Dynamische Plug-ins mit Eclipse 3

Martin Lippert (martin.lippert@it-agile.de, www.it-agile.de)
Tammo Freese (freese@acm.org)

Überblick

- Die Ausgangslage
  - Dynamische Plug-ins – Warum?
- Eclipse 3
  - Die OSGi-basierte Runtime von Eclipse 3
- Dynamic-enabled Plug-ins
- Dynamic-aware Plug-ins
- Live-Demo!!!
- 3.1-Neuigkeiten

Die Ausgangslage

- Eclipse-basierte Installationen werden immer größer
  - Rational Application Developer: >1000 Plug-ins
- Nicht alle Plug-ins werden immer benötigt
- Eclipse-basierte Anwendungen laufen lange
  - Meine IDE läuft den ganzen Tag
  - Im Standby-Modus sogar noch viel länger

Anforderungen an die Plattform

- Speicherplatz:
  - Nur benötigte Plug-ins werden tatsächlich geladen
  - Nicht mehr benötigte Plug-ins werden wieder aus dem Speicher entfernt
  - Es darf kein ausschließlich wachsendes System sein
- Keine Neustarts
  - Plug-ins sollen dann (de-)installiert werden können, wenn sie (nicht mehr) gebraucht werden
  - Plug-ins sollen zur Laufzeit aktualisiert werden können

Eclipse 2.1

- Mit Eclipse 2.1 und davor konnten Plug-ins nicht zur Laufzeit installiert, deinstalliert oder aktualisiert werden
  - Es war immer ein Neustart der Plattform notwendig
- Der Grund:
  - Die Plattform hat nur beim Startup analysiert, welche Plug-ins vorhanden sind
  - Weite Teile der UI basierten auf dieser Annahme

Der Weg zu Eclipse 3

- Seit Eclipse 3.0 ist die Plattform in der Lage, Plug-ins dynamisch zur Laufzeit zu installieren, zu deinstallieren und zu aktualisieren

- Aber: Dieses Feature ist nicht „*for free*“:
  - Plattform managt den Lebenszyklus von Plug-ins
  - Plug-ins müssen sich aber „konform“ verhalten

Die Eclipse 3 Runtime

- Es wurde diskutiert, wie die Runtime verändert werden sollte, damit Plug-ins dynamisch verwaltet werden können
  - Entweder selbst-implementierte Eclipse-2-Runtime erweitern
  - Oder den Kern der selbst-implementierten Runtime ersetzen
- Man hat sich für die zweite Variante und OSGi entschieden

OSGi – Open Service Initiative

„The OSGi™ specifications define a standardized, component oriented, computing environment for networked services.

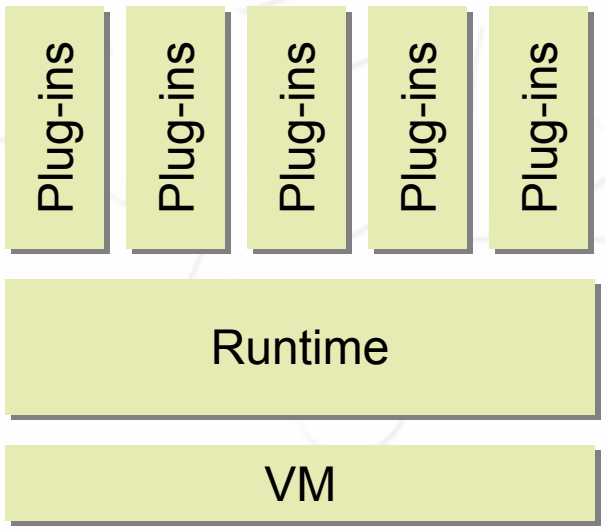
Adding an OSGi Service Platform to a networked device (embedded as well as servers), adds the capability to manage the life cycle of the software components in the device from anywhere in the network. Software components can be installed, updated, or removed on the fly without having to disrupt the operation of the device.“

[http://www.osgi.org]

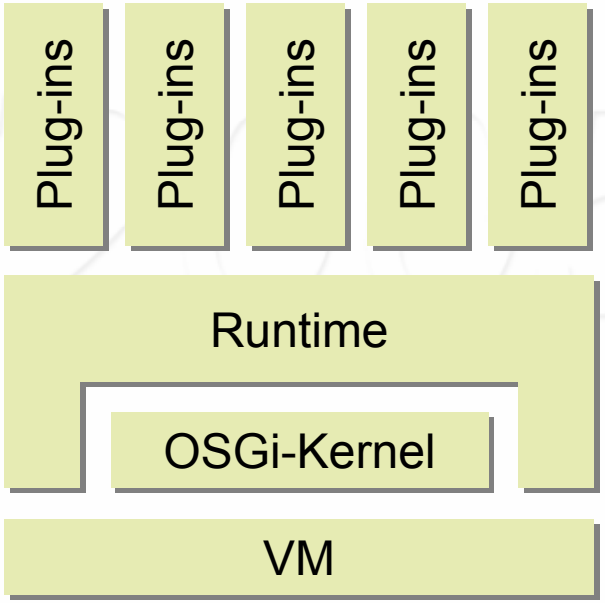
Eine OSGi-basierte Runtime

- OSGi (Open Service Gateway Initiative, siehe auch http://www.osgi.org)
  - Stammt ursprünglich aus dem Embedded-Bereich
  - Dient dazu, Komponenten (genannt **Bundles**) zu verwalten
- Eclipse 3 basiert auf einer OSGi-Implementation von IBM
  - Sie wurde erweitert um Eclipse-spezifische Elemente

Aufbau



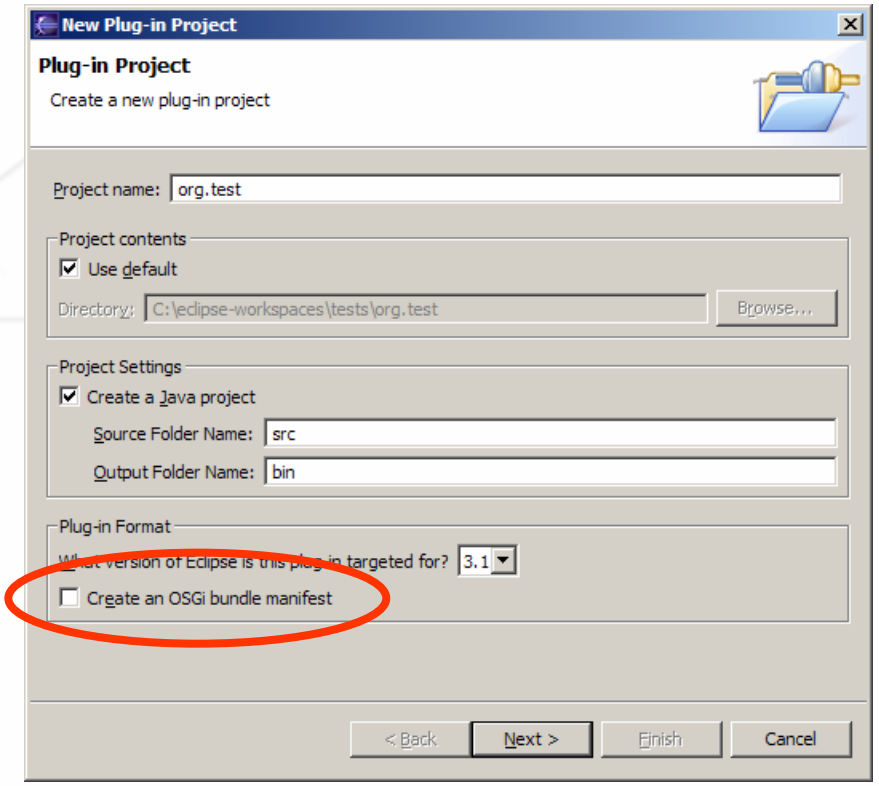
Eclipse 2.1



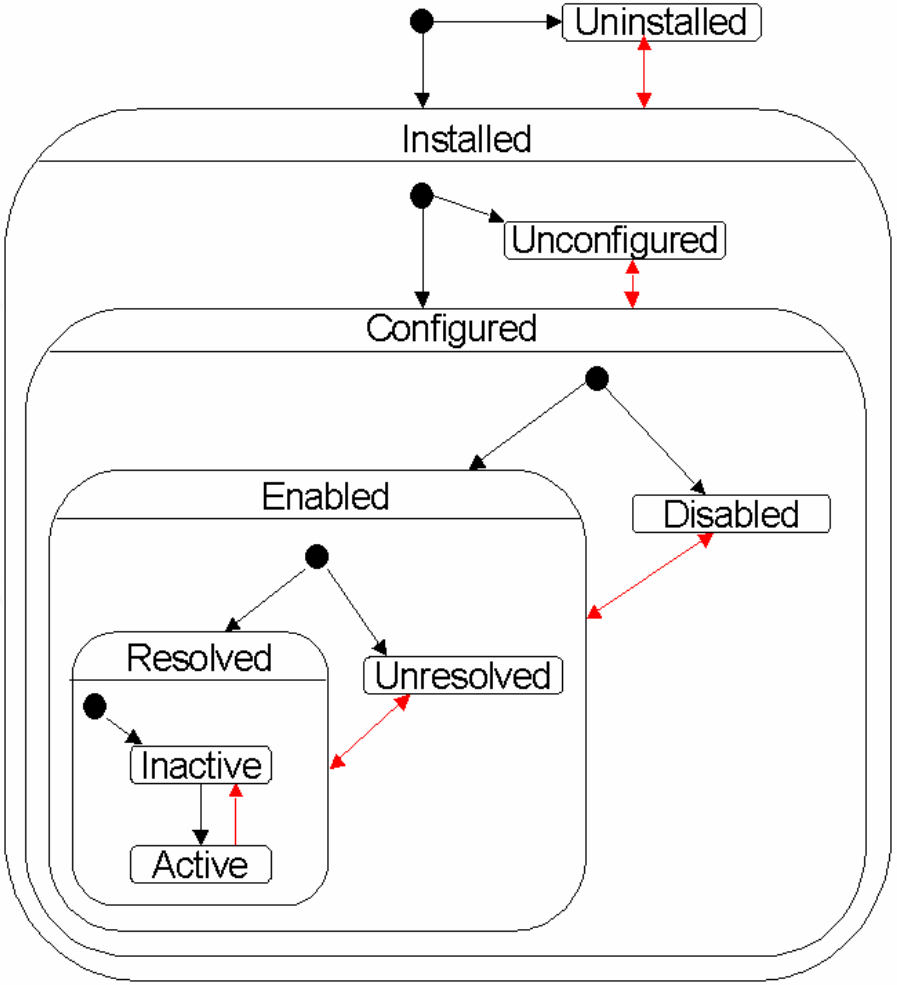
Eclipse 3.0/3.1

Plug-ins vs. OSGi-Bundles

- Nur zur Klärung: Intern bildet die Eclipse-Runtime alle Plug-ins auf OSGi-Bundles ab
  - Für „normale“ Eclipse-Plug-ins passiert dies automatisch
- OSGi-Bundles können auch direkt implementiert werden



Lebenszyklus eines Plug-ins



http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/equinox-home/dynamicPlugins/state_description.html

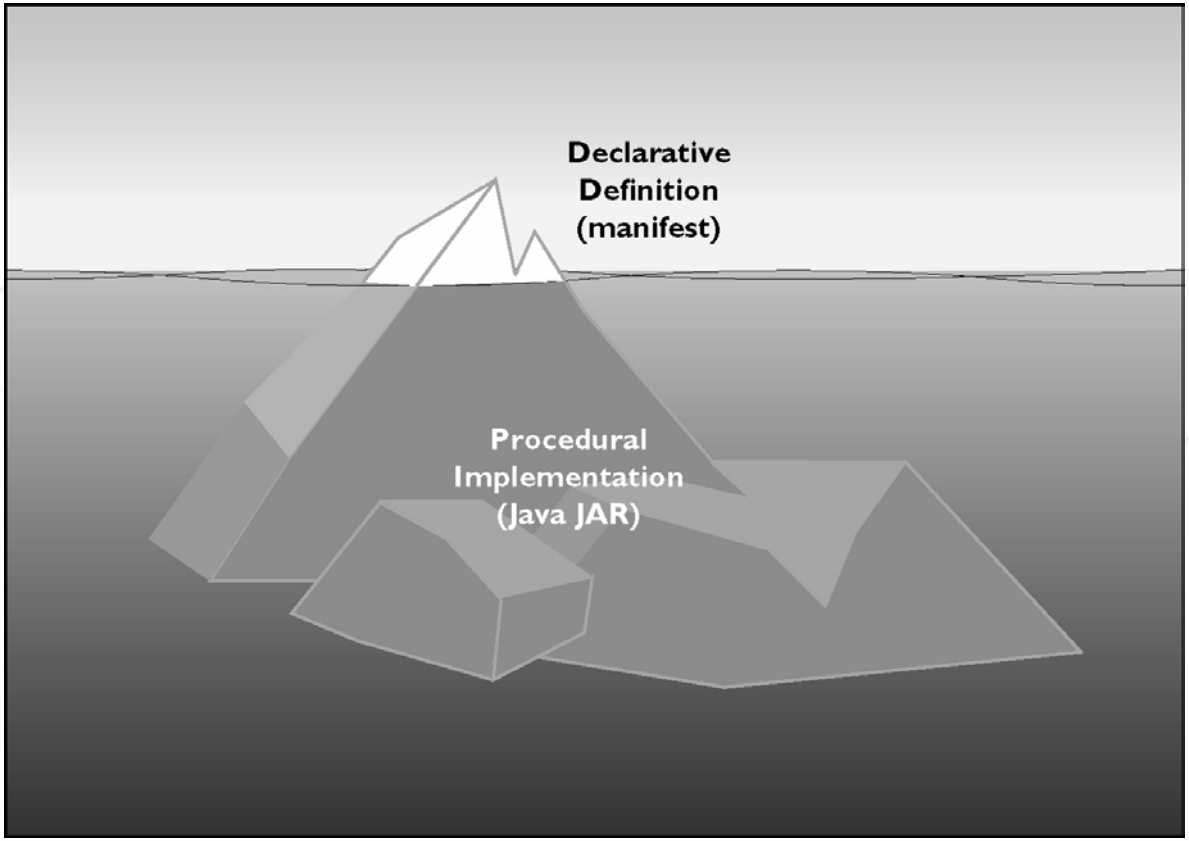
Dynamische Plug-ins

- Ein Plug-in kann „*dynamic enabled*“ sein
  - Es ist selbst dynamisch (de-)installierbar
- Ein Plug-in kann „*dynamic aware*“ sein
  - Es kann mit anderen dynamisch (de-)installierten Plugins umgehen

Dynamic-enabled Plug-ins

- „Dynamic-enabled“ bedeutet, dass das Plug-in selbst dynamisch (de-)installierbar ist
  - Plug-in wird erst aktiviert, wenn es benötigt wird
    - Realisiert Eclipse für uns
  - Code wird erst geladen, wenn er wirklich gebraucht wird
    - Realisiert Eclipse für uns
  - Plug-in „räumt auf“, wenn es deaktiviert wird
    - Sollte auch ein sauber implementiertes Eclipse-2-Plug-in bereits erfüllen

Plugin-Deklaration



Aus: Contributing to Eclipse, Addison-Wesley, 2004

Startup time: O(# used plug-ins), not O(# installed plug-ins)

Das große Aufräumen

- Ein Plug-in sollte brav alles aufräumen, was es erzeugt oder benutzt hat
  - Notification-Listener abmelden, z. B.
    - IResourceChangeListener
  - Resourcen freigeben (Icons, Fonts, Colors)
  - Geöffnete Files, Pipes oder Sockets schließen
  - Adaptor-Factories abmelden
  - Eigene Hintergrund-Jobs oder Threads beenden
  - …

Beispiel

```
public class MyPlugin
   extends AbstractUIPlugin {

   ...

   public void start(BundleContext context) throws Exception {
      super.start(context);
   }

   public void stop(BundleContext context) throws Exception {
      super.stop(context);
      plugin = null;
      resourceBundle = null;

      // weitere Resourcen freigeben
   }
}
```

Dynamic-aware Plug-ins

- Ich biete ein Extension-Point an, ein anderes Plug-in eine Extension
- Das Extension-Plug-in wird dynamisch (de)installiert
- Ich muss dafür sorgen, dass mein Extension-Point-Plug-in auf diese Situation korrekt reagiert

Neue Extensions

- Was muss ich tun, wenn dynamisch eine neue Extension installiert oder aktiviert wird?
  - z. B. entsprechend UI updaten (beispielsweise die Workbench ihre Menü-Einträge)

Alte Extensions

- Problem: Bereits erzeugte Objekte existieren erst einmal weiter, auch wenn das Plug-in deaktiviert ist
  - Dadurch verbleiben die Objekte und auch die Klassen in der VM
  - Die alten Objekte leben weiter

Alte Extensions

- Wenn eine Extension dynamisch deinstalliert oder deaktiviert wird, muss das Extension-Point-Plug-in dafür sorgen, dass…
  - Das UI aktualisiert wird (z.B. Menü-Einträge)
  - Bereits erzeugte Extension-Objekte entfernt werden (Referenzen kappen)
  - Ggf. auch entsprechende UI-Elemente entfernen (beispielsweise ein View)

IRegistryChangedListener

- Listener-Interface, mit dem über Veränderungen an der Extension-Registry benachrichtigt wird

```
public interface IRegistryChangeListener
   extends EventListener {

   public void registryChanged(IRegistryChangeEvent event);
}
```

IRegistryChangedEvent

- Event beinhaltet alle Informationen, was sich an der Extension-Registry verändert hat

```
public interface IRegistryChangeEvent {
   public IExtensionDelta[] getExtensionDeltas();
   public IExtensionDelta[] getExtensionDeltas(
      String namespace);
   public IExtensionDelta[] getExtensionDeltas(
      String namespace, String extensionPoint);
   public IExtensionDelta getExtensionDelta(
      String namespace, String extensionPoint,
      String extension);
}
```

IExtensionDelta

- IExtensionDelta-Objekte beinhalten detaillierte Informationen über einzelne Änderungen

```
public interface IExtensionDelta {
   public int ADDED = 1;
   public int REMOVED = 2;
   public int getKind();
   public IExtension getExtension();
   public IExtensionPoint getExtensionPoint();
}
```

Auf Veränderungen reagieren

```
registry.addRegistryChangeListener(new IRegistryChangeListener() {
  public void registryChanged(IRegistryChangeEvent event) {
    IExtensionDelta[] extensionDeltas =
      event.getExtensionDeltas(pid, extid);

    for(int i = 0; i < extensionDeltas.length; i++) {
      if (extensionDeltas[i].getKind() == IExtensionDelta.ADDED) {
        IExtension extension = extensionDeltas[i].getExtension();
        // do something with new extension
      }
      else if(extensionDeltas[i].getKind() == IExtensionDelta.REMOVED) {
        IExtension extension = extensionDeltas[i].getExtension();
        // cut all references to objects from old extension
      }
    }
  }
});
```

Live-Demo !!!

3.1-Neuigkeiten

- In 3.0 konnte die Workbench nur mit dynamisch installierten Plug-ins umgehen
- In 3.1 berücksichtigt die Workbench nun auch dynamisch deinstallierte Plug-ins

- Neue Hilfsklasse: IExtensionTracker
  - org.eclipse.core.runtime.dynamicHelpers
  - Vereinfacht den Umgang mit dynamischen Plug-ins
  - Siehe auch Blog-Eintrag von Kim Horne:
    - http://eclipse.pookzilla.net/2005/03/supporting-dynamic-bundle-loading-in.php

Vielen Dank

- Fragen jederzeit gerne!

Martin Lippert: martin.lippert@it-agile.de

Tammo Freese: freese@acm.org