XML を用いた
コンポーネントカタログ言語XSCL の開発と評価

松本至由　柳沢祐史

Prototyping and Evaluation of XSCL based on XML

Noriyoshi Matsumoto and Yuji Yanagisawa

目次

第1章 はじめに

1.1 研究の背景

ハードウェアの高速化・低価格化やインターネットの急速な発達で，パーソナルコンピュータの普及はめざましいものがある．これに伴ってソフトウェアの需要も大きくなっており，ソフトウェア開発の効率の向上が重要になってきた．

近年，オブジェクト指向言語やコンポーネントウェアの出現により，ソフトウェア開発技術も急速に変化してきた．これに伴って，インターネットを利用してコンポーネントを流通させ，開発時間やコストを削減しようという動きが注目されている．

1.2 研究の目的

コンポーネントウェアはハードウェア部品のように，動的に組み込み可能ですぐに使えるプラグ＆プレイ型ソフトウェア部品(以下，部品またはコンポーネントと呼ぶ)の再利用を可能にする技術である．

これまでのソフトウェア開発では，ソースコードを再利用する方法が中心であり，技術面とアーキテクチャの両面で再利用を実現する仕組みが不完全であった．この問題点を解決するために複数のオブジェクトを部品としてパッケージ化する技術を提供し，部品間の多彩なインタフェースを標準化することが必要である．コンポーネントウェアはこれらを標準化した．また，現在ではこのような基盤技術の発展と標準が進み，多種多様なコンポーネントが，インターネットなどを利用して提供されるようになった．

インターネット上でコンポーネントを流通するためには，コンポーネントを効率よく公開，提供する必要がある．しかし，インターネット上でコンポーネントを提供しているサイトでは，それぞれに個別のカタログ表現を用いているため，情報の提供と利用の両面で問題があった．そこで本研究では，SCL(Software specification and Commerce Language)と XML(eXtensible Markup Language)を用いて，コンポーネントに関する情報を効率よく提供するカタログ言語 XSCL を開発した．

1.3 本論文の構成

本研究論文の構成は次のようになっている．

第 2 章では，コンポーネントウェアの基礎技術について説明する．XSCL の開発にあたり，使用したコンポーネント技術は JavaBeans である．これは Java 言語で開発されたコンポーネントのことをいう．このため，基礎知識として Java 言語についても説明する．

第 3 章では，SCL についての説明とその問題点について説明する．

第 4 章では, XML について説明する. これは XSCL が, SCL の問題点を解消すべく XML の技術を用いたためである.

第 5 章では XSCL の仕様と記述例を，第 6 章では，XSCL の記述方法と記述内容のについて評価した結果，第 7 章では今後の課題と本論文のまとめを示す.

最後に，本論文で利用した参考文献と，例として作成したプログラムのソースコードを示す.

第2章 コンポーネントウェアの基礎技術

2.1 Java 言語とは

2.1.1 背景

Java 言語は，Sun Microsystems 社によって 1995 年に作成された言語である．当初の目的は，各種家電製品用のソフトウェアを作成できる言語を開発することであり，基盤の異なる環境で実行可能なプログラムを作成することのできるコンピュータ言語の開発であった．これらの開発環境では多種多様なハードウェア／ソフトウェア環境が使われていたため，Java はプラットフォームに依存しない言語にする必要があった．

インターネットの爆発的な成長で，Java 言語は大いに注目されるようになった．これは，インターネットでは，根本的に異なる CPU やオペレーティングシステムなど，数多くの異なる種類のコンピュータが接続されているためであり，プラットフォームに依存しない移植可能なプログラムを作成できる能力は，非常に魅力的であった．

2.1.2 バイトコードと JVM (Java Virtual Machine)

プログラムには，ソースコードとオブジェクトコードの２つの形態がある．ソースコードはプログラムのテキスト版であり，オブジェクトコードはプログラムの実行可能形態である．オブジェクトコードは，特定の CPU 専用であるのが一般的で，このため異なるプラットフォームでは実行できない．しかし Java 言語は，バイトコードを使うことによってこの制限を取り除いている．

Java 言語も他の言語と同様に，ソースコードを作成する．違いはコンパイルの時にある．Java コンパイラは，実行可能コードを作成するのではなく，バイトコードを含んだオブジェクトファイルを作成する．これは特定の CPU に依存しない命令のことであり，JVM (Java Virtual Machine)によって解釈される．

Java のプラットフォーム非依存の鍵は，同じバイトコードをあらゆるプラットフォーム上の JVM で実行できる点にある．その環境用に作成された JVM がある限り，どのような Java プログラムでも実行できる．

2.1.3 アプリケーションとアプレット

Java で作成できるプログラムは，アプリケーションとアプレットの２種類がある．アプリケーションは JVM によって直接実行することができる．アプレットは Web ブラウザ上，あるいは JDK(Java Development Kit)に含まれる Applet Viewer で実行する．多くのブラウザには JVM が組み込まれており，アプレットの実行環境を提供する．一般に，アプレットは Web サーバからユーザのマシンにダウンロードされる．これは，Web ページにアプレットへの参照が含まれていた場合に，自動的に行われる．このようにして，世界中のどの

開発者が作成したアプレットでも，Web サーバから自動的にダウンロードして，クライアントのどのマシンでも実行できる．

Java 言語の開発チームが直面した重要な問題の１つに，セキュリティの問題があった．ユーザのコンピュータにダウンロードされるアプレットは，害を及ぼす可能性がないことが保証されていなければならない．しかし，これはバイトコードと JVM によってセキュリティを実現している．JVM は Java プログラムの実行を制御できるため，権限のない操作を実行することを防ぐ事ができる．この設計は一般にサンドボックスモデルと呼ばれる．

2.1.4 クラスとオブジェクト

クラスとオブジェクトは，全ての Java プログラムの構成要素である．したがってこれらの基本的なことを理解しておく必要がある．

オブジェクトとは，状態と動作の両方を定義する記憶領域のことである．記憶領域はメモリであることもディスクであることもある．状態は，一連の変数とその中の値によって表現される．したがって，オブジェクトとは，データとそれを操作するコードの組み合わせであるといえる．

クラスとは，オブジェクト作成の基盤となるテンプレートのことである．つまり，オブジェクトはクラスのインスタンスであるといえる．新しいオブジェクトを作成することをインスタンス化と呼ぶ．

2.1.5 オブジェクト指向のプログラミング

オブジェクト指向プログラミングの基盤となる３つの言語機能として，カプセル化，継承，ポリモルフィズムがある．ここでは，これらの３つの機能について簡単に説明する．

(1) カプセル化

カプセル化とは，データとデータを操作するコードを関連付けるメカニズムである．コードは，データの周りに施される保護カプセルのようなものと考えることができる．他のソフトウェアからこのデータに直接アクセスすることはできない．

カプセル化には重要な利点がある．第一に，情報を保存するために使用するデータの形式を簡単に変更することができる．たとえば，リンクリストを使って情報を保存しており，パフォーマンスを向上させるために，後からこれをハッシュテーブルに変更することになったとする．このデータ構造へのすべてのアクセスが，厳密に定義された一連のアクセスメソッドによって行われていれば，これらのメソッドを呼び出す他のソフトウェアを修正することなく，データ構造に修正を加えることができる．第二に，機密情報へのアクセスを調整することがとても楽になる．たとえば，データの特定の部分を修正するメソッドでは，ログインとパスワードを受け取るようにすることができる．第三に，複数のスレッドからデータアクセスを同期化することも簡単になる．

(2) 継承

継承とは，あるクラスを特化することによって，ほかのクラスを定義するメカニズムのことである．１つのクラスによってすでに定義されている構造体や動作を基に別のクラスを定義することができるため，ソフトウェアの再利用が促進される．クラス Y がクラス X を継承する場合，Y を「X のサブクラス」と呼び，X を「Y のスーパークラス」と呼ぶ．また，「Y は X を拡張する」と表現することもある．さらに，他のクラスが Y を拡張することもできる．このようにして，一連のクラスに対して継承の階層関係が形成される．

１つのクラスは複数のサブクラスを持つことができる．しかし Java では，１つのクラスは直接的なスーパークラスを１つしか持つことができない．継承階層のルートには Object という名前のクラスがある．したがって，すべての Java クラスは Object クラスを直接的または間接的に継承していることになる．

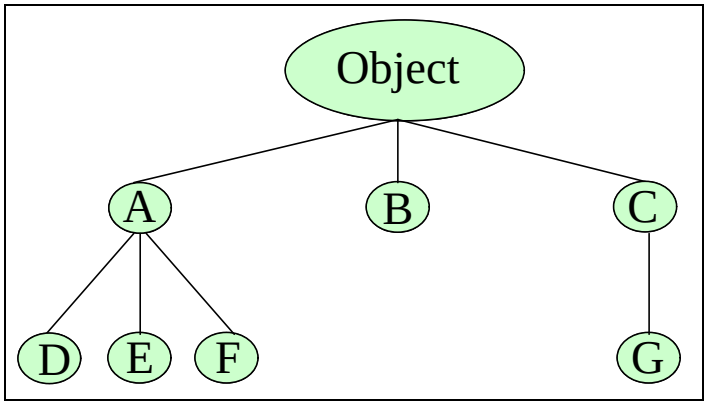


図 2.1.1 クラス継承階層

図 2.1.1 は, Object のサブクラスとして A,B,C というクラスがある. クラス A には D,E,F というサブクラスがある．クラス C はクラス G のスーパークラスである．クラス A が a1,a2,a3 というメソッドを持っているとすると，これらのメソッドはすべてクラス A のサブクラスに継承される．

(3) ポリモルフィズム

ポリモルフィズムとは,「１つのインターフェース，複数の実装」と表現されるメカニズムである．図 2.1.2 に示すクラス階層について考えると，この階層ではクラス Shape が Ellipse, Rectangle, Triangle という名前のサブクラスを持っている．

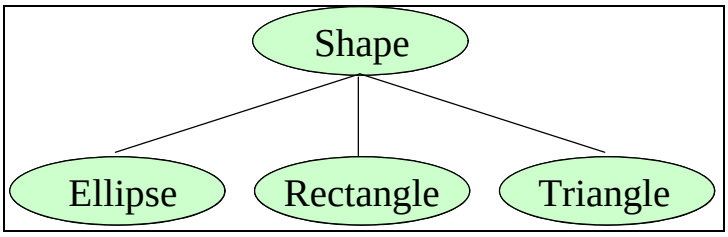


図 2.1.2 クラス継承階層

Shape クラスでは，getArea()というメソッドが定義されている．これは図形の面積を返すメソッドである．しかし，このメソッドはこのクラスによって実装されていない．したがって，これは抽象メソッドであり，Shape は抽象クラスである．つまり，Shape クラスのオブジェクトを作成することはできない．しかし，このクラスの機能はサブクラスによって継承することができる．

Shape クラスの各サブクラスは，getArea()メソッドを実装する．ただし，面積の計算方法は，Ellipse, Rectangle, Triangle の各オブジェクトによって全く異なる．したがって，Shape クラスのオブジェクトがいくつかあり，それらの総面積を計算する必要がある場合は，これらの各オブジェクトの getArea()メソッドを呼び出すことになる．

各種の Shape クラスを管理するための, 100 万行もの Java コードがすでに存在するとする．そして，Trapezoid という名前の Shape クラスの新しいサブクラスを作成する．Trapezoid サブクラスでも，getArea()メソッドを実装する．このとき，既存の 100 万行のコードを検索し，このソフトウェアを更新する必要はない．Trapezoid クラスはすべての Shape クラスが持っているものと想定されるメソッドを実装しているため，それを操作するソフトウェアを修正しなくても正しく動作する．

2.1.6 Java クラスライブラリ

Java 言語そのものに加えて，Java では豊富なクラスライブラリが定義されている．これらのライブラリには数多くの標準クラスとメソッドが含まれており，すべての Java プログラムから利用することができる．入出力や数学演算，ネットワーク，イベント処理，その他さまざまな処理を行うクラスライブラリが用意されている．

表 2.1.1 は，最も広く使われるクラスライブラリをまとめ，その主な機能を簡単に説明したものである．これらの機能は各種パッケージにグループ化されている．パッケージとはクラスの集まりである．パッケージには，ピリオドで区切られた一連の識別子が名前として使われている．例えば，入出力(I/0)ライブラリを含むパッケージは java.io である．

表 2.1.1　Java の主なパッケージ

パッケージ	説明
Java.applet	アプレットを作成する
Jaba.awt	グラフィカルユーザインタフェースを作成するためのAbstract Window Toolkit(AWT)を提供する
Java.awt.event	AWT コンポーネントからのイベントを処理する
Java.awt.image	画像処理を行う
Java.beans	Java ソフトウェアコンポーネントの基盤を形成する
Java.io	入出力をサポートする
Java.lang	Java の中枢機能を提供する
Java.net	ネットワーキングを可能にする
Java.util	ユーティリティ機能を提供する

2.2 コンポーネントウェア

コンポーネントウェア(Componentware)とは，オブジェクト指向に基づきプラグ&プレイ型ソフトウェア部品を組み合わせてシステムを開発する技術体系である．

2.2.1 コンポーネントウェアの利点

コンポーネント技術は非常に有用であり，Visual Basic のような非オブジェクト指向言語環境でもこの仕組みを採用し，大きな成功を収めた．企業の開発者は，必要なコンポーネントを使うことが開発期間の短縮につながり，お金を払ってコンポーネントを購入した方が，開発費用よりも安上がりということに気づいた．一方，ソフトウェア開発者側では，小さなコンポーネントの単位であれば，売り物になるソフトウェアが簡単にでき，巨大企業の支配する市場においても十分戦っていくことができること，つまり商売になることが容易に理解できた．

各コンポーネントはそれぞれの部品として機能する．すべてのコンポーネントは決められた命令規則にしたがう．これによりコンポーネント同士，あるいは開発環境とコンポーネントとが相互に動作する．さらに開発ツール側では，コンポーネントのソースコードにアクセスすることなくそのコンポーネントの機能を調べることが可能となる．また既存の開発環境に対して新規コンポーネントをプラグインすることも可能となる．開発環境内では，開発者が各コンポーネントをカスタマイズすること，さらには，その内容を後で再利用できるよう保存することも可能である．

従来のソフトウェアは「コンパイル→リンク→実行」という開発サイクルにしたがっていた．ソースコードがネイティブの機械語の実行形式へとコンパイルされ，各ソースコードから１つ，あるいは複数のオブジェクトコードが生成される．次に，リンクがオブジェクトコードファイルや，ライブラリファイルを１つにまとめることにより，ネイティブの機械語で書かれた実行ファイルを生成する．最後にユーザが完成したプログラムを実行する．現在の開発環境の多くは，プログラムをコンパイルした後は自動的にリンクが行われる．プログラマ側では，図 2.2.1 に示すように，「コンパイル→実行」というサイクルで行われるものと考えていることが多い．

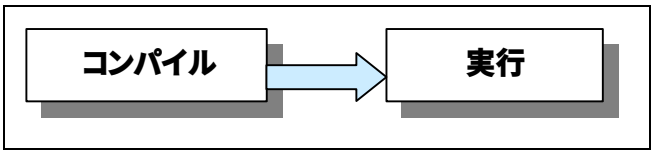


図 2.2.1　従来の開発サイクル

コンポーネントウェア開発においては，全く異なるサイクルが適用される．図 2.2.2 にコンポーネントにおける開発サイクルを示す．

単一のアプリケーションとは異なり，各コンポーネントごとにソースコードは独立しれ記述されている．各コンポーネントはそれぞれ独立してコンパイルされ，テストされてい

る．ここまでがコンパイル時に相当する．

あるプロジェクトに必要なコンポーネントを集めたり，あるいはコンポーネントの開発を始めたときから，アプリケーションのデザインは始まる．開発者は，ツールを使ってコンポーネントを配置し，色や最大値，サイズ，位置などのプロパティ値を変更することでカスタマイズする．また，コンポーネント間にリンクを張ることも可能である．
ここまでが「デザイン時」に相当する．

コンポーネントがプログラムに追加され，カスタマイズされ，リンクを張られることにより，アプリケーションが構築される．一般的に，開発ツール内にコンポーネントを配置したり，関連付けを行うと，設定した内容をテキスト形式のソースコードファイルとして出力される．出力されたファイルはコンパイルされ，ネイティブコードとなる．ここの部分は「ビルド時」に相当する．

最後に，ユーザがコンパイルされたプログラムを実行する．プロジェクトによっては，あらかじめビルド済みのコンポーネントだけを使用することにより，コンパイル時が存在しないケースも考えられる．

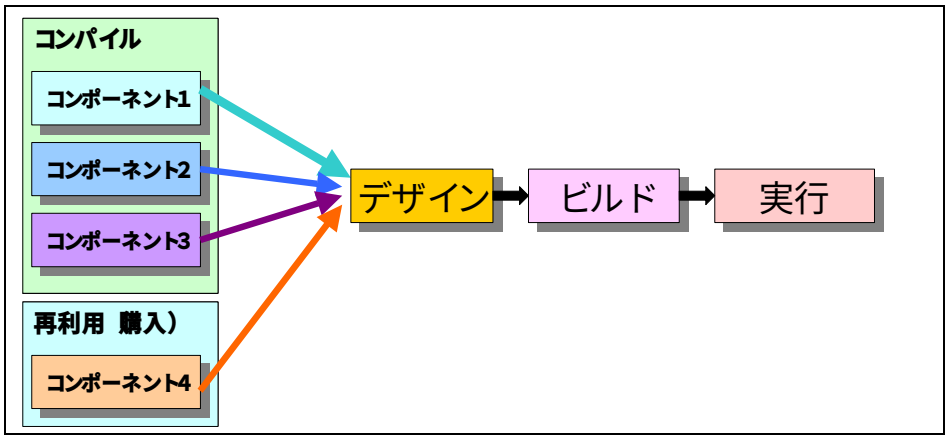


図 2.2.2　コンポーネントの開発サイクル

2.2.2 コンポーネントモデル

コンポーネントモデルとは，アーキテクチャ，API，ソフトウェアにおける一連の命令規則から成り，これらによってプログラマはコンポーネントを作成することが可能になる．複数のコンポーネントは，コンテナと呼ばれるアプリケーション内において互いに接続することが可能である．コンポーネントモデルにおいて，コンポーネント自身，およびコンポーネントを格納するコンテナの構造が定義されていなければならない．コンテナは，複数のコンポーネントが相互に動作するための環境を提供している．多くの場合，コンテナ自身もコンポーネントであり，他のコンテナに含めることも可能である．コンポーネントは様々なものがあるが，必ずしもユーザインタフェースを持つ必要はない．例えば，タイマーや，ネットワークソケットをコンポーネントとして実装することも可能である．

現在のコンポーネントモデルは，次に挙げる基本的な機能のほとんどを提供している．

- カスタマイズ機能
- イントロスペクション機能
- パーシステンス機能
- イベント機能
- コンポーネントのパッケージング機能
- 分散コンピューティング機能

(1) カスタマイズ機能

ボタンのラベルを変えたい場合，新たにコンポーネントを新規に作成しなければならないとしたら，全くの無意味である．コンポーネントは，適切なレベルでカスタマイズ可能でなければならない．ほとんどのインターフェースウィジェットは，色や大きさ，位置をプロパティとして設定可能である．優れたコンポーネントモデルにおいて，コンポーネントが持っているカスタマイズ可能なプロパティの種類は，コンポーネントを利用する側の想像力の範囲に収まるようにデザインされている．

(2) イントロスペクション機能

イントロスペクション機能は，開発ツールにとっては必須の機能である．

開発ツールは，イントロスペクション機能を使うことによって，コンポーネントが持っているプロパティや，各コンポーネントのカスタマイズ方法，ビジュアル環境におけるコンポーネントの表示方法などをあらかじめ知ることなくコンポーネントをロードできるのである．

(3) パーシステンス機能

コンポーネントは，パーシステンス機能を提供する必要がある．この機能により，コンポーネントのプロパティが変更されると，変更状態が保存される．統合開発環境(IDE)を起動するたびに，全プロパティがリセットされてしまうことはない．

(4) イベント機能

コンポーネントは，インタラクティブに動作しなければならない．つまりコンポーネントは，他のコンポーネントに何をすべきかを伝達する機能を持っていなければならないのである．

イベントとは，きわめて大雑把にいえば，非同期に，あらかじめ予測できないタイミングで発生する事象のことである．イベントの発生源のうち，最も一般的なものはユーザであるが，コンポーネントやスレッド，プログラムも他のコンポーネントに対してイベントを送信することが可能である．この場合，イベントを送信するコンポ－ネントを送信元

(source)，イベントを受ける側のコンポーネントのことをリスナ(listener)と呼ぶ．

(5) コンポーネントのパッケージング機能

コンポーネントモデルは，コンポーネントを流通する場合などの目的で，単一のファイルとしてまとめる手段も提供しなければならない．Java においては，jar アーカイブ(Java ARchive)を使うことによって実現可能である．

(6) 分散コンピューティング機能

分散コンピューティング機能とは，異なるマシン上で動作しているコンポーネント間で通信を行うことができる機能である．この機能が加わったのは最近のことであるが，インターネット時代においては基本的な機能である．

分散コンポーネントモデルは，非分散コンポーネントモデルが提供する旧来のあらゆるサービスを提供しなければならない．また，ネットワークを経由してあるホストから別のホストへと実行コードの転送をサポートしなければならない．さらに，該当コードについては妥当なセキュリティモデルを提供する必要がある．

2.2.3 各種コンポーネントモデル

コンポーネントウェアを実現する代表的なモデルとして VBX, ActiveX/DCOM, CORBA, JavaBeans がある．これらのモデルは，全く同じサービスが提供されているわけではない．ここではこれら代表的なモデルについて簡単に説明する．

2.2.3.1 Visual Basic Extension(VBX)

Microsoft 社の Visual Basic は 1990 年代前半から中頃にかけて，最も成功を収め，広く利用されたプログラミング言語である．この言語が成功を収めた背景には，主に次の３つの理由が考えられる．

- BASIC は簡単であること．ほとんどのプログラマがそのことを知っているので，プログラマでない人にとっても学習しやすい言語であった．
- Microsoft 社の製品であったこと．OS 内部の情報を持っていることからも，ユーザ心理からすればシェアは獲得できていたため，勝利を収めるべき言語だった．
- Visual Basic がサポートしているコンポートモデルは，サードパーティのコンポーネントが参入可能な広い市場を形成し，人気を得ることができた．

Visual Basic は VBX と呼ばれるカスタムコントロールという概念を長期に渡ってサポートしてきた．Visual Basic は，製品に添付されている特定のコンポーネントに縛られずに，ほかにも多様な VBX を自由に選択できた．さらに，Microsoft 社はサードパリティが VBX を自分で作成するうえで，必要な情報は十分に公開してきた．このため多くのプログラマ

が VBX を作成し，多くの Visual Basic プログラマに利用され，マーケットが形成された．多くの場合，有能なプログラマを１時間雇うより，VBX の製品を購入する方が安く，繰り返し使用することができる．

このように Visual Basic を使えば，他の人が書いたコードを安く利用できるという恩恵を受けることができる．ただし，言語の実行速度や明晰などとはまた別の話である．どちらも Visual Basic よりは，C や Java の方が優れているためである．

2.2.3.2 ActiveX と DCOM

ActiveX と DCOM は，Microsoft が提供するコンポーネントウェアを実現する技術である．

COM(Component Object Model)とは，メモリ上に保存されるソフトウェアオブジェクトの形式のバイナリ標準と，オブジェクトの内部の動きを効率的に隠すプログラム作成モデルを提供する．これは，言語に依存しない多様な種類のプログラム言語を使った COM オブジェクト開発が可能となることを意味する．

COM を発表した後で，Microsoft は OLE(Object Linking and Embedding)を発表した．OLE の当初の目標は，Windows にドキュメント中心のアーキテクチャを提供することであったが，再利用可能な，統合化されたソフトウェアコンポーネントを作成するための標準的なフレームワークを形成するオブジェクト指向システムのインタフェースサービスとして発展した．OLE は COM のオブジェクトアーキテクチャのうえに構築されているため，基本的には COM のオブジェクト間通信機能を使っている．

OLE は，アプリケーションオートメーション，再利用可能なコントロール，バージョン管理，標準化されたドラッグ＆ドロップ，ドキュメント，オブジェクトのリンクと埋め込みなど，広範囲のサービスを提供する．ActiveX は簡単にいうと，インターネットをサポートするために OLE を拡張したものである．

COM は，ActiveX の基礎となる技術であるが，インターネット上の分散型の処理に必要な機能を完全にサポートしていないことから，Microsoft は DCOM(Distributed COM)を準備した．COM はオブジェクトのバイナリ仕様を定義するが，DCOM はオブジェクト間のリモート通信を処理するためのアプリケーションレベルの問題を扱う．

2.2.3.3 CORBA と IDL

OMG(Object Management Group)とは，ソフトウェアベンダによるオブジェクト技術に関わる企業のコンソーシアムである．このメンバーは，IBM，Apple Computer，富士通といった大企業や，ソフトウェアの開発側・ユーザ側企業 800 社から構成されている．

OMG によって提出された標準が CORBA(Common Object Request Broker Architecture)と呼ばれている．CORBA では，全く異なったプログラミング言語で書かれたオブジェクトに互換性をもたせるための手段を定義している．このために，インタフェ

ース定義言語(IDL，Interface Definition Language)コンパイラを使って，クラスのシグネチャの一覧を作成している．そのほかにもいろいろな点で，CORBA は Java のイントロスペクション機能に類似している点が多くある．メソッドの呼び出しは，ORB(Object Request Broker)を通じて行われる．これにより，互換性のないバイナリ形式のオブジェクトに対しても会話を行うことが可能となる．このようなことが可能になるのは，ネットワーク上の分散システム間で IIOP(Internet Inter-ORB Protocols)を利用しているからである．

2.2.3.4 JavaBeans

JavaBeans(以下 Beans と呼ぶ)は，Java 言語で記述されたコンポーネントウェアである．また Java RMI(Remote Method Invocation)は，異なるマシン上で動作しているオブジェクト間で通信を行う分散オブジェクト環境である．これらの機能を利用することで，コンポーネントを組み合わせた大規模なソフトウェアを構成することができる．ここでは，Beans について簡単に説明する．

Beans は，Java 言語のもつ移植性および機種非依存性，ならびに完全にオブジェクト指向型で，他に類のないセキュリティのため安全に利用でき，ネットワーク分散型といった利点を最大限に利用することができる．

Beans を作成するには，これまでのクラス作成方法に若干のルールを付加するだけでよく，作成が容易である．また，ブリッジと呼ばれる中間に位置する API を用いることにより，Beans を ActiveX などのプラットフォーム固有のコンポーネントに組み込むことが可能となる．Beans の API は，java.beans パッケージとして JDK(Java Development Kit)に含まれている．また，java.awt パッケージに含まれる Java の GUI コンポーネントであるウィンドウやフレームなどの部品も Beans である．

Beans は，プロパティ，メソッド，イベントの３つの要素から構成される．

プロパティとは，名前を持つ属性である．setter メソッドで属性を変更し，getter メソッドで属性の値を取り出す．

メソッドは，その Beans が提供する機能であり，ほかの Beans から利用できるように公開される．後述する BeanInfo に宣言することにより，公開するメソッドを制限できる．

イベントとは，先に述べたように，Beans 内で発生した何らかの事象のことであり，この事象の情報をもつイベントオブジェクトと，事象が発生したときに呼び出されるメソッド宣言の対として定義する．イベントソースは，イベントリスナーを登録，削除するメソッドをもつ．

BeanInfo は公開するプロパティ，イベント，メソッドの情報として，それぞれ，プロパティディスクリプタ(property descriptors)，イベントセットディスクリプタ(event set descriptors)，メソッドディスクリプタ(method descriptors)と，プロパティを編集するためのプロパティエディタをもつことができる．BeanInfo は Beans の情報であり，実行時には

不要である．このため，Beans 本体から独立して生成することが望ましい．

図 2.2-3 に BDK(Beans Development Kit)で提供される Bean Box の例を示す．図の左側にある部品を格納する Tool Box ウィンドウから必要なコンポーネントを中央の Bean Box に配置した例である．コンポーネントが配置されると，そのプロパティが右の Properties ウィンドウに表示される．

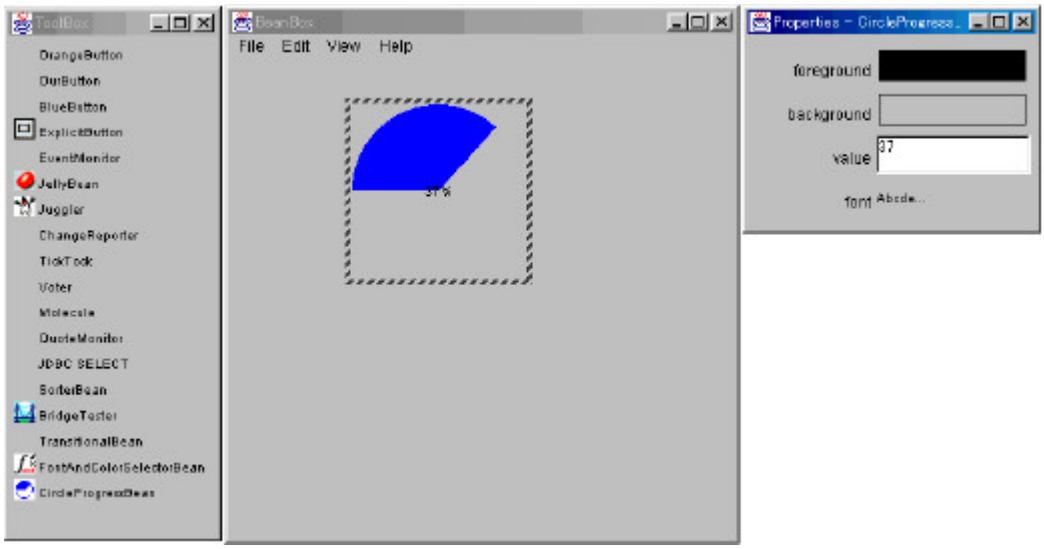


図 2.2.3　BDK の例

実際に，パーセンテージを円グラフで表示する Circle Progress Bean を作成した．このコンポーネントのソースを付録 9.1，9.2 に示す．

次章は，SCL とその問題点について示す．

第3章 SCLとその問題点 について

3.1 SCL とは

SCL(Software specification and Commerce Language)とは，ソフトウェア部品のデジタルカタログ言語のことである．この章では，SCL の構造，メタモデル，文法，HTML による記述方法について説明する．

3.1.1 SCL の構造

SCL の情報構造を，図 3.1.1 に示す．

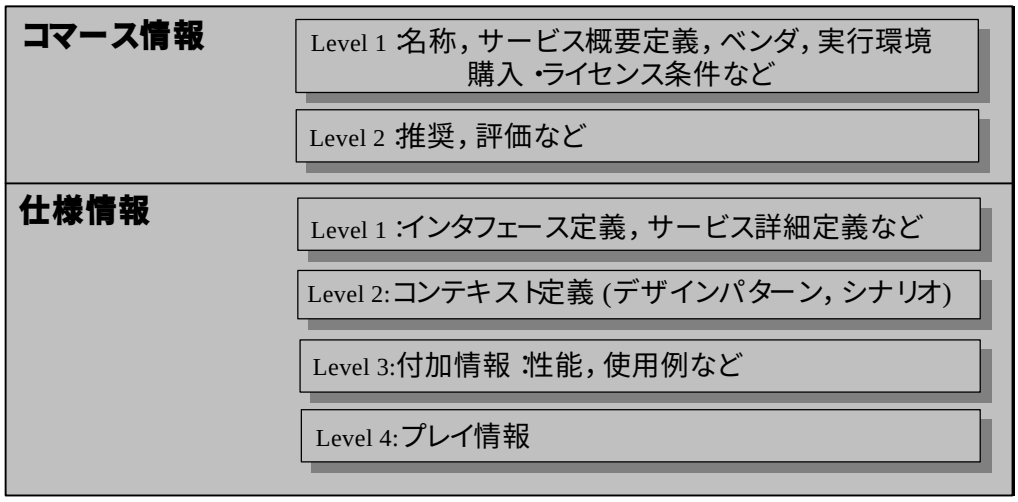


図 3.1.1　表現モデル

SCL は，コマース情報と仕様情報から成り，それぞれ段階的な表現が可能である．また，簡潔さと豊富な情報の記述という相反する要求を満たすよう，それぞれの情報の特性と記述の目的に適した記述方法をとる．

(1)　コマースプロトコル仕様記述部

販売カタログとしての情報を記述する．最小限の情報であり，評価情報以外は省略できない．

(2)　仕様記述部

仕様記述部は次の４つの部から成る．

(a)インタフェース定義
(b)コンテキスト定義：デザインパターン
(c)付加情報部
(d)プレイ情報

3.1.2 SCL のメタモデル

SCL のメタモデル(情報構造)は UML を用いて図 3.1.2 に示す階層構造となる.

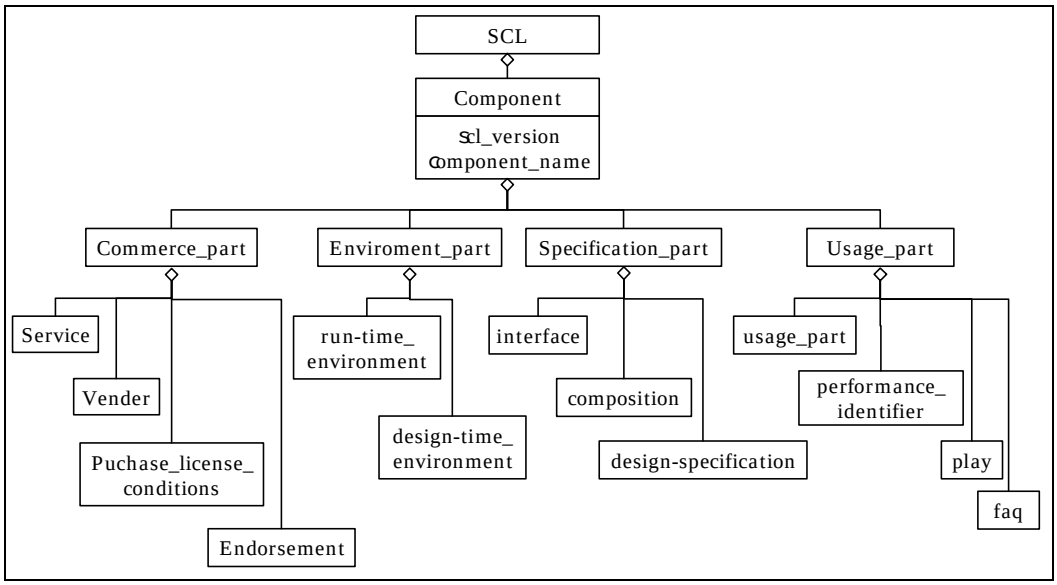


図 3.1.2 SCL のメタモデル

Component の直下にある各パーツは次の意味を持つ.

(1)Commerce Part：商取引のための情報
(2)Environment Part：設計や実行時の環境条件に関する情報
(3)Specification Part：インタフェースの仕様の情報
(4)Usage Part：利用方法やプレイの情報

3.1.3 SCL の文法

SCL の文法を拡張 BNF で示す．ただし，次の表記法を共通に用いる.

<identifier_list>::=identifier>[{ “**,**”<identifier>}*]

<logical_expression>::={<expression>{“**and**”|”**or**”}[“**not**”]<expression>}*

<expression>::=<identifier>[“**not**”]=<identifier>{**+**|*****|**/**}<identifier>|<identifier>

<link_identifier>は http の URL を記述する.

<idl_dcl>は OMG の CORBA IDL2.0 のインタフェース記述に準拠する.

<struture-pattern_dcl>は UML のクラス図を用いる.

<scenario-pattern_dcl>は UML のシーケンス図を用いる.

<text>はテキストによる記述を表す.

<component>::=“**component[コンポーネント]**”
<scl_version_dcl><component_name_dcl><component_dcl>
<scl_version_dcl>::=“//SCL Version:”<version_identifier>
<version_ identifier>::=<identifier>
<component_name_dcl>::=“**name[名称]:**”<identifier>

<component_dcl>::=<commerce_part><environment_part>
<specification_part><usage_part>
<commerce_part>::=<service><vendor><puchase_license_conditions><endorsement>
<service>::=<service_dcl><category_dcl><version_dcl>
<time_of_creation_dcl>[<change-history_dcl>]
<service_dcl>::=“**service[サービス]:**” <text>
<category_dcl>::=“**category[カテゴリ]:**”<idetifier_list>
<version_dcl>::=“**version[版]:**”<version_ identifier >
<time_of_creation_dcl>::=“**time-of-creation[作成日時]:**”<date_time_dcl>
<date_time_dcl>::=year“/”month“/”day“ ”hour“:”minutes“:”seconds
<change-history_dcl>::=“**change-history[更新履歴]:**”
<idetifier_list>|link_identifier
<vendor>::=“**vendor[ベンダ]:**”
<vendor_name_dcl><contact_address_dcl>< distribution_media_dcl>
<vendor_name_dcl>::=“**vendor-name[ベンダ名]:**”
<contact_address_dcl>::=<net_contact_address_dcl>[<postal_contact_adress_dcl>]
<net_contact_address_dcl>::=“**net-address[インターネットアドレス]:**”
<link_identifier>
<postal_contact_address_dcl>::=“**postal-address[住所]:**”
<postal_address_identifier>
<distribution_media_dcl>::=“**distribution-media[提供媒体]:**”<media_list_dcl>
<media_list_dcl>::=”network”|”CD-ROM”|”FDD”
<puchase_license_conditions>::=“purchase-and-license-conditions[購入使用条件]:”
<puchase_conditions><license_conditions>
<maintenance_conditions>
<puchase_conditions>::=“**purchase-conditions[購入条件]:**”<purchase_condition_list>
<purchase_condition_list>::={<price>[<purchase-condition>]}*
<license_conditions>::=“**license-conditions[使用条件]:**”<license_conditions_list>
<maintenance_conditions>::=“**maintenance-conditions[保守条件]:**”
<maintenance_conditions_list>

<endorsement>::=“**endorsement[推奨]:**”<endorsement_list>
<endorsement_list>::={“**recommended-by[推奨者]:**” <identifier>
“**evaluation[評価]:**” <identifier>}*

<environment_part>::=“**environment[環境]:**”
<run-time_environment_dcl><design-time_environment_dcl>
<run-time_environment_dcl>::=“**run-time_environment[実行環境]:**”<environment_list>
<design-time environment_dcl>::=“**design-time_environment[設計環境]:**”
<environment_list>
<environment_list>::={environemnt_identifier“**:**”<name-version_list>}*
<environment_identifier>::=“**os**”|“**object-broker**”|“**web-browser**”
|“**framework**”| “**design-environmemnt**”
|“**language**”|“**processor**”|[{”<identifier>”}*]
<name-version_list>::=<name>”**:**”<version_expression>
[{“**,**”<name>”**:**”<version_expression>}*]
<version_expression>::={“**version**”{**>**|**>=**|**=<**|**<**}<identifier>}|<identifier_list>

<specification_part>::=“**specification[仕様]:**”
<interface><composition>[<design_specification>]
<interface>::=“**interface[インタフェース]:**”
{<idl_dcl>[“**precondition:**”<logical_expression>]
[“**postcondition:**”<logical_expression>]
“interface-service:”<text>}*
<composition>::=“**composition:**”
[{<composition-type_dcl>“**:**”<compsition-interface_dcl>}*]
<composition-type_dcl> ::= <identifier>
<compsition-interface_dcl>::={<identifier> <composition-identifier>
“**from**”<identifier>“**:**”<version-expression>}*
<composition-identifier>::=“**inherit**”|“**aggregate**”|<identifier>
<design_specification>::=“**design-specification:**”
[<strucuture_pattern>][<scenario_pattern>]
<strucuture_pattern>::=“**structure-pattern:**”<struture-pattern_dcl>
<scenario _pattern>::=“**scenario-pattern:**”<scenario-pattern_dcl>

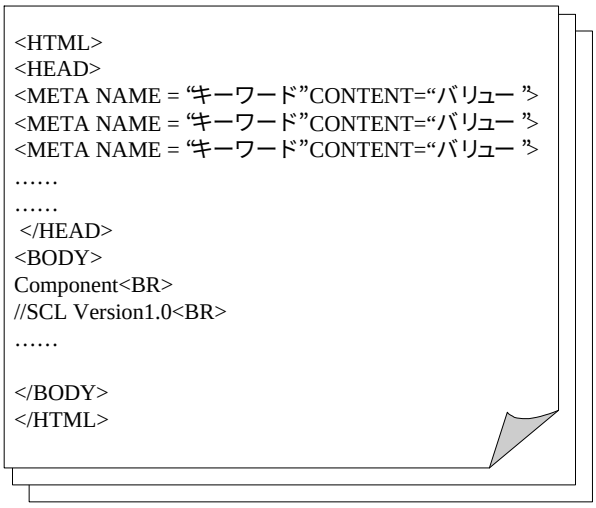
<usage_part>::=“**usage[利用方法]:**”<usage>[<performance>][<play>][<faq>]
<usage>::=[<manual_dcl>][<applications_dcl>]

<manual_dcl>::=“**manuals[利用マニュアル]:**”<text> | <link_identifier>
<applications_dcl>::=“**applications[利用例]:**”<text> | <link_identifier>
<performance>::=“**performance[性能]:**”{performance-identifier“**:**”<expression>}*
<performance-identifier>::=“**storage-size**” | “**memory-size**” | “**execution-time**” |
[{”<identifier>”}*]
<play>::=“**play[プレイ]:**”<link_identifier>
<faq>::=“**faq:**”<text> | <link_identifier>

3.1.4 HTML による記述

「1Web ページ＝1 部品仕様記述」の形式を取る．また部品の記述は，検索エンジンなどを利用できるようにするため，<META>タグを用いて表現し，図 3.1.3 の様な構造を取る．

図 3.1.3　SCL の HTML 表現



HEAD の部分には，<META>タグにより NAME で SCL のキーワード，CONTENT でキーワードに対する値を記述する．ただし，このままでは Web ブラウザによって表示することができないため，BODY 部分に SCL の構文にしたがって記述する．SCL の HTML による記述例は付録 9.3 を参照願いたい．

3.2 SCL の問題点

すでに説明したように SCL は，Web サーバ上でインターネット上に公開するために，表現言語として HTML(Hypertext Markup Language)を用いている．このため，次に示す問題があった．

(1) 意味情報の表現ができない
(2) 要素(情報)が欠落する可能性がある
(3) ページデザインの変更に手間がかかる

上記の問題点について詳しく説明する.

(1) 意味情報の表現ができない

非常に多くの，多種多様なコンポーネントから，目的のコンポーネントを得ようとする場合，検索システムを利用することが多い．しかし，HTML 文書の場合，より確実な検索システムを作るのは非常に困難である．なぜならば，一般に HTML タグは，特定のテキストの固まりを，どのように，またはどのコンテキストの中で表示するかを規定するためのものである．文書中のデータの持つ意味構造，すなわち，そのデータが実際に何を意味しているかは HTML では表現できない．それゆえに，インターネット上で目的のコンポーネントを探し出すのは非常に困難である．例えば，検索エンジンに「red」とタイプすると，「Red Skeleton」「red herring」「the red scare」などへのリンクが見つかるであろう．しかし，中には「Books l've Red」などというページが紛れ込んでいるかもしれない．これは，特定のページ項目が実際に何を意味しているのかが指定されていないからである．このことは，記述のしやすさにも関係してくる．コンポーネントに関する情報以外に，表示方法についても記述しなければならないからである．これについては，第７章　評価で詳しく説明する.

(2) 要素(情報)が欠落する可能性がある

カタログにおいて，情報の欠落は非常に大きな問題である．SCL は必要最小限の情報を提供しているために，各情報が欠落した場合，利用者が損害を被る可能性がある．したがって，要素の欠落をさけるために，パーサによる解析が必要となる．しかし HTML 文書は，ほとんどすべての HTML 要素が，ブラウザ内における情報表示の方法に関連したものである．そのため，HTML はデータ・アプリケーションやアプリケーション・サービスなど，一般的なネットワーク・アプリケーションなどでは役に立たない．つまり，HTML は，データ処理系システム用の情報フォーマットとしてはきわめて利便性が低いのである．したがって，アプリケーションで，要素(情報項目)の欠落の有無を解析させることは困難である.

(3) ページデザインの変更に手間がかかる

HTML 文書のデザインを変更する場合，情報表示の方法に関連した要素を変更する．しかし，HTML ファイルの数が増えることで，その修正個所は莫大な量になってしまう．最新の HTML4.0 では CSS(Cascading Style Sheets)によりある程度スタイル表現の分離ができるようになったが，根本的な表示の修正はできない．これは先に述べたように，HTML

要素は特定のテキストの固まりを，どのように，またはどのコンテキストの中で表示するかを規定するためのものであるためである．

以上の問題点を解決する手段として，SCL の表現に XML を用い，その DTD(Document Type Definition)を作成した．このことによって，以下の事が解決できる．

- 要素の意味表現が可能
- DTD と XML パーサによって要素の解析が可能
- スタイルシート(XSL, eXtensible Style Language)による意味構造と表現の分離

次章では XML について説明する．

第4章 XMLについて

4.1 XML とは

XML (eXtensible Markup Language：拡張可能なマーク付け言語) は，WWW (World Wide Web)のための新しい文書記述言語である．XML も HTML(HyperText Markup Language)も，構造化情報の国際規格である SGML(Standard Markup Language)がベースとなっている．ここで両者を比較すると，以下に示すように，HTML はどのようにデータを表示するかに対し，XML ではデータが何を意味するかを指示する．

HTML

```
<p>P200　ラップトップ
<br>フレンドリーコンピュータ商会
<br>1438 ドル
```

XML

```
<product>
        <model>P200　ラップトップ</model>
        <dealer>フレンドリーコンピュータ商会</dealer>
        <price>1438 ドル</price>
</product>
```

XML では，ブラウザがそこに製品(product)があることを認識し，そのモデル(model)，ディーラ(dealer)，価格(price)を認識する．ブラウザはこうした情報の集合から，最も安価な製品を表示させることなどが，サーバに再度問い合わせることなく可能になる．

XML では，知りたいことを正確に表現するタグを独自に作成できる．これによりクライアント側アプリケーションは，Web 上の任意の場所にある任意の形式のデータソースにアクセスすることが可能になる．新たに「中間層」サーバがデータソースとクライアントの間を仲介し，全てをユーザ自身の XML に翻訳してくれるからである．

HTML のテキストは均一的な方法で表現される単純なテキストに過ぎないが, XML のテキストは知的であるため，その表現を任意にコントロールすることが可能になる．

また，XML は大きく分けて次の 3 つの部分から成り立っている．

- XML 文書
- DTD
- XSL

この章では，これら３つの部分に関して詳しく説明する．

4.2 XML の文法

4.2.1 構文上の規則

XML は，Unicode 文字セットの文字で構成される．こうした文字が連続したものを文字列(string)という．本文中の文字群は，１つの長いテキスト文字列として考えることができる．同様にして XML 文書も，文字列中の文字列から構成されている．

XML も英語などの自然語のように構文を持つ．この節では，XML の構文の基本的な事柄を説明する．

(1) 大文字と小文字

XML では大文字と小文字は区別される(case-sensitive)．つまり，"ELEMENT"という語を挿入するよう XML の仕様で指示されている場合には，"element"や"Element"では不可となる．

HTML や SGML アプリケーションは, 大文字小文字を区別しないよう設計されている．それは大文字小文字の違いによるエラーを増やしたくないという主張である．しかし，大文字小文字といった考え方は，Unicode のような国際的な文字セットにおいては，様々な理由から非常に複雑な問題があると指摘する人々もいる．例えば，大文字小文字に関する変換規則が国によって異なったり，大文字小文字といった概念が全くない言語もあったりするためである．結局，XML の設計において，大文字小文字は区別されるようになった．

(2) マーク付けとデータ

XML 文書はマーク付け(markup)と文字データ(character data)から構成されている．マーク付けとはタグ，実体参照，宣言といった構成子である．これらは，XML プロセッサによって理解されることを前提とする文書の部分である．また，文字データとは，マーク付けとマーク付けの間にある部分であり，一般には人間によってのみ理解されると想定されている．いずれも Unicode で記述され，XML テキストと呼ばれている．マーク付けに関しては，XML 仕様書では下記のように述べられている．

仕様書抜粋 4.2.1 マーク付け

マーク付けは，開始タグ，終了タグ，空要素タグ，実体参照，コメント，CDATA セクション の区切り子，文書型宣言ならびに処理命令の形を取る．

(3) 空白

XML マーク付けの中で，XML プロセッサが異なった扱いをする空白(white space)と呼ばれる文字群がある．これには，半角スペース，タブ，復帰，改行などがある．これらは，Tab キー，Enter キー，スペースバーなどに対応する．

XML の仕様で空白が許されている箇所では，これらの文字を任意に使うことができ，印

刷された際に見やすいよう段落間に行を２つ挿入したりできる．これらの文字は，該当文書が処理される際には無視される．

空白が意味を持つ場合もある．例として，文書中の語と語の間の空白は削除されてしまうと読みにくくなってしまう．したがって XML ではマーク付け内の空白は常に保存され，マーク付けの外の空白は，保存されることも，無視されることも，時には結合されることもある．

(4) 名前と名前トークン

XML を使う際には，名前(name)をつけなければならないことがよくある．論理的構造体には要素型名をつけ，再利用可能なデータには実体名をつけ，特定の要素には ID をつけるなどである．XML における名前に関する規則は以下のようになる．

仕様書抜粋 4.2.2 名前

名前(name)は，字(日本語のひらがな，カタカナ，漢字も含まれる)もしくは少数の区切り子で始まり，その後に，字，数字，ハイフン，下線，コロンまたはピリオドが続く．これらをすべて含めて，名前 文字という．"XML"(大文字小文字は区別せずにマッチされる)という文字で始まる名前は，この仕様の 現在，または将来のバージョンでの標準化のために予約される．

これに関連して，名前トークン(name token)と呼ばれる構文上の構成子がある．名前トークンは，数字，ハイフン，ピリオド，文字列”xml”で始まっても構わないという点を除いて，名前と同じである．

仕様書抜粋 4.2.3 名前トークン

名前トークン(name token)は，名前文字の任意の組み合わせとする．

言い換えると，妥当な名前はすべて妥当な名前トークンであるが，下記に示すように妥当な名前でない名前トークンもある．

例 4.2.1 名前トークンの例

.1.a.name.token.but.not.a.name
2-a-name-token.but.a-name
XML-valid-name-token

名前と名前トークンは，大文字と小文字を区別してマッチされる．名前や名前トークン中では, 空白, 大多数の区切り文字や, その他の特別な文字以外が名前文字(name character)となる．

(5) 文字列リテラル

文書のテキスト中では，区切り子や空白を利用できる必要があるが，こうした文字がマーク付けの内部でも必要になる場合がある．例として，ある要素がハイパーリンクを表し

ており，URL を含める必要がある．この場合 URL はマーク付け内に入れることになるが，通常マーク付け内では，名前文字以外の文字は許されていない．

そこで，文字列リテラル(literal strings)は，マーク付け内で名前文字以外の文字を使うことを可能にする．ただし，こうした文字が必要となる文脈に限られる．例として，ハイパーリンクで URL を指定するには，スラッシュ文字が必要となる．この要素の例を下記に示す．

```
<REFERENCE URL="http://ies045.iee.niit.ac.jp/~team-a/xml/document.xml">
```

ここで URL を定義しているのは文字列リテラルである．文字リテラルは，常に一重引用符か二重引用符のいずれかで囲まれ，引用符は文字列の一部にはならない．

仕様書抜粋 4.2.4 リテラルデータ

> リテラルデータは，引用符で囲まれた任意の文字列とし，その文字列の区切り子として使用される引用符は含まない．リテラルは，内部実体，属性値，外部識別子の内容を指定するのに使用する．

XML 文書中でこれらの文字列の開始と終了を示す区切り子としては，一重引用符(')あるいは二重引用符(")のいずれを使っても構わない．ただし，開始と終了と出引用符の種類を一致させる必要がある．文字列リテラル中には，区切り子として使用した引用符とは別の種類の引用符を挿入しても構わないが，この場合にその引用符は特別な意味は持たない．また，二重引用符で囲まれたリテラル中に二重引用符を含める方法もあるということを心に留めておく必要がある．これは，単一の文字列リテラル中に両方の種類の引用符がまれに必要になる場合があるためである．

4.2.2 前書きとインスタンス

ほとんどの文書は，その文書をどのように解釈すべきかなど，実際の文書に関する情報などを含むヘッダから始まる．例えば，HTML では HEAD というようにその中に TITLE や META といった要素を含めることができる．

同様に XML 文書も，2 つの主要な部分に分けられる．前書き(prolog)と文書インスタンス(document instance)である．前書きは，XML のバーションや文書型など，その文書に関する情報を提供する．文書インスタンスは前書きの後に続き，要素の階層として構成された実際の文書データが含まれる．

仕様書抜粋 4.2.5 文書

```
Document  : : =  prolog element Misc*
```

4.2.3 論理構造

XML 文書の実際のコンテンツは，文書インスタンスに記述する．文書インスタンスと呼ばれるのは，それが DTD を持っている場合には，DTD で定義された文書クラスのインスタンスに相当するためである．特定のメモは「メモ文書」というクラスのインスタンスである．「メモ文書」の正式な定義は，メモ DTD の中に記述する．

簡単な XML 文書の例を下記に示す．

```
<?xml version="1.0"?>
<!DOCTYPE MEMO SYSTEM "memo.dtd">
<memo>
<from>
    <name>Matumoto</name>
    <email>i9612080@cc02.cc.niit.ac.jp</email>
</from>
<to>
    <name>Yanagisawa</name>
    <email>i9612088@cc01.cc.niit.ac.jp</email>
</to>
<subject>メモの例</subject>
<body>
<paragraph>人とアプリケーションの両方がその目的に添って理解し，処理できる形式で記述すること，これを行うのが<emphasis>XML</emphasis>です．つまりそれは，<emphasis>汎用的なデータ記述言語</emphasis>と言えます．
</paragraph>
</body>
</memo>
```

コンピュータは，最初にタグ(tag)つまり，小なり不等号(<)と大なり不等号(>)との間のマーク付けを見る．タグは，各種の要素の始まりと終わりの区切りを示す．そしてコンピュータは，要素群を一種の木として考える．

図 4.2.1 はこの文書の論理構造を表したものである．ここで memo という要素を，文書要素(document element)あるいはルート要素(root element)という．

文書要素(memo)は全体としての文書に相当し，他のすべての要素は，文書の構成要素に相当する．文書の論理構造が要素型の名前に表され，これを，要素のセマンティクス(semantics)という．マーク付け文書を読んだり書いたりしている際に「これはなにを意味するのか？」「これはどんな外観をしているのか？」と考えている場合，セマンティクスについて質問していることになる．文書型の設計者は，何らかの方法でこのセマンティクス

を文書作成者に説明する必要がある．しかし，コンピュータが処理するのは，その要素をどのようにフォーマット化すべきか，どのように処理すべきかといったことであり，これらはスタイルシートやコンピュータプログラムによって指示される．

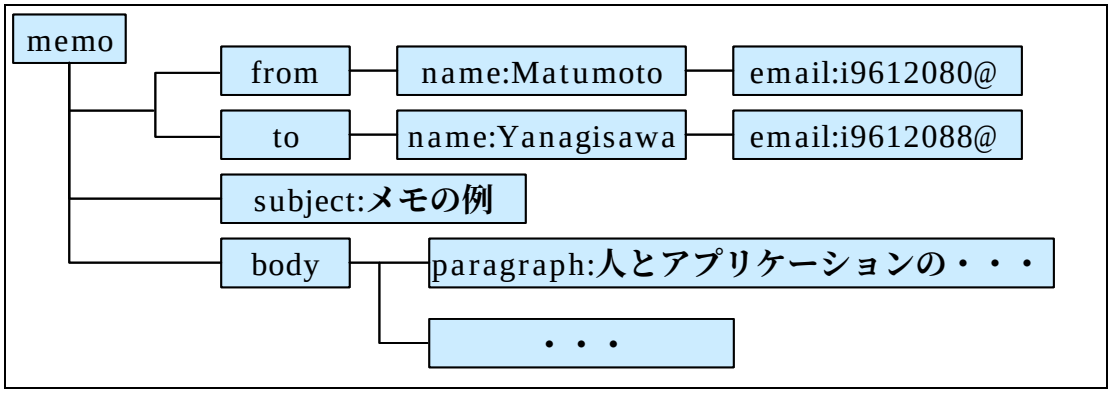


図 4.2.1　memo 文書の木構造表現

4.2.4 要素

要素は，内容を持つものと内容を持たない空要素(empty element)のいずれかで，内容を持つ要素は，開始タグ，内容，終了タグによって表される．

内容をもつ要素の例を下記に示す．

例 4.2.2　簡単な要素の例

```
<title>これはタイトルです</title>
```

例 4.2.2 では「これはタイトルです」が内容となり，開始タグは<title>で，その共通識別子(GI, Generic Identifier)は title である．同様に終了タグは</title>であり，終了タグは開始タグと必ず同じ GI を指定する必要がある．

空要素は，開始タグと終了タグの間に何も書かないか，もしくは単一の空要素タグ<EmptyTag/>といった形で表される．ある要素の型は，常にそのタグ内の共通識別子によって特定される．

要素型と共通識別子とを区別する理由は，「共通識別子」という用語は XML 文書の構文(つまり現実文書の文字群)に関連した用語だからである．「要素型」という用語は，現実文書を構成する構成要素の特性を指す．

4.2.5 属性

内容の他に，要素は属性(attribute)をもつことができる．属性は，文書の要素に特性あるいは特質を割り当てるための方法の一つであり，名前(name)と値を持つ．

DTD では，各要素型に対して個々の属性を定義することも許されている．複数の要素型が同じ名前の属性を持つことも可能であるが，厳密に言うと，それらは別物である．ところが，それを「同じ属性」として考えた方が便利なときもある．

属性はセマンティクスも持つ．属性は必ず意味を持ち，その例として，height という名前の属性は，person という要素に対して割り当てられ，数である値を持ち，その人物の cm 単位での身長を表すといった具合である．

例として，person 要素の属性を下記に示す．

```
<person height="165cm">Dale Wick</person>
<person height="165cm" weight="165lb">Bill Bunn</person>
```

この例で分かるように，他のリテラル値同様，属性の値は一重引用符もしくは二重引用符で囲む．

DTD は，属性を指定できる場所と，その属性で指定できる属性値を制約する．ある属性を，すべての要素に対して指定しなければならない場合もあり，その属性を指定しなかった場合，妥当性エラーとなる．

通常，空要素は属性を持つ．また，下位要素を持つ要素を，空要素と属性の形で表現できる場合もある．

4.2.6 前書き

XML 文書は，XML のバージョン，その他の文書の特性を記述する前書きで始める．

前書き(prolog)は，XML 宣言(XML declaration)と文書型宣言(document type declaration)とから構成されるが，これらはいずれもオプションであり，指定しなことも可能であるが，指定した方が処理の信頼性が高まる．

両方指定する際は，文書型宣言よりも前に XML 宣言を記述しなければならない．また，この 2 つの宣言の間には，コメント，処理命令(PI)，空白を挿入することができる．前書きは，最初の開始タグが始まった時点で終了する．

下記に，簡単な前書きの例を示す．

```
<?xml version="1.0"?>
<!DOCTYPE DOCBOOK SYSTEM "http://ies045.iee.niit.ac.jp/~team-a/xml/docbook">
```

4.2.6.1 XML 宣言

XML 宣言は，いくつかの部分から構成されるが，それらを順番に指定するだけである．

仕様書抜粋 4.2.6 XML 宣言

```
XMLDecl : : = ‘?xml’ VersionInfo EncodingDecl? SDDecl? S? ‘?>’
```

最小の XML 宣言は下記のようになる．

```
<?xml version="1.0"?>
```

下記はより拡張したもので，すべての部分を指定している．

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

XML 宣言と属性との重要な相違は，属性の場合はどのような順序で指定しても構わないのに対し，XML 宣言では，各部分を決められた順序で指定しなければならないという点にある．

(1) バージョン情報

XML 宣言のバーション情報部では，使用している XML のバーションを宣言する．これはすべての XML 宣言で必項である．XML 宣言がオプションとなっている唯一の理由は，「下位互換性」に対処するためであり，あくまでも一時的なものである．

XML のバージョン情報は,自己特定的(self-identifying)な情報表現の一部である．これは，XML 文書を見ただけで，それが XML であり，XML のどのバージョンにしたがっているかが分かることを意味する．

(2) 符号化宣言

XML 宣言には，符号化宣言(encoding declaration)を含めることも可能である．これはどのような種類の文字符号が利用されているかを示す．これは自己特性の別の面である．もし，従来の 7 ビット ASCII で符号化されているなら，この符号化を考慮することは必要ない．7 ビット ASCII は，XML プロセッサが自動的に認識できる UTF-8 という Unicode 符号化のサブセットだからである．

(3) スタンドアロン文書宣言

スタンドアロン文書宣言(standalone document declaration)は，該当文書を完全に処理するのに，文書型宣言のどのような構成要素が必要になるのかを宣言する．それは，XML プロセッサが文書を「厳密に正しく」処理するのに，DTD の外部の宣言を参照する必要があるかどうかを指定するためのものである．この宣言は，値として yes あるいは no をとる．

4.2.7 その他のマーク付け

(1) 定義済み実体

XML 文書を作成している際に，ある特定の文字をマーク付けとしての解釈から保護したい場合がある．例えば，HTML 文書のソースを説明したい場合などである．しかし，これらのマーク付けは現在作成している XML 文書の一部と解釈される．そこで，定義済み実体あるいは CDATA セクションを使う．

定義済み実体(predefined entity)とは，開始タグなど本来なら特殊な意味を持つものとして解釈されるはずの文字を表現するために利用される，XML のマーク付けである．XML では 5 つの定義済み実体がある．表 4.2.1 にその一覧を示す．

表 4.2.1　定義済み実体

実体参照	文字
&amp;	&
&lt;	<
&gt;	>
&apos;	‘
&quot;	“

仕様書抜粋 4.2.7　定義済み実体

> アンパサンド文字(&)ならびに小なり不等号(<)は，マーク付けの区切り子として，あるいはコメント，処理命令，CDATA セクションの中で使われる場合のみそのままの形で出現してよい．(中略)これらの文字が他の場所で必要な場合には，数値による文字参照か，それぞれ"&"ならびに"<という文字列を使うことによって別扱いしなければならない．

仕様書抜粋 4.2.8　属性値

> 属性値の中に一重引用符と二重引用符の両方を含めるには，アポストロフィすなわち一重引用符(')は "'"として，二重引用符は"&quot"として表現する．

XML プロセッサは文書を構文解析する際に，実体参照を実際の文字で置き換える．XML プロセッサは挿入する文字を，マーク付けではなくプレーンな文字データとして解釈する．

(2)　CDATA セクション

実体参照は確かに便利であるが，文書の作成者が後で文書を編集し直す際は非常に見にくく，解読に時間がかかってしまう．

CDATA は文字データ(Character DATA)を表し，CDATA セクションと呼ばれる構成子は，ひとまとまりのテキストをマーク付けが含まれないものとして解釈するよう，XML プロセッサに指示する．

あるセクションを文字データとして印すには，下記に示す特殊な構文を使う．

```
'<![CDATA[' content ']]>'
```

(3)　コメント

コメント(comment)は，文書やマーク付けに関する情報を，コンピュータによる処理や文書の表示からは無視される方法で埋め込むために使用する．コメントはアプリケーションから隠され，ブラウザ上に表示されることも，検索エンジン中でインデクスを付けられる

ことも，文書のデータの一部として処理されることもないが，メタデータとして扱われる可能性はある．

コメントは，"<!--"という文字列と，それに続くほとんど任意のものと，終わりの"-->"で構成される．ただし，コメント中に"--"という文字列を含めることはできない．

```
<!-- このセクションは非常によくできています 変更しないようにしましょう．-->
```

4.3 文書型定義

4.3.1 文書型宣言

XML の仕様書では，文書型を DTD(Document Type Definition：文書型定義)を用いて宣言する．DTD は，許される要素型，属性，実体を定義し，これらの組み合わせに関する制約を表現できる．

例 4.3.1 は，文書型宣言をもつ XML 文書の例である．文書型宣言と文書型定義があり，それに続いて，その文書型のインスタンスがある．

例 4.3.1 文書型宣言をもつXML 文書

```
<!DOCTYPE  label[
        <!ELEMENT  label (name,street,city,sate,country,code)>
        <!ELEMENT  name (#PCDATA)>
        <!ELEMENT  street (#PCDATA)>
        <!ELEMENT  city (#PCDATA)>
        <!ELEMENT  country (#PCDATA)>
        <!ELEMENT  code (#PCDATA)>
]>
<label>
<name>Rock N. Robyn</name>
<street>Jay Bird Street</street>
<city>Baltimore</city>
<country>USA</country>
<code>43214</code>
</label>
```

文書型宣言は第１行から始まり，"]>"で終わる．DTD 宣言は，"<!ELEMENT"で始まる行であり，これらは要素型宣言(element type declaration)である．DTD では他に，属性，実体，記法も宣言できる．

例 4.3.1 では宛名を宣言する DTD 宣言は，すべて文書実体の中に書かれているが，DTD はその一部，あるいはその全てを別の場所で定義することも可能である．この場合の文書

型宣言は，DTD 宣言が記述されている別の実体への参照を含むことになる．

外部の DTD 宣言だけをもつ文書の文書宣言は，例 4.3.2 に示すような形になる．

例 4.3.2　外部 DTD 宣言をもつ文書型宣言

```
<?xml  Version="1.0"?>
<!DOCTYPE  label  SYSTEM  "http://www.niit.ac.jp/dtds/label.dtd">
<label>
...
</label>
```

システムというキーワードにより，次に記述してある URI から外部の情報リソースを参照するよう XML プロセッサに指示している．

例 4.3.2 により参照された外部の情報リソースは, label の DTD を定義する宣言で構成され，その内容は例 4.3.1 と同様である．しかし，こちらは，いくつもの宛名(label)文書で再利用できる点で大きく異なる．

全ての文書型宣言は，<!DOCTYPE という文字列で始まり，続いて DTD で定義される要素型の名前が来る．文書インスタンスのルート要素は，この文書宣言中で宣言された型の要素でなければならない．DTD 宣言の一部が外部に格納されている場合には，文書型宣言の第 3 の部分は，SYSTEM もしくは PUBLIC になる．これが SYSTEM の場合には，それに続く最後の部分は，その外部宣言を示す URI でなければならない．PUBLIC ではそれは公開識別子で記述する．

仕様書抜粋 4.3.1　DOCTYPE 宣言

```
Doctypedecl  : : = '<!DOCTYPE'  S  Name  (S  ExternalID)?  S?  ('['
                        (markupdecl  |  PEReference  |  S)*  ']'  S?)?  '>'
ExternalID  : : =  'SYSTEM'  S  SystemLiteral
                      |  'PUBLIC'  S  PubidLiteral  S  SystemLiteral

markupdecl  : : =  elementdecl  |  AttlistDecl  |  EntityDecl
                      |  NotationDecl  |  PI  |  Comment
```

4.3.2 内部サブセットと外部サブセット

ここでは内部サブセットと呼ばれる内部の部分と外部サブセットと呼ばれる外部の部分とを組み合わせる方法を説明する．

例 4.3.3 に，DTD の例を挙げる．

例 4.3.3　ガレージセールの案内のDTD

```
<!ELEMENT GARAGESALE (DATE,TIME,PLACE,NOTES)>
<!ENEMENT DATE (#PCDATA)>
<!ELEMENT TIME (#PCDATA)>
<!ELEMENT PLACE (#PCDATA)>
<!ELEMENT NOTES (#PCDATA)
```

ここでは 5 つの要素型が宣言されており，これらの宣言文は，例えば garage.dtd という別のファイルに格納し，この DTD に適応させたい全ての文書の文書型宣言中でこのファイルを指定する方法がある．

例 4.3.4　ガレージセールの文書

```
<!DOCTYPE GARAGESALE SYSTEM "garage.dtd">
<GARAGESALE>
<DATE>February 29, 1999</DATE>
<TIME>7:30 AM</TIME>
<PLACE>249 Cedarbrae</PLACE>
<NOTES>Lots of high-quality junk for sale!</NOTES>
</GARAGESALE>
```

例 4.3.4 では DTD のファイル名だけを指定した．実質的にはこれも URL で，相対 URL になっている．テスト中は，完全なサーバをインストールする必要がないため，相対 URL の方が便利である．

しかし，DTD と文書インスタンスとをもっと近くに置く方法がある．DTD を，文書インスタンスと同じ実体内に組み入れるのである．

例 4.3.5　DTD を文書インスタンス組み入れる

```
<!DOCTYPE GARAGESALE
<!ENEMENT DATE (#PCDATA)>
<!ELEMENT TIME (#PCDATA)>
<!ELEMENT PLACE (#PCDATA)>
<!ELEMENT NOTES (#PCDATA) ]>
<GARAGESALE>
.....
</GARAGESALE>
```

"["と"]"の間の部分を，文書宣言の内部サブセット(internal subset)という．これは，実体間を移動することなく，文書インスタンスと DTD とを編集できるため，テストの際には非常に便利である．

2 つのアプローチを組み合わせる場合も多くある．DTD 宣言の一部は再利用可能な外部

実体に置き，他の部分は文書インスタンスと同じ実体内に置くのである．多くの場合，画像の実体などは内部サブセットで宣言される．これは各個別の文書に特有な場合が多いからである．一方，要素型宣言は通常，文書型宣言の外部である外部サブセット(externl subset) に置かれる．

例 4.3.6　外部サブセットへの参照

```
<!DOCTYPE GARAGESALE SYSTEM "garage.dtd"
[
<!ENTITY LOGO SYSTEM "logo.gif">
]>
<GARAGESALE>  .....  </GARAGELASE>
```

内部サブセット中の宣言は，外部サブセット中の宣言よりも前に処理される．これにより文書の作成者は，共有する DTD の一部の宣言を上書きすることが可能となる．

ここで注意しなければならないのは，外部サブセットを格納しているファイルだけが DTD ではないということである．内部サブセットと外部サブセットを組み合わせたもが，DTD 宣言の全体となるのである．

4.3.3 要素型宣言

要素型宣言は，"<!ELEMENT"という文字列で始まり，宣言される要素型の共通識別子である名前がそれに続く．そして最後に，内容仕様(content spedification)を記述する．

例 4.3.7　要素型宣言

```
<!ELEMENT memo (to,from,body)>
```

上の例における内容仕様は，この型の要素中には，to，from，body という要素が順番に現れなければならないことを示している．下記に要素型宣言の文法規則を示す．

仕様書抜粋 4.3.2　要素型宣言

```
<!ELEMENT'  S  Name  S  contentspec  S?  '>'
```

要素型の名前は，XML の名前(name)である．従って，要素型の名前として使える文字列には特定の制限がある．また，各要素型宣言で，異なる名前を指定する必要がある．属性や実体とは異なり，要素型は 1 回だけしか宣言できない．

4.3.4 要素型の内容仕様

各要素型には，特定の許される内容というものがある．例として，ある文書型定義では，章(chapter)の内容として title を中に含むことを許されるが，脚注(footnote)の内容として chapter を含むことを許されない．内容仕様には表 4.3.1 に示す４つの種類がある．

表 4.3.1　内容仕様の種類

内容仕様の種類	許される内容
空(EMPTY)内容	内容を持つことは許されない．一般に，属性を指定するために使われる．
任意(ANY)内容	任意の内容を持つことができる．
混合内容	文字データあるいは，指定された下位要素が混在する文字データを含めることができる．
要素内容	指定された下位要素だけを含めることができる．

(1) 空(EMPTY)内容

いかなる内容も持たない要素型が必要な場合に，EMPTY という内容仕様を指定する．例として，HTML の img のような画像の要素型は，どこか他の場所にある画像を含むことになるであろう．これは属性を使って行われるわけであり，下位要素や文字データの内容は必要とされない．

(2) 任意(ANY)内容

任意の要素や文字データを含められる要素型が欲しくなる場合に，内容仕様で ANY を指定する．

任意内容は希にしか使われない．通常は，文書型の構造を表現するために要素型宣言を導入する． ANY という内容仕様を持つ要素型は，完全に非構造である．こうした要素型では，文字データや下位要素をどのように組み合わせても構わない．任意内容の要素型が有用なものの例として，DTD を開発中の場合がある．既存の文書の DTD を作成する場合に，最初は全ての要素型を任意内容として宣言し，妥当性をチェックする．それから各要素の正確な内容仕様を１つずつ検討していくことができる．

(3) 混合内容

混合内容(mixed content)を持つ要素型は，文字データのみ，もしくは，子要素が混在した文字データを内容として持つことができる．段落(paragraph)などは，典型的な混合内容要素の例である． paragraph の中では，文字データの中に強調(emphasis)や，引用(quotation)などといった下位要素が混在するであろう．

例 4.3.8　文字データだけの混合内容

```
<!ELEMENT emph (#PCDATA)>
<!ELEMENT foreign-language (#PCDATA)>
```

“(”や“)”と“#PCDATA”の間には，好みに応じて空白を入れることができる．上の例では，下位要素を中に含むことのできない要素型を作っている．もし，下位要素が検出されれば，妥当性エラーが報告される．

例 4.3.9 文字データと要素を混在させる

```
<!ELEMENT paragraph (#PCDATA | emph)*>
<!ELEMENT abstract (#PCDATA | emph quot)*>
<!ELEMENT title (#PCDATA | foreign-language | emph)*>
```

アスタリスク記号(*)については後で説明する．また，｜の前後に空白を入れることもできる．

これらの宣言は，文字データと下位要素を混在する要素型を作る．|の後で列挙されている要素型は，許される下位要素である．下記は，例 4.3.8 と例 4.3.9 の宣言を組み合わせた場合の妥当な title の例である．

```
<title>this is a <foreign-language>tres gros</foreign-language>
        title for an <emph>XML</emph> book</title>
```

emph，foreign-language，文字データは，どのような順序で並べられても構わない．また，下位要素 emph 並びに foreign-language は，必要に応じて何度でも出現できる．

4.3.5 内容モデル

内容仕様の最後の種類は，「子」に関する仕様であり，このタイプの使用は，当該の型の要素が，その内容として子要素だけを含められることを示す．ある要素型を要素内容(element content)をもつものとして宣言するには，前述の混合内容やキーワードの代わりに，内容モデル(content model)を指定する．内容モデルとは，どんな下位要素型がどんな順序で出現できるかを宣言するパターンである．

単純な内容モデルであれば，下記に示すように下位要素を１つだけもつ場合もある．

```
<!ELEMENT WARNING (PARAGRAPH)>
```

このモデルは，WARNINR は PARAGRAPH を 1 つだけ含まなければならないことを表している．混合内容の場合と同様，括弧の前後には空白を入れても構わない．WARNING 中に複数の要素を含ませる場合，下記のようになる．

```
<!ELEMENT MEMO (FROM ,TO , SUBJECT , BODY)>
```

ここで，"FROM"と"TO"の間のカンマ(,)は，MEMO 要素中で TO の前に FROM がなければならないことを表している．これを列(sequence)という．列の２つの部分を区切るカンマ(,)の前後には空白を入れても構わない．

列ではなく，選択(choice)を指定したい場合は下記のようになる．

```
<!ELEMENT FIGURE (CODE | TABLE | FLOW-CHART | SCREEN-SHOT)>
```

|という記号は,いずれかの要素を選択できることを示す．|の前後に空白を入れることも可能である．

括弧を使って，列と選択を組み合わせることも可能である．選択もしくは列を括弧で囲むと，それは内容素子(content particle)となる．個々の共通識別子(GI)もまた内容素子である．内容モデル中で GI を指定できる箇所であればどこでも，任意の内容素子を使うことができる．

```
<!ELEMENT FIGURE
        (CAPTION, (CODE | TABLE | FROW-CHART | SCREEN-SHOT) )>
```

上の FIGURE の内容モデルでは，２つの内容素子の列から構成されている．第 1 の内容素子は単一の要素型名で，第 2 の内容素子は複数の要素型名の選択である．

XML では，出現指示子(occurrence indicator)を使うことによって，内容素子のオプションや繰り返しを指定することができる．

表 4.3.2　出現指示子

指示子	意味
?	オプション(0 回か 1 回)
*	オプションかつ繰り返し可能(0 回かそれ以上)
+	必須かつ繰り返し可能(1 回かそれ以上)

出現指示子は，GI，列，あるいは選択の後ろに，直接続けて指定する．前に空白を入れることはできない．

下記は，図の見出し(CAPTION)をオプションにした例である．

```
<!ELEMENT FIGURE
        ( CAPTION? , (CODE | TABLE | FROW-CHART | SCREEN-SHOT ) )>
```

複数の段落から構成される脚注(FOOTNOTE)を許すには，下記のようにする．

```
<!ELEMENT FOOTNOTE (P+)>
```

ここでは+を指定したため，脚注には少なくとも 1 つの段落が必要になる．

出現指示子は，列や選択と組み合わせて指定することもできる．

```
<!ELEMENT QUESTION-AND-ANSWER
        (INTRODUCTIO , QUESTION , ANSWER )+ , COPYRIGHT?)>
```

内容モデル中の全ての要素型をオプションにすることも可能である．

```
<!ELEMENT IMAGE ( CAPTION? )>
```

IMAGE という要素は時には空に，時には空でなくなる．?記号は CAPTION がオプションであることを示す．空となる場合は，属性によって外部の画像とリンクされていると考えられる．見出し(CAPTION)を指定したい場合のみ，内容を記述する．

文書インスタンス上では，空の IMAGE 要素は，IMAGE が常に空であると宣言された場合と同じ形式をとる．空として宣言されたのか，それともたまたま今の場合には空であるだけなのか，という区別は，文書インスタンスからは判断できない．

4.3.6 属性

文書内の要素に追加の情報を付加することが可能にするため, 属性(attribute)を用いる. 要素と属性の最大の相違点は，属性の中に要素を含めることは不可能であり，「下位属性」と言ったものもない，という点である．属性は，一般に小さく，単純で，構造化されていない「追加の」情報を示すのに用いられる.

要素と属性との別の重要な相違は，要素の各属性は 1 度だけ，任意の順序で指定できるという点にある．順序を覚える必要がないため，これは非常に便利である.

属性は副次的な重要性を持つデータを表し，それは多くの場合，情報に関する情報，つまりメタ情報である.

また，一般に属性の名前はオブジェクトの特性を表すのに対して，要素型の名前は通常，オブジェクトの部分を表す．例として，person という要素の場合，その下位要素は身体の各部分を表し，属性は体重や身長といった特性を表す.

(1) 属性リスト宣言

属性は，特定の要素型に対して宣言する．各要素型の属性は，属性リスト宣言(attribute-list declaration)によって宣言される.

```
<!ELEMENT  PERSON  (#RCDATA)>
<!ATTLIST  PERSON  EMALL  CDATA  #REQUIRED>
```

属性の宣言は，“ <!ATTLIST ”という文字列で始まり，空白に続いて要素型の共通識別子, 属性の名前，属性の型(type)，属性のデフォルトの値が続く．上記の例では，属性の名前は EMAIL で，PERSON 要素に対して有効である．この属性の値は文字データでなければならず，必須である.

仕様書抜粋 4.3.3　属性リスト宣言

```
AttlistDecl  : : = '<!ATTLIST'  S  Name  AttDef*  S?  '>'
AttDef  : : =  S  Name  S  AttType  S  DefaultDecl
```

単一の属性リスト宣言中は，複数の属性を宣言できる．また，単一の要素型に対して，複数の属性リスト宣言を記述することも可能である．これは，単一の属性リスト宣言中で全ての宣言をまとめて指定したのと同じことになる.

例 4.3.10　複数属性,同じ要素型に対する複数の属性リスト宣言

```
<!ATTLIST PERSON HONORIFIC CDATA #REQUIRED>
<!ATTLIST PERSON POSITION CDATA #REQUIRED
                 ORGANIZATION CDATA #REQUIRED>
```

また，同じ要素型の同じ属性に対して，複数の宣言を行うことも可能であるが，この場合，最初の宣言だけが有効になる.

２つの異なる要素型が，同じ名前の属性を衝突することなく持つこともできる．これら属性は同じ名前であるが，実際は異なる属性なのである．例として，SHIRT という要素が，SMALL，MEDIUM，LARGE という値をとりうる SIZE という属性を持つ一方で，同じ DTD 中の PANTS という要素も，インチ単位の大きさを表す SIZE という属性を持つことが可能である．

```
<!ATTLIST  SHIRT  SIZE (SMALL | MEDIUM | LARGE) #REQUIRED>
<!ATTLIST  PANTS  SIZE  NUMBER  #REQUIRED>
```

しかし，同一文書中で同じ名前の属性が，異なる意味や値を持つのは，混乱を招くもととなり，好ましくない．

(2) 属性のデフォルト値

属性はデフォルト値(default value)を持つことができる．

```
<!ATTLIST  SHIRT  SIZE (SMALL | MEDIUM | LARGE) MEDIUM>
<!ATTLIST  SHOES  SISE  NUMBER  "13">
```

属性リスト宣言で指定した制約条件に合致する値であれば，どんな値でも有効である．

特定のデフォルト値を強制せずに，属性値の指定を省略できるようにしたい場合には，暗示可能(impliable)属性を用いる．例として，NUMBER が宣言された SIZE という属性を持つ SHIRT という要素があるものとする．しかし，いかなるサイズにもフィットするシャツもある．こうしたシャツは，サイズというものを持たない．このような場合，文書の作成者はこの属性を指定せずに，プロセッサにはそれが「どんなサイズにもフィットする」シャツであると暗示(imply)する．

```
<!ATTLIST  SHIRT  SIZE  NUMBER  #IMPLIED>
```

"#IMPLIED"と指定することにより，適切と思われる任意の値を挿入する権利が，処理プログラムに与えられる．一般的には，暗示可能属性は単に無視されるだけである．またこの属性は，文書の作成者が，その値を気にする必要がない場合のみ，あるいは値の省略が実際には何を意味するかに関する明確な慣行が存在する場合のみ，値の省略をすべきである．DTD の設計者は，何らかの形で値の省略が何を意味するかを作成者に伝えるべきである．

必須ではない属性の値を指定しないのは，容易なことである．その属性を記述しなければよいのである．ただし，属性値に空の文字列を指定するということは，属性を記述しないこととは違う点に注意しなければならない．

```
<SHIRT>                <!--  これは上記の宣言に適合します．  -->
<SHIRT  SIZE=" ">      <!--  これは上記の宣言に適合しません．  -->
```

ある属性の値が重要な意味を持ち，デフォルト値に頼るのが危険な場合には，必須

(required)のデフォルトを指定することによって，設計者は作成者に属性値の指定を強要することができる．

```
<!ATTLIST  IMAGE  URL  CDATA  #REQUIRED>
```

この例では，DTD の設計者は全ての IMAGE 要素について，URL という属性を必須にしている．URL がなければ画像ファイルのある位置が分からないため，IMAGE という要素は意味のないものになってしまう．

全く値を変更できない属性値を供給することが有用な場合がある．これは希なことであるが，このような場合，固定(fixed)属性が利用される．

```
<!ATTLIST  H1  TITLE-ELEMENT  CDATA  #FIXED  "TITLE-ELEMENT">
<!ATTLIST  HEAD  TITLE-ELEMENT  CDATA  #FIXED  "TITLE-ELEMENT">
<!ATTLIST  TITLE  TITLE-ELEMENT  CDATA  #FIXED  "TITLE-ELEMENT">
```

(3) 属性の型

属性の重要な特徴は，ある種の語彙的ならびに意味論的な制約を強要する型(type)を持っているという点にある．語彙的な制約とは，「この属性では数値しか指定できない」といった類のもので，意味論的な制約とは，「この属性値は，宣言済みの実体の名前でなければならない」といったものである．

ただし，属性の値は，必ずしも引用符で囲まれた文字列と正確に一致するとは限らない．最初に属性値の正規化(attribute-value normalization)と呼ばれる処理を経る．属性の型は正規化された値(normalized value)に対して適用されるため，まず，正規化について説明する．

(4) 属性値の正規化

XML プロセッサは属性値を正規化することによって，文書の作成者の仕事を軽減する．もし正規化がなければ，属性値の中に空白を入れる場合にはそれを注意する必要がある．

```
<GRAPHIC ALTERNATE-TEXT="This is a picture of a penguin
                                    doing the ritual  mating  dance">
```

上記の様な改行の場合，単一の行ではエディタのウィンドウサイズに収まらないと言う理由だけで改行を行っていることもある．この種のものは XML プロセッサによって正規化される．

XML パーサが正規化の準備として最初に行うことは，属性値を囲む引用符を除去することである．次に，文字参照が対応する文字に置き換えられる．続いて，一般実体参照が置き換えられる．これは，まだ正規化されていない属性値はテキスト，つまりマーク付けとデータとの混合体であるため，データだけを残す必要があるためである．

実体参照を展開した結果，さらに他の実体参照が中に含まれている場合には，その実体参照も展開される．

属性値中の改行文字は，スペースによって置き換えられる．もしその属性値がトークンとなる種類のもの(時節参照)であると分かっている場合なら，パーサはさらに，先行と後続のスペースを除去する．したがって，"　token　"は"token"となる．さらにトークン間の複数のスペースは，単一のスペースにまとめられる．正規化されていない属性値テキストと正規化された属性値データとの区別は，見誤る可能性がある．属性の型について議論している場合には，それは，正規化されたデータに対して適用されるものであること注意しなければならない．

(5) CDATA 型ならびに名前トークンの属性

最も単純な属性の型は CDATA 型である．CDATA とは「文字データ」(character data)を表す．

例 4.3.11　CDATA 型の属性

```
<!DOCTYPE　ARTICLE[
<!ELEMENT　ARTICLE>
<!ATTLIST　ARTICLE　DATE　CDATA　#REQUIRED>
・・・・・
]>
<ARTICLE　DATE="January 15,1999">
・・・・・
</ARTICLE>
```

CDATA 型の属性値は，任意の文字の列である．この型の属性値では，基本的にどのような文字も正当である．

名前トークン(NMTOKEN)型の属性は，いくらか CDATA 型に似ている．最大の違いは，名前トークンで許される文字に限定されるという点である．

例 4.3.12　NMTOKEN 型の属性

```
<!DOCTYPE　PARTS-
・・・・・
<!ATTLIST　PART　DATE　NMTOKEN　#REQUIRED>
・・・・・]>
<PARTS-LIST>
・・・・・
<PART　DATE="1998-05-04">・・・</PART>
</PARTS-LIST>
```

空の文字列は有効な名前トークンではないが，CDATA 型の属性値としては有効である．

名前トークンは,特殊な文字を必要とする数値の属性を指定するのに使える．

複数の名前トークンを含めることが適切な場合には，NMTOKENS 型の属性を使うことができる．

例 4.3.13 NMTOKENS 型の属性

```
<!DOCTYPE  DATABASE
・・・・・
<!ELEMENT  TABLE  EMRTY>
<!ATTLIST  TABLE  NAME  NMTOKEN  #REQUIRED  FIELDS  NMTOKENS
#REQUIRED>
・・・・・
]>
<DATABASE>
・・・・・
<TABLE        NAME="SECURITY"        FIELDS="USERID        PASSWEORD
DEPARTMENT">
・・・・・
</DATABASE>
```

前節でも述べたように，CDATA 型と NMTOKEN 型のもう一つの違いは，正規化にある．

(6) 列挙型ならびに記法型の属性

DTD を設計する際に，一連の候補のうち，1 つの候補を取りうる属性を作りたくなる場合，列挙型(enumerated attribute type)属性を使う．ある意味でこれらは，選択メニューを提供するものといえる．

```
<!ATTLIST CHOICE (OPTION1|OPTION2|OPTION3) #REQUIRED>
```

選択候補はいくつでも指定でき，個々の候補値は XML の名前トークンであるため，名前トークンの構文要件を満たす必要がある．

これに関連して，記述型と呼ばれる属性もある．この属性により文書の作成者は，要素の内容が宣言済みの記法に従っていることを示すことができる．下記は，日付の様々な表現方法を可能にしている例である．

```
<!ATTLIST  DATE  DATETYPE  NOTATION
          (EUROPEAN-DATE | US-DATE | ISO-DATE) #REQUIRED)>
```

妥当な文書において，候補にあげられる記法は，記法宣言によって宣言されている必要がある．

(7) ID 型ならびにIDREF 型の属性

ある要素型の特定の要素に対して，名前を付けたい場合がある．例えば，ある要素から別の要素への単純な相互参照を作成する場合，特定のセクションや図に名前を付けておく．そして後でそれを参照する場合に，名前を使う．参照先の要素には，ID 型の属性によってラベルを付けておく．一方参照元の要素は，IDREF 型の属性によって参照する．

例 4.3.14　相互参照のためのID と IDREF

```
<!DOCTYPE BOOK
・・・・・
<!ELEMENT SECTION (TITLE,P*)>
<!ATTLIST SECTION MY-ID ID #IMPLIED>
<!ELEMENT CROSS-REFERENCE EMPTY>
<!ATTLIST CROSS-REFERENCE TARGET IDREF #REQUIRED>
・・・・・
]>
<BOOK>
・・・・・
<SECTION MY-ID="Why.XML.Rocks"><TITLE>Features of XML</TITLE>
・・・・・
</SECTION>
・・・・・
If you want to recall why XML is so great, please see the section
titled <CROSS-REFERENCE TARGET="Why.XML.Rocks"/>.
・・・・・
</BOOK>
```

ID は XML における名前であるため，名前の構文条件を満たす必要がある．また，各要素は複数の ID を持つことができない．そして，１つの XML 文書中で指定された全ての ID は，一意でなければならない．同一の ID 型の属性値を含む文書は，妥当な文書でなければならない．したがって，chapter などは，複数の箇所で使われる可能性があるため，ID としては適した名前ではない．

IDREF 型の属性は，文書中のある要素を参照している必要がある．ある１つの要素に対しては，必要に応じて複数の IDREF から参照することができる．また，IDREFS 型として宣言することにより，複数の IDREF を持ちうる属性を宣言することが可能になる．

```
<!ATTLIST  RELATED-CHAPTERS  TARGETS  IDREFS  #REQUIRED>
```

ここで TARGETS という属性は，その値に１つ以上の IDREF を含めることができる．ただし，２つ以上あるいは３つ以上の IDREF を持つなどといった指定はできない．要素型

の宣言においては，内容モデルを使うことによって，この種の指定は可能であったが，属性に関しては, このようなものはなく, 下記の例に示すように, 1 つ以上の CHAPTER-REF 要素を内容とする RELATED-CHAPTERS という要素を宣言し，各 CHAPTER-REF では単一の IDREF 型の属性を指定する.

例 4.3.15 IDREF 型の属性

```
<!DOCTYPE
・・・・・
<!ELEMENT RELATED-CHAPTERS (CHAPTER-REF+)>
<!ELEMENT CHAPTER-REF EMPTY>
<!ATTLIST CHAPTER-REF TARGET IDREF #REQUIRED>
・・・・・
]>
<BOOK>
・・・・・
<RELATED-CHAPTERS>
<CHAPTER-REF TARGET="introduction.to.xml"/>
<CHAPTER-REF TARGET="xml.rocks"/>
</RELATED-CHAPTERS>
・・・・・
</BOOK>
```

(8) ENTITY 型の属性

外部解析対象外実体は，XML の規則に従って解析処理すべきではない Web 上のオブジェクトを参照するための方法である．ENTITY 型として宣言された属性を利用することにより，解析対象外実体を属性で参照することが可能になる．一般にこれは，外部のオブジェクトをリンク，参照あるいは包含する目的で行われる.

例 4.3.16 ENTITY 型の属性

```
<!DOCTYPE
<!ATTLIST BOOK-REF TARGET ENTITY #REQUIRED> ・・・
<!ENTITY anoter-book SYSTEM
                    "http://village.infoweb.ne.jp/~fwgh9848/TheOtherBook.html">
・・・]>
<BOOK>
・・・・・
<BOOK-REF TARGET="another-book"> ... </BOOK>
```

(9) 属性の型のまとめ

列挙式(enumerted)の属性には，列挙(enumeration)型の属性と記法(NOTATION)型の属性の２種類ある．

トークン型(tokenized)の属性としては，各属性値が単一のトークンを表すもの(ID，IDREF，ENTITY，NMTOKEN)と，トークンのリストを表すもの(IDREFS，ENTITES，NMTOKENS)の合計７種類ある．

最後に CDATA という文字列型がある．これは制約が最も少なく，XML の特殊文字が適切に入力されている限りは，どのような文字の組み合わせも保持できる．

表 4.3.3 属性の型

型	語彙的制約	意味論的制約
CDATA	なし	なし
列挙 (enumeration)	名前トークン	宣言の候補一覧中にある
NOTATION	なし	宣言の候補一覧中にあり，記法名が宣言されている
ID	名前	文書内で一意
IDREF	名前	ある要素のID と一致する
IDREFS	名前群	それぞれがある要素のID と一致する
ENTITY	名前	宣言済みの実体名と一致する
ENTITIES	名前群	それぞれが宣言済みの実体名と一致する
NMTOKEN	名前トークン	なし
NMTOKENS	名前トークン群	なし

4.3.7 記法宣言

記法(notation)は，別のデータコンテンツを示すために，XML 文書の様々な部分で出てくる．データコンテンツの記法とは，オブジェクトのクラスのビットやバイトを，どのように解釈すべきかについて定義したものである．

データリソースと結びつけるための XML の機能は，実体宣言である．これらは 解析対象外実体 (unparsed entities) と言われる．解析対象外実体の宣言中では，外部の記法定義への一種のポインタを提供する宣言済みの記法名を指定する．外部の記法定義は，その記法についてのドキュメント，正式な仕様書，あるいはその記法で表現されたオブジェクトを処理できるヘルパアプリケーションを表す公開識別子，もしくはシステム識別子である．

例 4.3.17 解析対象外実体用の記法宣言

```
<!NOTATION HTML SYSTEM "http://www.w3.org/Markup">
<!NOTATION GIF SYSTEM "gifmagic.exe">
```

記法は，記法型の属性においても出てくる．この属性の型は，XML 要素のデータ内容の記法を明確にしたい場合に利用する．例として，ISO もしくは EU の日付形式を採用した日付要素を作りたい場合，それぞれの形式に対応する記法を宣言する．

例 4.3.18　要素内容の記法を宣言

```
<!NOTATION ISODATE PUBLIC "適切な識別子をここに指定">
<!NOTATION EUDATE PUBLIC "適切な識別子をここに指定">
<!ELEMENT TODAY (#PCDATA)>
<!ATTLIST TODAY DATE-FORMAT NOTATION (ISODATE|EUDATE)
#REQUIRED>
```

最後に，処理命令(PI)のターゲットに XML の名前を付与するものにも記法が使える．これは厳密な XML の必須要件ではないが，よい習慣である．なぜなら，これにより当該の処理命令のために一種の資料が提供され，さらに処理命令ターゲットを起動するのにアプリケーションを利用することも可能であるからである．

DTD はパフォーマンスの改善，データの長寿と幅広い再利用，そして処理の信頼性に関係する．DTD をうまく使うことにより，DTD は時間と費用を節約してくれる．

4.4 XSL

XSL (Extensible Style Language)は，標準化された方法で XML をフォーマット化するために W3C で開発されている仕様である．XSL の設計方針を下記に挙げる．

- XSL はインターネット上で簡単に利用できること．
- XSL スタイルシートは人間にとって読みやすく，適度に明確であること．
- XSL スタイルシートは作成が容易であること．

また，現在，Microsoft 社の Internet Explorer 5 (IE5)が実装している XSL は W3C が標準化している XSL とは多少異なる．そこで，ここでは現段階で実用的な，IE 5 で実装可能な MS-XSL について記述する．

4.4.1 概説

XSL は XML 文書にスタイルを適用するために使われる．XML 文書は通常，その抽象構造のみに従ってマーク付けされており，スタイルの適用やその他の特殊な処理のためだけに付けられたマーク付けは理論的には存在しない．従って XSL は，コンピュータの処理のためにコード化されたデータと，読みやすくフォーマット化された表現との間の架け橋となる．

XSL 言語は，要素型と属性を定義し，それらの出現を特定の箇所に制約し，意味を与える．XSL は正式な DTD の形で定義されているわけではないが，1 つの文書型として考えることができる．

例えば，<p>要素や，<title>要素とそのアクションを指定することにより，「段落では 12 ポイントのフォントを使う」，「タイトルでは 20 ポイントのボールド体を使う」などといった宣言をすることが出来る．

4.4.2 XSL スタイルシートの参照

現時点では，XML 文書からスタイルシートを参照するための案が１つ提案されている．これは標準にされていないが，簡潔かつシンプルで，必要な機能を果たす．したがって，いずれは事実上の標準になると思われる．この提案に関連するテキストを下記に示す．

仕様書抜粋 4.4.1　処理命令 xml:stylesheet

処理命令 xml:stylesheet は，XML 文書の前書き中のどこに書いても良い．この処理命令では，href(必修)，type(必修)，title(オプション)，media(オプション)，charset(オプション)という疑似属性を指定できる．

ここで最も重要な疑似属性は，スタイルシートの URI を指定する「href」と，当該のスタイルシートが DSSSL(Document Style Semantics and Specification Language)でも CSS(Cascading Style Sheets)でもその他のスタイルシート言語でもなく，XSL であることを示す「type」である．「title」も指定できるが，ブラウザはこれを使ってスタイルシートの選択肢を表示できる．「media」は，どんな媒体向けのものかを指定するためのものである．例 4.4.1 にスタイルシート処理命令(PI)の例を示す．

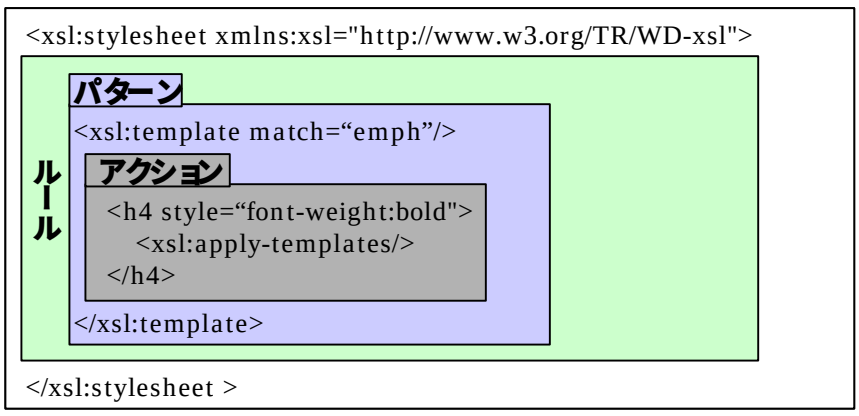
例 4.4.1　スタイルシート処理命令

```
<?xml:stylesheet type="text/xsl" href="style.xsl" ?>
```

4.4.3 パターンとアクション

XML 要素をフローオブジェクトに変換するための指定は，ルールと呼ばれ，変換元の XML 要素と変換先のフローオブジェクトを指定する．MS-XSL では，次に示す例のように，stylesheet タグ内で templat タグを用いて，ルールを記述していく．この構築規則では，XML 要素を指定する部分を **パターン** (patern)と呼び，変換先の **フローオブジェクト** (flow object)の生成に関する指定を **アクション** (action)と呼ぶ．

例 4.4.1　単純なルール



例 4.1.1 のルールは，emph 要素にマッチするパターンをボールド体にすることを表している．

4.4.4 フローオブジェクト

最終的なレイアウトを実際に記述するためにはコマンドが必要となり，XSL では，実際に表示される文書の構成要素に対応する一連のフローオブジェクトを提供している．

手書きの原稿をワードプロセッサに入力する場合を考えてみると，段落，行頭記号付きリスト，ハイパーテキストリンクなど，ワードプロセッサが提供している構成要素を使う必要がある．XSL の用語では，これらの構成要素を「フローオブジェクト」と言う．「フローオブジェクト」と呼ばれる理由は，テキストが次から次へと流し込まれ(flow)，文字，段落，クリック可能リンク，ページといったものを表す個別のオブジェクトになるためである．簡単な XSL スタイルシートであれば，段落，外部画像，罫線，表などといったフローオブジェクトから構成される．

論理的に，これらフローオブジェクトは木構造を成す．ページがルートになり，表や列などの「コンテナ」オブジェクトは枝，文字や画像などの「原子」オブジェクトは葉になる．葉のオブジェクトが「原子」と言われるのは，原子が他の原子から構成されないように，葉のオブジェクトは他のいかなるオブジェクトからも構成されないためである．また，フローオブジェクトの木構造は「フローオブジェクトツリー」と呼ばれる．

すべてのフローオブジェクトは，それぞれ特徴を持っている．あるフローオブジェクトが提示する特徴は，そのクラス(class)によって決まる．例として，Web ページにはスクロールバーが，クリック可能リンクにはリンク先が，フォントにはフォントサイズが，画像には高さと幅がある．

XSL の現在のバージョンでは，数多くの Web 設計者の HTML に関する知識を利用する特別なフローオブジェクトも用意されている．これらは「HTML フローオブジェクト」と呼ばれており，HTML の DTD 中に現れる要素型に対応している．これらを使って文書をフォーマット処理すれば，その文書は直接 HTML で作成されたかのように見える．こうした処理は，XML によるマーク付けへの変換と考えることが出来る．

目的は，実際の HTML ファイルを作ることではなく，文書の作成者がすでに知っている用語で文書のフォーマット処理を記述することにある．理論的には，HTML 文書は概念的に存在するだけである.現実問題としては，まだ XSL をサポートしていないブラウザのために，XSL プロセッサが HTML 文書を出力する．それを，あたかも直接 HTML で作成したかのように，Web 上で利用できるのである．

4.4.5 パターン

パターンは，スタイルのルールを適用すべき要素を，XSL プロセッサが選択するためのものである．すべてのパターンは，マッチさせるべき文書中の要素を特定する xsl:template

エレメントをもつ．match 属性によって，特定の要素型の要素だけをマッチングの対象にできる．したがって，下記のルールは emph 型の要素だけをボールド体にする．

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match= "emph"/>
    <h4 style=”font-weight:bold”>
      <xsl:apply-templates/>
    </h4>
  </xsl:template>
</xsl:stylesheet >
```

一方，下記のルールは，任意の要素をボールド体にする．

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template mach=”*”>
    <h4 style=”font-weight:bold”>
      <xsl:apply-templates/>
    </h4>
  </xsl:template>
</xsl:stylesheet >
```

あるルールが任意の要素にマッチする場合，そのルールをデフォルトルールという．他のどのルールともマッチしない場合に呼び出されるルールだからである．ほとんどの場合，デフォルトルールは用意しておくべきである．デフォルトルールを用意しておかないと，ある特定のルールを指定し忘れた際に，その型の要素は出力上に表示されなくなる．

また，特定の属性値を持つ要素だけをマッチさせることも可能である．

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match= “security”>
      <xsl:value-of select=" top-secret "/>
      <h4 style=”font-weight:bold”>
        <xsl:apply-templates/>
      </h4>
  </xsl:template>
</xsl:stylesheet >
```

xsl:value-of 要素では，select という属性を指定できる．その属性値は，特定の値を条件とすることも，任意の値を条件にすることも可能である．上の例では top-secret という値を条件としている．また，select の属性値を@list にすることで任意の値を条件にすることができる．

要素の前後関係に基づいたマッチングも可能である．例として，warning 要素中すべてのｐ要素の色を変えたい場合には，下記のルールを指定する．また，基本的に，xsl:template 要素は，文書中の要素とマッチされるパターンを定義する．

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match= “warning”>
    <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match= “p”>
    <h4 style=”color:red”>
      <xsl:apply-templates/>
    </h4>
  </xsl:template>
</xsl:stylesheet >
```

4.4.6 アクション

XSL プロセッサは，パターンのマッチングによってルールを選択すると，そのルールのアクション部を調べる．アクションは，出力の木構造中にどういうオブジェクトを作成すべきかを表す．アクションでは，フローオブジェクトを直接生成し，定型的な文言などのテラルのテキストを追加し，ソース要素の内容をフローオブジェクトの中に含めるなどを，XSL プロセッサに対して指示することができる．例えば，入力文書中の<line/>という要素を，<hr/>というフローオブジェクトに変換する．この場合は下記のようになる．

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match= “line”>
    <hr/>
  </xsl:template>
</xsl:stylesheet >
```

ここでは line は空要素であり，内容を持たないと仮定しているため，簡単になっている．しかし，ほとんどの要素は内容を持っている．内容は，文字データと下位要素とから成る．この文字列や下位要素のことを，子(child)と言う．子は<xsl:apply-templates/>という要素を使って処理できる．また，下記のようにリテラルのテキストを挿入することも可能である．

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match= “warning”>
  <hr/>
  <h4 style=”font-weight:bold;color:red”>
      警告：
  </h4>
  <xsl:apply-templates/>
  <hr/>
  </xsl:template>
</xsl:stylesheet >
```

また，フローオブジェクトの要素に属性を追加することもできる．下記の例では，XML 文書中の link という要素を指定し，a という要素で属性が href というフローオブジェクトに変換している．属性値は http://www.niit.ac.jp/ になり，新潟工科大学にリンクが張られる．

```
<xsl:stylesheet result-ns="html">
  <xsl:template match= “link”>
    <a>
      <xsl:attribute name="href">
        http://www.niit.ac.jp/
      </xsl:attribute>
        新潟工科大学
    </a>
  </xsl:template>
</xsl:stylesheet >
```

4.4.7 フローオブジェクトとその特性

フローオブジェクトには 2 つの主要な種類がある. HTML をベースにしたものと, DSSSL をベースにしたものの 2 つである．これら 2 つは同時に組み合わせて使うようには意図されていない．HTML に関する知識があれば，HTML のフローオブジェクトの方が使いやすいであろう．これらは Web 上での HTML 要素と全く同じように表示され，動作する．XSL での利用可能な HTML の要素型を次に挙げる．

仕様書抜粋 4.4.2　XSL におけるフローオブジェクト

```
SCRIPT
PRE
HTML
    TITLE
    META
    BASE
BODY
DIV
BR
SPAN
FORM
    INPUT
    SELECT
    TEXTAREA
A
 HR
TABLE
    CAPTION
    COL
    COLGROUP
    THEAD
    TBODY
    TFOOT
    TR
    TD
IMG
    MAP
    AREA
OBJECT
    PARAM
    FRAMESET
```

これらの要素型は，HTML の仕様で規定されている属性を付けて作成することが，可能である．また，CSS(Cascading Style Sheet)のプロパティに対応する属性も指定できる．

DSSSL のフローオブジェクトには，次に挙げるものがある．

仕様書抜粋 4.4.3　XSL における DSSSL フローオブジェクト

scroll ―― **オンライン表示用**
paragraph,paragraph-break ―― **段落用**
character ―― **テキスト用**
line-field ―― **リスト用**
external-graphic ―― **画像の取り込み用**
horizontal-rule,vertical-rule ―― **罫線用**
score ―― **下線ならびに抹消線用**
embedded-text ―― **双方向テキスト用**
box ―― **境界用**
表関連のフローオブジェクト

table	table-row
table-part	table-cell
table-column	table-border

sequence ―― **特性の継承用**
display-group
―― **フローオブジェクトの位置決め用**
simple-page-sequence
―― **単純なページレイアウト用**
link ―― **ハイパーテキストリンク用**

4.4.8 XSL と JavaScript

XSL では，複雑な処理を行うためのプログラミング言語を，JavaScript の標準化バージョンである ECMAScript の形で埋め込む. XSL で使われる JavaScript は, 本質的には Web ページに対話性をもたらすために使われているものと同じであるが，文書に対してスタイルを適用するために設計された新しい機能を備えている.

Web ページにおいて HTML だけでは表現が不十分な場合に JavaScript を使うように, スタイルシートにおいても単純な宣言だけでは扱えない問題を解決するために, JavaScript を使うことができる. XSL 文書中で JavaScript を使うと, その基盤となる XML システムの高度な機能が利用できるようになり，これによりスタイルの適用が容易になる.

第5章 XSCLの開発

5.1 XSCL の仕様

XSCL も HTML での表現と同様,「1 ページ＝1 部品仕様記述」の形式を取る．SCB によっては数百ものコンポーネントを取り扱っている．１つのコンポーネントにつき複数のファイルで記述した場合，ファイル数が増加し，管理が困難となる．このため上記の形式とした.

XSCL は，SCL の文法を XML で表現する．従って XSCL の構造は，基本的に SCL の構造と同様である．XSCL は，下記の５つの部分から成る.

① Component Part:XSCL 本体．Component 名，SCL Version 情報
② Commerce Part:商取引情報
③ Environment Part:環境条件に関する情報
④ Specification Part:インタフェース仕様
⑤ Usage Part:利用方法や試行情報

また XSCL は，SCL の文法に以下の変更が加えられている.

(1) Environment Part の変更点

従来の SCL では，プロセッサやメモリサイズ，ファイルサイズが定義されていなかった．このため，この項目を追加した.

(2) Specification Part の変更点

Specification Part では 2 カ所変更した．第 1 に，module 要素を追加し，interface 要素を module 要素の下位要素とした．これは，従来の SCL ではインタフェースの記述に関し，Module に対応する要素がないため，CORBA-IDL を正しく表現できなかったためである.

第 2 に，Scenario-pattern 要素の要素名を Behavir-pattern に変更した．これは，単語の意味が Scenario より Behavir の方が適当であると判断したためである.

次に，XSCL の構成を図 5.1.1 から 5.1.5 に示す.

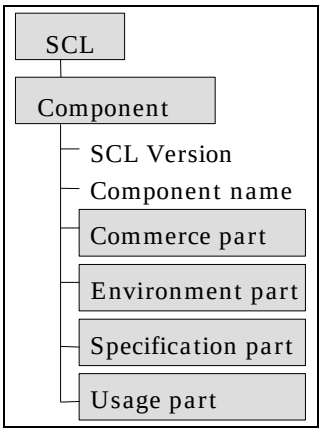


図 5.1.1 Component Part

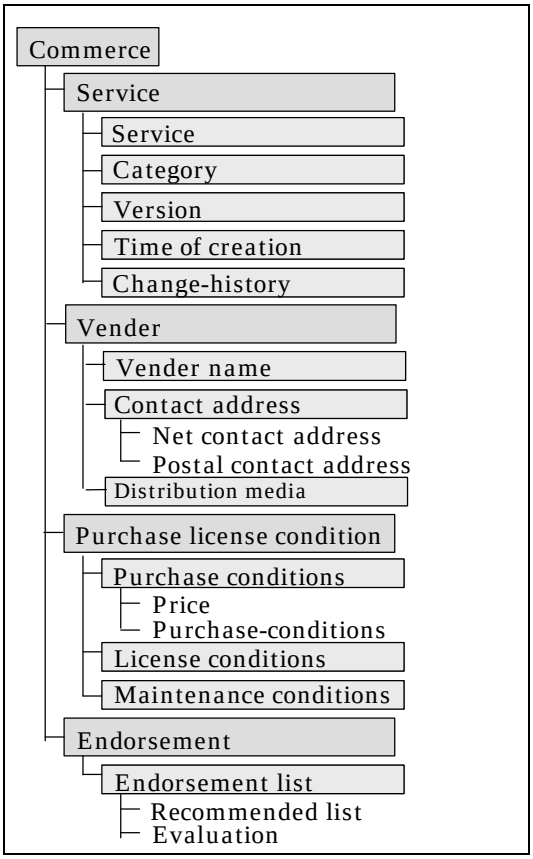


図 5.1.2 Commerce Part

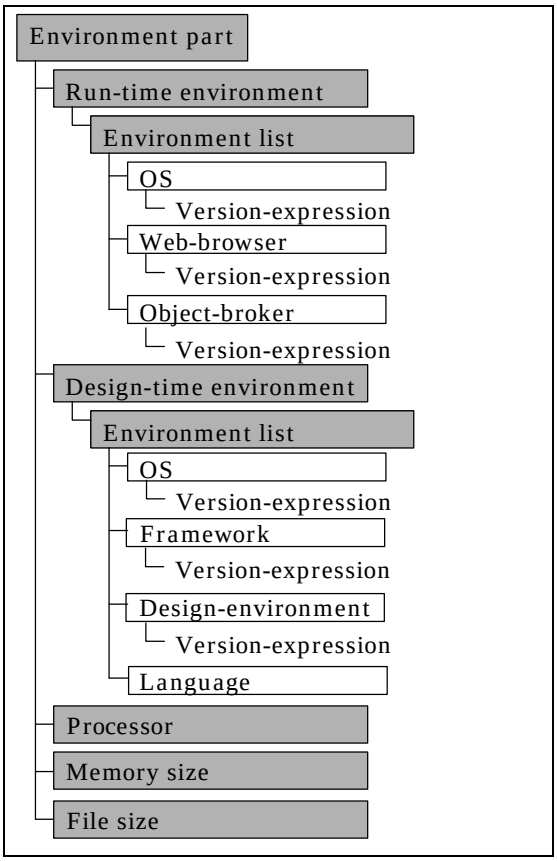


図 5.1.3 Environment Part

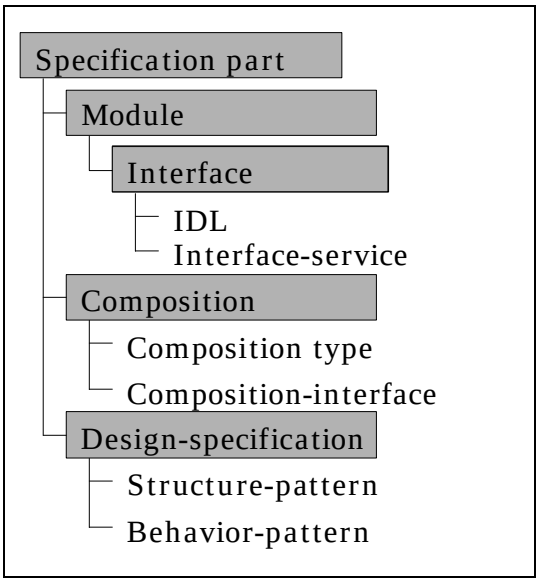


図 5.1.4 Specification Part

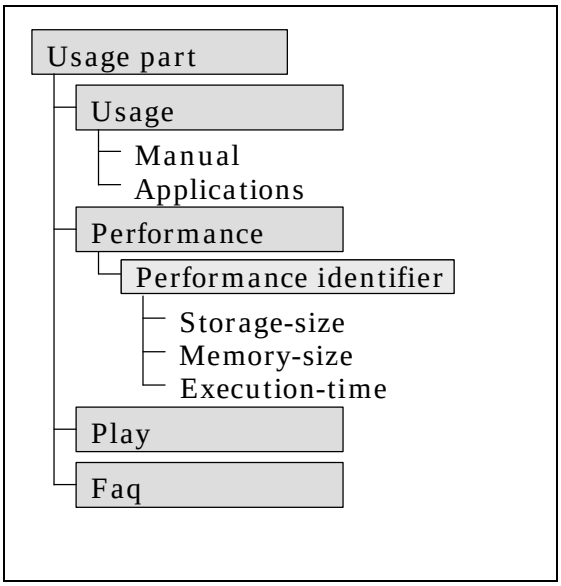


図 5.1.5 Usage Part

5.1.1 DTD

XSCL の DTD は，下記の５つの部分から成る．

① Component Part:SCL の DTD 本体
② Commerce Part:商取引情報
③ Environment Part:環境条件に関する情報
④ Specification Part:インタフェース仕様
⑤ Usage Part:利用方法や試行情報

XSCL の DTD は，XML 文書内に直接記述せず，外部サブセット(externl subset)に置いた．これは，他の XML 文書でも共有できるようにするためである．

例 5.1.1　外部サブセットへの参照(XSCL 文書)

```
<?xml version="1.0" encoding="shift_jis"?>
<!DOCTYPE SCL SYSTEM "scl.dtd">
<SCL>
……
</SCL>
```

例 5.1.1 の太字で記述された部分が，外部サブセットへの参照を示している．XML において，DTD を外部サブセットに置くのは一般的な方法である．

SCL の文法を DTD で書き直した場合，ソース記述量が増える．これは，開発のしやすさや，ソースの解読，変更などが困難となる．これを避けるために各 Part ごとに分割した．DTD 自体も外部パラメータ実体で５つの部分に分けた．外部パラメータで分割した例を例 5.1.2 に示す．この結果ファイル数は増えるが上記の問題が回避でき，開発や修正が容易になった．

例 5.1.2　外部パラメータ実体

```
<!ENTITY % commerce SYSTEM "commerce.dtd">
        %commerce;
```

第４章で述べたように，ENTITY で実体を宣言し，%は一般実体とパラメータ実体の宣言を区別している．%の次の文字列” commerce”が実体の名前，その後はシステム識別子と URI(または相対 URI)が続く．この実体は，次の行で参照されている．パラメータ実体の参照は%で始まり；で終わる．

作成した DTD において，外部パラメータ実体はすべて同一フォルダ内に置かなければならない．これは，外部パラメータ実体の場所指定を相対 URI で記述したためである．それぞれを別のフォルダあるいは別のマシンに置く場合，URI の修正が必要である．

5.1.2 属性

SCL ではいくつかの項目において，選択肢の中からどれか１つの値を取る List を定めている．例えば， distribution media(提供媒体)の項目では，CD-ROM，FDD，network の中からいずれかのメディアを選択する．これを他の項目と同様に，要素の内容として記述する場合の DTD は下記のようになる．

```
<!ELEMENT distribution_media (#PCDATA)>
```

この場合 distribution_media 要素の内容は任意となる．ここで問題なのは，CD-ROM，FDD，network の中からいずれかの値を選択するように指定していないという点である．したがって，これ以外の文字列が記述されても XML では誤りとならない．例 5.1.3 のように記述された場合でも，XML ソースとしては妥当な文書である．しかし SCL の文法的には誤りとなる．

例 5.1.3 要素の内容として指定した場合の問題点の例

```
<distribution_media>Internet</distribution_media>
```

この問題は，属性を使うことによって解決した．属性は，"ある種の語彙的並びに意味論的な制約を強要する型(type)を持っている" という特徴があり，その型の一つである列挙型を使うことによってそれが可能となる．属性を使った場合の DTD を例 5.1.4 に示す．

例 5.1.4 属性値の指定

```
<!ELEMENT distribution_media EMPTY>
<!ATTLIST distribution_media list (network | CD-ROM | FDD) #REQUIRED>
```

まず１行目で distribution_media を空要素として定義し，2 行目でその属性について定義している．属性名は"list"であり，CD-ROM，FDD，network のいずれかの値を取るとしている．さらに#REQUIRED は，この属性は必須であると定めている．この DTD に基づいた XML の記述を下記に示す．

```
<distribution_media list="network" />
```

この様に属性の列挙型を使うことにより，SCL の文法に基づいた XML の作成が可能となった．

XSCL の DTD では，List の他にリンクについても属性を利用した．SCL では，ベンダの詳細や利用マニュアル，faq などは直接ソースには載せず，その情報を提供しているページの URL を記述する．しかし，要素の内容として URL を記述した場合，そのページのタイトルを記述できない．このため XSCL では，URL を属性値として記述することとした．その結果，タイトルをその要素の内容として記述することにより可能となる．例 5.1.5 に URL の記述例を示す．例 5.1.5 の場合 FAQ がタイトルとして記述されている．

例 5.1.5 URL の記述

```
DTD:
<!ELEMENT faq (#PCDATA)>
<!ATTLIST faq url CDATA #REQUIRED>
XML:
<faq url="http://ies045.iee.niit.ac.jp/~team-a/scl/AboutThisDemo.html"> FAQ </faq>
```

DTD のソースは付録 9.4 を参照願いたい.

5.2 XSCL のパーサによる解析

XSCL の開発において，DTD および XML 文書が整形式(well-formed)で記述されているか，あるいは妥当(valid)な文書であるかをチェックするためにパーサを用いて解析した.

新たに XSCL 文書を作成した場合，パーサに解析させる必要がある．これは，XSCL 文書が整形式であるかどうかを確かめるだけでなく，要素の記述漏れを防ぐためである．つまり，XSCL の DTD に従った妥当な文書であれば，要素の内容を空にしない限り情報の欠落はない．このことからパーサによる解析は重要である.

今回使用したパーサは，IBM が提供している XML Viewer である.

このツールは本来 XML の解析が主な目的ではなく，XML のツリー構造やソースを GUI で表示するためのツールである．しかし，このツールには同じく IBM が提供している XML Parser for Java が組み込まれている．この XML Viewer は DTD と XML 文書を同時に解析し，さらにツリー構造や DTD および XML 文書のソース，属性値を表示できるため，使い勝手がよい．図 5.2.1 と図 5.2.2 に XML Viewer の使用例を示す.

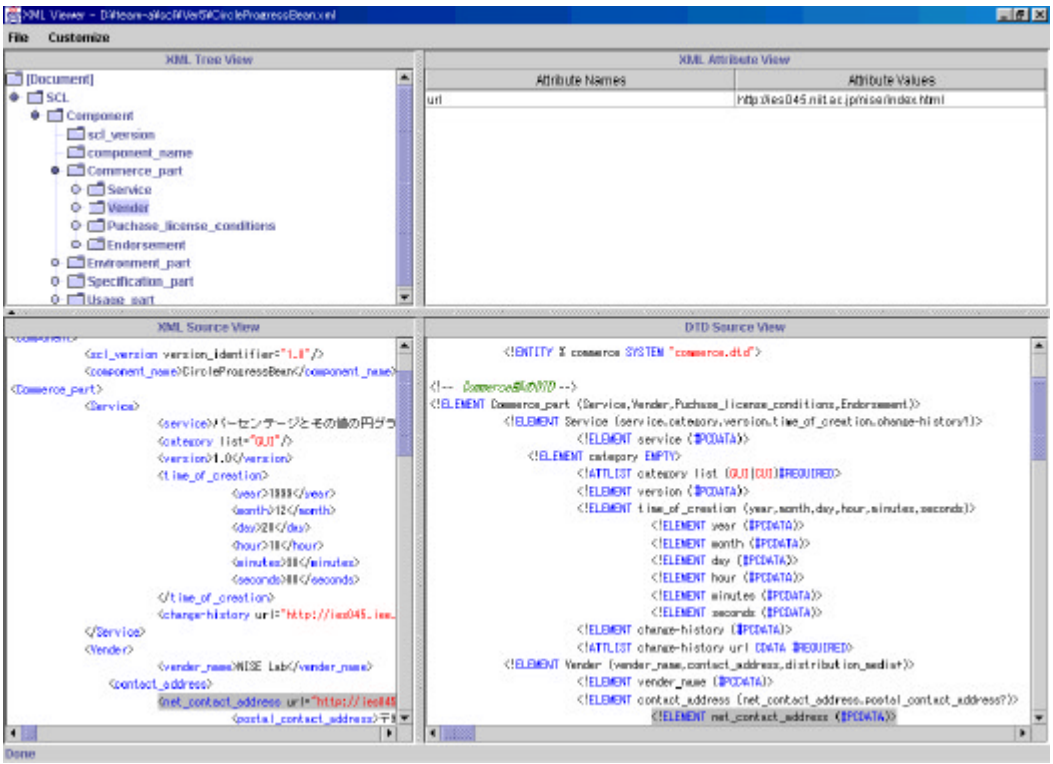


図 5.2.1　XML Viewer の使用例

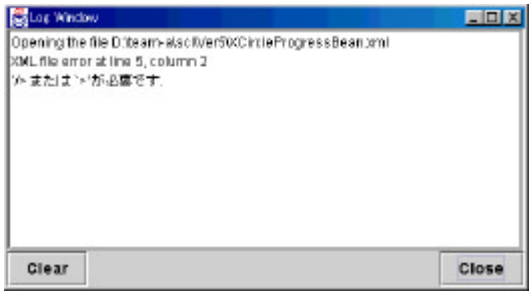


図 5.2.2 Log window

図 5.2.1 の左上にはツリー構造，右上には属性と属性値，左下には XML ソース，右下には DTD のソースがそれぞれ表示されている．

図 5.2.2 の Log Window は，DTD もしくは XML 文書が整形式ではない，また XML 文書が妥当な文書ではない場合，どこが誤りかを指摘してくれる．

次に XSCL 文書作成の流れを図 5.2.3 に示す．

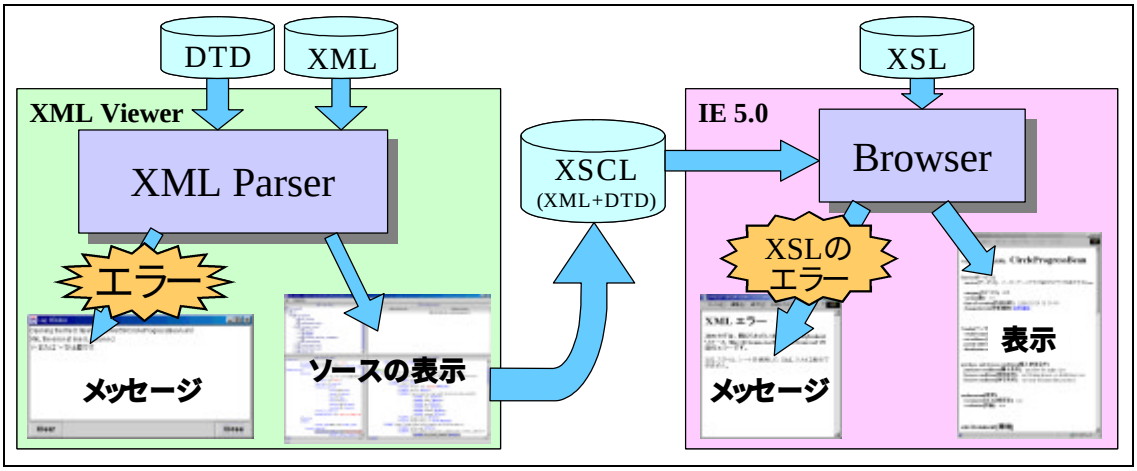


図 5.2.3　文書作成の構成図

XSCL 文書作成の流れについて説明する．

① テキストエディタ等を使用し，XSCL の DTD に基づいて XML 文書を記述する．

② 作成した XML 文書を，XSCL の DTD と共にパーサに解析させる．

③ エラーが発見された場合エラーメッセージが表示される．エラーがない場合，XML Viewer では XML や DTD のソースなどが表示される．

④ 妥当な文書であると証明された XSCL 文書は，XSL を利用してブラウザに表示することが可能である．もし，XSL に誤りがあった場合は，ブラウザがエラーメッセージを表示する．

5.3 XSCL の記述例

パーセンテージを円グラフで表示する CircleProgressBean コンポーネントを XSCL で記述した例を付録 9.5 に示す．またその表示を図 5.3.1 に示す．

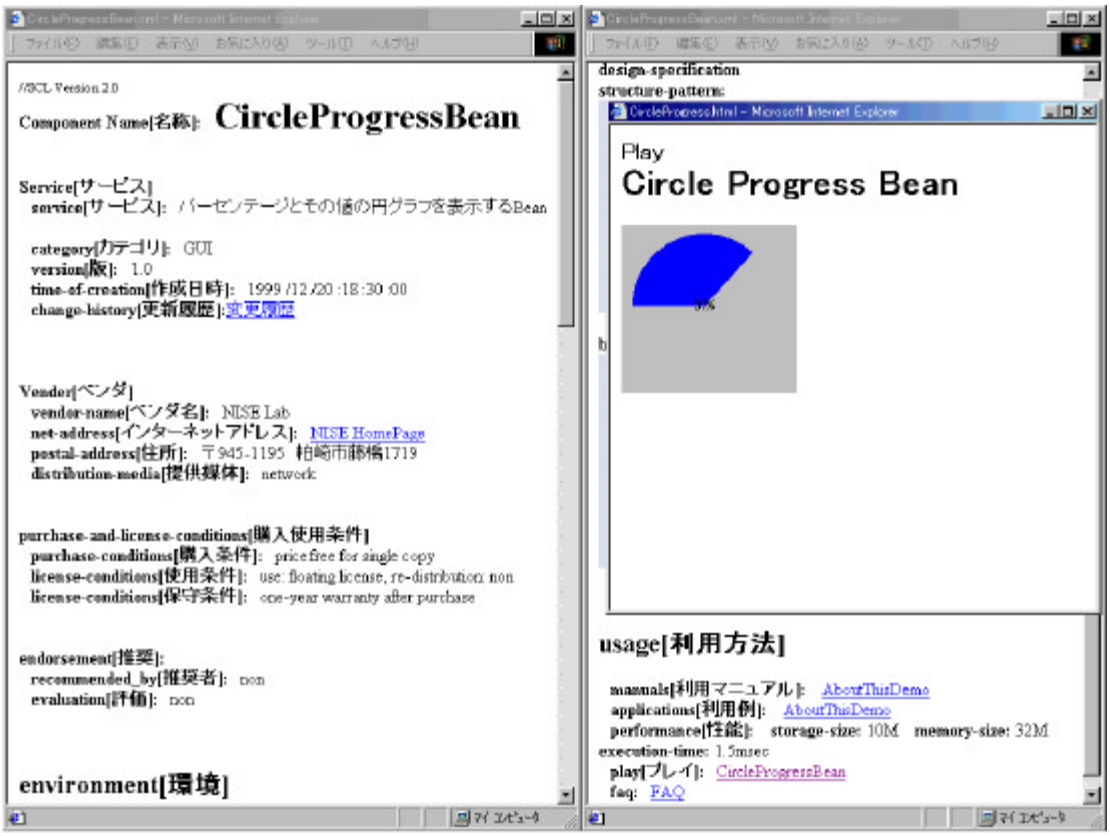


図 5.3.1 XSCL の表示例

XSCL による記述をインターネット上で公開するためには，XSCL のスタイルシート(XSL)を作成する必要がある．これは，XSCL が XML の仕様に基づき記述されているためである．図 5.3.1 は，XSL を用いて表現しており，現段階で表示可能なブラウザは Internet Explorer5.0 のみであるが，今後他のブラウザでもサポートされると思われる．

スタイルシートは，各 SCB が独自に作成する事ができる．これは，XSCL の DTD にしたがって記述された文書であれば，情報の欠落がないためである．

5.3.1 XSCL 用スタイルシート

記述例を示すために，スタイルシートを試作した．これについて説明する．

スタイルシートは，下記の目的で作成した．

(1) XSCL の開発に伴う内容の確認
(2) 記述の表示
(3) SCB との記述内容等の比較

(1) XSCL の開発に伴う内容の確認

XML は，基本的にスタイル表現を記述しない．このため，各要素をどのように設定するか，あるいは属性を使用すべきなのかなどを検討するときに，XML ソースだけでは不便であった．そこで開発に伴い，スタイルシートを簡易的に作成し，完成イメージを作ることによって，よりよい設定ができるようにした．

(2) 記述の表示

完成した XSCL 文章を表示できなければ，カタログ言語としての意味はない．先に述べたように，XSCL は XML の仕様に基づき記述されている．このため，XSCL 文書を表示させるには，スタイルシートが必要であった．

(3) SCB との記述内容等の比較

第 6 章の評価で，SCB と表示方法などの比較のため，スタイルシートを必要とした．

スタイルシートの記述方法は第４章の 4.4 XSL を参照願いたい．

作成したスタイルシートでは，JavaBeans に限り試行可能にした．試行システムを試作した理由は，利用者が実際に試行できるようにすることで，そのコンポーネントへの理解が深まるからである．試行システムに JavaBeans を使用したのは，アプレット化する事によって実行が容易に可能となるからである．Beans とアプレットは構造が似ているために，容易にアップレット化が可能である．このため，他のコンポーネントと比べ試行システムの試作が容易である．図 5.3.2 に，試行システムを示す．

試行方法としては，Web 上で Play ボタンをクリックすると，XSL に埋め込まれた JavaScript によって，新規ウィンドウが開かれ，その上にアプレット化された JavaBeans を表示させるようになっている．例 5.3.1 に XSL 内に記述された JavaScript のソースを示す．

例 5.3.1　JavaScript

```
<xsl:template match="play"><br/>
 <B>play[プレイ]:</B>
  <a><xsl:attribute name="href">JavaScript:miniW()</xsl:attribute>
                                                    <xsl:apply-templates/></a><BR/>
 <script language="JavaScript">
     function miniW()
             {window.open("<xsl:value-of select="@url"/>","","status=0,menubar=0,,")}
 </script>
</xsl:template>
```

太字で記述されているものが JavaScript のソースである．

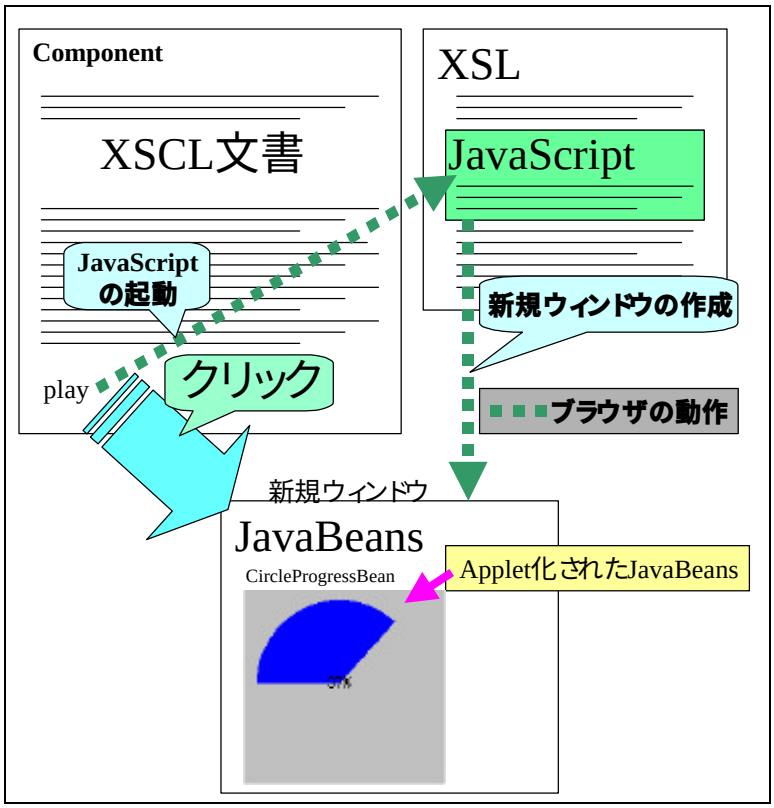


図 5.3.2　JavaBeans 試行システム

試行が可能となったことで，容易にそのコンポーネントのユーザインタフェースを理解することができるようになった．また，実際に使ってみることによって，文章では表現することが困難な動作なども容易に理解できるようになった．

XSL のソースは付録 9.6 を参照願いたい．

次章では，XSCL の評価を示す．

第6章 評価

6.1 記述方法

(1) 評価内容

SCL に，XML と HTML を用いた場合では，XML を用いた方が記述量，記述の容易性の点で優れている．これを実証するため，コンポーネント例として CircleProgressBean を用いて実際に記述し，評価した．

(2) 評価方法

SCL の文書インスタンス，DTD，XSL を，XML と HTML を用いた場合でそれぞれ記述し，その記述容量を計測，比較する．また，Beans 数の増加に伴う総記述量の推移を，次に示す式より算出し，評価した．

総記述量=文書インスタンス×Beans 数＋(DTD+XSL)

(3) 評価結果

表 6.1.1 に文書インスタンスでの比較，表 6.1.2 に DTD と XSL の比較を示す．

表 6.1.1 文書インスタンスにおける記述量の比較

	文字数(行数)	容量[KB]
XML	3,170(98)	3.71
HTML	4,678(117)	5.21

表 6.1.2 DTD，XSL における記述量の比較

記述方法	DTD		XSL		DTD+XSL	
	文字数(行数)	容量[KB]	文字数(行数)	容量[KB]	文字数(行数)	容量[KB]
XSCL	3,760(105)	4.26	6,622(203)	8.23	10,382(308)	12.49
SCL(HTML)	―	―	―	―	―	―

文書インスタンスだけを比較した場合，HTML を用いると，部品の記述に META 情報が必要なため，類似内容の記述が必要である．このため，XSCL より記述量が増える．また，XSCL では，DTD と XSL が必要となる．そのため総記述量は，Beans 数が１つの場合，HTML を用いた SCL の約３倍となる．ただし，それらは複数の XML インスタンスに対し１つでよいため，コンポーネントが増加しても新たに書く必要がない．そのため，XSCL の方が，記述量が削減できる．

総記述量の推移を計測するため，１つの Bean の記述量を一定値として考えると，Beans

数の増加に伴いその推移は図 6.1.1 のグラフで表すことができる．グラフより，Beans 数が８個以上で，XSCL が HTML の総文字数を下まわることが分かる．したがって，扱うコンポーネント数が増加するほど, XSCL の方がより少ない文字数で記述することが可能である．表 6.1.3 に実際の値を示す．

表 6.1.3 Beans 数の増加に伴う総容量(記述文字数)の推移

Beans 数	1	4	8	10	12	16	20
XSCL	13,552	23,062	35,742	42,082	48,422	61,102	73,782
SCL(HTML)	4,716	18,864	37,728	47160	56,592	75,456	94,320

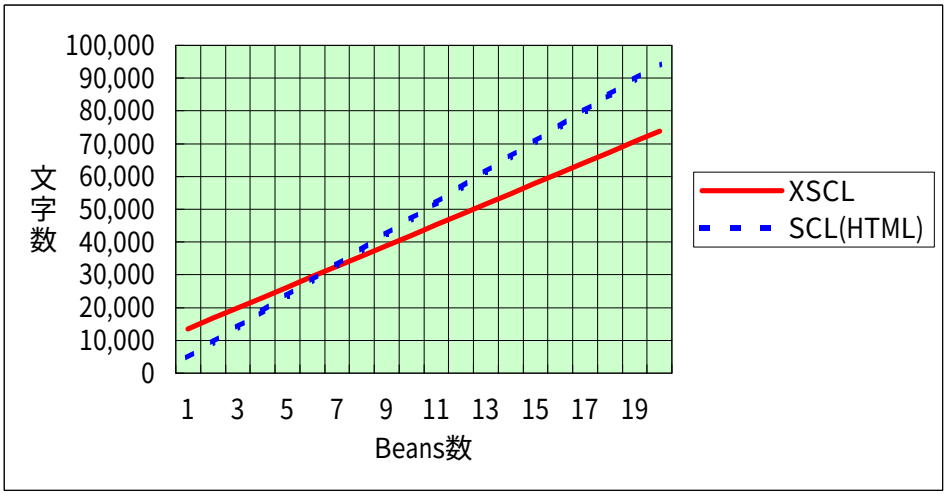


図 6.1.1 Beans 数の増加に伴う総記述量の推移

また，記述の容易性の点においても文字数の低減により記述者の負担が減る．さらに XSCL は要素が意味情報を持つため，HTML よりも記述が容易である．スタイル表現を 1 つにまとめてあるため，1 度に全ての XSCL の表示形式を変更することもできる．

6.2 記述内容

6.2.1 記述項目

(1) 評価内容

コンポーネント情報を閲覧する利用者にとって，その記述内容はより詳細で，分かりやすい方がよい. そこで, XSCL の部品情報の記述内容を表 6.2.1 に示す代表的 SCB(Software Commerce Broker)と比較し，評価した.

(2) 評価方法

図 6.2.2 に示す XSCL の部品情報の項目を SCB と比較し，次の網羅率で評価する.

網羅率=商用 SCB で利用している XSCL の項目数／XSCL の項目数

また，代表的 SCB の部品情報の解説方法についても調査し，評価した.

(3) 評価結果

表 6.2.1　実験対象 SCB

No.	ベンダ名	URL
[1]	Flashline	http://www.flashline.com/
[2]	AlphaBeans	http://www.alphaworks.ibm.com/
[3]	ComponentSource	http://www.componentsource.com/
[4]	Wildcrest Associates	http://www.wildcrest.com/
[5]	EarthWeb	http://www.gamelan.com/
[6]	PEERNET	http://www.peernet.com/barcode/java/
[7]	Javacofeebreak	http://www.javacoffeebreak.com/
[8]	Christopher Longo(個人)	http://www.users.cloud9.net/~cal/beans/

① 網羅率

表 6.2.2 に示すとおり，XSCL の記述項目の網羅率は実験対象の SCB に比べ高いことが確認できる．

表 6.2.2　実際の部品仕様情報の比較

記述比較項目	XSCL	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Name	○	○	○	○	○	○	○	○	○
Service	○	○	○	○	○	○	○	○	○
Category	○	○	○	○	○	○	○	○	○
Version	○	○	×	○	○	×	×	×	×
Time-of-creation	○	×	○	×	×	○	×	×	×
Vendor-name	○	○	○	○	○	○	○	○	○
Postal-address	○	○	×	×	×	×	○	×	×
Distribution-media	○	×	×	×	×	×	○	×	×
Purchase-conditions	○	○	○	○	○	×	○	○	×
Run-time-environment	○	○	○	○	○	×	○	○	○
File Size	○	×	○	○	○	×	×	×	×
網羅率 [%]	100	72.7	72.7	72.7	72.7	45.5	72.7	54.5	45.5

② 解説方法

SCB の解説方法の違いを調べてみると，大きく分けて次の 2 種類になる．

- いくつかの項目に分け詳しく解説している方法
- 簡単な動作説明と，ソースコードやデモなどを表示する方法

前者の解説方法の表示例を，図 6.2.1 に示す．この方法では，各項目ごとに詳細な解説があり閲覧しやすい．しかし，記述項目は統一されているがその内容は統一されていない場合が多いため，コンポーネントによって表現方法が異なっており，比較が困難である．

また，後者の解説方法の表示例を図 6.2.2 に示す．この例では，動作説明とデモを同時に表示させているため，Beans の動作をすぐに確認できる．しかし，解説方法を統一していないため，必要な項目が欠落する可能性がある．

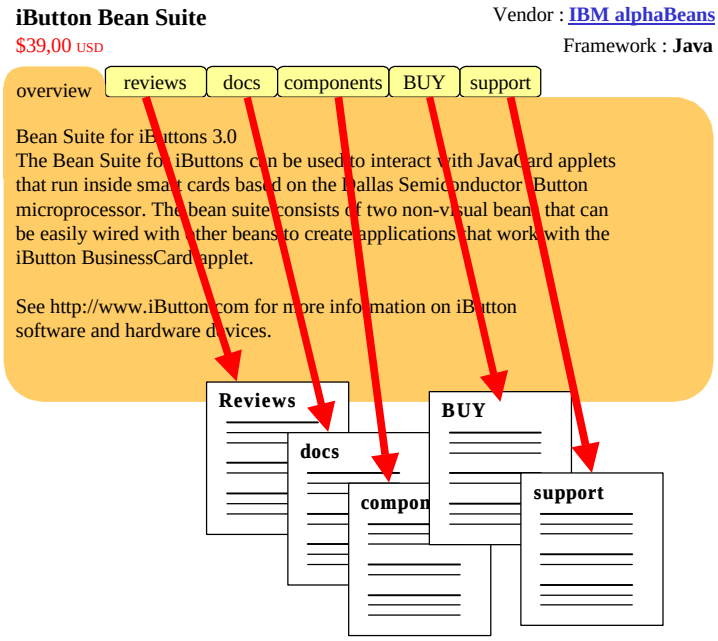


図 6.2.1　コンポーネントの解説例1

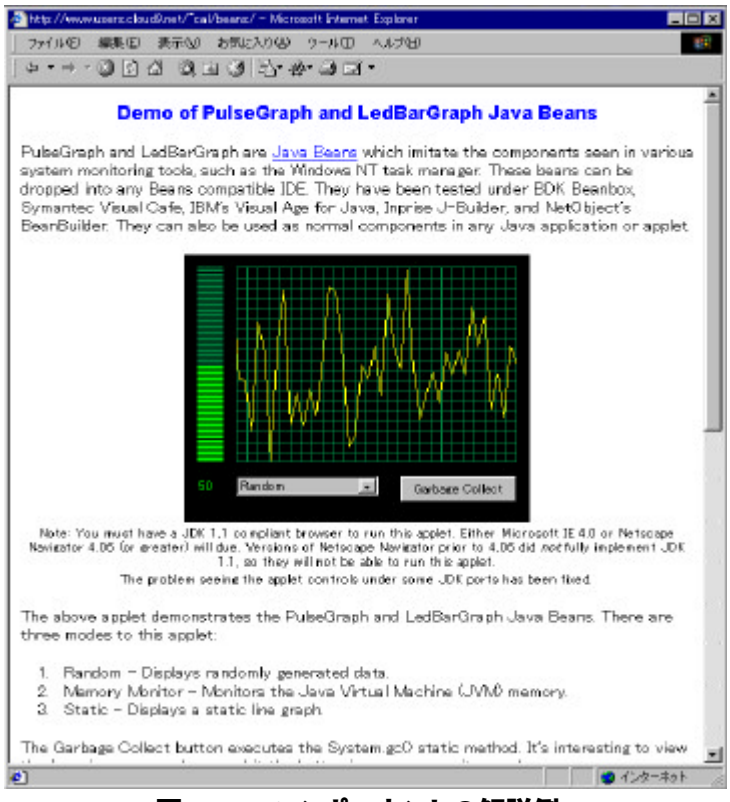


図 6.2.2　コンポーネントの解説例2

図 6.2.3 に，いくつかの項目に分けた場合の解説方法の構造を示す．これは，表 6.2.1 の[1]と[3]の SCB を例に構造を比較したものであり，記述項目を木構造で表した．この図から，対応している項目と独自の項目が存在することが分かる．よって，異なる SCB 間ではコンポーネント同士を項目別に比較することが困難である．また，項目名は同一であっても，その記述方法は SCB ごとに異なる場合もある．

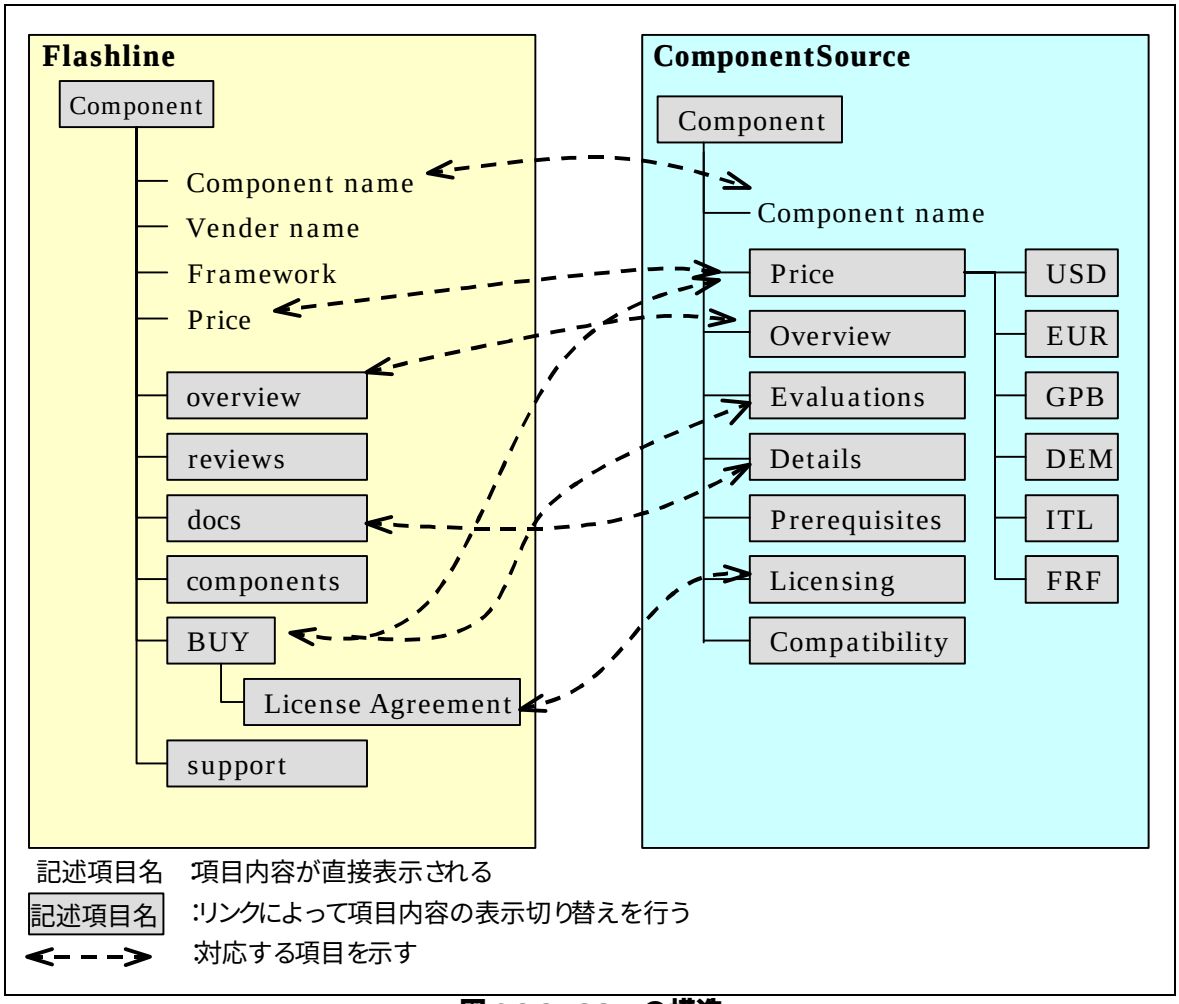


図 6.2.3　SCB の構造

XSCL では，記述項目が統一され，その網羅率も他の SCB に比べ高い．よって，項目別に他のコンポーネントとの比較するときに，部品情報の欠落によって比較できないといった問題を削減することができる．

6.2.2 コンポーネント試行システム

(1) 評価内容

部品情報を解説する場合，ただ単に文書だけで解説するよりも，実際に試行することができるようにしたい．試行が可能になることにより，利用者は，そのコンポーネントがどのようなものなのかを瞬時に把握でき，文書だけの解説による曖昧さと，それによる誤解を減らすことができる．そこで，より良い試行システムを用いるため，今回試作したコンポーネント試行システムと，すでに流通している SCB のコンポーネント試行システムとを

比較，評価する．

(2) 評価方法

表 6.2.1 に示した代表的 SCB のコンポーネント試行システムを比較分析することにより，その特徴と問題点を見出し，評価する．それにより，試作したコンポーネント試行システムを評価する．

(3) 評価結果

① 試行システムの比較

表 6.2.3 に比較した SCB とその特徴を示す．

表 6.2.3 コンポーネント試行システムの比較

SCB	料金	Bean 数	試行	試行方法	ソースコード	特徴
[1]	有料	多い	△	Applet+Beans (Download)	×	複数のコンポーネントベンダーが作った Bean を販売. 複数の項目があり，詳しく説明している．
[2]	Free, 有料	多い	○	Beans (Download)	×	複数の Bean がある. 項目ごとに分けて説明されている．一部の Bean を FLASHLINE で販売している．
[3]	有料, Free	多い	△	Evaluation version (Download)	×	複数のコンポーネントを販売しており，解説も詳しい. Component Source への登録が必要である．
[4]	Free	多い	△	Beans (Download)	×	複数の項目があり, その Bean について詳しい説明がある．比較的多くの Bean がある．
[5]	Free	少ない	○	Applet (Download)	○	Java に関するいろいろな情報を提供している. Bean についても詳しく解説してある．
[6]	有料	少ない	○	Applet+Beans	○	サンプルのみソースコードの表示及び試行が可能である.Index ページで動作などについて簡単に説明している．
[7]	Free	少ない	△	Applet	○	プログレスバーを表示する Bean など数種がある. Index で Bean について簡単に説明されている．
[8]	Free	少ない	○	Applet+Beans	○	個人のサイト．動作についての説明が簡単に記述してある．

試行　○：試行可能，△：一部試行可能，×：試行できない

② 試行方法の特徴

試行可能なサイトの試行システムは，下記の３つに分類することができた．

- Applet+Bean…BDK の MakeApplet で作った場合の様に，Bean そのものは手を加えず，ブラウザで動作させるためのアプレットと組み合わせている方法
- Applet…ソースを直接書き直し，Bean を完全にアプレット化する方法
- Download…サンプルをダウンロードし，試行する方法．Bean だけの場合，アプレット化されている場合など，ベンダーによって異なる

小規模で提供する Bean 数の少ないサイトは，Applet を用いている．その場合，Web 上で試行可能である傾向が強い．ただし，SCB によっては Applet 化できない Beans のことを考え，試行手順を統一させるため，意図的に Download させる場合もある．

逆に，Bean 数の多いサイトの試行は，Download を用いる場合が多い．この場合，ヘルプファイル，サンプル，ソースコードなどを同時に Download が可能なため，Web 上でソースコードを直接参照できない傾向がある．

また，Beans ではなくコンポーネントを配布している SCB の場合，評価版を Download することで試行を行う．

③ 試行システムの問題点

Bean をアプレット化して試行させる場合，ユーザインタフェースを持つ Bean はアプレット化することにより試行できる．しかし，ユーザインタフェースを持たない Bean についてはアプレット化し，実際にその動作確認をすることは困難なため，どのサイトでも試行できなかった．また，ダウンロードして試行させる場合，ダウンロードに時間がかかったり，利用者が自分で試行する環境を整える手間がかかるなどの問題もある．

④ 結論

これらの試行方法の比較を通し，コンポーネント試行の容易さという点に着目すると，時間をかけずにその場ですぐ試行できるアプレット化の方法が一番良いと思われる．また，ユーザインタフェースを持たない Bean についてはその動作について詳しく文書や図で説明する事により，試行ができないという欠点を補うことも可能である．従って，コンポーネント試行システムは，アプレット化する方法が，現時点で最も良いといえる．

6.3 XML Parser の解析時間

(1)目的

・IBM の XML Parser for Java 2.0.11 及び XML Parser for Java 1.1.16 を使用して, XSCL における Parser の解析時間を計測する.

・サンプルの XML 文書と XSCL 文書の解析時間を比較する.

・XML 文書の文字数と解析時間の関係を調べる.

(2)計測環境

CPU: MMX Pentium200MHz
Memory: 64MB
OS: Windows98
JavaVM: JDK 1.1.6

(3)計測方法

タイマーを挿入した Paser を起動するアプリケーションを作成し，XML ファイルを 10 回解析させ，計測時間の平均を取る.

(4)計測結果

XSCL の計測結果を表 6.3.1 に示す. 計測に使用した XSCL 文書は, CircleProgress Bean である.

表 6.3.1　XML Parser fot Java の解析時間

Version / 回数	1.1.16 [sec]	2.0.11 [sec]
1 回目	5.110	4.730
2 回目	4.290	3.630
3 回目	4.340	3.620
4 回目	4.290	3.630
5 回目	4.290	3.620
6 回目	4.280	3.680
7 回目	4.340	3.680
8 回目	4.290	3.680
9 回目	4.280	3.620
10 回目	4.340	3.630
平均	4.385	3.752

表 6.3.1 より Parser の解析時間は，新しい Version の方が 14.4%(約 600msec)短縮された.

次に，XSCL とは内容が異なるサンプル XML 文書の計測結果を表 6.3.2 に示す.

表 6.3.2　SampleXML 文書の解析時間

回数	Sample1 (4,086 文字) [sec]	Sample2 (1,449 文字) [sec]
1 回目	4.000	4.670
2 回目	4.010	3.680
3 回目	3.960	3.620
4 回目	3.960	3.680
5 回目	3.950	3.680
6 回目	4.010	3.680
7 回目	3.950	3.680
8 回目	4.010	3.680
9 回目	3.950	3.680
10 回目	3.950	3.680
平均	3.975	3.773

Sample1 と Sample2 の文字数に対し CircleProgressBean の文字数は 6,930 文字である．ただし，いずれも DTD の文字数を含む．Sample1 は CircleProgressBean よりも文字数が少ないが，解析に時間がかかった．

次に文字数と解析時間の関係を表 6.3.3 に示し，そのグラフを図 6.3.1 に示す．ただし，使用した XML 文書は DTD を含まない．

表 6.3.3　文字数と解析時間の関係

文字数 [X1000]	1.116	16.266	32.426	64.746	129.386	258.666	387.946	517.226
平均 [sec]	1.860	1.951	2.066	2.342	3.152	4.847	6.459	8.322

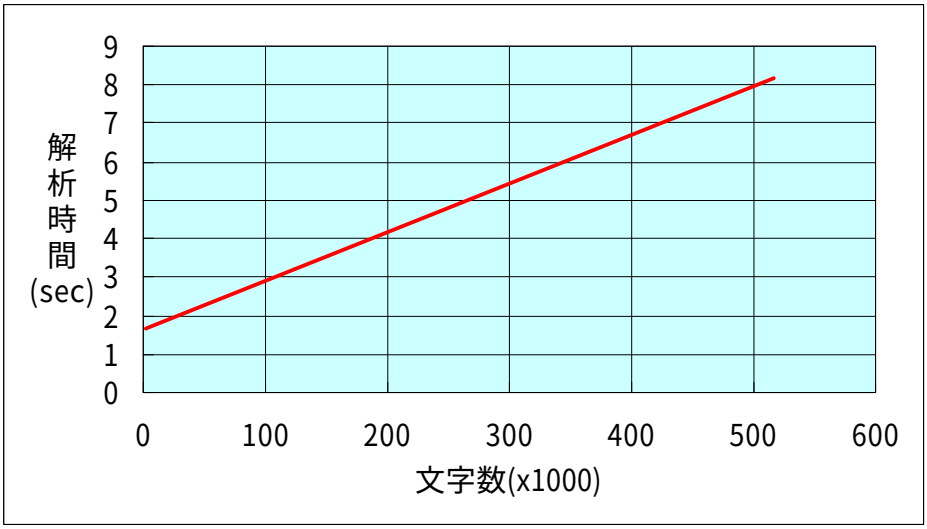


図 6.3.1　文字数と計測時間の関係

グラフより，文字数と解析時間の関係はほぼ比例関係であるといえる．しかし，Sample1 は文字数が CircleProgressBean より少ないが，解析に時間がかかっており矛盾している．Sample1 の XML 文書と XSCL 文書を比較したところ，Sample1 は属性値の数が XSCL 文

書に比べ多かった．そこで，1,116 文字の属性値を含まない XML 文書と属性値を含む XML 文書を作成し，比較した．表 6.3.4 に，属性の有無により生じる解析時間の違いを示す．

表 6.3.4　属性の有無により生じる解析時間の違い(DTD は含まない)

回数	属性　無 [sec]	属性　有 [sec]	CircleProgressBean [sec]
1 回目	1.870	2.250	2.750
2 回目	1.680	2.250	2.090
3 回目	1.870	2.260	2.080
4 回目	1.860	2.300	2.090
5 回目	1.860	2.300	2.090
6 回目	1.870	2.260	2.030
7 回目	1.810	2.250	2.030
8 回目	1.870	2.250	2.030
9 回目	1.870	2.250	2.090
10 回目	1.860	2.250	2.090
平均	1.860	2.262	2.137

このことから，属性の解析時間が 21.6%(約 402msec)増加していることが判った．

(5)結論

XML 文書の解析は，文字数の増加に伴い解析時間も比例して増加する．また，XML 文書に属性を含めることで解析時間はさらに増加する．このことから，XML 文書の文字数が多くなる場合には，属性を多用すべきではない．XSCL 文書では，属性の利用が少ないことから，素早い解析が可能である．

第7章 まとめ

本研究では，XML を用いて XSCL を開発した．そして，SCL の DTD を作成し，XML で記述することによって意味情報の表現が可能となり，XSL を用いることで，スタイル表現の分離が可能となった．さらに，SCL の表現に HTML と XML を用いた場合の両者を比較し，他の SCB とも比較することで XSCL の特徴を評価した．

今後の課題として，次の３つがあげられる．

最初に，本研究では Beans を中心に記述の評価をしてきたが，Bean 以外のコンポーネントでは同様に扱うことができるのか，またその問題点は何かということも評価する必要がある．

次に，現在，XSCL の記述は人手によるものであったが，支援ツールなどを利用した場合の効率の向上についても評価する必要がある．

最後に，XSCL の応用として，インターネット上でコンポーネントを登録・試行するサービスを提供する JavaBeans コンポーネントプレイヤにおける活用について検討する必要もある．

参考文献

[1] 神吉達郎，Java プログラミング入門，オーム社，1996.

[2] 河西朝雄，Java 入門，技術評論社，1998.

[3] J.O'Neil，Teach Yourself Java，McGraw-Hill，1999 [トップスタジオ(訳)，独習 Java，翔泳社，1999].

[4] 青山幹雄ほか，コンポーネントウェア，共立出版，1998.

[5] W. Ernst，Presenting ActiveX，Sams.net Publishing，1996 [日方俊二(訳)，ActiveX を知る，プレンティスホール出版，1996].

[6] M. C. Chan, et al., 1001 Java Programmer's Tips, Jamsa Press, 1997 [舟木将彦(訳)，Java プログラミング 1001Tips，オーム社，1997].

[7] J. Feghhi, Web Developer's Guide to Java Beans, The Coriolis Group, 1997 [豊福剛(訳)，Java Beans 入門，インターナショナルトムソンパブリッシングジャパン，1997].

[8] E. R. Harold, JavaBeans, IDG Books, 1998 [舟木将彦(訳)，Java で作るコンポーネントソフトウェア，オーム社，1998].

[9] 青山幹雄ほか，ソフトウェアコマースのためのカタログ記述言語 SCL，情報処理学会ソフトウェア工学研究会，No. 115-5, Jul, 1997, pp. 33-40.

[10] C. F. Goldfarb, et al., The XML Handbook, Prentice Hall, 1998 [安藤慶一(訳)，XML 技術大全，ピアソン・エデュケーション，1999].

[11] XML/SGML サロン，標準 XML 完全解説，技術評論社，1998.

[12] 村田　真，XML 入門，日本経済新聞社，1998.

[13] 齋木太郎ほか，JavaBeans コンポーネントプレイヤの開発，情報処理学会第 59 回全国大会論文集，No. 6ZB-06, Mar. 2000.

[14] IBM AlphaWorks http://www.alphaworks.ibm.com/

付録

Circle Progress Bean.java

```
package beans.cprogress;

import java.awt.*;
import java.beans.*;

public class CircleProgressBean extends Component {
  protected int maximum = 100;
  public synchronized void setMaximum (int m) {
    if (maximum != m) {
      maximum = m;
      repaint ();
    }
  }
  public int value;
    public synchronized void setValue (int v) {
      if (value != v) {
        value = v;
        repaint ();
        fireValueChange ();
      }
    }
  public int getValue (){
    return value;
  }
  protected Color barground;
    public void setBarground (Color c) {
      barground = c;
      repaint ();
    }
  public Color getBarground () {
    return barground;
  }
  protected PropertyChangeSupport listeners = new PropertyChangeSupport (this);
```

```
public void addPropertyChangeListener (PropertyChangeListener l) {
   listeners.addPropertyChangeListener (l);
}
public void removePropertyChangeListener (PropertyChangeListener l) {
   listeners.removePropertyChangeListener (l);
}
protected Integer oValue = new Integer (value);
protected void fireValueChange () {
   listeners.firePropertyChange
   ("value", oValue, oValue = new Integer (value));
}
public void update (Graphics g) {
   paint (g);
}
public void paint (Graphics g) {
   int w = getSize ().width, h = getSize ().height, m = Math.max (1, maximum),
         v = Math.max (0, Math.min (m, value)), b = v / m;
   g.setColor (Color.blue);
   if (v > 0) {
      if (barground != null)
         g.setColor (barground);
         g.fillArc (0, 0, w-1, h-1, 180, -360*v/100);
      }
      g.setColor (getForeground ());
      String pc = (100 * v / m)+ "%";
      FontMetrics fm = g.getFontMetrics ();
      g.drawString (pc, (w - fm.stringWidth (pc)) / 2,
            (h + fm.getAscent () - fm.getDescent ()) / 2);
   }
   public Dimension getMinimumSize () {
      FontMetrics fm = getFontMetrics (getFont ());
      return new Dimension (fm.stringWidth ("100%") + 10,
            fm.getAscent () + fm.getDescent () + 10);
   }
   public Dimension getPreferredSize () {
      FontMetrics fm = getFontMetrics (getFont ());
```

```
        int h = fm.getAscent () + fm.getDescent () + 14;
        return new Dimension (h*5 , h*5);
    }
    public Dimension getMaximumSize (){
        Dimension d = getPreferredSize ();
        return new Dimension (d.width * 10, d.height * 10);
    }
}
```

Circle Progress Bean BeanInfo.java

```
package beans.cprogress;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;

public class CircleProgressBeanBeanInfo
  extends SimpleBeanInfo{
    public Image getIcon (int iconKind) {
      if (iconKind == ICON_COLOR_16x16) {
        return loadImage ("CircleProgressBeanIconColor16.gif");
      } else {
        return null;
      }
    }
  }
```

SCL の HTML による記述

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=ISO-2022-JP">
<META NAME="SCL" CONTENT="//SCL Version 1.0">
<META NAME="name" CONTENT="CircleProgressBean ">
<META  NAME="service"  CONTENT="パーセンテージとその値の円グラフを表示する
```

```
Bean">
<META NAME="category" CONTENT="GUI">
<META NAME="version" CONTENT="1.0">
<META NAME="time-of-creation" CONTENT="1999/12/20:18:30:00">
<META NAME="change-history" CONTENT="http://www.niit.ac.jp/nise.index.html">
<META NAME="Vendor" CONTENT="">
<META NAME="vendor-name" CONTENT="NISE Lab.">
<META NAME="net-address" CONTENT="http://www.niit.ac.jp/nise/index.html">
<META NAME="postal-ddress" CONTENT="〒945-1195 柏崎市藤橋 1719">
<META NAME="distribution-media" CONTENT="network">
<META NAME="purchase-and-license-conditions" CONTENT="">
<META NAME="purchase-conditions"CONTENT="price:free for single copy">
<META NAME="license-condition"
                                CONTENT="use:floating license,re-distribution:non">
<META NAME="maintenance-condition"
                                    CONTENT="one-year warranty after purchase">
<META NAME="endorsement" CONTENT="">
<META NAME="recommended_by" CONTENT="non">
<META NAME="evaluation" CONTENT="non">
<META NAME="environment" CONTENT="">
<META NAME="run-time-environment" CONTENT="">
<META NAME="os"
            CONTENT="Windows95,Windows98,WindowsNT:version &GT;= 4.0">
<META NAME="object-broker" CONTENT="">
<META NAME="web-browser" CONTENT=":Internet Explorer:version &GT;= 5.0">
<META NAME="design-time-environment" CONTENT="">
<META NAME="os" CONTENT="Windows 98">
<META NAME="framework" CONTENT="JDK &GT;= 1.1.6">
<META NAME="design-environment" CONTENT="BDK &GT;= 1.0">
<META NAME="language" CONTENT="Java">
<META NAME="specification" CONTENT="">
<META NAME="interface" CONTENT="">
<META NAME="composition" CONTENT="---from---:Version---">
<META NAME="design-specification" CONTENT="">
<META NAME="structure-pattern" CONTENT="Now_Painting.gif">
<META NAME="scenario-pattern" CONTENT="Now_Painting.gif">
```

```
<META NAME="usage" CONTENT="">
<META NAME="manuals"
        CONTENT="http://ies045.iee.niit.ac.jp/~team-a/scl/AboutThisDemo.html">
<META NAME="applications"
        CONTENT="http://ies045.iee.niit.ac.jp/~team-a/scl/AboutThisDemo.html">
<META NAME="Performance"
        CONTENT="storage-size:10M,memory-size:32M,execution-time:1.5m sec">
<META NAME="play" CONTENT="CircleProgress/CircleProgress.html">
<META NAME="faq" CONTENT="http://www.niit.ac.jp/nise/faq.html">
<TITLE>CircleProgressBean</TITLE>
</HEAD>

<BODY BGCOLOR="#FFFFFF">
<h2>component[コンポーネント]</h2>
//SCL Version 1.0<BR><BR>
<B>name[名称]:</B>
    CircleProgressBean<BR><BR><BR><BR>
<B>Service[サービス]</B><BR>
  <B>Service[サービス]:</B>  パーセンテージとその値の円グラフを表示する Bean<BR>
  <B>category[カテゴリ]:</B>  GUI<BR>
  <B>version[版]:</B>  1.0<BR>
  <B>time-of-creation[作成日時]:</B>  1999 /12 /20:18:30:00<BR>
  <B>change-history[更新履歴]:</B>
   <a href="http://ies045.iee.niit.ac.jp/~team-a/scl/AboutThisDemo.html">変更履歴</a>
<BR><BR><BR>
<B>Vender[ベンダ]</B><BR>
  <B>vendor-name[ベンダ名]:</B>  NISE Lab.<BR>
  <B>net-address[インターネットアドレス]:</B>
       <a href="http://ies045.niit.ac.jp/nise/index.html">NISE Home Page</a><BR>
  <B>postal-address[住所]:</B>  〒945-1195 柏崎市藤橋 1719<BR>
  <B>distribution-media[提供媒体]:</B>  network<BR>
<BR><BR>
<B>purchase-and-license-conditions[購入使用条件]</B><BR>
  <B>purchase-conditions[購入条件]:</B>  price:free for single copy<BR>
  <B>license-conditions[使用条件]:</B>  use:floating license,re-distribution:non<BR>
  <B>license-conditions[保守条件]:</B>  one-year warranty after purchase<BR>
```

```
<BR><BR>
<B>endorsement[推奨]:</B><BR>
  <B>recommended_by[推奨者]:</B>  non<BR>
  <B>evaluation[評価]:</B>  non<BR>
<BR><BR>
<H2>environment[環境]</H2>
<B>run_time-environment[実行環境]</B><BR>
<B>os:</B>  Windows95,Windows98,WindowsNT Version &GT;= 4.0<BR>
<B>web-browser:</B>  Internet Explorer:version &GT;= 5.0<BR>
<BR>
<B>design_time-environment[設計環境]</B><BR>
<B>os:</B>  Windows 98<BR>
<B>framework:</B>  JDK:version &GT;= 1.1.6<BR>
<B>design-environment:</B>  BDK:Version &GT;= 1.0<BR>
<B>language:</B>  Java<BR>
<BR><BR>
<H2>specification[仕様]</H2>
  <B>interface[インターフェース]:</B>
<BR><BR><BR>
<B>Composition</B><BR>
    ---from---Version---<BR><BR>
<B>design-specification</B><BR>
<B>structure-pattern:</B><BR>
    <img src="Now_Painting.gif"><BR><BR>
<BR>
<B>scenario-pattern:</B><BR>
    <img src="Now_Painting.gif"><BR><BR>
<BR>
<BR>
<H2>usage[利用方法]</H2>
  <B>manuals[利用マニュアル]:</B>
  <a href="http://ies045.iee.niit.ac.jp/~team-a/scl/AboutThisDemo.html">
                                                    AboutThisDemo</a>
<BR>
  <B>applications[利用例]:</B>
  <a href="http://ies045.iee.niit.ac.jp/~team-a/scl/AboutThisDemo.html">
```

```
                                                    AboutThisDemo</a><BR>
  <B>performance[性能]:</B>  <B>storage-size:</B>10M  <B>memory-size:32M</B>
<B>execution-time:</B>1.5m sec<BR>
  <B>play[プレイ]:</B>
  <a href=" http://ies045.iee.niit.ac.jp/~team-a/scl/AboutThisDemo.html ">
                                                    CircleProgressBean</a><BR>
  <B>faq:</B>
  <a href="http://ies045.iee.niit.ac.jp/~team-a/scl/AboutThisDemo.html">
                                                    AboutThisDemo</a>
<BR><BR><BR>
</BODY>
</HTML>
```

XSCL のDTD

Component Part (scl.dtd)

```
<?xml version="1.0" encoding="shift_jis"?>
<!ELEMENT SCL (Component)>
  <!ELEMENT Component (scl_version, component_name, Commerce_part,
Environment_part, Specification_part, Usage_part)>
    <!ELEMENT scl_version EMPTY>
    <!ATTLIST scl_version version_identifier CDATA #REQUIRED>
    <!ELEMENT component_name (#PCDATA)>

    <!ENTITY % commerce SYSTEM "commerce.dtd">
      %commerce;
    <!ENTITY % environment SYSTEM "environment.dtd">
      %environment;

    <!ENTITY % specification SYSTEM "specification.dtd">
      %specification;

    <!ENTITY % usage SYSTEM "usage.dtd">
      %usage;
```

Commerce Part (commerce.dtd)

```
<?xml version="1.0" encoding="shift_jis"?>
<!ELEMENT  Commerce_part
          (Service,Vender,Puchase_license_conditions,Endorsement)>
  <!ELEMENT Service (service,category,version,time_of_creation,(change-history)?)>
    <!ELEMENT service (#PCDATA)>
    <!ELEMENT category EMPTY>
    <!ATTLIST category list (GUI | CUI) #REQUIRED>
    <!ELEMENT version (#PCDATA)>
    <!ELEMENT time_of_creation (year,month,day,hour,minutes,seconds)>
       <!ELEMENT year (#PCDATA)>
       <!ELEMENT month (#PCDATA)>
       <!ELEMENT day (#PCDATA)>
       <!ELEMENT hour (#PCDATA)>
       <!ELEMENT minutes (#PCDATA)>
       <!ELEMENT seconds (#PCDATA)>
    <!ELEMENT change-history (#PCDATA)>
    <!ATTLIST change-history url CDATA #REQUIRED>
  <!ELEMENT Vender (vender_name,contact_address,distribution_media+)>
    <!ELEMENT vender_name (#PCDATA)>
    <!ELEMENT contact_address (net_contact_address,(postal_contact_address)?)>
       <!ELEMENT net_contact_address (#PCDATA)>
       <!ATTLIST net_contact_address url CDATA #REQUIRED>
       <!ELEMENT postal_contact_address (#PCDATA)>
    <!ELEMENT distribution_media EMPTY>
    <!ATTLIST distribution_media list (network | CD-ROM | FDD) #REQUIRED>
  <!ELEMENT Puchase_license_conditions
                (puchase_conditions,license_conditions,maintenance_conditions)>
    <!ELEMENT puchase_conditions (price,purchase-conditions)>
    <!ELEMENT price (#PCDATA)>
    <!ELEMENT purchase-conditions (#PCDATA)>
    <!ELEMENT license_conditions (#PCDATA)>
    <!ELEMENT maintenance_conditions (#PCDATA)>
  <!ELEMENT Endorsement (endorsement_list)>
    <!ELEMENT endorsement_list (recommended-by,evaluation)>
       <!ELEMENT recommended-by (#PCDATA)>
       <!ELEMENT evaluation (#PCDATA)>
```

Environment Part

```
<?xml version="1.0" encoding="shift_jis"?>
<!ELEMENT Environment_part
    (run-time_environment,design-time_environment,processor,memory_size,file_size)>
  <!ELEMENT run-time_environment (environment)*>
  <!ELEMENT design-time_environment (environment)*>
    <!ELEMENT environment (#PCDATA|version_expression)*>
    <!ATTLIST environment list (os | object-broker | web-browser | framework |
                                design-environment | language) #REQUIRED >
      <!ELEMENT version_expression (#PCDATA)>
  <!ELEMENT processor (#PCDATA)>
  <!ELEMENT memory_size (#PCDATA)>
  <!ELEMENT file_size (#PCDATA)>
```

Specification Part

```
<?xml version="1.0"    encoding="shift_jis"?>
<!ELEMENT Specification_part (module, composition, (design-specification)?)>
  <!ELEMENT module (interface)>
  <!ATTLIST module name CDATA #IMPLIED>
    <!ELEMENT interface ANY>
      <!ELEMENT idl (#PCDATA)>
      <!ELEMENT interface-service (#PCDATA)>
  <!ELEMENT composition (composition-type,composition-interface)?>
    <!ELEMENT composition-type (#PCDATA)>
    <!ELEMENT composition-interface
                    (identifier1,composition-identifier,identifier2,version_expression)>
      <!ELEMENT identifier1 (#PCDATA)>
      <!ELEMENT composition-identifier EMPTY>
      <!ATTLIST composition-identifier
                                list (CDATA|inherit-from|aggregate-to) #REQUIRED>
      <!ELEMENT identifier2 (#PCDATA)>
  <!ELEMENT design-specification ((structure_pattern)?,(behavior_pattern)?)>
    <!ELEMENT structure_pattern EMPTY>
    <!ATTLIST structure_pattern url CDATA #REQUIRED>
    <!ELEMENT behavior_pattern EMPTY>
```

```
    <!ATTLIST behavior_pattern url CDATA #REQUIRED>
```

Usage Part

```
<?xml version="1.0"    encoding="shift_jis"?>
<!ELEMENT Usage_part (usage,(performance)*, (play)?, (faq)?)>
  <!ELEMENT usage ((manual)?,(applications)?)>
    <!ELEMENT manual (#PCDATA)>
    <!ATTLIST manual url CDATA #REQUIRED>
    <!ELEMENT applications (#PCDATA)>
    <!ATTLIST applications url CDATA #REQUIRED>
  <!ELEMENT performance (performance_identifier+,(expression)?)>
    <!ELEMENT performance_identifier (#PCDATA)>
    <!ATTLIST performance_identifier
              identifier (storage-size | memory-size | execution-time | text) #REQUIRED >
    <!ELEMENT expression (#PCDATA)>
  <!ELEMENT play (#PCDATA)>
  <!ATTLIST play url CDATA #REQUIRED>
  <!ELEMENT faq (#PCDATA)>
  <!ATTLIST faq url CDATA #REQUIRED>
```

XSCL の記述例(Circle Progress Bean)

```
<?xml version="1.0" encoding="shift_jis"?>
<?xml:stylesheet type="text/xsl" href="SCL.xsl" ?>
<!DOCTYPE SCL SYSTEM "scl.dtd">
<SCL>
<Component>
  <scl_version version_identifier="2.0"/>
  <component_name>CircleProgressBean</component_name>
  <Commerce_part>
    <Service>
      <service>パーセンテージとその値の円グラフを表示する Bean</service>
      <category list="GUI"/>
      <version>1.0</version>
      <time_of_creation>
        <year>1999</year>
```

```
      <month>12</month>
      <day>20</day>
      <hour>18</hour>
      <minutes>30</minutes>
      <seconds>00</seconds>
    </time_of_creation>
    <change-history
                  url="http://ies045.iee.niit.ac.jp/~team-a/scl/AboutThisDemo.html">
      変更履歴</change-history>
</Service>
<Vender>
    <vender_name>NISE Lab</vender_name>
    <contact_address>
      <net_contact_address url="http://ies045.niit.ac.jp/nise/index.html">
                                          NISE HomePage</net_contact_address>
      <postal_contact_address>〒945-1195　柏崎市藤橋 1719
                                                  </postal_contact_address>
    </contact_address>
    <distribution_media list="network" />
</Vender>
<Puchase_license_conditions>
    <puchase_conditions>
      <price>free</price>
      <purchase-conditions>for single copy</purchase-conditions>
    </puchase_conditions>
    <license_conditions>use: floating license,
                                          re-distribution: non</license_conditions>
    <maintenance_conditions>one-year warranty after purchase
                                                  </maintenance_conditions>
</Puchase_license_conditions>
<Endorsement>
    <endorsement_list>
      <recommended-by>non</recommended-by>
      <evaluation>non</evaluation>
    </endorsement_list>
</Endorsement>
```

```
</Commerce_part>

<Environment_part>
  <run-time_environment>
    <environment list="os">Windows95</environment>
    <environment list="os">Windows98</environment>
    <environment list="os">WindowsNT
                  <version_expression>>=4.0</version_expression></environment>
    <environment list="web-browser">Internet Explorer
                  <version_expression>>=5.0</version_expression></environment>
  </run-time_environment>
  <design-time_environment>
    <environment list="os">Windows98</environment>
    <environment list="framework">JDK
                <version_expression>>=1.1.6</version_expression></environment>
    <environment list="design-environment">BDK
                  <version_expression>>=1.0</version_expression></environment>
    <environment list="language">Java</environment>
  </design-time_environment>
  <processor>MMX Pentium 200MHz</processor>
  <memory_size>32MB</memory_size>
  <file_size>2.94KB</file_size>
</Environment_part>

<Specification_part>
  <module>
    <interface></interface>
  </module>
  <composition>
    <composition-type></composition-type>
    <composition-interface>
      <identifier1>CircleProgressBean</identifier1>
      <composition-identifier list="aggregate-to"/>
      <identifier2>CircleProgressBean</identifier2>
      <version_expression>1.0</version_expression>
    </composition-interface>
```

```
    </composition>
    <design-specification>
      <structure_pattern url="Now_Painting.gif"/>
      <behavior_pattern url="Now_Painting.gif"/>
    </design-specification>
  </Specification_part>

  <Usage_part>
    <usage>
      <manual url="http://ies045.iee.niit.ac.jp/~team-a/scl/AboutThisDemo.html">
                                                        AboutThisDemo</manual>
      <applications url="http://ies045.iee.niit.ac.jp/~team-a/scl/AboutThisDemo.html">
                                                      AboutThisDemo</applications>
    </usage>
    <performance>
      <performance_identifier identifier="storage-size">10M</performance_identifier>
      <performance_identifier identifier="memory-size">
                                                            32M</performance_identifier>
      <performance_identifier identifier="execution-time">
                                                          1.5msec</performance_identifier>
    </performance>
    <play url="CircleProgress/CircleProgress.html">CircleProgressBean</play>
    <faq url="http://ies045.iee.niit.ac.jp/~team-a/scl/AboutThisDemo.html">FAQ</faq>
  </Usage_part>
</Component>
</SCL>
```

XSCL の XSL

```
<?xml version="1.0" encoding="shift_jis" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl"
          xmlns:html="http://www.w3.org/TR/REC-html40"
          result-ns="html">
<xsl:template>
  <xsl:copy>
    <xsl:apply-templates/>
```

```
  </xsl:copy>
</xsl:template>

<xsl:template match="*|@*" priority="-1">
  <xsl:copy>
    <xsl:apply-templates select="@*|*|text()" />
  </xsl:copy>
</xsl:template>

<xsl:template match="/">
  <HTML>
    <HEAD>
      <TITLE><xsl:value-of select='/Component/component_name'/></TITLE>
    </HEAD>
    <BODY>
      <xsl:apply-templates/>
    </BODY>
  </HTML>
</xsl:template>

<xsl:template match="scl_version">
  <span style="font-size:10pt">
     //SCL Version <xsl:value-of select="@version_identifier"/>
  </span>
  <BR/>
</xsl:template>
<xsl:template match="component_name">
    <B>Component Name[名称]:
       <span style="font-size:25pt"><xsl:apply-templates/></span></B><BR/>
</xsl:template>

<xsl:template match="Commerce_part">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="Service"><BR/><BR/>
```

```
  <B>Service[サービス]</B><BR/><xsl:apply-templates/>
</xsl:template>

<xsl:template match="service">
  <B>service[サービス]:</B>  <xsl:apply-templates/>  <BR/>
</xsl:template>
<xsl:template match="category">
  <B>category[カテゴリ]:</B>  <xsl:value-of select="@list"/>  <BR/>
</xsl:template>
<xsl:template match="version">
  <B>version[版]:</B>  <xsl:apply-templates/>  <BR/>
</xsl:template>
<xsl:template match="time_of_creation">
  <B>time-of-creation[作成日時]:</B>  <xsl:apply-templates/>  <BR/>
</xsl:template>
<xsl:template match="year"><xsl:apply-templates/>/</xsl:template>
<xsl:template match="month"><xsl:apply-templates/>/</xsl:template>
<xsl:template match="day"><xsl:apply-templates/>:</xsl:template>
<xsl:template match="hour"><xsl:apply-templates/>:</xsl:template>
<xsl:template match="minutes"><xsl:apply-templates/>:</xsl:template>
<xsl:template match="seconds"><xsl:apply-templates/></xsl:template>
<xsl:template match="change-history">
  <B>change-history[更新履歴]:</B>
  <a><xsl:attribute name="href"><xsl:value-of select="@url"/></xsl:attribute>
                                                <xsl:apply-templates/></a><BR/><BR/>
</xsl:template>

<xsl:template match="Vender"><BR/><BR/>
  <B>Vender[ベンダ]</B><BR/><xsl:apply-templates/>
</xsl:template>

<xsl:template match="vender_name">
  <B>vendor-name[ベンダ名]:</B>  <xsl:apply-templates/>  <BR/>
</xsl:template>
<xsl:template match="contact_address">
  <xsl:apply-templates/>
```

```
</xsl:template>
<xsl:template match="net_contact_address">
  <B>net-address[インターネットアドレス]:</B>
  <a><xsl:attribute name="href"><xsl:value-of select="@url"/></xsl:attribute>
                                                    <xsl:apply-templates/></a><BR/>
</xsl:template>
<xsl:template match="postal_contact_address">
  <B>postal-address[住所]:</B>  <xsl:apply-templates/>  <BR/>
</xsl:template>
<xsl:template match="distribution_media">
  <B>distribution-media[提供媒体]:</B>
  <xsl:value-of select="@list"/>  <BR/>
</xsl:template>

<xsl:template match="Puchase_license_conditions"><BR/><BR/>
  <B>purchase-and-license-conditions[購入使用条件]</B><BR/><xsl:apply-templates/>
</xsl:template>

<xsl:template match="puchase_conditions">
  <B>purchase-conditions[購入条件]:</B>  <xsl:apply-templates/>
</xsl:template>
<xsl:template match="price">
  price:<xsl:apply-templates/>
</xsl:template>
<xsl:template match="purchase-conditions">
  <xsl:apply-templates/><BR/>
</xsl:template>
<xsl:template match="license_conditions">
  <B>license-conditions[使用条件]:</B>  <xsl:apply-templates/>  <BR/>
</xsl:template>
<xsl:template match="maintenance_conditions">
  <B>license-conditions[保守条件]:</B>  <xsl:apply-templates/>  <BR/>
</xsl:template>


<xsl:template match="Endorsement"><BR/><BR/>
```

```
  <B>endorsement[推奨]:</B><BR/><xsl:apply-templates/>
</xsl:template>

<xsl:template match="endorsement_list">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="recommended-by">
  <B>recommended_by[推奨者]:</B>  <xsl:apply-templates/>  <BR/>
</xsl:template>
<xsl:template match="evaluation">
  <B>evaluation[評価]:</B>  <xsl:apply-templates/>  <BR/><BR/>
</xsl:template>

<xsl:template match="Environment_part"><BR/><BR/>
        <h2>environment[環境]</h2><xsl:apply-templates/>
</xsl:template>

<xsl:template match="run-time_environment">
  <B>run_time-environment[実行環境]</B><BR/><xsl:apply-templates/>
</xsl:template>

<xsl:template match="environment">
  <B><xsl:value-of select="@list"/>:</B><xsl:apply-templates/><BR/>
</xsl:template>

<xsl:template match="version_expression">
  version<xsl:apply-templates/>
</xsl:template>

<xsl:template match="design-time_environment"><P/>
  <B>design_time-environment[設計環境]</B><BR/><xsl:apply-templates/>
</xsl:template>

<xsl:template match="processor"><BR/>
  <B>processor[プロセッサ]:</B>
```

```
    <xsl:apply-templates/><BR/>
</xsl:template>

<xsl:template match="memory_size"><BR/>
    <B>memory_size[メモリ]:</B>
    <xsl:apply-templates/><BR/>
</xsl:template>

<xsl:template match="file_size"><BR/>
    <B>file_size[ファイルサイズ]:</B>
    <xsl:apply-templates/><BR/>
</xsl:template>


<xsl:template match="Specification_part"><BR/><BR/>
  <h2>specification[仕様]</h2><xsl:apply-templates/>
</xsl:template>

<xsl:template match="module"><B>
  Module[モジュール]:</B><xsl:value-of select="@name"/><BR/>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="interface">
  <B>interface[インタフェース]:</B><BR/><xsl:apply-templates/>
</xsl:template>

<xsl:template match="idl">
  <xsl:apply-templates/><BR/><BR/>
</xsl:template>
<xsl:template match="interface-service">
  <B>interface-service:</B><BR/><xsl:apply-templates/>
</xsl:template>

<xsl:template match="composition"><BR/><BR/>
  <B>Composition:</B><BR/><xsl:apply-templates/>
```

```
</xsl:template>

<xsl:template match="composition-interface">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="identifier1">
  <xsl:apply-templates/>
</xsl:template>
<xsl:template match="composition-identifier">
  <xsl:value-of select="@list"/>
</xsl:template>
<xsl:template match="identifier2">
 <xsl:apply-templates/>:
</xsl:template>

<xsl:template match="design-specification"><BR/><BR/>
  <B>design-specification</B><BR/><xsl:apply-templates/>
</xsl:template>

<xsl:template match="structure_pattern">
  <B>structure-pattern:</B><BR/>
  <img><xsl:attribute name="src"><xsl:value-of select="@url"/></xsl:attribute></img>
  <BR/><BR/>
</xsl:template>

<xsl:template match="behavior_pattern">
  <B>behavior-pattern:</B><BR/>
  <img><xsl:attribute name="src"><xsl:value-of select="@url"/></xsl:attribute></img>
  <BR/><BR/>
</xsl:template>


<xsl:template match="Usage_part"><BR/><BR/>
  <h2>usage[利用方法]</h2><xsl:apply-templates/>
</xsl:template>
```

```
<xsl:template match="manual">
  <B>manuals[利用マニュアル]:</B>
  <a><xsl:attribute name="href"><xsl:value-of select="@url"/></xsl:attribute>
                                                    <xsl:apply-templates/></a><BR/>
</xsl:template>
<xsl:template match="applications">
  <B>applications[利用例]:</B>
  <a><xsl:attribute name="href"><xsl:value-of select="@url"/></xsl:attribute>
                                                    <xsl:apply-templates/></a><BR/>
</xsl:template>

<xsl:template match="performance">
  <B>performance[性能]:</B><xsl:apply-templates/>
</xsl:template>

<xsl:template match="performance_identifier">
  <B><xsl:value-of select="@identifier"/>:</B> <xsl:apply-templates select="text()"/>
</xsl:template>
<xsl:template match="expression">
 <BR/><xsl:apply-templates/><BR/>
</xsl:template>

<xsl:template match="play"><br/>
 <B>play[プレイ]:</B>
  <a><xsl:attribute name="href">JavaScript:miniW()</xsl:attribute>
                                                    <xsl:apply-templates/></a><BR/>
 <script language="JavaScript">
  <xsl:comment>
   function miniW()
              {window.open("<xsl:value-of select="@url"/>","","status=0,menubar=0,,")}
  </xsl:comment>
 </script>
</xsl:template>

<xsl:template match="faq">
```

```
 <B>faq:</B>
 <a><xsl:attribute name="href"><xsl:value-of select="@url"/></xsl:attribute>
                                                    <xsl:apply-templates/></a><BR/><BR/>
</xsl:template>


</xsl:stylesheet>
```