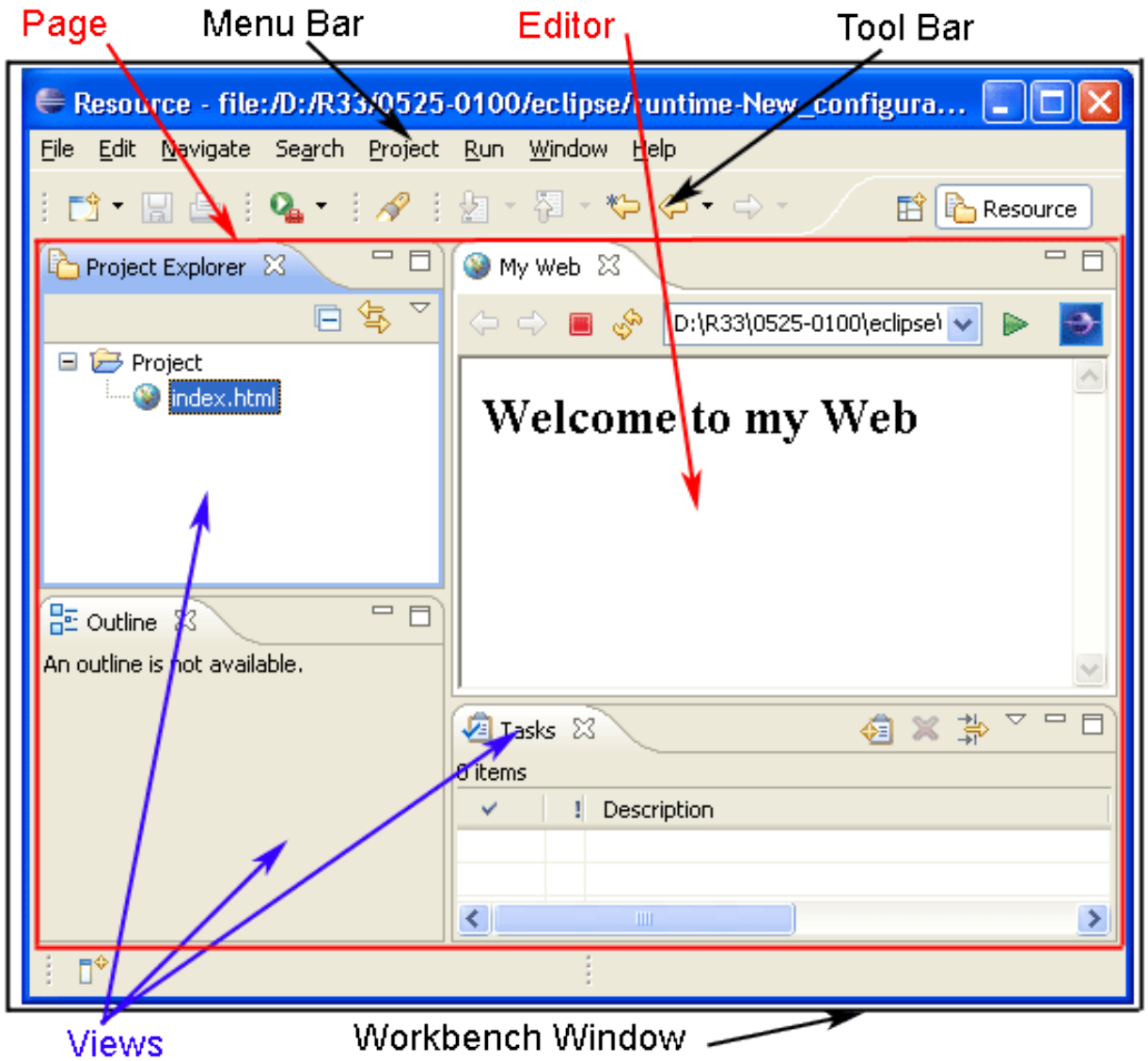
- Plugin Development Environment
- Stellt Tools für die Automatisierung von Erstellung, Veränderung, Debugging und Verteilung von Plug-ins zur Verfügung

# Teil 2

# Extension Points

# Workspace UI

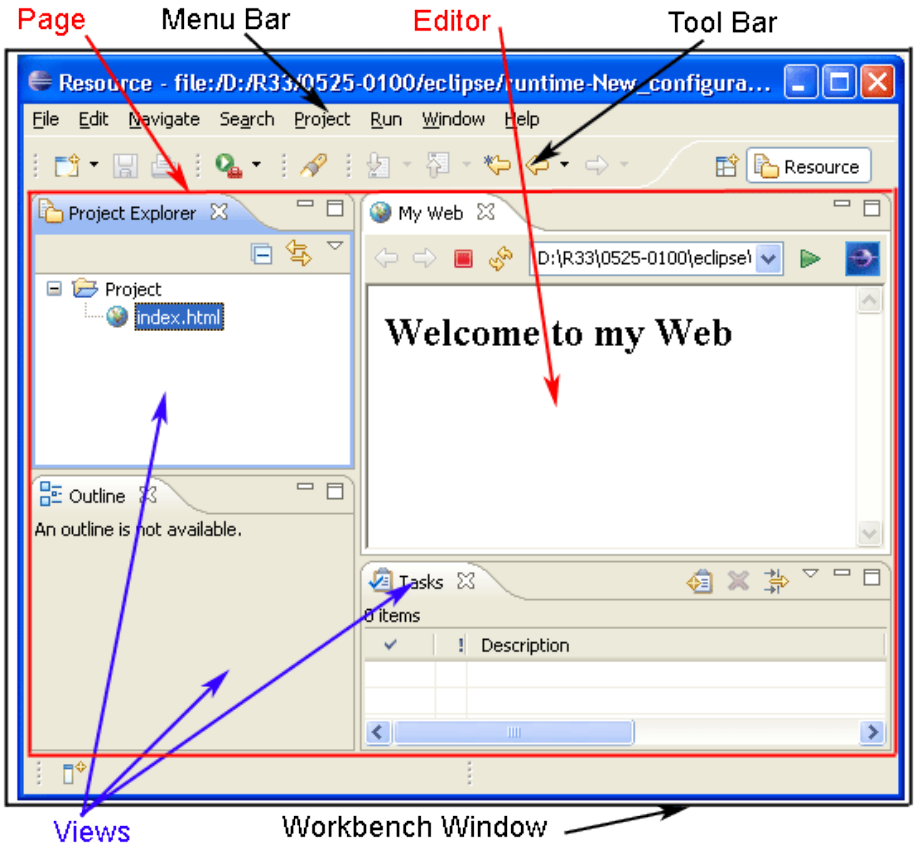




# Extension Points

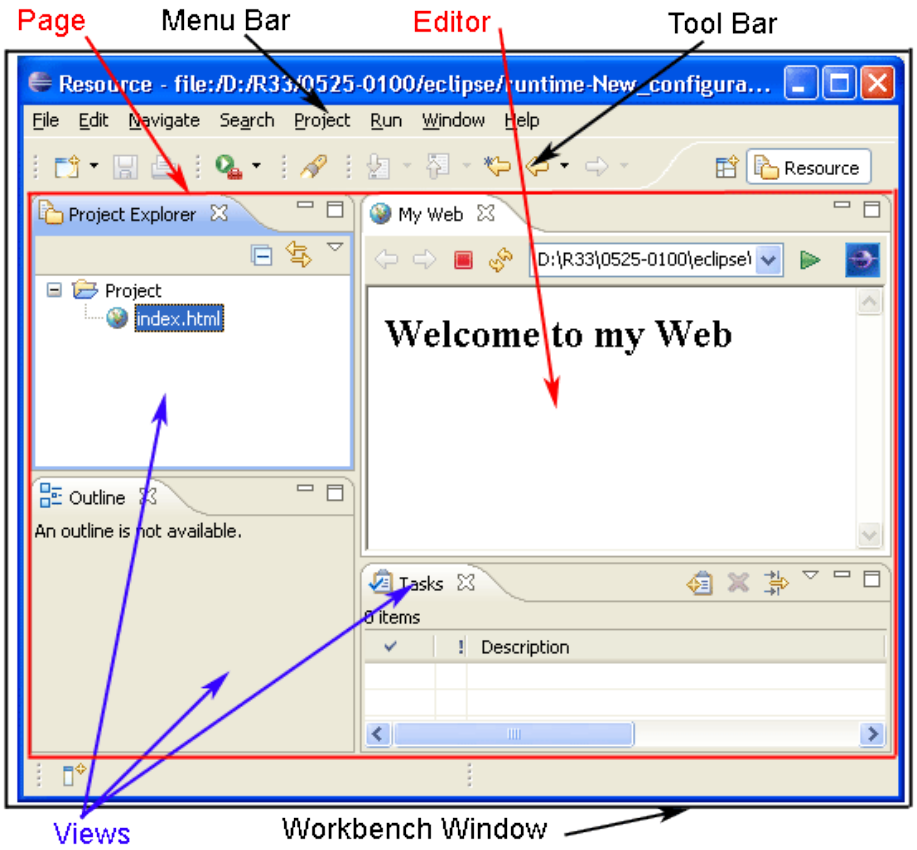
## Workspace UI

- Eine **Seite (Page)** kann veschiedene Komponenten enthalten
- Pages müssen typischerweise nicht programmiert werden



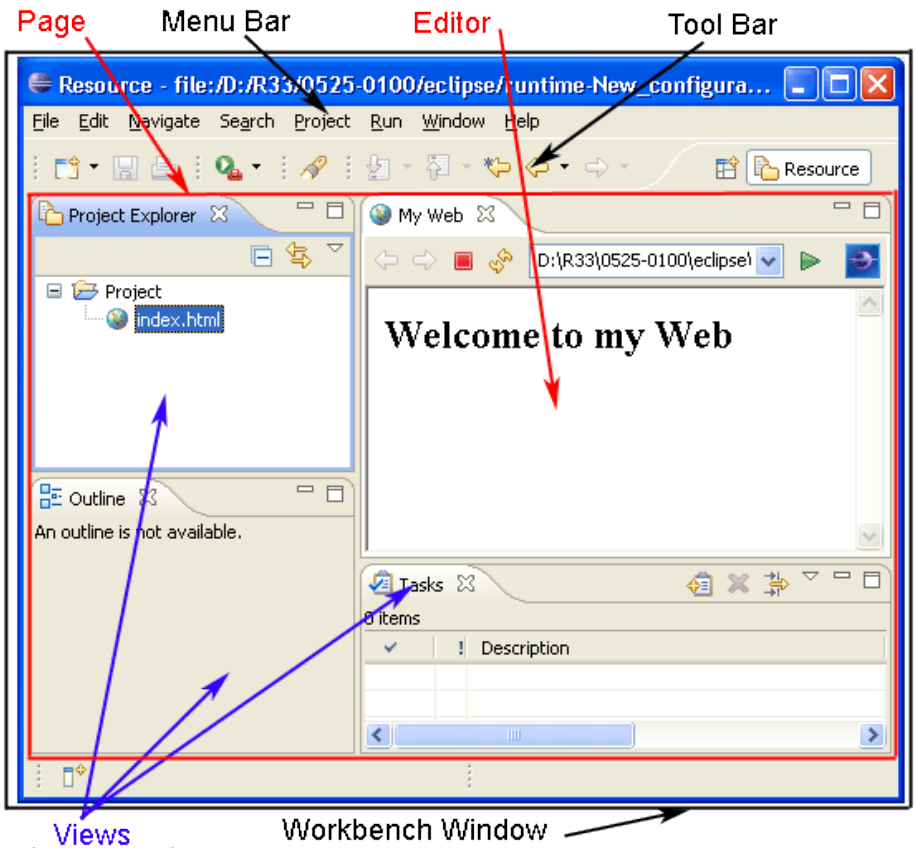
# Workspace UI

- Eine **Perspektive** definiert eine der Aufgabe angemessene Menge von Views, deren Layout und ausführbare Aktionen



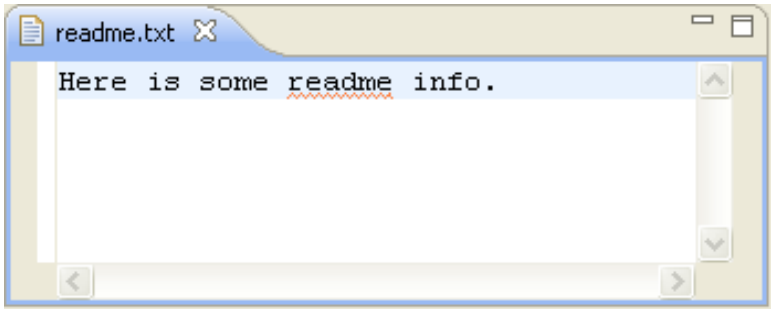
# Workspace UI

- Eine **View** wird verwendet, um durch eine Informationshierarchie zu navigieren, Editoren zu öffnen oder Einstellungen des Editors zu bearbeiten
- `org.eclipse.ui.views`



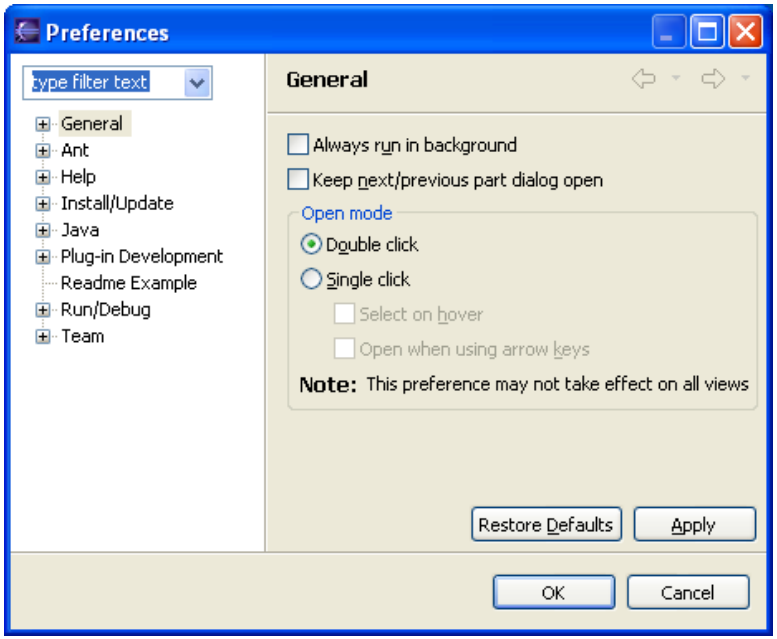
# Workspace UI

- Ein Editor wird verwendet, um ein Dokument oder ein anderes Objekt anzuzeigen und zu bearbeiten
- `org.eclipse.ui.editors`



# Workspace UI

- Plugins können Einträge zur Baumstruktur der Einstellungen hinzufügen
- `org.eclipse.ui.preferencePages`



# Workspace UI

- **Menu Bar**: die globale Tool Bar von Eclipse
- Weitere Tool Bars und Kontextmenüs können erstellt werden
- Einträge führen **Commands** aus oder beinhalten weitere Menüs
- `org.eclipse.ui.menus`

# Workspace UI

- **Dialoge:**
  - Anzeige von Meldungen (z.B. Fehlern)
  - Bestätigung einer Aktion
  - Eingabe von Werten
- **Wizard:**
  - Geführte Eingabe von Daten in einer Sequenz verschiedener Formulare (z.B. Erstellen eines Projekts)
  - Besteht aus **Wizard Dialog** und mehreren **Wizard Pages**
  - Vordefinierte Extension Points existieren für Wizards zum Erstellen, Importieren und Exportieren von Objekten

# Commands

- Identifizierung durch ID
- Werden durch Menüs und Tastenkombinationen ausgeführt
- Definieren lediglich Verhalten
- Implementiert durch **Handler**
  - verschiedene Plugins können dasselbe Command implementieren
- `org.eclipse.ui.commands`

# Handler

- Implementiert das Verhalten eines **Commands**
- Workbench entscheidet, welcher Handler ein Command behandelt
- `org.eclipse.ui.handlers`

# Key Binding

- Assoziation zwischen Command und Tastenkombination
- Tastenkombinationen haben einen Kontext in dem sie aktiv sind
- `org.eclipse.ui.bindings`

# Teil 3

# Plugins

# Plugins



- Ein Plugin ist ein OSGi **Bundle**
- Bundle spezifiziert und implementiert:
  - Laden von Java-Klassen
  - Management von Vorbedingungen
  - Life-Cycles (Start, Stop, Deinstallation, ...)
    - Plugin wird erst gestartet, sobald eine Klasse des Plugins geladen wird
    - Start-Methode zur Initialisierung
    - Methoden haben BundleContext als Argument, über den Informationen zu allen installierten Plugins abgerufen werden können

# Teil 4

# Struktur eines Plugins

# Struktur eines Plugins



- `plugin.xml`
- `MANIFEST.MF`
- Bibliotheken
- Implementierung
- Ressourcen (z.B. Icons)

# Struktur eines Plugins



## plugin.xml

- Definition der Extension Points, die vom Plugin erweitert werden
- Definition der Extension Points, die das Plugin bereitstellt, sodass andere Plugins diese erweitern können

# Struktur eines Plugins



## plugin.xml

- Struktur:
- 
```
<plugin>
<!-- Erweiterung vorhandener Extension Points -->
  <extension point="..">
    <..> (abhängig vom Extension Point)
  </extension>
<!-- Definition eigener Extension Points -->
  <extension-point id=".." name=".." schema=".."/>
</plugin>
```

# Struktur eines Plugins



## MANIFEST.MF

- OSGi Manifest
- Definition von Metadaten des Plugins
  - Name, Version, Package
  - Abhängigkeiten
- 

```
1 Manifest-Version: 1.0
2 Bundle-ManifestVersion: 2
3 Bundle-Name: Helloworld
4 Bundle-SymbolicName: com.example.helloworld;singleton:=true
5 Bundle-Version: 1.0.0.qualifier
6 Bundle-Vendor: EXAMPLE
7 Require-Bundle: org.eclipse.ui
8 Bundle-RequiredExecutionEnvironment: JavaSE-1.8
```

# Literature



- Eclipse Dokumentation
  *http://help.eclipse.org/neon/nav/2_0*
- Extension Points
  *http://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/reference/extension-points/index.html?cp=2_1_1*