Reversible objektorientierte Programme

Andreas Turk
Bayerisches Forschungszentrum für Wissensbasierte Systeme
Am Weichselgarten 7, 91 058 Erlangen

Zusammenfassung

Reversible Programme sind mittlerweile *state of the art*. CLOS kann reflexiv um Reversibilität erweitert werden. Wenn diese Erweiterung einmal vorgenommen worden ist, dann sind Anwendungsprogramme ohne weiteres reversibel.

Schlüsselwörter: Software-Technik, *Reuse*, Reversible Programme, Rücksetzbarkeit, *Undo*, *History*, Invertierung von Seiteneffekten, Objektorientierte Programmierung, Metaobjektprotokoll, ANSI Common Lisp, CLOS

1 Reversible Programme

Noch vor wenigen Jahren stellte das Zwischenschalten von *Sicherheitsabfragen* vor die Ausführung von Aktionen gewisser Tragweite die einzige Möglichkeit dar, Benutzer interaktiver Software vor den Konsequenzen ihres Fehlverhaltens zu bewahren. Mittlerweile findet sich der Anwender nur noch selten mit Fragen wie „Wollen Sie das wirklich löschen?" konfrontiert. Stattdessen beinhalten selbst primitive Programme zumeist eine Funktion, die das *einfache Zurücksetzen* (*Undo*) der letzten Aktion erlaubt. In komplexeren Anwendungen und insb. in der auf *Windows* ausgerichteten Standardsoftware trifft man immer öfter auf (mehr oder weniger) vollständige *History*-Listen, die eine *vollständige Rücksetzbarkeit* der bis dahin durchgeführten Aktionen gewährleisten.[1] Eine *selektive Rücksetzbarkeit* ist darüber hinaus gegeben, wenn die Auswahl der zurückzusetzenden Aktion nicht mehr an die Reihenfolge ihrer Ausführung gebunden ist. Dies erfordert jedoch eine nichttriviale Verwaltung der wechselseitigen Abhängigkeiten von Aktionen.

Rücksetzbar sind prinzipiell ausschließlich Seiteneffekte, ausgenommen Ein- und Ausgabeaktionen. Es muß also in der objektorientierten Programmierung lediglich das Erzeugen und Vernichten von Objekten, sowie die Manipulation von Attributwerten überwacht werden, um Rücksetzbarkeit handhaben zu können.

Die übliche Realisierung derartiger *Undo*-Mechanismen erfordert die Bereitstellung inverser Funktionen, die bei der Ausführung jeder rücksetzbaren Aktion gemeinsam mit den aktuell relevanten Parametern in einen Kellerspeicher (*Stack*) abgelegt werden. Programmierung und nachträgliche Modifikation der rücksetzbaren Programme erfordern i. d. R. die explizite Berücksichtigung dieser Mechanismen, da die Datenmanipulationen für jede Objektklasse und jedes Attribut gesondert zu handhaben sind.

Im folgenden soll eine allgemeine Realisierung der vollständigen Rücksetzbarkeit von Objektmanipulationen[2] vorgestellt werden, die dem Programmierer keine besondere Disziplin abverlangt und die nachträgliche Änderungen der Klassendefinitionen (z. B. durch das Hinzufügen von Attributen) toleriert. Die vorgestellte Lösung hat bereits in ähnlicher Form bei der Entwicklung des *Requirements Analysis Assistant*, eines Assistenzsystems für die interviewbasierte Anforderungsanalyse in der Systementwicklung, am FORWISS Verwendung gefunden (Verbundprojekt EPK-fix[3]). Wir verwenden als Programmiersprache CLOS[4], eine Erweiterung von *Common Lisp*[5], und implementieren die Rücksetzbarkeit als ergänzende Eigenschaft des CLOS-Metaobjektprotokolls. Einmal mehr erweist sich *Common Lisp* so als prädestiniert, komplexe Probleme mit geringem Programmieraufwand zu lösen.

2 Ergänzung des Metaobjektprotokolls

Ein Metaobjektprotokoll (MOP) definiert Prozesse, die der Verwaltung von Metaobjekten (Klassen, Methoden u. a.) dienen. In CLOS existieren definierte Schnittstellen zum MOP, mit deren Hilfe der Programmierer in die Verwaltung der Metaobjekte eingreifen kann.[6] Das MOP steht dem Programmierer homogen in Form eines reflexiven CLOS-Programms

[1] Manchem Emacs-Benutzer ist das ein alter Hut.

[2] Der *Garbage Collector* in *Common Lisp* macht die Beachtung der Lebensdauer von Objekten obsolet.

[3] [SKT*97]

[4] *Common Lisp Object System* [Pae93a]

[5] [Gra96]

[6] [KdRB91, Pae93b]

zur Verfügung. In der Regel basiert die Ergänzung des MOP auf der Wiederverwendung bereits vorhandener Komponenten durch Subklassenbildung und Methodendefinition. So läßt sich die Programmiersprache CLOS „minimalinvasiv" an die aktuellen Belange der Programmierung anpassen, hier durch die Bereitstellung vollständiger Rücksetzbarkeit als neue sprachimmanente Eigenschaft. Programme sind danach ohne weiteres reversibel.

Wir definieren zunächst eine neue Metaklasse `monitored-class`, die die Klassen der rücksetzbaren Objekte umfaßt. Ein Exemplar dieser Metaklasse ist z. B. die Klasse `article`.

```
(defclass monitored-class (standard-class)
    ())

(defclass article ()
    ((contents :initform "" :accessor contents))
   (:metaclass monitored-class))
```

Nun modifizieren wir die Methoden zur Manipulation von Attributwerten (*slot values*) derjenigen Objekte, die Exemplare einer Klasse (d. h. eines Metaobjekts, z. B. der Klasse `article`) sind, welche selbst wiederum ein Exemplar der neu definierten Metaklasse `monitored-class` ist. Die Modifikation sorgt für die Anwendung der Registrierfunktionen `register-slot-manipulation` und `register-slot-makunbound`.

```
(defmethod (setf slot-value-using-class) :before
    (value (class monitored-class) object slot)
   (unless (and (slot-boundp-using-class
                  class object slot)
                (eq (slot-value-using-class
                      class object slot)
                    value))
     ;; this may not manipulate the same slot!
     (register-slot-manipulation
       object slot value)))

(defmethod slot-makunbound-using-class :before
    ((class monitored-metaclass) object slot)
   (when (slot-boundp-using-class
           class object slot)
     ;; this may not manipulate the same slot!
     (register-slot-makunbound object slot)))
```

Die korrekte Implementierung dieser Registriermethoden vorausgesetzt sind Manipulationen dann unmittelbar rücksetzbar:

```
USER<1> (setf *my-article*
              (make-instance 'article))
#<ARTICLE #xFC48B4>
USER<2> (setf (contents *my-article*)
              "On Undos")
"On Undos"
USER<3> (setf (contents *my-article*)
              "On Undos, by Andreas Turk")
"On Undos, by Andreas Turk"
USER<4> (back)
NIL
USER<5> (contents *my-article*)
"On Undos"
USER<6>
```

Mit einer Klassen- und zwei Methodendefinitionen ist also eine Erweiterung des MOP abgeschlossen, die die Basis für eine automatische Erzeugung von *Undo*-Funktionen für jede mögliche Art und Weise der Änderung eines Attributwertes bildet. Im weiteren muß lediglich die Verwaltung der Information zu Rücksetzbarkeit und die Anwendung dieser Information realisiert werden.

3 Undo-Verwaltung

Um reversible Aktionen zu protokollieren, stellen wir durch die Definition der Klasse `action` benannte Objekte zur Verfügung. Diese enthalten eine Funktion, deren Anwendung die jeweilige Aktion unmittelbar rückgängig macht, und einen Namen, der der lesbaren Darstellung der Objekte durch die Methode `print-object` dient.

```
(defclass action ()
    ((name :initarg :name :reader name)
     (undo-fn
       :initarg :undo-function
       :accessor undo-function)))

(defmethod print-object ((o action) s)
   (format s "#<~a>" (name o)))
```

Die Exemplare der Klasse `action` werden sukzessive in einem Kellerspeicher deponiert, der *History*, die durch die globale Variable `*history*` repräsentiert wird.[7] Das Hinzufügen einer Aktion zur *History* erfolgt durch die Methode `add-action-to-history` (in deren letzer Zeile).

```
(defvar *history* nil)
(defvar *group-actions-p* nil)

(defmethod add-action-to-history
    ((action action))
   (if (and *group-actions-p*
            (listp *history*))
      (extend-action (first *history*) action)
      (push action *history*)))
```

Es bleibt, die Verbindung zwischen MOP-Ergänzung und *History*-Keller herzustellen. Dazu definieren wir die ausstehenden Registriermethoden, die geeignete Exemplare der Klasse `action` erzeugen und in der *History* plazieren.[8]

---

[7] Eine weitere globale Variable `*group-actions-p*` dient der Gruppierung von Einzelaktionen in Abschnitt 4.

[8] Gegebenenfalls wird eine Bindungsaktion eingetragen, um die programmiersprachliche Besonderheit der fehlenden Bindung korrekt zu handhaben.

```
(defmethod register-slot-manipulation
    (object slot new-value)
   (let* ((slot-name
            (slot-definition-name slot))
          (action
            (if (slot-boundp object slot-name)
               (make-slot-manipulation-action
                 object
                 slot-name
                 (slot-value object slot-name))
               (make-slot-binding-action
                 object slot-name))))
      (add-action-to-history action)))

(defmethod register-slot-makunbound
    (object slot)
   (let* ((slot-name
            (slot-definition-name slot))
          (action
            (make-slot-manipulation-action
              object
              slot-name
              (slot-value object slot-name))))
      (add-action-to-history action)))
```

In den Methoden `make-slot-...` definieren wir Bezeichnung und 0-stellige *Undo*-Funktion einer Aktion. Um präzise zu sein: die *Undo*-Funktion ist eine *Closure*, also eine Funktion, die ihre lexikalische Umgebung bewahrt und so im vorliegenden Fall Zugriff auf den früheren Attributwert eines Objektes hat. Der Aufruf dieser *Undo*-Funktion bewirkt das Wiedereinsetzen des früheren Wertes.[9]

```
(defmethod make-slot-manipulation-action
    (object slot-name old-value)
   (make-instance 'action
     :name
     (format nil
       "Object: ~a; Slot: ~a" object slot-name)
     :undo-function
     #'(lambda ()
         (setf (slot-value object slot-name)
               old-value))))

(defmethod make-slot-binding-action
    (object slot-name)
   (make-instance 'action
     :name
     (format nil
       "Object: ~a; Slot: ~a" object slot-name)
     :undo-function
     #'(lambda ()
         (slot-makunbound object slot-name))))
```

Die Funktion `back` führt eine über den optionalen Parameter festzulegende Anzahl von *Undo*-Schritten aus (Aufruf der *Undo*-Funktion in der letzen Zeile). Da auch ein *Undo* Seiteneffekte bewirkt und so den *Undo*-Mechanismus selbst in Kraft setzt, machen wir bereits von der Möglichkeit einer Gruppierung gebrauch, die noch eingeführt werden muß (Abschnitt 4).

```
(defun back (&optional (steps 1))
   (let ((selected-actions
           (subseq *history* 0 steps)))
      (with-action-group
        (format nil "UNDO ~a steps" steps)
        (dolist (action selected-actions)
           (funcall (undo-function action))))))
```

Die *History* präsentiert sich am Ende des obigen Beispiels (Abschnitt 2) dann so:

```
USER<6> *history*
(#<UNDO 1 steps>
 #<Object: #<ARTICLE #xFC48B4>; Slot: CONTENTS>
 #<Object: #<ARTICLE #xFC48B4>; Slot: CONTENTS>)
USER<7>
```

Die Ausgabenotation der Aktionen in der *History* ergibt sich aus ihrem Attribut `name`.

4 Gruppierung von Einzelaktionen

Bei zusammenhängenden komplexen Aktionen, die eine Vielzahl einzelner Seiteneffekte bewirken, spiegelt die *History* das Benutzerverhalten nicht wie erwartet wider, da sie Einzelaktionen per separatem Eintrag enthält. Dies gilt in besonderem Maße für *Undo*-Aktionen selbst. Wir führen deswegen eine Möglichkeit der Gruppierung von Einzelaktionen ein.

Bei dem Ausdruck `(with-action-group ...)` in der Funktionsdefinition `back` (Abschnitt 3) handelt es sich um ein Makro, dessen Expansion der *History* eine Aktion neuen Typs, die Gruppierungsaktion, hinzufügt und die darüber hinaus ein *Flag* für die Auswertung der eingeschachtelten Terme setzt.

```
(defmacro with-action-group (name &body body)
   `(progn
      (push (make-grouping-action ,name)
            *history*)
      (let ((*group-actions-p* t))
         ,@body)))
```

Die Bezeichnung (Attribut `name`) der Gruppierungsaktion ergibt sich durch Auswertung des ersten Terms im Makro, die zugehörige *Undo*-Funktion bewirkt nichts.

```
(defmethod make-grouping-action (name)
   (make-instance 'action
     :name name
     :undo-function #'(lambda () nil)))
```

[9] bzw. programmiersprachenspezifisch die Auflösung der Bindung

In Abhängigkeit vom Wert des *Flags* `*group-actions-p*` ruft die in Abschnitt 3 definierte Methode `add-action-to-history` die Methode `extend-action` auf, statt die *History* zu bedienen. `extend-action` ersetzt den Wert des Attributs `undo-function` der ersten Aktion im *History*-Keller durch eine neue Funktion, die die Hintereinanderausführung der *Undo*-Funktionen der modifizierenden Aktion und der modifizierten Gruppierungsaktion bewirkt.

```
(defmethod extend-action
    ((modified action) (modifier action))
   (setf (undo-function modified)
         (let ((previous-modified-undo
                 (undo-function modified)))
           #'(lambda ()
               (funcall
                 (undo-function modifier))
               (funcall
                 previous-modified-undo)))))
```

Komplexe Aktionen, z. B. das Kopieren oder Löschen einer Menge von Objekten, lassen sich also mit Hilfe des Makros `with-action-group` zu einer frei benennbaren Gruppenaktion zusammenfassen.

5 Anwendung

Die Vorzüge einer Implementierung der Rücksetzbarkeit durch Ergänzung des Metaobjektprotokolls verdeutlicht die folgende Änderung einer Anwendung:

```
USER<7> (setf *history* nil)
NIL
USER<8> (describe *my-article*)
#<ARTICLE #xFC48B4> is a ARTICLE:
CONTENTS:                    "On Undos"
USER<9> (defclass article ()
          ((contents
             :initform ""
             :accessor contents)
           (key :accessor key))
          (:metaclass monitored-metaclass))
#<MONITORED-METACLASS ARTICLE #xF86568>
USER<10> (setf (key *my-article*) 41)
41
USER<11> *history*
(#<Object: #<ARTICLE #xFC48B4>; Slot: KEY>)
USER<12> (back)
NIL
USER<13> (describe *my-article*)
#<ARTICLE #xFC48B4> is a ARTICLE:
CONTENTS:                    "On Undos"
KEY:                         unbound
USER<14>
```

Die Ergänzung der Definition der Klasse `article` um das Attribut `key` wirkt sich automatisch auf das `article`-Exemplar `*my-article*` aus (CLOS-Standard). Die Belegung des Attributs mit dem Wert 41 und das anschließende Zurücksetzen verdeutlichen, daß der *Undo*-Mechanismus auch nach der Änderung ohne weiteres funktioniert.

6 Fazit

Objektorientierte Programme reversibel zu gestalten erfordert keinen großen Aufwand, wie am Umfang des hier vollständig vorgestellten Programmcodes zu erkennen ist. Ein über die Kennzeichnung der Klassen reversibler Objekte hinausgehender Aufwand fällt bei der Programmierung von Anwendungen nicht an. Die Programmiersprache CLOS eignet sich in besonderer Weise dazu, eine universelle Funktionalität wie die Reversibilität zu implementieren, weil die Schnittstelle zum Metaobjektprotokoll Änderungen der Sprache selbst erlaubt. Ein weiteres typisches Beispiel solcher universeller Funktionalität ist die Implementierung von Persistenz[10].

Eine Erweiterung der vorgestellten Lösung z. B. um die individuelle Kennzeichnung einzelner Attribute als rücksetzbar ist leicht möglich.

Literatur

[Gra96] Graham, P.: *ANSI Common Lisp*, Prentice-Hall, 1996.

[KdRB91] Kiczales, G.; des Rivières, J.; Bobrow, D. G.: *The Art of the Metaobject Protocol*, MIT Press, 1991.

[Pae93a] Paepcke, A. (Hrsg.): *Object-oriented programming: the CLOS perspective*, MIT Press, 1993.

[Pae93b] Paepcke, A.: *User-Level Language Crafting*, in Paepcke, A. (Hrsg.): *Object-oriented programming: the CLOS perspective*, MIT Press, 1993.

[SKT*97] Schneeberger, J. W.; Koch, N.; Turk, A.; Lutze, R.; Wirsing, M.; Fritzsche, H.; Closhen, P.: *EPK-fix: Software-Engineering und Werkzeuge für elektronische Produktkataloge*, in Jarke, M.; Pasedach, K.; Pohl, K. (Hrsg.): *Informatik'97, Informatik als Innovationsmotor, 27. Jahrestagung der Gesellschaft fur Informatik*, Informatik aktuell, Springer, Sep. 1997.

[10] [Pae93b]