**JTS拓扑套件技术规范**

**Version 1.4**



**Document Change Control**

|  |  |  |  |
| --- | --- | --- | --- |
| **REVISION NUMBER** | **DATE OF ISSUE** | **AUTHOR(S)** | **BRIEF DESCRIPTION OF CHANGE** |
| 1.3 | March 31, 2003 | M. Davis | Updated to cover changes in JTS 1.3 |
| 1.4 | October 17, 2003 | J. Aquino | CoordinateSequences and the user-data field |
| 1.4.1 |  | M Davis | * Fixed definition of contains |

**Table of Contents**

1. OVERVIEW 7
2. OTHER RESOURCES 7
3. DESIGN GOALS 7
4. [TERMINOLOGY 8](#_TOC_250093)
5. [NOTATION 8](#_TOC_250092)
6. [JAVA IMPLEMENTATION 8](#_TOC_250091)
7. [COMPUTATIONAL GEOMETRY ISSUES 9](#_TOC_250090)
   1. [PRECISION MODEL 9](#_TOC_250089)
      1. [Fixed Precision 9](#_TOC_250088)
      2. [Floating Precision 10](#_TOC_250087)
   2. [CONSTRUCTED POINTS AND DIMENSIONAL COLLAPSE 10](#_TOC_250086)
   3. [ROBUSTNESS 11](#_TOC_250085)
   4. [NUMERICAL STABILITY 11](#_TOC_250084)
   5. [COMPUTATIONAL PERFORMANCE 12](#_TOC_250083)
      1. [Monotone Chains 12](#_TOC_250082)
8. [SPATIAL MODEL 13](#_TOC_250081)
   1. [DESIGN DECISIONS FOR SPATIAL MODELS 13](#_TOC_250080)
   2. [GEOMETRIC DEFINITIONS 14](#_TOC_250079)
      1. [Geometry 15](#_TOC_250078)
      2. [Empty Geometry 15](#_TOC_250077)
      3. [GeometryCollection 15](#_TOC_250076)
      4. [Curve 15](#_TOC_250075)
      5. [MultiCurve 15](#_TOC_250074)
      6. [LineString 16](#_TOC_250073)
      7. [LinearRing 16](#_TOC_250072)
      8. [Polygon 16](#_TOC_250071)
      9. [MultiPolygon 18](#_TOC_250070)
   3. [SIMPLE FEATURE CLASSES 18](#_TOC_250069)
      1. [Geometry 18](#_TOC_250068)
      2. [GeometryCollection 18](#_TOC_250067)
      3. [Point 18](#_TOC_250066)
      4. [MultiPoint 18](#_TOC_250065)
      5. [Curve 18](#_TOC_250064)
      6. [LineString 18](#_TOC_250063)
      7. [Line 18](#_TOC_250062)
      8. [LinearRing 18](#_TOC_250061)
      9. [MultiCurve 18](#_TOC_250060)
      10. [MultiLineString 18](#_TOC_250059)
      11. [Surface 18](#_TOC_250058)
      12. [Polygon 18](#_TOC_250057)
      13. [MultiSurface 19](#_TOC_250056)
      14. [MultiPolygon 19](#_TOC_250055)
   4. [NORMAL FORM FOR GEOMETRY 19](#_TOC_250054)
   5. [SUPPORT CLASSES 19](#_TOC_250053)
      1. [Coordinate 19](#_TOC_250052)
      2. [CoordinateSequence 20](#_TOC_250051)
      3. [Envelope 20](#_TOC_250050)
      4. [IntersectionMatrix 20](#_TOC_250049)
      5. [GeometryFactory 20](#_TOC_250048)
      6. [CoordinateFilter 20](#_TOC_250047)
      7. [GeometryFilter 20](#_TOC_250046)
   6. [SPATIAL REFERENCE SYSTEM 20](#_TOC_250045)
9. [BASIC GEOMETRIC ALGORITHMS AND STRUCTURES 21](#_TOC_250044)
   1. [POINT-LINE ORIENTATION TEST 21](#_TOC_250043)
   2. [LINE INTERSECTION TEST 21](#_TOC_250042)
   3. [LINE INTERSECTION COMPUTATION 21](#_TOC_250041)
   4. [POINT-IN-RING TEST 22](#_TOC_250040)
   5. [RING ORIENTATION TEST 22](#_TOC_250039)
10. [TOPOLOGICAL COMPUTATION 22](#_TOC_250038)
    1. [TOPOLOGY GRAPHS 22](#_TOC_250037)
    2. [LABELS 22](#_TOC_250036)
    3. [COMPUTING THE INTERSECTION MATRIX FROM A LABELING 23](#_TOC_250035)
    4. [THE RELATE ALGORITHM 24](#_TOC_250034)
       1. [Labeling isolated components 24](#_TOC_250033)
    5. [THE OVERLAY ALGORITHM 24](#_TOC_250032)
11. [BINARY PREDICATES 25](#_TOC_250031)
    1. [GENERAL DISCUSSION 25](#_TOC_250030)
    2. [METHOD SPECIFICATIONS 26](#_TOC_250029)
       1. [Equals 26](#_TOC_250028)
       2. [Disjoint 26](#_TOC_250027)
       3. [Intersects 26](#_TOC_250026)
       4. [Touches 26](#_TOC_250025)
       5. [Crosses 27](#_TOC_250024)
       6. [Within 27](#_TOC_250023)
       7. [Contains 27](#_TOC_250022)
       8. [Overlaps 27](#_TOC_250021)
12. [SPATIAL ANALYSIS METHODS 27](#_TOC_250020)
    1. [GENERAL DISCUSSION 27](#_TOC_250019)

[1.1.1 Representation of Computed Geometries 27](#_TOC_250018)

* 1. [CONSTRUCTIVE METHODS 28](#_TOC_250017)
  2. [SET-THEORETIC METHODS 28](#_TOC_250016)
  3. [METHOD SPECIFICATIONS 29](#_TOC_250015)
     1. [Buffer 29](#_TOC_250014)
     2. [ConvexHull 31](#_TOC_250013)
     3. [Intersection 31](#_TOC_250012)
     4. [Union 31](#_TOC_250011)
     5. [Difference 31](#_TOC_250010)
     6. [SymDifference 31](#_TOC_250009)

1. [OTHER METHODS 31](#_TOC_250008)
   * 1. [Boundary 31](#_TOC_250007)
     2. [IsClosed 32](#_TOC_250006)
     3. [IsSimple 32](#_TOC_250005)
     4. [IsValid 33](#_TOC_250004)
2. [WELL-KNOWN TEXT INPUT/OUTPUT 33](#_TOC_250003)
   1. WELL-KNOWN TEXT SYNTAX 34
   2. [WELL-KNOWN TEXT READER 35](#_TOC_250002)
   3. [WELL-KNOWN TEXT WRITER 35](#_TOC_250001)
3. [REFERENCES 36](#_TOC_250000)

**1. 概述**

**名词:**

**概貌 (overview, general picture)**

**概观 (conspectus, general view, overview, survey)**

JTS拓扑套件是一种Java API，它使用显式精度模型和强大的几何算法实现一组核心空间数据操作。 JTS旨在用于开发支持空间数据集的验证，清理，集成和查询的应用程序。本文档是JTS拓扑套件中实现的类，方法和算法的设计规范。

JTS尝试尽可能准确地实现OpenGIS简单特征规范（SFS）。在某些情况下，SFS不清楚或省略规范;在这种情况下，

JTS会尝试选择合理且一致的替代方案。本规范中记录了SFS的差异和详细说明。

类层次结构和方法的详细文档将以JavaDoc的形式呈现给源代码。

**2. 其他资源**

* *SQL版本1.1的OpenGIS简单特性规范（在本文档中称为SFS）。本文档提供了空间数据模型的主要规范以及JTS实现的空间谓词和功能的定义。*

**3. 设计目标**

JTS的设计旨在实现以下目标：

* 空间模型和方法定义将尽可能准确地符合OpenGIS简单要素规范，与正确的实现一致。
* API设计将尽可能遵循Java惯例。例如：
  + accessor函数将使用Java getX和setX约定
  + 谓词将使用isX约定
  + 方法将以小写字母开头
* JTS功能将支持用户定义的精度模型。在该精度模型下，JTS算法将是稳健的。
* 方法将尽可能在定义的精度模型中返回拓扑和几何上正确的结果。
* 正确性是最高优先级;空间和时间效率很重要但次要。
* JTS足够快，可以在生产环境中使用。
* JTS中使用的算法和代码将清晰且结构良好，以便于其他开发人员的理解。

# 4.术语

|  |  |
| --- | --- |
| **术语** | **定义** |
| **Coordinate** | 空间中的一个点，在定义的精度模型下可以精确表示 |
| **Exact Computation** | 通过所有操作维护数字的所有数字的数值计算。通常需要算法昂贵的算法 |
| **Node** | 相同或不同几何形状中的两条线相交的点。该点不一定由坐标表示，因为交点计算的输出通常需要比输入点更高的精度。 |
| **Noding (also Noded)** | 计算一个或多个几何相交的节点的过程。 |
| **Non-coordinate** | 一个不能表示为坐标的点 |
| **Numerical Stability** | 数值算法的稳定性由其输出中的误差的最大界限确定。如果此边界很小，则认为算法是稳定的。 |
| **Point** | R3中的任意点。一般而言，不是有限可表示的。 |
| **Proper intersection** | 两个线段之间的交点，其中交点是单个点并且在两个线段的内部 |
| **Robust Computation** | 数值计算保证返回所有输入的正确答案。通常需要专门设计用于处理舍入误差的算法。 |
| **SFS** | OGC Simple Features Specification |
| **Unit of Resolution** | 定义精度模型下的最小可表示距离。 |
| **Vertex (pl. vertices)** | 几何对象的“顶点”。这些是显式存储以定位几何对象的坐标。 |

# 5.注释

* 符合SFS的规范中的项目通过参考括号中的SFS中的相关部分来指示：（SFS 1.0）
* 规范中详细说明或与SFS不同的项目将在括号中用“JTS”表示：（JTS）

# 6. JAVA实现

Java编码风格在某些情况下与SFS中使用的编码风格不同。两者在一般情况下不同，JTS遵循Java惯例。 JTS编码风格在以下方面与SFS编码风格不同：

* SFS有时使用Integer来表示布尔值。在这种情况下，JTS将使用布尔值
* SFS中的方法名称以大写字母开头。在JTS中，所有方法名称都以小写字母开头
* JTS中的方法名称有时会添加前缀“get”或“set”，以符合Java Bean的约定。

# 7.计算几何问题

* 1. **精度模型**

所有数值计算都在某种形式的精度模型下进行。有几种可能的精度模型：

|  |  |
| --- | --- |
| **Fixed** | 坐标表示为具有均匀间距的网格上的点。计算坐标将四舍五入到此网格。 |
| **Floating** | 坐标表示为浮点数。计算坐标可能具有比输入值更多的精度位数（达到有限浮点表示所允许的最大值）。 |
| **Exact** | 坐标完全表示（通常为具有整数分子和分母的有理数）。实施该模型会对空间和时间性能造成损害，这通常被认为是不可接受的。 |

通常，计算的精确模型没有明确说明，但是由用于表示值的模型（例如浮点或整数）暗示。这种方法的局限性在于用户无法在精度较低的精度模型中工作。通常情况下，计算结果的精度高于输入。对于进一步的计算或以具有原始（或更低）精度的格式存储，较高精度值可能是不可接受的。

JTS通过允许用户指定显式精度模型来处理此问题。精度模型允许客户端在输入坐标值中声明要假设的精度位数，并在任何计算坐标中保持。

在JTS方法中，输入几何可能具有不同的精度模型。在返回Geometrys的方法的情况下，返回结果的精度模型是两个输入精度模型（即具有最大精度的模型）的最大值。请注意，这仅适用于两个精度模型兼容的情况。如果比例因子1是另一个的比例因子的整数倍，则两个精度模型是兼容的。没有尝试协调不兼容的精度模型。

JTS支持两种基本类型的精度模型：固定和浮动。

* + 1. 固定精度

在固定精度模型中，假设坐标精确地落在离散网格的交叉点上。网格的大小由比例因子确定。网格大小是比例因子的倒数。比例因子也可以被认为是确定保持多少小数精度。比例因子可以大于或小于1，这取决于“精确点”是小数点的右侧还是左侧。

根据以下等式精确调整坐标：

jtsPt.x = round( inputPt.x \* scale ) / scale jtsPt.y = round( inputPt.y \* scale ) / scale

精确坐标将在内部表示为双精度值。这被称为“精确的内部表示”。 由于Java使用IEEE-754浮点标准，因此提供53位精度。 （因此，最大可精确表示的值是9,007,199,254,740,992）。

输入程序负责在创建JTS结构之前将坐标舍入到精度模型。 （JTS提供的输入例程将自动执行此舍入。）

* + 1. 浮动精度

支持浮点精度模型有两种类型，双精度和单精度。这两者都基于Java浮点模型，而两者又基于IEEE-754浮点标准。这为双精度提供了大约16位精度，为单精度提供了6位精度。

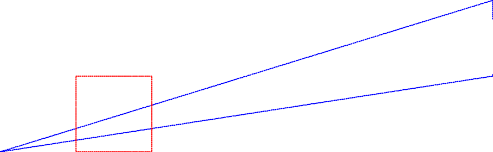
在浮动双精度模型中，坐标可以具有Java双精度浮点数的完整精度。假设输入坐标不被舍入，并且计算构造点的内部操作不会使计算的坐标四舍五入。请注意，这并不意味着构造的点是准确的;它们仍局限于双精度数的精度，因此可能仍然只是精确点的近似值。

在浮动单精度模型中，计算的坐标舍入为单精度。这支持计算几何的最终目的是单精度格式（例如Java2D）的情况。

## 构造点和空间收缩

通过空间分析方法计算的几何图形可以包含输入几何图形中不存在的构造点。这些新点来自输入几何边缘中线段之间的交叉点。在一般情况下，不可能精确地表示构造点。这是因为交点的坐标可能包含与输入线段坐标一样多的精度位。为了明确地表示这些构造的点，JTS必须对它们进行舍入以适合给定的精度模型。

不幸的是，舍入坐标会略微移动它们。在精确结果中不重合的线段可能在截断的表示中变得重合。对于线 - 线组合，这可以生成包含不在输入几何内部的点的结果几何。更严重的是，对于线区域组合，这可能导致维度崩溃，这是计算组件的维度低于精确结果的维度。



**A**

A.difference(B)

**B**

1 unit

**图1 – 空间收缩的例子**

JTS可以使几何图形尺度变低的部分重新形成，所以可以很轻易地解决空间紧缩的问题。例如，一个空间紧缩的面面相交之处可以产生一个线性或点的几何图形，并以此来组构成原始的结果

## 鲁棒性

几何算法包括了组合计算算法和数值计算的结合。所有的数值计算都是用到有限精度数字的，因此所选用的

算法容易受健壮性问题的影响。数值计算由于舍入误差产生错误结果的时候，健壮性问题就产生了。

健壮性问题在几何计算中更为严重，因为数值误差会发展成组合计算，从而导致算法产生错误的结果。

（参见[Bri98], [Sch91]）。

有许多方法可以解决几何计算中的鲁棒性问题。毫不奇怪，大多数强大的算法比非鲁棒版本更复杂且性能更差。 JTS试图以两种方式处理鲁棒性问题：

* 使用稳健算法实现了重要的基本几何算法（例如线方向，线交点和多边形点测试）。特别是，几种算法的实现依赖于[Ava97]中提出的稳健的行列式计值.
* 开发了用于实现SFS谓词和函数的算法，以消除或最小化鲁棒性问题。二元谓词算法是完全健壮的。空间覆盖和缓冲算法是非稳健的，但在大多数情况下将返回正确的答案。

## 数值稳定性

数值算法的一个优点就是稳定性。数值算法的稳定性是由输出结果的最大误差的范围决定的。如果这个范围很小，那么算法就可被认为是稳定的。。

JTS中使用的主要数值算法是计算两个线段之间的交叉点。该算法本质上是不精确的，因为表示交叉点所需的精度位是输入精度的几倍。用于该计算的稳定算法将始终产生接近确切答案的近似答案。特别是，计算的点应该至少位于输入线段的边界框内！理想情况下，计算的点将位于精确答案的单个精度模型网格单元内。

提高数值算法稳定性的一种方法是调整其输入。调节输入涉及以某种方式对它们进行数字操作，以产生相同的答案，同时在计算期间保持更高的精度。 JTS使用将输入线段“标准化”到线交叉计算的技术。归一化的线段已被转换为尽可能接近原点。这具有从每个纵坐标移除共同有效数字的效果，并因此增加可用的精度位以维持线交叉计算的准确性。

* 1. **计算性能**

运行时性能是几何算法的生产质量实现的重要考虑因素。 JTS中使用的计算密集度最高的算法是交叉检测。许多JTS方法需要确定单个几何（自相交）中的线段与两个不同几何的线段之间的所有交叉点的所有交叉。

交叉检测的明显算法，即将每个段彼此进行比较，具有令人无法接受的缓慢性能。有大量关于交叉检测的有效算法的文献。不幸的是，其中许多涉及大量的代码复杂性。 JTS尝试平衡代码简单性和性能提升。它使用一些特殊技术为常见类型的输入数据产生显着的性能提升。这些技术包括各种类型的内存空间索引，以及用于构造数据的复杂方法，例如Monotone Chains技术。

* + 1. 单调链

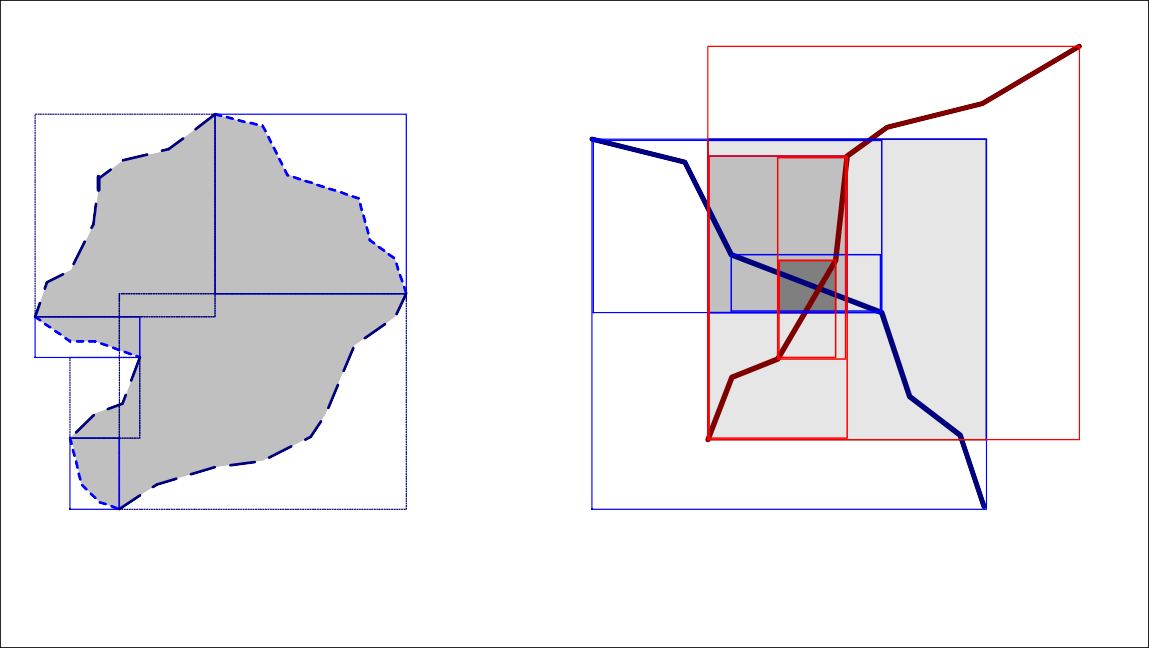
JTS使用“Monotone Chains”技术以最小的额外代码复杂性获得实质性的性能改进。该技术涉及将边缘分成单调的链段。单调链由一系列段组成，其方向向量都位于同一象限中。单调链具有两个重要特性：

**不相交性:** 单调链内的片段不相交。

**端点包络属性**: 单调链中任何连续子集的包络同时也是该子集端点的包络。

非交叉属性意味着不需要在相同的单调链内测试段对以进行交叉。端点包络属性允许使用二进制搜索来查找单调链上的交叉点。此外，相对于各个区段的单调链的较大边界框充当区段的“聚类”形式，这减少了所需的交叉测试的总数。

对于具有显着百分比单调链的数据，这些属性消除了大量的段比较。单调链在通过沿自然特征流数字化生成的数据中是常见的。已经观察到比天真算法高出100倍的性能改进。

**(1)**

The Monotone Chains for a Polygon

**(1)**

The bisection process used to find an intersection between two monotone chains

**Figure 2 - Monotone Chains**

# 8.空间模型

* 1. **空间模型的设计决策**

SFS只是现有空间数据库和API中使用的几种空间模型之一。这些模型在很大程度上非常相似。通常，它们都支持表示二维点，线和多边形。然而，Geometrys的表示方式之间存在一些细微差别。这些差异代表了空间API设计者的设计决策。下面列出了一些重要的设计选择（在每种情况下，都指出了在SFS和JTS中做出的选择）。

|  |  |
| --- | --- |
| ***Design Decision*** | **几何中允许重复点** |
| ***SFS Choice*** | 允许重复点数 |
| ***JTS Choice*** | 与SFS相同 |
| ***Comments*** | 通常，空间算法不能容忍重复点。允许重复点会导致性能和空间损失，因为每个空间方法都必须检查重复点并将其删除。 JTS确实支持重复点，因为不这样做是与OGC模型不兼容的一个主要问题。但是，这样做的内存和性能成本很低。 |

|  |  |
| --- | --- |
| ***Design Decision*** | **线串允许自相交（即可以是非简单的）** |
| ***SFS Choice*** | 允许线串自相交 |
| ***JTS Choice*** | 与SFS相同 |
| ***Comments*** | 允许非简单的线串确实会产生很小的性能损失，因为这意味着在用于空间方法之前必须对线串进行节点化。但是，希望能够表示非简单的线串，因此如果LineString类本身被定义为简单，则必须引入另一个类来表示非简单的线（有时称为“Spaghetti”）。 |

|  |  |
| --- | --- |
| ***Design Decision*** | **在独立点上多边形环可以自行连接** |
| ***SFS Choice*** | 在独立点上多边形环不可以自行连接 |
| ***JTS Choice*** | 与SFS相同 |
| ***Comments*** | 这种判定出现在需要支持表现多边形连接口，这些连接口联系独立点的位置（“反向多边形”）。它也出现在表现单个连接口，这种连接口包含一个不相关的外界区域（“顺向连接口”）。为了表现反向多边形和顺向连接口，多边形环必须允许在单个点上自行连接或者在多个独立点上互相连接。  这种设计决定是多边形环是否可以在单个点处相互接触的选择的双重意义  不幸的是，选择多边形环不能自连接会导致稍微复杂的算法，因为通常的多边形构建算法会导致自我连接的外壳。有必要执行进一步的步骤以将通过自行连接隔离的区域的边界转换为孔。 |

|  |  |
| --- | --- |
| ***Design Decision*** | 多边形环能够在独立点上互相连接 |
| ***SFS Choice*** | 多边形环能够在独立点上互相连接 |
| ***JTS Choice*** | 同SFS |
| ***Comments*** | 这种判定是对对多边形环能否在独立点上自行连接的双重判定。 |

在大多数情况下，这些设计选择对API的用户没有影响，因为它们不会更改可以表示的Geometrys集。但是，它们确实会影响API中实现的算法的性能和复杂性。此外，在两个已经做出不同设计选择的API的表示之间进行转换通常是非常简单的（特别是，如果两个API对于多边形环是否可以自行连接做出不同的选择，则需要一些相对复杂的处理来转换多边形表示）。

* 1. **几何定义**

根据SFS中给出的定义，所有JTS方法都假定它们的参数是有效的Geometric对象。

以下定义阐述或阐明了SFS中给出的定义。

* + 1. 几何(Geometry)

一个精度模型(Precision Model)对象将是每个Geometry对象的成员。

根据SFS几何对象，只能表示闭合集。这是一个合理的决定，允许实际实施。但是，空间分析方法的语义有一些含义（参见第12节空间分析方法）。

JTS有一个简单的方案，用于向Geometry添加属性：应用程序可以将Geometry的用户数据字段设置为任何对象。

* + 1. 空几何(Empty Geometry)

SFS指定每个Geometry子类的对象可以为空。有时需要构造Geometry类的通用空对象（例如，如果要返回的几何的确切类型未知）。 SFS不会为了表现这个一般空Geometry而去定义一个特殊的类或对象。JTS使用的约定就是返回一个空的几何集合（GeometryCollection）。

* + 1. 几何集合(GeometryCollection)

不同的GeometryCollection的维度是其元素的最大维度。

* + 1. 曲线(Curve)

曲线可能不会退化。也就是说，非空曲线必须至少有2个点，并且没有两个连续点可能相等。

* + 1. 多曲线(MultiCurve)

SFS指定使用“Mod-2”规则来确定MultiCurve的边界。如果MultiCurve位于MultiCurve的奇数个元素的边界上，则该点位于MultiCurve的边界上。应当注意，这导致SFS边界中的点集大于直觉或点集拓扑所指示的情况。也就是说，根据SFS规则，入射在其上的奇数> 1的边的点在边界上，但可能不直观地被视为边界的一部分。这也与边界的拓扑定义不一致，边界是“几何中的一组点的任何开放子集中不包含的点集”。

例如，在图3（3）中，点B根据SFS位于边界内，但是根据点集拓扑结构是内点。

Boundary = { A, C, D, E }

Boundary = { A, B, C, D }

**(4)**

**(3)**

**(2)**

Boundary = { A, C }

**(1)**

Boundary = { A, B }

**D**

**D**

**B**

**B**

**B**

**B**

**E**

**C**

**A**

**C**

**A**

**C**

**A**

**A**

**Figure 3 - Effect of the Mod-2 rule in MultiLineStrings**

JTS中需要额外的逻辑来实现Mod-2规则。

* + 1. 线段(LineString)

我们正在使用OGC SFS中给出的LineString的定义。这与其他一些空间模型（例如ESRI ArcSDE使用的模型）的重要方式不同。不同之处在于LineStrings可能不简单。它们可以在点或线段中自相交。

实际上，曲线的边界点（例如，端点）可以与曲线的内部相交，从而产生在技术上拓扑闭合但根据SFS不闭合的曲线。在这种情况下，拓扑上的交点不会在曲线的边界上。但是，根据SFS的定义，该点被认为是在边界上。 JTS遵循SFS定义。



**A**

*B is a boundary point, not an interior point*

**B**

LineString: Boundary = { A, B }

**Figure 4 - A LineString with a boundary point intersecting an interior point**

* + 1. 线段环(LinearRing)

LinearRings是多边形的基本构建块。 LinearRings可能不会退化;也就是说，LinearRing必须至少有3个点。 LinearRings简单的要求暗示了其他非简并标准。例如，并非所有点都可以共线，并且环可能不会自相交。 SFS未指定对LinearRing方向的要求。 JTS遵循这一点，允许LinearRings顺时针或逆时针定向。

* + 1. 多边形(Polygon)

Polygon的外壳和孔是LinearRings。 Polygon的SFS定义具有以下含义：

* 壳和孔不能自相交（这是因为它们是LinearRings隐含这个特征）
* 孔只能在一个点接触外壳或另一个孔。这意味着孔不能在多个点或线段中彼此相交。
* 多边形内部必须连接（这是由先前的声明中暗示）.
* 不要求孔连接外壳的点是顶点

需要注意的是多边形的SFS定义不同于在其他一些常用的空间模型。例如，ESRI ArcSDE空间模型允许外壳自行相交于顶点，但不允许孔碰壳。他们描述完全相同的一组区域的SFS和ArcSDE的模型，在这个意义等同。然而，他们可能需要不同的多边形结构来描述同一地区。

*This hole touches the shell at a vertex*

*This hole touches the shell at a non-vertex*

A Polygon with 4 holes

**Figure 5 - An example of a Polygon containing holes**

**(7)**

Shell self-intersects

**(6)**

Holes touch in line segment

**(5)**

Holes cross

**(4)**

The polygon interior is disconnected

**(3)**

Hole touches shell in line segment

**(2)**

Hole touches shell at more than one point

**(1)**

Hole crosses shell

**Figure 6 - Examples of objects not representable as polygons**

Empty Polygons may not contain holes.

Since the shell and holes of Polygons are LinearRings, there is no requirement on their orientation. They may be oriented either clockwise or counterclockwise.

* + 1. MultiPolygon

MultiPolygon中的元素多边形可能仅触摸有限数量的点（例如，它们可能不会触摸线段）。元素的内部必须是不相交的（例如它们可能不交叉）。不要求交点是顶点。

* 1. **SIMPLE FEATURE CLASSES**

All Geometry classes allow empty objects to be created, and support the isEmpty method. Empty Geometries will be represented by their internal arrays having zero length.

All Geometry classes support the equalsExact() method, which returns true if two Geometry subclasses are equivalent and have identical sequence(s) of coordinates. Two objects are “equivalent” if their classes are identical. The only exception is LinearRing and LineString, which JTS considers to be equivalent.

All Geometry classes support the clone() method, which will return a deep copy of the object.

* + 1. Geometry

Geometry is non-instantiable and is implemented as an abstract class.

* + 1. GeometryCollection

A GeometryCollection is implemented as an array of Geometry objects.

* + 1. Point

A Point is implemented as a single Coordinate.

* + 1. MultiPoint

A MultiPoint inherits the implementation of GeometryCollection, but contains only Points.

* + 1. Curve

Curve is non-instantiable and is implemented as an interface.

* + 1. LineString

A LineString is implemented as an array of coordinates.

* + 1. Line

JTS does not implement the Line class, since LineString offers equivalent functionality.

* + 1. LinearRing

A LinearRing containing n coordinates is implemented with an array of Coordinates containing n+1 points, and coord[0] = coord[n].

* + 1. MultiCurve

MultiCurve is non-instantiable and is implemented as an interface.

* + 1. MultiLineString

A MultiLineString inherits the implementation of GeometryCollection, but contains only LineStrings.

* + 1. Surface

Surface is non-instantiable and is implemented as an interface.

* + 1. Polygon

A Polygon is implemented as a single LinearRing for the outer shell, and an array of LinearRings for the holes. The outer shell is oriented CW and the holes are oriented CCW.

* + 1. MultiSurface

MultiSurface is non-instantiable and is implemented as an interface.

* + 1. MultiPolygon

A MultiPolygon inherits the implementation of GeometryCollection, but contains only Polygons.

* 1. **NORMAL FORM FOR GEOMETRY**

JTS defines a normal (or canonical) form for representing Geometrys. Normal form is a unique representation for Geometrys. It can be used to test whether two Geometries are equal in a way that is independent of the ordering of the coordinates within them. Normal form equality is a stronger condition than topological equality, but weaker than pointwise equality.

The definitions for normal form use the standard lexicographical ordering for coordinates. “Sorted in order of coordinates” means the obvious extension of this ordering to sequences of coordinates.

|  |  |
| --- | --- |
| **Geometry Class** | **Definition of normal form** |
| Point | Points are always in normal form |
| MultiPoint | Element Points are sorted in order of their coordinates |
| LineString | Obeys the following condition:  If there is an i such that coord[i] != coord [n – i – 1] then coord [i] < coord [n – i –1] |
| LinearRing | same as LineString |
| MultiLineString | Element LineStrings are in normal form, and are sorted in order of their coordinates |
| Polygon | The LinearRings of the Polygon are ordered such that the smallest point is first. The shell is ordered clockwise, and  holes are ordered counterclockwise. Holes are sorted in order of their coordinates |
| MultiPolygon | Element Polygons are in normal form, and are sorted in order of their coordinates |
| GeometryCollection | Element Geometrys are in normal form.  The list of elements is ordered by class (using the order of this list). Within each subsequence of like class, elements are sorted in order of coordinates. |

* 1. **SUPPORT CLASSES**
     1. Coordinate

Coordinate is the lightweight class used to store coordinates. It is distinct from Point, which is a subclass of Geometry. Unlike objects of type Point (which contain additional information such as an envelope, a precision model, and spatial reference system information), a Coordinate only contains ordinate values and accessor methods.

Coordinates are two-dimensional points, with an additional z-coordinate. JTS does not support any operations on the z-coordinate except the basic accessor functions. Constructed coordinates will have a z-coordinate of NaN.

Coordinate implements the standard Java interface Comparable. The implementation uses the usual lexicographic comparison. That is,

*c1.compareTo(c2) =*

*-1 : c1.x < c2.x Ú ((c1.x = c2.x) Ù (c1.y < c2.y)) 0 : (c1.x = c2.x) Ù (c1.y = c2.y)*

*1 : c1.x > c2.x Ú ((c1.x = c2.x) Ù (c1.y > c2.y))*

Coordinate implements equals() using the obvious implementation of pointwise comparison.

* + 1. CoordinateSequence

A CoordinateSequence is the internal representation of a list of Coordinates inside a Geometry. Because it is an interface, it is possible to create alternatives to the default implementation (an array of Coordinates). For example, one may choose to store the data as an array of some entirely different coordinate class, or as an array of x’s and an array of y’s. Note that non-Coordinate-array implementations will pay a performance penalty when the #toArray method is called.

* + 1. Envelope

A concrete class containing a maximum and minimum x and y value.

* + 1. IntersectionMatrix

An implementation of the Dimensionally Extended 9-Intersection Model (DE-9IM) matrix. The class can be used to represent both actual instances of a DE-9IM matrix as well as patterns for matching them. Methods are provided to:

* set and query the elements of the matrix in a convenient fashion
* convert to and from the standard string representation (specified in SFS Section 2.1.13.2).
* test to see if a matrix matches a given pattern string.
  + 1. GeometryFactory

A GeometryFactory supplies a set of utility methods for building Geometry objects from lists of Coordinates.

* + 1. CoordinateFilter

GeometryImpl classes support the concept of applying a coordinate filter to every coordinate in the Geometry. A coordinate filter can either record information about each coordinate or change the coordinate in some way. Coordinate filters implement the interface CoordinateFilter. (CoordinateFilter is an example of the Gang-of-Four Visitor pattern). Coordinate filters can be used to implement such things as coordinate transformations, centroid and envelope computation, and many other functions.

* + 1. GeometryFilter

GeometryImpl classes support the concept of applying a Geometry filter to the Geometry. In the case of GeometryCollection subclasses, the filter is applied to every element Geometry. A Geometry filter can either record information about the Geometry or change the Geometry in some way. Geometry filters implement the interface GeometryFilter. (GeometryFilter is an example of the Gang-of-Four Visitor pattern.)

* 1. **SPATIAL REFERENCE SYSTEM**

JTS will support Spatial Reference System information in the simple way defined in the SFS. A Spatial Reference System ID (SRID) will be present in each Geometry object. Geometry will provide basic accessor operations for this field, but no others. The SRID will be represented as an integer.

The SRID of constructed objects will be copied from the SRID of one of the input objects if possible, or will be 0.

# BASIC GEOMETRIC ALGORITHMS AND STRUCTURES

## POINT-LINE ORIENTATION TEST

This function is fundamental to operations such as ordering edges around a node. Since it is essentially a geometric calculation, it is susceptible to robustness problems unless implemented using robust algorithms. JTS implements this method using a robust algorithm which returns the correct result for all input values. The algorithm used is based on the robust method of evaluating signs of determinants developed by Avanim et. al. ([Ava97]).

*[diagram of point-line orientation]*

## LINE INTERSECTION TEST

This function tests whether two line segments intersect. It uses the robust Point-Line Orientation function specified above. It does not actually compute the point of intersection, and thus returns an exact answer. The function computes full information about the topology of the intersection, including the following data:

|  |  |
| --- | --- |
| HasIntersection() | True if the line segments intersect |
| getIntersectionNum() | The number of intersection points found (0, 1, or 2) |
| IsProper() | True if the intersection point is proper (i.e. is not equal to one of the endpoints) |

## LINE INTERSECTION COMPUTATION

This function computes the intersection of two line segments. Two line segments may intersect in a single point, a line segment, or not at all. If the intersection is representable with coordinates in the Precision Model, it will be computed exactly. Otherwise, an approximation will be computed.

Intersections which are line segments will always be representable with coordinates, since each endpoint of the intersection segment must be equal to an endpoint of one of the input segments. Obviously, null intersections can also be computed exactly (although the intersection test must be performed with robust code to be correct). Intersections which are points may or may not be representable, since in general computed intersections require greater precision than the input points, and will not necessarily fall exactly on the precision model grid.

An important property of the line intersection algorithm is that it is numerically stable. Computed approximate points should be within the Precision Model tolerance of the exact intersection point.

In addition to the information computed by the Line Intersection test, the Line Intersection Computation computes information about the actual points of intersection:

|  |  |
| --- | --- |
| GetIntersection(int i) | The coordinate for the I’th intersection point |

Determining the edge graph requires further information about the precise order of intersection points along each line segment. The Line Intersection class provides other functions to determine the order of intersection points along each segment, and to compute the (approximate) distance of a given intersection point along a segment.

## POINT-IN-RING TEST

The Point-In-Ring predicate is implemented in a robust fashion by using the usual stabbing- line algorithm and making use of the robust Line Intersection Test.

In some cases it is necessary to test for the inclusion of multiple points in a given ring (e.g. in the IsValid predicate to test for the correct inclusion of holes). In this case performance can be gained by using a spatial index for the line segments of the ring. JTS implements a 1-dimensional Interval Tree to speed up the intersection tests made in the stabbing-line algorithm.

## RING ORIENTATION TEST

This test returns true if a ring of coordinates is oriented in a clockwise direction. The test is used to determine on which side of the rings of the shell and holes of a Polygon the interior and exterior of the Polygon lie.

# TOPOLOGICAL COMPUTATION

## TOPOLOGY GRAPHS

The computation of the Intersection Matrix relies on the use of a structure called a “topology graph”. The topology graph contains nodes and edges corresponding to the nodes and line segments of a Geometry. Each node and edge in the graph is labeled with its topological location relative to the source geometry.

Note that there is no requirement that points of self-intersection be a vertex. Thus to obtain a correct topology graph, Geometries must be self-noded before constructing their graphs.

Two fundamental operations are supported by topology graphs:

* Computing the intersections between all the edges and nodes of a single graph
* Computing the intersections between the edges and nodes of two different graphs

## LABELS

Topology graphs support the concept of labeling nodes and edges in the graph. The label of a node or edge specifies its topological relationship to one or more geometries. (In fact, since JTS operations have only two arguments labels are required for only two geometries). A label for a node or edge has one or two elements, depending on whether the node or edge occurs in one or both of the input Geometries. Elements contain attributes which categorize the topological location of the node or edge relative to the parent Geometry; that is, whether the node or edge is in the interior, boundary or exterior of the Geometry.

Attributes have a value from the set *{Interior, Boundary, Exterior}*. In a node each

element has a single attribute <**On**>. For an edge each element has a triplet of attributes

##### <Left, On, Right>.

A: < **Left** = *Interior*, **On** = *Interior*, **Right** = *Interior* > B: < **Left** = *Exterior*, **On** = *Boundary*, **Right** = *Interior* >

B

A: < **Left** = *Exterior*, **On** = *Boundary*, **Right** = *Interior* > B: < **Left** = *Exterior*, **On** = *Exterior*, **Right** = *Exterior* >

A

If A and B are simple polygons and A contains B, the labels on their edges are:

**Example 1**

* 1. **COMPUTING THE INTERSECTION MATRIX FROM A LABELING**

The Intersection Matrix (IM) for an overlay graph is computed from the labeling of nodes and edges in the graph. To compute the IM, we sum the contributions to the IM of each node and edge whose label contains elements for *both* Geometries. The IM contribution for a node is *dim >= 0* for the IM entry corresponding to the topological location of the node in the parent Geometries. (For example, a node which is in the Interior of Geometry A and in the Boundary of Geometry B would have *label[0][On] = Interior* and *label[1][On] = Boundary*, and *IM(Interior, Boundary) = 0*.) The IM contribution for an edge is *dim >= 1* for the IM entry corresponding to the topological location of the edge itself in the parent Geometries, and *dim >= 2* for the entries corresponding to the topological locations of the areas on the left and right sides of the edge.

The algorithmic expression of these rules is:

**function** Node.computeIM(im : IntersectionMatrix)

**if** (label[0] != null and label[1] != null) **then**

im.setAtLeast(label[0][On], label[1][On], 0)

**end if end function**

**function** Edge.computeIM(im : IntersectionMatrix)

**if** (label[0] != null and label[1] != null) **then**

im.setAtLeast(label[0][On], label[1][On], 1)

im.setAtLeast(label[0][Left], label[1][Left], 2)

im.setAtLeast(label[0][Right], label[1][Right], 2)

**end if end function**

For each combination of Geometries there is a maximum possible IM value. For efficiency this maximum value can be tested after each IM summation and the computation terminated if the value is obtained.

It is always the case that *dim(Ext(A) n Ext(B)) = 2*.

**Example 2**

Using the labels in Example 1 we have

|  |  |  |  |
| --- | --- | --- | --- |
| The full IM is: | 2 | 1 | 2 |
|  | F | F | 1 |
|  | F | F | 2 |

* 1. **THE RELATE ALGORITHM**

for the labeling of the edge of A

IM(Boundary, Exterior) = 1 IM(Exterior, Exterior) = 2 IM(Interior, Exterior) = 2

for the labeling of the edge of B

IM(Interior, Boundary) = 1 IM(Interior, Exterior) = 2 IM(Interior, Interior) = 2

The relate algorithm computes the Intersection Matrix describing the relationship of two Geometries. The algorithm for computing relate uses the intersection operations supported by topology graphs. Although the relate result depends on the resultant graph formed by the computed intersections, there is no need to explicitly compute the entire graph. Instead the structure of the graph is computed locally at each intersection node.

The relate algorithm is robust, by virtue of the robustness of the underlying operations. It is not subject to dimensional collapse problems, since it avoids calculating intersection points which might not lie on precise coordinates.

The algorithm to compute relate has the following steps:

1. Build topology graphs of the two input geometries. For each geometry all self- intersection nodes are computed and added to the graph.
2. Compute nodes for all intersections between edges and nodes of the graphs.
3. Compute the labeling for the computed nodes by merging the labels from the input graphs.
4. Compute the labeling for isolated components of the graph (see below)
5. Compute the Intersection Matrix from the labels on the nodes and edges.
   * 1. Labeling isolated components

Isolated components are components (edges or nodes) of an input Geometry which do not contain any intersections with the other input Geometry. The topological relationship of these components to the other input Geometry must be computed in order to determine the complete labeling of the component. This can be done by testing whether the component lies in the interior or exterior of the other Geometry. If the other Geometry is 1- dimensional, the isolated component must lie in the exterior (since otherwise it would have an intersection with an edge of the Geometry). If the other Geometry is 2-dimensional, a Point-In-Polygon test can be used to determine whether the isolated component is in the interior or exterior.

* 1. **THE OVERLAY ALGORITHM**

The Overlay Algorithm is used in spatial analysis methods for computing set-theoretic operations (boolean combinations) of input Geometries. The algorithm for computing the overlay uses the intersection operations supported by topology graphs. To compute an

overlay it is necessary to explicitly compute the resultant graph formed by the computed intersections.

The algorithm to compute a set-theoretic spatial analysis method has the following steps:

1. Build topology graphs of the two input geometries. For each geometry all self- intersection nodes are computed and added to the graph.
2. Compute nodes for all intersections between edges and nodes of the graphs.
3. Compute the labeling for the computed nodes by merging the labels from the input graphs.
4. Compute new edges between the compute intersection nodes. Label the edges appropriately
5. Build the resultant graph from the new nodes and edges.
6. Compute the labeling for isolated components of the graph. Add the isolated components to the resultant graph.
7. Compute the result of the boolean combination by selecting the node and edges with the appropriate labels. Polygonize areas and sew linear geometries together.

# BINARY PREDICATES

* 1. **GENERAL DISCUSSION**

The binary predicates can be completely specified in terms of an Intersection Matrix pattern. In fact, their implementation is simply a call to relate with the appropriate pattern.

It is important to note that binary predicates are topological operations rather than pointwise operations. Even for apparently straightforward predicates such as Equals it is easy to find cases where a pointwise comparison does not produce the same result as a topological comparison. (for instance: A and B are MultiPoints with the same point repeated different numbers of times; A is a LineString with two collinear line segments and B is a single line segment with the same start and endpoints; A and B are rings with identical sets of points but which start at different points). The algorithm used for the relate method is a topology-based algorithm which produces a topologically correct result.

**(2)**

LINESTRING ( 20 20, 15 15, 10 10 )

**(1)**

LINESTRING ( 10 10, 20 20 )

**Figure 7 - Two Geometries that are pointwise unequal but topologically equal**

As in the SFS, the term P is used to refer to 0-dimensional Geometries (Point and MultiPoint), L to 1-dimensional Geometries (LineString, and MultiLineString), and A to 2-

dimensional Geometries (Polygon and MultiPolygon). The dimension of a GeometryCollection is equal to the maximum dimension of its components.

In the SFS some binary predicates are stated to be undefined for some combinations of dimensions (e.g. touches is undefined for P/P). In the interests of simplifying the API, combinations of argument Geometries which are not in the domain of a predicate will return false (e.g. touches(Point, Point) => false).

If either argument to a predicate is an empty Geometry the predicate will return false.

Because it is not clear at this time what semantics for spatial analysis methods involving GeometryCollections would be useful, GeometryCollections are not supported as arguments to binary predicates or the relate method.

* 1. **METHOD SPECIFICATIONS**

Binary predicates are implemented as calls to relate, with the appropriate pattern supplied for the input Geometries. The specifications for most of the binary predicates are well described in the SFS, and are here simply specified by their relate pattern(s). Equals is not described in the SFS, however, so it is specified symbolically as well.

* + 1. Equals

The Equals relation applies to all combinations of Geometries. Two Geometries are topologically equal iff their interiors intersect and no part of the interior or boundary of one Geometry intersects the exterior of the other. Symbolically,

#### a.equals(b) Ü I(a) » I(b) µ ¸ Ù (I(a) ¼ B(a)) » E(b) = ¸ Ù (I(b) ¼ B(b)) » E(a) = ¸ Ü a.relate(b, “T\*F\*\*FFF\*”)

Equals() is a topological relationship, and does not imply that the Geometries have the same points or even that they are of the same class. (This more restrictive form of equality is implemented in the equalsExact() method.)

|  |  |
| --- | --- |
| ***Argument Dimensions*** | ***Relate Pattern*** |
| all | T\*F\*\*FFF\* |

* + 1. Disjoint

|  |  |
| --- | --- |
| ***Argument Dimensions*** | ***Relate Pattern*** |
| all | FF\*FF\*\*\*\* |

* + 1. Intersects

A.intersects(B) = ! A.disjoint(B)

* + 1. Touches

|  |  |
| --- | --- |
| ***Argument Dimensions*** | ***Relate Pattern*** |
| P/L, P/A, L/L, L/A, A/A | FT\*\*\*\*\*\*\*  or F\*\*T\*\*\*\*\* or F\*\*\*T\*\*\*\* |
| P/P | undefined |

* + 1. Crosses

|  |  |
| --- | --- |
| ***Argument Dimensions*** | ***Relate Pattern*** |
| P/L, P/A, L/A | T\*T\*\*\*\*\*\* |
| L/L | 0\*\*\*\*\*\*\*\* |
| P/P, A/A | undefined |

* + 1. Within

|  |  |
| --- | --- |
| ***Argument Dimensions*** | ***Relate Pattern*** |
| all | T\*F\*\*F\*\*\* |

* + 1. Contains

A.contains(B) = B.within(A)

* + 1. Overlaps

|  |  |
| --- | --- |
| ***Argument Dimensions*** | ***Relate Pattern*** |
| P/P, A/A | T\*T\*\*\*T\*\* |
| L/L | 1\*T\*\*\*T\*\* |
| P/L, P/A, L/A | undefined |

# SPATIAL ANALYSIS METHODS

* 1. **GENERAL DISCUSSION**

The SFS lists a number of spatial analysis methods including both constructive operations (buffer, convex hull) and set-theoretic operations (intersection, union, difference, symmetric difference).

1.1.1 Representation of Computed Geometries

The SFS states that the result of a set-theoretic method is the “point-set” result of the usual set-theoretic definition of the operation (SFS 3.2.21.1). However, there are sometimes many ways of representing a point set as a Geometry.

**(2)**

The canonical form of A.union(B) returned by JTS

**(1)**

Topologically equivalent representations for the point-set A.union(B)

**A**

**B**

**Figure 8 - Representation of computed Geometries**

The SFS does not specify an unambiguous representation for point sets returned from a spatial analysis method. One goal of JTS is to make this specification precise and unambiguous. JTS uses a canonical form for Geometries returned from spatial analysis methods. The canonical form is a Geometry which is simple and noded:

* **Simple** means that the Geometry returned will be simple according to the definition in

*Section 13.1.3*

* **Noded** applies only to overlays involving LineStrings. It means that all intersection points between the argument LineStrings will be present as endpoints of LineStrings in the result.

This definition implies that for non-simple geometries which are arguments to spatial analysis methods, a line-dissolve process is performed on them to ensure that the results are simple.

## CONSTRUCTIVE METHODS

Because the convexHull() method does not introduce any new coordinates, it is guaranteed to return a precisely correct result. Since it is not possible to represent curved arcs exactly in JTS, the buffer() method returns a (close) approximation to the correct answer.

GeometryCollections are supported as arguments to the convexHull() method, but not to the buffer() method.

**(2)**

A.buffer(*dist*)

**(1)**

A.convexHull()

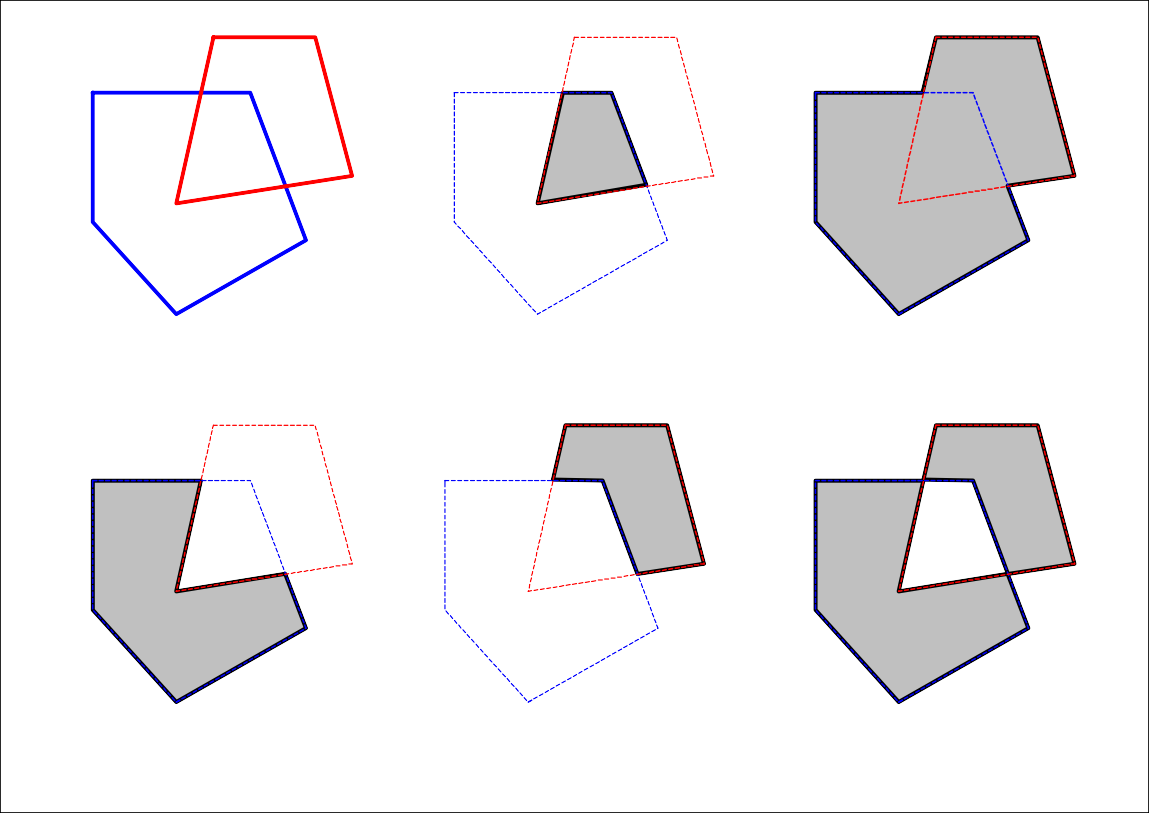
**A**

**Figure 9 - The constructive spatial analysis methods**

## SET-THEORETIC METHODS

The spatial analysis methods will return the most specific class possible to represent the result. If the result is homogeneous, a Point, LineString, or Polygon will be returned if the result contains a single element; otherwise, a MultiPoint, MultiLineString, or MultiPolygon will be returned. If the result is heterogeneous a GeometryCollection will be returned.

Because it is not clear at this time what semantics for set-theoretic methods involving GeometryCollections would be useful, GeometryCollections are not supported as arguments to the set-theoretic methods.



**(6)**

A.symDifference(B)

**(5)**

B.difference(A)

**(4)**

A.difference(B)

**(3)**

A.union(B)

**(2)**

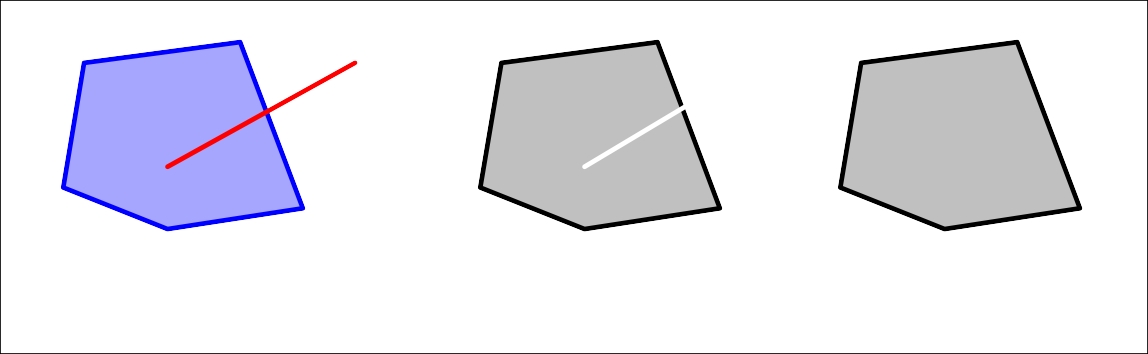
A.intersection(B)

**A**

**B**

**Figure 10 - The set-theoretic spatial analysis methods**

For certain inputs, the Difference and SymDifference methods may compute non-closed sets. This can happen when the arguments overlap and have different dimensions. Since JTS Geometry objects can represent only closed sets, the spatial analysis methods are specified to return the closure of the point-set-theoretic result.



**(2)**

A.difference(B) (a closed set)

**(1)**

A - B : the set-theoretic result (a non-closed set)

**A**

**B**

**Figure 11 - JTS always returns closed Geometries**

* 1. **METHOD SPECIFICATIONS**
     1. Buffer

The buffer of a Geometry at a distance d is the Polygon or MultiPolygon which contains all points within a distance d of the Geometry. The distance d is interpreted according to the Precision Model of the Geometry. Both positive and negative distances are supported.

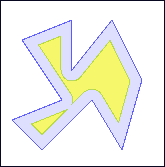
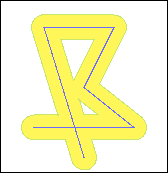
### a.buffer(d) =

### d > 0 : { x ³ ¥2 | dist(x, a) ã d }

### d < 0 : { x ³ ¥2 | x ³ a Ù dist(x, boundary(a)) > d }

In mathematical terms, buffering is defined as taking the Minkowski sum or difference of the Geometry with a disc of radius equal to the absolute value of the buffer distance.

Positive and negative buffering is also referred to as **dilation** or **erosion**. In CAD/CAM terms, buffering is referred to as computing an offset curve.



**Figure 12 – Positive and Negative buffers**

JTS allows specifying different **end cap styles** for buffers of lines. The end cap style is available when using the BufferOp class directly. The following end cap styles are supported:

|  |  |
| --- | --- |
| ***Style Name*** | ***Description*** |
| CAP\_ROUND | The usual round end caps |
| CAP\_BUTT | End caps are truncated flat at the line ends |
| CAP\_SQUARE | End caps are squared off at the buffer distance beyond the line ends |

The following diagrams illustrate the effects of specifying different end cap styles:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **CAP\_ROUND** | **CAP\_BUTT** | **CAP\_SQUARE** |

* + 1. ConvexHull

The convex hull of a Geometry is the smallest convex Polygon that contains all the points in the Geometry. If the convex hull contains fewer than 3 points, a lower dimension Geometry is returned, specified as follows:

|  |  |
| --- | --- |
| **Number of Points in convex hull** | **Geometry Class of result** |
| 0 | empty GeometryCollection |
| 1 | Point |
| 2 | LineString |
| 3 or more | Polygon |

JTS will return a Geometry with the minimal number of points needed to represent the convex hull. In particular, no more than two consecutive points will be collinear.

* + 1. Intersection

The intersection of two Geometries A and B is the set of all points which lie in both A and B.

### a.intersection(b) = { x ³ ¥2 | x ³ a Ù x ³ b }

* + 1. Union

The union of two Geometries A and B is the set of all points which lie in A or B.

### a.union(b) = { x ³ ¥2 | x ³ a Ú x ³ b }

* + 1. Difference

The difference between two Geometries A and B is the set of all points which lie in A but not in B. This method returns the closure of the resultant Geometry.

### a.difference(b) = closure( { x ³ ¥2 | x ³ a Ú x ´ b } )

* + 1. SymDifference

The symmetric difference of two Geometries A and B is the set of all points which lie in either A or B but not both. This method returns the closure of the resultant Geometry.

### a.symDifference(b) = closure( { x ³ ¥2 | (x ³ a Ù x ´ b) Ú (x ´ a Ù x ³ b) } )

# OTHER METHODS

* + 1. Boundary

As stated in SFS Section 2.1.13.1, “the boundary of a Geometry is a set of Geometries of the next lower dimension.” JTS uses GeometryCollections to represent sets of Geometries.

For all empty Geometrys, boundary(G) = empty GeometryCollection (JTS).

For non-empty Geometries, the boundaries are defined as follows:

|  |  |
| --- | --- |
| **Geometry Class** | **Definition of boundary()** |
| Point | empty GeometryCollection |
| MultiPoint | empty GeometryCollection |
| LineString | if closed: empty MultiPoint  if not closed: MultiPoint containing the two endpoints. |
| LinearRing | empty MultiPoint |
| MultiLineString | MultiPoint obtained by applying the Mod-2 rule to the boundaries of the element LineStrings |
| Polygon | MultiLineString containing the LinearRings of the shell and holes, in that order (SFS 2.1.10) |
| MultiPolygon | MultiLineString containing the LinearRings for the boundaries of the element polygons, in the same order as they occur in the MultiPolygon (SFS 2.1.12/JTS) |
| GeometryCollection | (SFS Section 2.1.13.1) “The boundary of an arbitrary collection of geometries whose interiors are disjoint consist of geometries drawn from the boundaries of the  element geometries by application of the Mod-2 rule.” |

* + 1. IsClosed

The SFS meaning of “closed” is different to the topological meaning of closed. The SFS “isClosed” method applies to Curves only. It tests whether the start point and end point of the Curve are the same point. In contrast, topological closure depends on whether a geometry contains its boundary. As discussed earlier, all instances of SFS geometry classes are topologically closed by definition.

For empty Curves, isClosed is defined to have the value false.

* + 1. IsSimple

In general, the SFS specifications of simplicity seem to follow the rule:

*A Geometry is simple if and only if the only self-intersections are at boundary points.*

For Point, MultiPolygon and GeometryCollection the SFS does not provide a specification for simplicity. JTS provides a specification for these Geometry types based on the above rule.

For all empty Geometrys, isSimple = true. (JTS)

|  |  |
| --- | --- |
| **Geometry Class** | **Definition of isSimple()** |
| Point | true (JTS) |
| MultiPoint | true if no two Points in the MultiPoint are equal (SFS 2.1.4) |
| LineString | true if the curve does not pass through the same point twice (excepting the endpoints, which may be identical) (SFS 2.1.5) |
| LinearRing | true (SFS 2.1.6) |
| MultiLineString | true iff all of its element LineStrings are simple and the  only intersections between any two elements occur at points that are on the boundaries of both LineStrings. |

|  |  |
| --- | --- |
|  | (SFS 2.1.7) |
| Polygon | true (SFS 2.1.10) |
| MultiPolygon | true (JTS) |
| GeometryCollection | true if all its elements are simple and the only  intersections between any two elements occur at points that are on the boundaries of both elements. (JTS) |

* + 1. IsValid

Since JTS Geometry objects are constructed out of user-supplied point sequences, it is possible that a Geometry object does not in fact specify a topologically valid Geometry according to the SFS. JTS does not validate Geometries when they are constructed, for reasons of efficiency. The isValid() method is provided to test whether a Geometry is valid according to the SFS spec.

The validation rules checked are as follows:

|  |  |  |
| --- | --- | --- |
| **Rule** | **Description** | **Applies To** |
| Valid Coordinates | Coordinates must contain valid numeric values | All |
| Valid Point Count | Coordinate sequences must contain a valid number of points for their containing geometry:  LineString – 0 or 2 or more LinearRing – 0 or 4 or more | All |
| No Invalid Self- Intersections | Any two rings may intersect in at most a single point. | A |
| No Duplicate Rings | Rings within an area must not be duplicated. Duplicate rings are rings which have identical point sequences up to order. | A |
| No Self-Intersecting Rings | Rings must not self-intersect. | LR, A |
| Holes Contained In Shell | Holes must be contained within their parent shell. | A |
| Holes Not Nested | Holes must not be nested. | A |
| Shells Not Nested | Shells must not be nested. | mA |
| Interiors Connected | The interior of a Polygon must be connected. | A |
| Interiors Connected | The interior of a Polygon must be connected. | A |
| Invalid Coordinates | The interior of a Polygon must be connected. | A |

JTS also provides the IsValidOp class, which performs the same checks as isValid but which returns the exact nature and location of a validation failure.

# WELL-KNOWN TEXT INPUT/OUTPUT

The Well-Known Text format for SFS Features is defined in SFS Section 3.2.5. The Well- Known Text Reader and Writer will parse and output this format.

Note that there is an inconsistency in the SFS. The WKT grammar states that MultiPoints are represented by “MULTIPOINT ( ( x y), (x y) )”, but the examples show MultiPoints as “MULTIPOINT ( x y, x y )”. Other implementations follow the latter syntax, so JTS will adopt it as well.

The SFS does not define a WKT representation for Linear Rings. JTS has extended the WKT syntax to support these, using the keyword LINEARRING.

d

* 1. **SYNTAX FOR WELL-KNOWN TEXT**

The syntax for the Well-known Text representation of Geometry is defined below.

*The notation {}\* denotes 0 or more repetitions of the tokens within the braces. The braces do not appear in the output token list.*

<Geometry Tagged Text> :=

<Point Tagged Text>

| <LineString Tagged Text>

| <LinearRing Tagged Text>

| <Polygon Tagged Text>

| <MultiPoint Tagged Text>

| <MultiLineString Tagged Text>

| <MultiPolygon Tagged Text>

| <GeometryCollection Tagged Text>

<Point Tagged Text> := POINT <Point Text>

<LineString Tagged Text> := LINESTRING <LineString Text>

<LinearRing Tagged Text> := LINEARRING <LineString Text>

<Polygon Tagged Text> := POLYGON <Polygon Text>

<MultiPoint Tagged Text> := MULTIPOINT <Multipoint Text>

<MultiLineString Tagged Text> := MULTILINESTRING <MultiLineString Text>

<MultiPolygon Tagged Text> := MULTIPOLYGON <MultiPolygon Text>

<GeometryCollection Tagged Text> := GEOMETRYCOLLECTION <GeometryCollection Text>

<Point Text> := EMPTY | ( <Point> )

<Point> := <x> <y>

<x> := double precision literal

<y> := double precision literal

<LineString Text> := EMPTY

| ( <Point> {, <Point> }\* )

<Polygon Text> := EMPTY

| ( <LineString Text> {, <LineString Text> }\*)

<Multipoint Text> := EMPTY

| ( <Point > {, <Point > }\* )

<MultiLineString Text> := EMPTY

| ( <LineString Text> {, <LineString Text> }\* )

<MultiPolygon Text> := EMPTY

| ( <Polygon Text> {, <Polygon Text> }\* )

<GeometryCollection Text> := EMPTY

| ( <Geometry Tagged Text>

## WELL-KNOWN TEXT READER

The Well-Known Text reader (WKTReader) is designed to allow extracting Geometry objects from either input streams or internal strings. This allows it to function as a parser to read Geometry objects from text blocks embedded in other data formats (e.g. XML).

A WKTReader is parameterized by a GeometryFactory, to allow it to create Geometry objects of the appropriate implementation. In particular, the GeometryFactory will determine the PrecisionModel and SRID that is used.

The WKTReader will convert the input numbers to the precise internal representation.

## WELL-KNOWN TEXT WRITER

The Well-Known Text writer outputs the textual representation of a Geometry object to a Java Writer.

The WKTWriter will output coordinates rounded to the precision model. No more than the maximum number of necessary decimal places will be output.

# REFERENCES

[AS] The OpenGIS Abstract Specification: An Object Model for Interoperable Geoprocessing, Revision 1, OpenGIS Consortium, Inc, OpenGIS Project Document Number 96-015R1, 1996.

[Ava97] F. Avnaim, J-D. Boissonnat, O. Devillers, F. Preparata and M. Yvinec. *“*Evaluating signs of determinants using single-precision arithmetic” [prisme- 2306a. In *Algorithmica*, Vol. 17, pp. 111-132, 1997. [http://www-](http://www-/) sop.inria.fr/prisme/publis/abdpy-esdus-97.ps.gz

[Bri98] A. Brinkmann, K. Hinrichs. “Implementing exact line segment intersection in map overlay”. In *Proceedings of the 8th International Symposium on Spatial Data Handling Vancouver, July 11-15, 1998*, pp. 569-579, 1998.

[Sch97] Stefan Schirra. "Precision and robustness in geometric computations". In

*Algorithmic Foundations of Geographic Information Systems*, M. van Kreveld,

J. Nievergelt, T. Roos, and P. Widmayer Eds., LNCS 1340, Springer, pp. 255- 287, 1997.

[SFS] OpenGIS Simple Features Specification For SQL Revision 1.1. Open GIS Consortium, Inc. OpenGIS project Document 99-049. <http://www.opengis.org/techno/specs/99-049.pdf>