Anleitung zur Erstellung eines Plug-ins für Alvis

Simon Groth

31. August 2011

Inhaltsverzeichnis

1 Einleitung

Ein Plug-in dient dazu das Alvis-Programm zur Visualisierung von Algorithmen um weitere Algorithmen-Typen zu erweitern. Die Algorithmen werden von dem Benutzer des Programms definiert. Daher muss das Plug-in die Datentypen, welche für eine bestimmte Algorithmenart benötigt werden liefern. Außerdem die Möglichkeit diese Datentypen zu erstellen und zu visualisieren.

2 Vorkenntnisse

Sie sollten über diese Kenntnisse verfügen.

- Entwicklung von Java mit dem Eclipse SDK.
- Eclipse-RCP und Plug-in Mechanismus [1]
- SWT-Kenntnisse [2]
- Grober Überblick über die Arbeitsweise von Alvis

3 Vorbereitung

Sie müssen das Alvis Projekt in Ihrem Workspace importiert haben, da bei der Entwicklung des Plug-ins Abhängigkeiten auf das bereits vorhandene Projekt entstehen. Um sicherzustellen, dass Sie alle benötigten Plug-ins des Projektes haben lesen Sie sich den Artikel zum Start des Projektes aus dem Wiki [3] durch.

[1] http://www.vogella.de/articles/EclipseRCP/article.html
[2] http://www.vogella.de/articles/SWT/article.html
[3] http://vcs.informatik.uni-siegen.de/trac/algvistool/wiki/AlVisStart

4 Konventionen

4.1 Namenskonventionen

Für bessere Übersichtlichkeit unter den Plug-ins wollen wir das neue Plug-in ***de.unisiegen.informatik.bs.alvis.pluginname*** nennen.
Weiterhin wird noch ein zweites Plug-in für die Datentypen mit dem Namen ***de.unisiegen.informatik.bs.alvis.pluginname.datatypes*** angelegt.

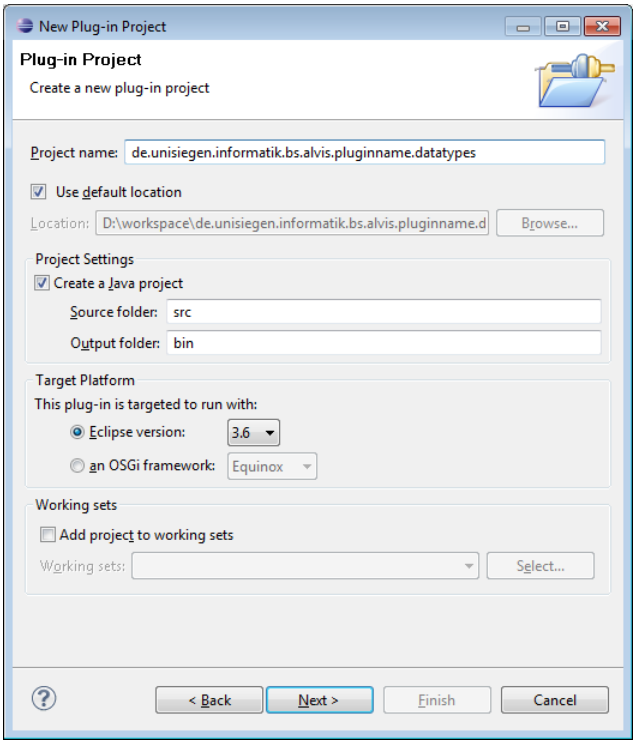
Die langen Namen kommen von der RCP Konvention seine Plug-ins nach der Internetadresse (alvis.bs.informatik.uni-siegen.de) zu benennen, auf der man diese veröffentlicht.

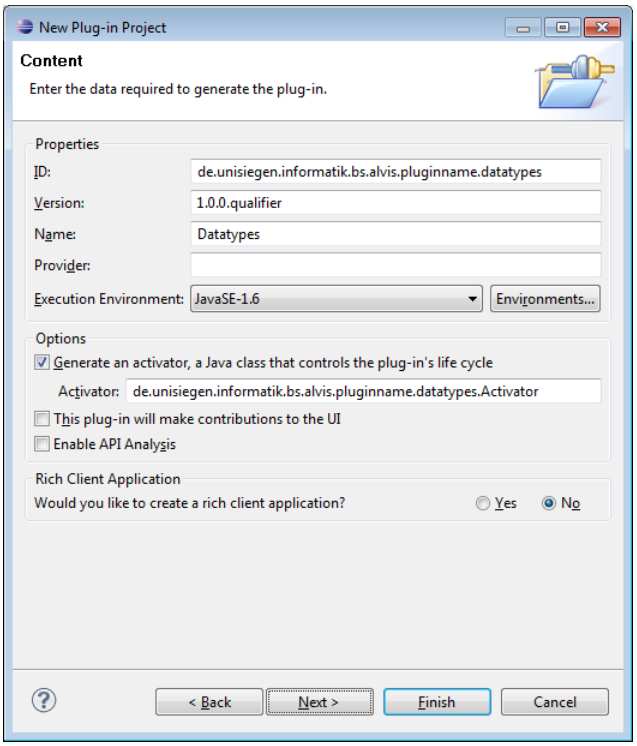
5 Bestandteile der Plug-ins

Primär stellt ein Plug-in nur die neuen Datentypen zur Verfügung. Daher ist der Kern des neuen Plug-ins das Plug-in *de.unisiegen.informatik.bs.alvis.pluginname.datatypes*. Im Plug-in *de.unisiegen.informatik.bs.alvis.pluginname* werden die Elemente implementiert, die es ermöglichen die neuen Datentypen zu visualisieren.

6 *de.unisiegen.informatik.bs.alvis.pluginname.datatypes*

Dieses Plugin stellt nur Java-Klassen für andere Plug-ins zur Verfügung. Wir erstellen daher ein neues Plug-in Project mit den folgenden Einstellungen:





6.1 Erstellen von Klassen für Datentypen

Als nächstes erstelle man für jeden Datentyp eine Klasse *PCDatentypname*. Diese Klasse muss von *PCObject* erben. Es ist wichtig darauf zu achten, dass der Klassenname mit den Buchstaben *PC* beginnt. Dies ist für das Funktionieren des Compilers nötig.

Listing 1: Klasse PCDatentyp

```
1  package de.unisiegen.informatik.bs.alvis.pluginname.datatypes;
2
3  import de.unisiegen.informatik.bs.alvis.primitive.datatypes.PCObject;
4
5  public class PseudoCodeDatentypname extends PCObject {
6          //Hier die Logik des Datentyps implementieren
7
8          // Methoden von PCObject
9          // TODO: Implementieren !
10         public String toString() {
11                 return null;
12         }
13
14         public String getTypeName() {
15                 return null ;
16         }
17
```

```
18          public PCObject set(String memberName, PCObject value) {
19                  return null;
20          }
21
22          public boolean equals(PCObject toCheckAgainst) {
23                  return false;
24          }
25  }
```

Man implementiere die abstrakten Methoden und überschreibe je nach Bedarf weitere Methoden. Natürlich müssen noch weitere Attribute und Methoden hinzugefügt werden, welche die Logik des Datentypen stellen. Als Beispiel sehe man sich den Datentypen Graph[4] aus dem Graph Plugin an. Dieser beinhaltet lediglich Knoten und Kanten. Die definierte Schnittstelle
public PCObject get(String memberName);
ermöglicht dann den Zugriff auf die Attribute.

6.2 Erstellen von Interfaces zur graphischen Repräsentation

Zu diesem Datentyp erstelle man nun ein Interface *GraphicalRepresentation-Datentypname*. Dieses Interface muss von *GraphicalRepresentation* erben.

Listing 2: Interface GraphicalRepresentationDatentyp

```
1  package de.unisiegen.informatik.bs.alvis.graph.datatypes;
2
3  import de.unisiegen.informatik.bs.alvis.primitive.datatypes
4                  .GraphicalRepresentation;
5
6  public interface GraphicalRepresentationDatentypname
7          extends GraphicalRepresentation {
8
9          // Hier benoetigten Methoden angeben.
10
11 }
```

Dieses Interface wird nachher von der graphischen Repräsentation des neuen Datentyps implementiert. Hier ist es möglich die Schnittstellen zu definieren, welche die graphische Repräsentation auf den reinen Datentypen abbilden. Die graphische Repräsentation eines Datentypen enthält in der Regel noch weitere Metadaten (x- und y-Koordinaten auf dem Bildschirm, Schriftart...). Zum besseren Verständnis sehe man sich hier ebenfalls die Implementierung des Graph Plug-ins an.

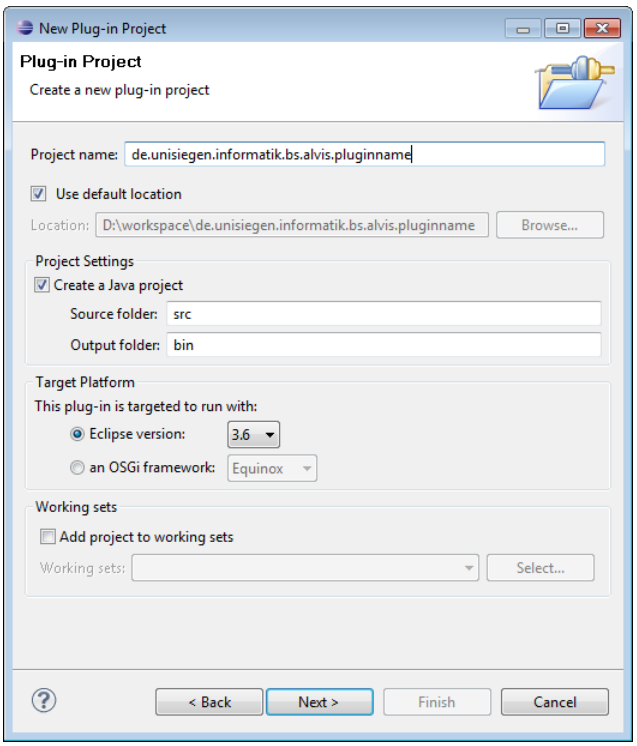
7 *de.unisiegen.informatik.bs.alvis.pluginname*

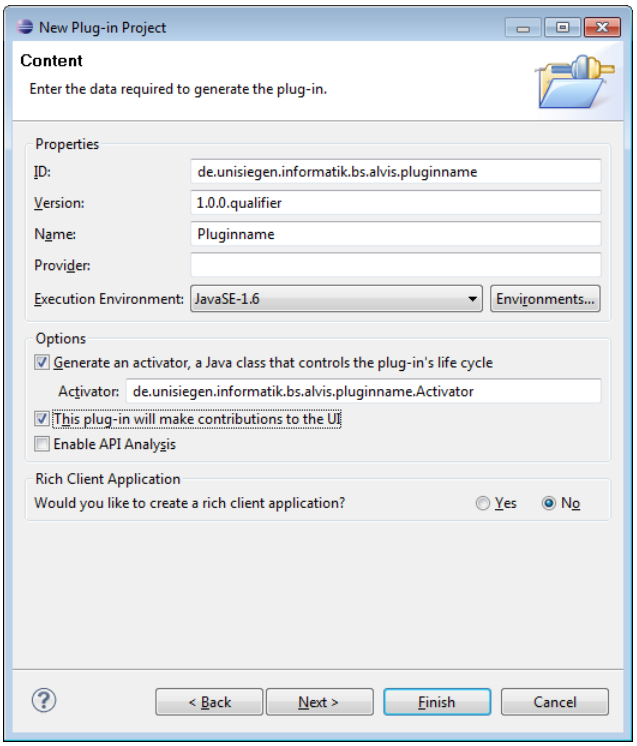
Als möchte ich einen Überblick geben, welche verbleibenden Teile in diesem Plug-in zusammengefasst werden.

[4] de.unisiegen.informatik.bs.alvis.graph.datatypes.PCGraph

1. Für jeden **Datentyp** wird mindestens eine eigene Klasse erstellt, die diesen Datentyp visualisiert.
2. Es wird ein **Editor** erstellt, der es ermöglicht ein Beispiel zu erstellen.
3. Es wird ein Klasse erstellt, die den Benutzer nach **Einstellungen** für den **Run** fragt.
4. Es wird eine Klasse erstellt, die die **Visualisierung** der Datentypen im **Run**-View ermöglicht.
5. Es wird eine Klasse erstellt, die es ermöglicht den momentanen Zustand des Datentyps zu **exportieren**.

Dieses Plug-in beinhaltet Klassen, die das Userinterface erweitern deswegen setzte man dieses mal bei *„This plug-in will make contributions to the UI"* einen Haken.





7.1 Visualisierung des Datentyps

Klassen zur Visualisierung des Datentyps wollen wir in das Package *„de.unisiegen.informatik.bs.alvis.pluginname.graphicalrepresentations”* ablegen.

Zur Visualisierung von Datentypen benötigt man eine Klasse, welche man an einen Editor oder an einen View hängen kann. Dazu muss diese Klasse einen Konstruktor der Form Klasse(Composite parent, ...) haben. Die neue Klasse kann sich dazu aus dem Paket „org.eclipse.swt.widgets[5]” bedienen.
Damit aus der graphischen Repräsentation muss die Klasse das von *GraphicalRepresentation*[6] abgeleitetes Interface aus dem Plug-in *„de.unisiegen.informatik.bs.alvis.pluginname.datatypes”* implementieren.

7.2 Erstellen eines Editors für ein Beispiel

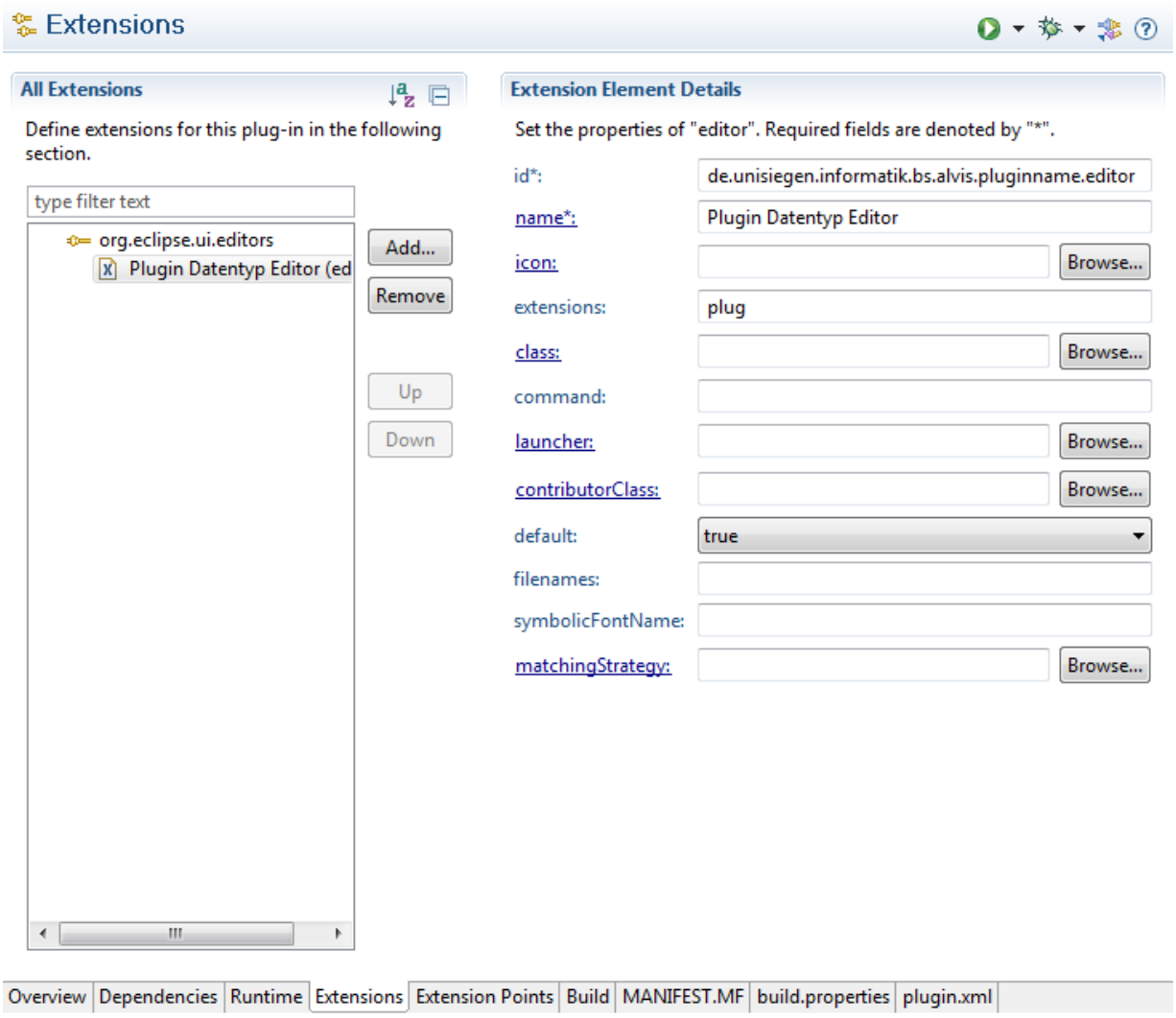
Editoren für unser Beispiel wollen wir in das Package *„de.unisiegen.informatik.bs.alvis.pluginname.editors”* ablegen.

Wir wollen zum Anlegen und Bearbeiten eines Beispiels einen Editor verwenden. Der Editor speichert unser Beispiel in serialisierter Form in eine Datei.
Wir fügen nun in der „MANIFEST.MF” die Extension „org.eclipse.ui.editors” und

[5] http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/widgets/package-summary.html

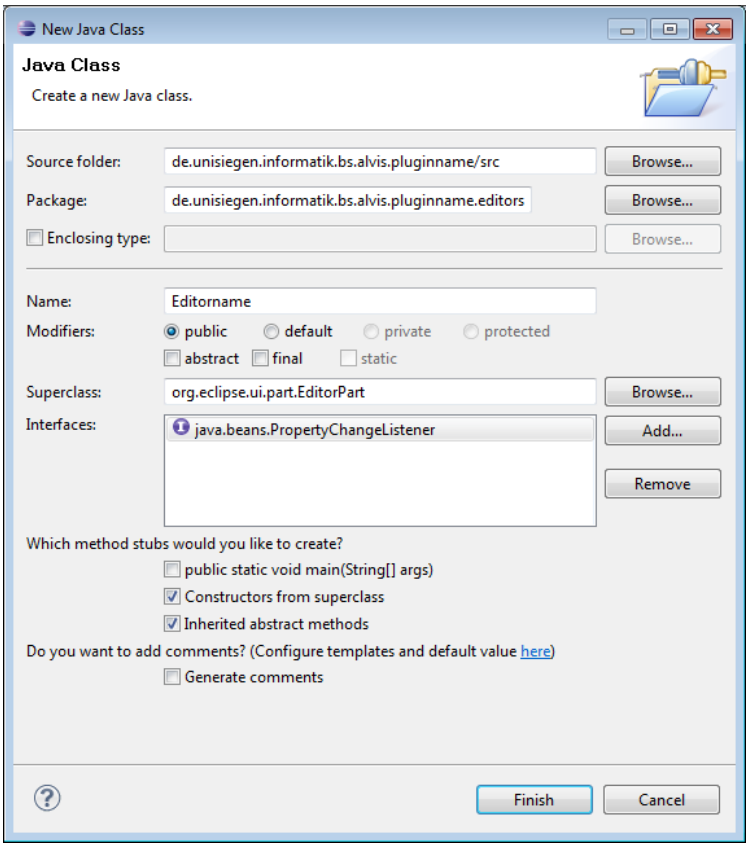
[6] Siehe dazu: Erstellen von Interfaces zur graphischen Repräsentation

einen neuen Editor hinzu.



Die *id* ist auch hier möglichst aussagekräftig zu wählen.
Als *name*:* wähle man einen Namen. Dieser wird als Editor Name angezeigt, wenn man eine Datei per „öffnen mit...” öffnet.
extensions: ist die Dateiendung (ohne .), für die dieser Editor zuständig ist.
default: gibt an, ob der Editor der Standard Editor für die Extension ist.

Mit einem klick auf class: wird die Implementierung für den Editor hinzugefügt.



Um Ordnung in dem Plugin zu halten erstelle man den Editor im Package *de.unisiegen.informatik.bs.alvis.pluginname.editors*.
Weiterhin lässt man den Editor das Interface *PropertyChangeListener* implementieren.
Nach einem Klick auf *Finish"* öffnet sich die Implementierung des Editors.

Listing 3: Editor Editorname

```
1  package de.unisiegen.informatik.bs.alvis.pluginname.editors;
2
3  import java.beans.PropertyChangeEvent;
4  import java.beans.PropertyChangeListener;
5
6  import org.eclipse.core.runtime.IProgressMonitor;
7  import org.eclipse.swt.widgets.Composite;
8  import org.eclipse.ui.IEditorInput;
9  import org.eclipse.ui.IEditorSite;
10 import org.eclipse.ui.PartInitException;
11 import org.eclipse.ui.part.EditorPart;
12
13 public class EditornameextendsEditorPart
14                     implements PropertyChangeListener {
15
```

```
16          public Editorname() {
17          }
18
19          public void init(IEditorSite site, IEditorInput input)
20          throws PartInitException {
21          }
22
23          public void createPartControl(Composite parent) {
24          }
25
26          public boolean isDirty() {
27                  return false;
28          }
29
30          public void doSave(IProgressMonitor monitor) {
31          }
32
33          //UnusedMethods
34          public void propertyChange(PropertyChangeEvent arg0) {}
35          public void doSaveAs() {}
36          public boolean isSaveAsAllowed() { returnfalse; }
37          public void setFocus() {}
38  }
```

Um den Editor zu verstehen stelle man sich vor der Benutzer hätte eine Datei aus dem Package Explorer mit diesem Editor geöffnet. Der Ablauf der Methodenaufrufe sieht dann folgendermaßen aus:[7]

1. Der Editor wird per Konstruktor erzeugt.
2. init(IEditorSite, IEditorInput) wird aufgerufen und das für uns interessante IEditorInput Objekt wird übergeben.
3. createPartControl(Composite) wird aufgerufen und das Parent, dem wir unsere Visualisierungen hinzufügen wird übergeben.

Auf Basis dieses Wissens erweitern wir den Editor um einige Funktionen. Wir fügen private Attribute hinzu, die die übergebenden Objekte speichern. In der init-Methode werden einige setMethoden aufgerufen und die privaten Attribute gesetzt.

In der createPartCotrol-Methode wird das Parent gesetzt und eine Methode zum erstellen des Editors aufgerufen.

Die Methode *createEditorFromInput(...)* ist für das laden des Beispiels zuständig. Falls myInput bereits ein gespeichertes Beispiel beinhaltet hat, ist es die Aufgabe dieser Methode den gespeicherten Zustand des Beispiels wieder herzustellen. Falls die Datei frisch erstellt wurde so muss diese Methode einen leeren Editor Zustand herstellen.

[7] Mehr Informationen zum Editor unter http://www.ralfebert.de/eclipse_rcp/editors/

Die *doSave*-Methode serialisiert[8] den derzeitigen Zustand des Editors in die geöffnete Datei.

Listing 4: Editor Editorname erweitert

```
1  package de.unisiegen.informatik.bs.alvis.pluginname.editors;
2
3  import java.beans.PropertyChangeEvent;
4  import java.beans.PropertyChangeListener;
5
6  import org.eclipse.core.runtime.IProgressMonitor;
7  import org.eclipse.swt.widgets.Composite;
8  import org.eclipse.ui.IEditorInput;
9  import org.eclipse.ui.IEditorPart;
10 import org.eclipse.ui.IEditorSite;
11 import org.eclipse.ui.PartInitException;
12 import org.eclipse.ui.part.EditorPart;
13 import org.eclipse.ui.part.FileEditorInput;
14
15 public classEditornameextendsEditorPart
16                 implements PropertyChangeListener {
17
18         private IEditorInput myInput;
19         private String myInputFilePath;
20         private Composite myParent;
21         private boolean dirty;
22
23         public Editorname() {
24         }
25
26         public void init(IEditorSite site, IEditorInput input)
27         throws PartInitException {
28                 setSite(site);
29                 setInput(input);
30                 // Set the title to the filename
31                 setPartName(input.getName());
32                 myInput = input;
33                 myInputFilePath = ((FileEditorInput) input)
34                         .getPath().toString();
35         }
36
37         public void createPartControl(Compositeparent) {
38                 this.myParent = parent;
39                 createEditorFromInput(this.myParent, this.myInput);
40         }
41
42         private void createEditorFromInput(Composite parent,
43                 IEditorInput input) {
44                 // if input contains an object
```

[8]Es wird empfohlen http://xstream.codehaus.org/ zum Serialisieren zu verwenden.

```
45                  // then put the object to parent
46                  // else put a new object to parent
47          }
48
49          // call this method when you change your code
50          private void checkDirty() {
51                  // if myInput != current state of editor
52                  // then setDirty(true);
53                  // else setDirty(false);
54          }
55
56          // setDirty and fire change
57          private void setDirty(boolean dirty) {
58                  this.dirty = dirty;
59                  firePropertyChange(IEditorPart.PROP_DIRTY);
60          }
61
62          // return dirty
63          public boolean isDirty() {
64                  return this.dirty;
65          }
66
67          public void doSave(IProgressMonitor monitor) {
68                  // save the current state of the editor to
69                  // the file at myInputFilePath;
70                  // setDirty(false);
71          }
72
73          // Unused Methods
74          public void propertyChange(PropertyChangeEvent arg0) {}
75          public void doSaveAs() {}
76          public boolean isSaveAsAllowed() { returnfalse; }
77          public void setFocus() {}
78  }
```

7.3 Registrieren eines Wizards

Damit der Benutzer auch eine Möglichkeit hat Dateien des neuen Typs anzulegen benötigt man einen **Wizard** für diese Dateiendung. Ein Wizard ist ein Dialog, der es dem Benutzer ermöglicht Dateien eines bestimmten Typs anzulegen. Dieser muss die Extension *„org.eclipse.ui.newWizards"* bedienen.

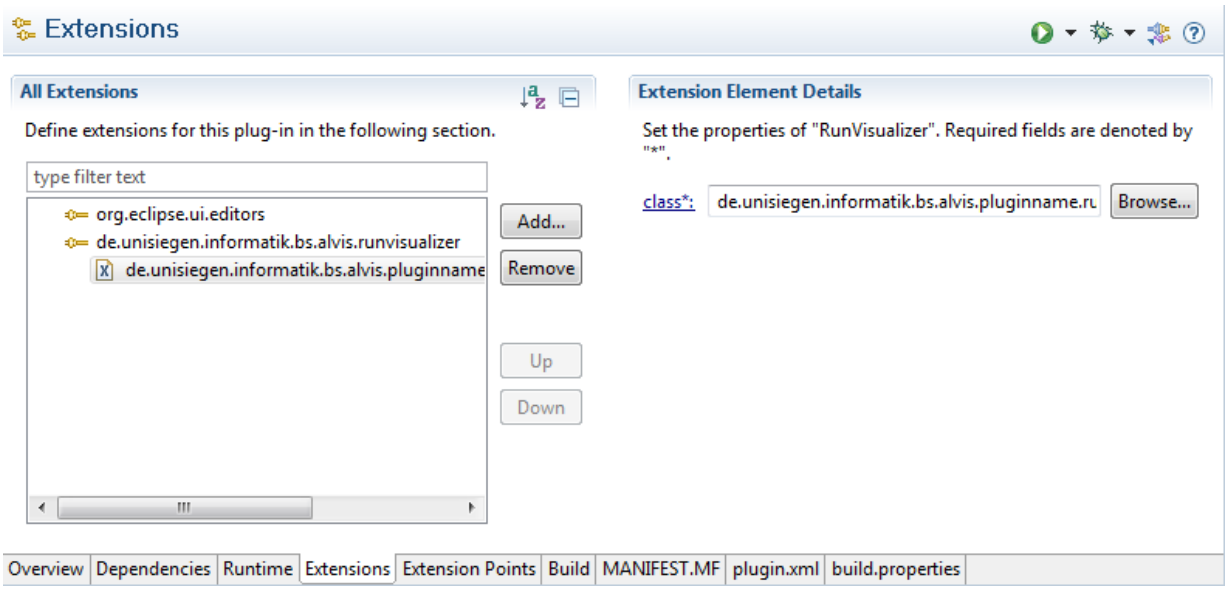
7.4 Visualisierung des Runs

Wenn der Benutzer den Algorithmus und sein Beispiel definiert hat und einen Run startet wird nach Plug-ins gesucht, die die Extension „de.unisiegen.informatik.bs.alvis.runvisualizer" erweitern. Diesen Plug-ins werden dann alle Datentypen und das Parent herunter gegeben. Das Plug-in prüft dann, ob es sich bei einem der Datentypen

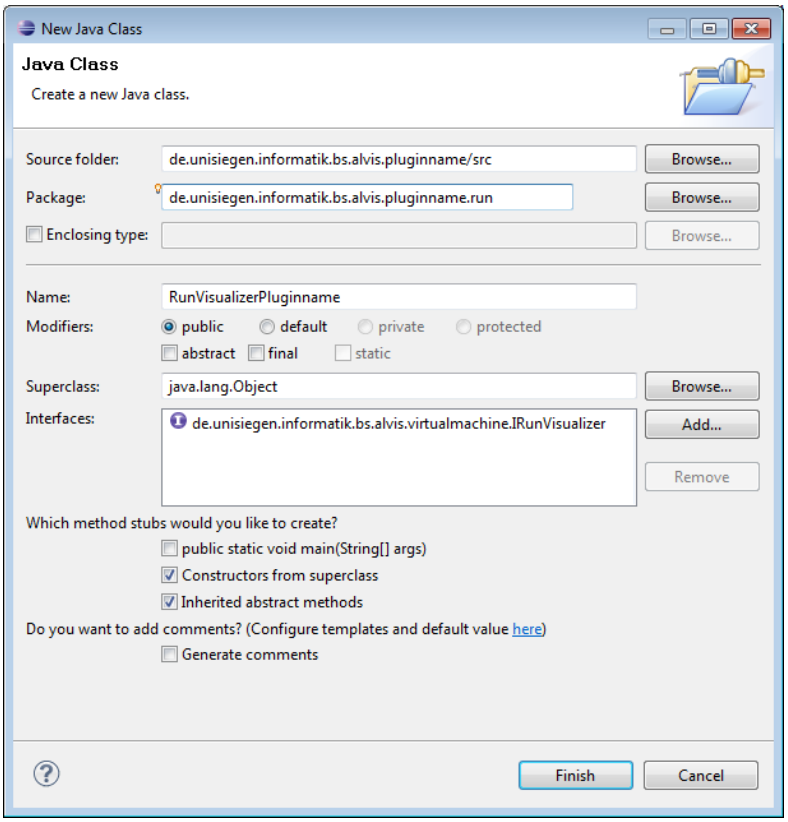
um einen Datentyp des Plug-ins handelt und hängt dann die Visualisierung seiner Datentypen an das Parent.
Diesen Extension Point wollen wir nun in unserem Plug-in erweitern.
Dazu öffnen wir als nächstes die „MANIFEST.MF” in unserem Plug-in und fügen eine „Dependencie” zu „de.unisiegen.informatik.bs.alvis” hinzu. Danach wählen den Reiter „Extensions” aus.

Wir fügen den Extension Point „de.unisiegen.informatik.bs.alvis.runvisualizer” aus dem Plug-in „de.unisiegen.informatik.bs.alvis” hinzu.

Mit der rechten Maustaste auf den neu hinzugefügten Extension Point lässt sich unter „New” ein neuer „RunVisulizer” hinzufügen.



Der Link „*class*:*” ermöglicht es uns eine neue Klasse anzulegen, die folgende Anforderungen erfüllt.



Wir bestätigen die Einstellungen mit einem Klick auf „Finish” und erhalten eine Klasse mit folgendem Inhalt.

Listing 5: Klasse RunVisualizerPluginname

```
1  package de.unisiegen.informatik.bs.alvis.pluginname.run;
2
3  import org.eclipse.swt.widgets.Composite;
4
5  import de.unisiegen.informatik.bs.alvis.
6          extensionpoints.IRunVisualizer;
7
8  public class RunVisualizerPluginname implements
9          IRunVisualizer {
10
11         public RunVisualizerPluginname() {}
12
13         @Override
14         public boolean addVisualizing(Composite parent,
15                 String inputFile){
16                 return false;
17         }
18 }
```

Hier gehen wir genau so vor, wie wir es schon bei der Load-Methode des

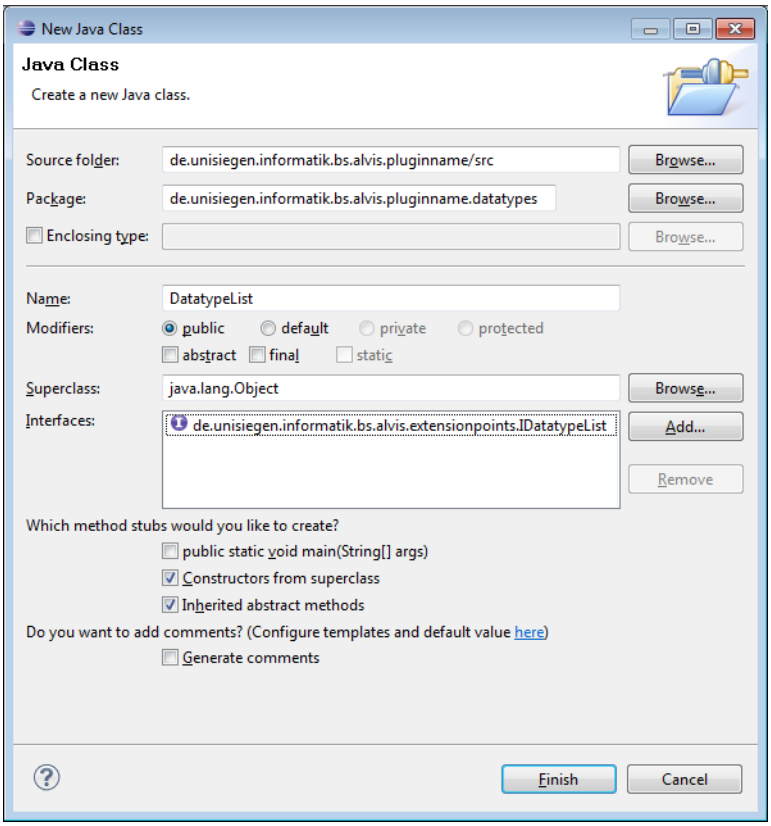
Editors gemacht haben.

7.5 Datentypen registrieren

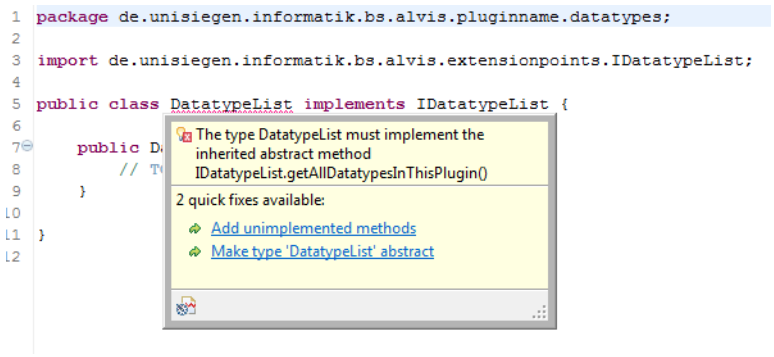
Als letztes muss unser Plug-in den anderen Plug-ins noch mitteilen können, welche Datentypen es beinhaltet. Dafür gibt es natürlich auch wieder eine definierte Schnittstelle.

Wir öffnen die „MANIFEST.MF" und fügen die Extension „de.unisiegen.informatik.bs.alvis.extensionpoints.datatypelist" hinzu. Wir wählen „New" → „DatatypeList" und klicken den Link „class*:".

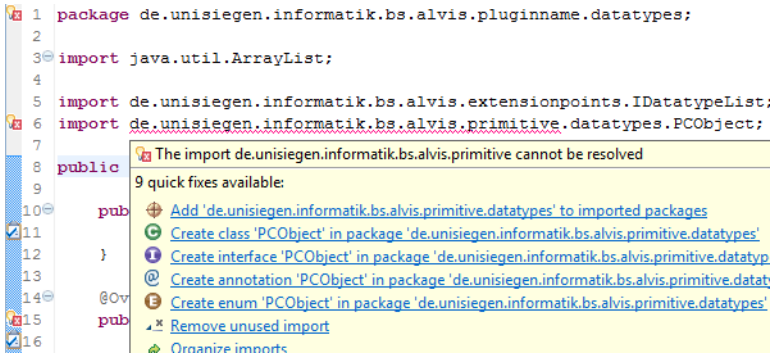
Als nächstes öffnet sich ein neues Fenster, dass uns anbietet eine neue Klasse anzulegen. Wir legen die Klasse in das Package „de.unisiegen.informatik.bs.alvis.pluginname.datatypes".



Anschließend bestätigen wir mit „Finish" werfen einen Blick auf die Implementierung. Sollte Eclipse sich über fehlende Methoden beschweren, so kann man den Mauszeiger über den Klassennamen halten und auf das Kontext Menü warten. Anschließend lassen sich die fehlenden Methoden mit einem Klick auf den Link „*Add unimplemented methods*" hinzufügen.



Wir wollen sichergehen, dass der Import von PCObject[9] funktionieren. Wir ändern auf „de.unisiegen.informatik.bs.alvis.primitive.datatypes.PCObject". Und fügen „de.unisiegen.informatik.bs.alvis.primitive.datatypes" via „Add ... to imported packages" zu den „Imported packages" hinzufügen.



Danach sollte die Klasse in etwa so aussehen.

Listing 6: Klasse DatatypeList

```
1  package de.unisiegen.informatik.bs.alvis.pluginname.datatypes;
2
3  import java.util.ArrayList;
4
5  import de.unisiegen.informatik.bs.alvis.extensionpoints.IDatatypeList;
6  import de.unisiegen.informatik.bs.alvis.primitive.datatypes.PCObject;
7
8  public class DatatypeList implements IDatatypeList {
9
10         publicDatatypeList() {
11         }
12
13         public ArrayList<PCObject> getAllDatatypesInThisPlugin() {
14                 return null;
15         }
16 }
```

[9] de.unisiegen.informatik.bs.alvis.primitive.datatypes.PCObject

Nun wollen wir noch die Methode *getAllDatatypesInThisPlugin()* implementieren. In dieser Methode geben wir ein Null-Object von jedem Datentyp den wir erstellt haben in einer ArrayList zurück.

Listing 7: Klasse DatatypeList

```
1  package de.unisiegen.informatik.bs.alvis.pluginname.datatypes;
2
3  import java.util.ArrayList;
4
5  import de.unisiegen.informatik.bs.alvis.extensionpoints.IDatatypeList;
6  import de.unisiegen.informatik.bs.alvis.primitive.datatypes.PCObject;
7  // Import all datatypes
8  import de.unisiegen.informatik.bs.alvis.pluginname.datatypes.*;
9
10 public class DatatypeList implements IDatatypeList {
11
12         public DatatypeList() {}
13
14         public ArrayList<PCObject>
15                 getAllDatatypesInThisPlugin() {
16                 ArrayList<PCObject> datatypeList =
17                         new ArrayList<PCObject>();
18                 datatypeList.add(
19                         PseudoCodeDatentypname.getNull());
20                 // If you have more datatypes add them also.
21                 return datatypeList;
22         }
23 }
```

8 Anhang

In diesem Anhang möchte ich noch auf ein paar Dinge aufmerksam machen, die uns bei der Arbeit mit RCP einiges an Zeit gekostet haben, eigentlich aber recht simpel zu lösen waren.

8.1 Der Klabautermann

Es kam vor, dass das Projekt einfach nicht gestartet ist, obwohl das Projekt frisch ausgecheckt wurde. Das Projekt lief auf dem einen PC und auf dem anderen PC nicht. Man wird nach 300 Stunden mit RCP abergläubig, aber wir sind uns sicher, dass das Problem eine Ursache hat:
Der Klabautermann ist in deinem Workspace und treibt dort sein Unwesen.

Wenn du festgestellt hast, dass der Klabautermann in deinem Workspace sein Unwesen treibt...

- markiere alle Plugins, klicke dann dann auf „Project”, dann auf „Clean...”.
- starte Eclipse neu.

- refresh den Workspace indem du alle Plugins markierst und „F5” drückst.

8.2 Probleme beim Start

Hast du beim Start des Projekts Probleme kann dir vielleicht eine der folgenden Lösungen helfen.

8.2.1 Fehlende Plugins

Wird beim Start hingewiesen, dass Plugins fehlen, dann Wähle „Run” -> „Run Configurations...”. Es öffnet sich ein neues Fenster. Es kommt vor, dass nicht alle Buttons sichtbar sind. Ziehe das Fenster so groß wie nötig um alle Elemente zu sehen. Anschließend wählst du im linken Teil des Fensters unter „Eclipse Application” die Application die du starten möchtest. Anschließend klickst du auf der rechten Seite auf den Reiter „Plug-ins” und drückst den Button „Add Required Plug-ins”.

8.2.2 .jar Dateien einbinden

Um externe .jar Dateien in ein Plugin einzubinden darf man nicht, wie sonst in Java üblich, das jar einfach zum Build Path hinzufügen, sondern man muss es in der MANIFEST.MF unter „Runtime” unter „Classpath” einbinden.