# UML - Unified Modeling Language im Einsatz[1]

## Teil 1+2: Hintergrund und Notationen der Standard UML

Bernhard Rumpe, Robert Sandner

Dr. Bernhard Rumpe ist Träger des Bayerischen Habilitationsförderpreises und arbeitet derzeit an der Technischen Universität München. Er ist Mitbegründer der „precise UML"-Group, die die OMG bei der Standardisierung der UML berät. Er hat die <<UML'99>>-Konferenz und zahlreiche Workshops im Umfeld von Modellierungssprachen und Vorgehensweisen mitorganisiert und berät Firmen bei der Einführung neuer Softwaretechnikkonzepte. Seine Interessensgebiete: Objektorientierte Modellierung: Sprachen, Techniken, Methodik und ihre Anwendung auf E-Commerce und verteilte Systeme.

Robert Sandner ist wissenschaftlicher Mitarbeiter am Lehrstuhl Software & Systems Engineering der Fakultät für Informatik an der Technischen Universität München. Er arbeitet im DFG-Projekt *InTime* sowie in Kooperationen mit Anwendern an einer methodischen Fundierung der Entwicklung technischer Systeme unter Einbeziehung der UML-RT. Seine Interessensgebiete: Objektorientierte Modellierungssprachen, ihre Anwendung auf eingebettete Systeme sowie im inkrementellen Softwareentwurf.

Adresse: Institut für Informatik, Arcisstraße 21,
80333 München. Email: {rumpe|sandnerr}@in.tum.de

*Dieser Artikel umfaßt die ersten beiden Teile einer Reihe von drei Beiträgen, die einen Überblick über den Einsatz der Unified Modeling Language in technischen Anwendungen geben. In diesen Teilen wird der Hintergrund der UML und die hauptsächlich eingesetzte Kernsprache der UML anhand von Beispielen beschrieben. Darauf aufbauend wird im dritten Teil der Einsatz des UML-Dialekts UML-RT für verteilte Echtzeitanwendungen erörtert. Dieser eignet sich besonders für technische Applikationen.*

### Unified Modeling Language in Practice

*This document comprises the first two parts in a series of three articles which provide an overview on the usage of the Unified Modeling Language in technical applications. This part describes the background of the UML. The main diagrams of UML's core language are explained and discussed by examples. Part 3 focuses on the dialect UML-RT which has been designed for distributed real-time applications.*

## 1. Einführung

In fast allen Ingenieursdisziplinen sind für Modellbildung und -beschreibung standardisierte Notationen selbstverständlich. In der noch jungen Softwaretechnik haben sich bis vor kurzem keine allgemein anerkannten Standard-Entwurfssprachen oder gar ISO-Normen etablieren können. Die Unified Modeling Langage (UML), die von der Object Management Group (OMG) – einem Industriekonsortium, in dem fast alle Softwarefirmen vertreten sind – standardisiert wird, ist deshalb ein wesentlicher Fortschritt. In der Entwicklung sogenannter Business Software hat sich die UML in der Praxis inzwischen weitgehend durchgesetzt. Weil in technischen Applikationen der Softwareanteil rapide steigt, wird auch dort der Einsatz der UML immer interessanter. Um den Anforderungen dieses Anwendungsbereichs gerecht zu werden, wird derzeit unter dem Namen UML-RT eine Erweiterung der UML entwickelt, die im dritten Beitrag zur UML vorgestellt wird. In diesem und im nächsten Artikel werden zunächst die wesentlichen Elemente der UML betrachtet und ihre Anwendbarkeit in der Praxis diskutiert.

### 1.1 Historie der UML

Die UML basiert auf einer Reihe von objektorientierten Modellierungsansätzen für die Entwicklung von Softwaresystemen mit komplexen Datenstrukturen, die Anfang der 90er Jahre entwickelt wurden. Wesentliche Einflüsse hatten "Object-Oriented Software-Engineering" (Jacobson's Objectory), "Object-Modeling Technique" (Rumbaugh's OMT) und "Object-Oriented Analysis and Design" (die Booch-Methode), die 1995 in einem gemeinsamen Ansatz "*Unified* Modeling Language" [1,2] mündeten. Zahlreiche weitere Einflüsse, wie zum Beispiel SDL [8] (für die Telekommunikation) oder Statecharts [9] (für eingebettete Steuerungssysteme), fanden in die UML konzeptuellen Eingang. Im Jahr 1997 wurde die UML von der OMG zum internationalen Standard erhoben [1]. Die Standardisierung und kontinuierliche Weiterentwicklung (derzeit aktuelle Version 1.4) durch die OMG sichert der UML eine breite Akzeptanz, insbesondere auch durch Werkzeughersteller, welche die für eine derartige Sprache notwendigen Editoren, Animations-, Analyse-, Transformations- und Generierungswerkzeuge entwickeln.

### 1.2 UML-basierte Vorgehensweisen

Es ist wichtig, zwischen der UML als Beschreibungssprache und einem darauf basierenden Softwareentwicklungsprozess zu unterscheiden. Die UML beinhaltet bewusst keine Vorgehensweise zur Software- oder Systementwicklung, sondern konzentriert sich auf die deskriptive Modellierung von Systemen. Obwohl die Verfügbarkeit

[1] Erschienen in: at-Automatisierungstechnik, Heft 9/2001 und 10/2001, S. A6-A14, Oldenbourg Verlag, München.

[RS01a] B. Rumpe, R. Sandner.
UML - Unified Modeling Language im Einsatz. Teile 1-2.
Hintergrund und Notation der Standard UML.
In: at - Automatisierungstechnik, Reihe Theorie für den Anwender, Band 49, Heft 9 und 10. 2001.
www.se-rwth.de/publications

von Beschreibungskonzepten starken Einfluss auf die verwendbare Methodik hat, konnten sich in letzter Zeit mehrere, teilweise recht unterschiedliche Methodiken etablieren, die weitgehend auf der Sprache UML basieren. Zu nennen sind hier Catalysis [10], der Unified Development Process [11], der Rational Unified Process [12], Fusion [13], Open [14], oder die Weiterentwicklung von ROOM [15]. Mit Ausnahme von ROOM sind die genannten Methodiken stark auf Business Software konzentriert. Dagegen bietet die Weiterentwicklung von ROOM einige für eingebettete Systeme geeignete Erweiterungen der UML. Diese Spezifika werden im dritten Beitrag dieser Artikelreihe gesondert behandelt.

Allen Methodiken ist gemeinsam, dass sie den Softwareentwicklungsprozess in eine Reihe von Aktivitäten gliedern und diese in die Phasen *Analyse*, *Entwurf* und *Implementierung* gruppieren. Diese Phasen sind oft weiter zerteilt, z.B. in logische und physische Implementierung. Durch Iteration und die zeitlich überlappende Durchführung kann eine starke Verschränkung der Aktivitäten erreicht werden. So kann bereits in den ersten Iterationen ein Prototyp für den kritischen Funktionskern entwickelt werden, um damit technische und funktionelle Risiken zu erkennen und bei der weiteren Entwicklung darauf zu reagieren.

## 1.3 Überblick

Wie viele andere technische Produkte sind auch Softwaresysteme zu komplex, um in einer einzigen, allumfassenden Darstellung beschrieben zu werden. Stattdessen werden mehrere einander ergänzende Pläne zur Darstellung spezifischer Sichten verwendet. Die Modellierung der *Struktur* bildet, wie bei praktisch allen technischen Produkten, eine wichtige Basis der Softwareentwicklung. Jedoch führt die inhärente Komplexität dazu, dass der *Funktionalität*, also dem *Verhalten* und den *Interaktionen* der Softwarekomponenten, eine bedeutendere Rolle zufällt als bei anderen Produkten. Deshalb bietet die UML spezialisierte Diagramme zur Darstellung unterschiedlicher Aspekte an.

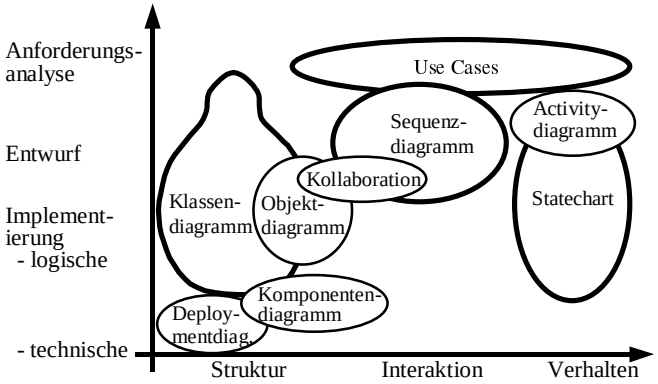


Abbildung 1: Die UML-Diagrammarten

Obwohl die UML keine Vorgehensweise beinhaltet, die allgemein verbindliche Systemsichten definiert, lassen sich den einzelnen Diagrammformen doch bestimmte Entwicklungsaktivitäten zuordnen. Abbildung 1 zeigt die Einordnung der neun UML-Diagrammarten in die Entwicklungsaktivitäten Analyse, Design und Implementierung sowie, für welche der Sichten Struktur, Interaktion und Verhalten sie sich besonders eignen. Die Hervorhebungen spiegeln dabei die derzeitige Bedeutung der Diagramme in der Praxis der Softwareentwicklung wider. Konzeptuelle Überlappungen verschiedener Diagrammarten erlauben die alternative Darstellung bestimmter Sachverhalte mit Hilfe verschiedener Diagramme. Es ist zu erwarten, dass mit dem Entstehen neuer Werkzeuge und Methoden für einige Diagramme weitere Anwendungen entstehen bzw. andere zurückgedrängt werden.

Durch die Anzahl der zur Verfügung stehenden Diagrammarten und die Reichhaltigkeit der darin angebotenen Konzepte entsteht anfangs der Eindruck, dass die UML unglaublich komplex sei. Es ist jedoch keineswegs notwendig, alle neun Diagrammarten zu kennen und zu verstehen, um die UML effektiv zu nutzen. Vielmehr erlaubt die UML, ähnlich einer natürlichen Sprache oder vielen DIN/ISO-Normen, die Konzentration auf eine für die jeweilige Aufgabenstellung hilfreiche Teilmenge der Gesamtsprache. Dadurch kann zum einen die Lernkurve für die UML deutlich verbessert werden. Zum anderen eignet sich die Restriktion auf einen begrenzten Sprachkern der UML häufig, etwa innerhalb eines Projekts oder einer Firma, um die Kommunizierbarkeit von Entwürfen zwischen Entwicklern zu verbessern. Ein solcher Sprachkern besteht beim Einsatz der UML in der Praxis zumeist aus den folgenden vier Modellierungsnotationen[2] oder sogar nur einer Teilmenge davon:

- *Klassendiagramme* zur Strukturdarstellung,
- *Use-Case-Diagramme* zur Darstellung von Anwendungsfällen,
- *Statecharts* zur Verhaltensbeschreibung, und
- *Sequenzdiagramme* zur Darstellung von Interaktion.

Dementsprechend werden diese Diagrammarten ausführlicher besprochen, während die übrigen Diagramme nur in konzentrierter Form vorgestellt werden.

Am meisten verbreitet ist heute die Verwendung der UML zur Kommunikation von Entwürfen zwischen Entwicklern, insbesondere in der Analyse- und Designphase. In diesen Phasen ist es wichtig, keine vollständige, sondern eine elliptische, auf das Wesentliche reduzierte Darstellung zu finden. Vollständige, detaillierte Darstellungen verhindern oft den Blick auf die Kernkonstruktionen (Architektur) eines Systems und erschweren so den Zugang. Im Hinblick auf den Einsatz der UML in technischen Applikationen gewinnt dagegen die Generierung von Code und Testfällen aus UML-Modellen an Bedeutung. Weil die UML in diesem Fall zur Beschreibung der Implementierung verwendet wird, sind hier detaillierte, vollständige Modelle notwendig. Die zur Generierung verwendeten Teile der UML bilden dann eine Programmiersprache, die im Vergleich zu üblichen Programmiersprachen stark kompakt

[2] Bei der Bezeichnung werden bewußt sowohl deutsche als auch englische Namen verwendet. Für einige Bezeichnungen der UML existieren bereits allgemein aktzeptierte deutsche Übersetzungen, bei anderen hat sich der englische Originalname durchgesetzt.

und daher verständlicher sein sollte. Derzeit ist eine Generation von Werkzeugen im Entstehen, die elaborierte Analysemethoden für Modelle anbieten. Dazu zählen etwa die Untersuchung nach bestimmten metrischen Kriterien, Simulation und Animation zur Validierung des Modells gegen Anwenderanforderungen sowie die logische Verifikation von Eigenschaften z.B. mittels Model Checking.

Eine häufig geäußerte Kritik an der UML ist, dass die präzise semantische Integration der Beschreibungstechniken bis heute nicht völlig zufriedenstellend gelöst ist. Vielmehr gibt es in unterschiedlichen Anwendungsbereichen oder verschiedenen Phasen der Softwareentwicklung oft verschiedene Interpretationen desselben Konstrukts, die dort auch jeweils ihre Berechtigung haben. Diese Interpretationsfreiheit ist einerseits problematisch, weil sie Missverständnisse hervorrufen kann. Andererseits kann dadurch eine Anpassung der Sprache auf neue Anwendungsfelder oder unterschiedliche Phasen der Softwareentwicklung vorgenommen werden. Gewisse Interpretationsfreiheiten werden daher von erfahrenen Entwicklern an der UML geschätzt. Leider ist jedoch innerhalb der UML die Möglichkeit, die konkrete Interpretation zu beschreiben, sehr begrenzt[3]. Die an technische Applikationen angepasste Interpretation von Diagrammen der UML wird in Teil 3 eingehender besprochen.

## 2. Diagramme der Kernsprache der UML

Wie bereits bemerkt, kristallisiert sich in der Praxis aus den neun angebotenen Diagrammformen ein Kern von vier essentiellen Diagrammformen heraus. Dieser Artikel fokussiert außerdem auf die wesentlichen Konzepte der UML und erläutert sie anhand des Anwendungsbeispiels im Einführungsartikel [0]. Für weitere Feinheiten und Fähigkeiten der Diagramme sei auf die zahlreich erschienenen Bücher verwiesen [2,3,4,5,6].

Die Stärken der Standard-UML liegen in der Modellierung von Systemen mit komplex strukturierten Daten, jedoch nicht vordringlich in der Modellierung echtzeit-behafteter Prozesse. Die nachfolgenden Beispiele konzentrieren sich deshalb auf den Datenmodellierungsanteil und den damit verbundenen Arbeitsabläufen und Verarbeitungseinheiten des Produktionsprozesses. Bezüglich technischer Aspekte verweisen wir auf den dritten Artikel dieser Reihe.

### 2.1 Klassendiagramme

Abbildung 2 zeigt ein Klassendiagramm zur Modellierung der Daten im Produktionsprozess. Als zentrales Konzept wurde eine Klasse Teil eingeführt, die vier Subklassen A, B, C und D besitzt. Die gemeinsame Oberklasse Teil erlaubt die Modellierung gemeinsamer Eigenschaften der Teile, wie die Verwendung im Produktionsprozess oder die gemeinsame Speicherbarkeit in einem Lager.

[3] Die UML bietet die Konzepte Stereotypen und Tagged Values, um Sprachelementen eine spezifische Bedeutung zuzuweisen. Mechanismen zur Definition von Stereotypen existieren allerdings nicht.

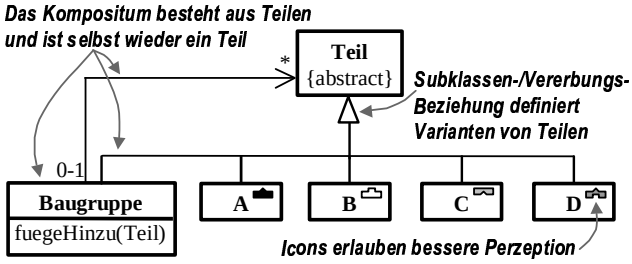


Abbildung 2: Klassendiagramm mit Teile-Ganzes Struktur

Ein Klassendiagramm sollte sich auf wesentliche Aspekte konzentrieren und in sich geschlossen sein. Deshalb sind mehrere Klassendiagramme zur Modellierung unterschiedlicher Systemteile sinnvoll. In der folgenden Abbildung 3 wird eine Lagerverwaltung modelliert. In ihr wird eine geordnete Menge von Teilen einem Lager zugeordnet. Jedes Teil kann in nur einem Lager abgelegt sein.

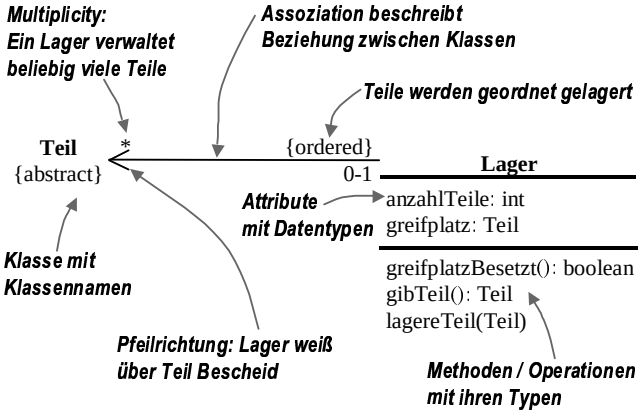


Abbildung 3: Klassendiagramm für Teile und Lager

Wesentliche Kernkonzepte der Klassendiagramme sind *Klassen*, *Assoziationen* und *Vererbungsbeziehungen*. Eine Klasse modelliert eine konzeptuelle Einheit und wird durch ihren Namen, eine Liste von *Operationen* (synonym: Methoden) und eine Liste von *Attributen* beschrieben. Im Beispiel sind in der Klasse Lager einige Operationen zur Verwaltung von Teilen definiert (Holen, Einbringen und Abfrage, ob eines vorhanden ist). Ein Merkmal einer Klasse ist die Fähigkeit zur wiederholten Instantiierung: *Objekte* bilden zur Laufzeit des Systems ein softwaretechnisches Abbild des Produktionsprozesses, weil zum Beispiel jedes Objekt der Klasse „D“ ein entsprechendes Teil in der Produktion repräsentiert.

Eine Assoziation beschreibt eine Beziehung zwischen Klassen bzw. deren Objekten. Die Assoziation in Abbildung 3 modelliert, dass ein Lager beliebig viele Teile enthalten kann (Multiplicity „*“), diese jedoch geordnet aufgehoben werden. Umgekehrt kann sich ein Teil in höchstens einem Lager befinden (Multiplicity „0-1“). Pfeilrichtungen werden vor allem in der Implementierungsphase interessant, denn sie beschreiben, in welcher Richtung ein Zugriff möglich ist, spiegeln aber kein Konzept der modellierten Realität wider.

Die dargestellte Assoziation modelliert die Lagerhaltung. Für den speziell ausgezeichneten Greifplatz wurde das Attribut „greifplatz“ angelegt, auf das über die Methode „gibTeil()“ zugegriffen werden kann. Diese Methode wurde eingeführt, weil das Lager bei Abholen eines Teils

weitere Aufgaben (z.B. Auffüllen des Greifplatzes) durchführen muss und daher eine direkte Abholung durch Auslesen des Attributs nicht zugelassen wird.

Die Subklassenrelation erlaubt die Bildung von Gruppen „ähnlicher“ Klassen. Dieses Konzept wird zum Beispiel in der Biologie zur Klassifikation von Tierarten oder hier zur Bildung von Materialgruppen verwendet. In der Implementierungsphase wird dieses Konzept auch zur Wiederverwendung von Funktionalität und Datenstrukturen eingesetzt. Die Oberklasse „Teil“ in Abbildung 2 ist *abstrakt*: Sie dient nur zur Gruppierung der Teil-Objekte, wird aber selbst nicht instantiiert (analog: es gibt kein Tier, das zur Klasse Säuger, aber keiner Unterklasse gehört).

Ein Standardverfahren zur Modellierung von Teile-Ganzes Beziehungen wird verwendet, um zusammensetzte Produkte sowie Teilprodukte zu modellieren: Entsprechend dem *Composite Pattern* [7] modelliert das Diagramm in Abbildung 2 ein zusammengesetztes Teil („Baugruppe“) durch die Subklassenbeziehung selbst als Teil, so dass es für eine weitere Komposition mit anderen Teilen zur Verfügung steht. Dieses Datenmodell ist sehr flexibel und erlaubt, dass Änderungen wie die Erweiterung um neue Teile oder das Ersetzen eines Teils durch eine Baugruppe oft nur lokale Auswirkungen haben – einer der großen Vorteile echter objektorientierter Modellierung. In der heutigen, von Flexibilität und Wandlungsfähigkeit der Produktionsprozesse geprägten Welt ist es notwendig, die Software schon bei der Erstellung auf spätere Änderungen vorzubereiten. Gute Softwareentwicklung zeichnet sich durch einen Balanceakt zwischen Flexibilität, Erweiterbarkeit und Einfachheit des Entwurfs aus.

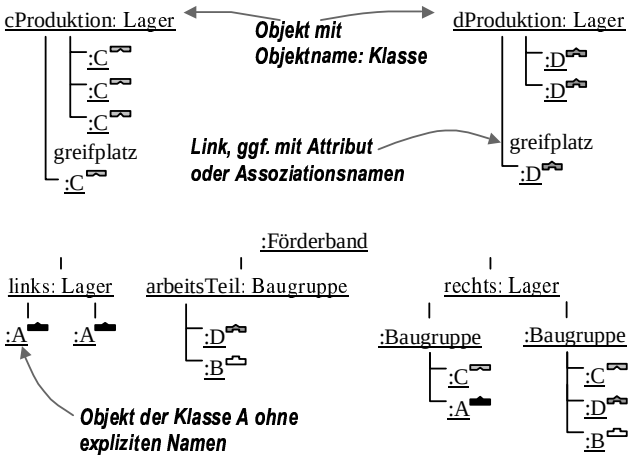


Abbildung 4: Objektdiagramm modelliert eine Situation

Die Klassendiagramme der Abbildungen 2 und 3 drücken weder aus, dass es zwei Lager gibt, noch das jedes Lager nur eine Sorte von Teilen verwaltet. Ersteres lässt sich durch ein Objektdiagramm darstellen, das in Abbildung 4 angegeben ist. Dabei wurde ausgenutzt, dass vom Standpunkt der Datenhaltung aus ein Förderband durch zwei Lager beschrieben werden kann.

Objektdiagramme sind formal Teil der Klassendiagramme, können und sollten aber wie eine eigenständige Notation verwendet werden. Von Klassen können beliebig viele Objekte als Instanzen gebildet werden. Deshalb erscheint z.B. die Klasse Lager im Klassendiagramm nur einmal, in diesem Objektdiagramm aber treten vier Objekte auf. *Links* stellen Verbindungen zwischen Objekten dar, die entsprechend der Assoziationen und Objekt-wertigen Attribute (Bsp: „greifplatz“) ihrer Klassen gebildet werden. Eine Subklassenstruktur der Teile-Hierarchie wird im Objektdiagramm nicht erkennbar, denn die abstrakte Klasse *Teil* wird nicht instantiiert.

Ein Objektdiagramm beschreibt immer nur eine Situation zu einem bestimmten Zeitpunkt. Abbildung 4 stellt die im Einführungsartikel zu dieser Reihe [0] auf S. A4 abgebildete Situation im Montageprozeß dar. Objektdiagramme dienen daher eher zur Illustration: Sie können mittels geeigneter Simulationswerkzeuge eine sehr anschauliche Visualisierung und Animation von Abläufen ermöglichen.

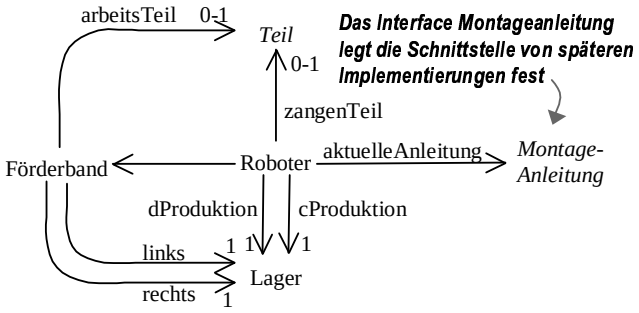


Abbildung 5: Akteure des Produktionsprozesses und die Montageanleitungen

Bis jetzt wurden Klassendiagramme nur zur Datenmodellierung verwendet. Die Modellierung der Akteure im Produktionsprozess – Roboter, Ortschalter, und Diskriminator – erfolgt in Abbildung 5. Das *Interface* Montageanleitung erlaubt es, zum jetzigen Zeitpunkt von der Implementierung von Montageanleitungen zu abstrahieren (nur die Schnittstelle wird definiert) und diese erst in späteren Entwicklungsschritten festzulegen.

Im nächsten Teil der Artikelserie werden alle noch fehlenden Diagrammformen, insbesondere aber Statecharts, Use-Cases und Sequenzdiagramme vorgestellt.

## 2.2 Sequenzdiagramme

Sequenzdiagramme dienen zur exemplarischen Darstellung eines zeitlichen Programmablaufs. Ihr Fokus liegt auf der Interaktion von Objekten, ohne deren inneren Zustand zu beachten. Sequenzdiagramme sind eine Adaption der aus der Telekommunikations-Industrie bekannten MSCs [16]. Das Grundgerüst des Sequenzdiagramms bilden die interagierenden *Objekte,* die wie im Objektdiagramm als rechteckige Kästen dargestellt werden, und die *Lifeline* für jedes Objekt, die das Voranschreiten von Zeit (nach unten) symbolisiert. Die Interaktion der Objekte wird durch *Messages* (Nachrichtenübermittlungen, als Pfeile dargestellt), und Aktivitätsblöcke modelliert. Das Sequenzdiagramm in Abbildung 7 beschreibt den Vorgang des Zusammenbaus eines AC-Produkts.

Der dargestellte Produktionsablauf ist exemplarisch, weil eine unbeschränkte Anzahl alternativer Abläufe möglich ist. Zum Beispiel kann der Greifplatz gerade nicht besetzt sein, das Zusammenfügen zweier Teile fehlschlagen, oder

ein anderes Produkt (BCD) montiert werden. Der Ablauf ist ausschnitthaft, denn er verbirgt z. B. die internen Abläufe im Lager, die parallel zur Montage notwendig sind, um den Greifplatz wieder zu besetzen. Er beschreibt auch nur einen kleinen zeitlichen Ausschnitt im gesamten Systemablauf, der sich beliebig wiederholen kann.

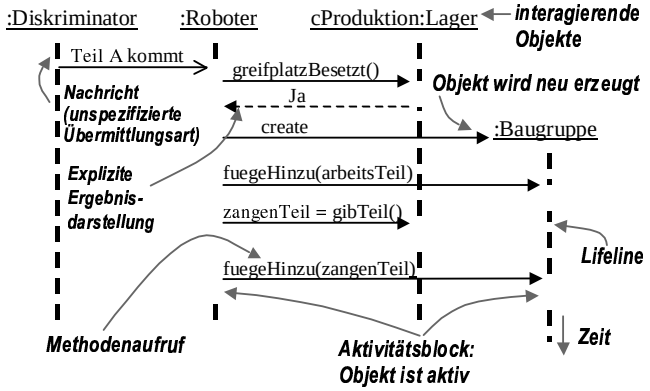


Abbildung 7: Produktionsablauf

Aufgrund der expliziten Darstellung der Zeitachse eignen sich Sequenzdiagramme besonders, um zeitliche Reihenfolgen zu beschreiben. Die Lifeline hat allerdings nur qualitative Bedeutung. Für quantitative Zeitangaben sind in der Standard-UML einige – allerdings noch wenig ausgearbeitete – Annotationsformen vorgesehen. Sind Aktivitätsblöcke dargestellt, so zeigen sie an, dass ein Objekt in eine Aufgabe involviert ist. In implementierungsnahen Diagrammen kann so die Aufrufstruktur dargestellt werden. Eine synchrone Kommunikation, die auf Methodenaufrufen basiert und auch Rückgabewerte erlaubt (gestrichelter Pfeil), ist vor allem für sequentielle Systeme geeignet. In verteilten Systemen wird dagegen eine losere Kopplung durch asynchrone Nachrichtenübermittlung modelliert. Die Entscheidung über die Kommunikationsform kann im Modell zunächst auch zugunsten einer informelleren Beschreibung offen gelassen werden, wie im Beispiel für „Teil A kommt“. Dadurch werden verschiedene Abstraktionsstufen der Modellierung möglich, die auch gemischt werden können.

Ein weiteres Merkmal von Sequenzdiagrammen ist die Darstellung von Objekterzeugung (z.B. einer Baugruppe). Ebenfalls möglich, jedoch nicht in Abbildung 7 dargestellt, sind alternative Zweige und Iterationen. Die Benutzung dieser Mechanismen wird sehr schnell komplexer, so dass die Intuitivität der Sequenzdiagramme dadurch verloren geht. Es ist daher zu empfehlen, solche Mechanismen sparsam einzusetzen. Sequenzdiagramme eignen sich im Entwurf hervorragend zur Visualisierung von einzelnen Szenarien wie den Standard-Abläufen eines Systems. Hilfreich sind Sequenzdiagramme aber auch zur Darstellung von Abläufen bereits existierender Systeme, beim Testen und beim Reengineering.

Die UML hat das klassische Problem einer Breitband-Sprache, die versucht, durch die Einbeziehung unterschiedlicher Kommunikationsparadigmen und – ansatzweise – auch Zeitmodellierung viele verschiedene Anwendungsgebiete zu unterstützen. Deshalb haben sich einige Sequenzdiagramm-Dialekte herausgebildet. Beispielsweise kann idealisierend eine globale Uhr und ein synchrones Voranschreiten der Zeit in allen Objekten und verzögerungsfreie bzw. verzögerungsbehaftete Kommunikation angenommen werden. Zusammenfassend ist festzustellen, dass die UML derartige Details bei der Interpretation von Sequenzdiagrammen in vielen Fällen explizit offen lässt[4]. Diese sind entweder durch das spezifische Projekt in einem Sprach-Glossar oder implizit durch die Wahl eines Werkzeugs festzulegen. Ein Vorteil der Interpretationsfreiheit von Sequenzdiagrammen ist jedoch, dass sie bereits frühzeitig im Entwurf eingesetzt werden können, um bestimmte Interaktionsmuster zu charakterisieren, ohne Implementierungs-Entscheidungen vorwegzunehmen.

## 2.3 Statecharts

Statecharts wurden bereits 1967 von Harel als Erweiterung der Automaten eingeführt [9]. Sie sind heute Basis ausgereifter Werkzeuge zur Modellierung eingebetteter Systeme [17,18]. Statecharts beschreiben das Ablaufmodell einer Komponente der Software. In der UML werden Statecharts häufig eingesetzt, um den Lebenszyklus eines Objekts, also Abhängigkeiten zwischen den Methodenaufrufen und dem *Zustand* des Objekts zu beschreiben. Ein Statechart kann aber auch benutzt werden, um das Ablaufmodell einer komplexen Methode darzustellen.

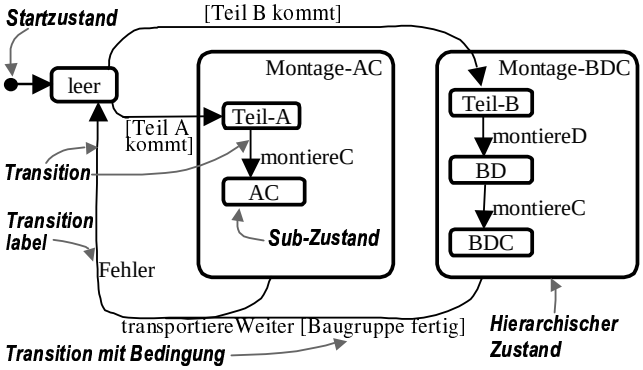


Abbildung 8: Statechart für die Klasse Baugruppe

Das Statechart in Abbildung 8 beschreibt Zustände (als abgerundete Rechtecke dargestellt) der Klasse Baugruppe und wird zur Montage benutzt. Es ist, wie im Entwurf üblich, noch nicht festgelegt, in welcher Form das Statechart von den Methoden der Klasse umzusetzen sein wird, obwohl hier sichtbar ist, dass es sich um einen Lebenszyklus für die Montage handelt. Das angegebene Statechart abstrahiert ausserdem von vielen Einzelschritten indem es diese z.B. in “montiereC” bündelt.

Mit Hilfe hierarchischer Zustände können Zustandsräume zusammengefasst werden. *Transitionen* (als Pfeile dargestellt) sind im allgemeinen mit einem Ereignis oder einem Methodennamen behaftet. Zusätzlich kann eine Bedingung angegeben werden, unter der die Transition erfolgen kann. Über die gezeigten Elemente hinaus kann jede Transition

[4] Offene Punkte bei der Festlegung der Bedeutung eines Diagramms werden in der UML als „Variationspunkte“ bezeichnet und können projektspezifischen Bedürfnissen angepasst werden.

mit einer *Aktion* attributiert sein, die bei der Transition ausgeführt wird. Zusätzlich können Zustände mit *Entry-*, *Exit-* und *Do-Aktionen* versehen werden, die beim Betreten und Verlassen eines Zustands bzw. während des Verweilens in einem Zustand zusätzliche Funktionalität erbringen.

Im Gegensatz zu Sequenzdiagrammen zeigt ein Statechart keine Interaktion zwischen Objekten, sondern modelliert gleichzeitig den Zustand und das Verhalten eines Objekts. Dabei können Statecharts Verhalten vollständig beschreiben, während Sequenzdiagramme meist exemplarisch sind. Leider wird dadurch die Statechart-Notation komplexer und benötigt einigen Lernaufwand.

Abbildung 8 zeigt ein für die Benutzung von Statecharts in der UML typisches, abstraktes Zustandsmodell der Klasse Baugruppe. Eine Detaillierung bzw. Umsetzung des Statechart erfolgt erst durch die Methoden der Klasse. In der Praxis hat sich gezeigt, dass viele Objekte kein komplexes Zustandsmodell besitzen, so dass eine Modellierung mit Statecharts nicht notwendig ist. Dagegen sind Varianten von Zustandsautomaten für eingebettete Systeme sehr erfolgreich im Einsatz. Beispiele sind ROOMcharts [15], Statecharts in Rhapsody [17] oder SDL-Automaten [8]. Statecharts werden dort meist zur detaillierten Beschreibung des Verhaltens eines Objektes, etwa zur Beschreibung aller Schritte des Montageablaufes genutzt. Die detaillierte Verhaltensmodellierung mit ROOMcharts ist ein wesentlicher Bestandteil der UML-RT und wird im nächsten Artikel besprochen.

Während bei eingebetteten Systemen eine Übersetzung des Statechart in Code zumindest für Simulations-Zwecke üblich ist, hat sich in der objekt-orientierten Softwareentwicklung ein andere Verwendung durchgesetzt: In einer abstrakten Modellierung werden zunächst zahlreiche Freiheiten, z.B. die verwendete Kommunikationsform für „Teil A kommt“, offen gelassen. Die Darstellung ist meist nicht vollständig, im Beispiel wurde etwa auf Fehlerbehandlung sowie die Implementierung von Aktionen durch Methoden bewusst verzichtet. Diese Entscheidungen werden in einer späteren Phase bei der manuellen Übersetzung in Code getroffen. OO-Statecharts werden daher meist zur Darstellung essentieller Eigenschaften, nicht aber für alle Implementierungsdetails benutzt.

Ähnlich zu Sequenzdiagrammen ist somit auch die Interpretation von Statecharts in hohem Maße vom Anwendungsbereich und Verwendungszweck geprägt. Auch Statecharts unterstützen verschiedene Kommunikationsformen. In klassischen objekt-orientierten Systemen herrscht der Methodenaufruf vor; in verteilten, eingebetteten Systemen wird nachrichtenbasierte Kommunikation in verschiedenen Varianten verwendet. Üblich sind gepufferte als auch ungepufferte Kommunikationsmechanismen. Wesentlich für die Anwendung ist deshalb auch hier die Auswahl eines für das Zielsystem geeigneten Werkzeugs.

### 2.4 Use Case Diagramme

Use Cases (Anwendungsfälle) werden vor allem in der Analyse verwendet. Sie werden eingesetzt, um die wesentlichen Funktionalitäten eines Systems zu identifizieren und zu gruppieren. Ein Use Case Diagramm zeigt, wie diese Funktionalitäten untereinander in Beziehung stehen und welche *Akteure* (Anwender) involviert sind. Ursprung und Haupteinsatzgebiet von Use Case Diagrammen sind Systeme mit vielen Anwendern, weshalb Akteure normalerweise als Männchen dargestellt werden. In technischen Anwendungen werden Use Case Diagramme bislang seltener eingesetzt, obwohl, wie in der Abbildung 9 gezeigt, die Möglichkeit besteht, auch Systemkomponenten als Akteure darzustellen.

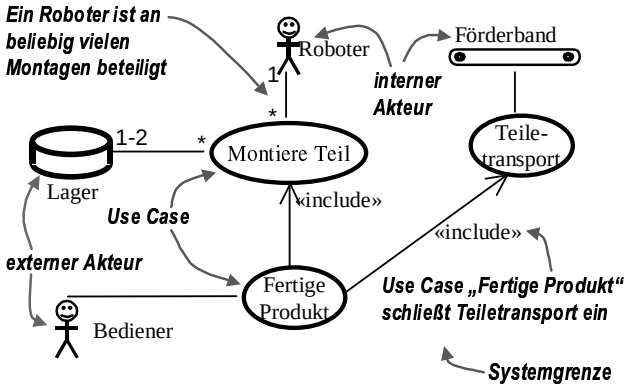


Abbildung 9: Use Case Diagramm des Systems

Abbildung 9 definiert den Use Case „Fertige Produkt“, der auf zwei weiteren Use Cases basiert. Obwohl Use Cases eine informelle Notation sind, haben sie sich als sehr hilfreich zur Strukturierung der Anforderungen und Verantwortlichkeiten eines Systems etabliert. Das Beispiel strukturiert die Funktionalität des zu entwickelnden Systems und legt die Systemgrenze fest. Typischerweise wird eine Modellierung der internen Daten- und Komponentenstruktur erst nach der Entwicklung von Use Cases durch Klassendiagramme vorgenommen. Im Verlauf der Entwicklung werden aus komplexen Use Cases oft eigenständige, mit Sequenzdiagrammen und Statecharts beschriebene Klassen (sog. Steuer- oder Vorgangsklassen), die die für die Realisierung notwendige Funktionalität implementieren.

## 3. Spezialisierte Diagrammformen

In diesem Abschnitt diskutieren wir kurz die spezialisierten Diagrammformen sowie Strukturierungsmechanismen der UML. Aufgrund ihrer spezifischen Anwendung können sie beim Erlernen der UML zunächst zurückgestellt werden.

### 3.1 Activity Diagramme

Activity Diagramme erlauben es, (nicht näher bestimmte) Aktivitäten zeitlich zu ordnen und nicht nur sequentielle, sondern auch nebenläufige Abfolgen darzustellen. Hervorgegangen aus Odell’s Event Diagrammen haben Activity Diagramme Charakteristiken von Datenflussdiagrammen und Petri-Netzen, gelten aber offiziell als Spezialisierung von Statecharts.

Unter allen in der UML zusammengefassten Diagrammarten sind Activity Diagramme die noch am wenigsten

verstandenen. Nach dem derzeitigen Zustand der Activity Diagramme kann keine allgemeine Verwendung empfohlen werden. Trotzdem oder vielleicht wegen des entstandenen Freiraums ist es möglich, dass einzelne Werkzeuge spezifische Varianten dieser Diagrammform erfolgreich einsetzen.

### 3.2 Kollaborationsdiagramme

Mit den Kollaborationsdiagrammen bietet die UML eine inhaltlich den Sequenzdiagrammen sehr ähnliche Notationsform an. Kollaborationsdiagramme werden in der UML sowohl als eine statische Sicht (wer kommuniziert mit wem) als auch als dynamische Sicht (die zeitliche Abfolge von Interaktionen) verstanden. Sie unterscheiden sich von Sequenzdiagrammen vor allem dadurch, dass Objekte zweidimensional angeordnet sind, und die Reihenfolge der Methodenaufrufe durch Nummerierung angegeben wird. Zur Darstellung von Interaktionsszenarien sind sich beide daher sehr ähnlich, unterscheiden sich aber in der Übersichtlichkeit der Darstellung. In der UML-RT werden Kollaborationsdiagramme als rein statische Beschreibung verwendet und zu Capsule-Diagrammen erweitert, um die Kommunikationsarchitektur eines Systems zu beschreiben. Capsule-Diagramme werden in Teil 3 eingehend diskutiert.

### 3.3 Object Constraint Language (OCL)

Immer wieder gibt es Anforderungen oder Einschränkungen, die sich mit Diagrammen nicht oder nur sehr schlecht formulieren lassen. Deshalb wurde die UML um eine textuelle Spezifikationssprache erweitert: Die OCL. Die Syntax der OCL orientiert sich an der objektorientierten Programmiersprache Smalltalk und wurde um Konzepte funktionaler Sprachen zur Evaluierung von Ausdrücken über endliche Mengen und Sequenzen von Objekten erweitert. Aufgrund ihrer speziellen Syntax ist die Frage nach ihrer Lesbarkeit umstritten.

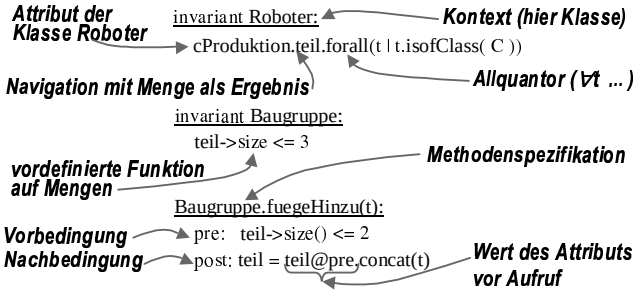


Abbildung 10: OCL Bedingungen

OCL Ausdrücke dienen zur Beschreibung von Invarianten in Klassendiagrammen, sowie von Bedingungen in Statecharts oder Sequenzdiagrammen. Ein weiteres interessantes Verwendungsgebiet für OCL sind Vor- und Nachbedingungen für Methoden.

In Abbildung 10 sind zwei OCL Invarianten und eine Methodenspezifikation angegeben. Die erste Invariante nutzt einen Mechanismus zur Navigation zwischen Objekten, um die Umgebung in die Bedingung einzubeziehen. Sie sagt aus, dass der Roboter vom Lager an der Assoziation cProduktion tatsächlich nur C-Teile bekommt. Die zweite Invariante besagt, dass eine Baugruppe maximal drei Teile beinhaltet. Entsprechend wird für die Methode „fuegeHinzu“ als Vorbedingung gefordert, dass maximal zwei Teile in der Baugruppe vorhanden sind. Es ist Aufgabe der Implementierung sicherzustellen, dass diese Bedingungen erfüllt werden.

### 3.4 Komponenten- und Verteilungsdiagramme

Zwei weitere Diagrammformen – Komponenten- und Verteilungsdiagramme – sind zur Darstellung von implementierungsspezifischen Sachverhalten gedacht. Sie sind jedoch in der UML noch relativ wenig ausgearbeitet. Komponentendiagramme dienen zur Darstellung von Abhängigkeiten zwischen Softwarekomponenten, etwa bei der Kompilierung von Sourcecode. Verteilungsdiagramme beschreiben die physische Verteilung von Komponenten, z.B. auf Rechnernetzen.

Der Begriff der Komponente ist unscharf gefasst: Die Definition lässt unter anderem offen, wie Modelle aus der Analyse- und Designphase, etwa Klassendiagramme, mit Komponenten in diesen Diagrammen in Beziehung stehen. Ebenso ist der Begriff einer Abhängigkeit offen gehalten. Der effektive Einsatz beider Diagramme erfordert daher in jedem Fall eine selbst zu erbringende sorgfältige Definition dieser Begriffe.

### 3.5 Packages

Zur Organisation und Strukturierung von UML-Modellen werden Packages verwendet. Das Package-Konzept wurde aus Java übernommen und adaptiert. Es erlaubt die Zusammenfassung mehrerer Modelle nach beliebigen logischen oder strukturellen Kriterien. Beispielsweise kann ein Klassendiagramm mit zugehörigen Automaten zusammengefasst werden, um damit ein Subsystem zu beschreiben. Packages dienen auch als Mechanismus, um Abhängigkeiten zwischen Modellen, z.B. die Weiterverwendung von Modellteilen in einem anderen Modell, zu verwalten. Der Nutzen des Package-Konzepts hängt stark von der Unterstützung durch Werkzeuge ab.

### 3.6 Erweiterungsmechanismen der UML

Die UML besteht aus neun Notationen, die in unterschiedlichen Kontexten angewendet werden können. Für die anwendungsgetriebene Spezialisierung und Erweiterung dieser Diagrammformen können *Stereotypen* und *Tagged values* verwendet werden. Beispielsweise können Klassendiagramme in verschiedenen Phasen der Entwicklung unterschiedliche Bedeutungen erhalten. In der Analyse stehen Klassen oft nicht für Softwarekomponenten, sondern modellieren Konzepte der realen Welt. Diese finden sich dann in stark veränderter Form in der Implementierung wieder. Klassen können deswegen z. B. mit Stereotypen wie «implementation class» oder «type» ausgezeichnet werden. Die Definition eigener, applikationsspezifischer Stereotypen, wie «technical-class», wird von der UML zwar erlaubt, aber kein Definitions-Mechanis-

mus vorgeschlagen. Dennoch wird ihre Einführung von vielen Werkzeugen unterstützt.

Mit *Tagged values* können den Modellelementen zusätzlich anwendungsspezifische Informationen, z.B. Echtzeitbedingungen in technischen Anwendungen, Versionsnummern oder ein Reviewstatus mitgegeben werden.

## 4. Zusammenfassung

Die UML hat in den letzten Jahren unter den Sprachen für die Entwicklung von Softwaresystemen eine dominierende Stellung erlangt. Ihre Stärken liegen vor allem in der Entwicklung datenintensiver Anwendungen. Aufgrund der sich stark erweiternden Aufgaben der Software in technischen Anwendungen werden zumindest Teile der UML – etwa Klassendiagramme – in zunehmenden Maße auch für diesen Anwendungskreis interessant. Die UML ist jedoch als Sprachfamilie zu verstehen, die jeweils auf projekt- oder firmenspezifische Rahmenbedingungen anzupassen ist. So sind die tatsächlich verwendete Teilsprache zu identifizieren, in der UML explizit offen gelassene Interpretationen von Modellelementen festzulegen oder spezifische Erweiterungen einzuführen. Ein Ausgangspunkt für die Anpassung an technische Anwendungen ist der Dialekt UML-RT, der im nächsten Artikel behandelt wird.

Wir danken Dr. S. Kowalewski, G. Popp und W. Prenninger für eine kritische Durchsicht der Artikel.

### Literatur:

[0] Kowalewski, S.: Modellierungsmethoden aus der Informatik. In: at – Automatisierungstechnik, Heft 09/2001, S. A1-A5, Oldenbourg Verlag, München.

[1] *OMG – Object Management Group*: Unified Modeling Language Specification. V1.4. March 2001.

[2] *Booch, G., Rumbaugh, J. und Jacobson I.*: The Unified Modeling Language User Guide. Addison-Wesley, 1998.

[3] *Rumbaugh, J., Jacobson, I. und Booch, G.*: The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.

[4] *Oesterreich, B.*: Objektorientierte Softwareentwicklung: Analyse und Design mit der UML, Oldenbourg, München, 1998.

[5] *Fowler, M. und Scott, K.*: UML Distilled – Applying the Standard Object Modeling Language. Addison-Wesley, 1997.

[6] *Hitz, M. und Kappel G.*: UML @ Work. Von der Analyse zur Realisierung. Dpunkt.verlag. 1999.

[7] *Gamma, E., Helm, R., Johnson, R. und Vlissides, J.*: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[8] *ITU-T*: Recommendation Z.100, Specification and Description Language (SDL). ITU-T, Geneva, 1993.

[9] *Harel, D.*: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, 1987, 231 –274.

[10] *D'Souza, D. und Wills A. C.*: Objects, Components and Frameworks with UML. The Catalysis Approach. Addison-Wesley. 1998.

[11] *Jacobson, I., Booch, G. und Rumbaugh, J.*: The Unified Software Development Process. Addison-Wesley, 1999.

[12] *Kruchten, P.*: The Rational Unified Process. An Introduction. Second Edition. Addison-Wesley, 2000.

[13] *Coleman, D., Arnold, P., Bodoff, S., Dollin, C. Gilchrist, H. und Jeremaes, P.*: Object Oriented Development – The Fusion Method. Prentice-Hall, 1993

[14] *Henderson-Sellers, B., Simons, A. und Younessi H.*: The OPEN Toolbox of Techniques. Addison-Wesley, 1998.

[15] *Selic, B., Gullekson, G. und Ward, P.*: Real-Time Object-Oriented Modeling. John Wiley & Sons, 1994.

[16] *ITU-T*: Message Sequence Chart (MSC). Z.120:96. ITU-T, Geneva, 1996.

[17] I-*Logix*: Rhapsody-Overview. 2000.
url: http://www.ilogix.com/fs_prod_rhap.htm

[18] Rational Software: Rational Rose Real-Time,.
http://www.rational.com/products/rose/index.jsp, 2001.