Introduction to the SiDiff 2.0 Development Environment

Timo Kehrer

16. September 2009

Inhaltsverzeichnis

Dokumenthistorie

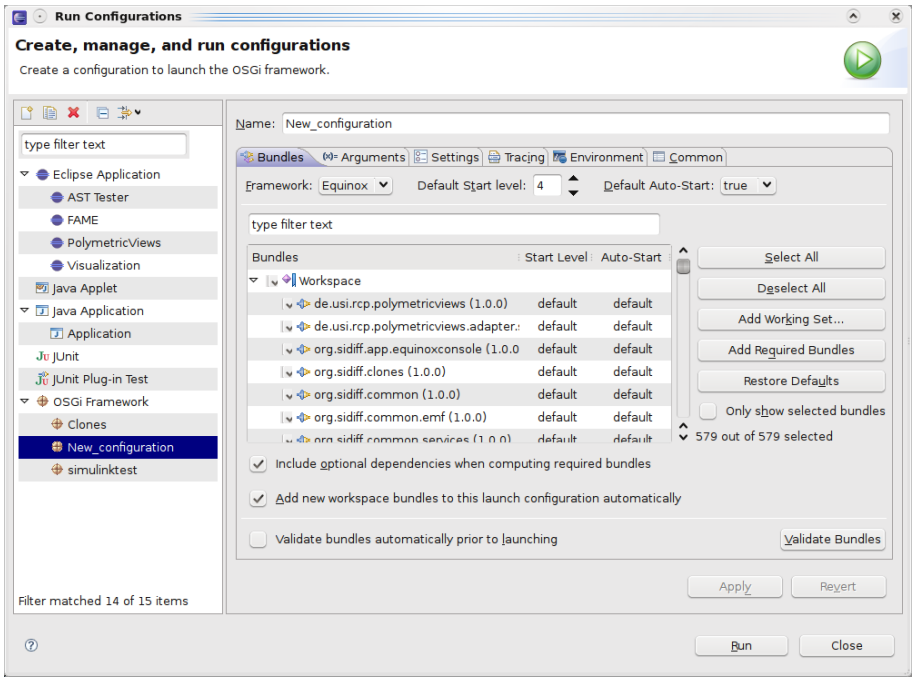
Dieses Dokument wird fortlaufend gepflegt. Die nachfolgende Tabelle gibt eine Übersicht über die Änderungen in einzelnen Versionen.

Datum	Änderungen
16.09.09	erste Version

1 Erstellen und Starten einer Laufzeitumgebung

Das Einrichten einer Laufzeitumgebung erfolgt in Eclipse mittels *Run* → *Run configurations ...*. Der Dialog zeigt bestehende Laufzeitumgebungen und ermöglicht diese zu konfigurieren. Wir richten hier eine neue ein.

Hierzu ist links *OSGi Framework* auszuwählen und der *New*-Button (oben) zu klicken. Eine neue Laufzeitumgebung wird angelegt. Im rechten Teil des Fensters kann die Umgebung konfiguriert werden.



Unter *Bundles* können die Bundles ausgewählt werden, die man in der Laufzeitumgebung haben möchte. Hier können auch Eclipse-Plugins ausgewählt werden, weil diese prinzipiell auch OSGi-Bundles sind.

Hinweis: Um zu verhindern, dass man letzlich wieder ein komplettes Eclipse startet sollte man:

1. Target-Platform deselektieren (alle untergeordneten Bundles werden automatisch deselektiert)
2. Mit dem Button *Add Required Bundles* werden alle notwendigen Bundles (z.B. für Dateizugriff) wieder ausgewählt.

Unter *Arguments* können Parameter für das OSGi-Framework und die Virtual Machine (VM) gesetzt werden. Die OSGi-Parameter sollten nicht verändert werden. Die VM-Parameter können nach belieben erweitert werden, z.b. um mit `-Xmx512m` mehr Arbeitsspeicher bereitzustellen.

Hinweis: Konsolenapplikation können durch die Kombination des Arguments `-console` und des VM-Parameters `-Dosgi.noShutdown=true` realisiert werden.

Die übrigen Konfigurations-Tabs werden für unsere Zwecke nicht benötigt.

TODO: Auch andere Tabs können von Bedeutung sein, so z.B. für das Anlegen von *.launch-Dateien. Auch weitere Konfigurationsmöglichkeiten wie das Workind-Directory lohnt es sich im Rahmen eines neuen WhitePapers evtl. zu beschreiben.

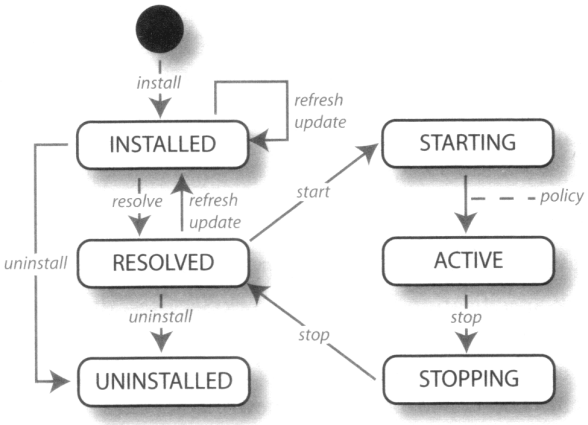
1.1 Starten der Laufzeitumgebung

Die Laufzeitumgebung wird einfach mit *Run* gestartet. Nach wenigen Sekunden steht uns im *Colsole*-View von Eclipse die OSGi-Konsole zur Verfügung. Hier ein paar nützliche Kommandos:

- `ss` gibt eine Übersicht (short status) über vorhandene Bundles aus. Die Liste sollte zwischen 50 und 100 Bundles betragen, jenachdem welche Bundles ausgewählt wurden. Werden 500 oder mehr Bundles gelistet, ist dies ein Indiz für eine vollständige Eclipse-Instanz. Die Liste gibt außerdem an, in welchem Zustand ein Service ist. (siehe Service-Status)
- `start XX` aktiviert den Service mit der Nummer XX (aus der Liste). Bei diesem manuellen Start werden auch ggf. vorhandene Exceptions ausgegeben, die beim automatischen Startversuch nicht anzegeigt werden.
- `stop XX` deaktiviert den Service mit der Nummer XX.

  `start` und `stop` können genutzt werden, um abwechselnd verschiedene Service-Implementierungen einzusetzen.

Service-Status Der Lebenszyklus eines OSGi-Bundles sieht folgende Zustände vor:



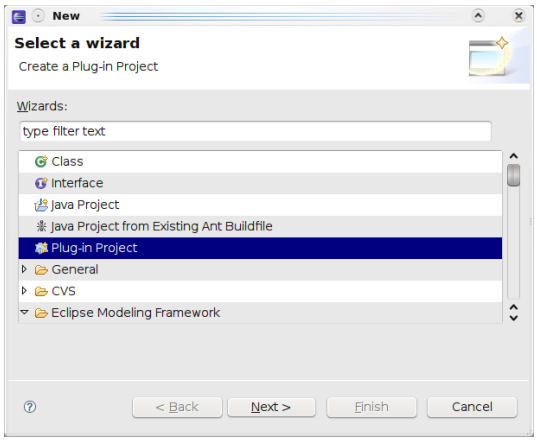
Bundles sind zunächst *installiert*, d.h. sie stehen dem System zur Verfügung, wenn wir sie der Laufzeitumgebung hinzufügen. Das OSGi-Framework prüft sofort nach der Installation, ob alle Voraussetzungen erfüllt sind, damit das Bundle "einsatzfähig" ist. Ist dies der Fall wechselt das Bundle in den Zustand *resolved*. Hier kann das Bundle mit `start` und `stop` aktiviert bzw. deaktiviert werden. Ist der Start nicht möglich (z.B. aufgrund einer Exception), bleibt das Bundle *resolved*. Bei erfolgreichem Start wird es *active*. *Starting* und *Stopping* beschreiben die Zeitspanne der (De-)Aktivierung.

(Für Fortgeschrittene: Sie entsprechen den Methoden im Activator.)

2 Kochrezept A: Erstellen eines Bundles, das Klassen und Ressourcen bereitstellt

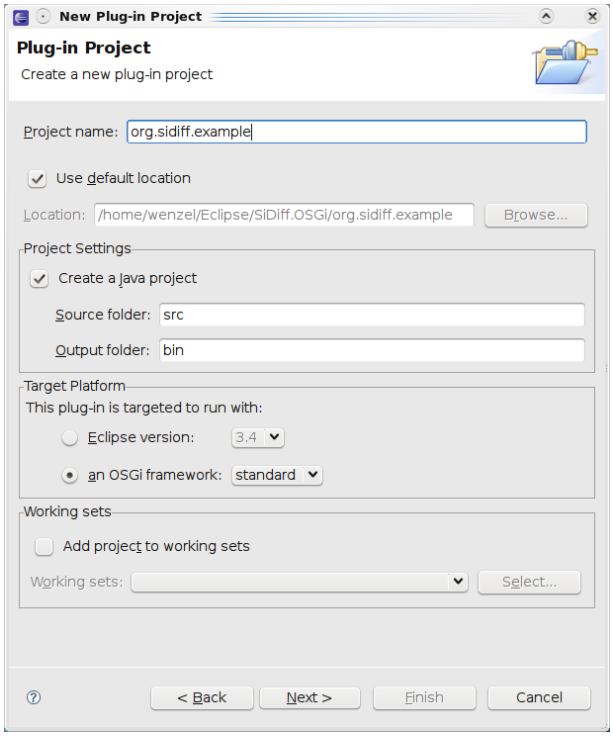
Inhalt des Kapitels wurde noch nicht reviewed

Schritt 1:



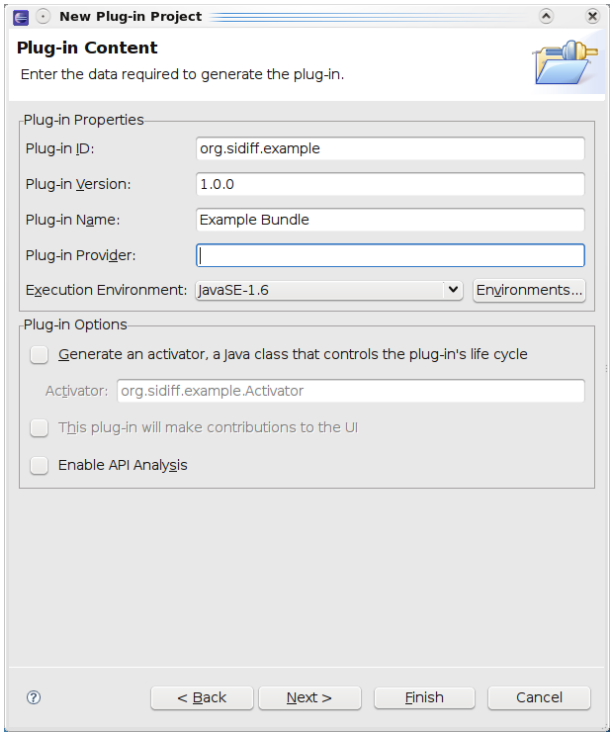
Erstelle ein neues Plugin-Projekt mit dem Eclipse-Wizard.

Schritt 2:



Wähle den Bundlenamen gemäß Konvention und setze ihn als Projektnamen. Bei den Project Settings wähle *Create a Java Project* aus und definiere *src* als Source folder und *bin* als Output folder. Als Target Platform ist *an OSGi framework* zu wählen. Achtung: Hier muss in der Dropdown-Box noch *standard* ausgewählt werden.

Schritt 3:



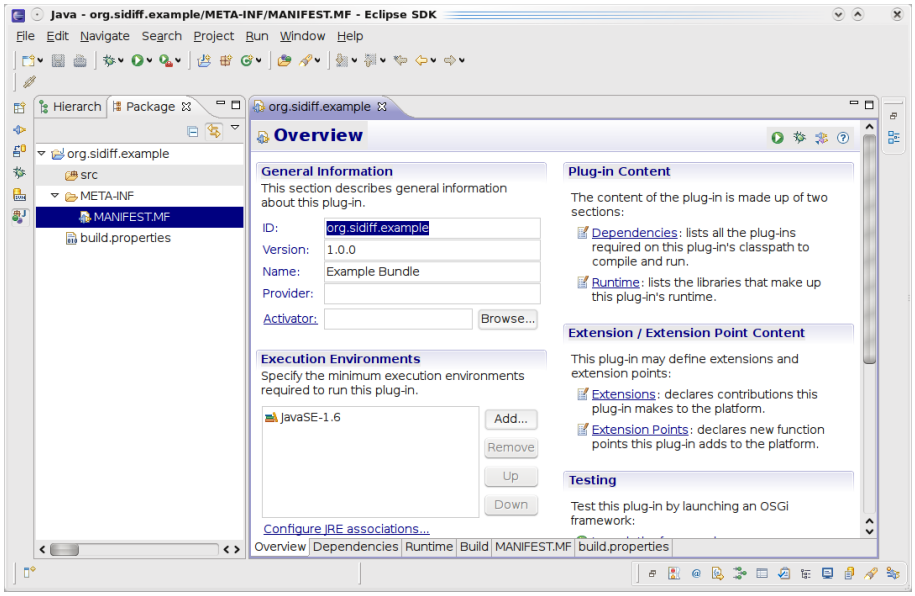
Im nächsten Dialog sollte bei Plugin-ID der Bundlename stehen. Ist dies nicht der Fall, so ist nochmal einen Schritt zurückzugehen und der Projektname neu zu setzen. Die Version sollte defaultmäßig bei 1.0.0 bleiben. Der Plugin-Name ist frei wählbar, ebenso der Plugin-Provider[1]. Als Execution-Environment ist Java 1.6 auszuwählen.

Bei Plugin-Options kann man noch auswählen, ob ein Activator erstellt werden soll. Ein Activator ermöglicht die Ausführung von Code, wenn ein Bundle innerhalb des OSGi-Frameworks aktiviert wird (d.h. i.d.R. bei erstmaligem Zugriff). Zur Bereitstellung von Klassen und Ressourcen ist das i.d.R. nicht nötig.

Sollte dieser Schritt vergessen worden sein, kann auch nachträglich ein Activator angelegt werden. Hierzu klickt man in der Übersicht des Manifest-Editors (siehe Schritt 4) auf das unterstrichene Label “Activator”. Mithilfe des erscheinenden Dialogs kann der Activator angelegt werden. Wie der Activator konkret zu implementieren ist, wird später gezeigt.

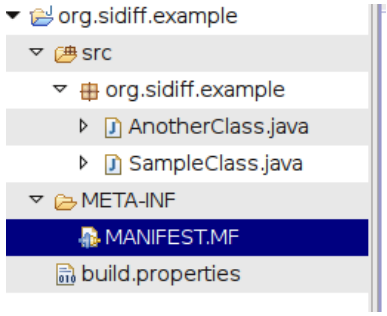
Schritt 4:

[1] Hierfür wird es später mal ein geeignetes Template geben.



Der Manifest-Editor zeigt nun eine Übersicht über das neue Bundle. Hier können später Äbhängigkeiten zu anderen Bundles, Freigaben und Build-Eigenschaften eingestellt werden (siehe Tabs am unteren Rand).

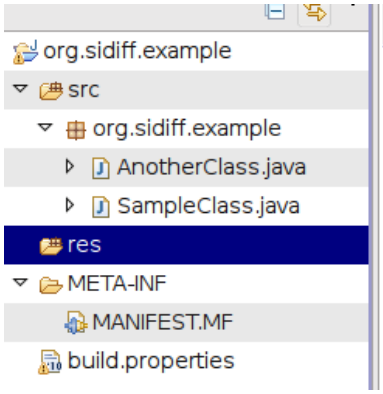
Schritt 5:



Zunächst muss das Basis-Paket angelegt werden. Es heißt wie das Bundle. Darin können nach Belieben Klassen und weitere Pakete angelegt werden.

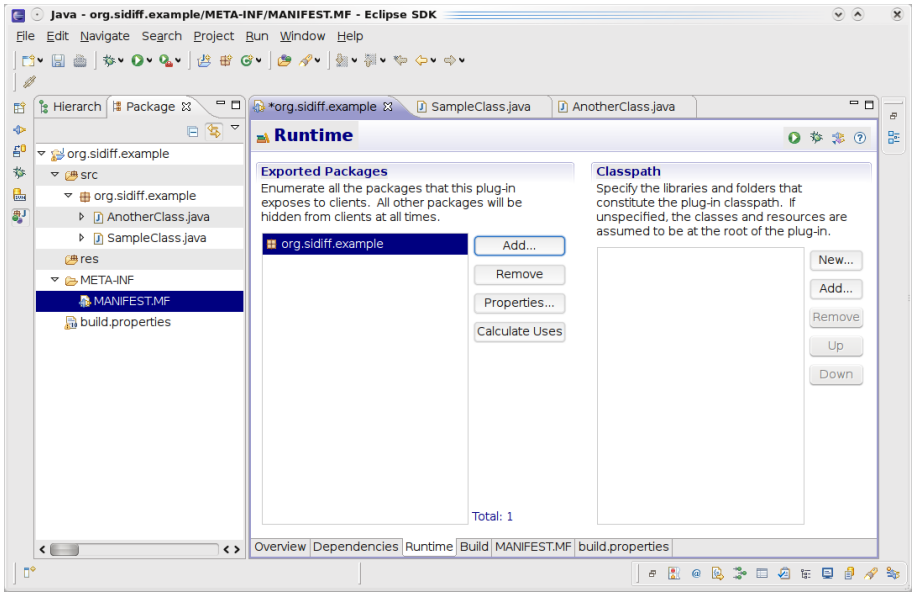
Schritt 6: Implementierung der Klassen

Schritt 7:



Sollen neben Java-Code auch Ressourcen, z.B. Bilder, XML-Dateien oder sonstiges bereitgestellt werden, so können weitere Source-Ordner angelegt werden. Dateien dieser Ordner landen später automatisch im Jar-Archiv eines Bundles.

Schritt 8:



Damit die Java-Klassen von anderen Bundles genutzt werden können, müssen diese explizit freigegeben werden. Dieses geschieht über den Tab “Runtime” des Manifest-Editors. Bei *Exported Packages* sind diejenigen Pakete auszuwählen, die nach aussen freigegeben werden sollen.

Falls Abhängigkeiten zu anderen Bundles bestehen, können diese unter dem Tab *Dependencies* eingetragen werden.

Benutzen des Bundles: Die Klassen und Ressourcen des neuen Bundles können nun einfach in anderen Bundles benutzt werden. Hierzu ist einfach eine Abhängigkeit zu unserem neu erstellten Bundle einzutragen.

3 Häufige Stolperfallen

Wenn mal etwas nicht läuft wie es soll, kann oft eine der nachfolgenden Stolperfallen ein Grund dafür sein:

3.1 Beim Kompilieren

Abhängigkeiten zwischen Bundles Sind alle benötigten Bundles im Manifest-Editorals "Required Plugins" in den *Dependencies* eingetragen?

Freigabe von Paketen Sind alle Pakete, die in anderen Bundles zur Verfügung stehen sollen im Manifest-Editor unter *Runtime* als "Exported Packages" angegeben?

3.2 Zur Laufzeit

Activator wird nicht ausgeführt Oftmals wird der Activator nicht ausgeführt weil schlichtweg vergessen wurde, ihn in der MANIFEST.MF einzutragen. Ein weiteres Problem besteht, wenn der Namespace nicht eindeutig ist. Hierzu empfehle ich, den Activator immer in ein Paket mit dem Namen des Bundles abzulegen.

Ladereihenfolge von Bundles / Exceptions im Activator Bundles können in unterschiedlichster Reihenfolge beim OSGi-Framework registriert werden. Explizite Abhängigkeiten werden zwar i.d.R. durch das Framework aufgelöst, jedoch erfolgt diese Auflösung erst im Zuge der Aktivierung der Bundles. Deshalb sollte man vermeiden, im Activator bereits die Aktivierung anderer Bundles vorauszusetzen.

Wenn andere Bundles zwingend notwendig sind kann man mit Hilfe eines Listeners auf die Aktivierung der Bundles warten:

```
public class Activator implements BundleActivator {
  BundleContext context;
  BundleListener listener = new BundleListener() {
    @Override
    public void bundleChanged(BundleEvent event) {
      if (event.getType() == BundleEvent.STARTED ||
       org.sidiff.example.myservice.Activator.BUNDLE_NAME.equals(
       event.getBundle().getSymbolicName())) {
        OtherService os = ServiceHelper.getService(context, OtherService.class);
        if (os != null) {
          os.doSomethingElse();
          context.removeBundleListener(listener);
        }
      }
    }
  };

  public void start(BundleContext context) throws Exception {
    this.context = context;
```

```
    context.addBundleListener(listener);
  }

  public void stop(BundleContext context) throws Exception {
    context.removeBundleListener(listener);
  }
```

Klassen aus anderen Bundles lassen sich nicht per Reflection laden Der OSGi-Standard sieht vor, dass jedes Bundle seinen eigenen ClassLoader zugeteilt bekommt, damit sich verschiedene Bundles nicht gegenseitig stören können. Dieses Prinzip behindert die Idee, mit einem Bundle Klassen bereitzustellen, die an anderer Stelle reflektiv (also mit einem anderen ClassLoader) geladen werden sollen.

Das Problem kann umgangen werden, indem das Bundle, das die Klassen anbietet, seinen ClassLoader dem `ResourceUtil` aus dem Bundle `org.sidiff.common` bekannt macht:

```
ResourceUtil.registerClassLoader(this.getClass().getClassLoader());
```

Anschließend können die alle Klassen dieses Bundles mit Hilfe des `ReflectionUtil` (ebenfalls aus `org.sidiff.common`) reflektiv geladen werden.