Adaptives Programmieren mit Inject/J

O. Seng, T. Genssler, V.
Kuttruff, B. Schulz
Forschungsbereich
Programmstrukturen
Forschungszentrum Informatik
(FZI)
Haid-und-Neu-Straße 10-14
76131 Karlsruhe
{seng | genssler | kuttruff |
bschulz}@fzi.de

2. Juli 2001

Zusammenfassung

Softwaresysteme müssen ständig weiterentwickelt werden, um den sich ändernden Kunden- und Benutzerbedürfnissen zu genügen. Bestehende Systeme mit neuen Methoden und Schnittstellen zu ergänzen, ist eine fehleranfällige und aufwendige Aufgabe, falls dies " von Hand " durchgeführt wird.

In diesem Papier wird ein Werkzeug vorgestellt, mit dem diese Erweiterungen schnell und einfach durchgeführt werden können. Ausgangspunkt ist die Kombination des Werkzeugs Inject/J mit Konzepten des Adaptiven Programmierens. Inject/J bietet automatisierbare Programmtransformationen auf Javacode, Adaptives Programmieren erlaubt die getrennte Spezifikation von Struktur und Verhalten eines Programms. Neue Funktionen können in einem Skript beschrieben werden, aus welchem dann der entsprechende Code in dem bestehendem System generiert wird.

Keywords:
Aspektorientiertes Programmieren, Adaptives Programmieren, Skriptsprachen, Software-Sanierung

1 Einleitung

Existierende Softwaresysteme "von Hand" mit neuen Funktionen auszustatten ist eine schwierige, fehleranfällige und aufwendige Aufgabe. Dies liegt daran, daß oftmals nicht nur Erweiterungen an der Schnittstelle vorgenommen werden müssen, sondern viele Klassen, die weit über das System verstreut sein können, bearbeitet werden müssen.

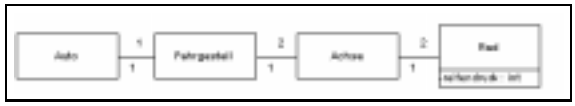

Abbildung 1: Physikalische Autostruktur

Das Klassendiagramm in Abbildung 1 zum Beispiel stellt auszugsweise die Struktur eines Systems dar, das die physikalische Struktur eines Autos modelliert. Will man der Klasse Auto eine Funktion hinzufügen, die den Druck aller Reifen ausgibt, so muß man sich zuerst Zugriff auf die Attribute der Klasse Rad verschaffen. Hierfür müssen alle vier Klassen zusammenarbeiten, da es keine direkte Attributsbeziehung zwischen Auto und Rad gibt.

Der Programmierer muß sich deshalb tief in den Code einarbeiten, er benötigt eine Whitebox-Sicht auf das System. Dieses Verständnis ist oft nicht leicht zu erhalten, da die Programmierer, die den Code ursprünglich geschrieben haben, nicht mit denen identisch sind, die die Erweiterungen ausführen sollen[7].

Solche funktionalen Erweiterungen kommen nicht

nur bei Neuentwicklungen vor, sondern auch in der Software-Sanierung und Wartung. Es ist unmöglich ein Softwaresystem irgendeiner Größe herzustellen, das nicht im Laufe der Zeit abgeändert bzw. angepaßt werden muß. Dies liegt daran, daß die ursprünglichen Anforderungen modifiziert werden, um den sich ändernden Kunden- und Benutzerbedürfnissen zu genügen. In der Wartung spricht man hier von “perfective maintenance“, dieser Bereich macht ca. 65% der gesamten Wartungsaktivitäten aus[7].

Der Extreme Programming Softwareprozess erfaßt diese ständigen Erweiterungen sogar als festen Prozessbestandteil. Anfangs wird nur die minimale Basisfunktionalität implementiert, um diese später auszubauen[2].

Es ist also wünschenswert ein Werkzeug zu haben, das dem Programmierer diese Aufgabe erleichtert. Ein solches Werkzeug muß den folgenden Kriterien genügen:

- **Skalierbarkeit**
  Es muß sowohl mit kleinen als auch mit Programmen umgehen können, deren Größe im Bereich einer Million LOCs liegt.
- **Anwendbarkeit**
  Das Werkzeug muß auf jeden Fall mit bestehendem Code umgehen können.
- **Bedienbarkeit**
  Das Werkzeug darf nicht zu komplex sein, und sollte einfach zu bedienen und zu beherrschen sein.
- **Abstraktion**
  Um Erweiterungen durchzuführen sollte eine lokale Sicht auf das System genügen, und das Werkzeug sollte den Benutzer bei dieser Aufgabe unterstützen.
- **Codequalität**
  Der entstehende Code sollte verständlich und wartbar sein.

Ein Ansatz für ein derartiges Werkzeug sieht folgendermaßen aus: Wie oben schon beobachtet passiert nur an wenigen Stellen wirklich etwas, d.h nur in manchen Klassen muß Code eingefügt werden, der die neue Funktionalität erbringt. Diese Klassen sollen von nun an als Analyseklassen bezeichnet werden, da sie oft schon explizit in der Anforderungsanalyse erwähnt werden. In vielen Klassen müssen nur sogenannte Durchreichmethoden generiert werden, die die neue Aufgabe an die zuständigen Klassen weiterreichen. Hilfreich wäre es, wenn man nur den Code für die Analyseklassen angeben müsste, und wenn die Aufrufe von der Schnittstelle zu den beteiligten Klassen automatisch berechnet werden könnten. Genau diese Vorgehensweise wird in der Theorie des Adaptiven Programmierens beschrieben.

2 Adaptives Programmieren

Adaptives Programmieren (AP) ist ein Spezialfall von Aspektorientiertem Programmieren. Hier werden die beiden “Aspekte“ Struktur und Verhalten getrennt betrachtet. Unter Verhalten versteht man alle Schnittstellen und Methodenrümpfe, die Struktur wird durch das Klassendiagramm bestimmt[6].

Adaptives Programmieren ist eine Weiterentwicklung des Objektorientierten Programmierens. Beim Objektorientierten Programmieren können Struktur und Verhalten nicht getrennt voneinander spezifiziert werden, sie sind ineinander verwoben.

2.1 Struktur

Die Strukturinformation muß in geeigneter Form vorliegen. In einigen Implementierungen von AP muß sie noch explizit, entweder textuell oder graphisch, als Klassendiagramm angegeben werden. In einer neueren Implementierung für Java kann diese Information zur Laufzeit aus bestehendem Code extrahiert werden[8].

2.2 Verhalten

Das Verhalten wird von sogenannten Propagation Patterns bestimmt. Ein Propagation Pattern besteht im wesentlichen aus drei Teilen:

- Der Signatur der neuen Funktion
- Angabe der beteiligten Klassen durch eine Navigationsanweisung. Dies geschieht durch Spezifizierung einer Pfadmenge bzgl. des Klassengraphen. Beteiligt sind alle Klassen die Knoten dieser Pfadmenge sind. Z.B. würden mit der Anweisung **from** `'Klasse1'` **to** `'Klasse2'` alle Klassen ausgewählt, die mit Vererbungs- oder Delegationsbeziehungen von `'Klasse1'` aus erreicht werden können,

und von denen aus man die `'Klasse2'` erreichen kann.

- Codeinjektionen. Die eigentliche Funktionalität wird in den Analyseklassen erbracht. Welchen Code diese ausführen, kann in den Codeinjektionen angegeben werden.

Zusätzlich ist es noch möglich, pfadglobale Variablen zu verwenden.

Die Ausführung der neuen Funktion geschieht dann folgendermaßen: Bei einem Aufruf in der Startklasse wird dieser entlang der Pfadmenge weiterpropagiert. Gelangt man zu einer Klasse, für die eine Codeinjektion geschrieben wurde, so wird zusätzlich dieser Code ausgeführt.

```
// Navigationsanweisung
in class 'Auto' do {
  add navigation
  to 'Rad'

 // Signatur der neuen Funktion
  add method
 'void druckeReifendruck()' {

 //Einzumischender Code
   in 'Rad' do {
    beforeNavigation ${
    System.out.println(reifenDruck);
    }$
  }
 }
}
```

Abbildung 2: Adaptives Programm

In dem Beispiel aus Abbildung 1 würde das Propagation Pattern so aussehen, wie in Abbildung 2 angegeben. Ruft man in der Klasse `'Auto'` die Methode **'void druckeReifendruck()'** auf, so wird durch die Navigationsanweisung **in class** `'Auto'` **do { add navigation to** `'Rad'` festgelegt, daß dieser Aufruf bis zur Klasse `'Rad'` weitergereicht wird. In der Klasse `'Rad'` angekommen wird dann das Codestück `"System.out.println..."` ausgeführt, das in Abbildung 2 für die Klasse `'Rad'` angegeben wurde.

Als Vorteile gegenüber herkömmlichem Objektorientiertem Programmieren ergeben sich:

- **Ausdruckskraft**
  Adaptive Programme sind kürzer, da die Navigation und das Weiterreichen automatisch berechnet werden können.
- **Übersichtlichkeit / Verständlichkeit**
  Der gesamte Code für die neue Funktion an steht logisch zusammenhängend an einer Stelle.
- **Robustheit**
  im Bezug auf Strukturänderungen. Eine Änderung an der Struktur erfordert nicht unbedingt eine Änderung des Verhaltens.

Es existieren bereits mehrere Implementierungen von Systemen, die Adaptives Programmieren unterstützen. Die ersten beiden Implementierungen der Demetergruppe für Java und C++ zeichnen sich dadurch aus, daß sie einen Präprozessor zur Verfügung stellen, der aus einem Klassendiagramm und einer Menge von Propagation Patterns übersetzbaren Javacode erzeugt[5]. Der Nachteil dieser Implementierungen besteht darin, daß man nur sehr schlecht mit bestehendem Code umgehen kann. Man kann mit diesem Ansatz keine neue Funktion in bestehenden Code einfügen, man kann nur neue Klassen entwickeln, die mit dem bestehenden Code zusammenarbeiten können. Diese Implementierungen können also das Kriterium **Anwendbarkeit** nicht erfüllen.

Eine neuere Version ist DJ, eine AP Implementierung für Java, die ebenfalls von der Demetergruppe entwickelt wurde. In DJ wird das AP-Konzept dem Java Programmierer als Bibliothek zur Verfügung gestellt. Hier wird der Code nicht generiert, sondern die Navigation wird zur Laufzeit ermittelt. Dies hat den Vorteil, daß sich die Navigationsanweisungen zur Laufzeit ändern können, und sogar neue Klassen hinzugefügt werden können. Allerdings kann der einmal damit modifizierte Code jetzt nicht mehr ohne diese Bibliothek weiterbearbeitet werden. Die Benutzung von Java-Reflection zur Berechnung der Navigation zur Laufzeit kostet auch viel Zeit, so daß bei großen Systemen die Performance leidet, und DJ somit nur mit kleineren Systemen benutzbar ist und die Anforderung **Skalierbarkeit** verletzt.

Ziel dieser Arbeit ist es, ein Werkzeug, basierend auf Adaptivem Programmieren, zu entwickeln, welches die zuvor beschriebenen Anforderungen erfüllt, also insbesondere **mit bestehendem Code und großen Systemen** umgehen kann.

3 Unser Ansatz

In dieser Arbeit soll ein Werkzeug entstehen, mit dem Erweiterungen bestehender Systeme weniger fehleranfällig und aufwendig sind, als wenn sie von Hand durchgeführt werden. Adaptive Programme werden als Metaprogramme geschrieben, die die neue Funktionalität in einem Skript beschreiben.

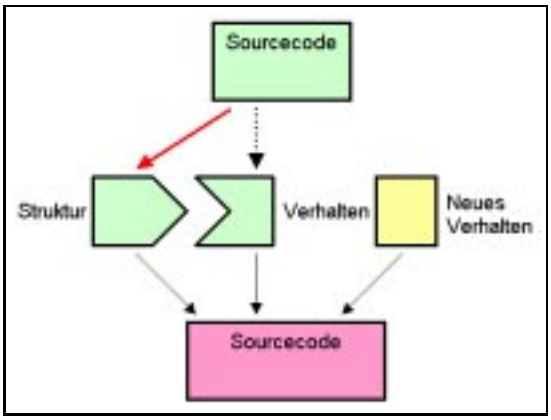

Abbildung 3: Unser Ansatz

Ausgangspunkt ist der Javacode eines bestehenden Systems. Dieser Code liefert zum einen die Information über die Klassenstruktur des Systems, zum anderen hat er schon "implizites" Verhalten, da die Klassen in der Regel bereits Schnittstellen und Methodenrümpfe besitzen. Mit einem Adaptiven Skript, bestehend aus einer Menge von Propagation Patterns, wird das hinzuzufügende Verhalten beschrieben. Anhand dieser Muster wird nun der vorhandene Code mit neuem Code ergänzt. Um die Navigation berechnen zu können wird aus dem Code implizit ein Strukturmodell erzeugt, es muß nicht wie bei den älteren Demeterimplementierungen explizit vorhanden sein. Das Ergebnis besteht wieder aus korrektem, erweiterbarem Javacode, der ohne das Werkzeug übersetzbar und ausführbar ist.

Das Adaptive Konzept ergänzt das Konzept von Inject/J hervorragend, da das Hinzufügen neuer Funktionen auch als Anpassung von Software zu sehen ist.

Die Integration des Konzepts in Inject/J hat den Vorteil, daß viele Funktionen, die zur Generierung von Code aus Adaptiven Skripten nötig sind, schon vorhanden sind. Inject/J bietet automatisierbare Programmtransformation zur Anpassung von Software an sich verändernde Anforderungen. Das heißt, daß Inject/J unter anderem die Möglichkeit bietet, Sourcecode einzulesen und diesen mit neuen Methodensignaturen und Methoderümpfen zu ergänzen. Dies sind zwei wichtige Aufgaben, die sich bei der Generierung von Javacode aus Propagation Patterns ergeben.

3.1 Sprache

Bei der Bestimmung der Sprache wurde eine sinnvolle Auswahl aus den bestehenden AP-Konstrukten getroffen, die möglichst einfach und aussagekräftig ist, aber trotzdem die volle Mächtigkeit von AP unterstützt.

Ein AP Skript in Inject/J besteht wie in normales AP Skript aus drei Teilen:

- **Navigationsanweisung**

```
in class 'A' do {
   add navigation
   to 'B'
   [ via 'C']
   ( through|bypassing 'D')*
}
```

  Inject/J AP-Skripte beginnen mit der Navigationsanweisung, weil sich die AP-Funktionalität so besser in die Inject/J Sprache einfügt. Der Kern der Anweisung ist der **in class ... add navigation ... to ...** Teil, hier wird der Navigationspfad mit seinen Start und Endklassen festgelegt. `'A'` und `'B'` stehen für eine nichtleere Menge von Klassennamen. Mit **via** können für den Pfad noch Wegpunkte, in diesem Falle Klassen angegeben werden, die auf jeden Fall auf dem Pfad liegen müssen. Die Anweisungen **through | bypassing** dienen dazu den Pfad bezüglich der Kantenmenge genauer zu spezifizieren. **through** `'D'` hat zur Folge, daß die Kante `'D'` des Klassengraphen auf jeden Fall Bestandteil der Navigationspfade sein muß, **bypassing** D sorgt dafür daß die Kante `'D'` auf keinem Pfad von `'A'` nach `'B'` vorkommt. Gegenüber den Navigationsanweisungen wie sie in [1] beschrieben sind, lassen die Anweisungen nicht so komplexe Pfadkonstrukte zu, allerdings fördert dies Verständlichkeit und Nachvollziehbarkeit der Navigation. Komplexere Pfade können aus mehreren Pfaden zusammengesetzt werden, d.h. die Mächtigkeit von AP wird dadurch nicht eingeschränkt.

- **Methodensignatur**

  add method '<SIGNATUR>'

  Als nächstes muß die Signatur der neuen Methode angegeben werden. Hierbei gibt es keinerlei Einschränkungen, Methodensignaturen können gemäß der Java Sprachspezifikation angegeben werden.

- **Codeinjektionen**

  in 'C' **do** {
  beforeNavigation | **afterNavigation** $ {
  <JAVACODE> } $
  }

  Diese Codefragmente bestimmen die eigentliche Funktionalität der neuen Methode. Der Javacode wird in den durch 'C' bestimmten Klassen eingefügt. 'C' kann eine Menge von Klassen oder Kanten sein. Ist 'C' eine Klassenmenge so wird bei **beforeNavigation** der Code in der neuen Methode der Klasse eingefügt, bevor das Weiterreichen stattfindet. Bei **afterNavigation** wird zuerst die Navigation ausgeführt und anschließend der Javacode. Steht 'C' für eine Kante zwischen zwei Klassen 'A' und 'B', so wird bei **beforeNavigation** der Code in der Klasse 'A' eingefügt, und dann der Aufruf an die Klasse 'B' weitergereicht. Mit **afterNavigation** wird in 'A' nur der Aufruf an 'B' erzeugt, in 'B' wird dann der Code eingemischt. Diese Art der Spezifizierung ist sinnvoll, falls man für 'A' oder 'B' ein Wildcardsymbol angibt. Mit 'C' = '*, :>, A' würde man z.B. alle Kanten selektieren, die eine Vererbungsbeziehung von einer beliebigen Unterklasse zur Oberklasse 'A' darstellen, also alle Klassen auswählen, die von der Klasse A erben.

Auch in dieser Sprache soll die Möglichkeit existieren, pfadglobale Variablen zu verwenden. In [1] werden hierfür spezielle Transportation Patterns eingeführt. Im Rahmen dieser Arbeit wird zur Zeit noch untersucht, ob man ohne diese zusätzlichen Muster auskommen und die pfadglobalen Variablen einfacher verfügbar machen kann.

3.2 Beispiel

Um unseren Ansatz zu verdeutlichen, soll noch einmal das Beispiel aus Abbildung 2 betrachtet werden.

```
class Auto {
...
   Fahrgestell chassis;
...
}

class Fahrgestell {
...
   Achse vorderAchse;
   Achse hinterAchse;
...
}

class Achse {
...
   Rad links;
   Rad rechts;
...
}

class Rad {
...
   int reifenruck;
...
}
```

Abbildung 4: Vorhandener Code

Der zu diesem Klassendiagramm gehörende Code würde in etwa so wie in Abbildung 4 dargestellt aussehen.

Die Aufgabe war nun, die Klasse 'Auto' mit einer neuen Funktion **'void druckeReifendruck()'** auszustatten, die den Druck aller Autoreifen auf der Standardausgabe ausgibt. Das zugehörige AP-Skript ist in Abbildung 2 zu sehen. Der Schreiber des Skripts muß nur wissen, daß ein Auto über Räder verfügt, und daß die Klasse 'Rad' ein Attribut "reifendruck" besitzt. Wie die Klasse 'Rad' genau mit der Klasse 'Auto' in Beziehung steht, d.h. wieviele Räder das Auto hat und welche anderen Elemente wie Fahrgestell und Achse noch dazwischen liegen, muß der Programmierer nicht wissen.

Das Werkzeug liest zuerst den Code aus Abbildung 4 ein, um daraus das implizite Strukturmodell zu generieren. Als nächstes wird dann die Navigationsanweisung **in class** 'Auto' **do { add navigation to** 'Rad' ... ausgewertet. Es wird festgestellt, daß jeder mögliche Weg von 'Auto' nach 'Rad' über die Klassen 'Fahrgestell' und 'Achse' führt, und somit diese Klassen zusätzlich benötigt

werden. Dann wird in der Klasse `'Auto'` eine neue Methode **'void druckeReifendruck()'** generiert, der Rumpf dieser Methode besteht aus einem Aufruf von **'void druckeReifendruck()'** an die Klasse `'Fahrgestell'`. Dieses Weiterreichen wird bis zur Klasse `'Rad'` durchgeführt, hier wird nun als Methodenrumpf der Code eingefügt der im AP-Skript durch die Anweisung **in** `'Rad'` **do** `{` **beforeNavigation** `... }` angegeben ist. **beforeNavigation** hat hier keine Auswirkungen, da die Navigation in der Klasse Rad endet.

Das Ergebnis dieses Vorgangs ist in Abbildung 5 zu sehen.

```
class Auto {
...
 Fahrgestell chassis;

 void druckeReifendruck() {
    chassis.druckeReifendruck();
  }
...
}

class Fahrgestell {
...
  Achse vorderAchse;
  Achse hinterAchse;

 void druckeReifendruck() {
   vorderAchse.druckeReifendruck();
   hinterAchse.druckeReifendruck();
 }
...
}

class Achse {
...
 Rad links;
 Rad rechts;

 void druckeReifendruck() {
   links.druckeReifendruck();
   rechts.druckeReifendruck();
 }
...
}

class Rad {
...
 int reifenDruck;

 void druckeReifendruck() {
   System.out.println(reifenDruck);
 }
...
}
```

Abbildung 5: Code nach dem Einmischen

3.3 Erfüllung der Kriterien

In diesem Abschnitt zeigen wir, daß das von uns entwickelte Werkzeug die geforderten Kriterien erfüllt.

- **Anwendbarkeit**
  AP-Skripte lassen sich direkt mit bestehendem Code kombinieren. Aus dem Skript wird der entsprechende Code generiert, der dem existierenden Code hinzugefügt wird.

- **Skalierbarkeit**
  Unser Werkzeug kann problemlos mit Programmen im Größenbereich von einer Million LOCs umgehen. Der verwendete Parser ist ausreichend leistungsstark, und da die Methoden statisch eingemischt werden, können zur Laufzeit auch keine Geschwindigkeitseinbußen entstehen, wie sie beim dynamischen Ansatz auftreten können.

- **Bedienbarkeit**
  Um die Bedienbarkeit des Werkzeuges zu erleichtern, wurde die Sprache möglichst einfach und verständlich gestaltet ohne Abstriche bzgl. der Mächtigkeit von AP machen zu müssen. Der initiale Lernaufwand kann dem Benutzer allerdings nicht erspart werden.

- **Abstraktion**
  Die lokale Sicht auf das System reicht aus, weil man die neue Funktion als separates AP-Skript schreiben kann, und weil in den Navigationsanweisungen nur die Analyseklassen angegeben werden müssen. Das Werkzeug berechnet automatisch die Navigation und fügt entsprechend der angegebenen Codeinjektionen Code hinzu.

- **Codequalität**
  Das Resultat nach dem Einmischen eines AP-Skripts ist korrekter, typischer und erweiterbarer Javacode. Zusätzlich wird in den Durchreichmethoden noch Kommentarcode generiert, der den Ursprung dieser Methode und die Navigationsanweisung mit den beteiligten Klassen enthält.

Bestehende Implementierungen von AP Programmieren erfüllen nicht gleichzeitig die Kriterien Skalierbarkeit und Anwendbarkeit, so daß der Ansatz der in dieser Arbeit verfolgt wurde eine erhebliche Verbesserung darstellt.

4 Zusammenfassung und Ausblick

In dieser Arbeit wird ein Werkzeug beschrieben, mit dem Erweiterungen an vorhandenem Code handhabbarer werden. Erweiterungen kommen sowohl bei der Software-Sanierung als auch bei der Wartung und Weiterentwicklung von Software vor. Sie kosten, falls von Hand durchgeführt, viel Zeit. Mit dem Werkzeug kann der Benutzer schnell, und ohne tieferes Verständnis des Sourcecodes haben zu müssen, diese Erweiterungen vornehmen.

Er muß sich nur die Elemente der Skriptsprache aneignen, welche aber bewußt einfach und aussagekräftig gehalten sind. Zudem muß er die Grundkonzepte von Adaptivem Programmieren verstanden haben. Dieser initiale Mehraufwand lohnt sich aber, da er nun Modifikationen schnell und einfach durchführen kann.

Im weiteren Verlauf wird das Werkzeug auf seine Praxistauglichkeit hin getestet, indem mit ihm Erweiterungen an bestehenden Systemen, z.B. JHotDraw durchgeführt werden. Zudem wird untersucht, ob und wie sich die erwähnte Robustheit gegenüber Strukturänderungen in das Werkzeug integrieren läßt.

Literatur

[1] Karl Lieberherr. *Adaptive Object-Oriented Software - The Demeter Method - With Propagation Patterns*. PWS Publishing Company, 1995.

[2] Kent Beck. *Extreme Programming Explained: Embrace Change* Addison Wesley, 1999.

[3] Volker Kuttruf. Inject/J - Ein Werkzeug zur skriptgesteuerten Quelltexttransformation Studienarbeit 1999, FZI Karlsruhe

[4] Inject/J. http://injectj.fzi.de, 2001.

[5] Demeter. http://www.ccs.neu.edu/research/demeter/

[6] Uwe Assmann. A Conceptual Comparison of Current Component Systems. Springer, 1999.

[7] Ian Sommerville. *Software Engineering* Addison Wesley, 1996.

[8] Doug Orleans and Karl Lieberherr. *DJ: Dynamic Adaptive Programming in Java*. http://www.ccs.neu.edu/research/demeter/biblio/DJ-reflection.html, 2001.