1.1.1 Softwareentwicklung

Die professionelle Entwicklung großer Softwaresysteme wird in der Softwaretechnik unter verschiedenen Aspekten betrachtet (Floyd & Züllighoven, 1999): als Prozess der Formalisierung von Anforderungen, als ingenieurtechnische Konstruktionsaufgabe oder als sozialorientierte Gestaltungsaufgabe. Für CSCW ist vor allem der letztgenannte Gesichtspunkt wichtig: Wie lässt sich interaktive Software menschengerecht als Arbeitsmittel und Kommunikationsmedium gestalten. Im weiteren bezeichnen wir diese Sicht als anwendungsorientierte Softwareentwicklung (Züllighoven, 1998). Folgende Merkmale von Software werden betont:

- Die Funktionalität des Softwaresystems orientiert sich an den Aufgaben im Anwendungsbereich.
- Die Handhabung des Softwaresystems ist benutzergerecht.
- Die in der Software festgelegten Abläufe lassen sich problemlos an die jeweilige Anwendungssituation anpassen.

1.1.1.1 Grundbegriffe

Anwendungsorientierte Softwareentwicklung ordnet sich in eine aktuelle Strömung der Softwaretechnik ein (Floyd, 1994), bei der Software als Mittel um Zweck betrachtet wird (Meyer, 1997, Jacobson, et al. 1992). Zum Verständnis des Themas sind einige Grundbegriffe notwendig:

Softwareentwicklung

Softwareentwicklung bezeichnet nach Floyd und Züllighoven (1999) die Gesamtheit aller Aktivitäten, die zu einem Softwaresystem im Einsatz führen. Diese Aktivitäten beziehen sich auf das Produkt Software (z.B. Analyse, Entwurf und Programmierung) oder den Entwicklungsprozess (z.B. Planung und Organisation eines Projektes).

Anwendungssoftware

Anwendungssoftware ist aus anwendungsorientierter Sicht immer ein Mittel, um fachliche Aufgaben in einem oder mehreren Anwendungsbereichen zu erledigen. Zunehmend werden dabei auch Anwendungsbereiche jenseits der Arbeitswelt betrachtet (z.B. Freizeit, Kultur, Wissens- und Erfahrungsaustausch).
Anwendungssoftware modelliert einen Ausschnitt der realen Welt und orientiert sich an einem Einsatzkontext. Sie wird auf einer Systembasis realisiert, die aus Hardware und Basissoftware besteht.
Interaktive Anwendungssoftware wird auf Großrechnern im Wechsel von Auswahlmenüs und Datenbearbeitung in sog. Bildschirmmasken realisiert. Auf Arbeitsplatzrechnern und PCs finden wir sog. reaktive Software: Programmereignisse werden durch Aktionen der Benutzer oder externer Geräte ausgelöst und in einem Ereignis-Reaktions-Zyklus interpretiert. Dabei ist *benutzergesteuerte Software* sozial eingebettet während *technisch eingebettete Software* vorrangig technische Anlagen unter expliziter Modellierung zeitlicher Randbedingungen steuert.

Gebrauchsqualität von Software

Eng mit dem Begriff Gebrauchstauglichkeit der Software-Ergonomie ist der Begriff Gebrauchsqualität der Softwaretechnik verwandt. Sie wird anhand äußerer Qualitätsmerkmale (Meyer, 1997) durch die Benutzer und andere Anwendergruppen letztlich im tatsächlichen Einsatz festgestellt.
Gebrauchsqualität kann nach DIN 66234 anhand von Aufgabenangemessenheit, Transparenz, Steuerbarkeit, Fehlertoleranz, Selbstbeschreibungsfähigkeit, Erwartungskonformität und Fehlerrobustheit bewertet werden.

1.1.1.2 Kontexte der Anwendungssoftwareentwicklung

Entwicklung und Einsatz von Anwendungssoftware wird durch unterschiedliche Kontexte bestimmt. Für die Beteiligten ist es wichtig, diese Kontexte zu verstehen, um realistische Anforderungen an Software zu stellen.

Besonderheiten von Software

Vielfach wird Software mit anderen Industrie- oder Ingenieurprodukten verglichen. Dies trifft nur begrenzt zu, da Software spezifische Eigenschaften aufweist (Brooks, 1987, Keil-Slawik, 1992). Dazu zählt:

- *Software besteht aus Sprache und symbolischen Notationen* der sie definierenden Texte. Sie ist daher fast beliebig „formbar“. Andererseits ist Software nur über die Erprobung des bereits existierenden Produktes erfahrbar.
- *Software ist digital*, d.h. die Verfahren der traditionellen kontinuierlichen Ingenieurmathematik greifen nicht, was ihre formale Beweisbarkeit sehr einschränkt.
- *Software zeigt Fehler* in unvorhersehbaren Situationen. Diese Fehler können nicht durch präventive Wartung verhindert werden.
- *Software ist heute oft so komplex,* dass sie zwar schrittweise konstruiert aber nicht mehr detailliert verstanden werden kann.

Spezifikation und Formalisierung von Software

Für die Beantwortung der Frage, in welchem Umfang Software anhand von (schriftlichen) Spezifikationen entwickelt werden kann, unterscheidet Lehman (1980) zwischen Spezifikationsprogrammen (S-Programme), problemlösenden Programmen (P-Programme) und eingebetteten Programmen (E-Programme). Für S-Programme existiert eine vollständige und formale Spezifikation, die die Aufgabenstellung und ihre prinzipielle Lösung beschreibt (z.B. Sinusberechnungen oder die Fibonacci-Zahlen). S-Programme lassen sich formal auf die Übereinstimmung mit ihrer Spezifikation prüfen. P-Programme lassen sich in ihren funktionalen Anforderungen ebenfalls vollständig formalisieren (z.B. das Schachspiel) und damit auch überprüfen. Handhabbarkeit und Angemessenheit lassen sich nur nach den Kriterien der Gebrauchsqualität durch ihre Benutzer prüfen.
E-Programme sind in einen sozialen Kontext eingebettet. Schon die Spezifikation des „Problems“, das durch ein E-Programm gelöst werden soll, ist ein sozialer Prozess. Ebenso hängt die sinnvolle Funktionalität der Software von den beteiligten Personen und ihren Aufgaben ab. Korrektheit im mathematischen Sinne kann für diese Art von Programmen nicht nachgewiesen werden. Anwendungssoftware im Bereich CSCW kann durchgängig als E-Programme klassifiziert werden.

Zu dieser Einteilung passen die Ergebnisse von Wegner (1997). Er geht davon aus, dass interaktive Software durch ihre nicht-deterministische Benutzereingaben formal so komplex ist, dass sie im Sinne von Gödel nicht mehr als korrekt beweisen werden kann. Wegner schlägt deshalb vor, interaktive Programme „einzuzäumen“, das heißt, mit Schutz- und Kontrollmechanismen zu versehen. Konstruktionstechniken auf der Basis von Zusicherungen etwa im Rahmen des Vertragsmodells (Meyer, 1997) sind hier ein erster Schritt.

Kooperation und Verteilung

Anwendungsorientierte Software kann heute kaum noch als isolierte Arbeitsplatzsoftware entwickelt werden. Die Unterstützung von Kooperation führt unmittelbar in den Bereich verteilter Software. Hier wird oft über Transparenz gesprochen.
Interessanterweise ist der Begriff *Transparenz* doppeldeutig, je nachdem ob er technisch oder anwendungsfachlich verwendet wird. Der technische Begriff „Verteilungstransparenz“ bedeutet, dass die Verteilung von Komponenten aus Sicht der Anwendung und auch der Anwendungsentwicklung möglichst unsichtbar (im Sinne von „durchsichtig“) sein soll.
Im CSCW-Kontext wird von Transparenz meist im anwendungsfachlichen Sinne gesprochen. Verteilung bei der Kooperation soll sichtbar also durchschaubar sein soll. Die Beteiligten sollen ein nachvollziehbares Bild von begrenzten Ressourcen und ihrer räumlichen Verteilung erhalten. In diesem Bild sollen die konkurrierenden und kooperierenden Benutzer erkennbar sein. Dies führt zur Forderung nach einem expliziten Benutzungs- und Kooperationsmodell. Ein *Benutzungsmodell* beschreibt aus fachlicher Sicht die Handhabung und Präsentation der Software mit den darin repräsentierten Gegenständen, Konzepten und Abläufen. Ein *Kooperationsmodell* verdeutlicht die im Benutzungsmodell vorhandene Unterstützung von Kooperation und Koordination.

Von der Kundenorientierung zur Anwendungsorientierung

Kundenorientierung ist heute für viele Unternehmen zur Geschäftsstrategie geworden. Ein Unternehmen soll sich kontinuierlich bemühen, die Kunden besser zufrieden zu stellen, um zu langfristig an das Unternehmen zu binden und damit den *Unternehmenserfolg* sicherzustellen. Wenn auf dieser Basis Arbeitsprozesse, Organisationsstrukturen und das Produkt- oder Dienstleistungsangebot kritisch hinterfragt werden, stellt sich die Frage, wie Anwendungssoftware aussehen muss, die Kundenorientierung umzusetzen hilft. Anwendungsorientierung soll den Benutzern von Anwendungssoftware die Mittel an die Hand geben, mit denen diese selbst kundenorientiert arbeiten können.

1.1.1.3 Konzepte der Anwendungsorientierung – produktbezogen

Interaktive Anwendungssoftware muss eine *aufgabengerechte Funktionalität* mit einer *geeigneten Handhabung und Präsentation* verbinden (Züllighoven, 1998). Dies ist eine typische Designaufgabe: Form und Inhalt müssen zueinander passen. Dies geht weit über die traditionelle softwaretechnische Aufgabe hinaus, einen geeigneten Algorithmus zur Erledigung einer fachlichen „Funktion“ zu finden und zu implementieren. Softwareentwickler müssen bei der Frage unterstützt werden, welche „Gestalt“ ein Softwareprodukt annehmen und wie es bei der Aufgabenerledigung gehandhabt werden soll. Vorgeschlagen werden Leitbilder und Entwurfsmetaphern.

Leitbilder

Spätestens seit Anfang der 90er-Jahre wird der allgemeine Begriff des Leitbilds in seiner Bedeutung für die Softwareentwicklung diskutiert (Müller und Senghaas-Knobloch 1993, Mambrey et al. 1995). Dabei ist ein Leitbild eine orientierende Sichtweise in der Softwareentwicklung, die den Beteiligten hilft, Anwendungssoftware zu entwerfen, zu verstehen und zu bewerten. Allgemein diskutierte Leitbilder sind z.B. die „Softwarefabrik“ und das „papierlose Büro“. Die Leitbildidee in der Softwareentwicklung ist aber viel älter. So gehen die Wurzeln des Personalcomputers auf die Vorstellungen von intelligenten oder intelligenzverstärkenden Maschinen in den Arbeiten von V. Bush (1945), D. Engelbart (1994) und A. Kay (1977) zurück.

Entwurfsmetaphern

Die Idee, Metaphern für den Entwurf von Anwendungssoftware einzusetzen, ist ebenfalls nicht neu (Carroll et al., 1988, Mambrey et al. 1995). Insbesondere in Skandinavien gibt es eine lebhafte Diskussion zu diesem Thema (Ehn, 1988, Madsen, 1988). *Entwurfsmetaphern* beschreiben ein Konzept oder eine Element des Anwendungssystems durch einen bekannten Gegenstand der Alltagswelt – Fenster, Ordner, Schreibtisch, Agent. Eigenschaften des ursprünglichen Gegenstands werden dabei in den Kontext des Softwaresystems übertragen. So schaffen Entwurfsmetaphern eine Sprache, um über die Elemente eines Softwaresystemen zu reden und sie lenken die Gestaltung gezielt in eine Richtung. Den Zusammenhang von Entwurfsmetaphern und Leitbildern oder Sichtweisen diskutieren Maaß und Oberquelle (1992). Ein Beispiel für die Verwendung von Entwurfsmetaphern und Leitbildern für die anwendungsorientierte Softwareentwicklung ist der Werkzeug & Material-Ansatz (Züllighoven, 1998).

Strukturähnlichkeit

Strukturähnlichkeit ist ein Gestaltungsprinzip für Software; es bezeichnet das Verhältnis von Software und Anwendungsbereich. Schon die Designer der ersten objektorientierten Programmiersprache Simula 67 haben die Klassen ihrer Software anhand von Gegenständen aus dem Anwendungsbereich modelliert. Diese Ähnlichkeit „im Kleinen“ ist zum durchgängigen Prinzip vieler objektorientierter Vorgehensweisen geworden (Meyer, 1997, Booch et al., 1999, Züllighoven, 1998). Bäumer (1998) hat gezeigt, dass sich dieses Prinzip auch auf die Architektur großer Softwaresysteme übertragen lässt.

1.1.1.4 Konzepte der Anwendungsorientierung – prozessbezogen

Für die Entwickler bedeutet Anwendungsorientierung, dass sie die Aufgaben verstehen müssen, die sie durch entsprechende Software unterstützen sollen. Dazu müssen sie sich den Zugang zum Fachwissen und zur Erfahrung der Anwendungsexperten verschaffen. Das Schlüsselwort heißt hier „evolutionäre Systementwicklung“, da diese auf eine enge Zusammenarbeit der Entwickler mit den Anwendern ausgerichtet ist. Zusammenarbeit muss aber eine Grundlage haben. Mit der UML (Booch et al., 1999) als einer vereinheitlichten Modellierungssprache im Bereich objektorientierter Softwareentwicklung und dem Unified Process (Jacobson et. al, 1999) ist es allgemein akzeptierter Stand der Technik, Software-

projekte auf der Basis von Dokumenten durchzuführen, die jeweils unterschiedliche Sichten auf das System zulassen.

Anwendungsorientierte Dokumenttypen

Anwendungsorientierte Dokumenttypen als notwendige Basis zum Verständnis der Konzepte und Aufgaben im Anwendungsbereich werden seit einiger Zeit diskutiert (Carroll and Rosson 1990, Jacobson et al., 1992). Für die Integration der Anwendungsexperten in den Entwicklungsprozess müssen aber wesentliche Dokumente in der Fachsprache des Anwendungsbereichs formuliert werden. In den verschiedenen methodischen Ansätzen werden dazu Vorschläge gemacht – Szenarios, Glossare, Use Cases, Rich Pictures, Kooperationsbilder. Wichtig ist vor allem, dass sie eine anwendungsfachliche Sicht spiegeln und als Arbeitsgrundlage sowohl für Entwickler als auch für Anwender dienen können.

Lernprozesse und Rückkopplung

Anwendungsorientierte Softwareentwicklung sollte nicht als eine vorrangig technische oder formale Aufgabe, sondern als einen Lern- und Kommunikationsprozess betrachtet werden (Floyd und Züllighoven, 1999). Daher ist die Projektarbeit so auszurichten, dass Lernen und Kommunikation durch ständige Rückkopplung zwischen den Beteiligten gefördert werden. Durch die Arbeit mit anwendungsorientierten Dokumenten haben die Beteiligten eine Kommunikationsgrundlage, auf der ein gegenseitige Verständnis für den Anwendungsbereich und die Möglichkeiten seiner softwaretechnischen Unterstützung entsteht. Im zyklischen Wechsel zwischen Analysieren, Modellieren und Bewerten kann zudem sichergestellt werden, dass sich Anforderungen an ein System und dessen Realisierung nicht zu weit von einander entfernen.

Evolutionäre Vorgehensweise mit Prototyping

Die Verknüpfung von analysierenden, modellierenden und bewertenden Aktivitäten steht erkennbar im Widerspruch zu den Prinzipien der klassischen Wasserfall- oder Phasenmodelle. Dort soll eine festgelegte Folge von Meilensteindokumenten sequenziell abgearbeitet werden. Dies ist vielfach kritisiert worden. Der Unified Process beschreibt aktuell, wie zyklische und iterative Entwicklungsprozesse dokumentgetrieben gestaltet werden können. Der Einsatz von anwendungsorientierten Dokumenttypen sollte aber durch Prototyping ergänzt werden. Erst die Konstruktion und Bewertung von ablauffähigen Prototypen hilft den Anwendern, die Gebrauchsqualität des zukünftigen Systems einzuschätzen und den Entwicklern, die technische Realisierbarkeit sicherzustellen.
Üblich ist die Einteilung in *exploratives, experimentelles und evolutionäres Prototyping* nach Floyd (1984). Dabei werden unterschiedliche Prototypen zur Beantwortung der jeweils aktuellen Fragestellungen eingesetzt, z.B. *Demonstrationsprototypen, funktionelle Prototypen* und *Pilotsysteme* (Kieback et al., 1992).

1.1.1.5 Modellierung anwendungsorientierter Software

Aus softwaretechnischer Sicht kann Softwareentwicklung als Modellierungsprozess verstanden werden. Offenkundig wird dabei ein Modell des Softwaresystems schrittweise in eine ablauffähige Form gebracht wird. Im Software Engineering ist aber erst allmählich die

Einsicht gewachsen, dass dazu ein explizites Modell des Anwendungsbereichs die *logische* Voraussetzung ist. Dieses Modell sollte nicht auf die üblichen Systemanforderungen in einem sog. Pflichtenheft reduziert werden.

Modell des Anwendungsbereichs

Das Modell des Anwendungsbereichs umfasst diejenigen Aspekte des Anwendungsbereichs, die durch ein Anwendungssystem unterstützt werden sollen. Es sollte die relevanten Aufgaben, Abläufe und Begriffen aus Anwendungssicht darstellen. Dazu werden anwendungsorientierte Dokumenttypen wie Szenarios, Business Use Cases und Glossare eingesetzt. Sie werden z.B. auf der Basis von Anwenderinterviews und gemeinsamen Workshops erarbeitet. Dabei können auch ethnografische Methoden eingesetzt werden.

Modell des Anwendungssystems

Im Modell des Anwendungssystems werden nach dem Prinzip der Strukturähnlichkeit die wesentlichen Konzepte des Modells des Anwendungsbereichs aufgegriffen. Diese werden um weitere Aspekte der Konstruktion ergänzt. Aus Sicht der Anwendungsorientierung müssen folgende Fragen beantwortet werden:

- Welche Aufgaben, Abläufe und Gegenstände des Anwendungsbereichs sollen im Anwendungssystem konzeptionell realisiert werden?
- Wie soll sich das Anwendungssystem im Benutzungsmodell repräsentieren und wie soll es zu handhaben sein?
- Welche Technologie kommt dabei zum Einsatz?

Der anwendungsorientierte Modellierungsprozess ist scheinbar widersprüchlich: Ein Anwendungssystem kann nur auf der Basis eines fachliches Modell des entsprechenden Anwendungsbereichs entwickelt werden. Für dieses fachliche Modell muss der Rahmen des zukünftigen Anwendungssystems bekannt sein. Dieser Widerspruch löst sich aber durch eine zyklische Vorgehensweise, bei der im Wechsel an beiden Modellen gearbeitet wird.

1.1.1.6 Konstruktion anwendungsorientierter Software

Bei der Entwicklung anwendungsorientierter Software sind heute durchgängig bestimmte Konstruktionstechniken festzustellen (Floyd, Züllighoven, 1998).

Objektorientierung

Objektorientierung ist für die anwendungsorientierte Softwareentwicklung auch dann von großer Bedeutung, wenn keine rein objektorientierte Programmiersprache eingesetzt wird. Das Programmiermodell (Meyer, 1997) erleichtert Konstruktion interaktiver Software und fördert die Strukturähnlichkeit.

Objektorientierte Konstruktion beruht auf dem Prinzip der abstrakten Datentypen: Klassen beschreiben durch ihre an der Schnittstelle sichtbaren Operationen das Verhalten der Objekte, die zur Laufzeit dynamisch erzeugt werden können. Gekapselt und damit verborgen werden die Implementierungen dieser Operationen und die objektspezifischen Attribute, die den veränderbaren Zustand jedes Objekts ausmachen. Klassenbeschreibungen lassen sich hierarchisch durch den Vererbungsmechanismus anordnen, wobei ein Oberklasse das Verhalten und die Strukturmerkmale ihrer Unterklassen vorzeichnet. Jede Unterklasse hat aber die Möglichkeit, diese Vorgaben zu erweitern oder zu modifizieren.

die Möglichkeit, diese Vorgaben zu erweitern oder zu modifizieren. So lassen sich Begriffshierarchien des Anwendungsbereichs auf Programmeinheiten abbilden. Im laufenden System verschicken Objekte Botschaften, um die Dienstleistungen oder Services der anderen Objekte in Anspruch zu nehmen. Die Zuordnung von Botschaft und Operationsimplementierung (Bindung) muss im Programmtext nicht statisch festgelegt werden. Es genügt sicherzustellen, dass die Objekte die Botschaft verstehen. Objektorientierte Systeme lassen sich nach dem Offen-Geschlossen-Prinzip (Meyer, 1998) flexible weiterentwickeln und durch dynamisches Linken um Klassen erweitern.

Entwurfsmuster

Objektorientierte Programmierung ermöglicht durch die Vererbung zwischen Klassen die Wiederverwendung von Programmcode. Entwurfsmuster beschreiben dagegen eine allgemeine Lösungsidee für ähnliche Problemstellungen. Sie können auf unterschiedlichen Detaillierungsebenen angesiedelt werden. Kennzeichnend ist, dass jedes Muster benannt und nach einem einheitlichen Schema (z.B. Problem – Kontext – Lösung – Diskussion) beschrieben ist. Weit verbreitet sind objektorientierte Muster, die ein Entwurfsproblem durch eine Konfiguration von Klassen oder Objekten lösen. Bekannte Entwurfsmuster sind Model-View-Constroller, das dem Smalltalk-System zu Grunde liegt, und das Beobachter-Muster, das oft zur losen Kopplung von grafischen Oberflächen und anwendungsfachlichen Klassen dient. Unter den verschiedenen Mustersammlungen ist Gamma et al. (1996) sicherlich die derzeit verbreitetste.

Rahmenwerke

Rahmenwerke oder Frameworks ermöglichen die Wiederverwendung von (objektorientierten) Softwarearchitekturen und generischen Anwendungsteilen (Johnson and Foote, 1988). Während Entwurfsmuster nur eine Lösungsidee beschreiben, sind Rahmenwerke sozusagen Halbfertigprodukte, die durch Programmteile an festgelegten Schnittstellen ergänzt werden müssen. Dies kann durch Programmierung oder durch Parametrisierung mit vorgefertigten Bausteinen geschehen. Rahmenwerke unterscheiden sich von Programmbibliotheken dadurch, dass sie den Kontrollfluss eines Programms festlegen. Wenn größere Teile einer Anwendung und evtl. die Benutzungsschnittstelle in ihrem sog. Look and Feel vorgegeben sind, spricht man auch von einem Anwendungsrahmenwerk (application framework) (Communications of the ACM, 1997). Der Einsatz und die Entwicklung von Rahmenwerken stellt heute noch hohe Anforderungen an die softwaretechnische Qualifikation der Entwickler.

Komponenten

Die Wiederverwendung einzelner vorgefertigter Bausteine steht hinter der Idee der sog. Komponenten (Nierstrasz et al., 1992, Szyperski, 1997). Dabei sollen einsatzfertige, möglichst in binärer Form vorliegende Softwareeinheiten ausgewählt und durch Komposition zu ganzen Anwendungen zusammengesetzt werden. Das Zusammenspiel der einzelnen Komponenten und die evtl. noch nötige Anpassung soll durch einfache Programmzwischenstücke (Glue Code) geleistet werden. Dieses Konzept ist vor allem bei VisualBasic und COM/DCOM erfolgreich. Aktuell hat die steigende Verbreitung von Java mit den bei-

den Komponentemodellen Java Beans und Enterprise Java Beans die Diskussion sehr belebt. Im Rahmen von CORBA wird ebenfalls an einem Komponentenmodell gearbeitet.

Bäumer, D. (1998), *Softwarearchitekturen für die rahmenwerkbasierte Konstruktion großer Anwendungssysteme*. Disserationsschrift zur Vorlage am Fachbereich Informatik der Universität Hamburg.

Booch, G., Rumbaugh, J., and Jacobson, I. (1999), *The Unified Modeling Language User Guide*, Addison-Wesley: Reading.

Brooks, F.P. (1987), No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20, 4, 10-19.

Communications of the ACM (1997), 40, 10.

Bush, V. (1945), As We May Think. *Atlantic Monthly*, 176, 1, 101-108.

Carroll, J. M., Mack, R. L., and Kellogg, W. A. (1988), Interface Metaphors and User Interface Desig*n*. In Helander, M. (Ed.) *Handbook of Human-Computer Interaction*, 283–307.

Carroll, J. M. and Rosson, M. B. (1990), Human Computer Interaction Scenarios as Design Representation. *Proceedings of the Hawaii International Conference on System Sciences*, IEEE Computer Society Press: Los Alamitos CA, 555-561.

Ehn, P. (1988), *Work–oriented Design of Computer Artifacts*. Almquist and Wiksell International: Stockholm.

Engelbart, D., English, W. (1994): A Research Center for Augmenting Human Intellect, Re-printed in ACM SIGGRAPH Video Review, Original 1968.

Floyd, C. (1984), A Systematic Look at Prototyping. In: Budde, R.; Kuhlenkamp, K.; Mathiassen, L.; Züllighoven, H. (Eds.), *Approaches to Prototyping*, Springer: Berlin, 1-18.

Floyd, C (1994): Software-Engineering - und dann? *Informatik Spektrum*, 17, 1, 29-37.

Floyd, C. und Züllighoven, H. (1999), Softwaretechnik. In: P. Rechenberg, G. Pomberger (Hrsg.), *Informatik Handbuch*. 2. Auflage, Carl Hanser Verlag: München, Wien, S. 763–790.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1996) *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software.* Addison Wesley: Bonn.

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1992), *Object-oriented Software Engineering. A Use Case Driven Approach*. Addison-Wesley: Reading.

Meyer, B. (1997), *Objekt-oriented software construction*. Second Edition, Prentice Hall: New York.

Jacobson, I, Booch, G., and Rumbaugh, J. (1999), *The Unified Software Development Process*, Addison-Wesley: Reading.

Johnson, R. and Foote, B. (1988) Designing reusable classes. *Journal of Object-Oriented Programming* 1, 2, 22-35.

Kay, A. (1977), Microelectronics and the Personal Computer, Scientific American, 237, 3, 230-244.

Keil-Slawik, R. (1992), Artifacts in Software Design. In: Floyd, C., Züllighoven, H., Budde, R., Keil-Slawik, R. (Hrsg.), *Software Development and Reality Construction*, Springer Verlag: Berlin, 168-188.

Kieback, A., Lichter, H., Schneider-Hufschmidt, M., and Züllighoven, H. (1992), Prototyping in industriellen Software-Projekten, Informatik-Spektrum, 15, 2, 65-78.

Lehman, M.M. (1980), Programs, Life Cycles, and Laws of Software Evolution. *Proc. of IEEE*, 68, 9, 1060-1076.

Maaß, S. and Oberquelle, H. (1992), Perspectives and metaphors for Human-Computer-Interaction. In: Floyd, C.; Budde, R.; Keil–Slawik, R.; Züllighoven, H. (Eds.): *Software Development and Reality Construction*, Springer: Heidelberg, 233 – 251.

Madsen, K.H. (1988), Breakthrough by breakdown: Metaphors and structured domains. In: Klein, H.K.; Kumar, K. (eds.), *Information Systems Development for Human Progress in Organizations*. North-Holland: Amsterdam.

Mambrey, P., Paetau, M. und Tepper, A. (1995), *Technikentwicklung durch Leitbilder. Neue Steuerungs- und Bewertungsinstrumente*. Campus: Frankfurt.

Meyer, B. (1997), *Object-oriented software construction*. Second Edition, Prentice Hall: New York.

Müller, W., Senghaas-Knobloch, E. (Hrsg.) (1993). *Arbeitsgerechte Technikgestaltung*. Lit: Münster, Hamburg.

Nierstrasz, O., Gibbs, S., and Tsichritzis, D. (1992) Component-Oriented Software Development. *Commmunications of the ACM*, 35, 9.

Szyperski, C. (1997); Component Software, Addison-Wesley: Reading.

Wegner, P. (1997), Why Interaction Is More Powerful Than Algorithms, *Communications of the ACM*, 40, 5, 80-91.

Züllighoven, H.(1998), *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz*. dpunkt-Verlag: Heidelberg.