Code–Transformationen

(Refactorings)

Code–Transformationen

Lernziele

- Anwendungen
- Grundsätzliche Bestandteile und Vorgehen
- Implementierung

Kontext

- Code–/Entwurfsebene
- Automatische Code–Transformationen beinhalten Mustersuche
- Aspekte des Reengineering, nicht nur des Reverse Engineerings

(Semi−)Automatische Transformationen

- (semi−)automatisch und beliebig wiederholbar
- dokumentarisch
- Transformation ist Spezifikation der Änderung
- kontrollierbar
- Transformationen repräsentieren Implementierungswissen
- Vor− und Nachbedingungen der Transformationen sind prüfbar
- Erhalt der Semantik bei einer Transformation lässt sich zeigen

Transformation

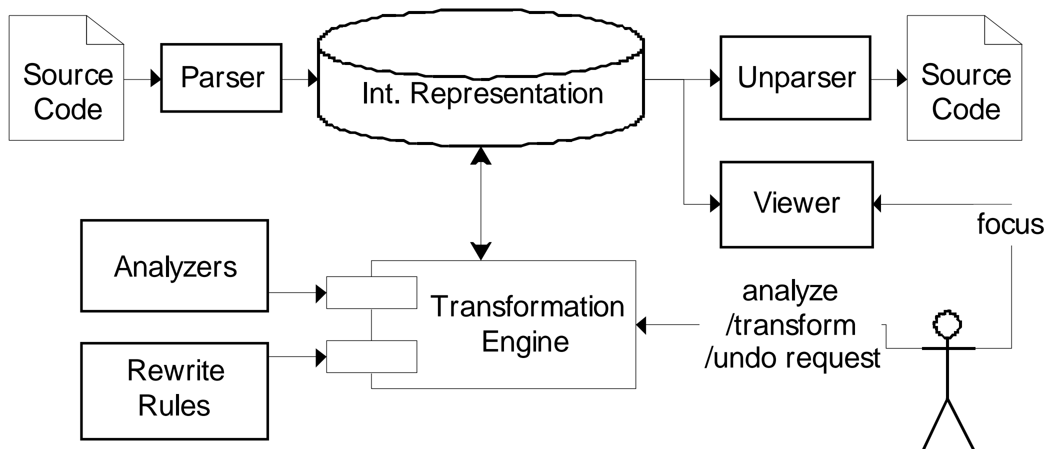
- Eine **Transformation** ist eine partielle Funktion t:
  - t: Spezifikation/Programm → Spezifikation/Programm
  - Beispiele: Compiler, YACC, Programmrestrukturierer
- Transformationen werden oft als Rewrite−Regeln mit Pattern−Variablen repräsentiert:

```
rule eliminate_additive_identity
    replace [expression]
        T [expression] + 0
    by
        T
end rule
```

Syntax von TXL; siehe www.txl.ca

Transformationssystem

- Ein **Transformationssystem** ist ein System, das semantisch wohl–definierte (teil–)mechanisierte Programmmodifikationen ermöglicht.



Schritte einer Transformation

1. Lokation: Identifikation der Stelle, an der Transformation angewandt werden soll
   1. Mustersuche
   2. Lokation durch Benutzer
2. Prüfung der Vorbedingungen der Transformation
3. Anwendung, falls Vorbedingungen erfüllt sind

Vorbedingungen von Transformationen

Die Anwendbarkeit von Transformationen ist oft an Bedingungen geknüpft:

```
rule eliminate_additive_identity
    replace [expression]
        T1 [expression] + T2 [expression]
    where
        value (T2) = 0
    by
        T1
end rule
```


Umsetzung von Transformationen

Ersetzung wird als Prozedur auf dem abstrakten Syntaxbaum implementiert:

```
procedure eliminate_additive_identity (n : in out node) is
begin
 if type (n) = plus then -- Mustersuche
   if value (n.right) = 0 then -- Bedingung
       -- Ersetzung
       replace_child (parent (n), from => n, to => n.left);
       n.left.parent := n.parent;
       delete_tree (n.right); delete (n);
   end if;
 end if;
end;
```

Eigenschaften von Transformationen

- Bestimmte Eigenschaften müssen bei der Transformation erhalten bleiben, andere Eigenschaften sollen sich ändern:
  - z.B. Performanz, Strukturiertheit, Änderbarkeit, ...
- In vielen (aber nicht allen) Fällen soll die Semantik erhalten bleiben.

Semantikerhaltende Transformationen

Formale Betrachtung:

- Objekt o ist definiert in einem Kalkül
  - grundsätzliche Wahrheiten (Axiome) A
  - Menge von Inferenzregeln: $\therefore$ = {i: f $\rightarrow$ f'}
- Objekt o hat die Eigenschaften P(o) = {f: o,A $\therefore^*$ f}
- Transformation *t* ist **semantikerhaltend** genau dann, wenn für alle Objekte o gilt:
  - die Eigenschaften von o bleiben durch die Transformation erhalten: P (o) $\subseteq$ P(t(o))
  - P (t (o)) ist konsistent (enthält keine Widersprüche)

Semantikerhaltende Transformationen

- Die Erhaltung der Semantik ist in der Praxis nur sehr schwer nachweisbar:
  - Beweis muss nicht nur die Semantik der Programmiersprache, sondern auch die aller aufgerufenen Betriebssystemfunktionen u.ä. einbeziehen, und ist im Allgemeinen schwierig.
- Nach Transformationen stets Regressionstests durchführen!

Beispiel eines Transformationssystems

TXL (Queens University, Canada): generisches Transformationssystem; ausgeprägt für C, C++, Java, Javascript, Modula, Object Pascal, XML

- abstrakter Syntaxbaum wird generiert
- funktionale Programmiersprache mit Transformationsregeln mit “native Patterns”:
- automatische Traversierung des abstrakten Syntaxbaums und Anwendung der Transformationen, solange dies möglich ist

TXL

```
rule eliminate_redundant_declarations
 replace [repeat statement]
    var X [id] : T [type_spec]
    Rest_of_Scope [repeat statement]
 where not
    Rest_of_Scope [references X]
 by
    Rest_of_Scope
end rule
function references X [id]
    match * [id] X
end function
```


Refactorings

Refactorings sind semantikerhaltende, restrukturierende Code–Transformationen für objekt–orientierte Programme (zur Verbesserung der Wartbarkeit)

Beschreibung (von Fowler):

- Name
- Anwendbarkeit
- Motivation
- mechanische Schritte (die eigentliche Transformation), die von Hand ausgeführt werden
- Beispiel

Sehr viele dieser Refactorings sind genauso auf prozedurale Programme anwendbar.

Refactorings

Angestoßen von Änderungswunsch.

Prozess (inkrementell, iterativ):

1. Identifikation eines „schlechten Geruchs“ (bad smell)
2. Refactoring
3. Compile & Test
4. Eigentliche Änderung
5. Compile & Test

Stink Parade of Bad Smells (Fowler)

- duplizierter Code
- lange Methoden
- große Klassen
- lange Parameterlisten
- divergente Änderung
  - eine Klasse wird stets geändert in verschiedener Weise und für unterschiedliche Gründe
- Schrotflinten–Chirurgie (Shotgun Surgery)
  - kleine Änderungen überall
- Feature–Neid
  - sehr viele Attribute einer anderen Klasse werden für eine Berechnung benutzt

Stink Parade of Bad Smells

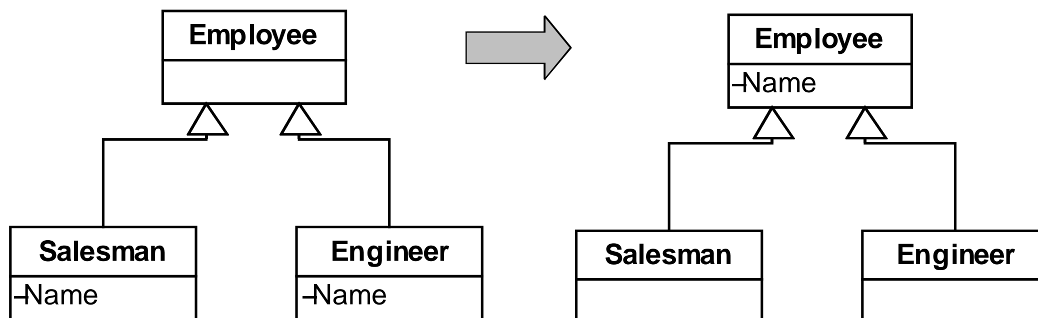
- Datenklumpen (Data Clumps)
  - eine Menge von Datenelementen, die häufig gemeinsam benutzt werden
  - z.B.: Attribute einer Klasse, Parameter in Methodensignaturen
- Fixierung aufs Primitive (Primitive Obsession)
  - einfache Typen werden nicht als Klasse sondern als primitive Datentypen deklariert
- Switch–Anweisungen
  - händisches dynamisches Binden
- Parallele Vererbungshierarchien
- Faule Klassen
  - Klassen, die nichts Nützliches (mehr) tun
- ...

70 Refactorings von Fowler

- Methodenzusammensetzung
  - z.B. Extraktion von Methoden
- Eigenschaften zwischen Klassen bewegen
  - z.B. Verschiebung von Attributen oder Methoden
- Organisation von Daten
  - z.B. Verbergung von Attributen
- Vereinfachung bedingter Ausdrücke
  - z.B. Zerlegung komplexer Bedingungen
- Vereinfachung von Methodenaufrufen
  - z.B. Separierung von bloßem Zugriff von Manipulation
- Generalisierungen
  - z.B. Attribute oder Methoden in der Hierarchie auf– oder abwärts bewegen

Beispiel: Pull–Up Field

Gleiches Attribut in Unterklassen wird nach Oberklasse verlegt.



Motivation für Pull–Up Field

“If subclasses are developed independently, or combined through refactoring, you often find that they duplicate features. In particular, certain fields can be duplicates. Such fields sometimes have similar names but not always. The only way to determine what is going on is to look at the fields and see how they are used by other methods. If they are being used in a similar way, you can generalize them.

Doing this reduces duplication in two ways. It removes the duplicate data declaration and allows you to move from the subclassses to the superclass behavior that uses the field.”

Mechanics für Pull−Up Fields

1. Inspect all uses of the candidate fields to ensure they are used in the same way.
2. If the fields do not have the same name, rename the fields so that they have the name you want to use for the superclass field.
3. Compile and test.
4. Create a new field in the superclass.
5. If the fields are private, you will need to protect the superclass field so that the subclasses can refer to it.
6. Delete the subclass fields.
7. Compile and test.
8. Consider using *Self Encapsulate Field* on the new field.

Bewertung Refactorings

- Die “Mechanics” werden von Hand durchgeführt
  - es gibt Werkzeuge, die manche Refactorings automatisieren (Refactoring Browser für Smalltalk, Eclipse für Java).
- Beschreibung ist zu informell für eine automatisierte Transformation, aber zumindest eine gute Checkliste.
- Die genauen Vorbedingungen sind nicht ausreichend angegeben.
- Hinter “Compile & Test” kann sich viel Arbeit verbergen.