M. Golm, J. Kleinöder

MetaJava

Kurzfassung

Heutige objektorientierte Applikationen verlangen in zunehmendem Maße eine Adaptierbarkeit von Objekten an verschiedene Ausführungsumgebungen sowie eine Wiederverwendbarkeit von Klassen mit unterschiedlichen Anwendungsanforderungen. Mobile Objekte müssen in Umgebungen mit unterschiedlichen Sicherheitsanforderungen lauffähig sein, nebenläufige Objekte müssen in der Lage sein, zwischen optimistischer und pessimistischer Nebenläufigkeitskontrolle zu wechseln, replizierte Objekte müssen je nach Umgebung eine Replikationsstrategie wählen können, die optimale Performance erzielt.

Reflexion und Metaprogrammierung bilden eine konzeptionelle Grundlage, um diese Ziele zu erreichen. Reflexion ist die Fähigkeit eines Berechnungssystems, Informationen über den eigenen Ablaufzustand zu erhalten und diese Berechnungen zu beeinflussen. Die Unterteilung des Systems in eine Basisebene, welche das Anwendungsproblem löst, und eine Metaebene, welche die Berechnungen des Basissystems überwacht und beeinflußt, führt zur Wiederverwendbarkeit von Anwendungscode und reflexivem Code.

Wir haben einen erweiterten Java-Interpreter (JVM) - MetaJava - entwickelt, der Reflexion in Java ermöglicht. MetaJava erlaubt das selektive Binden von Metaobjekten an Objekte. Durch das selektive Binden erleiden nur die Teile der Applikation einen Performance-Verlust, die die reflexiven Fähigkeiten auch benutzen. Metaobjekte können einzelne Mechanismen ihres Basisobjektes, wie Methodenaufrufe oder Instanzvariablenzugriffe reimplementieren und damit die von der JVM vorgegebene Semantik erweitern. Wir haben diese Möglichkeiten unter anderem benutzt, um einen entfernten Methodenaufruf zu implementieren und um Objekte aktiv und passiv zu replizieren. Weitere Metaobjekte, etwa für Sicherheitsmechanismen und Synchronisation, befinden sich gerade in Entwicklung.

Adaptierbarkeit und Wiederverwendbarkeit

Moderne verteilte Applikationen stellen Anforderungen an Betriebssysteme[1], die die heutigen Systeme nur unzureichend erfüllen können. Solche Anforderungen bestehen zum Beispiel darin, Objekte persistent zu halten, zu replizieren oder die Aktivitäten in nebenläufigen Anwendungen zu koordinieren. Weiterhin erfordert die Migration von Objekten zwischen verschiedenen Rechnern, daß Objekte an verschiedene Ausführungsumgebungen adaptierbar sind. Eine solche Adaptierbarkeit kann nur erreicht werden, indem Programme ihre eigene Struktur und ihr Ausführungsverhalten beobachten und modifizieren können. Es gibt einige Ansätze, dies in herkömmlichen Systemen, zum Beispiel durch Verwendung spezieller Design-Pattern, zu ermöglichen. Diese Lösungen haben allerdings nicht die Mächtigkeit und breite Anwendbarkeit, die die Verwendung von Reflexion bietet. Ebenso wie der Übergang von prozeduraler zu objektorientierter Programmierung, verbessert der Übergang von objektorientierter Programmierung zu reflexiver Programmierung die Wiederverwendbarkeit von Programmfragmenten.

---

[1] Wir verstehen unter Betriebssystem alle Komponenten, die die Ausführungsumgebung einer Applikation darstellen, also auch Laufzeitsystem-Bibliotheken und virtuelle Maschinen.

Reflexion

Nach Maes [Maes87] ist Reflexion die Fähigkeit eines Berechnungssystems „Aussagen über sich selbst zu treffen und auf sich selbst einzuwirken“. *Metaprogrammierung* ist eine Methode, um Programmteile, die der Lösung des Anwendungsproblems dienen (*funktionaler Code*) von Programmteilen zu trennen, die Überwachungs- und Steuerungsaufgaben wahrnehmen (*nicht-funktionaler Code*). Die Aufteilung in *Basisebene* (funktionaler Code) und *Metaebene* (nicht-funtionaler Code) ermöglicht es, das Ausführungsverhalten von Basisebenen-Objekten in einer für die Basisebenen-Objekte transparenten Art zu kontrollieren. Um diese Kontrolle zu ermöglichen, müssen einige Aspekte der Berechnung reifiziert werden. Unter *Reifikation* versteht man das Verfügbarmachen von Dingen, die normalerweise nicht Bestandteil des Programmiermodells sind. Zum Beispiel ist die Operation eines Methodenaufrufs in den meisten objektorientierten Programmiersprachen nur implizit verfügbar und der Mechanismus „Methodenaufruf“ damit auch nicht änderbar.

In [Ferber89] wird zwischen *struktureller Reflexion* und Reflexion des Berechnungsvorgangs (behavioral reflection, *Verhaltensreflexion*) unterschieden. Strukturelle Reflexion reifiziert strukturelle Aspekte eines Programms, wie Datentypen, Methodennamen oder Vererbungsbeziehungen. Ein Beispiel für strukturelle Reflexion ist das Java 1.1 Reflection API [Sun97]. Verhaltensreflexion reifiziert Operationen des Laufzeitsystems, wie Methodenaufrufe, Variablenzugriffe oder Objektinstantiierungen. MetaJava ist ein System, welches sowohl strukturelle als auch Verhaltensreflexion unterstützt.

Die Verwendung von Reflexion und insbesondere die Trennung von Basis- und Metaebene bewirken eine Trennung von Zuständigkeiten (*Separation of Concerns*) [HüLo95] sowie die Transparenz der Metaebene. Dies wiederum führt zu einer erhöhten Produktivität des Programmierers durch bessere Strukturierungsmöglichkeiten und bessere Verständlichkeit der Programme. Die Wiederverwendbarkeit und Konfigurierbarkeit von Objekten wird verbessert.

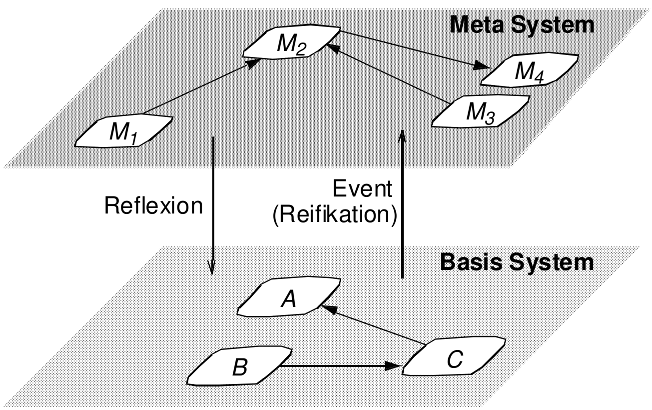


Abbildung 1: Reflexion und Reifikation

MetaJava Programmiermodell

Traditionell werden Systeme durch ein Schichtenmodell strukturiert. Untere Schichten (z.B. ein Betriebssystem) implementieren Dienste, die höhere Schichten (z.B. eine Anwendung) durch ein sogenanntes Application Programmer Interface (API) benutzen. Die Verwendung dieses Modells hat zur Folge, daß die Anwendungsschicht keinen Einfluß auf die Implementierung in der Betriebssystemschicht hat und diese nicht an ihre eigenen Bedürfnisse anpassen kann.

Da ein Betriebssystem Überwachungs- und Steuerungsaufgaben für Anwendungen erbringt, entspricht die Funktionalität eines Betriebssystems (oder Laufzeitsystems) der Funktionalität einer Metaebene, wie sie im vorigen Abschnitt definiert wurde.

Dementsprechend ermöglicht das MetaJava-System, Mechanismen, die das Laufzeitsystem anbietet, an spezielle Anwendungsbedürfnisse anzupassen. Dies geschieht durch die Installation von Metaobjekten, welche ausgewählte Mechanismen des Laufzeitsystems umdefinieren können.

Einem Metaobjekt steht eine Schnittstelle zur Manipulation von internen Strukturen der *MetaJava-Virtual-Machine* (MJVM) zur Verfügung. Diese Schnittstelle wird *Meta-Level-Interface* (MLI) der MJVM genannt.

Wenn ein Metaobjekt eine spezialisierte Ausführungsumgebung für ein Objekt herstellen soll, muß dieses Metaobjekt zuerst an das Objekt gebunden werden[2]. Dies geschieht durch den Aufruf `attachObject` am MLI. Um die Kontrolle von der Basisebene an die Metaebene zu transferieren (Reifikation), verwendet MetaJava einen Event-Mechanismus. In der derzeitigen Implementierung können folgende Operationen eines Basisebenen-Objektes *O* ein Event auslösen und zu einem Kontrolltransfer an das an *O* gebundene Metaobjekt führen:

- eine Methode von *O* wird aufgerufen
- *O* greift auf seine Instanzvariablen zu
- die mit *O* assoziierte Sperre (Lock) wird angefordert oder freigegeben
- *O* ruft eine Methode auf

Weiterhin existieren Events, die nicht mit bestimmten Objekten assoziiert sind:

- eine Klasse wird geladen
- eine neue Instanz einer Klasse wird erzeugt

Nachdem das Metaobjekt an ein Basisobjekt gebunden wurde, kann das Metaobjekt dem System mitteilen, daß es Interesse an bestimmten Events hat, d.h. es registriert sich für diese Events. Zu jedem der Events kann das Metaobjekt den von der Klasse `MetaObject` geerbten Default-Eventhandler überschreiben. In diesem Event-Handler reimplementiert das Metaobjekt den entsprechenden Mechanismus. Für alle Mechanismen gibt es eine Default-Implementierung, die über das MLI aktiviert werden kann.

Die Metaebene ist nur für die Bindung von Metaobjekten an Basisobjekte sichtbar. Nachdem die Metaobjekte gebunden und konfiguriert wurden, ist die Metaebene vollkommen transparent für das Basissystem.

Implementierung

Bei der Implementierung der Reflexion in MetaJava haben Effizienzüberlegungen eine große Rolle gespielt. Viele Systeme, die Reflexion auch zur Laufzeit eines Programms ermöglichen, beruhen auf einem interpretativen Ansatz, d.h. die Anweisungen des Programms werden interpretiert und während der Interpretation wird überprüft, ob ein Kontrolltransfer an ein Metaobjekt nötig ist. MetaJava verwendet einen anderen Ansatz und ist nicht auf die Modifizierung der Hauptschleife (Fetch-Decode-Loop) des Java-Interpreters angewiesen.

---

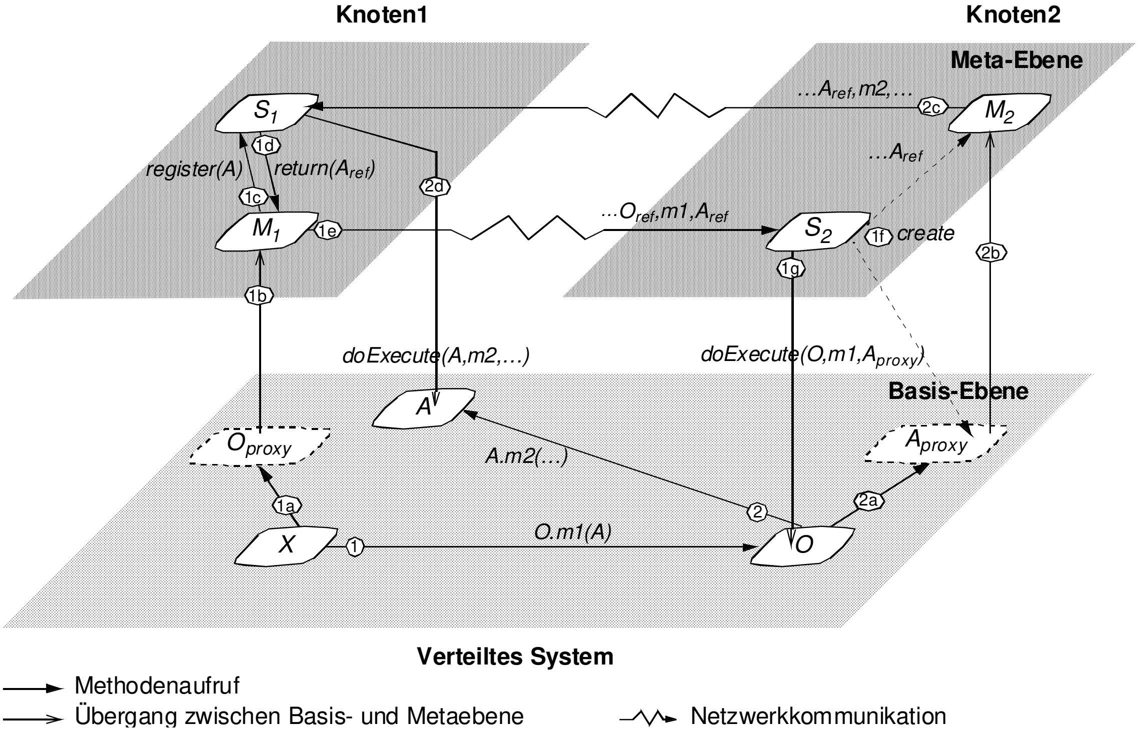
[2] Es gibt außerdem die Möglichkeit, Metaobjekte an Referenzen und Klassen zu binden.

Ein grundlegendes Problem bei der Implementierung von objektbasierter Reflexion in klassenbasierten Sprachen wird dadurch verursacht, daß die Semantik eines Objektes durch die Klasse bestimmt wird und deshalb alle Objekte diese Klasse gemeinsam benutzen, andererseits ein Metaobjekt aber nur die Semantik eines ganz bestimmten Objektes ändern soll. In MetaJava wird dieses Problem durch die Verwendung von sogenannten *Shadow-Klassen* gelöst. Wenn ein Metaobjekt an ein Basisobjekt gebunden wird und das Metaobjekt die Klasse des Basisobjektes modifiziert, wird eine Kopie dieser Klasse angelegt, die vom restlichen System von der Originalklasse nicht unterscheidbar ist. Die Probleme, die mit der Einführung von Shadow-Klassen zusammenhängen, sind in [GoKl97] und [Golm97] erläutert. Der Shadow-Klassen-Mechanismus begrenzt die Performanceeinbußen, die durch Reflexion entstehen, auf die Programmteile und Objekte, die die Reflexion auch benutzen.

Beispiel: Entfernter Methodenaufruf

Abbildung 2 skizziert das Szenario eines entfernten Methodenaufrufs unter Verwendung von Metaobjekten.

Im Unterschied zu Javas Remote-Method-Invocation (RMI) ist der Mechanismus des entfernten Methodenaufrufs in MetaJava nicht Bestandteil des Laufzeitsystems. Es lassen sich daher verschiedene Protokolle verwenden, sogar mehrere Protokolle gleichzeitig in einer Applikation.

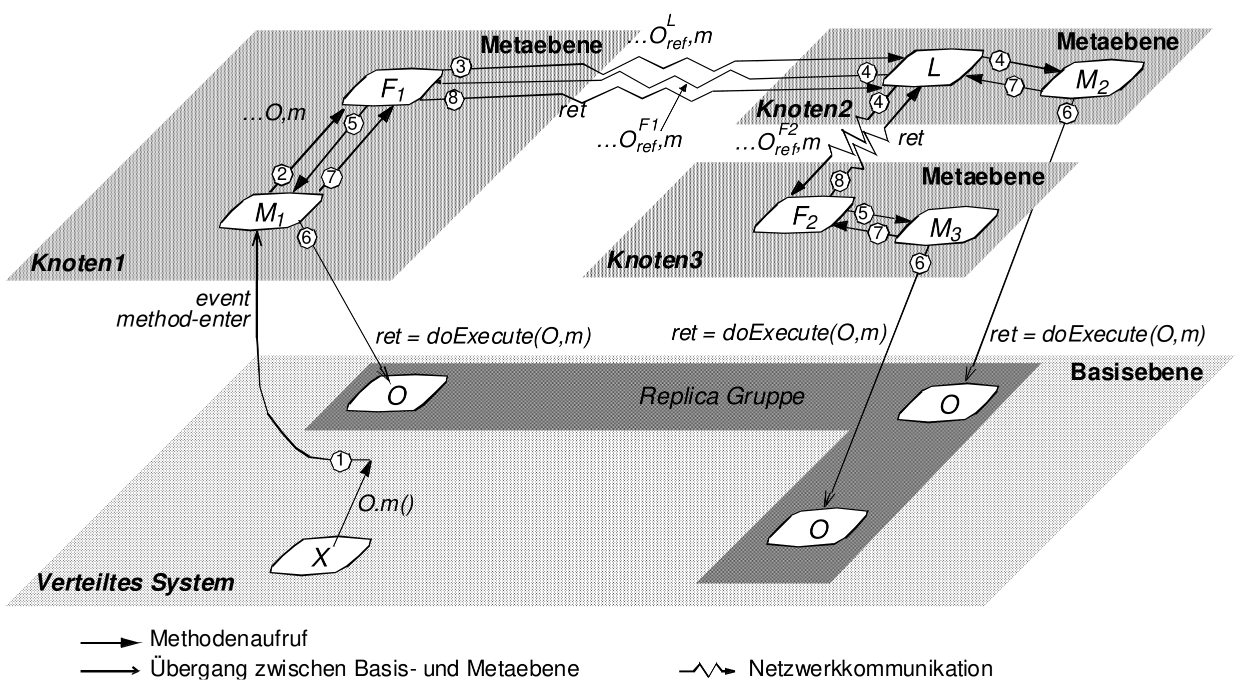


(1) Objekt X ruft Methode m_1 des entfernten Objektes O auf. Diese Basisebenen-Aktion ist auf der Metaebene wie folgt implementiert: (1a) Da O auf einem anderen Knoten läuft, ruft X tatsächlich O_{proxy} auf. (1b) Das Metaobjekt M_1 ist an O_{proxy} gebunden und empfängt das *method-enter-Event*. (1c) M_1 registriert das Argument-Objekt A beim Objektserver S_1 und (1d) erhält das ortsunabhängige Handle A_{ref}. (1e) Mit diesem Handle, dem Methodennamen und dem Handle O_{ref} für das Basisobjekt O wird der Server S_2 aufgerufen. (1f) Der Server S_2 installiert das Proxy A_{proxy} für das Argument-Objekt auf dem Knoten 2 und bindet Metaobjekt M_2 daran. (1g) Der Server S_2 ruft das Basisebenen-Objekt O unter Verwendung der MLI-Methode `doExecute` auf. O führt den Methodencode aus (Basisebenen-Berechnung) und (2) ruft Methode m_2 des Argumentobjektes A auf (und erreicht A_{proxy} (2a)). (2b) Dieser Aufruf wird von Metaobjekt M_2 abgefangen und (2c) an S_1 delegiert (Knoten 1). S_1 bildet A_{ref} auf die lokale Referenz A ab und (2d) springt zurück auf die Basisebene mit `doExecute`.

Abbildung 2: Ablauf eines entfernten Methodenaufrufs

Beispiel: Aktive Replikation

Replizierte Objekte sind ein weiteres Beispiel für ein erweitertes Objektmodell. In diesem Beispiel verwenden wir aktive Replikation, d.h. jeder Methodenaufruf am replizierten Objekt wird durch jedes Replikat ausgeführt. Dabei läuft auf jedem Knoten ein Metaobjekt und ein *ReplicationController*. Der ReplicationController ist entweder *Leader* oder *Follower*. Ein Leader wartet auf Methodenaufruf-Nachrichten und verteilt diese an die Follower. Abbildung 3 illustriert die Abläufe beim Aufruf einer Methode an einem replizierten Objekt.



Objekt *X* ruft Methode *m* des replizierten Objektes O auf. Diese Basisebenen-Aktion ist auf der Metaebene wie folgt implementiert: (1) Metaobjekt *M* ist an *O* gebunden und erhält dadurch das *method-enter-Event*. (2) *M* informiert den lokalen *ReplicationController* über den versuchten Methodenaufruf. (3) Der *ReplicationController* ist ein *Follower* und sendet deshalb die Nachricht weiter zum *Leader* der Gruppe. Die Nachricht enthält die Objekt-ID, den Methodennamen und die Methodensignatur. (4) Nachdem er die Nachricht erhalten hat, erzeugt der *Leader* einen neuen Thread, um die Nachricht an alle *Follower* und die lokalen Replicate zu schicken. Der *Leader* wartet auf die Resultate aller *Follower* und des lokalen Metaobjekts. (5) Die *Follower* informieren das Metaobjekt, mit der Methodenausführung zu beginnen. (6) Die Metaobjekte $M_{1..3}$ führen die Methode *m* am Basisebenen-Objekt *O* unter Verwendung der MLI-Methode `doExecute` aus. (7) Der Rückgabewert von Methode *m* wird an $F_{1..2}$ und *L* geliefert. (8) $F_{1..2}$ senden ihr Resultat an *L*.

Abbildung 3: Ablauf eines Methodenaufrufs an einem replizierten Objekt

Weitere Anwendungen von MetaObjekten

Das von MetaJava bereitgestellte Framework läßt sich zur Implementierung beliebiger erweiterter Objektmodelle benutzen. Einige von uns untersuchte Gebiete sind aktive Objekte, Koordinierung und Sicherheit.

Aktive Objekte. Aktive Objekte werden benutzt, um Nebenläufigkeit zu kontrollieren. Ein aktives Objekt besitzt einen eigenen Thread. Nur dieser Thread wird zur Ausführung von Methoden des Objektes verwendet. Ankommende Methodenaufrufe werden in einer Warteschlange verwaltet. In unserer Implementierung sind aktive Objekte normale Objekte, an die ein Metaobjekt `MetaActive` gebunden wurde, welches sich für den Methodenaufruf-Event registriert und alle ankommenden Methodenaufrufe in die Warteschlange einfügt.

Koordinierung. Neben der Verwendung von aktiven Objekten gibt es noch eine Reihe weiterer Koordinierungsmechanismen, die mit Hilfe von Metaobjekten implementiert werden können. Ein schon relativ altes, aber mit nicht-reflexiven Systemen schwer zu realisierendes Verfahren sind Pfadausdrücke [Campbell74].

Sicherheit. Mit Metaobjekten ist es möglich, den Zugriff auf Objekte auf verschiedene Weise zu regulieren. Ein mit MetaJava einfach zu implementierendes Verfahren sind Access-Control-Listen. Überlegungen zur Implementierung von Sicherheitsmechanismen basierend auf Metaobjekten sind in [Riechmann96] erläutert.

Objekt-Migration. Ein Meta-System kann zur Implementierung migrierender Objekte und zur Steuerung der Migration verwendet werden.

Verwandte Arbeiten

Die Erforschung von Reflexion begann im Bereich der Lisp-basierten Programmiersprachen mit den Arbeiten von Smith [Smith82]. Maes [Maes87] hat die Anwendung von Reflexion in objekt-orientierten Programmiersprachen studiert und das Konzept des Metaobjekts eingeführt. In den folgenden Jahren entstand eine Reihe objekt-orientierter reflexiver Programmiersprachen [ChMa93], [OkIsTo93], [Cointe93], [McAffer95], [Chiba96]. Bei der überwiegenden Anzahl der Systeme haben Performance-Überlegungen nur eine untergeordnete Rolle gespielt. Eine Ausnahme stellt [Chiba96] mit der Sprache OpenC++ dar. OpenC++ unterstützt Reflexion zur Übersetzungszeit und ist dadurch sehr effizient aber auch weniger mächtig als ein System, das Reflexion zur Laufzeit erlaubt.

Zusammenfassung und zukünftige Arbeiten

Es wurde beschrieben, wie Reflexion in MetaJava verwendet werden kann und wie mit Hilfe von Reflexion Erweiterungen der Laufzeitumgebung von Objekten realisiert werden können. Beispielhaft wurden die Systeme für entfernten bzw. replizierten Methodenaufruf erläutert.

Zukünftige Arbeiten werden sich mit der Komposition und Konfiguration von Metasystemen beschäftigen. Weiterhin ist geplant, andere Mechanismen des Laufzeitsystems, wie zum Beispiel Thread-Wechsel oder Garbage Collection, zu reifizieren.

Weitere Informationen über MetaJava sind im WWW unter
http://www4.informatik.uni-erlangen.de/metajava
verfügbar.

Literatur

[Campbell74]
R. H. Campbell, N. Habermann. The Specification of Process Synchronization by Path Expressions. *Lecture Notes in Computer Science*, vol. 16, Springer Verlag, New York, 1974, pp. 89-102.

[ChMa93]
S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. *Proceedings of ECOOP '93, the 7th European Conference on Object-Oriented Programming*, Kaiserslautern, Germany, LNCS 707, Springer-Verlag, pp. 482–501.

[Chiba96]
S. Chiba. *OpenC++ Programmer's Guide for Version 2*. Technical Report SPL-96-024, Xerox PARC, 1996.

[Cointe93]
P. Cointe. Definition of a Reflective Kernel for a Prototype-Based Language. *International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, LNCS 742, Springer-Verlag, Nov. 1993.

[Ferber89]
J. Ferber. Computational Reflection in class based Object-Oriented Languages. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '89*, New Orleans, La., Oct. 1989, pp. 317–326.

[GoKl97]
M. Golm, J. Kleinöder. MetaJava – A Platform for Adaptable Operating-System Mechanisms. *ECOOP '97 Workshop on Object-Orientation and Operating Systems*. June 1997.

[Golm97]
M. Golm. *Design and Implementation of a Meta-Architecture for Java*. Diplomarbeit, Universität Erlangen, Januar 1997.

[HüLo95]
W. L. Hürsch, C. V. Lopes. *Separation of Concerns*. Technical Report NU-CCS-95-03, Northeastern University, Boston, February 1995.

[Maes87]
P. Maes. *Computational Reflection*. Technical Report 87_2, Artificial Intelligence Laboratory, Vrieje Universiteit Brussel, 1987.

[McAffer95]
J. McAffer. Meta-Level Architecture Support for Distributed Objects. Proceedings of the 4th International Workshop on Object Orientation in Operating Systems, Lund, Sweden, IEEE, 1995, pp. 232–241.

[OkIsTo93]
H. Okamura, M. Ishikawa, and M. Tokoro. Metalevel Decomposition in AL-1/D. *International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, LNCS 742, Springer-Verlag, Nov. 1993.

[Riechmann96]
T. Riechmann. *Security in Large Distributed, Object-Oriented Systems*. Technical Report TRI4-02-96, University of Erlangen-Nürnberg, IMMD IV, Mai 1996.

[Smith82]
B. C. Smith. *Reflection and Semantics in a Procedural Language*. Ph.D. Thesis, MIT LCS TR-272, Jan. 1982.

[Sun97]
Sun Microsystems. *Java Core Reflection, API and Specification*, 1997.

Autoren:

Dipl.-Inf. Golm, M.
Dr.-Ing. Kleinöder, J.
Universität Erlangen, Martensstr. 1
91058 Erlangen
Tel.: 09131/857909, Fax: 09131/858732
e-mail: {golm,kleinoeder}@informatik.uni-erlangen.de