Semantik (fast) ohne Syntax: Anmerkungen

Meta-Programmierung in der Praxis

Jürgen Lampe

Verglichen mit der Frühzeit der Informatik spielen Standardprogrammiersprachen als Gegenstand der Forschung nur noch eine sehr untergeordnete Rolle. Das ist angesichts der angehäuften Erkenntnisse und der beschränkten Möglichkeiten, diese in die Praxis umzusetzen (Wie oft wird eine neue Sprache entwickelt?), zwar verständlich, aber aus den folgenden Gründen bedauerlich:

- Trotz des von R. P. Gabriel schon 1993 verkündeten „Endes der Geschichte der Programmiersprachen" [1] ist die Entwicklung seitdem lebhaft weitergegangen. Der Großteil der Neuerungen wurde und wird dabei allerdings von Praktikern getragen.
- Da dieses Gebiet nicht mehr so attraktiv erscheint, findet das vorhandene Wissen weniger Verbreitung, was dazu führt, dass vermeidbarer Aufwand getrieben wird oder mögliche bessere Lösungen übersehen werden.

Das Zusammenwirken dieser beiden Faktoren stellt auf längere Sicht ein Risiko für die Softwareentwicklung dar. Immer noch wird der weitaus größte Teil der Software in den verbreiteten Programmiersprachen C/C++, Java oder C# geschrieben. Nicht nur deren Entwicklungsumgebungen und Bibliotheken haben sich weiterentwickelt, sondern auch die Sprachen selbst. Leistungsfähige Prozessoren und riesige Speicher erlauben die breite Verwendung von Methoden, die früher speziellen Anwendungen vorbehalten waren. Eine dieser Techniken ist die Bereitstellung von Meta-Informationen zur Programmlaufzeit. Ursprünglich für das „Laden bei Bedarf" [13] notwendige Informationen über Routinen und sichtbare Felder sind zu *Reflection*-API ausgebaut worden. Das vereinfacht nicht nur die Implementierung von Werkzeugen aller Art, es erlaubt auch mittels Meta-Programmierung das Schreiben sich selbst modifizierender Programme.

Eine andere bemerkenswerte Entwicklung stellen die Annotationen oder Attribute dar. Sie sind prinzipiell nichts anderes als eine Emanzipation dessen, was früher Compiler-Option hieß und in sogenannte Pseudokommentare gepackt wurde, zu einem vollwertigen Sprachbestandteil – daraus ist ein mächtiges Werkzeug erwachsen. Auf diesem Weg, sozusagen durch die Hintertür und ohne dass dies im erforderlichen Maß wissenschaftlich reflektiert worden wäre, ist Meta-Programmierung mittlerweile in der täglichen Praxis angekommen und nimmt – mit wachsendem Tempo – neue Anwendungsfelder in Beschlag.

Meta-Programmierung ist in der Forschung intensiv untersucht worden. Weniger Beachtung hat die Frage gefunden, wie diese Technik zweckmäßig in den Softwareentwicklungsprozess eingebunden werden kann, ohne wichtige Qualitätsmerkmale wie Codelesbarkeit und Wartbarkeit zu beeinträchtigen.

Im Folgenden werden am Beispiel von Javas Annotationen einige wichtige Fragen diskutiert, die sich in dieser Hinsicht ergeben.

DOI 10.1007/s00287-014-0837-x

Jürgen Lampe
Agon Solutions GmbH,
Franfurter Str. 71–75, 65760 Eschborn
E-Mail: juergen.lampe@agon-solutions.de

Begriff und Anwendung

Annotationen

Wie der Name vermuten lässt, sind Annotationen oder Attribute zunächst einmal lediglich zusätzliche Informationen, die bestimmten Programmelementen zugeordnet werden können. Ein typisches Beispiel ist die Annotation *@Override* an einer Methodendeklaration, die anzeigt, dass damit eine in der Basisklasse vorhandene Methode überschrieben wird. Der Compiler kann so eine Prüfung auf beispielsweise Schreibfehler vornehmen, die sonst unentdeckt bleiben würden.

Neben solchen vordefinierten Annotationen können beliebig weitere definiert werden. Dabei ist jeweils anzugeben, ob die Zugreifbarkeit auf den Quellcode oder den übersetzten Bytecode (.class-Datei) begrenzt ist oder auch die Laufzeit umfasst. Dementsprechend kann die angemerkte Information auf maximal drei Ebenen ausgewertet werden:

1. Durch den Compiler: Neben der Prüfung der syntaktischen Richtigkeit, die immer erfolgt, können über eine Schnittstelle *Annotation Processors* eingebunden werden. Das sind beliebige Java-Bibliotheken, die diese Schnittstelle bedienen und die Funktionalität des Compilers erweitern.
2. Durch den Klassenlader (*class loader*): Beim Laden der Bytecode-Dateien können die Annotationen durch anwendungsspezifische Lader-Implementierungen ausgelesen werden. Die Möglichkeiten, dabei den Bytecode zu verändern oder zu ersetzen, bevor er als ausführbarer Code in den Speicher geschrieben wird, sind ebenfalls praktisch unbegrenzt.
3. Während der Laufzeit: Über Methoden des Reflection-API kann jederzeit auf die zur Laufzeit verfügbaren Annotationen zugegriffen werden. Zudem gibt es Wege, auch bereits geladene Klassen durch modifizierte zu ersetzten.

Daneben können Annotationen natürlich auch von weiteren Werkzeugen ausgewertet werden, um zum Beispiel erweiterte Dokumentationen oder Datenbank-Zugriffscode zu generieren.

Die Vielfalt der mit Annotationen verbindbaren Funktionen macht es nahezu unmöglich, ihre Rolle innerhalb der Programmiersprache klar zu charakterisieren. Aus diesem Grund werden hier einige typische Verwendungen dargestellt. Schon diese wenigen Beispiele erlauben es, mögliche Konsequenzen für die Softwareentwicklung zu diskutieren.

Der einfache Fall, dass Annotationen im unmittelbaren Wortsinn als ergänzende Informationen eingesetzt werden, die die Verständlichkeit des Codes verbessern, gibt keinen Anlass zu Kontroversen und bleibt deshalb im Weiteren unberücksichtigt.

Beispiel

Das folgende kleine Beispiel versucht, einen Eindruck davon zu vermitteln, wie durch die Verwendung von Annotationen der Programmierstil insgesamt verändert wird. Als Demonstrationsobjekt dienen die in der Java-Welt verbreiteten *Enterprise Java Beans* (EJB). Das sind standardisierte Komponenten, die in Applikationsservern verschiedener Hersteller eingesetzt werden können. Wenn eine EJB zwischen Aufrufen ihrer Servicemethoden keine Informationen speichert, wird sie *zustandslos* genannt. Eine solche zustandslose EJB hat ohne Annotationen (EJB 2.1 [7]) folgende Struktur:

```
public class BspBean implements javax.
    ejb.SessionBean{ // vorgeg. Interface

  private SessionContext context;
  private DataSource dbDatasource;

  public void setSession
       Context(SessionContext ctx) {
    context= ctx; // Initialisierung
  }
  public void ejbCreate() {
    dbDatasource= (DataSource)context.
      lookup(…); // Initialisierung
  }
  public void ejbActivate() { }
                          // weitere Methoden
  public void ejbPassivate() { } // des
  public void ejbRemove() { } // Interfaces
  …
  public void bspService(…) {…}
            // implementierte Funktion(en)
  …
}
```

Die Bean muss ein vorgegebenes Interface implementieren und erforderliche Initialisierungen sind auszuprogrammieren. Zusätzlich ist ein *Deployment-Descriptor* im XML-Format zu schrei-

ben, in dem die Eigenschaften, zum Beispiel „zustandslos", oder die Einbindung in Transaktionen definiert werden. Mit der EJB-Version 3.0 [8] hat sich das durch die Verwendung von Annotationen wesentlich vereinfacht:

```
@Stateless
public class BspBean {

    @Resource private DataSource
            dbDatasource;
    …
    public void bspService(…) {…}
    …
}
```

Jede Klasse kann durch Annotierung zur EJB gemacht werden. Die Initialisierung der Datenquelle erfolgt implizit. Es wird kein zusätzlicher Deployment-Descriptor benötigt. Der Anteil des formal vorgegebenen Codes ist wesentlich geringer. Die Vorteile beim Schreiben liegen auf der Hand. Allerdings verliert man dabei die Kontrolle über einen Teil der Ausführung. Während der Compiler Fehler bei der Implementierung eines Interfaces sofort bemerkt, kann das Vergessen einer Annotation möglicherweise erst zur Laufzeit Folgen haben.

Konfiguration

Bei dieser Form der Verwendung werden Angaben, die sich auf die Ausführungsumgebung des Programms beziehen, in den Code eingestreut. Das Persistenz-Framework Hibernate [3] erlaubt es beispielsweise, den Namen der für das Abspeichern einer Klasse in der Datenbank zu benutzenden Tabelle in der Form `@Table(name = "TABELLE_A")` direkt an die Klassendefinition zu heften. In analoger Weise können einzelne Felder mit Eigenschaften der jeweils zu nutzenden Tabellenspalten annotiert werden. Hibernate ist dabei nur einer von zahlreichen Anwendern dieses Wegs.

Der unmittelbare Vorteil von Annotationen gegenüber anderen Formen der Konfiguration liegt darin, dass die Spezifikation des jeweils betroffenen Elements, im Beispiel der vollständige Name der Entitätsklasse, entfällt. Zudem kann durch den Compiler überprüft werden, ob Parameternamen richtig geschrieben und erforderliche Parameter angegeben wurden. Das macht es leichter, Konfiguration und Code konsistent zueinander zu halten.

Nachteilig ist an diesem Vorgehen, dass Konfigurationsänderungen in ihren Auswirkungen Programmcode-Änderungen sind. Man versucht, diesem Dilemma dadurch zu entgehen, dass die Konfiguration aufgeteilt wird. Diejenigen Angaben, die stark mit dem Code verknüpft sind, zum Beispiel die Kennzeichnung von Entitätsklassen (@Entity) oder Relationsarten (@ManyToMany), werden als Annotationen in den Code eingebettet. Die Angaben, die relativ unabhängig sind, wie die Festlegung eines Tabellennamens, werden separiert in eigenen Dateien verwaltet. Abgesehen davon, dass die Zuordnung zu einem der beiden Teile nicht objektivierbar ist, wird die Wartung der Konfiguration durch diese Aufteilung nicht einfacher.

Konfiguration war eines der ersten Anwendungsgebiete, in denen sich selbstdefinierte Annotationen verbreitet haben. Das lag nicht zuletzt an der Unzufriedenheit mit den vorher fast ausschließlich verwendeten Konfigurationsdateien im XML-Format. Die Frage, ob stattdessen nicht einfacher zu verwendende (und weniger aufwendig zu verarbeitende) formale Sprachen geeignet wären, wird leider kaum diskutiert. Annotationen bieten vor allem in der Entwicklungsphase praktische Vorteile. Für die Bewertung innerhalb des gesamten Software-Lebenszyklus ist der folgenden Einschätzung nichts hinzuzufügen: „Annotations are new and cool now, but the more extensively they are being used, the more they are polluting your source code. Don't let annotation hell follow XML configuration hell" [9].

Codegenerierung

Wenn Annotationen für die Konfiguration verwendet werden, beeinflusst das zwar das Softwaresystem insgesamt, aber der auszuführende Code selbst wird nicht verändert. Das gilt nicht mehr, wenn mittels Annotationsprozessoren auf die Übersetzung durch den Compiler selbst Einfluss genommen wird. Die Schnittstelle zur Anbindung solcher Prozessoren ist offizieller Bestandteil der Compiler-API. Der Aufruf erfolgt im ersten Schritt der Quellcode-Verarbeitung, in dem ein abstrakter Syntaxbaum (AST) aufgebaut wird [10]. Da in Java zyklische Abhängigkeiten zwischen Klassen zulässig sind, kann dieser Schritt wiederholt ausgeführt werden, bis alle Abhängigkeiten aufgelöst und alle Annotationen verarbeitet sind. Die Einbeziehung zwischenzeitlich neu generierten Codes ist ausdrücklich zugelassen.

Darüber hinaus gibt es Werkzeuge, die neben der Generierung neuen Codes auch eine Manipulation des AST zulassen. Beim *Project Lombok* [11] ist es möglich, Bytecode erzeugen zu lassen, der keine direkte Entsprechung im Quellcode hat. Beispielsweise kann durch Annotieren einer Klasse mit @toString erreicht werden, dass für diese Klasse die ererbte Methode durch eine klassenspezifische überschrieben wird, ohne dass diese irgendwo explizit in Quellcodeform existiert.

Auch bei dieser Anwendung liegen die Vorteile vorrangig in der Entwicklungsphase. Die Generierung erspart das Schreiben großer Mengen schematischen, sogenannten *Boilerplate*-Codes. Generierter Code enthält außerdem erfahrungsgemäß weniger triviale Fehler.

Der Nachteil ist vor allem darin zu sehen, dass der ursprüngliche Quellcode nur noch einen Teil dessen dokumentiert, was während der Laufzeit tatsächlich ausgeführt wird. Auch wenn der generierte Code prinzipiell verfügbar ist, lässt sich die Verbindung, zwischen Quellen und generierten Artefakten nicht unmittelbar herstellen. Der betreffende Annotationsprozessor ist dadurch implizit Teil der Programmdokumentation. Das gilt in noch stärkerem Maße, wenn der Code *on the fly* nur im Compiler modifiziert wird. Dann ist man zudem weitgehend auf die Dokumentation des Annotationsprozessors angewiesen, weil es schwierig sein dürfte, aus dem Code, der einen AST manipuliert, zu erschließen, was da letztlich gemacht wird.

Bytecode-Modifikation

Die Modifikation des Codes beim Laden oder später während der Ausführung ist ein Vorgehen, dass vor allem bei der Laufzeitanalyse, z. B. beim Profiling, von Programmen seit langem angewendet wird. Auch im Rahmen der aspektorientierten Programmierung wird es eingesetzt. Relativ neu ist der Ansatz, bei umfangreicheren Systemen auf diese Weise eine gewisse Entkopplung ihrer Bestandteile zu erreichen. Die Idee dahinter ist, dass Komponenten unabhängig voneinander entwickelt und übersetzt werden und erst beim Laden auf der Basis der annotierten Informationen miteinander verbunden werden.

Technisch ist die Realisierung kein Problem, da die Laufzeitumgebung – in diesem Fall die Java Virtual Machine (JVM) – das Verwenden eigener Klassenlader unterstützt. Über das Reflection-API können Annotationen (bei entsprechender Gültigkeit auch die bereits geladener Klassen) abgefragt werden.

Ein typischer Fall dieser Art der Modifikation ist die *Context and Dependency Injection* (CDI) [4] genannte Technologie der *Java Enterprise Edition* (JEE). Leicht vereinfacht lässt sich das wie folgt darstellen. Durch CDI ist es u. a. möglich, ein Feld in der Form `@Inject private Context context;` zu deklarieren. Eine explizite Wertzuweisung wird nicht gebraucht, weil diese durch die Ausführungsumgebung (Container) beim Initialisieren erfolgt. Bedingung ist dabei lediglich, dass der Typ Context ein Interface ist und dass es unter allen ladbaren genau eine Klasse gibt, die dieses Interface implementiert. Richtig angewandt, ist auf diese Weise eine quasi automatische Anpassung an unterschiedliche Umgebungen möglich. Je nachdem, ob es um Test, Abnahme oder Produktion geht, immer werden die richtigen Objekte injiziert.

Tatsächlich ist es in der Praxis nicht ganz so einfach. Eine ähnliche Technik wird beispielsweise auch verwendet, um sogenannte Interzeptoren, das sind Methoden, die vor und nach einem Methodenaufruf aufgerufen werden, zu definieren. Da in diesem Fall die Interzeptoren, aber nicht die Stellen des Einfügens annotiert werden, ergibt sich das eigentlich unlösbare Problem, in welcher Reihenfolge diese bei einem Aufeinandertreffen ausgeführt werden sollen. JEE 7 führt aus diesem Grund mit der Möglichkeit, numerische Prioritäten zu vergeben, dann doch wieder indirekte Anhängigkeiten ein [5].

Als besonders problematisch an der Bytecode-Modifikation ist zu sehen, dass Fehler größtenteils erst zur Laufzeit, und zwar frühestens beim ersten Start erkannt werden können.

Schlussfolgerungen

Die vielfältige Verwendung von Annotationen zeigt zuerst einmal, dass es ein starkes Bedürfnis an Technologien gibt, die helfen, die Programmierung einfacher und schneller zu machen. Die Frage ist jedoch, ob Annotationen ein angemessenes Mittel sind. Es gibt gute Gründe, die dagegen sprechen.

Problematisches

Die syntaktische Form einer Annotation lässt keine Rückschlüsse darauf zu, welchem der aufgeführten Verarbeitungsmodelle sie angehört. Damit ist aus dem Code allein nicht ersichtlich, ob er die Ausfüh-

rung überhaupt vollständig beschreibt oder noch durch Generierung oder Modifizierung ergänzt wird.

Annotationen haben keine abgegrenzte Semantik. De facto führen sie eine Meta-Ebene in die Programmiersprache ein. Jeder Annotationstyp definiert eine eigene kleine deklarative Sprache. Um diese verstehen zu können, muss das jeweilige zugrunde liegende semantische Modell bekannt sein. Überdies sind Abhängigkeiten zwischen Annotationstypen stets implizit. Das Lesen annotierten Codes wird also nicht nur durch den zusätzlichen Text erschwert, sondern auch dadurch, dass zum Verstehen wiederholt zwischen den verschiedenen semantischen Modellen gewechselt werden muss.

Durch die Verwendung von Annotationen können Grundregeln der objektorientierten Programmierung außer Kraft gesetzt werden. Sie stehen einerseits außerhalb der üblichen Vererbungsregeln – die Vererbung kann bei der Definition eines Annotationstyps explizit erlaubt oder unterdrückt werden – andererseits ist es den Prozessoren und Klassenladern möglich, unabhängig davon die Klassenhierarchie zu durchlaufen und Annotationen von Oberklassen zu berücksichtigen. Bei einer als nicht vererbend definierten Annotation kann es also vorkommen, dass ein Objekt einer korrekt abgeleiteten Subklasse in einem Fall ein Objekt der Oberklasse ohne Probleme ersetzen kann, in einem anderen Fall aber nicht. Bei der erwähnten CDI-Technologie kann allein dadurch, dass zu einer Anwendung eine (zweite) Klasse hinzugefügt wird, die ein für die Injektion benutztes Interface implementiert, ein Laufzeitfehler verursacht werden.

Bewertung

Annotationen sind von der Softwaretechnologie wenig beachtet worden. Lange hat man sich von Definitionen wie der aus der Java-Sprachbeschreibung blenden lassen: „*An* annotation *is a marker which associates information with a program construct, but has no effect at run time*“ [6]. Währenddessen ist in der Praxis ausgelotet worden, was technisch realisierbar ist. Die Sprengkraft, die in diesem *Marker* steckt, wird trotzdem wohl immer noch unterschätzt. Wenn es darum geht, die Auswirkungen auf die Wartbarkeit von Programmen zu bewerten, werden fast ausschließlich solche Anwendungen betrachtet, die Annotationen im Wortsinn sind, also ergänzende Informationen, z. B. [2]. Demgegenüber werden die Fälle, in denen der ausgeführte Code wirklich beeinflusst wird, vorrangig unter dem Gesichtspunkt der Entwicklungseffizienz diskutiert.

Schon mit den derzeit verfügbaren Tools ist es wie oben gezeigt, möglich, wichtige Prinzipien des objektorientierten Softwaredesigns zu unterlaufen. Das liegt unter Anderem daran, dass es kein konsistentes Konzept für den Umgang mit Vererbung gibt.

Durch Annotationen erhält man eine zusätzliche Dimension der Schnittstellenbeschreibung, die faktisch keinen formal prüfbaren Regeln unterliegt. Wenn beispielsweise eine Klasse ein bestimmtes Interface nicht implementiert, führt das schlimmstenfalls zu einem Cast-Fehler. Oft wird dieser Fehler aber schon dem Compiler auffallen. Wenn hingegen eine Annotation vergessen wird, fehlt unter Umständen zwar eine Funktionalität, aber ohne dass irgendwo auch nur ein Hinweis auf diesen Fehler auftauchen muss.

Die Entwicklung der Technologien rings um Annotationen ist vor allem vom technisch Machbaren getrieben worden. Konzeptionelle Gesichtspunkte wurden vernachlässigt. So gibt es keine Mittel zur Abstraktion. Ebenso fehlt es an Vorstellungen, wie die Komposition verschiedener Spracherweiterungen vonstatten gehen soll. Die auftretenden Unklarheiten zeigen sich schon bei der Suche nach einer Begrifflichkeit, um sie zutreffend zu beschreiben. Wie soll man eine Gruppe von Annotationstypen einschließlich Annotationsprozessor und Laufzeitbibliothek nennen? Bilden sie zusammen eine Spracherweiterung oder zuweilen auch gleich mehrere? Definieren die Annotationstypen die Erweiterung, die dann von Prozessoren und Bibliotheken lediglich implementiert wird? Kann oder sollte ein Annotationstyp zu mehr als einer Erweiterungen gehören?

Wie sollen Abhängigkeiten zwischen ansonsten unabhängigen Spracherweiterungen berücksichtigt werden? Die Verwendung von Prioritäten (JEE 7) verschleiert die Beziehungen und ist längerfristig eher kontraproduktiv.

Annotationen erleichtern die Meta-Programmierung. Sie bieten aber keinerlei syntaktische Hilfen, die Vermischung unterschiedlicher Meta-Ebenen zu vermeiden oder diese Ebenen auch nur sauber abzugrenzen. Vermischung von Abstraktionsebenen gilt aber zu Recht als Merkmal schlechter Codequalität.

Das Grundproblem besteht einfach darin, dass Annotationen als Bestandteil der Programmiersprache der falsche, weil zu tief liegende Ansatzpunkt für die meisten der aufgeführten Techniken sind. Notwendig ist eine umfassendere Sprachschicht, die vielleicht als *programmatische Modellierung* bezeichnet werden könnte, in der dann die unterschiedlichen Artefakte spezifiziert/programmiert werden. In [12] wird dieser Ansatz als Menge kombinierbarer Sprachmodule genauer ausgeführt.

Was wir derzeit erleben, ist ein groß angelegter Versuch, mittels Annotationen fortgeschrittenere Methoden (Codegenerierung, Meta- bzw. aspektorientierte Programmierung usw.) zu verwenden, ohne die erforderliche Infrastruktur, diese angemessen kontrollieren zu können.

Schlussbemerkung

Die Kritik richtet sich nicht gegen das, *was* mit Annotationen gemacht wird. Sie richtet sich dagegen, dass dies alles *mit* Annotationen gemacht wird. Der Form der Anmerkung, ganz gleich ob als Attribut oder Annotation, kann einfach zu viel Bedeutung aufgeladen werden. Es fehlt an Struktur, diese semantische Vielfalt handhabbar und verständlich zu gestalten. Längerfristig werden sich daraus für die Wartung von Software große Probleme ergeben. In dieser Hinsicht gibt es Parallelen zum *Goto*. Seinerzeit ging es allerdings nur um die schlechte Erkennbarkeit des Kontrollflusses durch sogenannten *Spaghetticode*. Annotationen haben das Potenzial, die Les- und Nachvollziehbarkeit von Code in weitaus größerem Maße zu beeinträchtigen.

Literatur

1. Gabriel RP (1993) The end of history and the last programming language. JOOP 6:90–94
2. http://blog.bosch-si.com/how-java-annotations-might-help-to-bring-world-peace/, letzter Zugriff: 22.5.2014
3. http://docs.jboss.org/hibernate/orm/4.3/, letzter Zugriff: 22.5.2014
4. http://docs.oracle.com/javaee/7/tutorial/doc/cdi-basic.htm, letzter Zugriff: 22.5.2014
5. http://docs.oracle.com/javaee/7/api/, letzter Zugriff: 22.5.2014
6. http://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.6, letzter Zugriff: 22.5.2014
7. http://download.oracle.com/otndocs/jcp/ejb-2.1-fr-spec-oth-JSpec/, letzter Zugriff: 8.8.2014
8. http://download.oracle.com/otndocs/jcp/ejb-3_0-fr-eval-oth-JSpec/, letzter Zugriff: 8.8.2014
9. http://empire-db.apache.org/empiredb/hibernate.htm, letzter Zugriff: 8.8.2014
10. http://openjdk.java.net/groups/compiler/doc/compilation-overview/, letzter Zugriff: 22.5.2014
11. http://projectlombok.org/, letzter Zugriff: 22.5.2014
12. Volter M (2011) From programming to modeling – and back again. IEEE Softw 28(6):20–25
13. Wirth N, Gutknecht J (1992) Project Oberon. Addison-Wesley, Reading, Mass