Wiederverwendbarkeit und Softwareentwicklung − Probleme, Lösungsansätze und Bibliographie

Jürgen Börstler
RWTH Aachen, Lehrstuhl für Informatik III, Ahornstraße 55,
D−5100 Aachen

Zusammenfassung

Seit dem ITT Workshop über Wiederverwendbarkeit 1983 ([23]) häufen sich die Veröffentlichungen über dieses Thema der Softwaretechnik. Der vorliegende Beitrag gibt einen ausführlichen Literaturüberblick über das Themengebiet. Nach der Erläuterung der Motivation zur Wiederverwendung im Softwareentwicklungsprozeß wendet sich der Beitrag den Objekten der Wiederverwendbarkeit und ihren Eigenschaften zu. Bei der Vorstellung der Ansätze aus der Literatur werden jeweils besonders interessante Projekte kurz vorgestellt. Die Ansätze werden miteinander verglichen und es wird eine Klassifizierung versucht. Die Integration solcher Ansätze in eine Softwareentwicklungsumgebung wird in einem gesondertem Abschnitt genauer untersucht. Der Beitrag schließt mit einer umfangreichen Bibliographie, ohne Anspruch auf Vollständigkeit.

1. Einleitung

Seit es programmierbare Computer gibt, ist Wiederverwendbarkeit ein erstrebenswertes Ziel gewesen. Leider ist man über das Stadium der Installation von *mathematischen Unterprogrammbibliotheken* bis Ende der 70er Jahre nicht hinausgekommen ([79]).

Die sich zu dieser Zeit entwickelnde *Softwarekrise* hat die Bestrebungen nach mehr Wiederverwendbarkeit neu belebt, da man sich davon *signifikante Produktivitätssteigerungen* erhoffte. Um die kommende Krise zu überwinden, postulierte McIllroy bereits 1968 ([174])[1] die Notwendigkeit einer *industriellen Softwarekomponentenfertigung*, wie sie mittlerweile für Hardwarekomponenten nicht mehr wegzudenken ist. Warum seine Vision bis heute, von einigen Ausnahmen abgesehen, nicht in Erfüllung ging, hat sehr viele Gründe.

Einige dieser Gründe wurden von Dijkstra auf der "NATO conference on SE Techniques '69" ([38]) treffend beschrieben und haben immer noch Gültigkeit:

"What is actually happening, I am afraid, is that we tell each other and ourselves that software engineering techniques should be improved considerably, becausethere is a crisis. But there are a few boundary conditions which apparently have to

[1] *Dieser Artikel ist die erste Veröffentlichung die sich mit dem Thema Wiederverwendbarkeit beschäftigt ([79]).*

be satisfied. I will list them for you:
1. We may not change our thinking habits.
2. We may not change our programming tools.
3. We may not change our hardware.
4. We may not change our tasks.
5. We may not change the organizational set−up in which the work has to be done."

Dieser Beitrag soll einen Einblick in die verschiedenen Ansätze geben, die in der Literatur vorgeschlagen werden McIllroy's Ideen zu verwirklichen.

2. Warum Wiederverwendbarkeit unterstützen?

Heute geht der Trend der Softwareentwicklung hin zu speziellen, angepaßten Lösungen, die alle Anwenderwünsche berücksichtigen. So ist es nicht verwunderlich, daß sehr häufig *Anwendungen* entwickelt werden, die *nur in Details* voneinander *abweichen*.

Da diese Ähnlichkeiten nicht immer offensichtlich sind, oder die Entwicklungen räumlich und/ oder zeitlich weit auseinanderliegen, wird in vielen Fällen "das Rad neu erfunden", anstatt sich auf bestehende Lösungen abzustützen.

Studien, über deren Repräsentativität man streiten mag, zeigen, daß Software Wiederverwendung nicht nur sinnvoll sein kann, sondern es auch *vorzeigbare Erfolge* gibt. Solche Erfolge haben gerade die Unternehmen vorzuweisen, die sich frühzeitig mit diesem Thema beschäftigt haben. In der Fuchu Software Factory (Toshiba) arbeitet man bereits seit 1977 mit Software Wiederverwendbarkeit und benutzt 32% bis 48% wiederverwendbare Komponenten über den gesamten Software Lebenszyklus ([154]). Diese Zahlen werden auch von einer Untersuchung bei der NASA bestätigt ([219]), wo bei 25 Projekten Raten zwischen 0% und 82% erziehlt wurden (im Mittel 32%). Von der Rayetheon Company wird über Anwendungen berichtet, die im Schnitt aus 60% wiederverwendeten Codes bestehen ([129]). Diese Raten sind allerdings nur in einer Abteilung erziehlt worden, die nur Anwendungen aus einem speziellen Domain erstellt. Für das Banken− und Versicherungswesen werden gar Raten von 75% prognostiziert ([115]).

T. C. Jones schätzt in [117], daß nur etwa 15% des gesamten 1983 geschriebenen Codes tatsächlich "neu" war, und etwa 85% potentiell wiederverwendbar gewesen wären. In [114] wird geschätzt, daß sich jedes größere Anwendungs− bzw. Systemprogramm zu 50%−80% aus wiederverwendbaren Komponenten erstellen läßt. Nach Standish (in [23]) nimmt Software Wiederverwendbarkeit eine Schlüsselrolle unter den Maßnahmen zur Steigerung der Softwareproduktivität ein.[2]

---

[2] *"... since economic analysis indicates that the cost of software is an exponential function of software size, halving the size of the software which must be built much more than halves the cost of building it. Software reuse thus becomes a keystone in many current efforts to improve productivity ..."*

Inwiefern sich dies auf die Gesamtproduktivität auswirkt wird in [114] genauer untersucht. Von drei realen Anwendungen (10000 Zeilen COBOL−Code) mit unterschiedlichen Raten der Wiederverwendung wurde auf eine vierte (75%) extrapoliert, siehe Fig. 1a). Zu ähnlichen Ergebnissen kommt auch eine Untersuchung in [141] (20000−30000 Zeilen Objective−C), siehe Fig. 1b). In [70] wird eine Gleichung aufgestellt, welche die Produktivitätssteigerung in Abhängigkeit von der Wiederverwendungsrate und von den relativen Kosten der Wiederverwendung berechnet. Diese Formel berücksichtigt nicht den positiven Effekt von Wiederverwendung auf Test− und Wartungsphase. In Fig. 1c) sind Ergebnisse, welche nach dieser Formel errechnet wurden zusammengefaßt.

a)

[114]	Wiederverwendungsrate in % 0%	25%	50%	75%
Gesamtzeit in PM	81.5	45	32	15
# Mitarbeiter	8	6	5	4
Kosten je Zeile	40.75	22.5	16	7.5
Zeilen pro PM	165	263	370	833
Ersparnis	0%	45%	61%	82%

PM = Personenmonate

c)

[141]	Wiederverwendungsrate in % 0%	10%	30%	50%	80%
Gesamtzeit in PM	200	176	131	89	33
Ersparnis	0%	12%	34%	56%	84%

b)

[70]	Wiederverwendungsrate in % 10%	25%	30%	50%	75%	80%
$k_w = 0.1$	9.9	29	37	81.8	208	257
$k_w = 0.2$	8.7	25	31.6	66.7	150	178
$k_w = 0.3$	7.5	21.2	26.6	53.8	111	127
$k_w = 0.4$	6.4	17.6	22	42.9	81.8	92.3
$k_w = 0.5$	5.3	14.3	17.6	33.3	60	66.7

Produktivitätssteigerung in % in Abhängigkeit von den relativen Kosten der Wiederverwendung k_w

Fig. 1: Auswirkungen von Software Wiederverwendung auf Softwareproduktivität

Nach diesen äußerst optimistisch klingenden Fakten und Prognosen fragt man sich zurecht, warum die Wiederverwendbarkeit von Software nicht schon weitere Verbreitung erfahren hat. Das könnte verschiedene Ursachen haben: a) finanzielle, b) psychologische und c) technische.

Bei den *finanziellen Ursachen* sind vor allem die hohen Anlaufkosten zu erwähnen, welche sich erst im Laufe der Zeit amortisieren. Hierunter summieren sich längere Entwicklungszeiten für solche Komponenten, welche allgemein auf etwa 25% Mehraufwand geschätzt werden. In [241] wird dieser Mehraufwand sogar auf 30−200% geschätzt. Desweiteren braucht man spezielles Personal und brauchbare Werkzeuge zur Erstellung, Wartung und Verwaltung wiederverwendbarer Software. Für die Softwarekomponenten müssen geeignete Beschreibungsmittel entwickelt werden, damit man geeignete ”Angebote” und ”Nachfragen” formulieren kann. Ebenso müssen die einzelnen Entwickler auch lernen mit diesen Komponenten umzugehen. Zu guter Letzt muß möglicherweise das Unterneh-

men umstrukturiert werden um diese Konzepte zu verwirklichen (Mitarbeiterschulung, abteilungsübergreifende Zusammenarbeit, neue Abteilungen, neue Produkte, ...). Dies alles summiert sich nach Angaben von C. Jones aus dem Jahr 1986 ([114]) für größere Unternehmen der Banken− oder Versicherungsbranche leicht auf 1 Mio. US$.

Die *psychologischen Probleme* sind bereits sehr früh von Dijkstra (siehe Einleitung) treffend beschrieben worden. Eine große Rolle spielt hier auch das sogenannte "not invented here"−Syndrom. In [268] kommen die Autoren nach einem größeren Experiment zu dem Schluß, daß in Wiederverwendbarkeit ungeübte Softwareentwickler deren Nutzen sehr schlecht abschätzen können und daher auch einige Schwierigkeiten mit dieser Technik haben. Im weiteren soll nicht näher auf diesen Punkt eingegangen werden, sondern wir wollen uns in den nächsten Kapiteln den *technischen Problemen* und den Ansätzen zu deren Lösung widmen.

3. Was kann wiederverwendet werden?

Freeman ([77]) definiert die *Objekte* der *Wiederverwendbarkeit* als alle Informationen, die ein Entwickler im Softwareerstellungsprozeß möglicherweise gebrauchen kann. Diese Definition schließt auch die Programmiererfahrung und diffuses Wissen mit ein. Solche Objekte können Softwaredokumente, Wissen und Erfahrung sein. Ein Softwaredokument (im folgenden wird nur noch von Dokumenten gesprochen) ist (Teil einer) Anforderungsdefinition, Entwurfsspezifikation, Programmcode, techn. Dokumentation usw., also ein Teil− oder Zwischenergebnis, welches im Softwareentwicklungsprozeß anfällt. Diese Dokumente können auch generischer Natur sein bzw. Teile enthalten, welche noch genau zu spezifizieren sind. Wir wollen im folgenden unsere Betrachtungen auf die Wiederverwendung von Dokumenten beschränken.

Die Wiederverwendung von Dokumenten in Form von Programmcode beinhaltet die Objekte, mit der sich die bisherige Literatur am intensivsten befaßt hat. Darunter fallen:

- Programmfragmente
  Dies können sowohl sehr einfache, wie z.B. von syntaxgesteuerten Editoren erzeugte Fragmente sein, oder immer wiederkehrende Codestücke, welche erst in einer konkreten Umgebung einen Sinn ergeben (siehe z.B. in [252]).
- Muster/ Schablonen (Frames)
  Sie dienen als Vorlagen zur manuellen Erzeugung von Programmen oder Teilen davon.
- Prozeduren/ Funktionen
  Hierunter fallen anwendungsbezogene Kollektionen von Prozeduren/ Funktionen, Standardprozeduren/ −funktionen und Makros.
- Moduln
  Unter einem Modul soll ein Programmbaustein verstanden werden, welcher

über eine Schnittstelle mit seiner Außenwelt kommuniziert und eine bestimmte Entwurfsentscheidung verkapselt ([136]).
- Teilsysteme
  Sie fassen logisch zusammengehörige Moduln zu größeren Systemkomponenten zusammen und bilden aus den einzelnen Modulschnittstellen eine neue Teilsystemschnittstelle ([136]), wodurch eine weitere Abstraktionsebene entsteht.

Im allgemeinen kann man sagen, daß der *Nutzen* der *Wiederverwendung* umso höher ist, je mehr wiederverwendbare Dokumente man findet, je größer die wiederverwendeten Dokumente sind bzw. je früher man mit der Wiederverwendung im Lebenszyklus ansetzt. Allerdings nimmt, je größer das Dokument ist bzw. je früher es im Lebenszyklus angesiedelt ist, die Wahrscheinlichkeit ab, daß es den Benutzeranforderungen genau entspricht.

Unter den Punkt "Wissen und Erfahrung" fallen im wesentlichen alle "Umstände", die Programmierer/innen dazu befähigen, Entscheidungen im Softwareentwicklungsprozeß zu treffen. Dazu gehört Wissen über die Prinzipien des Software Engineering, wie z.B. Modularisierung, Abstraktionsmechanismen, Information Hiding, usw., sowie Erfahrungen über deren sinnvolle Anwendung. Ebenfalls unter diesen Punkt fallen Kenntnisse über das Verhalten von Algorithmen unter verschiedenen Umgebungsbedingungen wie z.B., daß die Geschwindigkeit von Sortieralgorithmen sehr stark von der Vorsortierung der Algorithmen abhängen kann.

Für alle Dokumente, die man wiederverwenden will lassen sich einige *grundsätzlichen Fragen* formulieren ([31]):
- welche (Teil−) Dokumente sind Kandidaten zur Wiederverwendung?
- wie beschreibt man solche Dokumente?
- wie speichert man solche Dokumente ab?
- wie findet man solche Dokumente wieder auf?
- wie paßt man solche Dokumente an die eigenen, detaillierten Anforderungen an?
- wie integriert man solche, gegebenenfalls angepaßten Dokumente in eine vorliegende Umgebung?
- wie komponiert/ konfiguriert man solche Dokumente?

Daraus ergeben sich einige wichtige *Forderungen*, die solche *wiederverwendbaren Dokumente* erfüllen müssen:
- *Verständlichkeit*, d.h. Entwicklung einfacher Beschreibungen solcher Dokumente und vorausgehend geeignete Notationen für Syntax, Semantik und Pragmatik von wiederverwendbaren Dokumenten.
- *Definiertheit*, d.h. das Verhalten der Dokumente muß für alle möglichen Situationen vorhersehbar sein.

- *Übertragbarkeit*, d.h. die Dokumente sollen auch in anderen als den Entwicklungskontexten anwendbar sein. Diese Forderung hängt eng mit der Forderung nach Verständlichkeit zusammen.
- *Zugreifbarkeit*, d.h. die Möglichkeit, wiederverwendbare Dokumente mit Hilfe einer Beschreibung oder eines Browsing−Mechanismus' aufzufinden. Auch diese Forderung hängt eng mit der Forderung nach Verständlichkeit zusammen.
- *Hohe Kohäsion*, d.h. die Dokumente müssen aus zusammengehörigen Komponenten bestehen.
- *Lose Kopplung*, d.h. die Dokumente müssen weitgehend unabhängig voneinander sein.

4. Wie kann Wiederverwendbarkeit unterstützt werden?

Eine erste Systematisierung der verschiedenen Ansätze zur Unterstützung von Wiederverwendbarkeit findet sich in [26]. Dort werden die *Ansätze* zunächst in *zwei Gruppen* unterteilt, welche sich durch die *Natur* der von ihnen betrachteten, wiederverwendbaren *Dokumente* unterscheidet. Diese Dokumente sind entweder "Reusable Building Blocks" oder "Reusable Patterns".

Ein *Building Block* wird dabei als atomares Dokument mit (relativ) stabiler Internstruktur verstanden. Die Ansätze, die mit Building Blocks arbeiten, werden in zwei Gruppen mit verschiedenen Schwerpunkten unterteilt. Ein Ansatz legt den Schwerpunkt auf den Aufbau von *Dokumentenbibliotheken*, der andere auf *Mechanismen* zur *Komposition* von Dokumenten.

Patterns sind im Gegensatz zu Building Blocks "aktive" Dokumente, durch deren Aktivierung Programme oder Teile davon erzeugt werden. Hierzu werden drei Ansätze mit verschiedenen Schwerpunkten unterschieden. Einen Ansatz bilden *sprachenbasierte Systeme*. Die weiteren Ansätze bilden *Generatoren* und *Transformationssysteme*.

In [223] wird eine Unterscheidung in "black−box−reuse" (Wiederverwendung ungeänderter Dokumente) und "white−box−reuse" (Wiederverwendung bearbeiteter Dokumente) vorgenommen. Mit der Frage, ob und wie man Dokumente in Form von Programmcode, zur "black−box−reuse" auf ein *Chip−Design* abbildet, beschäftigt sich Hartmann [103].

Eine andere *Klassifizierung* findet sich in [254]. Dort wird Wiederverwendbarkeit vom Standpunkt *theoretischer Konzepte* angegangen und es wird unterschieden nach:

- *Reduzierbarkeit:* Läßt sich ein Problem B auf ein Problem A reduzieren, kann man die zur Lösung von A entwickelten Techniken auch zur Lösung von B heranziehen, z.B. bei der Rückführung der Analysealgorithmen für kontextfreie Grammatiken auf die Matrixmultiplikation.

- *Äquivalenzklassenbildung:* Gehört ein Problem A zu einer Äquivalenzklasse K von Problemen, können Lösungen für Probleme aus K zur Lösung von A dienen.
- *Modellbildung:* Modelle erleichtern oder ermöglichen die Beschreibung komplexer Sachverhalte, z.B. erlauben es semantische Modelle die Äquivalenz von Programmen zu zeigen.
- *Abstraktion/ Spezialisierung:* Abstraktion erlaubt die Beschreibung von Gemeinsamkeiten einer Problemklasse durch das Auslassen von problemspezifischen Details, z.B. bei Generizität oder Vererbung.
- *Invarianten:* Eine Invariante einer Funktion muß für alle Programme gelten, die diese Funktion realisieren und kann zur Verifikation dieser Programme (wieder−) verwendet werden.
- *Mustererkennung:* Dieser Ansatz erleichtert das Auffinden von Gemeinsamkeiten.
- *Regularität:* Diese Eigenschaft erlaubt die konzise Darstellung von Sachverhalten, z.B. reguläre Ausdrücke zur Beschreibung regulärer Mengen.

In [132] werden die *Abstraktionsebene*, auf welcher die Wiederverwendung stattfindet, die *Anpassungsmethoden* von Dokumenten an wechselnde Einsatzumgebungen von wiederverwendbaren Dokumenten und die *Bedingungen* zur Wiederverwendbarkeit als drei Kriterien zur Klassifizierung von Ansätze beschrieben.

Hall ([100]) unterscheidet Wiederverwendung von *Ideen*, *vertikale/ horizontale Wiederverwendung*, sowie *totale Wiederverwendung*. Dabei bezeichnet vertikale Wiederverwendung die Verwendung vorhandener Dokumente in höheren Abstraktionsebenen und horizontale Wiederverwendung die Komposition von Dokumenten innerhalb einer Abstraktionsebene. Eine ausführliche Auflistung verschiedener Ansätze findet sich schließlich in [272].

Für die konkreten Ansätze zur Unterstützung von Wiederverwendbarkeit soll folgendes *Klassifikationsschema* dienen, welches wir im nächsten Abschnitt zum Ausgangspunkt unserer Betrachtung machen. Dieses Klassifikationsschema ist keineswegs orthogonal, wie auch im Forschungsgebiet Wiederverwendbarkeit bisher kein solches existiert. Neben dieser Abhängigkeit der einzelnen Aspekte kommt hinzu, daß konkrete Ansätze der Wiederverwendbarkeit zumeist Mischformen der aufgeführten Bereiche darstellen:

1. Standards und Konventionen
   (a) Datenaustauschformate
   (b) Standardschnittstellen/ −routinen
2. Customizing
   (a) manuell
   (b) maschinell
3. Organisationsmechanismen

   (a) Dateisysteme
   (b) Klassifikationsschemata
   (c) Integration von Dokumentbeziehungen
4. Kompositionsmechanismen
   (a) Pipes
   (b) Schnittstellenkontrolle
   (c) Message Passing
   (d) Vererbung
5. Metaprogrammierung
   (a) Generatoren
   (b) Very High Level Languages
   (c) End User Programming
6. wissensbasierte Programmierung
   (a) Transformationssysteme
   (b) intelligente Assistent/inn/en
7. Integrierte Methoden und Werkzeuge

Die Ansätze Prototyping und automatische Programmierung tauchen aus verschiedenen Gründen nicht in obigem Schema auf.

Prototyping ist die Erstellung eines ersten Entwurfs für ein Programm anhand einer Anforderungsspezifikation. Diese Spezifikation erfolgt zumeist in einer problemspezifischen, sehr hohen Programmiersprache oder als Eingabe in einen Generator, womit Prototyping unter die Punkte 5a) bzw. 5b) fällt. Andere Prototypingansätze erfolgen durch Komposition von vorhandenen Systemteilen, was unter Punkt 4) abgehandelt ist.

Für allgemeine Probleme ist die (voll−) *automatische Programmierung* immer noch Zukunftsvision ([206]). Bisher ist automatische Programmierung nur in sehr eingeschränkten Anwendungsbereichen, oder mit intensiver Benutzerführung möglich (z.B. Generatoren bzw. interaktive Transformationssysteme), was bereits die Punkte 5) und 6) beinhalten.

5. Die Ansätze im einzelnen

5.1. Standards und Konventionen

Standards und Konventionen sind für die Wiederverwendbarkeit enorm wichtig. So ist z.B. jede Art von Datenaustausch nur sinnvoll ist, wenn diese Daten auch verstanden werden können, also dementsprechende *Datenaustauschformate* definiert sind. Besteht mit Hilfe von *Standardschnittstellen* erst einmal Einigung über alle Details der Funktionalität (nicht der zugehörigen Implementierung), braucht sich bei der (Wieder−) Verwendung kein Anwender mehr um die Implementierung eines solchen Systemteils zu kümmern.

Im Bereich der Bibliotheksansätze scheinen sich gerade jetzt Standards zur Bezeichnung und zum Aufbau von Bibliotheksmoduln durchzusetzen (Klassifika-

tion). Diese von Booch für Ada eingeführten "Regeln" ([30]) werden nun auch von anderen Autoren, z.B. für Modula−2 (wieder−)verwendet ([276]).

Diese Punkte werden in den Klassifikationsschemata in [26] und [254] nicht berücksichtigt.

5.2. Customizing

Unter Customizing versteht man Techniken, bei denen Dokumente zur Verfügung gestellt werden, welche zur Wiederverwendung noch bearbeitet werden müssen. Dieses Vorgehen ist ein erster Abstraktionsschritt, um Dokumente in verschiedenen Kontexten wiederverwendbar zu machen. Die kontextspezifischen Teile werden speziell markiert, um die Anpassung zu erleichtern. Diese Methoden werden in der Literatur auch als parametrisierbare Programmierung ([91]), Frame Methode ([16]) oder Schablonentechnik ([70]) bezeichnet. Die Bearbeitung kann z.B. mit Unterstützung eines speziellen Editors *manuell* erfolgen ("fill in slots").

Maschinelle Unterstützung gibt es z.B. in den Systemen PARIS ([124]) und BB/LX ([132]), sowie in Programmiersprachen, die Generizität unterstützen, wie z.B. Ada™ ([86]). In PARIS werden sogenannte *Schemata* (Schablonen) u.a. zusammen mit *Bedingungen* abgespeichert, die an die noch zu bearbeitenden Teile gestellt werden. Sind diese Teile dann fertig bearbeitet, wird (automatisch) bewiesen, ob die Bedingungen erfüllt sind oder nicht. BB/LX ist eine Spracherweiterung für PL/S, welche Ada−ähnliche Konzepte (Pakete, Generizität, Ausnahmebehandlung) beinhaltet. Das "Herz" von BB/LX ist eine Bibliothek generischer Building Blocks. Diese Building Blocks sind abgeschlossene modulare Einheiten, welche Datentypen oder Funktionen verkapseln. Zur Übersetzungszeit werden diese generischen Building Blocks durch BB/LX Makros instantiiert. Es erfolgt eine Generierung von Code, welcher, nach einer Prüfung von Syntax und statischer Semantik, in−line in den umgebenden PL/S Code expandiert wird.

Die hier aufgeführten Ansätze arbeiten mit Building Blocks im Sinne von [26], jedoch finden dort die Customizing−Ansätze keine besondere Beachtung. Die Dokumente, die bei den beschriebenen Ansätzen bearbeitet werden, kann man sich als Modelle vorstellen, welche durch Auslassung problemspezifischer Details erstellt wurden. Damit fällt Customizing in die Kategorien Modellbildung und Abstraktion/ Spezialisierung des Schemas in [254].

5.3. Organisationsmechanismen

Die einfachste Organisationsform für Dokumente ist deren Ablage in einem *Dateisystem*. Diese Methode ist leicht anzuwenden, hat aber einige Nachteile. Die so angelegte Bibliothek wird sehr schnell unübersichtlich, da außer der Hierarchisierung und der Namensgebung von Dateien keine weiteren Organisationsmöglichkeiten vorhanden sind. Das bedeutet, daß man die Inhalte von Dateien zumindest ungefähr an ihren Namen erkennen muß, um in angemessener Zeit ein wiederverwendbares Dokument aufzufinden. Dabei können verwandte Doku-

mente in einer Datei abgelegt sein.

Den Suchvorgang kann man verbessern, indem man jedes Dokument mit einem Satz von Attributen klassifiziert. Diese Klassifizierung sollte nicht hierarchisch sein wie z.B. der Computing−Reviews Index[3]. Einen guten Vorschlag machen [193] mit ihrem sogenannten facettierten Klassifikationsschema.

Dazu werden sechs Facetten festgelegt. Jede dieser Facetten besteht aus einer Menge benutzerdefinierter Attribute, welche als Knoten im sogenannten *conceptual distance graph* abgelegt sind. Die Attribute der Facetten beschreiben Funktionalität und Anwendungsgebiet eines Dokumentes, vgl. Fig. 2. Die Kanten des

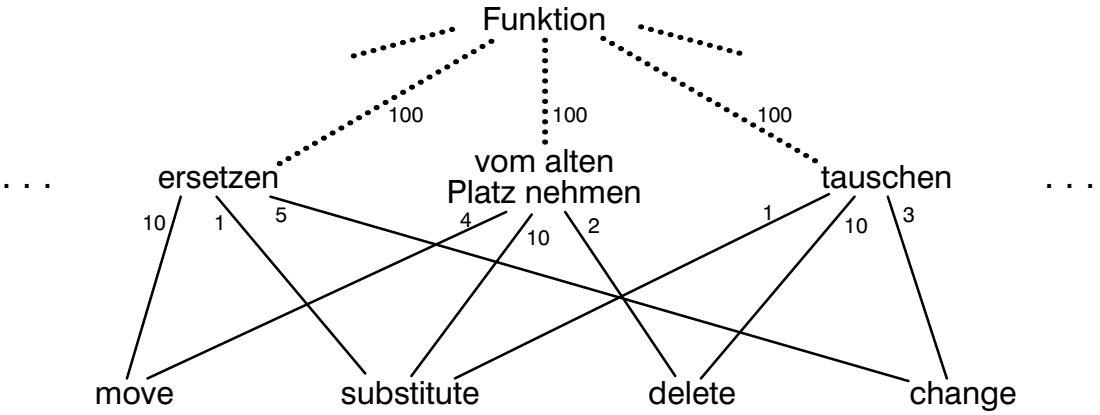

Fig. 2: Ausschnitt aus dem conceptual distance graph der Facette Funktion

conceptual distance graph sind mit einer Bewertung für den semantischen Abstand zweier Attribute versehen. So sind z.B. in der Facette Funktion die Attribute change und substitute näher (conceptual distance: 4) als change und delete (13). Zusätzlich zu diesem Graphen gibt es für jedes Attribut einen Wörterbucheintrag um synonyme Verwendung von Begriffen zu berücksichtigen.

Ein Zugriff auf die Komponenten erfolgt durch Angabe eines Sechstupels von Begriffen (je Facette ein Begriff), wobei auch "wild−cards" erlaubt sind. Ergebnis einer solchen Anfrage ist eine Liste von Kandidaten, welche nach der conceptual distance zur Originalanfrage geordnet ist. Jeder Kandidat ist dabei mit einem Satz von Attributen wie z.B. Programmgröße und benutzte Programmiersprache versehen, um eine Auswahl zu erleichtern.

Dieser Ansatz wird in [148] noch erweitert, so daß sich das Klassifikationsschema mit jedem neu in die Bibliothek aufgenommenen Dokument dynamisch anpaßt. Andere Klassifikationsschemata berücksichtigen auch Beziehungen zwischen Dokumenten. Diese werden genutzt, um auf Dokumente zuzugreifen, die zum gerade bearbeiteten Dokument in Beziehung stehen ([44, 69, 159, 173, 199,

[3] *Die Suche nach Literatur über Testgenerierung endet im Unterbaum 'Hardware' des Computing Review Index in sieben verschiedenen Blättern, die nur über die gemeinsame Wurzel in Beziehung stehen. Die Suche nach Literatur über Testdatengenerierung wäre dort ohne Erfolg, da kein Blatt mit dieser Markierung existiert.*

235, 267, 269]).

Um das Auffinden (klassifizierter) Dokumente zu erleichtern, werden auch *Information−Retrieval*−Methoden ([53, 74]) oder *Expertensysteme* ([34]) vorgeschlagen. Andere Ansätze beschäftigen sich mit der Modulauswahl durch *Pattern Matching* ([60, 233]), *prädikatenlogischen Ausdrücken* ([273]) und *algebraischen Spezifikationen* ([87, 258]). Die beiden letztgenannten Ansätze haben den Nachteil, nur "Volltreffer" als Ergebnisse von Anfragen zurückzuliefern, was deren praktische Einsatzmöglichkeiten sehr stark einschränkt, da man in der Regel nicht erwarten kann, in der Bibliothek genau das gesuchte Dokument zu finden.

In [26] werden diese Ansätze in der Kategorie Dokumentenbibliotheken zusammengefaßt. Je nach Art der Klassifizierung der Dokumente und der Methoden diese aufzufinden, fallen die Ansätze im Schema aus [254] in unterschiedliche Kategorien (z.B. Mustererkennung).

5.4. Kompositionsmechanismen

Ein bekannter und verbreiteter Kompositionsmechanismus sind UNIX™−*Pipes* ([125]). Dieser Mechanismus erlaubt es, Programme zu kombinieren, indem die Ausgabe des einen als Eingabe in das nächste dient. Dazu sind alle Programme unter UNIX™ so angelegt, daß sie eine Sequenz von Zeichen als Eingabe konsumieren und eine Sequenz von Zeichen als Ausgabe produzieren. Trotz seiner Einfachheit (oder gerade deswegen) ist dieser Mechanismus sehr mächtig. Eine Erweiterung dieses Mechanismus' bzgl. Verarbeitung mehrerer Eingabeparameter stellt *functional composition* ([85, 99]) dar.

Will man Dokumente mit komplexen Schnittstellen komponieren, so sind obige Mechanismen nicht mehr ausreichend. Einen Ausweg bilden *Softwarearchitektur Beschreibungssprachen* (engl. **m**odule **i**nterconnection **l**anguages, abg. MILs). MILs beschreiben das Ergebnis des Entwurfs einer Softwarearchitektur in Form einzelner Moduln und der Festlegung der Beziehungen unter diesen ([66, 137, 186, 194]). Ein sehr interessantes Beispiel einer solchen Sprache ist LIL (**L**ibrary **I**nterconnection **L**anguage, [92]). LIL bietet Sprachkonstrukte an, mit deren Hilfe sehr gut beschrieben werden kann, wie sich Architektur− oder Programmcodedokumente aus anderen zusammensetzen lassen. Eine theory beschreibt allgemeine Eigenschaften, wie z.B. die einer totalen Ordnung. Ein view weist Dokumenten solche Eigenschaften zu. Mit Hilfe des Konstrukts generic ist es möglich parameterisierbare Dokumente (auch wieder theories) zu definieren, wobei die Instantiierungsparameter bestimmte Eigenschaften erfüllen müssen, welche wieder durch *theories* festgelegt werden.

Mit dem Kompositions− und Instantiierungskonstrukt make werden komplexere Dokumente aus den bereits vorhandenen erstellt. Ähnliche Ansätze bezüglich der Komposition, allerdings ohne Generizität finden sich auch in [17] und [150].

Einige Autoren ([1, 84, 140, 167, 260]) gehen das Kompositionssproblem von der

theoretischen Seite an und definieren Algebren über Typbezeichnern (Sorten), Funktionen/ Prozeduren (Operatoren) und Kontexte der Funktionen/ Prozeduren (Axiome).

Sieht man von wenigen Ausnahmen ab ([229, 262], oder auch z.B. bei prozeßorientierten Programmiersprachen), treten Message Passing und Vererbung immer zusammen im Rahmen einer objektorientierten Programmiersprache auf. Auch bei der Diskussion um Wiederverwendbarkeit gingen von den *objektorientierten Programmiersprachen* wie z.B. Smalltalk wichtige Impulse aus ([61, 190]). Zwei sehr interessante Entwicklungen in diesem Bereich sind Eiffel™ und Meld. Beide Sprachen unterstützen Mehrfachvererbung und Generizität[4]. In Eiffel™ ([161−164]) gibt es die Möglichkeit, Vor− und Nachbedingungen für Methoden und Invarianten für Klassen zu definieren, welche ebenfalls vererbt werden. Verletzungen dieser Bedingungen können über einen Ausnahmenbehandlungsmechanismus abgehandelt werden. Um Namenskonflikte (bei Mehrfachvererbung) zu umgehen, erlaubt Eiffel beliebige Umbenennungen von ererbten Methoden und Instanzvariablen. Eiffel ist eine streng typisierte Sprache und wird nach C übersetzt, Aufrufe extern definierter Funktionen sind möglich. Ein weiterer Vorzug von Eiffel ist eine umfangreiche Standardbibliothek mit (optimierten) Basisklassen zur Behandlung von Feldern, Listen, Bäumen und Graphen. Eine Besonderheit in MELD ([118, 119]) ist die Möglichkeit, mehrere Klassen zusammenzufassen und ihnen eine eigene Schnittstelle zu geben. Dies erlaubt die Definition von Teilsystemen mit Methoden und Instanzvariablen, die nur innerhalb dieses Teilsystemes vererbbar sind.

Die Ansätze dieses Abschnittes korrespondieren zur gleichnamigen Einteilung in [26]. Alle hier beschriebenen Ansätze abstrahieren von dokumentinternen Details und fallen in die Kategorie Abstraktion/ Spezialisierung des Wegner'schen Klassifikationsschemas [254].

5.5. Metaprogrammierung

Unter Metaprogrammierung versteht man das Schreiben von Programmen, welche Programme produzieren ([135]), oder die Angabe ausführbarer Spezifikationen. Einen Ansatz zur Metaprogrammierung bilden *Generatoren*. Sie übersetzen eine Spezifikation eines Problems in ein Anwendungsprogramm. Dieses Vorgehen setzt voraus, daß es möglich ist, eine Spezifikationssprache zu finden, welche den Anwendungsbereich gut genug beschreibt. Generatoren erlauben eine schnelle Anpassung der Anwendung an geänderte Spezifikationen durch Neuübersetzung der Spezifikation. Dies kann aber zu schwerwiegenden Problemen führen, da dadurch alle, gegebenenfalls nachfolgenden, manuellen Änderun-

---

4 *Generizität läßt sich zwar mit Vererbung simulieren, ist aber für einfache Fälle zu umständlich (Hilfsklassen notwendig). Darum ist es sinnvoll objektorientierte Sprachen um einen einfachen generischen Mechanismus zu erweitern ([160]).*

gen am generierten Code verloren gehen. Um diese Probleme zu beseitigen werden in [16] Techniken vorgeschlagen, diese manuellen Änderungen zu unterstützen.

Very High Level Languages (VHLLs) können als Weiterentwicklung der höheren Programmiersprachen (z.B. Ada oder Modula−2) aufgefaßt werden. Sie sind zur Lösung von allgemeinen Problemen gedacht und ermöglichen es, sehr kompakte und verständliche Programme zu formulieren. Die Sprache SETL ([81]) z.B. unter stützt mengentheoretische Konzepte und erlaubt es dadurch, Algorithmen relativ unabhängig von speziellen Datenstrukturrealisationen zu formulieren. Zur Übersetzungszeit wird dann eine geeignete Repräsentation ausgewählt. Weitere Beispiele für VHLLs finden sich in [205].

Systeme für das sogenannte *End User Programming* (EUP) stellen eine Verbindung der beiden vorgenannten Ansätze dar. Dazu werden dem Anwendungsbereich angepaßte, problemorientierte Programmiersprachen entworfen, die den Endbenutzer/inne/n erlauben, ihre Anwendung selbst zu implementieren (generieren). Ein bekanntes Beispiel hierfür ist das Programmpaket dBase (in [106]), welches Hilfsmittel zum Entwurf von Datenbanken zur Verfügung stellt. Einen Ansatz zur Verallgemeinerung solcher Ideen auf die Konfiguration von Softwaresystemen aus noch manuell zu bearbeitenden Dokumenten findet man in [168].

Ähnlich verfährt man bei der sogenannten *Spreadsheetprogrammierung* ([127]). Spreadsheets werden zur Darstellung von Abhängikeiten bei Kalkulationen und Optimierungsmodelle genutzt. Dazu werden eine Reihe von Variablen einer Reihe von Ergebnissen gegenübergestellt, die aus diesen Variablen abgeleitet werden können. Eine Änderung der Variablen/ Ergebnisse wird dann direkt weiterpropagiert, so daß immer eine konsistente und korrekte Kalkulation vorliegt (z.B. für lineareare Optimierungsmodelle im MPS−Format). Eine gute (Markt−) Übersicht über Spreadsheetprogrammierung findet sich in [215].

Diese Ansätze finden sich in [26] unter den Punkten sprachbasierte Systeme und Generatoren wieder. Einige VHLLs werden unter dem Punkt Transformationssysteme beschrieben (z.B. SETL). Bezüglich des Schemas in [254] ist eine Zuordnung weniger einfach. Generatoren, EUP und Spreadsheetprogrammierung sind Modelle zur Lösung eingeschränkter Klassen von Problemen. VHLLs abstrahieren von Realisierungsdetails zur Lösung allgemeiner Probleme.

5.6. wissensbasierte Programmierung

Im Bereich wissensbasierte Programmierung sind die *Transformationssysteme* CIP ([182]) und Draco ([80, 171, 175]) besonders zu erwähnen, welche nachfolgende Entwicklungen stark beeinflußt haben. Draco besteht aus einer Hierarchie problemorientierter Programmiersprachen. Die Konstrukte dieser Sprachen werden in Sprachen niedrigerer Levels dieser Hierarchie implementiert. Das Programmierparadigma von Draco ist die source−to−source Transformationen zwis-

chen problemorientierten Programmiersprachen bis hin zu einer konventionellen höheren Programmiersprache als Endprodukt dieses Transformationsprozesses. Allerdings ist es meist äußerst schwierig diese Transformationen zu finden. Eine systematische Zusammenstellung von Transformationssystemen findet sich in [184].

"Kern" aller *intelligenten Assistent/inn/en*, wie z.B. dem Programmer's Apprentice (PA) Projekt ([204, 207]) sind Bibliotheken von Programmplänen. Diese Pläne sind programmiersprachenunabhängig und repräsentieren die algorithmische Struktur von Konzepten verschiedener Anwendungsbereiche (z.B. Suchen). Diese Pläne können mit Hilfe der intelligenten Assistent/inn/en zusammengesetzt und verifiziert werden. Gute Übersichten über den Bereich wissensbasierte Programmierung finden sich in [22, 93, 205].

Wissensbasierte Programmierung wird in [26] nur unter dem Punkt Transformationssysteme behandelt. Intelligente Assistent/inn/en werden nicht betrachtet. Im Schema aus [254] lassen sich Ansätze zur wissensbasierten Programmierung überall einordnen.

6. Integration von Konzepten der Wiederverwendbarkeit in eine Softwareentwicklungsumgebung

Die Möglichkeit der Wiederverwendung vorhandener Dokumente wird nur dann genutzt, wenn dies geringen Aufwand erfordert. Dies ist nur möglich, wenn den Benutzer/inne/n *Werkzeuge* zur Hand gegeben werden, welche die im vorigen Kapitel beschriebenen Ansätze im Softwareentwicklungsprozeß *unterstützen*. Die in der Literatur diskutierten Ansätze hierfür werden im folgenden skizziert.

Ein spezieller Ansatz ist die Anwendung von *ingenieurmäßigen Techniken* auf die *Entwicklung* von neuer *Software* aus *vorhandenen Dokumenten*. Diese Ansätze werden mit Software Computer Aided Design ([37, 222]) oder Computer Aided Programming bezeichnet ([16]) bezeichnet. In [37] wird anhand einer Prototypimplementierung die Integration einer Bibliothek wiederverwendbarer Dokumente (RSL) in ein Programmieren−im−Großen−Werkzeug (SoftCAD) beschrieben. Die Bibliothek enthält zu jedem dort abgelegten Dokument eine Beschreibung in Form einer Liste von Attributen. Sie enthält u.a. Name und Klassifizierung des Dokumentes, Beziehungen zu anderen Dokumenten, eine Minidokumentation und Testbeschreibungen. Ein menügesteuertes anderes Werkzeug erlaubt die Wartung der Bibliothek. Anfragen an die Bibliothek können durch Angabe verschiedener Attribute oder mit Hilfe einer natürlichsprachlichen Eingabe erfolgen. Ergebnis einer Anfrage ist eine Liste aller Dokumente, die die geforderten Eigenschaften erfüllen. Um eine Auswahl aus dieser Liste zu treffen gibt es ein weiteres Werkzeug, welches die einzelnen Dokumente bewertet. Dazu gibt der/ie Benutzer/in an, welche Bewertungskriterien (z.B. Komplexität, Lesbarkeit, Dokumentation,

...) ihm/r besonders wichtig erscheinen. Softwarearchitekturbeschreibungen in SoftCAD erfolgen mit Hilfe objektorientierter Diagramme (sogenannte *Booch diagrams*, siehe [30]), welche in Ada−PDL ([176]) übersetzt werden. In jeder Stufe des Entwurfs hat der/ie Benutzer/in online−Zugriff auf die Bibliothek und kann gefundene Dokumente, oder Teile davon, in sein/ihr/e Architektur einbetten. Die Einbettung erfogt im vorliegenden Prototypen manuell. Eine Unterstützung durch zusätzliche Werkzeuge wäre hier noch wünschenswert.

Software Factories (SFs) sind Erweiterungen dieser Ansätze und unterstützen nicht nur den Softwareentwicklungsprozeß im engeren Sinne, sondern auch Tätigkeiten, die indirekt mit der Softwareentwicklung zu tun haben, wie z.B. Produktmanagement und Personalplanung. Eine SF ist also eine Umgebung, welche es Softwareherstellern erlaubt, kommerzielle Softwareprodukte in einheitlicher Art und Weise zu entwerfen, testen, vermarkten, installieren und warten [154]). Beispielarchitekturen von SFs finden sich z.B. in [68, 82, 154].

Weitere Ansätze in dieser Richtung sind Object Oriented Design ([30]) und Computer Aided Software Engineering ([41]), wobei allerdings die Wiederverwendbarkeit von Software mehr als positiver Seiteneffekt der jeweiligen Ansätze betrachtet wird.

7. Schlußbemerkungen

Obwohl McIllroy bereits 1968 ([174]) die Notwendigkeit einer industrieellen Softwarekomponentenfertigung postulierte, ist die Wiederverwendung von Software auch 20 Jahre danach noch ein Stiefkind der Softwaretechnik.

Warum dies so ist, kann nicht nur technische Gründe haben, wie einige vielversprechende Ansätze zeigen. Vielmehr gibt es auch ökonomische, rechtliche und soziale Probleme bzgl. Wiederverwendbarkeit zu beseitigen. Dazu gehören z.B. die anfallenden Mehrkosten, um Softwaredokumente wiederverwendbar zu machen, die Festlegung von Standards für Beschreibungen von Komponenten und das Problem der Softwarepiraterie.

8. Bibliographie

1. Agnarsson, S. and M. S. Krishnamoorthy, "Towards a Theory of Packages," SigPlan Notices, vol. 20, no. 7, pp. 117−130, Jul 1985.
2. Agresti, W. W. and F. E. McGarry, "The Minnobrook Workshop on Software Reuse: A Summary Report," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
3. Alexandridis, N. A., "Adaptable Software and Hardware: Problems and Solutions," IEEE Computer, vol. 19, no. 2, pp. 29−39, Feb 1986.
4. Alws, K.−H. and I. Glasner−Schapeler, "Experiences with Object Oriented Programming," Proc TAPSOFT '85 (LNCS 186), pp. 435−452, 1985.
5. Arango, G., I. Baxter, P. Freeman, and C. Pidgeon, "TMM: Software Maintenance by Transformation," in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 213−225, IEEE Computer Society Press, Washington, D.C., 1987.

6. Arango, G. and P. Freeman, "Application of Artificial Intelligence," ACM SIGSoft Software Engineering Notes, vol. 13, no. 1, pp. 32−38, Jan 1988.
7. Arnold, R. S. el al, "Mechanisms for Extending the Applicability of Reusable Components," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
8. Arnold, S. P. and S. L. Stepoway, "The Reuse System: Cataloging and Retrieval of Reusable Software," 32nd IEEE COMPCON 87, pp. 376−379, San Francisco, California, Feb (23−27) 1987.
9. Balzer, R., "A 15 Year Perspective on Automatic Programming," IEEE Transactions on Software Engineering, vol. SE−11, no. 11, pp. 1257−1268, Sep 1984.
10. Balzer, R., "Evolution as a new Basis for Reusability," Proc. 8th Computer Software & Applications Conf. (COMPSAC) 84, pp. 471−473, Chicago, Ill., Nov (7−9) 1984.
11. Balzer, R., T. E. jr Cheatham, and C. Green, "Software Technology in the 1990's: Using a New Paradigm," IEEE Computer, vol. 16, no. 11, pp. 39−46, Nov 1983.
12. Bardin, B. M. and C. J. Thompson, "Composable Ada Software Components and the Re−Export Paradigm," ACM Ada Letters, vol. 8, no. 1, pp. 58−79, Jan/Feb 1988.
13. Bardin, B. M. and C. J. Thompson, "Using the Re−Export Paradigm to Build Composable Ada Software Components," ACM Ada Letters, vol. 8, no. 2, pp. 39−54, Mar/Apr 1988.
14. Barnes, B. el al, "A Framework and Economic Foundation for Software Reuse," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
15. Barstow, D., "Artificial Intelligence and Software Engineering," Proc. of the 9th Int. Conf. on Software Engineering, pp. 200−211, Monterey, California, Mar/Apr (30−2) 1987.
16. Basset, P., "Design Principles for Software Manufacturing Tools," Proc. ACM '84 Annual Conference; The Fifth Generation Challenge, pp. 85−93, Oct (8−10) 1984.
17. Basset, P. G., "Frame−Based Software Engineering," IEEE Software, vol. 4, no. 4, pp. 9−16, Jul 1987.
18. Batz, J. C. et al, "The Application−Specific Task Area," IEEE Computer, vol. 16, no. 11, pp. 78−85, Nov 1983.
19. Baxter, I. D., "Reusing Design Histories via Transformation Systems," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
20. Berard, E. V., "Software Reusability cannot be considered in a Vacuum," 32nd IEEE COMPCON 87, pp. 390−391, San Francisco, California, Feb (23−27) 1987.
21. Bergin, J. and S. Greenfield, "What does Modula−2 Need to Fully Support Object Oriented Programming?," SigPlan Notices, vol. 23, no. 3, pp. 73−82, Mar 1988.
22. Biermann, A. W., G. Guiho, and Y. Kodratoff, Automatic Program Construction Techniques, pp. 570, Macmillan Publ. Comp., New York, 1984.
23. Biggerstaff, T., Proceedings of the ITT Workshop on Reusability in Programming, ITT Programming, Stratford, CT, 1983.
24. Biggerstaff, T., "A Radical Hypothesis: Reusability is the Essence of Design," Proc. 8th Computer Software & Applications Conf. (COMPSAC) 84, pp. 474−475, Chicago, Ill., Nov (7−9) 1984.
25. Biggerstaff, T., "Design Recovery for Maintenance and Reuse," Unpublished Manuscript, pp. 26, MCC, Jan 1989.
26. Biggerstaff, T. and A. J. Perlis, "Foreword to Special Issue of IEEE TOSE," IEEE Transactions on Software Engineering, vol. SE−10, no. 5, pp. 474−475, Sep 1984.
27. Biggerstaff, T. and C. Richter, "Reusability Framework, Assessment and Directions," IEEE Software, vol. 4, no. 2, pp. 50−64, Mar 1987.

28. Boehm, B. W. and T. A. Standish, "Software Technology in the 1990's: Using an Evolutionary Paradigm," IEEE Computer, vol. 16, no. 11, pp. 30−38, Nov 1983.
29. Boehm, B. W. and P. N. Papaccio, "Understanding and Controlling Software Costs," IEEE Transactions on Software Engineering, vol. SE−14, no. 10, pp. 1462−1477, Oct 1988.
30. Booch, G., Software Components with Ada: Structures, Tools, and Subsystems, Benjamin/ Cummings, Menlo Park, 1987.
31. Börstler, J. E. W., "Die Wiederverwendbarkeit von Software − eine Einführung," unpublished manuscript, pp. 13, Aachen, Apr 1988.
32. Boyle, J. M. and M. N. Muralidharan, "Program Reusability through Program Transformation," IEEE Transactions on Software Engineering, vol. SE−10, no. 5, pp. 574−588, Sep 1984.
33. Braun, C. L. and J. B. Goodenough, "Ada Reusability Guidelines," 3285−2−208/2, pp. 79, SofTech, Inc, Waltham, MA, Apr 1985.
34. Braun, U., "Ein Expertensystem zum Auffinden von Programmbausteinen," TR 05.373, pp. 192, IBM Laboratory, Böblingen, Jun 1986.
35. Braun, U. and H. U. Schmid, "Wiederverwendbare abstrakte Datentypen und deren Auswahl durch ein Expertensystem," Informatik − Forschung und Entwicklung, no. 3, pp. 164−181, Mar 1988.
36. Budde, R. et al, Approaches to Prototyping, Springer, Berlin, 1984.
37. Burton, B. A. et al, "The Reusable Software Library," IEEE Software, vol. 4, no. 4, pp. 25−33, Jul 1987.
38. Buxton, J. N. and B. Randell (eds), Software Engineering Techniques (NATO Conference on Software Engineering 1969), NATO Science Commitee, Birmingham, 1970.
39. Cameron, R. D. and M. R. Ito, "Grammar−Based Definition of Metaprogramming Systems," IEEE Transactions on Programming Languages and Systems, vol. 6, no. 1, pp. 30−54, Jan 1984.
40. Card, D. N., V. E. Curch, and W. W. Agresti, "An Empirical Study of Software Design Practices," IEEE Transactions on Software Engineering, vol. SE−12, no. 2, pp. 264−271, Feb 1986.
41. Case, A. F. jr., "Computer−Aided Software Engineering (CASE): Technology for Improving Software Development Productivity," ACM SIGBDP, vol. 17, no. 1, pp. 35−43, Fall 1985.
42. Cheatham, T. E. jr., "Reusability Through Program Transformation," IEEE Transactions on Software Engineering, vol. SE−10, no. 5, pp. 589−594, Sep 1984.
43. Cheatham, T. E. jr., G. H. Holloway, and J. A. Townley, "Program Refinement By Transformation," Proc. of the 5th Int. Conf. on Software Engineering, pp. 430−437, San Diego, California, Mar (9−12) 1981.
44. Chen, Y.−F. and C. V. Ramamoorthy, "The C Information Abstractor," Proc. 10th Computer Software & Applications Conf. (COMPSAC) 86, pp. 291−298, Chicago, Ill., Oct 1986.
45. Clapp, J., "Software Reusability: A Management View," Proc. 8th Computer Software & Applications Conf. (COMPSAC) 84, pp. 479−480, Chicago, Ill., Nov (7−9) 1984.
46. Cleaveland, J. C., "Building Application Generators," IEEE Software, vol. 5, no. 4, pp. 25−33, Jul 1988.
47. Cloarec, J.−F. and R. Valent, "A Knowledge Based Environment for S/W System Configuration Reusing Components," Techniqual Report (ESPRIT Project No 974), pp. 23, 1987.
48. Collberg, C. S. and M. G. Krampell, "A Property−Based Method for Selecting Among Multiple Implementations of Modules," Proc. of the 1st European Software Engineering Conference (ESEC '87), pp. 207−215, Straßburg, France, Mar/Apr (30−2) 1987.
49. Conn, R., The Ada Software Repository Master Index, pp. 428, Echelon Inc, Los Altos, CA,

1986.
50. Conn, R., The Ada Software Repository and the Defense Data Network: A Resource Handbook, Echelon Inc, Los Altos, CA, Dec 1986.
51. Conn, R., "The Ada Software Repository," 32nd IEEE COMPCON 87, pp. 372−375, San Francisco, California, Feb (23−27) 1987.
52. Conn, R., "The Ada Software Repository and Software Reusability," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
53. Croft, W. B., "Approaches to Intelligent Information Retrieval," Information Processing & Management, vol. 23, no. 4, pp. 249−254, Apr 1987.
54. Curry, G. A. and R. M. Ayers, "Experience with Traits in the Xerox Star Workstation," IEEE Transactions on Software Engineering, vol. SE−10, no. 5, pp. 519−527, Sep 1984.
55. Czuchry, A. J. and D. R. Harris, "KBRA: A New Paradigm for Requirements Engineering," IEEE Expert, vol. 3, no. 4, pp. 21−35, Winter 1988.
56. Czyzowicz, J. and M. Iglewski, "Implementing Generic Types in Modula−2," SigPlan Notices, vol. 20, no. 12, pp. 26−32, Dec 1985.
57. Davis, A. M., E. H. Bersoff, and E. R. Comer, "A Strategy for Comparing Alternative Software Development Life Cycle Models," IEEE Transactions on Software Engineering, vol. SE−14, no. 10, pp. 1453−1461, Oct 1988.
58. Dennis, R. J. St. et al, "Measurable Characteristics of Reusable Ada Software," ACM Ada Letters, vol. 6, no. 2, pp. 41−50, Mar/Apr 1986.
59. Dennis, R. J. St., "Reusable Ada Software Guidelines," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
60. DePree, R. W., "Pattern Recognition in Software Engineering," IEEE Computer, vol. 16, no. 5, pp. 48−52, May 1983.
61. Deutsch, L. P., "Reusability in the Smalltalk−80 Programming System," in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 91−95, IEEE Computer Society Press, Washington, D.C., 1987.
62. Dierckx, W. and B. Philips, "Rewriting Programs to Have a Base for New Software," Microprocessing and Microprogramming, vol. 16, no. 11/ 12, pp. 211−214, Nov/Dec 1985.
63. Dillistone, B. R., "Configuration management within an IPSE and its implications for software re−use," in Software Engineering Environments, ed. P. Brereton, pp. 133−146, Ellis Horwood, 1988.
64. Doberkat, E.−E., "Zur Wiederaufbereitung von Software," Informatik − Forschung und Entwicklung, vol. 4, no. 1, pp. 14−24, Feb 1989.
65. Donzeau−Gouge, V., B. Lang, and B. Me'le'se, "Practical Applications of a Syntax Directed Pogram Manipulation Environment," Proc. of the 7th int. Conf. on Software Engineering, pp. 346−354, 1984.
66. Ebert, J. and G. Engels, "Ein Konzept zur Beschreibung von Software Architekturen," Fachberichte Informatik, Bericht 11/88, pp. 26, EWH Rheinland−Pfalz, Koblenz, 1988.
67. Eder, J. and W. R. Rossak, "A Concept For Software Reuse Based on Semantic Data Models," Technical Report TRUBWI 3/88, pp. 26, UBW Klagenfurt, Klagenfurt, Austria, 1988.
68. Eliot, L. B. and W. Scacchi, "Towards a Knowledge Based System Factory: Issues and Implementations," IEEE Expert, vol. 1, no. 4, pp. 51−58, Winter 1986.
69. Embley, D. W. and S. N. Woodfield, "A Knowledge Structure for Reusing Abstract Data Types," Proc. of the 9th Int. Conf. on Software Engineering, pp. 360−368, Monterey, Cal., Mar/Apr (30−2) 1987.
70. Endres, A., "Software−Wiederverwendung: Ziele, Wege und Erfahrungen," Informatik Spektrum, vol. 11, no. 2, pp. 85− 95, Apr 1988.

71. Feather, M. S., ”Reuse in the Context of a Transformation Based Methodology,” in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 226−234, IEEE Computer Society Press, Washington, D.C., 1987.
72. Fischer, G., ”Cognitive View of Reuse and Redesign,” IEEE Software, vol. 4, no. 4, pp. 60−72, Jul 1987.
73. Fischer, G., A. C. Lemke, and C. Rathke, ”From Design to Redesign,” Proc. of the 9th Int. Conf. on Software Engineering, pp. 369−376, Monterey, Cal., Mar/Apr (30−2) 1987.
74. Frakes, W. B., ”Software Reuse Through Information Retrieval,” ACM SIGIR Forum, vol. 21, no. 1/2, pp. 30−36, Fall/Winter 1986/87.
75. Frakes, W. B., ”Software Reuse Through Information Retrieval,” 32nd IEEE COMPCON 87, pp. 380−384, San Francisco, California, Feb (23−27) 1987.
76. Frakes, W. B. and B. A. Nejmeh, ”An Information System for Software Reuse,” in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
77. Freeman, P., ”Reusable Software Engineering: Concepts and Research Directions,” in IEEE Tutorial on Software Design Techniques (4th ed.), ed. P. Freeman, A. I. Wasserman, pp. 63−78, IEEE Computer Society Press, Washington, D.C., 1983.
78. Freeman, P., IEEE Tutorial: Software Reusability, pp. 298, IEEE Computer Society Press, Washington, D.C., 1987.
79. Freeman, P., ”A Perspective on Reusability,” in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 2−8, IEEE Computer Society Press, Washington, D.C., 1987.
80. Freeman, P., ”A Conceptual Analysis of the Draco Approach to Constructing Softwar Systems,” in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 192−212, IEEE Computer Society Press, Washington, D.C., 1987.
81. Freudenberger, S. M., J. T. Schwarz, and M. Sharir, ”Experience with the SETL Optimizer,” ACM Trans. on Programming Languages and Systems, vol. 5, no. 1, pp. 26−45, Jan 1983.
82. Fujino, K., ”Software Factory Engineering; Todays and Future,” Proc. of the 1987 Fall Joint Computer Conference, pp. 262−270, Dallas, Texas, Oct (25−29) 1987.
83. Futatsugi, K., J. A. Goguen, J.−P. Jouannaud, and J. Meseguer, ”Principles of OBJ2,” Proc. of the 12th Ann Symp on POPL, pp. 52−66, New Orleans, Louisiana, Jan (14−14) 1985.
84. Futatsugi, K., J. A. Goguen, J. Meseguer, and K. Okada, ”Parameterized Programming in OBJ2,” Proc. of the 9th Int. Conf. on Software Engineering, pp. 51−60, Monterey, Cal., Mar/Apr (30−2) 1987.
85. Garg, V. K., ”Functional Composition − A mechanism to compose programs,” ACM SIGSoft Software Engineering Notes, vol. 10, no. 3, pp. 37−39, Jul 1985.
86. Gargaro, A. and T. L. Pappas, ”Reusability Issues and Ada,” IEEE Software, vol. 4, no. 4, pp. 43−51, Jul 1987.
87. Gaube, W., P. C. Lockemann, and H. C. Mayr, ”Wiederfinden zum Wiederverwenden: Rechnergestützter Modul−Nachweis auf der Basis formaler Spezifikationen,” in Software−Architektur und modulare Programmierung, Berichte des German Chapter of the ACM No 26, ed. Wippermann, pp. 66−80, Teubner, 1986.
88. Gautier, R., H. Oliver, and M. Ratcliffe, ”A Language for Describing Reusable Software Components,” Techniqual Report, University College of Wales, 1988.
89. Gerhart, S. L., ”Reusability Lessons from Verification Technology,” in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 273−284, IEEE Computer Society Press, Washington, D.C., 1987.
90. Ghisio, O., P. Gouthier, and S. Truzzi, ”An Extended Approach to Reusability,” 32nd IEEE COMPCON 87, pp. 385−389, San Francisco, California, Feb (23−27) 1987.
91. Goguen, J. A., ”Parameterized Programming,” IEEE Transactions on Software Engineer-

ing, vol. SE−10, no. 5, pp. 528−543, Sep 1984.
92. Goguen, J. A., "Reusing and Interconnecting Software Components," in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 251−263, IEEE Computer Society Press, Washington, D.C., 1987.
93. Goldberg, A. T., "Knowledge−Based Programming: A Survey of Program Design and Construction Techniques," IEEE Transactions on Software Engineering, vol. SE−12, no. 7, pp. 752−767, Jul 1986.
94. Grabow, P. C. and W. B. Noble, "Reusable Software Concepts and Software Development Methodologies," Internal Report, pp. 8, Universitof Texas, Arlington, Dec 1984.
95. Gruman, G., "Early reuse practice lives up to its promise," IEEE Software, vol. 5, no. 6, pp. 87−91, Nov 1988.
96. Grune, D., "Generic Packages in C," SigPlan Notices, vol. 21, no. 8, pp. 31−39, Aug 1986.
97. Guariso, G. and H. Werthner, "A Software Base For Environmental Studies," The Computer Journal, vol. 31, no. 6, pp. 550−552, Jun 1988.
98. Guerrieri, E., "Searching for Reusable Software Components with the RAPID Center Library System," Technical Report TP 259, pp. 13, SofTech, Waltham, MA, Mar 1988.
99. Hall, P. A. V., "Reusable and reconfigurable software using C," in Software Engineering 86 (IEE Computing Series 6), ed. P. Barnes, P. Brown, pp. 165−174, Peter Peregrinus Ltd, London, 1986.
100. Hall, P. A. V., "Software components and reuse − getting more out of your code," Information and Software Technology, vol. 29, no. 1, pp. 38−43, Jan/Feb 1987.
101. Harrison, W., "A Program Development Environment for Programming by Refinement and Reuse," Proc. of the 19th Hawaii Int. Conf. on System Sciences, pp. 459−469, Hawaii, 1986.
102. Harrison, W., "RPDE3: A Framework for Integrating Tool Fragments," IEEE Software, vol. 4, no. 6, pp. 46−56, Nov 1987.
103. Hartmann, A. C., "Software or Silicon? The Designer's Option," Proc of the IEEE, vol. 74, no. 6, pp. 861−874, Jun 1986.
104. Honiden, S. et al, "Software Prototyping with Reusable Components," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
105. Horgen, H., "Tool Use": An Advanced Support Environment for Method−Driven Development and Evolution of Packaged Software," in ESPRIT '85: Status Report of Continuing Work, ed. Commission of the European Communities, pp. 545−562, Elsevier Science Publishers B. V. (North Holland), 1985.
106. Horowitz, E., A. Kemper, and B. Narasimhan, "A Survey of Application Generation," IEEE Software, vol. 2, no. 1, pp. 40−54, Jan 1985.
107. Horowitz, E. and J. B. Munson, "An Expansive View of Reusable Software," in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 39−49, IEEE Computer Society Press, Washington, D.C., 1987.
108. Hrycej, T., "A Knowledge−Based Problem−Specific Program Generator," SigPlan Notices, vol. 22, no. 2, pp. 53−61, Feb 1987.
109. Ichbiah, J. D., "On the Design of Ada," in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 59−68, IEEE Computer Society Press, Washington, D.C., 1987.
110. Iscoe, N., "Domain−Specific Reuse: An Object−Oriented and Knowledge−Based Approach," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
111. Jähnichen, S., F. A. Hussain, and M. Weber, "Program Development by Transformation and Refinement," Proc. IFIP − Int. Workshop on Advanced Programming Environments, pp. 471−486, Trondheim, Norwegen, Jun 1986.

112. Johnson, R. E. and B. Foote, "Designing Reusable Classes," Journal of Object Oriented Programming, vol. 1, no. 2, pp. 22−35, Jun/Jul 1988.
113. Jones, B. et al, "Issues in Software Reusability," ACM Ada Letters, vol. 4, no. 5, pp. 97−99, Mar/Apr 1985.
114. Jones, C., "The Impact Of Reusable Modules and Functions," in Programming Productivity, ed. C. Jones, pp. 151−160, 204−210, McGraw−Hill, 1986.
115. Jones, G., "Software Reusability: Approaches and Issues," Proc. 8th Computer Software & Applications Conf. (COMPSAC) 84, pp. 476−478, Chicago, Ill., Nov (7−9) 1984.
116. Jones, G., "Methodology/Environment Support for Reusability," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
117. Jones, T. C., "Reusability in Programming: A Survey of the State of the Art," IEEE Transactions on Software Engineering, vol. SE−10, no. 5, pp. 488−493, Sep 1984.
118. Kaiser, G. E. and D. Garlan, "Melding Software Systems from Reusable Building Blocks," IEEE Software, vol. 4, no. 4, pp. 17−24, Jul 1987.
119. Kaiser, G. E. and D. Garlan, "MELDing Data Flow and Object Oriented Programming," SigPlan Notices Special Issue (Proc OOPSLA '87), vol. 22, no. 12, pp. 254−267, Dec 1987.
120. Kaiser, G. E. and D. Garlan, "Synthesizing Programming Environments from Reusable Features," Unpublished Manuscript (to appear in '89), pp. 23, 1988.
121. Kandt, K., "Pegasus: A Tool for the Acquisition and Reuse of Software Designs," Proc. 8th Computer Software & Applications Conf. (COMPSAC) 84, pp. 288−293, Chicago, Ill., Nov (7−9) 1984.
122. Kang, K. C., "A Reuse−Based Software Development Methodology," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
123. Karimi, J. and B. R. Konsynski, "An Automated Software Design Assistant," IEEE Transactions on Software Engineering, vol. SE−14, no. 2, pp. 194−210, Feb 1988.
124. Katz, S., C. A. Richter, and K.−S. The, "PARIS: A System for Reusing Partially Interpreted Schemas," Proc. of the 9th Int. Conf. on Software Engineering, pp. 377−385, Monterey, Cal., Mar/Apr (30−2) 1987.
125. Kernighan, B. W., "The UNIX System and Software Reusability," in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 145−150, IEEE Computer Society Press, Washington, D.C., 1987.
126. Kleine, K., "A Catalogue of Data Types," TR 05.371, pp. 317, IBM Laboratory, Böblingen, Apr 1986.
127. Kokol, P., "Some Applications of Spreadsheet Programs in Software Engineering," ACM SIGSoft Software Engineering Notes, vol. 12, no. 3, pp. 45−50, Jul 1987.
128. Kristensen, B. B. et al, "An Algebra for Program Fragments," SigPlan Notices, vol. 20, no. 7, pp. 161−170, Jul 1985.
129. Lanergan, R. G. and C. A. Grasso, "Software Engineering with Reusable Designs and Code," in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 151−154, IEEE Computer Society Press, Washington, D.C., 1987.
130. Latour, L., "SEE: An Automated Tool to Facilitate Reuse in Ada Project Development," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
131. Ledbetter, L. and B. Cox, "Software−ICs," BYTE, vol. 10, no. 6, pp. 307−316, Jun 1985.
132. Lenz, M., H. A. Schmid, and P. F. Wolf, "Software Reuse through Building Blocks," IEEE Software, vol. 4, no. 4, pp. 34−42, Jul 1987.
133. Lerner, E. J., "Automatic programming," IEEE Spectrum, vol. 19, no. 8, pp. 28−33, Aug

1982.
134. Lerner, E. J., ”Programming for nonprogrammers,” IEEE Spectrum, vol. 19, no. 8, pp. 34−38, Aug 1982.
135. Levy, L. S., ”A Metaprogramming Method and its Economic Justification,” IEEE Transactions on Software Engineering, vol. SE−12, no. 2, pp. 272−277, Feb 1986.
136. Lewerentz, C., Interaktives Entwerfen großer Programmsysteme (IFB 194), Springer, Berlin, 1988.
137. Lewerentz, C. and M. Nagl, ”A Formal Specification Language for Software Systems defined by Graph Grammars,” in Proceedings WG ’84 on Graphtheoretic Concepts in Computer Science, ed. U. Pape, Trauner Verlag, Linz, 1984.
138. Lieberherr, K. J. and A. J. Riel, ”Demeter: A CASE Study of Software Growth Through Parameterized Classes,” Proc. of the 10th Int. Conf. on Software Engineering, pp. 254−264, Singapur, Mar (11−15) 1988.
276. C. Lins, ”The Modula−2 Software Component Library, Vol. 1,” Springer−Verlag, 1989.
139. Litvintchouk, S. D. and A. S. Matsumoto, ”An Algebraic Approach to Reusable Ada Components,” ACM Ada Letters, vol. 3, no. 1, pp. 51−54, Jul/Aug 1983.
140. Litvintchouk, S. D. and A. S. Matsumoto, ”Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification,” in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 264−272, IEEE Computer Society Press, Washington, D.C., 1987.
141. Love, T., ”The Economics of Reuse,” 33rd IEEE COMPCON 88, pp. 238−241, San Francisco, California, Feb/Mar (29−4) 1988.
142. Lubars, M. D., ”Code Reusability in the Large versus Code Reusability in the Small,” ACM SIGSoft Software Engineering Notes, vol. 11, no. 1, pp. 21−28, Jan 1986.
143. Lubars, M. D., ”Affording Higher Reliability through Software Reusability,” ACM SIGSoft Software Engineering Notes, vol. 11, no. 5, pp. 39−42, Oct 1986.
144. Lubars, M. D., ”Wide Spectrum Support for Software Reusability,” in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
145. Lubars, M. D. and M. T. Harandi, ”Intelligent Support for Software Specification and Design,” IEEE Expert, vol. 1, no. 4, pp. 33−41, Winter 1986.
146. Lubars, M. D. and M. T. Harandi, ”Knowledge−Based Software Design Using Design Shemas,” Proc. of the 9th Int. Conf. on Software Engineering, pp. 253−262, Monterey, Cal., Mar/Apr (30−2) 1987.
147. Luqi,, ”Normalized Specifications for Identifying Reusable Software,” Proc. of the 1987 Fall Joint Computer Conference, pp. 46−49, Dallas, Texas, Oct (25−29) 1987.
148. Maarek, Y. S. and G. E. Kaiser, ”Using conceptual clustering for classifying reusable Ada code,” Proc. of the ACM SIGAda Int. Conf. (ACM Ada−Letters), pp. 208−215, Boston Massachusetts, Dec (8−11) 1987.
149. Madhavji, N. H., N. Leoutsarakos, and D. Voulioúris, ”Software Construction Using Typed Fragments,” Proc TAPSOFT ’85 (LNCS 186), pp. 163−178, 1985.
150. Madhavji, N. H., ”Fragtypes: A Basis for Programming Environments,” IEEE Transactions on Software Engineering, vol. SE−14, no. 1, pp. 85−97, Jan 1988.
151. Mano, N., ”Modeling of Data−Processing Software for Generating and Reusing their Programs,” Proc. of the 10th Int. Conf. on Software Engineering, pp. 231−240, Singapur, Mar (11−15) 1988.
152. Mathis, R. F., ”The Last 10 Percent,” IEEE Transactions on Software Engineering, vol. SE−12, no. 6, pp. 705−712, Jun 1986.
153. Matsumoto, A. S., ”Some Experiences in Promoting Reusable Software: Presentation in

Higher Abstract Levels," IEEE Transactions on Software Engineering, vol. SE−10, no. 5, pp. 502−512, Sep 1984.
154. Matsumoto, A. S., "A Software Factory: An Overall Approach to Software Production," in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 155−178, IEEE Computer Society Press, Washington, D.C., 1987.
155. Matsumoto, A. S. and Y. Ohfude, "Software Reuse Based on a Process Programming System," Proc. of the 1987 Fall Joint Computer Conference, pp. 254−261, Dallas, Texas, Oct (25−29) 1987.
156. McCain, R., "A Software Development Methodology for Reusable Components," Proc. of the 18th Hawaii Int. Conf. on System Sciences, pp. 319−324, Hawaii, 1985.
157. McNicholl, D., "CAMP: Reusable Software Parts," Program Description, pp. 7, McDonnell Douglas, St. Louis, MO, Jun 1987.
158. Merkwitz, G., "Drei Building−Block Ansätze zur Wiederverwendbarkeit von Software im Vergleich: PARIS, Meld und BB/ LX," Seminarbericht SS 88, pp. 23, RWTH Aachen, Aachen, Jul 1988.
159. Meyer, B., "The Software Knowledge Base," Proc. of the 8th Int. Conf. on Software Engineering, pp. 158−165, London, GB, Aug (28−30) 1985.
160. Meyer, B., "Genericity versus Inheritance," SigPlan Notices Special Issue (Proc OOPSLA '86), vol. 21, no. 11, pp. 391− 405, Nov 1986.
161. Meyer, B., "EIFFEL: Programming for Reusability and Extendibility," Internal Report TR−EI−3/GI, Vers. 1.6, pp. 12, Interactive Software Engineering, Inc., 1987.
162. Meyer, B., "Reusability: The Case for Object Oriented Design," IEEE Software, vol. 4, no. 2, pp. 50−64, Mar 1987.
163. Meyer, B., "Eiffel: A Language and Environment for Software Engineering," Journal of Systems and Software, vol. 8, no. 3, pp. 199−246, Jun 1988.
164. Meyer, B., Object−oriented Software Construction, Prentice Hall, New−York, 1988.
165. Meyer, B., "EIFFEL: Reusability and Reliability," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
166. Micallef, J., "Encapsulation, Reusability, and Extensibility in Object−Oriented Programming Languages," Journal of Object Oriented Programming, vol. 1, no. 1, pp. 12−39, Apr/ Mai 1988.
167. Milner, R., A Calculus of Communicating Systems (LNCS 92), Springer−Verlag, Berlin, 1980.
168. Mittermeir, R. T. and M. Oppitz, "Software Bases for the Flexible Composition of Application Systems," IEEE Transactions on Software Engineering, vol. SE−13, no. 4, pp. 440−460, Apr 1987.
169. Mittermeir, R. T. and W. Rossak, "Software Bases and Software Archives," Proc. of the 1987 Fall Joint Computer Conference, pp. 21−28, Dallas, Texas, Oct (25−29) 1987.
170. Müller, H. A. and K. Klashinsky, "Rigi: A System for Programming−in−the−large," Proc. of the 10th Int. Conf. on Software Engineering, pp. 80−86, Singapur, Mar (11−15) 1988.
171. Nadarzinski, E., "Der Transformationsansatz zur Wiederverwendbarkeit von Software am Beispiel DRACO," Seminarbericht SS 88, pp. 26, RWTH Aachen, Aachen, Jun 1988.
172. Nakagawa, A. T., K. Futatsugi, S. Tomura, and T. Shimizu, "Algebraic Specification of MacIntosh's Quickdraw using OBJ2," Proc. of the 10th Int. Conf. on Software Engineering, pp. 334−343, Singapur, Mar (11−15) 1988.
173. Narfelt, K.−H. and D. Schefstrom, "Extending the Scope of the Program Library," ACM Ada Letters, vol. 5, no. 2, pp. 25−40, Sep/Oct 1985.
174. Naur, P. and B. Randell (eds), Software Engineering Techniques (NATO Conference on Software Engineering 1968), NATO Science Commitee, Birmingham, 1969.

175. Neighbors, J. M., ”The Draco Approach to Constructing Software from Reusable Components,” in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 181−191, IEEE Computer Society Press, Washington, D.C., 1987.
176. Nejmeh, B. A. and H. E. Dunsmore, ”A Survey of Program Design Languages (PDLs),” Proc. 10th Computer Software & Applications Conf. (COMPSAC) 86, pp. 94−99, Chicago, Ill., Oct 1986.
177. Nourani, C. F. and G. A. Jones, ”Software Reusability − A Perspective,” Proc. of the 18th Hawaii Int. Conf. on System Sciences, pp. 447−456, Hawaii, 1985.
178. Oktaba, H. and R. Berber, ”Crafting Reusable Software in Modula−2,” BYTE, vol. 12, no. 9, pp. 123−128, Sep 1987.
179. Onuegbe, E. O., ”Software Classification as an Aid to Reuse: Initial Use as Part of aRapid Prototyping System,” in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
180. Parker, J. and B. Hendley, ”The re−use of low−level programming knowledge in the UNIVERSE programming evironment,” in Software Engineering Environments, ed. P. Brereton, pp. 104−115, Ellis Horwood, 1988.
181. Parnas, D. L., P. C. Clements, and J. C. Wileden, ”Enhancing Reusability with Information Hiding,” in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 83−91, IEEE Computer Society Press, Washington, D.C., 1987.
182. Partsch, H., ”The CIP Transformational System,” in Program Transformation and Programming Environments, ed. P. Pepper, pp. 305−322, Springer−Verlag, Berlin, Heidelberg, 1984.
183. Partsch, H., ”Transformational Program Development in a Particular Problem Domain,” Science of Computer Programming, vol. 7, pp. 99−241, 1986.
184. Partsch, H. and R. Steinbrüggen, ”Program Transformation Systems,” Computing Surveys, vol. 15, no. 3, pp. 199−236, Sep 1983.
185. Perry, J. M., J. Roder, and F. Rosene, ”Reusability Benefits of a Structure Which Supports a Development Methodology and Environment,” in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
186. Perry, D. E., ”Software Interconnection Models,” Proc. of the 9th Int. Conf. on Software Engineering, pp. 61−69, Monterey, California, Mar/Apr (30−2) 1987.
187. Pinson, L. J., R. F. Sincovec, and R. S. Wiener, A First Course in Computer Science with Modula−2, John Wiley & Sons, New−York, 1987.
188. Polak, W., ”Maintainability and Reusable Program Designs,” Internal Report, pp. 8, Kestrel Institute, Palo Alto, 1986.
189. Polster, F. J., ”Reuse of Software Through Generation of Partial Systems,” IEEE Transactions on Software Engineering, vol. SE−12, no. 3, pp. 402−416, Mar 1986.
190. Pope, S. T., A. Goldberg, and L. P. Deutsch, ”Object− Oriented Approaches to the Software Lifecycle Using the Smalltalk−80 System as a CASE Toolkit,” Proc. of the 1987 Fall Joint Computer Conference, pp. 13−20, Dallas, Texas, Oct (25−29) 1987.
191. Poulter, K. J., ”Representing Programming Knowledge in the KBPA,” in ESPRIT ’85: Status Report of Continuing Work, ed. Commission of the European Communities, pp. 349−358, Elsevier Science Publishers B. V. (North Holland), 1985.
192. Prieto−Diaz, P., ”Domain Analysis for Reusability,” in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
193. Prieto−Diaz, P. and P. Freeman, ”Classifying Software for Reusability,” IEEE Software, vol. 4, no. 1, pp. 6−16, Jan 1987.
194. Prieto−Diaz, P. and J. M. Neighbors, ”Module Interconnection Languages,” in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 117−145, IEEE Computer Society Press,

Washington, D.C., 1987.

195. Prieto−Diaz, P. and G. A. Jones, "Breathing New Life," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
196. Pyster, A. and B. Barnes, "The Software Productivity Consortium Reuse Program," 33rd IEEE COMPCON 88, pp. 242− 247, San Francisco, California, Feb/Mar (29−4) 1988.
197. Ramamoorthy, C. V., V. K. Garg, and A. Prakash, "Programming in the Large," IEEE Transactions on Software Engineering, vol. SE−12, no. 7, pp. 769−783, Jul 1986.
198. Ramamoorthy, C. V., V. K. Garg, and A. Prakash, "Support for Reusability in GENESIS," Proc. 10th Computer Software & Applications Conf. (COMPSAC) 86, pp. 299−305, Chicago, Ill., Oct 1986.
199. Ramamoorthy, C. V., V. K. Garg, and A. Prakash, "Support for Reusability in GENESIS," IEEE Transactions on Software Engineering, vol. SE−14, no. 8, pp. 1145−1153, Aug 1988.
200. Ratcliffe, M., "Report on a Workshop on Software Reuse held at Hereford, UK on 1,2 May 1986," ACM SIGSoft Software Engineering Notes, vol. 12, no. 1, pp. 42−47, Jan 1987.
201. Rauch−Hindin, W. B., "Reusable Software," Electronic Design, vol. 31, no. 3, pp. 176−194, Feb 1983.
202. Reilly, A., "Roots of Reuse," IEEE Software, vol. 4, no. 1, p. 4, Jan 1987.
203. Rice, J. R. and H. D. Schwetman, "Interface Issues in a Software Parts Technology," in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 96−105, IEEE Computer Society Press, Washington, D.C., 1987.
204. Rich, C. and R. C. Waters, "The Programmer's Apprentice Project," Internal Bibliography, MIT Artificial Intelligence Laboratory, Cambridge, MA 02139, Sep 1985.
205. Rich, C. and R. C. Waters, Readings in Artificial Intelligence and Software Engineering, pp. 622, Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.
206. Rich, C. and R. C. Waters, "Automatic Programming: Myths and Prospects," IEEE Computer, vol. 21, no. 8, pp. 40−51, Aug 1988.
207. Rich, C. and R. C. Waters, "The Programmer's Apprentice: A Research Overview," IEEE Computer, vol. 21, no. 11, pp. 11− 25, Nov 1988.
208. Rickert, N. W., "Preconditions for widespread reuse of code," ACM SIGSoft Software Engineering Notes, vol. 11, no. 2, pp. 21, Apr 1986.
209. Rossak, W. R., "Software Reuse − A Melting Pot for Software Engineering, Data Base− and AI Concepts," Proc. of the 4th Int. Conf. on Systems Research Informatics and Cybernetics, Baden−Baden, FRG, Aug (15−21) 1988.
210. Rossak, W. R., "Software Development Reusing existing Components," Proc. of the 8th Phoenix Conference on Computers and Communication, Scottsdale, Arizona, Mar (22−24) 1989.
211. Rossak, W. R. and R. T. Mittermeir, "Structuring Software Archives for Reusability," Proc. of the IASTED Int. Symp. on Applied Informatics, pp. 157−160, Grindelwald, Switzerland, Feb (17−19) 1987.
212. Ruegsegger, T., "Making Reuse Pay: The SIDPERS−3 RAPID Center," IEEE Communications, vol. 26, no. 8, pp. 16−24, Aug 1988.
213. Russel, G., "Experiences Implementing a Reusable Data Structure Component Taxonomy," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
214. Scacci, W., "Models of Software Evolution: Life Cycle and Process," SEI Curriculum Module SEI−CM−10−1.0, pp. 29, Carnegy−Mellon Univ (SEI), Pittsburg, Pennsylvania, Oct 1987.
215. Scharfenberger, W. and P. Becker, "Auswertungsprogramme," Computerwoche, vol. 16, no.

1/2, pp. 19−29, Jan 1989.

216. Scherlis, W. L., "Abstract Data Types, Specialization, and Program Reuse," Proc. IFIP − Int. Workshop on Advanced Programming Environments, pp. 433−453, Trondheim, Norwegen, Jun 1986.
217. Schleicher, A., "Konzepte der Wiederverwendbarkeit in der Programmiersprache Ada: Eine Fallstudie," Seminarbericht SS 88, pp. 50, RWTH Aachen, Aachen, Jun 1988.
218. Schneider, M., "Programmierwerkzeuge − ihre Integration in eine objektorientierte Programmierumgebung," in GI − 16. Jahrestagung, ed. G. Hommel, S. Schindler, pp. 195−211, Springer, Berlin, Oct 1986.
219. Selby, R. W., "Analyzing Software Reuse at the Project and Module Design Levels," Proc. of the 1st European Software Engineering Conf. (ESEC '87), pp. 227−235, Straßburg, France, Mar/Apr (30−2) 1987.
220. Selby, R. W., "Empirically Analyzing Software Reuse in a Production Environment," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
221. Seppänen, V., "Reusability in Software Engineering," in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 268−297, IEEE Computer Society Press, Washington, D.C., 1987.
222. Serbanati, L.−D. I., "Interform: A CAD System for Program Development," Proc. of the 9th Int. Conf. on Software Engineering, pp. 190−197, Monterey, Cal., Mar/Apr (30−2) 1987.
223. Shriver, B. D., "Reuse Revisited," IEEE Software, vol. 4, no. 1, pp. 5, Jan 1987.
224. Simos, M. A., "The Domain−Oriented Software Life Cycle: Towards an Extended Process Modell for Reusability," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
225. Smith, D., G. Kotik, and S. Westfold, "Research on Knowledge Based Software Environments on Kestrel Institute," IEEE Transactions on Software Engineering, vol. SE−11, no. 11, pp. 1278−1295, Nov 1985.
226. Snyder, A., "Encapsulation and Inheritance in Object− Oriented Programming Languages," SigPlan Notices Special Issue (Proc OOPSLA '87), vol. 22, no. 12, pp. 38−45, Dec 1987.
227. Stahl, B., "The Trouble with Application Generators," Datamation, vol. 32, no. 7, pp. 93−94, Apr 1986.
228. Standish, T. A., "An Essay on Software Reuse," IEEE Transactions on Software Engineering, vol. SE−10, no. 5, pp. 494−497, Sep 1984.
229. Stanley, M. and S. Goodenough, "Some practical aspects of software reuse," in Software Engineering Environments (IEE Computing Series 7), ed. I. Sommerville, pp. 220−235, Peter Peregrinus Ltd, London, 1986.
230. Stefik, M. and D. G. Bobrow, "Object Oriented Programming: Themes and Variations," The AI Magazine, vol. 6, no. 4, pp. 182−204, Winter 1986.
231. Stoffel, C. and J. F. H. Winkler, "PROGES−A Language for the Generation of Program Variants," Unpublished Paper, pp. 9, 1987.
232. Stovsky, M. P. and B. W. Weide, "STILE: A Graphical Design and Development Environment," 32nd IEEE COMPCON 87, pp. 247−250, San Francisco, California, Feb (23−27) 1987.
233. Sueda, N., S. Honiden, Y. Kusui, and K. Mikame, "Prolog Application in Software Component Reuse," in The Role of Language in Problem Solving 1, ed. R. Jernigan, B. W. Hamill, D. M. Weintraub, pp. 209−220, Elsevier Science Publishers B. V. (North Holland), 1985.
234. Tajima, D. and T. Matsubara, "Inside the Japanese Software Factory," IEEE Computer, vol. 17, no. 3, pp. 34−43, Mar 1984.

235. Tarumi, H., K. Agusa, and Y. Ohno, "A Programming Environment Supporting Reuse of Object−Oriented Software," Proc. of the 10th Int. Conf. on Software Engineering, pp. 265−273, Singapur, Mar (11−15) 1988.
236. Tichy, W. F., "What can Software Engineers Learn from Artificial Intelligence," IEEE Computer, vol. 20, no. 11, pp. 43−54, Nov 1987.
237. Tracz, W., "Software Reuse: Motivators and Inhibitors," 32nd IEEE COMPCON 87, pp. 358−363, San Francisco, California, Feb (23−27) 1987.
238. Tracz, W., "ADA Reusability efforts: A Survey of the State of the Practice," Proc. of the JointADA Conference '87, 1987.
239. Tracz, W., "Reusability Comes of Age," IEEE Software, vol. 4, no. 4, pp. 6−8, Jul 1987.
240. Tracz, W., "Software Reuse Myths," ACM SIGSoft Software Engineering Notes, vol. 12, no. 1, pp. 17−21, Jan 1988.
241. Tracz, W., "Software Reuse Maxims," ACM SIGSoft Software Engineering Notes, vol. 13, no. 4, pp. 28−31, Oct 1988.
242. Tracz, W., IEEE Tutorial: Software Reuse − Emerging Technology, pp. 400, IEEE Computer Society Press, Washington, D.C., 1988.
243. Tracz, W., "Confession of a Used Program Salesman," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
244. Tracz, W., "RMISE Workshop on Software Reuse Meeting Summary," in IEEE Tutorial: Software Reuse − Emerging Technology, ed. W. Tracz, IEEE Computer Society Press, Washington, D.C., 1988.
245. Volpano, D. M. and R. B. Kieburtz, "Software Templates," Proc. of the 8th Int. Conf. on Software Engineering, pp. 55−60, London, GB, Aug (28−30) 1985.
246. Waite, W. (Chr), "Program Reuse and Transformations," Proc. IFIP − Int. Workshop on Advanced Programming Environments, pp. 487−493, Trondheim, Norwegen, Jun 1986.
247. Wald, E. E., "Software Engineering with Reusable Parts," 32nd IEEE COMPCON 87, pp. 353−356, San Francisco, California, Feb (23−27) 1987.
248. Walker, J. H. et al, "The Symbolics Genera Programming Environment," IEEE Software, vol. 4, no. 6, pp. 36−45, Nov 1987.
249. Wasserman, A. I., "TMM: Software Maintenance by Transformation," in Tutorial on Software Design Techniques, ed. P. Freeman, A. I. Wasserman, pp. 43−62, IEEE Computer Society Press, Washington, D.C., 1983.
250. Wasserman, A. I. and S. Gutz, "The Future of Programming," Communications of the ACM, vol. 25, no. 3, pp. 196−206, Mar 1982.
251. Wartik, S. P. and M. H. Penedo, "Fillin: A Reusable Tool for Form−Oriented Software," IEEE Software, vol. 3, no. 2, pp. 61−69, Mar 1986.
252. Waters, R. C., "Reuse of Cliches in The Knowledge Based Editor," Proc. IFIP − Int. Workshop on Advanced Programming Environments, pp. 536−550, Trondheim, Norwegen, Jun 1986.
253. Weber, H., "Wiederverwendbarkeit von Komponenten in Software−Systemen," Seminarbericht WS 1985/86 (MEMO Nr 6), p. 44, Lehrstuhl für "Software−Technologie", Dortmund, 1986.
254. Wegner, P., "Varieties of Reusability," in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 24−38, IEEE Computer Society Press, Washington, D.C., 1987.
255. Weiser, W., "The Programming Culture and the Reuse of Code," IEEE Computer, vol. 20, no. 11, pp. 72, Nov 1987.
256. Wimmers, W., "Der objektorientierte Ansatz zur Wiederverwendbarkeit von Software. Eiffel und OBJ2 − ein Vergleich," Seminarbericht SS 88, pp. 19, RWTH Aachen, Aachen, Jul 1988.

257. Winkler, J. F. H. and C. Stoffel, "Program−Variations−in− the−Small," in Proc of the Int Workshop on Software Version and Configuration Control, Berichte des German Chapter of the ACM No 30, ed. J. F. H. Winkler, pp. 175−196, Teubner, Stuttgart, 1988.
258. Wirsing, M., "Algebraic Description of Reusable Software Components," in Handbook of Software Engineering, ed. C. R. Vick, C. V. Ramamoorthy, pp. 359−379, van Nostrand Reinhold Comp., New−York, 1983.
259. Wirsing, M., "Algebraic Description of Reusable Software Components," in COMP EURO '88, ed. E. Milgrom, P. Wodon, pp. 300−312, Computer Society Press, Washington, D.C., 1988.
260. Wirsing, M., R. Hennicker, and R. Breu, "Reusable Specification Components," Internal Report MIP−8817, pp. 17, Universität Passau, Passau, 1988.
261. Wirth, N., "From Modula to Oberon and The Programming Language Obereon," Internal Report (?), pp. 27, Institut für Informatik, ETH Zürich, Sep 1987.
262. Wirth, N., "Type Extensions," ACM Transactions on Progr Lang (TOPLAS), vol. 10, no. 2, pp. 204−214, Apr 1988.
263. Wirth, R., "Software Reusability (Panel Session)," Proc. 8th Computer Software & Applications Conf. (COMPSAC) 84, pp. 470−480, Chicago, Ill., Nov (7−9) 1984.
264. Wolf, A. L., L. A. Clarke, and J. C. Wileden, "Ada−Based Support for Programming−in−the−Large," in IEEE Tutorial: Software Reusability, ed. P. Freeman, pp. 69−82, IEEE Computer Society Press, Washington, D.C., 1987.
265. Wolf, P. F. and H. A. Schmid, "Zur Wiederverwendbarkeit von Software," in Fachberichte & Referate, Bd 16: Methoden und Werkzeuge zur Entwicklung von Programmsystemen, ed. W. E. Proebster, R. Remshardt, H. A. Schmid, pp. 155−170, Oldenbourg Verlag, Munchen, Wien, 1985.
266. Wood, M. and I. Sommerville, "A knowledge−based software components catalogue," in Software Engineering Environments, ed. P. Brereton, pp. 116−132, Ellis Horwood, 1988.
267. Wood, M. and I. Sommerville, "A knowledge−based software components catalogue," Unpublished Manuscript, pp. 15, 1988.
268. Woodfield, S. N., D. W. Embley, and D. T. Scott, "Can Programmers Reuse Software?," IEEE Software, vol. 4, no. 4, pp. 52−59, Jul 1987.
269. Yamamoto, S. and S. Isoda, "SoftDA − A Reuse−Oriented Software Design System," Proc. 10th Computer Software & Applications Conf. (COMPSAC) 86, pp. 284−290, Chicago, Ill., Oct 1986.
270. Yeh, R. T., R. T. Mittermeir, N. Roussopoulos, and J. Reed, "A Programming Environment Framework Based on Reusability," Proc. of Int. Conf. on Data Engineering (ICDE) '84, pp. 277−280, Los Angeles, 1984.
271. Yeh, R. T. and T. A. Welch, "Software Evolution: Forging a Paradigm," Proc. of the 1987 Fall Joint Computer Conference, pp. 10−12, Dallas, Texas, Oct (25−29) 1987.
272. Yin, W. et al, "Software Reusability: A Survey and a Reusability Experiment," Proc. of the 1987 Fall Joint Computer Conference, pp. 65−72, Dallas, Texas, Oct (25−29) 1987.
273. Yoshida, H., H. Kato, and M. Sugimoto, "Retrieval of Software Module Functions using first−order Predicate Logical Formulae," in Logic Programming '85", Proc of 4th Conf, ed. E. Wada, pp. 117−127, LNCS 221, Springer, Tokyo, Jul 1985.
274. Zdonik, S. B. and P. Wegner, "A Database Approach to Languages, Libraries and Environments," Technical Report No CS−85−10, pp. 23, Brown University, Providence, May 1985.
275. Zeil, S. J. and E. C. Epp, "Interpretation in a Tool− Fragment Environment," Proc. of the 10th Int. Conf. on Software Engineering, pp. 241−248, Singapur, Mar (11−15) 1988.