Software-Entwicklungsumgebungen

Udo Kelter

02.02.2001

Zusammenfassung dieses Lehrmoduls

Dieses Lehrmodul führt in das Thema Software-Entwicklungsumgebungen ein. Wir diskutieren zunächst die Motive und den erwarteten Nutzen durch den Einsatz von SEU. Typische Funktionen von Werkzeugen einer SEU werden aufgelistet. Eine SEU soll mehrere Qualitätsmerkmale aufweisen; das wichtigste Qualitätsmerkmal ist, daß eine SEU in verschiedener Hinsicht integriert ist. Schließlich skizzieren wir einige Klassen von SEU und konstatieren, daß eine SEU i.w. als Standardsoftware anzusehen ist, die i.d.R. von ihren Benutzern, also Softwareentwicklern, nicht komplett neu entworfen werden kann; allenfalls können Standardprodukte lokal angepaßt werden.

Vorausgesetzte Lehrmodule:
obligatorisch: – Vorgehensmodelle

Stoffumfang in Vorlesungsdoppelstunden: 1.2

Inhaltsverzeichnis

1 Grundbegriffe

Fast alle Methoden der Softwaretechnik sind - zumindest ab einer bestimmten Systemgröße - zu aufwendig, um manuell praktiziert zu werden. Mit anderen Worten benötigt man computergestützte Software-Entwicklungswerkzeuge oder -Umgebungen (**SEU**), die die Entwicklung von Software unterstützen. Dieses Prinzip wird auch durch das Schlagwort **Computer-Aided** (oder **Assisted**) **Software Engineering** (**CASE**) bezeichnet. Mit CASE verfolgt man folgende abstrakten Ziele:

- Verbesserung der Qualität der entwickelten Software
- Reduktion der Gesamtkosten
- Verbesserung des Managements von Entwicklungsprojekten

Diese abstrakten Ziele werden wir später noch verfeinern und damit einzelne technische Anforderungen an SEU begründen.

Werkzeuge und Umgebungen. Unterstützung bei der Software-Entwicklung durch Werkzeuge war schon immer zwingend notwendig. Beispiele konventioneller Werkzeuge sind Text-Editoren, Assembler, Übersetzer, Testhilfen, Modul-Bibliotheken oder andere Werkzeuge mit einem begrenzten Funktionsumfang.

Hinzugekommen sind seit den 80er Jahren zusammen mit der Verbreitung von graphikfähigen PCs und Workstations vor allem Werkzeuge mit graphischer Bedienschnittstelle und Werkzeuge für die frühen Phasen der Software-Entwicklung, z.B. Editoren für Datenflußdiagramme, Entity-Relationship-Diagramme, Petri-Netze und in letzter Zeit vor allem UML-Modelle. Als **CASE-Werkzeug** bezeichnen wir hier alle Software-Produkte, die zumindest einzelne bei der Entwicklung von Software benötigte Funktionen anbieten.

Das Arbeiten mit mehreren isolierten Einzelwerkzeugen hat sich allerdings vielfach als zu ineffektiv erwiesen. Dies war eine Hauptmotivation dafür, "integrierte" Systeme mit einem breiten, "vollständigen" (s.u.) Funktionsumfang zu entwickeln. Letztere bezeichnet man als

Software-Entwicklungsumgebungen oder **CASE-Umgebungen**.

Einführung von CASE. CASE-Werkzeuge wurden und werden immer wieder – überspitzt formuliert – als Zaubermittel, mit dem sich alle Probleme bei der Softwareentwicklung lösen lassen, angepriesen und verkauft. Dies ist ähnlich lächerlich wie der Glaube, man würde zum Architekten, wenn man sich ein 3D-Zeichenprogramm für Häuser kauft.

Gute CASE-Werkzeuge können durchaus die Entwickler entlasten und deren Produktivität erhöhen, allerdings nur unter der Voraussetzung, daß die Entwickler die Entwicklungsmethoden, die das Werkzeug unterstützt, gut beherrschen (und diese Methoden überhaupt akzeptieren). Oft fehlen entsprechende Methodenkenntnisse; die Einführung von CASE bei einem Software-Produzenten erfordert daher häufig, zunächst einmal die zukünftigen Benutzer der Werkzeuge in den unterstützten Methoden auszubilden. Darüber hinaus muß u.U. die interne organisatorische Struktur der Software-Entwicklung verändert werden. Methodenschulung und organisatorische Anpassungen stellen das entscheidende Problem bei der Einführung von CASE dar. Neben den eigentlichen Werkzeugen benötigt ein CASE-Anwender daher häufig Beratung bei der Methodenauswahl, Schulungen, Studienmaterialien etc.

2 Ein "Lastenheft" für SEU

Eine SEU ist ein Softwaresystem, somit sollten im Prinzip die Methoden der Softwaretechnik auch auf SEU anwendbar sein. In diesem Abschnitt werden wir in Anlehnung an dieses Paradigma die allgemeinen Anforderungen an eine SEU diskutieren und uns dabei lose an der Struktur von Lastenheften (wie z.B. in [SASM] vorgeschlagen) orientieren. Auf folgende Punkte gehen wir ein:

- Ziele des Einsatzes von Werkzeugen
- Einsatzbereiche und Funktionsumfang

- Benutzertypen
- Qualitätskriterien

Im Gegensatz zu einem Lastenheft beschreiben wir aber nicht die Anforderungen an ein konkretes System, sondern die Spannbreite der Anforderungen, die in unterschiedlichen Kontexten auftreten.

Auf eine detaillierte Anforderungsanalyse im Sinne eines Pflichtenhefts und Realisierungsaspekte gehen wir später in Abschnitt 4 ein.

2.1 Ziele des Einsatzes von Werkzeugen

Durch den Einsatz von Werkzeugen bzw. SEU will man viele Ziele erreichen. Aus Sicht des Anwenders, also hier eines Softwareproduzenten – von dem wir annehmen, daß der mit der Softwareentwicklung Geld verdienen will –, sind vor allem folgende Ziele offensichtlich:

- Kostenreduktion
- Verbesserung der Qualität des Produkts
- Verbesserung des Managements des Entwicklungsprozesses
- Reduktion der Entwicklungszeit

Aus einer rein kommerziellen Sicht ist die Kostenreduktion das einzige primäre Ziel, die weiteren genannten Ziele tragen alle indirekt zur Kostenreduktion bei[1]. Die Ziele sind nicht unabhängig voneinander, so verursachen Maßnahmen zur Qualitätssteigerung Kosten, was dem Ziel der Kostenreduktion zuwiderläuft.

Klar sollte sein, daß Werkzeuge nicht der einzige Einflußfaktor hinsichtlich der Erreichung dieser Ziele sind. Ebenfalls erheblichen Einfluß haben u.a. die Wahl der Programmiersprache und der Entwicklungsmethoden und die Ausbildung der Entwickler. Ein gutes Werkzeug für eine schlechte Methode nützt wenig. Im folgenden gehen wir davon

[1] In gewisser Hinsicht ist die Reduktion der Entwicklungszeit eine Ausnahme hiervon, denn eine verkürzte Entwicklungszeit bei unveränderten Kosten kann z.B. entscheidend dafür sein, einen Auftrag überhaupt zu bekommen.

aus, daß die Methoden und Sprachen, die unterstützt werden sollen, bereits vorgegeben sind und daß Qualitätsmängel derselben nicht als Qualitätsmängel der Werkzeuge angesehen werden. Wir reduzieren die Diskussion also darauf, was Werkzeuge bei diesen Vorgaben zur Erreichung der o.g. Ziele beitragen können.

Kostenreduktion. Das Ziel der Kostenreduktion läßt sich in erster Linie dadurch erreichen, daß die Entwickler von aufwendigen automatisierbaren Tätigkeiten entlastet werden. M.a.W. wird die Produktivität der Entwickler gesteigert. Wichtige Beispiele für Tätigkeiten, die sowohl aufwendig als auch automatisierbar sind und für die eine SEU Werkzeuge anbieten sollte, sind:

- die Transformation von Dokumenten in andere Formate bzw. abhängige Dokumente. Triviales Beispiel ist die Übersetzung eines Programms, das in einer höheren Programmiersprache geschrieben ist, in Maschinensprache durch einen Compiler. Nicht weniger trivial, aber heute noch nicht selbstverständlich ist die automatische Generierung von Dokumentation wie z.B. Verweislisten, eine durchsuchbare HTML-Darstellung der Schnittstellen eines Systems usw.
- die Prüfung der Korrektheit bzw. Konsistenz von einzelnen Dokumenten bzw. mehreren zusammengehörigen Dokumenten. Für die Fehlersuche wird sehr viel Arbeitszeit verwendet, so daß hier Werkzeugunterstützung besonders lohnend ist. Die konkret durchzuführenden Prüfungen hängen natürlich vom vorliegenden Dokumenttyp ab. Beispiele sind Syntaxprüfungen in Quellprogrammen, wie sie üblicherweise von Compilern oder speziellen Testwerkzeugen durchgeführt werden, oder die Überprüfung von Konsistenzkriterien zwischen zusammengehörigen Einzeldokumenten, z.B. ER-Diagramm, Datenflußmodell und Data Dictionary in der Modernen Strukturierten Analyse.
- die Integration von Dokumenten über mehrere Phasen hinweg; hierauf gehen wir unten näher ein.
- die Unterstützung der Wiederverwendung

- das automatische Veranlassen der vorstehenden Prüfungen bzw. Transformationen, wenn sich das Ausgangsdokument verändert hat.

Qualitätsverbesserung. Viele der vorstehenden Tätigkeiten sind so aufwendig, daß sie von Hand praktisch nicht oder nur selten durchgeführt werden können und, sofern keine geeigneten Werkzeuge verfügbar sind, schlicht unterbleiben. Folge ist eine reduzierte Qualität. Die entsprechenden Dienste einer SEU wirken daher vielfach eher qualitätsverbessernd als kostensenkend.

Ferner können Arbeitsschritte, die von Hand gestartet werden, versehentlich mit falschen Optionen oder Dokumentversionen durchgeführt oder ganz vergessen werden. Gegen derartige Bedienungsfehler hilft ein weitgehend automatisiertes Anstoßen aller notwendigen Arbeitsschritte auf der Basis eines Vorgehensmodells. Eine derartige Unterstützung bedingt natürlich eine formale Spezifikation des Vorgehensmodells.

2.2 Einsatzbereiche und Funktionsumfang

2.2.1 Zu unterstützende Tätigkeiten bei der Software-Entwicklung

Bei der Entwicklung eines Softwaresystems werden verschiedene Typen von (meist papierlosen) Dokumenten mit entsprechenden Methoden und Verfahren produziert. Eine (Entwicklungs-) **Methode** ist dabei gegeben durch

1. einen **Systembeschreibungstyp**, z.B. ER-Diagramme, Petri-Netze, Architekturdiagramme, Quellprogramme usw.,
2. eine Menge von detaillierteren Verfahren bzw. Arbeitsschritten ("Tätigkeiten") und mehr oder minder präzise Regeln, in welcher Weise die Tätigkeiten auszuführen sind, um die gewünschte Systembeschreibung zu produzieren.

Eine **Systembeschreibung** eines bestimmten Typs beschreibt das Softwaresystem aus einem bestimmten Blickwinkel bzw. auf einem

bestimmten Abstraktionsniveau. Um sie mit Werkzeugen verarbeiten zu können, muß sie in der Syntax einer konkreten textuellen oder graphischen **Sprache** gespeichert werden.

Neben den Tätigkeiten, die unmittelbar zur (Weiter-) Entwicklung der Software und der zugehörigen Dokumentation beitragen, treten auch Tätigkeiten zur Projektadministration, Qualitätssicherung, Berichtswesen innerhalb des Unternehmens etc. auf.

In einem **Vorgehensmodell** (*software process model*) wird mehr oder weniger exakt festgelegt, welche Dokumente zu produzieren sind, welche Methoden und Verfahren dabei anzuwenden und welche Tätigkeiten in welcher Abfolge durchzuführen sind. Beispiele sind das Phasenmodell und das Spiral-Modell. Das Vorgehensmodell hängt vom Typ, der Größenordnung und der geforderten Qualität der zu entwickelnden Software sowie anderen Faktoren ab. Die Vorgehensmodelle für verschiedene Klassen von Software können so unterschiedlich sein, daß gewisse Tätigkeiten bei manchen Vorgehensmodellen auftreten und bei anderen nicht.

Die in einem Vorgehensmodell auftretenden Methoden sollten *integriert* sein in dem Sinne, daß die sukzessive zu erstellenden Dokumente semantisch konsistent sind und durch inkrementelle Erweiterung oder durch Transformation auseinander entstehen. Ferner sollen sie keine Teile enthalten, die nicht zum Endergebnis beitragen.

Wie schon oben erwähnt hängen die anfallenden Tätigkeiten vom Typ der zu entwickelnden Software und vom Vorgehensmodell ab. Es ist hier nützlich, zwei Klassen von Tätigkeiten zu unterscheiden:

- Tätigkeiten, die bei fast allen Vorgehensmodellen auftreten: Projektverwaltung, Entwicklungsprozeßsteuerung, Konfigurationsmanagement, Dokumentation, Textverarbeitung, Berichterstellung, Wiederverwendung von Komponenten (Bibliotheken), elektronische Post und andere Bürotätigkeiten.
- Vorgehensmodell-spezifische Tätigkeiten, z.B. editieren, prüfen, transformieren, übersetzen etc. von konkreten Systembeschreibungen bzw. Dokumenten.

2.2.2 Funktionen von Werkzeugen

Eine SEU umfaßt normalerweise (abhängig vom zu unterstützenden Vorgehensmodell) eine größere Anzahl methodenspezifischer oder allgemein einsetzbarer "Werkzeuge". Den Begriff Werkzeug verstehen wir hier nicht im Sinne eines ladbaren Programms, sondern in einem allgemeinen Sinn als eine Komponente oder ein Modul der SEU, das eine Funktionalität realisiert bzw. Dienstleistungen anbietet, die die Lösung einer bestimmten Klasse von Aufgaben unterstützen. Weniger relevant sind hier Details des Aufrufs und der Benutzungsschnittstelle (z.B. graphisch oder textuell, menügesteuert oder kommandoorientiert, selbständig startbar oder eingebettet). Beispielsweise erlauben es viele "Editoren" nicht nur, Dokumente zu editieren, sondern auch Konsistenzprüfungen zu veranlassen, Dokumente in eine druckbare Form zu konvertieren oder Konfigurationen zu verändern. Die Konsistenzprüfung, Konvertierung und Konfigurationsverwaltung betrachten wir jeweils als eigene Funktionalität, die in verschiedenen Kontexten aufrufbar sein kann und die nur einmal innerhalb der SEU durch ein Modul realisiert werden sollte. Es folgt eine Liste der wichtigsten Funktionalitäten einer SEU[2]:

1. Dokumentenverwaltung:
   - allgemeine Dokumentenverwaltung, typischerweise an Projekten und anderen Organisationsstrukturen orientiert (klassischerweise Aufgabe des Dateisystems)
   - Verwaltung von Versionen und Konfigurationen incl. Kontrolle des parallelen Arbeitens; Beispiele solcher Werkzeuge sind `SCCS` und `RCS`
   - Archivierung und Wiedereinspielung von Dokumenten
   - vage Suche nach Dokumenten, u.a. bei der Wiederverwendung von Software; Beispiele entsprechender Werkzeuge in UNIX-Systemen

[2] Die Liste enthält auch Dienste, die von sog. "Dienstprogrammen" des Betriebssystems angeboten werden; diese arbeiten stets mit Dateien. In SEU, die auf einem Objektmanagementsystem basieren, können diese Dienstprogramme natürlich nicht mehr unverändert eingesetzt werden.

sind `grep` und `apropos` .

2. Dokumentbearbeitung:
   - Dokumenteingabe und -Korrektur; Varianten hiervon:
     - textuell oder graphisch oder beides gemischt
     - syntaxorientiert oder nicht (d.h. generell oder methoden- / phasenspezifisch)
   - statische und dynamische Konsistenz- und Korrektheitsprüfungen, incl. Testhilfen
   - Dokumentformatierung und Berichterstellung: entspricht einem Übersetzungsvorgang (ggf. inkrementell) in eine Druckersprache
   - Konvertierung von Dokumenten, insb. Übersetzung aus höheren Programmiersprachen in Maschinensprache (ggf. inkrementell)
3. Simulatoren (z.B. für Petri-Netze)
4. Steuerung von Werkzeugen
   - Skriptsprachen und zugehörige Interpreter
   - Programmgenerierung und Übersetzungssteuerung: in Dateisystemen beispielsweise durch Systeme wie `make` abgedeckt; Basis sind Abhängigkeiten zwischen Übersetzungseinheiten ("makefile") und Informationen über Änderungen an Dokumenten (z.B. Zeitstempel)
5. Projektmanagement
   - Netzplantechnik und andere Planungsverfahren
   - Messung von Merkmalen zur Qualitäts- oder Aufwandsabschätzung
6. Kontrolle und Unterstützung eines formal modellierten Entwicklungsprozesses.

Neben den "normalen" Werkzeugen gibt es noch "**Meta-Werkzeuge**", also Werkzeuge zum Erzeugen von Werkzeugen:

- Makroprozessoren: durch diese kann beispielsweise der Umfang einer Sprache verändert werden
- Übersetzergeneratoren: diese erlauben es, aus einer Grammatik einer Sprache wesentliche Teile eines Übersetzers zu generieren; sofern auch die Semantik der Sprache formal angebbar ist, kann sogar der komplette Übersetzer generiert werden.
- Syntaxeditor-Generatoren: diese erlauben es, analog zu Übersetzergeneratoren aus einer Grammatik einer Sprache einen zugehörigen Syntaxeditor (manchmal gepaart mit einem inkrementellen Compiler) zu generieren.

Wir haben bisher unterstellt, daß die SEU und die Software, die mit der SEU entwickelt wird, auf dem gleichen Rechner laufen, Entwicklungs- und "Produktionsrechner" also identisch sind. Dies ist nicht immer möglich. Bei eingebetteten Prozessoren, z.B. in einer Chipkarte oder einem Netzwerk-Controller, kann es sein, daß keine Tastatur, kein Bildschirm oder keine Speichermedien angeschlossen werden können oder die Prozessorleistung und der verfügbare Hauptspeicher viel zu gering sind. Bei sehr teuren Hochleistungsrechnern kann es sein, daß diese für die eigentliche Produktionsaufgabe eingesetzt werden müssen und daß die Rechenlast, die durch die für die Entwickler laufenden SEU verursacht wird, den Betrieb stören würde. In solchen Fällen trennt man zwischen Entwicklungs- und Zielrechnern. Als Entwicklungsrechner verwendet man heute vernetzte PCs. Folgende zusätzliche Funktionen müssen hier durch die SEU angeboten werden:

- Herunterladen von ausführbaren Programmen auf den Zielrechner, incl. der Kontrolle der entsprechenden Vernetzungsmechanismen
- Starten und Kontrollieren der Programme auf dem Zielrechner, incl. Testunterstützung
- Simulatoren, die den Zielrechner auf dem Entwicklungsrechner simulieren

Wegen der Vielfalt existierender Vorgehensmodelle und der daraus folgenden Vielfalt von SEU kann man sehr viele Einzelfunktionen,

die irgendwo in irgendeiner SEU auftreten, identifizieren. Ein umfangreiches Klassifikationsraster derartiger Funktionen findet sich in [ECMATR69].

2.3 Benutzertypen

Der klassische Benutzer einer SEU ist der "qualifizierte Entwickler", also jemand, der i.w. die Methodenkenntnisse eines Diplom-Informatikers hat und der längerfristig mit der gleichen Umgebung arbeitet, also Zeit hat, viele Details und Facetten der SEU kennenzulernen.

Sehr häufig sind diese Annahmen aber nicht erfüllt. Software wird vielfach von Personen (mit-) entwickelt, die in erster Linie Fachleute auf einem Anwendungsgebiet sind und die in der Informatik nur rudimentäre Kenntnisse haben. Für solche Personen ist es sinnvoller, viele Details vorzugeben bzw. nur die wahrscheinlich sinnvollen Optionen zuzulassen. Ferner sind erhöhte Anforderungen an das Hilfesystem der SEU zu stellen.

Ein weiterer Benutzertyp sind gelegentliche Nutzer; für diese bildet der oft unüberschaubare Vorrat an Funktionen und Optionen ein wesentliches Hindernis.

Ähnlich sind "Anfänger" einzustufen, die noch ausgebildet werden, und die zunächst nur die "wichtigen" Funktionen sehen sollten.

2.4 Qualitätskriterien

Funktionale Vollständigkeit: Eine SEU soll Dienste anbieten, die *alle* bei der Entwicklung von Software anfallenden Tätigkeiten unterstützen. Wie schon oben erwähnt hängen die anfallenden Tätigkeiten vom Typ der zu entwickelnden Software und vom Vorgehensmodell ab. Gleiches gilt somit für den Begriff Vollständigkeit. Alle vorgehensmodellspezifischen und alle allgemein auftretenden Tätigkeiten sollten von einer SEU unterstützt werden.

Methodentreue: Sofern der Anbieter eines Werkzeugs behauptet, daß dieses die Methode X unterstützt, sollte die Methode X auch vollständig und exakt unterstützt werden. Wenn die Methode X z.B.

bestimmte Dokumenttypen, Dokumentelemente und Konsistenztests vorsieht, sollten diese alle darstellbar sein bzw. angeboten werden (dies kann zunächst noch als ein Aspekt der funktionalen Vollständigkeit angesehen werden) und exakt gemäß der Methodenbeschreibung arbeiten. Analog sollten graphische Notationen exakt unterstützt werden (bei der Anzeige auf dem Bildschirm oder beim Drucken auf Papier).

Leider sind viele Methoden in Lehrbüchern oder sogar in den Originalquellen nur recht vage beschrieben. Viele Details werden von den Methodenbeschreibungen überhaupt nicht behandelt (z.B. Eigenschaften von Editoren: Welche Dokumentfragmente können auf die Zwischenablage des Editors kopiert werden? Wie können mehrere Entwickler parallel auf überlappenden Dokumenten arbeiten?) oder sind nicht praktikabel (z.B. zu detailreiche graphische Notationen, die auf Bildschirmen mit üblicher Auflösung nicht darstellbar sind, unzureichende Unterstützung sehr großer Modelle usw.)

Integration: Eine besonders wichtige Anforderung an eine SEU ist, daß sie in mehrfacher Hinsicht integriert ist (wobei die SEU "semantisch" nur so weit integriert sein kann, wie die unterstützten Methoden integriert sind):

- *Verteilung, unterliegendes Betriebssystem:* Wenn eine SEU aus einzelnen Werkzeugen, die ggf. sogar auf verschiedenen Rechnern (PCs, Mainframes, Workstations) laufen, zusammengesetzt ist, dann müssen als elementarste Form der Integration diese Werkzeuge von einem Arbeitsplatz aus benutzbar gemacht werden.
- *Daten:* Die verschiedenen Arten von Systembeschreibungen, die in einem Vorgehensmodell auftreten, weisen fast immer gewisse Redundanzen (oder Konsistenzbedingungen untereinander) auf. Zum Beispiel kann der Datentyp einer Klasse in einem Klassendiagramm auch in einem zugehörigen Zustandsübergangsdiagramm und im Quellprogramm auftreten. Redundante oder ableitbare Daten sollten nicht erneut vom Software-Entwickler eingegeben werden müssen. Sofern Redundanzen nicht vermeidbar sind, muß die Beseitigung von Inkonsistenzen unterstützt werden.

- *Benutzungsschnittstelle:* Um den Lernaufwand zu begrenzen und die Benutzungsfreundlichkeit zu erhöhen, sollen die Sprachen, in denen der Entwickler mit verschiedenen Werkzeugen bzw. Funktionsgruppen der SEU kommuniziert, möglichst einheitlich sein. Dies gilt für alle Abstraktionsebenen der Kommunikation: einzelne Zeichen (lexikalische Ebene), Syntax und Semantik von Kommandos, ganze Dialoge. Die lexikalischen und syntaktischen Aspekte der Kommunikation können durch ein Fenstersystem konstruktiv vereinheitlicht werden. Wünschenswert ist ferner, daß die Werkzeuge die zum unterliegenden Basissystem gehörigen Gestaltungsrichtlinien einhalten[3].
- *(Werkzeug-) Steuerung / Automation:* Bei den meisten Vorgehensmodellen treten häufig wiederholte Sequenzen von Arbeitsschritten auf (z.B. Editieren - Prüfen - Übersetzen - Binden von Programmen). Entsprechend können durch die Gruppierung von Funktionen innerhalb einer SEU wiederholte Sequenzen von Benutzerkommandos erforderlich sein. Soweit möglich und sinnvoll, sollten einzelne Werkzeuge bzw. Funktionen der SEU automatisch aufgerufen und gesteuert werden.
- *Überwachung und Unterstützung des Software-Entwicklungsprozesses:* Die Einhaltung der im Vorgehensmodell enthaltenen Regeln sollte kontrolliert werden. Diese Regeln sind immer dann anwendbar, wenn ein Entwickler einen Arbeitsschritt beendet und ein Folgearbeitsschritt ausgewählt werden muß. Die Kontrolle bzw. Unterstützung kann z.B. darin bestehen,
  - Abweichungen vom Vorgehensmodell durch Warnungen anzuzeigen
  - mögliche nächste Arbeitsschritte vorzuschlagen
  - den Entwickler bei der Auswahl alternativer Arbeitsschritte zu beraten (z.B. anzuwendenden Prüfungen)

[3] Da die konkurrierenden Basissysteme in ihrer Funktionalität und den Gestaltungsrichtlinien nicht konsistent sind, kann eine komplexe Applikation immer nur an eine Plattform optimal angepaßt werden. Will ein Werkzeuganbieter mehrere Plattformen unterstützen, müssen u.U. Varianten des Werkzeugs gebildet werden.

- die Menge der zulässigen nächsten Arbeitsschritte einzuschränken
- den nächsten Arbeitsschritt automatisch zu starten[4]

Benutzungsfreundlichkeit: Die Bedienung der SEU sollte leicht erlernbar, bequem, konsistent (s. auch Integration der Benutzerschnittstelle) und an individuelle Benutzerbedürfnisse anpaßbar sein. Die Benutzerschnittstelle der SEU sollte software-ergonomische Standards (DIN 66234, Teil 8: Grundsätze der Dialoggestaltung) einhalten und ein Hilfesystem enthalten.

Teamarbeit: Große Softwaresysteme werden arbeitsteilig in Teams entwickelt. Die parallele Arbeit der Entwickler und die Kooperation innerhalb des Teams muß unterstützt werden.

Adaptierbarkeit: Die SEU muß an die Arbeitsumgebung und an die *organisatorisch/technischen Verhältnisse* beim Software-Entwickler (z.B. Layout von Ausdrucken, interne Prozeduren und Standards etc.) *adaptierbar* sein.

Offenheit: Die Architektur sollte interne Schnittstellen aufweisen, die die Integration mit anderen Werkzeugen (z.B. beim Software-Entwickler bereits vorhandenen oder für die Adaptierung zusätzlich benötigten Werkzeugen) erleichtern. Diese Schnittstellen sollten offen sein. Beispiele sind u.a. Datenaustauschformate.

Hintergrund dieser Forderung ist auch, daß viele Anwender sich ihre SEU aus Komponenten, die von verschiedenen Herstellern stammen, zusammensetzen wollen (s. Abschnitt 4); ohne normierte Schnittstellen ist nicht zu erwarten, daß Produkte unterschiedlicher Hersteller zusammenpassen. Unter einer SEU-Plattform versteht man einen Satz entsprechender Schnittstellen; unter Offenheit versteht man dann die Fähigkeit der SEU, in SEU-Plattformen integrierbar zu sein.

[4] Dieser Fall entspricht der schon erwähnten automatischen Werkzeugsteuerung.

3 Beispiele für Klassen von Umgebungen

Im folgenden werden aus der Vielzahl verschiedener Arten von Umgebungen nur einige wenige typische Formen vorgestellt. Die angegebenen Merkmale beziehen sich sowohl auf den für den Benutzer sichtbaren Funktionsumfang als auch auf die Architektur der Umgebung. [Na93] enthält ein sehr detailliertes Klassifikationsraster für SEU und viele weitere Beispiele. Literaturangaben zu den als Beispiel angeführten Umgebungen finden sich in [PeR88]. Beschreibungen weiterer Umgebungen finden sich in [Ba98, Na93].

Klassifikation nach Abdeckungsgrad der Phasen:

Programmierumgebungen unterstützen nur die "späten" Phasen der Software-Entwicklung ("*lower CASE*"), also Entwurf, Programmierung und Test von Programmen, meist nur in einer konkreten Programmiersprache.

"Upper-CASE"-Umgebungen enthalten primär Werkzeuge, die die frühen Entwicklungsphasen unterstützen. Typische unterstützte Methoden sind die UML[5], Datenmodellierung mit Entity-Relationship-Diagrammen oder Datenflußmodelle. Derartige Umgebungen werden vor allem bei der Entwicklung von betrieblichen Informationssystemen benutzt; dieser spezielle Anwendungsbereich der Softwaretechnik hat vermutlich die größte Zahl von Anwendern und das der Zahl nach umfangreichste Angebot an Werkzeugen. "Upper-CASE"-Umgebungen bzw. Werkzeuge müssen mit einem Data Dictionary System, einem Datenbanksystem und ggf. einer 4.-Generationssprache integriert sein; letztere werden in diesem Kontext meist nicht als CASE-Werkzeuge bezeichnet.

Klassifikation nach der Realisierungsmethode:

"Werkzeugkästen" bestehen aus mehreren durch das Betriebssystem verbundenen Einzelwerkzeugen. Die Daten sind in Dateien

[5] Bei objektorientierten Methoden ist die Trennung zwischen Analyse, Entwurf und Implementierung nicht scharf bzw. soll gerade überwunden werden; UML-Werkzeuge unterstützen daher oft auch die späten Phasen.

gespeichert; Werkzeuge tauschen Daten über Dateien aus. Der Benutzer muß die Werkzeuge durch Betriebssystemkommandos aufrufen; allerdings kann durch geeignete Kommandoprozeduren dieser Aufwand stark reduziert und insgesamt der Eindruck einer integrierten Umgebung erweckt werden. Bekanntestes Beispiel ist UNIX mit den zugehörigen Werkzeugen. Die einzelnen Werkzeuge können in unterschiedlichen Sprachen geschrieben sein.

In **sprachbezogenen SEU** sind alle Werkzeuge in der gleichen Programmiersprache geschrieben und werden typischerweise sogar zusammen in einem einzigen Betriebssystemprozeß ausgeführt bzw. interpretiert; hierdurch können sie über gemeinsame Hauptspeicherdatenstrukturen besonders effizient Daten untereinander austauschen und wirken daher aus Benutzersicht besser integriert.

Meta-CASE-Umgebungen bestehen im Kern aus einem Interpreter für Werkzeugspezifikationen. Motiviert sind solche Architekturen dadurch, daß z.B. die vielen in den frühen Phasen auftretenden Diagrammtypen sehr viele Gemeinsamkeiten aufweisen (z.B. Operationen in Formularen oder graphischen Editoren für netzartige Dokumente), die in einem generischen Kern realisiert werden; um eine Instanz der SEU für konkrete Dokumenttypen zu bilden, wird der generische Kern u.a. um Operationen zur Darstellung von Knoten und Kanten in Netzen erweitert. Die Werkzeugspezifikationen stellen ihrerseits Dokumente dar und können ggf. durch spezielle oder sogar "normale" Werkzeuge bearbeitet werden. Vorteil des Meta-CASE-Konzepts ist aus Sicht der Anbieter der reduzierte Implementierungsaufwand und aus Sicht der Benutzer ggf. die Möglichkeit, die Werkzeuge über ihre Spezifikationen weitgehend anpassen zu können.

Klassifikation nach der Art der Datenverwaltung:

Dateien sind das klassische Medium zur Dokumentspeicherung und typisch für Werkzeugkästen. Nachteilig an Dateien ist u.a., daß keine feingranularen Konsistenzbedingungen zwischen verschiedenen Dokumenten direkt dargestellt und überwacht werden können und daß nicht ohne weiteres zusätzliche (benutzerspezifische) Erweiterungen

vorgenommen werden können[6].

Vielfach ist versucht worden, die Nachteile von Dateien durch Einsatz konventioneller, insb. relationaler **DBMS** zu vermeiden. Es zeigt sich allerdings, daß die konventionellen Datenbankmodelle wenig geeignet sind, die komplexen Strukturen innerhalb von Softwaredokumenten nachzubilden; in der Folge kommt es typischerweise zu erheblichen Performance-Problemen. Weiterhin sind die Transaktionskonzepte und andere Details konventioneller DBMS weniger geeignet für diesen Anwendungsbereich.

Aufgrund der vorstehend skizzierten Probleme benutzen manche SEU eine Kombination von Dateisystem und DBMS: die einzelnen Werkzeuge arbeiten primär auf Dateien. Zusätzlich, also redundant, werden relevante Informationen in einem typischerweise relationalen DBMS abgelegt. Die Datenbank enthält nur solche Daten, die z.B. für dokumentübergreifende Konsistenztests oder die Projektverwaltung notwendig sind, also gerade die Daten, auf denen interessierende Abfragen gestellt werden können. Die relevanten beschreibenden Daten zu einem Dokument werden typischerweise nur auf explizite Benutzeranforderung hin extrahiert und in die Datenbank eingetragen; deswegen, aber auch aus anderen Gründen, kann man i.a. nicht garantieren, daß die Daten in der Datenbank und in den Dateien immer konsistent sind.

Alternativ kann man ein nichtkonventionelles DBMS einsetzen und die Werkzeuge direkt auf diesem arbeiten lassen; solche DBMS werden auch als **Repository** bezeichnet[7]. In technischer Hinsicht kann diese Lösung optimal gestaltet werden. Intern werden bei einigen kommerziell erhältlichen Umgebungen derartige Repositories eingesetzt. Versuche, Repositories zu standardisieren (PCTE, IRDS), um die Integration von Werkzeugen zu erleichtern, sind in der Praxis gescheitert; es handelt sich hier um sehr komplexe Systeme, für die der Markt eher eng

[6]Dies gilt auch, wenn z.B. XML als Dateiformat benutzt wird; XML ermöglicht es zwar, Dokumente feingranular zu modellieren, unterstützt aber keinen Sichtenmechanismus ähnlich wie Datenbanksysteme.

[7]Die Bezeichnung Repository wird allerdings auch für andere Systeme verwendet, z.B. in Versionsmanagementsystemen für Verzeichnisse und Dateien, die Daten über frühere Versionen enthalten, und für Datenbanken, die vor allem Metadaten enthalten.

ist, was zu einem geringen Angebot und zu einer schlechten Relation zwischen Preis und Qualität führt[8], ferner macht sich ein Werkzeughersteller abhängig von dem Repository-Anbieter und kann nicht mehr ohne weiteres im Repository Funktionalität ändern, wenn dies für die Weiterentwicklung der Werkzeug wünschenswert erscheint.

4 SEU als Standardsoftware

In den vorstehenden Abschnitten, speziell in Abschnitt 2.4, waren Anforderungen an eine SEU beschrieben worden, allerdings recht allgemein und etwa auf dem Niveau eines Lastenhefts. Die Frage ist, ob und wie man diese Anforderungen detaillierter ausarbeitet. Wenn wir eine SEU als Individualsoftware ansehen, wobei ein Entwickler oder ein Gruppe von Entwicklern der "Kunde" wären, und das übliche Phasenmodell beibehalten, dann würde der nächste Schritt darin bestehen, die Anforderungen detailliert zu analysieren, anschließend das System zu entwerfen, zu implementieren usw. Die grundlegenden Analyse- und Entwurfsmethoden, die üblicherweise im Rahmen von Softwaretechnik-Vorlesungen vermittelt werden, reichen allerdings für die Entwicklung von SEU nicht aus bzw. sind nicht sinnvoll anwendbar:

- SEU sind sehr komplexe Systeme, die unterschiedlichste Komponenten enthalten, zu deren Konstruktion ganz unterschiedliche Technologien eingesetzt werden. Für die fast immer enthaltenen Compiler wird man z.B. auf Methoden des Compilerbaus zurückgreifen. Dementsprechend müßte schon in der Analysephase eine Mixtur von Methoden für die unterschiedlichen Komponenten eingesetzt werden.
- Aufgrund des Umfangs und der Diversität der Funktionen einer SEU verursacht ein halbwegs vollständiges Pflichtenheft einen extremen Aufwand. Dieser Aufwand ist nur in seltenen Ausnahmefällen finanzierbar. Normalerweise ist es allenfalls realistisch, zwischen kompletten SEU verschiedener Hersteller auszuwählen oder Komponenten verschiedener Hersteller zusammenzustellen und zu integrieren. Eine SEU ist hier als Standardsoftware zu behandeln, die überwiegend

[8] Eine ausführliche Diskussion von Repositories findet sich u.a. in [IRA].

aus im Markt erhältlichen Komponenten besteht, die auf die speziellen Benutzeranforderungen hin adaptiert bzw. nur punktuell durch eigene Software ergänzt werden.

Daher ist das klassische Phasenmodell bei SEU (und generell bei Standardsoftware) nur in stark modifizierter Form anwendbar: zentral ist die Auswahl der Standard-Komponenten, klassische Entwicklertätigkeiten fallen nur noch am Rande an. Den Auswahlprozeß können wir hier nur kurz behandeln.

Die Auswahl von SEU bzw. von SEU-Komponenten erfordert eine Bewertung derselben, für die Bewertung müssen Bewertungsmaßstäbe festgelegt werden. Bewertungsmaßstäbe sind aber inhaltlich i.w. Anforderungen, allenfalls die Formulierung ist anders, und es kommen Wichtigkeitsgrade bzw. Wertigkeiten hinzu. Wie schon erwähnt können Anforderungen mit vertretbarem Aufwand nur oberflächlich formuliert werden.

Eine einigermaßen gründliche Bewertung einer SEU anhand gegebener Bewertungsmaßstäbe ist, wenn sie mit Experimenten unterfüttert wird, sehr aufwendig. Je nach Umfang der SEU ist von einem Aufwand von mehreren Wochen oder Monaten auszugehen. Ein Grund hierfür ist, daß sich die Bewerter normalerweise erst in das System einarbeiten müssen. Bewertungen, die von "Anfängern" angestellt werden, sagen meist mehr über den Lernzustand des Bewerters als über das bewertete System aus.

Dieses Aufwandsproblem vervielfacht sich, wenn mehrere SEU zur Auswahl stehen und bewertet werden sollen. Aus Aufwandsgründen muß daher i.d.R. auf recht oberflächliche Verfahren (einfache Checklisten), die Auswertung von Testberichten und Erfahrungen anderer und vor allem Beratung zurückgegriffen werden.

Die vorstehenden Probleme zeigen andererseits, wie wichtig es ist, daß eine SEU offen, adaptierbar und erweiterbar ist (vgl. Abschnitt 2.4), um ggf. punktuell eigene Software einbinden zu können.

Literatur

[Ba98] Balzert, H.: Lehrbuch der Software-Technik - Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung; Spektrum Akademischer Verlag; 1998

[ECMATR69] Reference Model for Project Support Environments; ECMA Technical Report TR/69; zugleich NIST Special Publication 500-213; 1994/12

[Na93] Nagl, M.: Software-Entwicklungsumgebungen: Einordnung und zukünftige Entwicklungslinien; Informatik-Spektrum 16:5, p.270-280; 1993/10

[PaS94] Pagel, B.-U.; Six, H.-W.: Software Engineering, Band 1; Addison Wesley (Deutschland) GmbH, Bonn; 1994

[PeR88] Penedo, L.; Riddle, W.E.: Software Engineering Environment Architectures; IEEE Trans. o. Software Engineering 14:6, p.689-696; 1988/06

[IRA] Kelter, U.: Lehrmodul "Integrationsrahmen für Software-Entwicklungsumgebungen"; 1999/11

[SASM] Kelter, U.: Lehrmodul "Systemanalyse und Systemmodellierung"; 1999/10

Index