FeatureIDE Refactoring-PlugIn

Das FeatureIDE Refactoring-PlugIn ist ein Eclipse-PlugIn, welches featureorientierte Refaktorisierungen auf jak-Dateien ermöglicht. Diese Refaktorisierungen basieren auf einem, ebenfalls featureorientiert geschriebenen, Typsystem, das die jak-Dateien mit Hilfe von abstrakten Syntaxbäumen analysiert und verwaltet. Zur Visualisierung der Refaktorisierungen wurde das PlugIn in FeatureIDE integriert.

Benötigte PlugIns

Damit das Refactoring-PlugIn laufen kann, werden einige PlugIns benötigt:

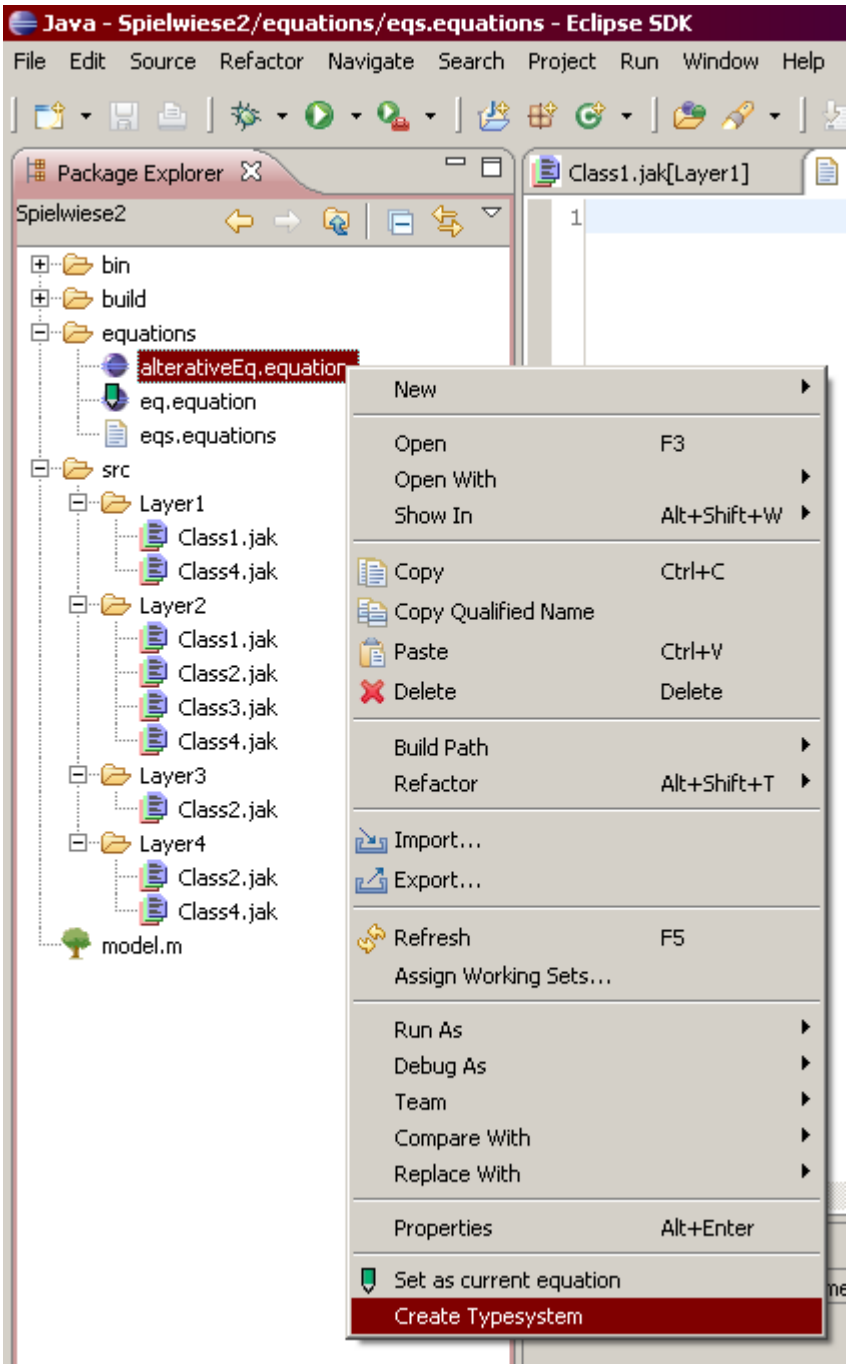
```
FeatureIDE_Core
FeatureIDE_UI
org.eclipse.core.filesystem
org.eclipse.core.resources
org.eclipse.core.runtime
org.eclipse.jdt.ui
org.eclipse.jface.text
org.eclipse.osgi
org.eclipse.ui
org.eclipse.ui.editors
org.eclipse.ui.ide
```

Die ersten beiden PlugIns werden durch Installation von FeatureIDE bereitgestellt. FeatureIDE kann mittels Eclipse unter http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/deploy/ heruntergeladen werden. Die anderen PlugIns werden standardmäßig von Eclipse bereitgestellt.

Wenn das PlugIn nicht schon mit FeatureIDE installiert wurde, genügt es die entsprechende jar-Datei in den PlugIn-Ordner zu kopieren (z.B. "Eclipse 3.4.2\plugins\") und Eclipse neu zu starten. Sollte das PlugIn danach nicht funktionieren (erkennbar daran, dass die entsprechenden Menu-Punkte nicht angezeigt werden), muss Eclipse einmalig über die Konsole mit dem Befehl "eclipse.exe –clean" gestartet werden, damit das PlugIn erkannt wird.

Initialisierung des Typsystems

Bevor die gewünschten Refaktorisierungen ausgeführt werden können, muss das Typsystem initialisiert werden. Dies geschieht automatisch, sobald eine Refaktorisierung durchgeführt werden soll. Dabei wird die momentan von FeatureIDE aktivierte equation[s]-Datei geparst um alle Layer zu erhalten, die das Typsystem verwalten soll. Soll alternativ eine andere Konfiguration als Basis für das Typsystem dienen, reicht ein Rechtsklick auf die entsprechende equation[s]-Datei und anschließende ein Klick auf "Create Typesystem", um das Typsystem zu erstellen.



Damit dieser Schritt allerdings erfolgreich ist, müssen einige Punkte beachtet werden:

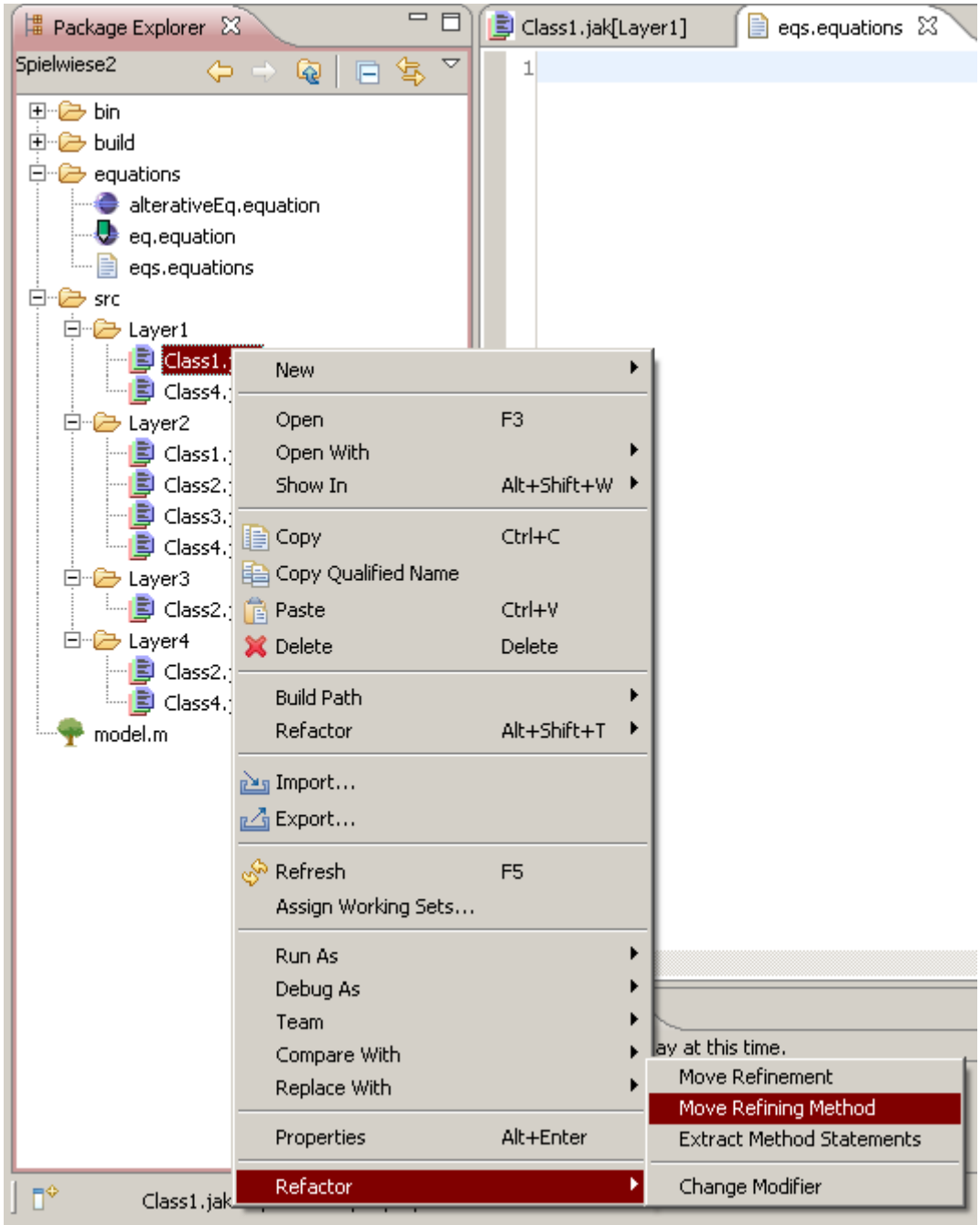
- Die zu parsende equation(s)-Datei muss im Ordner "equations" sein.
- Mindestens eine equation(s)-Dateien muss dort vorhanden sein.
- Refaktorisierungen an jak-Dateien können nur ausgeführt werden, wenn deren Layern in der equation(s)-Datei spezifiziert wurde
- Die Namen der Layer müssen ihrem Verzeichnisnamen entsprechen, d.h. wenn ein Layer Layer1 heißt, dann müssen dessen jak-Dateien im Verzeichnis "src/Layer1/" liegen.

Wie bereits angedeutet können auch equations-Dateien benutzt werden. Bei diesen prüft das PlugIn, ob in den dort spezifizierten Programmen irgendwelche Zyklen auftreten. Wenn dies nicht der Fall ist kann das Typsystem auch aus diesen Dateien die benötigten Layer extrahieren.

Refaktorisierungen

Die Refaktorisierungen, die mit dem PlugIn möglich sind, sind:

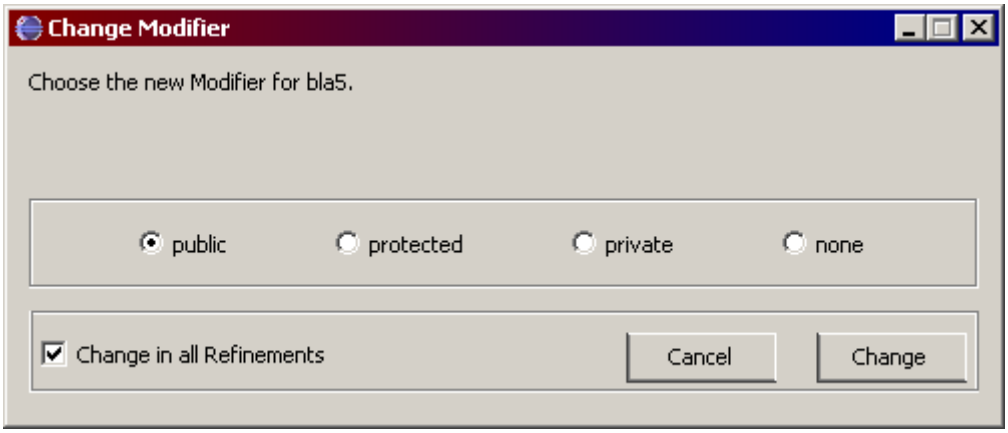
- Change Modifier
- Move Refinement
- Move Refining Method
- Extract Method Statements



Bevor eine Refaktorisierung durchgeführt werden kann, wird geprüft ob dadurch irgendwelche Konflikte entstehen. Sollte dies der Fall sein, werden diese tabellarisch aufgezählt und können durch einen Doppelklick auf den entsprechenden Konflikt angezeigt werden.

Change Modifier

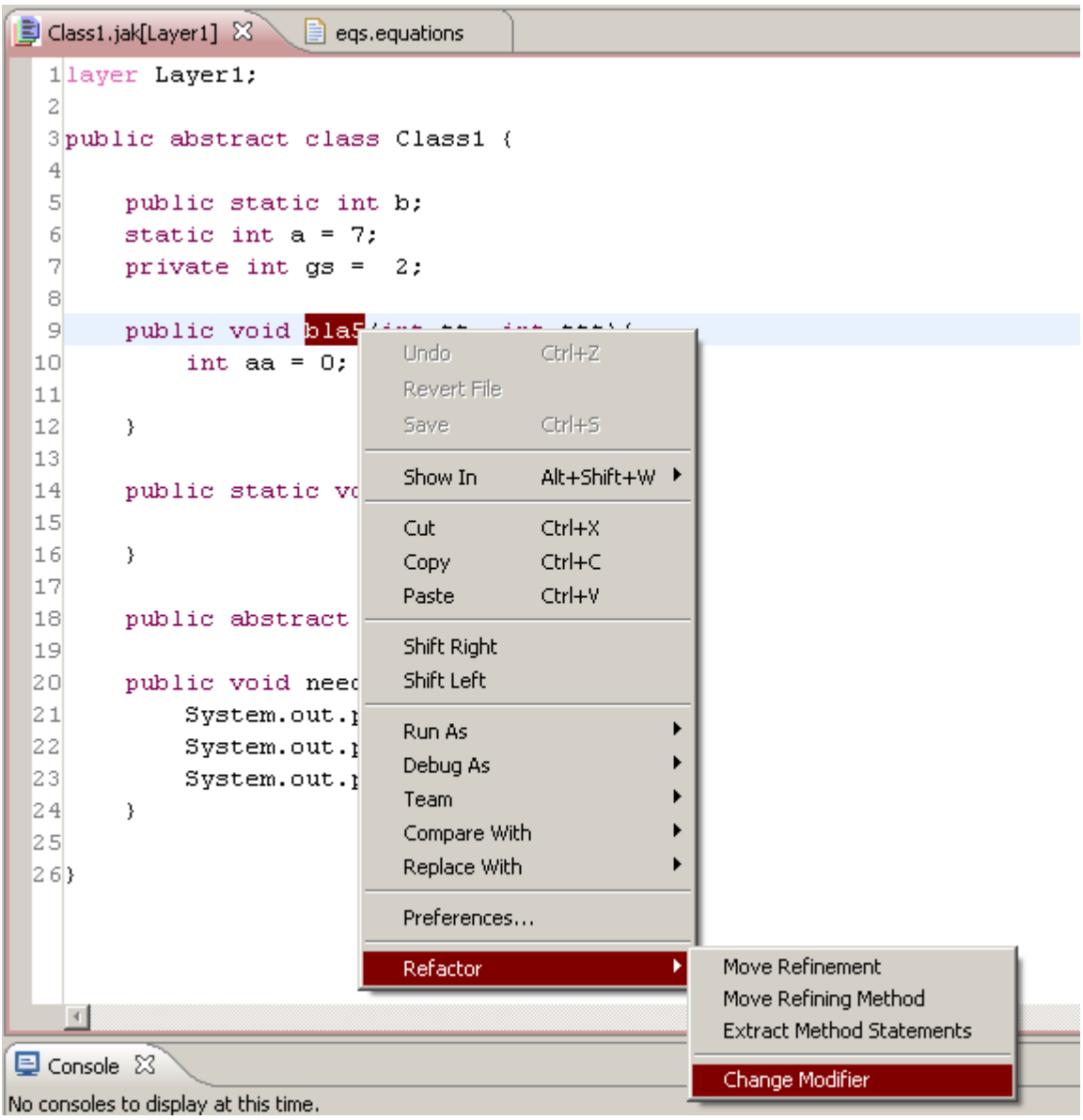
Change Modifier ändert die Sichtbarkeitsmodifier (private, protected, public, packageprivate) von Klassen, Methoden und Attributen. Bei Klassen wird darauf geachtet, dass diese nur public oder packageprivate sein können. Die Änderungen können durch die Option "Change in all Refinements" wahlweise nur in der aktuellen Klasse oder in allen Refinements einschließlich der Basisklasse durchgeführt werden.



Die Konflikte, die vor der Refaktorisierung erkannt werden, sind:

- Falls die Sichtbarkeit eingeschränkt wird, werden Methodenaufrufe und Feldzugriffe aus anderen Klassen, die dann nicht mehr möglich wären, sowie abstrakte Methoden, die dann nicht mehr erreichbar wären, ermittelt.

Die Refaktorisierung wird aktiviert, indem man den Namen der betroffenen Klasse, Methode bzw. des Attributs markiert und "Change Modifier" im Menu "Refactor" wählt. Wenn man keinen Text markiert oder die Refaktorisierung im Package Explorer aktiviert, wird der Sichtbarkeitsmodifier der Klasse geändert.



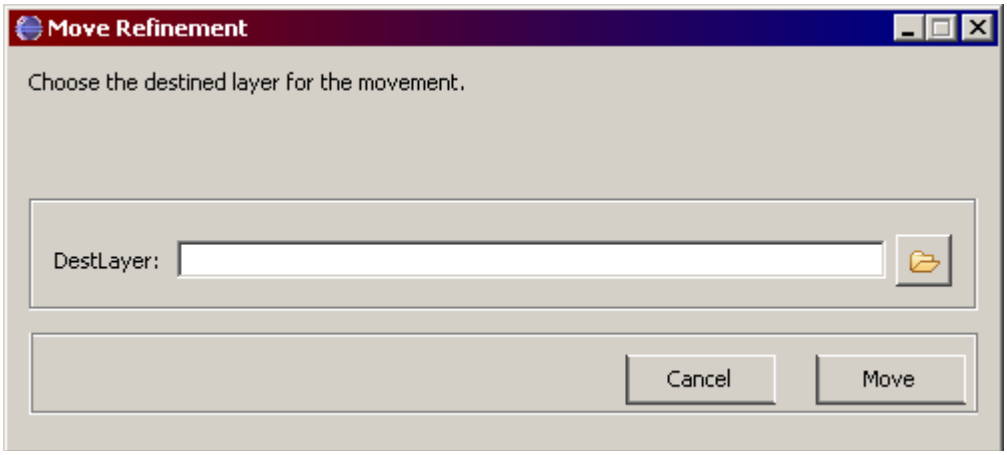
Move Refinement

Move Refinement verschiebt eine Refinementklasse, d.h. eine Klasse die eine Basisklasse erweitert, in ein anderes Layer. Wenn das angegebene Ziellayer noch nicht existiert, wird es angelegt und wenn im Ziellayer schon eine Klasse mit dem selben Namen existiert, wird sie überschrieben.

Die Konflikte, die vor der Refaktorisierung erkannt werden, sind:

- Das extrahierte Refinement darf keine neuen Methoden definieren.

Die Refaktorisierung wird mit einem Rechtsklick auf die entsprechende Klasse und der Option "Move Refinement" im Menu "Refactor" aktiviert.



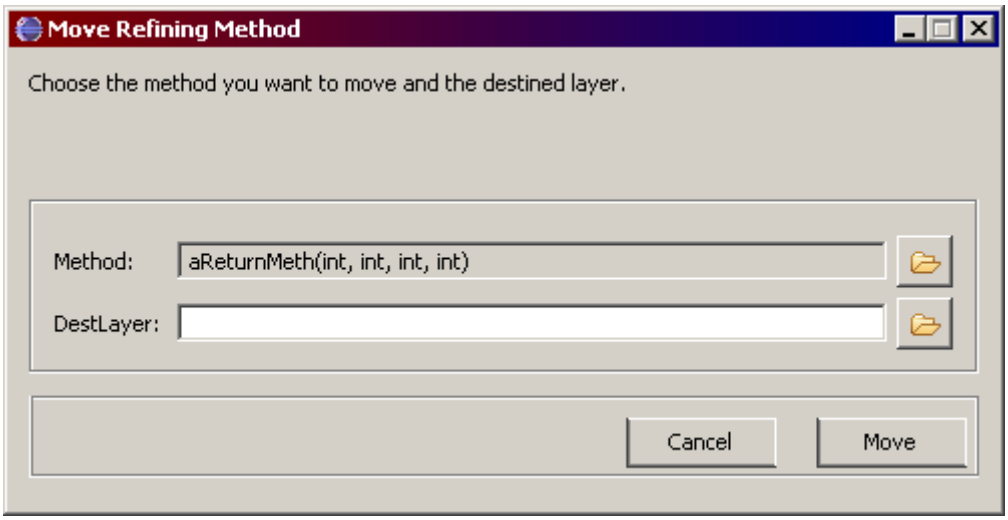
Move Refining Method

Move Refining Method verschiebt eine Refinementmethode, d.h. eine Methode die eine Methode aus einem anderen Layer verfeinert, in eine andere Refinementklasse. Wenn das Ziellayer oder die Refinementklasse noch nicht existieren, werden sie angelegt. Das Extrahieren von Konstruktoren wird noch nicht unterstützt. Methoden mit dem selben Namen und den selben Parametern dürfen noch nicht in der Zielklasse vorhanden sein.

Die Konflikte, die vor der Refaktorisierung erkannt werden, sind:

- Zugriffe innerhalb der extrahierten Methode auf private Objektattribute der Ausgangsklasse
- Zugriffe innerhalb der extrahierten Methode auf private Methoden der Ausgangsklasse

Die Refaktorisierung wird aktiviert, indem man den Namen der zu verschiebenden Methode markiert und die Option "Move Refining Method" im Menu "Refactor" wählt. Markiert man den Namen nicht, muss die Methode im folgenden Fenster manuell ausgewählt werden.



Extract Method Statements

Extract Method Statements extrahiert Codefragmente aus einer Methode in eine Refinementklasse. Die Methode darf im Zielrefinement noch nicht existieren. Die gewählten Codefragmente müssen sich alle im selben Scope befinden (und die Anfangszeile des Codebereichs muss größergleich der Endzeile sein). Da das Refactoring auf der Basis von Zeilennummern geschieht, müssen alle neuen Statements in einer neuen Zeile erfolgen, d.h. folgendes Statement könnte nicht extrahiert werden:

public void Methode1() { Methode2(); }

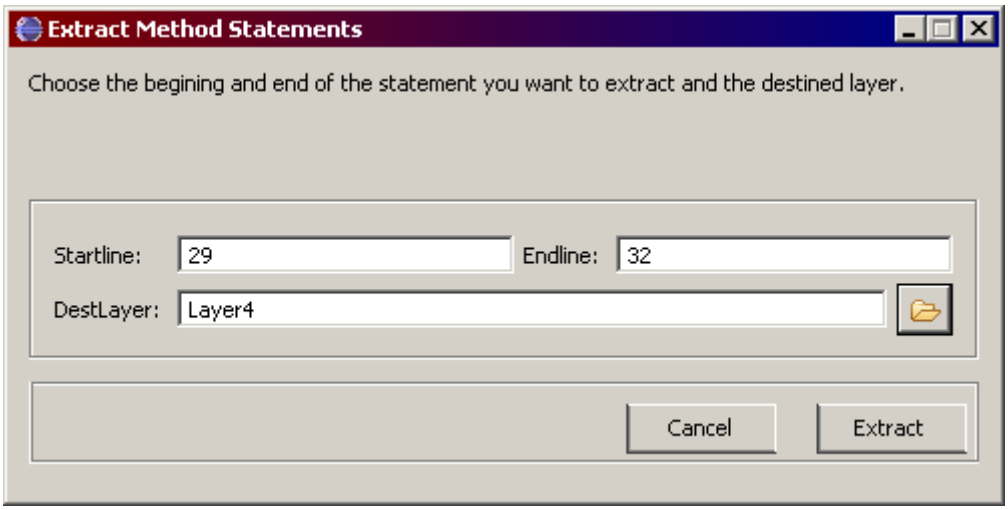
Wenn das Ziellayer bereits existiert, muss es innerhalb der equation(s)-Datei nach dem Ausgangslayer definiert wurden sein, wenn es noch nicht existiert, wird es angelegt. Je nach Situation werden die Statements einfach in die Methode des neuen Refinements, in eine Hook-Methode oder in ein Methodenobjekt extrahiert. Falls die Extraktion in eine Hook-Methode erfolgt, kann der Name dieser Methode festgelegt werden.

Die zu verwendende Strategie wird folgendermaßen festgelegt:

1. Ermitteln der lokalen Variablen, die im zu extrahierenden Bereich geändert und hinter diesem referenziert werden.
2. Wenn mehr als eine solche Variable existiert wird ein Methoden-Objekt angelegt.
3. Sonst wird geprüft ob eine Hookmethode benötigt wird. Das ist der Fall, wenn eine der folgenden Bedingungen gilt:
    a. Das Statement wird von anderen Statements umschlossen.
    b. Referenzen zu temporären Variablen existieren, die in vorherigen Statements deklariert wurden. (Diese Variablen werden dann als Methodenparameter an die Hook-Methode übergeben.).
    c. Genau eine lokale Variable wird geändert, die hinterher referenziert wird (Diese Variable kann dann als Rückgabewert der Hook-Methode verwendet werden.).
    d. Die Statements referenzieren vorher geänderte lokale Variablen. (Diese Variablen werden dann als Methodenparameter an die Hook-Methode übergeben.).
4. Wenn weder ein Methoden-Objekt noch eine Hook-Methode angelegt werden muss, werden die Statements einfach in die Methode des Ziel-Refinements verschoben.

Die Konflikte, die vor der Refaktorisierung erkannt werden, sind:

- Zugriffe innerhalb der extrahierten Statements auf private Objektattribute der Ausgangsklasse
- Zugriffe innerhalb der extrahierten Statements auf private Methoden der Ausgangsklasse
- Definitionen von temporären Variablen innerhalb der zu extrahierenden Statements, die von nachfolgenden Statements referenziert werden
- Return-Statements innerhalb der zu extrahierenden Statements bei Extraktion in eine Hook-Methode oder in ein Methodenobjekt bei dem zusätzlich eine Hook-Methode benötigt wird
- Return-Statements am Ende der zu extrahierenden Statements



Die Refaktorisierung wird aktiviert, indem man die zu extrahierenden Zeilen markiert und die Option "Extract Method Statements" im Menu "Refactor" wählt. Markiert man die Zeilen nicht, kann man sie im folgenden Fenster manuell eingeben.

Technische Details

Vorgehen beim Anlegen eines Methoden-Objekts:

1. Anlegen des Methodenobjekts im Layer der Ausgangsklasse. Name der so entstehenden Klasse: Methodenname+MethodObj.jak.
2. Im Methodenobjekt wird die Ausgangsklasse als Objektattribut mit dem Modifier final und dem Namen SOURCE angelegt.
3. Im Methodenobjekt werden die Methodenparameter der Ausgangsmethode und alle in der Ausgangsmethode deklarierten lokalen Variablen als Objektattribute angelegt (unabhängig davon, ob später auch wirklich alle benötigt werden).
4. Im Methodenobjekt wird ein Konstruktor angelegt mit der Ausgangsklasse (Name = SOURCE) und allen Parametern (Namen bleiben erhalten) der Methode, aus der die Statements extrahiert werden sollen, als Konstruktorparameter. Dabei werden auch die Statements "this.Parameter = Konstruktorparameter;" innerhalb des Konstruktors erzeugt.
5. Die Methode compute() wird ins Methodenobjekt eingefügt. Der Rückgabetyp der Methode ist der selbe, wie der der Ausgangsmethode.
6. Der gesamte Methodeninhalt der Ausgangsmethode wird in compute() verschoben.
7. In der Ausgangsmethode wird eine Instanz des Methodenobjekts angelegt und ein Aufruf von compute() erzeugt. Wenn der Rückgabetyp der Ausgangsmethode (und damit der Rückgabetyp von compute()) nicht void ist, wird zusätzlich das Schlüsselwort return vorangestellt.
8. Bei allen Attributreferenzen in der Methode compute(), die sich auf Objektattribute der Ausgangsklasse beziehen, wird als Scope "SOURCE." davor gesetzt (d.h. z.B. aus ObjektAttributA = 5; wird SOURCE.ObjektAttributA = 5;).
9. Bei allen Methodenaufrufen in der Methode compute(), die sich auf Methoden der Ausgangsklasse beziehen, wird als Scope "SOURCE." davor gesetzt (d.h. z.B. aus MethodeInAusgangsklasseA(); wird SOURCE.MethodeInAusgangsklasseA();).
10. Allen Variablendeklarationen in der Methode compute() die gleichzeitig initialisiert werden, werden in Variablenzuweisungen geändert (d.h. z.B. aus int VariableA = 5; wird VariableA = 5;).
11. Aus der compute()-Methode werden die Statements jetzt in ein Refinement des Methodenobjekts im gewünschten Layer extrahiert (je nach Situation einfach in die Methode des Ziel-Refinements oder in eine Hook-Methode).

Vorgehen beim Anlegen einer Hook-Methode:

1. Wenn das Refinement, in das extrahiert werden soll, noch nicht existiert, wird es angelegt.
2. Die Hook-Methode wird in der Ausgangsklasse sowie in der Zielklasse angelegt. Der Name kann selbst gewählt werden. Wenn das nicht geschieht ist der Name der Hook-Methode hook() (wenn möglich) oder hooki(), wobei i die erste positive Zahl ist, bei der der Name noch nicht im Ziel-Refinement vorkommt. Rückgabewert ist entweder void oder der Typ der Variable, die innerhalb der Statements geändert und hinterher referenziert wird. Die Methode in der Zielklasse erhält den Modifier OVERRIDES.
3. Die Statements aus der Ausgangsmethode werden in die Hookmethode der Zielklasse extrahiert.
4. Falls erforderlich (Rückgabewert ist nicht void) wird ans Ende der beiden Hookmethoden ein return-Statement erstellt.
5. In der Ausgangsmethode wird dort, wo vorher die zu extrahierenden Statements waren ein Aufruf zu der Hookmethode eingefügt (ggf. mit der Zuweisung der Variable die geändert wurde und hinterher referenziert wird : z.B. VariableA = hook();).

Vorgehen beim Extrahieren in die Methode des Ziel-Refinements:

1. Wenn das Refinement, in das extrahiert werden soll, noch nicht existiert, wird es angelegt.
2. Im Ziel-Refinement wird dieselbe Methode wie die Ausgangsmethode angelegt.
3. Die Statements aus der Ausgangsmethode werden in die Methode des Ziel-Refinements extrahiert.
4. Ein Super() Aufruf wird am Anfang oder am Ende der erstellten Methode erzeugt.

Grund, warum nicht der AST von FeatureIDE verwendet wird:

Bei FeatureIDE wird der AST durch die Ahead-Klasse JTSParseTree erzeugt, während das Typsystem die Klasse Parser nutzt. Dadurch werden im AST teilweise andere Knoten erzeugt. Beispiel Klasse HlkMethodObj:

```
layer Layer3;


refines class HlkMethodObj {

   overrides void hook() {
                a = a + 1;
                b = a + 1;
   }

}
```

Durch das Typsystem erzeugter AST:

```
0program
 1AstOptNode
  2AspectStm
   3AST_QualifiedName
    4AST_QualifiedNameElem
     5NameId
 1AstOptNode
 1AstOptNode
  2AST_Class
   3AST_ClassElem
    4ModTypeDecl
     5AstOptNode
     5Ute
      6UmodClassExt
       7NameId
       7AstOptNode
       7ClsBody
        8AstOptNode
         9AST_FieldDecl
          10AST_FieldDeclElem
           11MethodDcl
            12AstOptNode
             13AST_Modifiers
              14AST_ModifiersElem
               15ModOverrides
            12PrimType
             13VoidTyp
             13AstOptNode
            12MthDector
             13NameId
             13AstOptNode
             13AstOptNode
            12AstOptNode
            12MDSBlock
             13BlockC
              14AstOptNode
               15AST_Stmt
                16AST_StmtElem
                 17ExprStmt
                  18AsgExpr
                   19PPQualName
                    20AST_QualifiedName
                     21AST_QualifiedNameElem
                      22NameId
                   19Assign
                   19AddExpr
                    20PPQualName
                     21AST_QualifiedName
                      22AST_QualifiedNameElem
                       23NameId
                    20MoreAddExpr
                     21MoreAddExprElem
                      22AdEBod
                       23Plus
                       23IntLit
                16AST_StmtElem
                 17ExprStmt
                  18AsgExpr
                   19PPQualName
                    20AST_QualifiedName
                     21AST_QualifiedNameElem
                      22NameId
                   19Assign
                   19AddExpr
                    20PPQualName
                     21AST_QualifiedName
                      22AST_QualifiedNameElem
                       23NameId
                    20MoreAddExpr
                     21MoreAddExprElem
                      22AdEBod
                       23Plus
                       23IntLit
```

Durch FeatureIDE erzeugter AST:

```
0program
 1AstOptNode
  2AspectStm
   3AST_QualifiedName
    4AST_QualifiedNameElem
     5NameId
 1AstOptNode
 1AstOptNode
  2AST_Class
   3AST_ClassElem
    4SourceDecl
     5AST_QualifiedName
      6AST_QualifiedNameElem
       7NameId
   3AST_ClassElem
    4ModTypeDecl
     5AstOptNode
     5UmodClassDecl
      6NameId
      6AstOptNode
      6AstOptNode
      6ClsBody
       7AstOptNode
        8AST_FieldDecl
         9AST_FieldDeclElem
          10MethodDcl
           11AstOptNode
            12AST_Modifiers
             13AST_ModifiersElem
              14ModOverrides
           11PrimType
            12VoidTyp
            12AstOptNode
           11MthDector
            12NameId
            12AstOptNode
            12AstOptNode
           11AstOptNode
           11MDSBlock
            12BlockC
             13AstOptNode
              14AST_Stmt
               15AST_StmtElem
                16ExprStmt
                 17AsgExpr
                  18PPQualName
                   19AST_QualifiedName
                    20AST_QualifiedNameElem
                     21NameId
                  18Assign
                  18AddExpr
                   19PPQualName
                    20AST_QualifiedName
                     21AST_QualifiedNameElem
                      22NameId
                   19MoreAddExpr
                    20MoreAddExprElem
                     21AdEBod
                      22Plus
                      22IntLit
               15AST_StmtElem
                16ExprStmt
                 17AsgExpr
                  18PPQualName
                   19AST_QualifiedName
                    20AST_QualifiedNameElem
                     21NameId
                  18Assign
                  18AddExpr
                   19PPQualName
                    20AST_QualifiedName
                     21AST_QualifiedNameElem
                      22NameId
                   19MoreAddExpr
                    20MoreAddExprElem
                     21AdEBod
                      22Plus
                      22IntLit
```

Inwieweit das Typsystem geändert werden kann, damit es auch die neuen Knoten verarbeiten kann, müsste noch geprüft werden.

Hinzukommt, dass der FeatureIDE AST auch zusätzliche Source-Informationen liefert, die in den Quelldateien ursprünglich nicht vorhanden waren. In der Beispielklasse wären das die folgenden Knoten:

1AstOptNode
  2AST_Class
   3AST_ClassElem
    4SourceDecl
     5AST_QualifiedName
      6AST_QualifiedNameElem
       7NameId

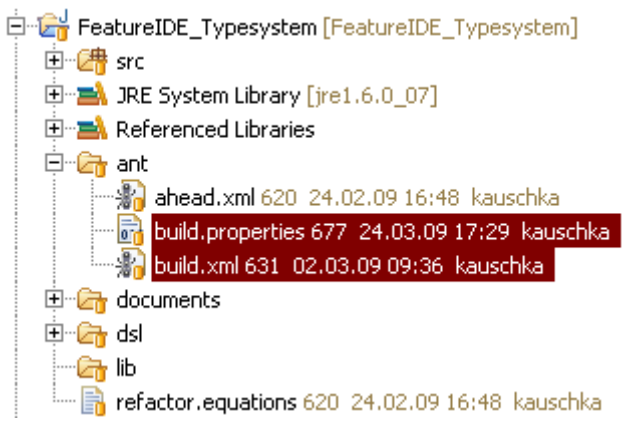
Wenn man jetzt eine Refaktorisierung basierend auf diesem AST durchführen und das Ergebnis speichern würde, wäre diese zusätzliche Source-Information auch in der Datei vertreten:

```
layer Layer3;

SoUrCe Layer3 "Workspace/Spielwiese2/src/Layer31/HlkMethodObj.jak";



refines class HlkMethodObj {

   overrides void hook() {
                a = a + 1;
                b = a + 1;
   }

}
```

Die zusätzlichen Knoten kann man auch nicht einfach aus den AST entfernen, weil dann die Zeileninformationen im Rest des AST nicht mehr stimmen würden.

Wie das featureorientiert geschriebene Typsystem in Java-Code umgewandelt wird:

Das Typsystem, auf dem alle Refaktorisierungen basieren, wurde featureorientiert als als zusätzliches Ahead-Layer geschrieben. Damit das PlugIn dieses jedoch nutzen kann muss es in Java-Code umgewandelt werden. Dazu gibt es ein Ant-Script im Projekt FeatureIDE_Typesystem namens build.xml. Damit es läuft muss in der Datei build.properties der Pfad zu Ahead angepasst werden.



Danach kann das Script mit dem Target makeRefactor gestartet werden, welches den Java-Code für das Typsystem im Ordner src speichert. Dieser Ordner muss dann nur noch als jar-Datei exportiert werden, die dann vom PlugIn genutzt werden kann.