Metaobjektprotokolle

Konzepte und Entwicklungen

Alexander Konstantinov

HAW Hamburg

AW1 WS11/12

1. Abstraktionsmittel in Programmiersprachen

Bei der grundsätzlichen Tätigkeit des Programmierens versucht ein Programmierer Algorithmen, Prozesse und Systeme so präzise zu beschreiben, dass eine Maschine in der Lage ist diese Beschreibungen, welche "Programme" genannt werden, auszuführen. Obwohl er dabei theoretisch jedes Bit im Speicher setzen kann wie er will, stellt er sich selber Einschränkungen bzw. schafft Struktur um später noch in der Lage zu sein die Programme zu überblicken, anzupassen und anderen erklären zu können.

Ein sehr wichtiges Werkzeug um sich zu strukturieren (oder sich einzuschränken) stellen Programmiersprachen dar. Die Programmiersprache erlaubt dabei dem Programmierer eigene Abstraktionen zu erstellen, welche er benennen und auf Abruf wiederverwenden kann. Diese Abstraktionen werden dabei mittels der von der Programmiersprache angebotenen *Abstraktionsmitteln* erstellt.

Je nach Programmierparadigma kommen unterschiedliche Abstraktionsmittel zum Einsatz: Benannte Sprunglabels und Speicheradressen für Assemblersprachen, Subroutinen und Records bei imperativen Programmiersprachen, Funktionen und parametrisierte Module bei funktionalen Programmiersprachen sowie Objekte, Klassen und Methoden in objektorientierten Sprachen.

Dabei können gleichnamige Abstraktionsmittel in verschiedenen Programmiersprache unterschiedliche Semantik besitzen. So mögen in der einen Programmiersprache Funktionen ihre Argumente von rechts nach links auswerten, während andere die Reihenfolge unspezifizitiert lassen und wiederum andere dem Aufrufer die Wahl überlassen, weil sie nur über in der Reihenfolge vertauschbare Keyword-Parameter verfügen. Module könnten je nach Programmiersprache Untermodule enthalten oder zyklische Parametrisierung zulassen.

Üblicherweise trifft ein Sprachdesigner eine für seine Sprache passende Wahl von Abstraktionsmitteln und legt deren Semantik in einer Spezifikation fest. Die dabei entstehenden Kompromisse aus Sprachfeatures, Implementierbarkeit, Performanz und statischer Verifikation sind für jede Programmiersprache unterschiedlich und stellen wichtige Designentscheidungen dar.

Wenn Alternativsemantiken für Abstraktionsmittel oder vollkommen neue Abstraktionsmittel erwünscht sind, dann wäre der Programmierer zur Nutzung einer anderen oder zur Implementierung einer neuen Programmiersprache gezwungen, es sei denn seine Programmiersprache erlaubt es ihm die Abstraktionsmittel an sein Anforderungen anzupassen oder neue zu erstellen.

Dieses Problem mit unterschiedlichen Semantiken für Abstraktionsmittel musste gelöst werden, als ende der 80er verschiedene Lisp-Dialekte zu Common Lisp standardisiert wurden. Dazu wurde ein *Metaobjektprotokoll* entworfen [1, S. 3], über welches die objektorientierten Abstraktionsmittel des Common Lisp Object System (CLOS) konfiguriert werden konnten.

2. Objekte und Metaobjekte

Metaobjektprotokolle existieren gewöhnlich für Objektorientierte Sprachen, doch findet sich schwer eine konkrete Definition für "Objekte" oder gar Objektorientierung. Um eine für unsere Zwecke nützliche Definition für Objekte zu finden, ist es ratsam die Repräsentation von Objekten in solchen Programmiersprachen zu untersuchen.

Als Beispiel dient dazu die Repräsentation einer Farbe. Grundsätzlich könnte man eine Farbe (bei 24 Bit Farbtiefe) mit nur 32 Bit an Speicher darstellen (je acht Bit für Rot-, Grün- und Blauwert sowie acht Bit für Transparenz).

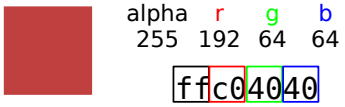

Abbildung 1. Repräsentation einer Farbe mittels 32 Bit

Obwohl der Zahlenwert `ffc04040` (in Hexadezimaldarstellung) eine Farbe darstellen kann, ließe er sich mit einer gewöhnlichen 32-Bit Ganzzahl verwechseln. Aus diesem Grund würde sich in einem objektorientierten System dieser Wert niemals gesondert auffinden, sondern stets zusammen mit einer Interpretation dieses Wertes. Die Interpretation kann in der Form eines Klassenpointers (bei klassenbasierter Objektorientierung) oder als Delegation an einen Prototypen (bei klassenloser Objektorientierung) erfolgen.

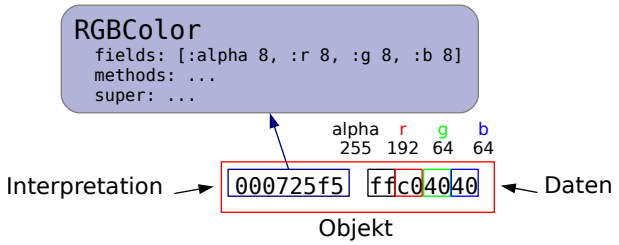

Abbildung 2. Ein RGB-Farbobjekt mit einem Klassenpointer zur Interpretation des Datums

Diese sehr allgemeine Definition für Objekte hat den Vorteil, dass sie sogleich die Frage aufwirft wie die Interpretation eines Datums repräsentiert wird. Wenn dies wiederum über ein "Objekt" geschehen kann, dann kann die Menge aller Objekte eines laufenden Programms aufgeteilt werden: Auf der einen Seite gibt es Objekte deren Daten etwas programmexternes beschreiben (z.B. Kunden, Dateien, Sockets etc.), auf der anderen Seite existieren *Metaobjekte* welche andere Objekte oder ähnliche programminterne Konzepte beschreiben (z.B. Klassen, Methoden, eigenen Quellcode etc.).

3. Metaobjektprotokolle

Idealerweise findet sich in den Metaobjekten die vom Programmierer geschaffene Strukturierung seines Programms wieder, wodurch es äußerst einfach ist rudimentäre Werkzeuge wie Klassenbrowser für Entwicklungsumgebungen zu implementieren [3, S. 4]. Dazu muss für diese Metaobjekte ein *Metaobjektprotokoll* spezifiziert werden, über welches das Programm selbstreferenziellen Zugriff auf die eigene Strukur erhalten kann.
Solche Metaobjektprotokolle können unterschiedliche Typen von Operationen anbieten, wobei sich diese in drei Gruppen ordnen lassen [3, S. 2]:

1. **Introspektion**: Lesezugriff auf die Programmstruktur.
2. **Modifikation**: Veränderung und Erweiterung der Programmstruktur.
3. **Interzession**: Veränderung der Programmiersprachensemantik.

3.1 Introspektion

Ein lesender Zugriff auf interne Programmstrukturen ist in praktisch allen objektorientierten Programmiersprachen vorhanden und wird häufig auch "reflection" genannt. Dazu können auch mutierende Operationen gehören, solange sie keinen Einfluss auf die Programmstruktur/Metaobjekte haben (z.B. das Setzen von Variablen oder Feldern über ihren symbolischen Namen).
Die Struktur der Metaobjekte spiegelt dabei die Ontologie der Abstraktionsmittel wieder. Wenn Klassen in Modulen existieren und über Felder und Methoden verfügen, dann sollten Modulobjekte Zugriff auf enthaltene Klassenobjekte, welche wiederum Methoden- und Feldobjekte enthalten, bieten. Diese Unterteilung kann grob bleiben oder auf die Ebene von Parsebäumen und Quellcode gehen.

3.2 Modifikation

Viele dynamische objektorientierte Sprachen erlauben das Erstellen und Modifizieren von Metaobjekten mittels ihres Metaobjektprotokolls. Ein Beispiel hierfür wäre die `Struct::new`-Methode in Ruby, welche in der Lage ist einfache Klassen zu generieren.

```ruby
Struct.new(:name, :age)
=> #<Class:0xb7723780>
# returns a class object

Person = Struct.new(:name, :age)
hans = Person.new("Hans", 25)
=> #<struct Person name="Hans", age=25>
# generated positional constructor

hans.age # generated reader method
=> 25

hans.age = 26 # generated writer method
hans.age
=> 26
```

Abbildung 3. Automatische Generierung von Klassen mittels `Struct::new`

Solche Techniken ersparen dem Programmierer zwar das Erstellen von einer Klassendefinition und fünf Methoden (eine Initialisierungsmethode sowie je zwei Lese- und Schreibmethoden), zerstören aber die Verknüpfung von Quellcode und Programm: Über welche Methoden Instanzen der `Person`-Klasse (bzw. des Klassenobjekts, welches der `Person`-Variablen zugewiesen wurde) verfügen ist nicht aus dem Quelltext ersichtlich.

Weitere Komplikationen entstehen sobald die Möglichkeit zur Modifikation bereits existierender Metaobjekte gegeben ist:

```
hans.city
!# NoMethodError: undefined method `city'
!#   for #<struct Person name="Hans", age=25>
# Person#city doesn't exist ...yet!

Person.attr_acessor(:city)
hans.city
=> nil # now it does

hans.city = "Hamburg"
hans.city
=> "Hamburg"
```

Abbildung 4. Modifikation von Klassen zur Laufzeit

In diesem Fall sind die Metaobjekte mit Zustand behaftet, was dafür sorgt dass ehemals ungültige Operationen nach einer Mutation gültig werden. Somit ist eine statische Repräsentation des Programms in Textform durch solche Konstrukte bedeutungslos geworden.
Aus diesem Grund verzichten Smalltalksysteme meist auf die traditionelle textuelle Repräsentation der Programmstruktur und bearbeiten und strukturieren den "Code" ausschließlich über Klassenbrowser die auf Metaobjekten operieren [4, S. 303].

3.3 Interzession

Eines der wichtigsten Prinzipien objektorientierten Designs ("Program to an interface, not an implementation.")[5, S. 30] besagt, dass ein vorbildliches objektorientiertes System, welches auf Eingabedaten grundsätzlich nur über abstrakte Schnittstellen zugreift, auch mit Eingabedaten beliebiger Implementierung arbeiten (vorausgesetzt die Schnittstellen wurden auch eingehalten) kann. Wenn man einen Interpreter einer Programmiersprache als objektorientiertes System auffasst welches Programme (in der Form von Metaobjekten) als Eingabedaten erhält, dann sollte es also keinen Unterschied machen dürfen was für eine konkrete Implementierung die Metaobjekte aufweisen, solange sie lediglich das Metaobjektprotokoll als Schnittstelle korrekt implementieren.
Bei einer Programmiersprache die über ein Metaobjektprotokoll verfügt welches eingermaßen flexible Implementierungen erlaubt, kann die Implementierung der Metaobjekte (und damit die Semantik der von ihnen repräsentierten Abstraktionen) vom Programmierer bereitgestellt werden. Diese alternative Implementation sollte dabei sich nicht nur bei Operationen auf Metaobjekten auswirken, sondern auch die Semantik von regulären Operationen beeinflussen.

Durch das Implementieren eigener Abstraktionsmittel ist zum Beispiel möglich Objekte zu haben, deren Instanzvariablen Schreibzugriffe abfangen und diese an Subscriber signalisieren (wodurch sie das Objerver-Pattern[5, S. 326] realisieren).
Wie sehr die Semantik der Programmiersprache abgeändert werden kann hängt dabei vom Design des spezifischen Metaobjektprotokolls ab, wobei die Implementierbarkeit und Performanz stets gegen Flexibilität abgewogen werden muss[6].

```
# subclass implementation for instance variables
class DebugInstVar < InstanceVariable
  def set(obj, newval)
    puts("setting '#{self.name}' to #{newval}")
    super(obj, newval) # set the instance variable
  end
end

employer_var = DebugInstVar.new(:employer)
Person.add_instance_variable(employer_var)

hans.employer = Company.get("Siemens")
#> setting 'employer' to #<Company: name="IBM"...>
```

Abbildung 5. Interzession von Zugriffen auf Instanzvariablen (in einer hypothetischen, Ruby-ähnlichen Sprache mit Metaobjekten für Instanzvariablen)

4. *Mirror*-basierte Metaobjektprotokolle

In Klassenbasierten objektorientierten Sprachen existiert in der Form von Klassen ein scheinbar idealer Ort zum Ansiedeln von Operationen eines Metaobjektprotokolls. Allerdings werden Klassen in Sprachen wie Smalltalk und Ruby auch für andere Operationen, welche keine reflektive Eingriffe auf die Programmstruktur darstellen, genutzt. Beispiele wären die Konstruktion von Objekten, für eine Klasse relevante Konstanten oder ein Verzeichnis aller jemals erstellten Objekte einer Klasse.

```
# base-level operation
Person.new("Stefan", 22)
=> #<struct Person name="Stefan", age=22>

# meta-level operation
Person.method_defined?(:age)
=> true
```

Abbildung 6. Klassenobjekt mit sowohl Base- als auch Metaleveloperationen

Somit haben Klassenobjekte zumindest zwei Zuständigkeiten: Einmal Operationen welche unabhängig von einen konkreten Instanz dieser Klasse sind (Konstruktoren, Klassenvariablen etc.), und einmal Operationen bezüglich der Struktur der Instanzen(Introspektion etc.).[1]

[1] Dies ist allerdings eher eine Beschränktheit des "message sending" OO-Paradigmas von Smalltalk etc.. In Objektsystemen mit dem "generic function"-Paradigma von Schemeimplementationen und Common Lisp können reflektive Operationen in einen gesonderten Namensraum (Modul/Package) gehören welcher nur bei Bedarf referenziert wird.

Auf der anderen Seite existieren objektorientierte Sprachen, welche nicht über Klassen verfügen und stattdessen mit Delegation und Prototypen auskommen. In solchen Programmiersprachen findet sich kein "natürlicher" Ort wo Operationen eines Metaobjektprotokolls angesiedelt werden könnten. Da die Designer der klassenlosen objektorientierten Programmiersprache *Self*, von Smalltalk inspiriert, die Entwicklung von Programmen nicht über Textdateien geschehen lassen wollten, mussten Modifikationsoperationen und die dazugehörigen Metaobjekte entworfen werden. Diese Objekte wurden "Mirrors" getauft [3, S. 9] und sind konzeptionell nicht von dem eigentlichen Objekt referenziert (im Gegensatz zu Klassenobjekten, welche mit einer `.getClass()`-Operation zugänglich sind).
Dieser Ansatz, wo der Zugriff auf sein Metaobjekt keine intrinsische Operation aller Objekte ist, erlaubt kontrollierten Zugriff auf die reflektiven Fähigkeiten einer Programmiersprache [7, S. 6]. Desweiteren kann für jedes Objekt ein eigenes Mirror-objekt erstellt werden, im Gegensatz zu Klassenobjekten, die von allen Instanzen einer Klasse geteilt werden.

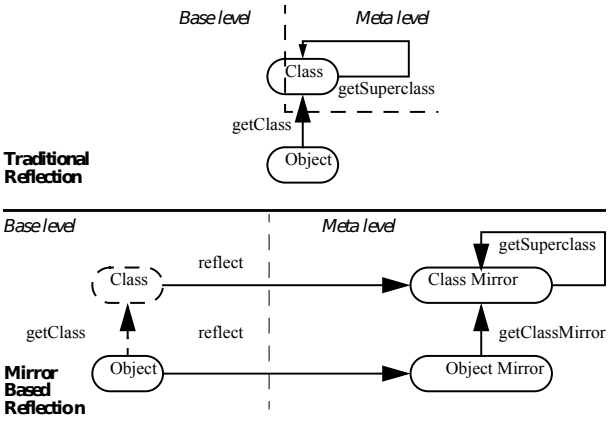

Abbildung 7. Explizite Trennung von regulären Objekten und Metaobjekten über *Mirror* (Quelle:[3])

In einer solchen Mirror-basierten Architektur ist für reflektiven Zugriff eine Referenz auf eine *mirror factory* nötig. Die erstellten Metaaobjekte und die Operationen auf diesen hängen dabei von der Bereitgestellten Factory ab. Abbildung 7 zeigt dabei, dass ein solcher Ansatz auch für klassenbasierte Objektorientierung möglich ist.

4.1 Interzession in *Mirror*-basierten Systemen

Während bei reflektiver Introspektion der Programmierer eine Operation (`.getClass()`, `mirrorOn:` etc.) hat um zu einem Objekt eine Beschreibung dieses Objektes zu erhalten (in der Form eines Klassen-, oder Mirrorobjektes), so ist zur Interzession eine inverse Operation nötig. Diese muss eine Beschreibung eines Objektes in der Form von einem Metaobjekt akzeptieren und daraus dann ein Objekt erstellen können.

```
def hans := object: {
  def name := "Hans";
  def age := 25;
}
=> <obj:108093{name,name:=,age,age:=}>
hans.age()
=> 25
def hansMirror := (reflect: hans);
=> <mirror on:<obj:108093{name,name:=,age,age:=}>>
hansMirror.listFields()
=> [<field:super>, <field:name>, <field:age>]
def ageFld := hansMirror.grabField(‘age);
=> <field:age>
ageFld.writeField(26);
hans.age();
=> 26
```

Abbildung 8. Introspektion mit der Mirror-basierten Reflection-API von AmbientTalk

```
def hansMirage := object:{}
                    mirroredBy: {|_| hansMirror};
=> <obj:108093{name,name:=,age,age:=}>
"created a transparent proxy"
hans.age := 30;
hansMirage.age();
=> 30
def hansMirage.city := "Hamburg"; "add a field"
hans.city();
=> "Hamburg"
"hans and hansMirage are operationally equivalent"
"except for object identity..."
hans == hansMirage
=> false

"create a tracing proxy"
def hansDebugMirror := extend: hansMirror with: {
  def invoke(slf, msg) {
    println("sending a "+msg.selector+" message");
    super^invoke(slf, msg);
  }
}
def debugHans := object: {}
                  mirroredBy:{|_| debugHansMirror};
debugHans.age();
>"sending a age message"
=> 30
```

Abbildung 9. Erstellung von Proxy-Objekten mittels Interzession in AmbientTalk

Diese inverse Operation wurde in der prototyp-basierten objektorientierten Programmiersprache *AmbientTalk* eingeführt. Sie erlaubt mittels der `object:mirorredBy`-Methode sogenannte "Mirages" zu erschaffen. Dies sind Objekte, deren Verhalten komplett über ein Mirror-Metaobjekt beschrieben wird.

Abbildung 9 zeigt hierbei das Verhältnis von Objeken, deren Mirrors und aus den Mirrors erschaffene Mirages[2]:
Die direkte Erstellung eines Mirages aus einem Mirror ergibt ein transparentes Proxyobjekt, welches sich bis auf Objektidentität[3] identisch zum Original verhalten muss, da die Komposition einer Operation mit ihrer Inversen die Identitionsfunktion ergibt. Wird hingegen die Implementation des Mirrors abgeändert, so ergibt sich ein Objekt mit einer veränderten Semantik. Dieser Mechanismus wird in AmbientTalk zur Realisierung verschiedenster Abstraktionsmittel der Programmiersprache genutzt (z.B. Module, Futures, Referenzen und Membranen [9, S. 682]).
Dieses Design für Interzession existiert als ein aktuelles Proposal für die Sprache *ECMAScript* (geläufig auch als JavaScript bekannt) und wird bereits von Implementierungen unterstützt [11, S. 8].

5. Fazit

Metaobjektprotokolle beschreiben wie Programme auf ihre eigene Struktur zugreifen können und werden daher bei dynamischen Programmiersprachen zur Implementierung von interaktiven Diagnose- und Entwicklungswerkzeugen. Wenn sie dabei Interzession ermöglichen, dann kann von einem Programmierer über solche Metaobjektprotokolle die Semantik der gegebenen Programmiersprache beeinflussen. Der Programmierer kann dadurch die Programmiersprache an seine spezifische Problemstellung anpassen und eventuell elegantere Programme schreiben.
Auch für den Programmiersprachendesigner bietet die Integration eines Metaobjektprotokolls in die Programmiersprache den Vorteil, dass er bestimmte Abstraktionsmittel nicht mehr in der Sprache spezifizieren muss, und sie als Libraryfeature anbieten kann. Dies erlaubt einen schlankeren Sprachkern und eine einfachere Prototypimplementationen von Sprachfeatures.
Allerdings muss beim Design der Metaobjekte einer Sprache und deren Metaobjektprotokoll darauf geachtet werden, dass Modularität in Programmen bewahrt wird auch beim Einsatz des Metaobjektprotokolls zur Metaprogrammierung. So muss sichergestellt werden, dass die eine modifizierte Semantik der Programmiersprache nur bei den Teilen des Programms gilt, wo sie auch erwünscht ist.

Literatur

[1] Gregor Kiczales and Jim Des Rivieres. 1991
The Art of the Metaobject Protocol. MIT Press, Cambridge, MA, USA.

[2] Gregor Kiczales, J. Michael Ashley, Luis H. Rodriguez, Amin Vahdat, and Daniel G. Bobrow. 1993
Metaobject protocols: why we want them and what else they can do. In *Object-oriented programming: The CLOS Perspective*, Andreas Paepcke (Ed.). MIT Press, Cambridge, MA, USA p.101-118.

[3] Gilad Bracha and David Ungar. 2004
Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (OOPSLA '04). ACM, New York, NY, USA, p.331-344.

[4] Adele Goldberg and David Robson. 1983
Smalltalk-80: The Language and its Implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995
Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[6] Gregor Kiczales and Andreas Paepcke 1996 **Open Implementations and Metaobject Protocols**. Xerox Corporation

[7] Gilad Bracha, Peter Ahe, Vassili Bykov, Eliot Miranda, and Yaron Kashai. 2008
The Newspeak Programming Platform. Cadence Design Systems

[8] Tom Van Cutsem, Stijn Mostinckx, Stijn Timbermont, and Éric Tanter. 2007
Mirages: behavioral intercession in a mirror-based architecture. In *Proceedings of the 2007 symposium on Dynamic languages* (DLS '07). ACM, New York, NY, USA, p.89-100.

[9] Tom Van Cutsem, Stijn Mostinckx, Stijn Timbermont, Elisa Gonzalez Boix, Éric Tanter, and Wolfgang De Meuter. 2009
Mirror-based reflection in AmbientTalk. In *Software—Practice and Experience* 39, 7 (May 2009), p.661-699.

[10] Henry G. Baker. 1993
Equal rights for functional objects or, the more things change, the more they are the same. In *SIGPLAN OOPS Messenger 4, 4* (October 1993), 2-27.

[11] Tom Van Cutsem and Mark S. Miller. 2010 **Proxies: design principles for robust object-oriented intercession APIs**. In *Proceedings of the 6th symposium on Dynamic languages* (DLS 2010). ACM, New York, NY, USA, p.59-72.

[2] Tatsächlich wird die `object:mirroredBy:` in der Praxis anders genutzt (was man an dem nicht verwendeten `object:`-Keywordparameter erkennt), aber zu Demonstrationszwecken kann die Operation auch mit vollständig initialisierten Mirrors verwendet werden.

[3] Eigentlich hat ein perfekt transparenter Proxy dieselbe Objektidentität wie das Original, da Identität sinnvollerweise über Mutation oder "equivalence under mutation" definiert werden sollte[10].