Zeitschrift: Bulletin des Schweizerischen Elektrotechnischen Vereins, des Verbandes Schweizerischer Elektrizitätsunternehmen = Bulletin de l'Association suisse des électriciens, de l'Association des entreprises électriques suisses

Herausgeber: Schweizerischer Elektrotechnischer Verein ; Verband Schweizerischer Elektrizitätsunternehmen

Band: 89 (1998)

Heft: 17

Artikel: Komponentensoftware : oder wie die Informatik doch noch zu einer Ingenieurdisziplin wird

Autor: Pfister, Cuno

DOI: https://doi.org/10.5169/seals-902099

Nutzungsbedingungen

Die ETH-Bibliothek ist die Anbieterin der digitalisierten Zeitschriften auf E-Periodica. Sie besitzt keine Urheberrechte an den Zeitschriften und ist nicht verantwortlich für deren Inhalte. Die Rechte liegen in der Regel bei den Herausgebern beziehungsweise den externen Rechteinhabern. Das Veröffentlichen von Bildern in Print- und Online-Publikationen sowie auf Social Media-Kanälen oder Webseiten ist nur mit vorheriger Genehmigung der Rechteinhaber erlaubt. Mehr erfahren

Conditions d'utilisation

L'ETH Library est le fournisseur des revues numérisées. Elle ne détient aucun droit d'auteur sur les revues et n'est pas responsable de leur contenu. En règle générale, les droits sont détenus par les éditeurs ou les détenteurs de droits externes. La reproduction d'images dans des publications imprimées ou en ligne ainsi que sur des canaux de médias sociaux ou des sites web n'est autorisée qu'avec l'accord préalable des détenteurs des droits. En savoir plus

Terms of use

The ETH Library is the provider of the digitised journals. It does not own any copyrights to the journals and is not responsible for their content. The rights usually lie with the publishers or the external rights holders. Publishing images in print and online publications, as well as on social media channels or websites, is only permitted with the prior consent of the rights holders. Find out more

Download PDF: 14.07.2025

ETH-Bibliothek Zürich, E-Periodica, https://www.e-periodica.ch

Softwarekomponenten sollten sich ohne Aufwand (Plug & Play) zu robusten und effizienten Anwendungen kombinieren lassen. Auch wenn die heutige Situation von einem solchen Idealzustand noch weit entfernt ist, treibt der Kostendruck die Softwareentwicklung in diese Richtung. Der vorliegende Beitrag befasst sich – ausgehend vom derzeitigen Entwicklungsstand – mit den Anforderungen, welche an den Entwickler von Softwarekomponenten gestellt werden, mit den Standardisierungsanstrengungen der Softwarehäuser, mit der Schnittstellenproblematik sowie mit der wichtigen Rolle der Programmiersprachen und des Softwaredesigns. Der Autor weist zudem auf die intakten Chancen der Schweizer Komponentenindustrie hin, welche dank Erfahrung und Anwendungswissen in verschiedenen Branchen (Beispiel Werkzeugmaschinenbau) bei der Softwareintegration eine hervorragende Stellung einnimmt und auch in Zukunft einnehmen wird, vorausgesetzt, dass der Heimmarkt mitzieht.

Komponentensoftware – oder wie die Informatik doch noch zu einer Ingenieurdisziplin wird

■ Cuno Pfister

Wenn ein Elektrotechniker oder ein Maschinenbauer ein Produkt konstruiert, dann muss er nicht bei null anfangen, sondern kann auf ein reichhaltiges Angebot an Marktkomponenten zurückgreifen. Das Angebot reicht von DIN-konformen Schrauben bis zu komplexen Maschinenanlagen, von einfachen TTL-Bausteinen bis zu komplexen Mikroprozessoren. Für die wichtigsten Komponenten gibt es Standards, die von offiziellen DIN-Normen bis zu De-facto-Standards wie zum Beispiel den Bus-Interfaces von Intel-Mikroprozessoren reichen. Diese Standards erlauben verschiedenen Anbietern, die gleiche Funktionalität in Konkurrenz zueinander anzubieten. Der Kunde kann so aus verschiedenen Angeboten das für ihn beste auswählen.

Auf dem Weg zur reifen Ingenieurdisziplin

Im Vergleich zu älteren Ingenieurdisziplinen hat die Informatik noch keinen vergleichbaren Entwicklungsstand erreicht. Märkte für Softwarekomponenten werden zwar schon seit 1967 gefordert, entstehen aber tatsächlich erst jetzt in nennenswertem Umfang. Für die Informatik markiert diese aktuelle Entwicklung nichts weniger als den Übergang zu einer reifen Ingenieurdisziplin. Es geht eigentlich um eine Verallgemeinerung des klassischen Software-Engineerings: Softwareteile (Komponenten) sollen so

Adresse des Autors
Dr. *Cuno Pfister*, Oberon Microsystems AG
Technopark Zürich, 8005 Zürich
Email pfister@oberon.ch

konstruiert werden, dass sie potentiell auch als binäre Produkte (d.h. ohne Quellcode) auf den Markt gebracht und vom Kunden zu minimalen Kosten integriert werden können. Neue Komponenten sollen zu existierenden Systemen hinzugefügt und existierende Komponenten individuell ersetzt werden können, ohne dass dadurch der Rest des Systems tangiert wird.

Damit wird Komponentensoftware nicht nur zu einem Ansatz, mit dem neue Software konstruiert werden kann, sondern auch zu einem Prinzip, das erlaubt, mit *Legacy-Software* («ererbte» Software, d.h. alte installierte Softwarebasis) systematischer und effektiver als bisher umzugehen. Der Lebenszyklus eines Softwaresystems kann durchaus länger werden als derjenige einer einzelnen Komponente des Systems. Meistens ist es aufgrund der bereits erfolgten Investitionen nötig, existierende Legacy-Software in Komponenten zu verpacken, man spricht hier von «Legacy Wrapping». Solche nie für eine Wiederverwendung gedachten «Komponenten» wird man zwar im allgemeinen nicht nachträglich selbst in Komponenten zerlegen können, trotzdem sind sie für einen schrittweisen Übergang zu komponentenbasierten Lösungen unerlässlich. Legacy Wrapping ist also eine legitime und absolut notwendige Tätigkeit, welche jedoch die langfristigen Perspektiven und Möglichkeiten der Komponentensoftware oft verdeckt. Dies birgt die Gefahr in sich, dass heute lediglich die Legacy-Software von morgen konstruiert wird.

Analogien zur konventionellen Technik sind irreführend

Die Entwicklung hin zu Komponentensoftware ist spannend und letztlich unvermeidlich, da es dabei gleichzeitig um ein grundlegendes Ingenieurprinzip (teile und herrsche) sowie auch um ein grundlegendes Prinzip der Marktwirtschaft, das Prinzip der Arbeitsteilung geht (betriebsübergreifende Wertschöpfungsketten). Bei der Komponentensoftware handelt es sich also um mehr als eine bloss nützliche neue Technologie – wie etwa im Falle der objektorientierten Programmierung. Sie löst aber auch nicht alle Probleme der Informatik. Zudem sind die für die Komponentensoftware nötigen Technologien und Methoden noch unreif; es gibt noch viele Fallstricke und Sackgassen, leere Versprechungen, Missverständnisse und enttäuschte Erwartungen. Dies rührt nicht zuletzt daher, dass Analogien zum Maschinenbau oder zur Elektrotechnik irreführend sind. Eine Softwarekomponente entspricht in der physischen Welt eher dem Bauplan einer Fabrik voller Produktionsanlagen als einem Objekt, das von einer solchen Anlage produziert wird. Deshalb entspricht eine Softwarekomponente nicht einem Objekt im Sinne der objektorientierten Programmierung, sondern eher einer Dynamic Link Library (Fabrik), bestehend aus einer oder mehreren Klassen (Produktionsanlagen). Im Gegensatz zur physischen Welt ist dann die Produktion von Objekten beim Kunden vor Ort möglich und praktisch kostenlos [1]. Diese Bemerkung ist nur ein Hinweis auf das konzeptionelle und terminologische Chaos, das in diesem Bereich noch herrscht.

Standardisierung – eine Herausforderung

Softwarekomponenten verschiedener Anbieter können auf unterschiedliche Weise zu einer neuen Lösung integriert (komponiert) werden. Beispielsweise gibt es auch heute noch aufwendige internationale Standardisierungsbemühungen, die lediglich auf die Spezifikation eines Datenschemas und Fileformates für den Datenaustausch hinauslaufen. Als Beispiel sei der Step-Standard (ISO 10303) im Bereich der Werkzeugmaschinen-Informatik genannt. Diese Offline-Integration von Softwarepaketen ist die rudimentärste Art, getrennt entwickelte Softwarepakete zu integrieren.

Dass dieser Ansatz mit seinem Hin- und Hertransportieren von Datenfiles unbefriedigend ist, wurde schon lange erkannt. Heute wird deshalb – was als Schritt in die richtige Richtung zu werten ist – vermehrt versucht, Netzwerkprotokolle und -Services zu definieren, die den Datentransfer von der Ebene der untypisierten Byte-Ströme auf die Ebene der typisierten Datenstrukturen anheben. Dadurch kann man sich viel Arbeit ersparen und Fehlerquellen bei der Datenkonversion ausschliessen. Ein Beispiel für eine Standardisierungsbemühung nach diesem Ansatz, auch wieder aus dem Bereich der Werkzeugmaschinen-Informatik, ist der europäische Osaca-Standard (Open System Architecture for Controls Association).

Anwendungs- und Plattformunabhängigkeit

Es ist nicht sinnvoll, wenn in jedem Anwendungsbereich die Infrastruktur für die Kommunikation zwischen Applikationen neu erfunden wird, wie das zum Beispiel bei Osaca noch der Fall ist. Statt dessen sollte man sich auf die Definition von (Programmier-)Schnittstellen konzentrieren, welche von der Implementierung der darunterliegenden Infrastruktur abstrahieren. Dieser Ansatz wurde mit Omac (Open Modular Architecture Controller) verfolgt, der amerikanischen Konkurrenz zu Osaca. Omac basiert auf dem Corba-Standard (Common Object Request Broker Architecture) der OMG (Object Management Group, http://www.omg.org). Dies ist ein Standard für verteilte Objekte und Legacy Wrapping, für den es eine grosse Zahl von Implementierungen verschiedener Hersteller und für verschiedene Plattformen gibt.

Die stärkere Trennung zwischen Schnittstellen und Implementierungen, die eine Infrastruktur wie Corba mit sich bringt, erlaubt auch die Verwendung von mehreren Programmiersprachen, während (Infrastruktur-)Eigenentwicklungen oft auf eine Sprache beschränkt sind, wie das zum Beispiel bei Osaca der Fall ist (C++). Eigenentwicklungen sind zudem oft auf Source-Code-Reuse beschränkt, da es zum Beispiel für C++ kein standardisiertes Objektmodell (d.h. keinen Integrationsstandard für bereits übersetzten Code) gibt.

Wenn man mit einer Infrastruktur (Middleware) für verteilte Objekte arbeitet, zum Beispiel mit Corba oder DCOM (Distributed Component Object Model, http://www.microsoft.com/com) von Microsoft, dann abstrahiert man von der geographischen Konfiguration der Objekte (Location Transparency). Eine Applikation arbeitet immer mit Objekten, ob sich diese Objekte nun im selben Prozess (d.h. Adressraum), in einem anderen Prozess auf derselben Maschine oder in einem Prozess auf einer anderen Maschine befinden. Wenn die Objekte sich nicht im gleichen Prozess befinden, dann übernehmen Platzhalter (Proxies) ihre Rolle, und Corba oder DCOM übernehmen die Kommunikation zwischen den Proxies und den realen Objektimplementierungen (Servers).

Von der Transparenz des Ortes zur In-Process-Integration

Die Transparenz des Ortes einer Objektimplementierung war lange Zeit der heilige Gral der verteilten Systeme. Inzwischen ist allerdings eine grosse Ernüchterung zu verspüren, hat sich doch die Performance von Applikationen, die ohne Rücksicht auf die Verteilung von Objekten entworfen und entwickelt wurden, in den meisten Fällen als unerträglich schlecht erwiesen. Der Grund dafür ist, dass die Kommunikation zwischen

Prozessen – oder gar über Rechnergrenzen hinweg – um mehrere Grössenordnungen langsamer ist als der direkte Zugriff auf Daten im selben Adressraum. In der Forschung setzt deshalb eine Abkehr von transparenten Objektsystemen ein, hin zu Infrastrukturen, bei denen bessere Möglichkeiten zum Eingriff und zum Tuning von Konfigurationen bestehen.

Wenn man sich reale Systeme anschaut, die zum Beispiel auf Corba basieren, dann wird es deshalb nicht überraschen, dass die erfolgreichen Systeme diese Infrastruktur bewusst als komfortables Kommunikationsvehikel für den Transfer von Daten benutzen und nicht als transparenten Zugriffsmechanismus auf Daten, die sich an beliebigen Orten befinden können.

Aus der Sicht der Anwendungssoftware ist es nun aber nicht wünschenswert, Daten zu transferieren, da dadurch Kopien der Daten entstehen, die zueinander inkonsistent werden können. Besser wäre es, die Daten in einer einzigen Komponente zu verwalten und von anderen Komponenten aus darauf zuzugreifen. Aus den genannten Effizienzproblemen folgt, dass eine solche Integration innerhalb eines Prozesses (in Process) erfolgen sollte. Diese Art der Integration von Softwarekomponenten ist mit Abstand am flexibelsten und erlaubt die bestmögliche Modularisierung. OLE for Design and Modeling (http://www.dmac.org) ist ein Beispiel für ein API, welches vorrangig die Integration zwischen Prozessen auf derselben Maschine und in Process unterstützt.

In bezug auf Infrastrukturstandards hat dies subtile Konsequenzen. Corba zum Beispiel betrachtet den verteilten Fall als Normalfall, erlaubt aber die In-Process-Integration als speziellen Fall auf herstellerabhängige(!) Art. Microsoft hat hingegen bei COM (Component Object Model) die In-Process-Integration als Normalfall betrachtet und diesen Fall deshalb stark optimiert und auf binärer Ebene standardisiert. Trotzdem können mit der DCOM-Erweiterung von COM auch entfernte Objekte transparent behandelt werden. Moderne Windows-Applikationen benutzen COM, sobald sie zum Beispiel OLE (Object Linking and Embedding) oder ActiveX-Controls unterstützen, da diese Technologien letztlich Sammlungen von COM-Schnittstellen darstellen.

Schnittstellenproblematik – ein zentrales Sicherheitsproblem

Wenn man Integration mittels kommunikationsorientierten Infrastrukturen erreichen will, so ignoriert man meistens ein zentrales Sicherheitsproblem: das Problem der Versionierung von Schnittstellen. Wenn eine Schnittstelle, zum Beispiel ein API für Motion Control, einmal publiziert worden ist, dann kann es von einer unbekannten Anzahl von Fremdkomponenten benutzt werden. Wenn man diese Schnittstelle dann ändert (syntaktisch oder semantisch), dann besteht eine grosse Wahrscheinlichkeit, dass existierende Komponenten damit nicht mehr korrekt interagieren. Konsequenterweise darf eine einmal publizierte Schnittstelle nie mehr geändert werden. Da in der Informatik aber nichts konstanter ist als die stetige Änderung von Anforderungen, muss es folglich möglich sein, dass ein Objekt gleichzeitig mehrere verschiedene Schnittstellen unterstützt. Es kann sich dabei um verschiedene Versionen desselben Services oder um sich gegenseitig ergänzende Services handeln. So implementiert etwa ein ActiveX-Control eine ganze Reihe von OLE-Schnittstellen, zum Beispiel zum Zeichnen des Controls, zum Abspeichern des Controls, zur Behandlung von Benutzereingaben usw.

Man beachte, dass die Versionierung von Schnittstellen nicht dasselbe wie die Versionierung von konkreten Komponenten ist. Ein Klient sollte sich immer nur auf eine Schnittstelle abstützen und unabhängig von deren Implementierung (und damit auch deren Version) bleiben. In bezug auf dieses Grundproblem offener Systeme ist Microsofts COM-Standard weitaus am besten durchdacht. Leider aber hat die enge Integration von COM-Komponenten in einer Microsoft-Umgebung auch Nachteile. Einerseits greifen die meisten COM-Objekte auch auf komplexe (mehr oder weniger) Windows-spezifische Services zu und sind dadurch nicht mehr voll portabel. Es gibt aber andererseits noch ein schwerwiegenderes Problem bei COM. Die In-Process-Integration von Komponenten verschiedener Hersteller führt zu einem Zuverlässigkeitsproblem: eine Komponente kann aus Versehen Daten einer anderen Komponente zerstören, da deren Daten im selben Adressraum liegen. Die Vorteile einer derart engen Integration zu erhalten ohne deswegen fragile Systeme zu schaffen, ist eine der grossen technischen Herausforderungen der Komponentensoftware-Entwicklung. Hier besteht der grösste Innovationsbedarf. Ein möglicher Ansatz wären neuartige Hardware-Schutzmechanismen, die aber in absehbarer Zukunft nicht zu erwarten sind. Als Alternative dazu bleiben sichere Programmiersprachen, das heisst Sprachen, welche die vollständige Speicherintegrität garantieren. Dies sind notwendigerweise Sprachen, die Garbage Collection unterstützen, also den Speicher automatisch verwalten. In diesem strengen Sinne sind Sprachen wie Pascal, Modula-2 oder C++ unsichere Sprachen, während Component Pascal (Oberon) und Java sicher sind. Java bietet mit *JavaBeans* die nötigen Voraussetzungen, um portable und im genannten Sinne sichere Softwarekomponenten zu entwickeln. Trotzdem hat auch Java Probleme, zum Beispiel im Bereich der Effizienz, der unausgereiften Bibliotheken und der unbefriedigenden Versionskontrolle von Schnittstellen und Komponenten.

Die Rolle der Programmiersprachen

Entgegen der landläufigen Meinung, dass die Wahl der Programmiersprache heute keine Rolle mehr spielt, stellt sichere Komponentensoftware in bezug auf Programmiersprachen also geradezu eine Wasserscheide dar, da sie ganz bestimmte Anforderungen an ein Laufzeitsystem stellt. Das war auch der Grund dafür, dass die Sprachen Oberon und Java überhaupt entwickelt werden mussten, und erklärt, wieso man praktisch dieselbe Laufzeitumgebung für beide Sprachen benutzen kann. Dies belegt das neue Echtzeitbetriebssystem JBed (http://www.oberon.ch/rtos), welches als erstes kommerzielles Betriebssystem komponentenbasiert entworfen wurde und dessen Laufzeitsystem Java und Component Pascal gleichermassen unterstützt.

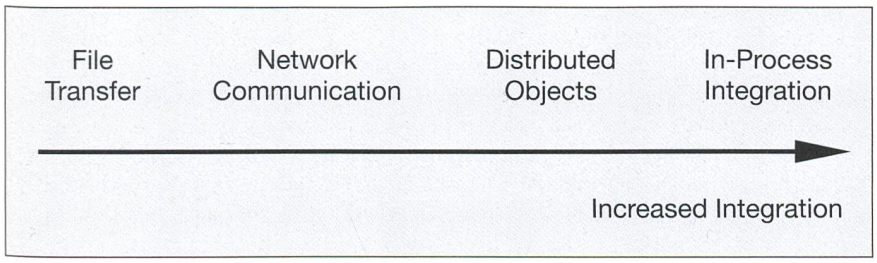

Bild 1 Von Fileformaten bis zur In-Process-Integration

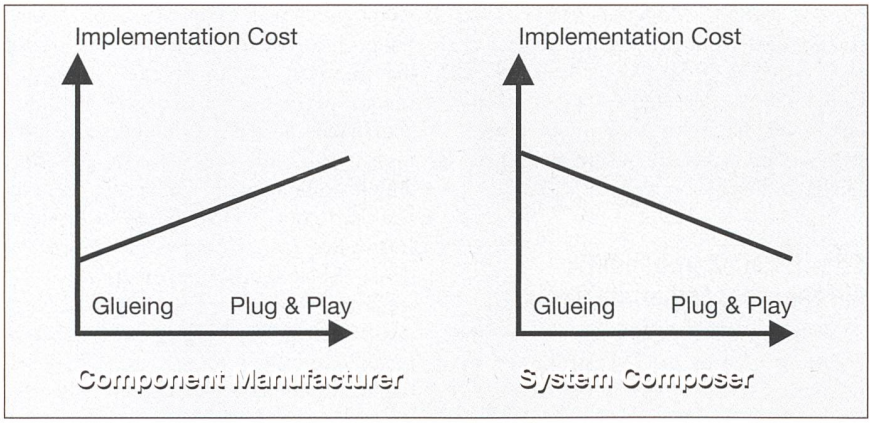

Bild 2 Spektrum zwischen ungeplanter Integration (Glueing) zu vorbereiteter Integration (Plug & Play)

Im Desktop-Bereich sind bisher etwa 6000 COM-Komponenten auf dem Markt; dieser Markt dürfte 1997 über eine Milliarde Franken Umsatz erreicht haben. Für Java sind bereits gegen 200 JavaBeans-Komponenten auf dem Markt, und eine rasante Zunahme kann erwartet werden. Es sieht deshalb so aus, dass echte Komponentensoftware in den nächsten Jahren entweder auf COM oder JavaBeans basieren wird, wobei für die Kommunikation zwischen Beans ein Subset von Corba zum Standard werden dürfte. Es existieren verschiedene Produkte, welche die COM- und die Java-Welt überbrücken helfen.

Echtes Plug & Play bleibt das Ziel

Zusammenfassend kann man sagen, dass die Integration von Softwarekomponenten von der Offline-Integration mittels Fileformaten über den Online-Versand von Daten mittels Netzwerken zur sicheren In-Process-Integration von Komponenten führt. Die Herausforderung liegt darin, wie man die Integrationskosten für separat entwickelte Komponenten minimieren kann. Testen genügt in einer offenen Komponentenwelt nicht mehr, denn wie kann man eine Komponente auf Kompatibilität zu einer anderen Komponente testen, wenn man diese gar nicht kennt oder wenn sie noch gar nicht existiert? Im Idealfall sollte man Komponenten ohne jeglichen Integrationsaufwand miteinander kombinieren können. Ein solches tatsächliches *Plug & Play* von Komponenten ist heute noch sehr selten und wird in seiner Idealform auch selten bleiben. Trotzdem muss das Ziel sein, die Integrationskosten minimal zu halten, das heisst, so nahe wie immer möglich an echtes Plug & Play heranzukommen. Die heutige Situation, in der noch intensiv Legacy Wrapping mittels aufwendigem «Glue»-Code betrieben wird, das heisst, wo Softwarepakete integriert werden, die nie dafür gedacht waren, ist langfristig schlicht zu teuer (Bild 2).

Um die Integrationskosten zu minimieren, müssen wir also zunehmend in Richtung Plug & Play und sicherer In-Process-Integration gelangen. Korrektes Funktionieren von derart integrierten Komponenten setzt gute Infrastrukturstandards und sichere Programmiersprachen voraus. Darauf aufbauend braucht es aber auch gute Designs von anwendungsspezifischen API, sogenannten Komponentenframeworks. Technisch gesehen liegt die Herausforderung darin, wie man möglichst viel Kompatibilität ohne Testen gewährleisten kann, das heisst, wie man dafür sorgen kann, dass sowohl der Klient eines Services als auch der Implementator des Services sich an die Schnittstelle, das heisst an einen «Vertrag», halten. Manche gängigen Entwicklungsmethoden, nicht zuletzt die objektorientierte Programmierung, haben hier so ihre Tücken [1]. Es wird letztlich aber immer einen Teil des Vertrages geben, dessen Einhaltung nicht mit vernünftigem Aufwand durch Werkzeuge oder Methoden gewährleistet werden kann. Dort wird der Markt durch scharfe Konkurrenz dafür sorgen, dass Komponentenanbieter mit zu schlechter Qualität vom Markt verdrängt werden.

Gute Chancen für Schweizer Komponentenindustrie

Die Schweiz gehört zu den Ländern, in denen am meisten Know-how im Bereich Komponentensoftware vorhanden ist, mindestens vergleichbar mit den USA. Dazu kommt die Erfahrung der Schweizer Industrie bei der Softwareintegration und das Anwendungswissen in verschiedenen Branchen, zum Beispiel im Werkzeugmaschinenbau. Das sind ideale Voraussetzungen für den Aufbau einer eigenen, sinnvoll spezialisierten Komponentenindustrie. Diese einmalige Chance gilt es zu nutzen. Aktivitäten an den Hochschulen und bei den Softwareanbietern wie auch Aktivitäten im Rahmen des Schweizer Automatik-Pools oder im Bereich der Werkzeugmaschinenindustrie lassen hoffen. Die bange Frage aber ist, ob der Heimmarkt ebenfalls innovativ genug ist oder lieber wartet, bis Anbieter im Ausland aufgeholt haben.

Literatur

[1] *C. Szyperski:* Component Software – Beyond Object-Oriented Programming, Addison-Wesley, 1998, ISBN 0-201-17888-5.

Logiciels pour composants

Ou comment l'informatique devient quand même une discipline d'ingénieur

On devrait pouvoir combiner sans dépense (Plug & Play) des composants logiciels en applications robustes et efficientes. Même si la situation actuelle est encore notablement éloignée d'un tel état idéal, la pression des coûts pousse le développement de logiciels dans cette direction. L'article ci-dessus traite – à la lumière de l'état actuel du développement – des exigences que l'on pose au développeur de composants logiciels, des efforts de standardisation des firmes de logiciels, de la problématique des interfaces ainsi que du rôle important joué par les langages de programmation et le design des logiciels. L'auteur attire en outre l'attention sur les chances intactes de l'industrie suisse des composants qui, grâce à l'expérience et les connaissances acquises dans les applications par différentes branches (exemple construction de machines-outils) prend une position remarquable dans l'intégration des logiciels – et prendra à l'avenir aussi – à condition que le marché intérieur suive.