Tektosyne User's Guide

Overview of Library Features by Package

Christoph Nahr
christoph.nahr@kynosarges.org

Published 08 April 2017 on the Tektosyne home page
http://www.kynosarges.org/Tektosyne.html

Abstract

This guide summarizes the contents of the Tektosyne Library for Java. The current versions of this document and of the library itself are available at the Tektosyne home page. Please see there for system requirements and other information.

This guide covers all Tektosyne packages and their public classes. The goal is to provide a compact overview of the library's functionality, while explaining some complex or unusual features in greater detail. Please see the Javadoc class reference included with the library for a complete documentation of all types.

Online Reading. This document contains a "Bookmarks" navigation tree. Click on any tree node to jump to the corresponding section. Moreover, all phrases in blue color are clickable hyperlinks that will take you to the section or address they describe.

Colophon. This document was written in LaTeX using MiKTeX 2.9 with XeLaTeX, KOMA-Script, and various other packages. See LaTeX Typesetting with MiKTeX for details.

The UML diagrams were reverse-engineered from the compiled Java JAR files, using my free Class Diagrammer application, and embedded as PDF files.

Body text is set in Minion 12 pt from Adobe's Minion Pro collection, designed by Robert Slimbach. Subtitles and diagram text are set in various sizes and weights of Myriad from Adobe's Myriad Pro collection, designed by Robert Slimbach and Carol Twombly.

Identifiers and code fragments outside of UML diagrams are set in Microsoft's Consolas, designed by Lucas de Groot. The font is artificially compressed by 20% to take up less space.

Date	Version	Library	Description
2017–04–08	2.0.1	6.0.1	Added NaN note, updated `PointDComparator`
2016–12–14	2.0.0	6.0.0	Revised for rewritten Java library
2012–06–09	1.2.0	5.6.3	Changed typesetting to LaTeX with MiKTeX
2012–05–30	1.1.1	5.6.3	Added `RectLocation`
2012–03–31	1.1.0	5.6.1	Changed typesetting to DITA with oXygen
2012–02–26	1.0.3	5.6.0	Added `VisualSource`, `ConcurrentVisualHost`
2012–01–09	1.0.2	5.5.6	Added `AssemblyExtensions`
2011–06–24	1.0.1	5.5.2	Updated `QuadTree<T>`, `Subdivision`, `IGraph2D<T>`
2011–05–31	1.0.0	5.5.1	Initial release, using DITA with FrameMaker

Contents

List of Figures

Chapter 1

Package Overview

The Tektosyne Library ships in a single Java JAR file. Its root package is `org.kynosarges.tektosyne` and splits further into three subpackages. Figure 1.1 shows an overview.

The download archive also contains unit tests and a self-explanatory GUI application with testing and demonstration dialogs, `Tektosyne.Demo`. This guide does not cover them, but Chapter 6 presents benchmark results obtained with the demo application.

1.1 Design Goals

Tektosyne provides algorithms for computational geometry and graph-based pathfinding, along with supporting mathematical algorithms and specialized collections. The library is designed to be independent of any specific environment or GUI framework, and accordingly only requires the Java SE 8 Compact 1 profile. (The demo application is based on JavaFX.)

I created Tektosyne for my own use in the implementation of computer strategy games and simulations, as detailed in the historical notes below. However, this should not limit the library's usefulness for other applications such as mapping. All general algorithms are textbook implementations with no built-in restrictions to any particular use case.

Not-a-Number. Tektosyne types and algorithms do *not* check floating-point arguments for IEEE 754 NaN (Not-a-Number) values. Valid inputs should never produce them, but you can expect "garbage in, garbage out" when supplying NaN inputs. Note that many `System.Math` methods likewise silently return invalid results for invalid floating-point inputs.

1.2 Design History

The rest of this chapter tracks the convoluted history of the Tektosyne project. It's not relevant to understanding or using the library but you might find it amusing nonetheless.

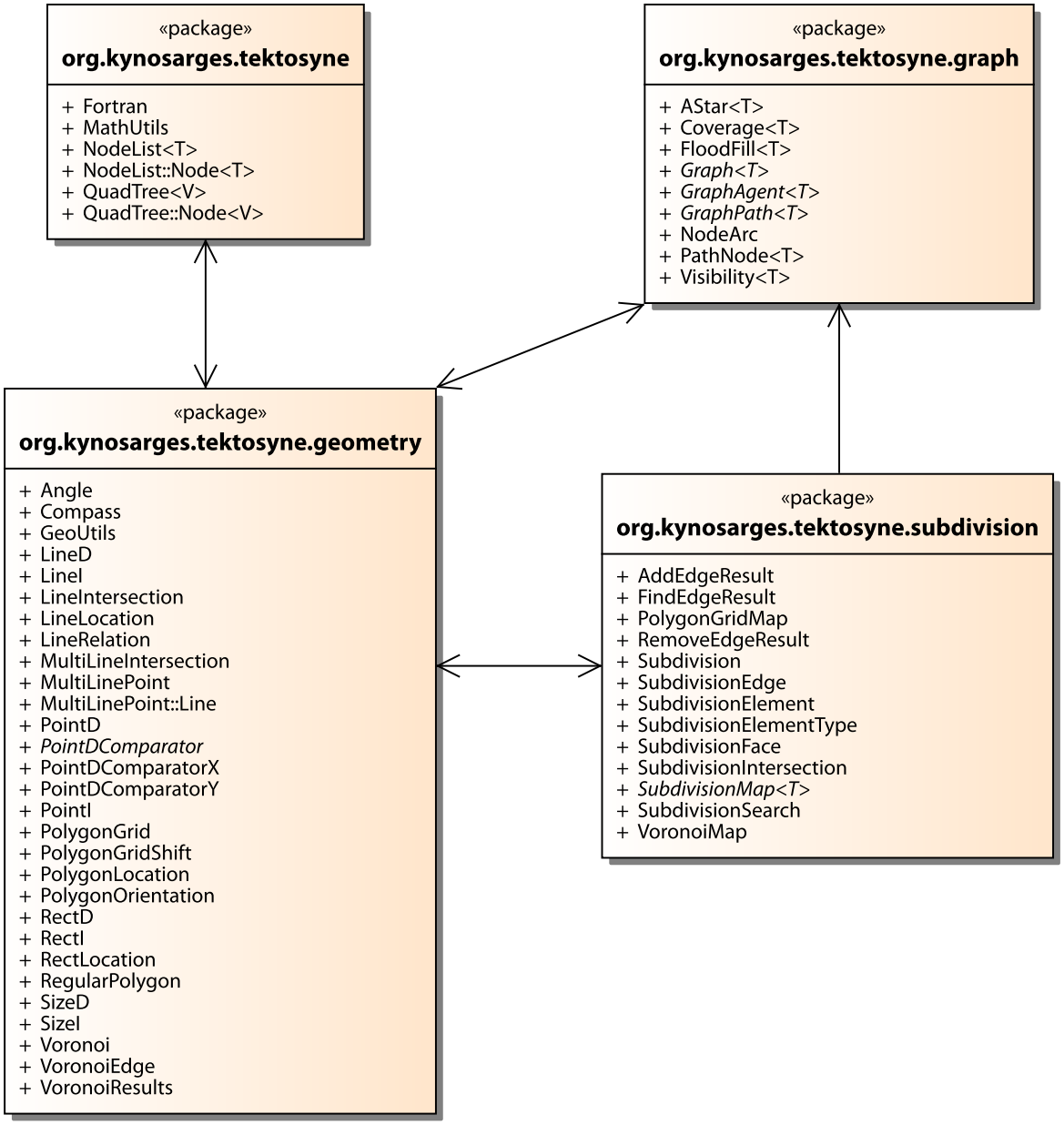

Figure 1.1: Package Overview

1.2.1 .NET Origins

Tektosyne originated in 2002 as a toolbox library for .NET that was literally called "Toolbox" and had no design goals whatsoever. I just threw in any number of string utilities, Windows Forms helpers, improved collections, Simple MAPI methods, and whatever else I found lacking in the early .NET base class library.

Additionally, though, there were already a number of mathematical, geometric, and pathfinding classes required by my simultaneously developed Hexkit Strategy Game System for .NET. Version 2.6 in 2004 renamed the library from Toolbox to the more distinctive Tektosyne. Version 3 in 2005 dropped all hard-coded primitive collections due to the introduction of .NET generics, and version 4 in 2008 dropped Windows Forms in favor of WPF. This was also the first release to start separating geometric types from on-screen drawing.

From that point on new development focused on computational geometry and graph algorithms, partly driven by the requirements of my new Myriarch Combat Simulator. Version 4.3 in 2010 and the following releases added Voronoi diagrams, point comparators, quadrant trees, convex hulls, multi-line intersections, and more. Version 5 later in 2010 introduced the current set of geometric primitives, and the next releases added planar subdivisions.

Version 5.5 in 2011 split these core types from unrelated Windows toolbox functionality. The former now resided in `Tektosyne.Core` which was a superset of the present Java library. Differences are noted in the enclosed `ReadMe` file. Version 5.6.5 in 2012 ended active development of Tektosyne for .NET, with just one minor update for recent Visual Studio and .NET Framework versions in 2015.

1.2.2 Moving to Java

Around 2012 it had become increasingly obvious that Microsoft was effectively abandoning .NET as a rich client development platform, while Java had simultaneously recovered from its takeover-induced hiatus and began rapidly making up lost ground in both language features and GUI tooling (i. e. JavaFX).

Running a variety of comparison tests and prototypes I noticed with some incredulity that Java's 64-bit Server VM quite dramatically outperformed .NET on Windows[1], and that its standard library was so much richer (e. g. the Java Collections Framework) that I could simply drop many of the helpers I had written for .NET.

So I decided to switch my open-source activity from .NET to Java, including successive porting of my existing projects. Tektosyne is the third completed port, after Star Chess (from C rather than C#) and Class Diagrammer, and Tektosyne now in turn enables ports of Hexkit and Myriarch. These will require a good deal more consideration, though: Hexkit in particular will need some radical simplification. Time will tell.

1. If you cannot reproduce this on your Windows system you are most likely running the wrong Java VM, thanks to Oracle's stupid distribution policy. See Java Client VM for details.

Chapter 2

Root Package

The root package `org.kynosarges.tektosyne` contains general mathematical utilities and two specialized collections.

2.1 Mathematics

This section comprises mathematical utilities unrelated to computational geometry; see Chapter 3 for the latter. Figure 2.1 shows the two classes described below.

`Fortran` — Static methods whose names and semantics mirror standard functions of Fortran 90, with overloads for `int`, `long`, `float`, and `double`. All methods operating on integral values throw `ArithmeticException` if an overflow occurs.

`MathUtils` — Static methods providing checked conversions, epsilon comparisons, prime number test, random element retrieval, and range restriction and normalization.

2.2 Collections

This section comprises two node-based collections that publicly expose their node structure as read-only objects, allowing O(1) navigation and manipulation once a node has been obtained. Both classes implement the appropriate interfaces of the Java Collections Framework. Figure 2.2 shows the two collection and node classes.

`NodeList<T>` — Provides a generic linked list that is exactly equivalent to Java's `LinkedList` but with publicly accessible nodes. The `T` values of nodes are directly settable, unlike their structural properties.

`NodeList.Node<T>` — Provides a node in a `NodeList<T>`.

`QuadTree<V>` — Provides a generic quadrant tree whose keys are `org.kynosarges.tektosyne.geometry.PointD` locations, i. e. a two-dimensional search tree that recursively divides a specified bounding rectangle into equal-sized quadrants. Finding the quadrant that contains a given point and finding all points within a given range are both logarithmic operations.

`QuadTree.Node<V>` — Provides a node in a `QuadTree<V>`.

The quad-tree implementation is based on Michael J. Laszlo's *Computational Geometry and Computer Graphics in C++*, Prentice Hall 1996. Additional features include a heuristic depth probe to speed up searches in large trees, inspired by Sariel Har-Peled's lecture *Quadtrees – Hierarchical Grids;* and a `move` method that can reduce successive key changes to O(1) operations, provided that old and new keys are clustered within the same leaf node.

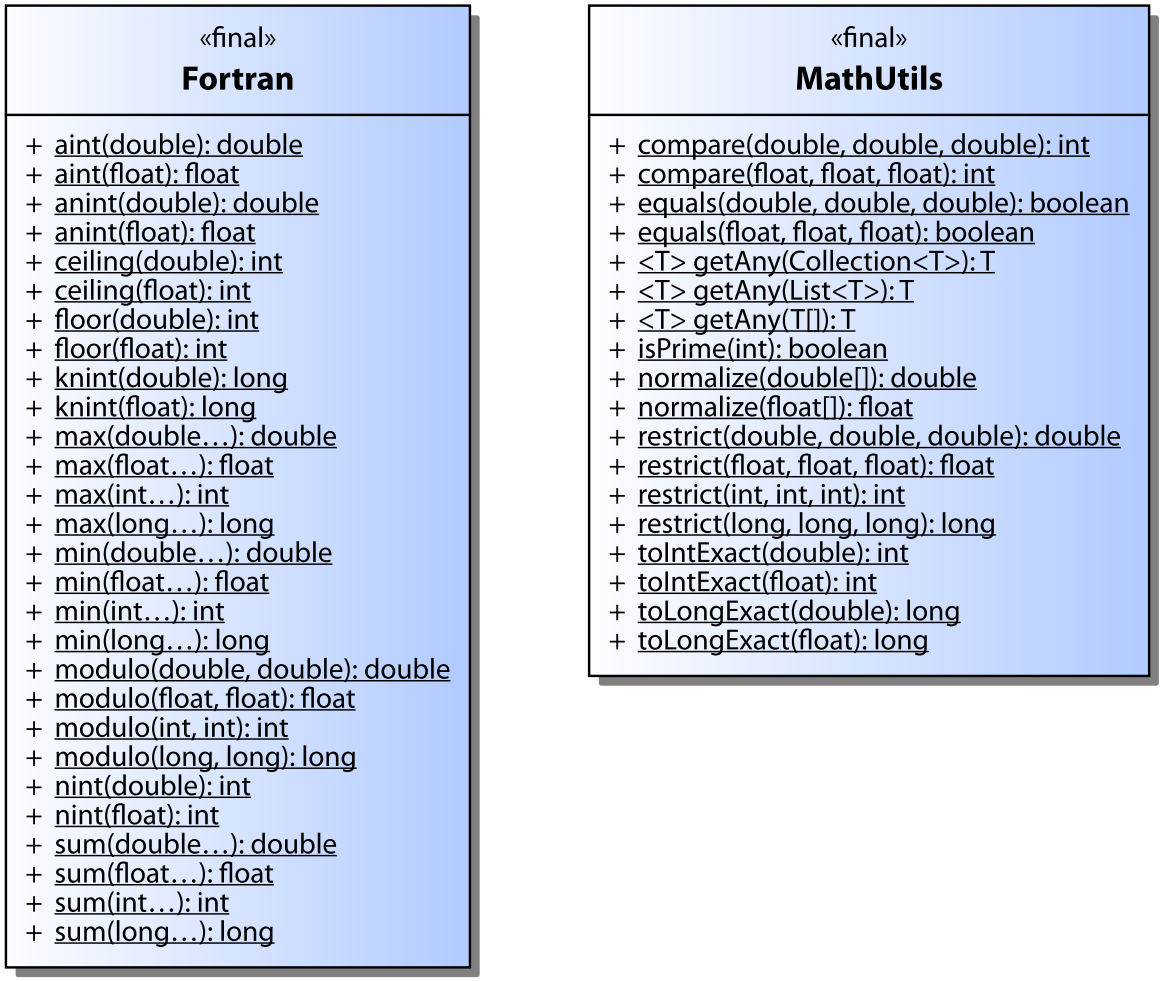

Figure 2.1: Utility Classes

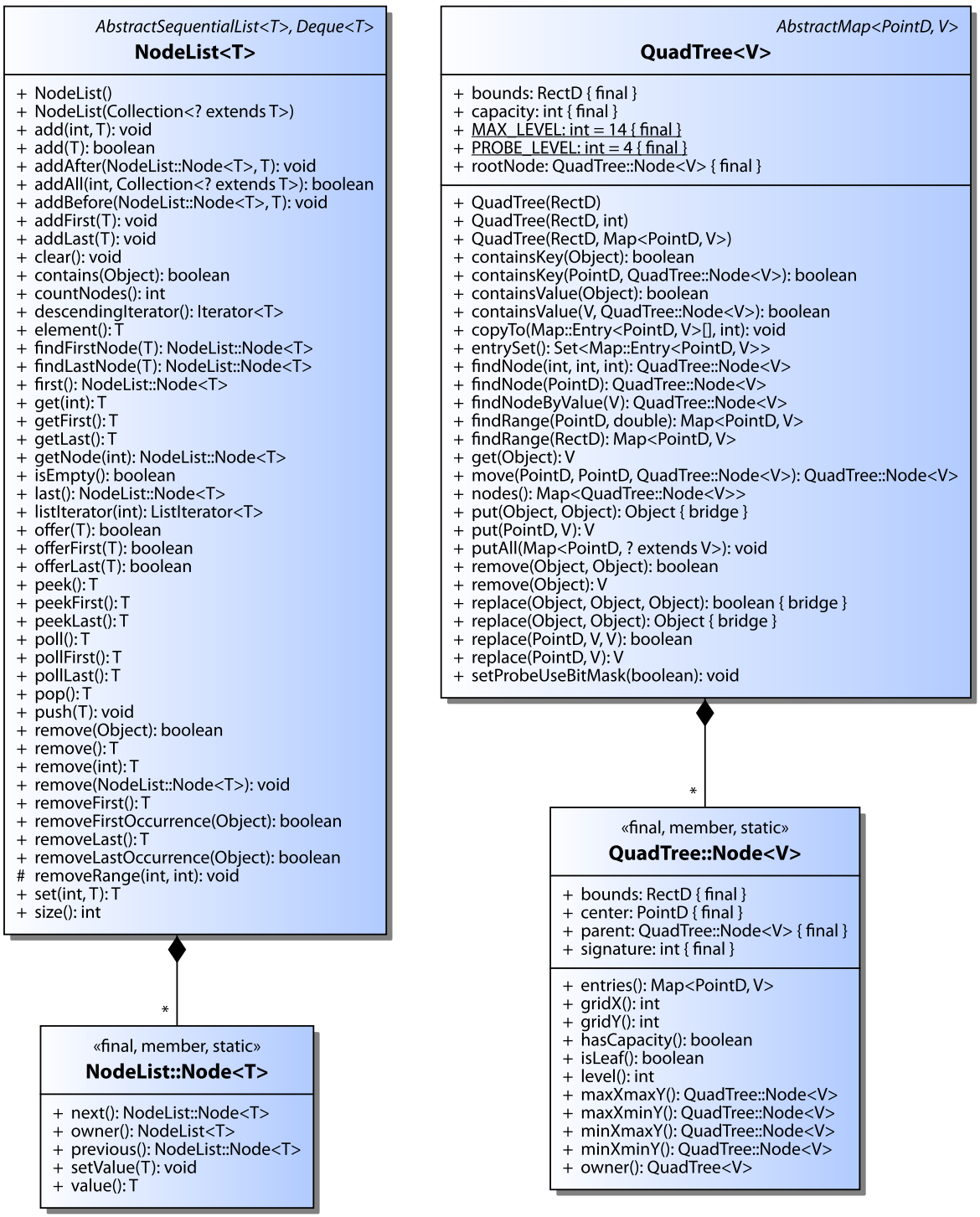

Figure 2.2: Collection Classes

Chapter 3

Geometry Package

Package `org.kynosarges.tektosyne.geometry` covers general computational geometry, including a set of geometric primitives as well as a variety of standard algorithms and data structures. All types use two-dimensional coordinates exclusively. Many algorithms were adapted from C/C++ and pseudocode programs in standard literature, including the following sources. Please consult the *Tektosyne Class Reference* for details.

— Mark de Berg et al., *Computational Geometry,* Springer-Verlag 2008 (3rd ed.)

— Michael J. Laszlo, *Computational Geometry and Computer Graphics in C++,* Prentice Hall 1996

— Joseph O'Rourke, *Computational Geometry in C,* Cambridge University Press 1988 (2nd ed.)

The orientation of the vertical axis is somewhat problematic in computational geometry. The standard mathematical orientation has y-coordinates increase upward, but the standard drawing orientation of computer graphics puts the origin in the upper-left corner of the screen and has y-coordinates increase downward. The *Tektosyne Class Reference* notes the actual orientation wherever it is relevant. Most algorithms assume mathematical orientation.

Precision. Another frequent source of trouble is floating-point imprecision. Some Tektosyne algorithms use a fixed comparison epsilon of 1^{-10} to achieve numerical stability, while others allow a user-defined epsilon. Some algorithms are available in both exact and epsilon variants. You need to experiment with your own data to determine the most suitable variant.

When an algorithm accepts a user-defined epsilon, you can usually choose a fairly large value that merges clearly distinct points rather than just compensating for floating-point imprecision. One application is to map the location of a user's mouse click on the screen to a nearby point in a geometric data structure. The `Tektosyne.Demo` application offers several test dialogs that let you experiment with super-sized comparison epsilons.

3.1 Geometric Primitives

While both the traditional Java library and more recently JavaFX define several geometric primitives, none of them are ideal for computational geometry (and they are incompatible to boot). So Tektosyne defines its own set of geometric primitives for two-dimensional coordinates, and with the following common features:

— All classes are immutable with meaningful `equals`, `hashCode`, and `toString` overrides. They can be declared value types once Java SE 10 introduces this concept.

— All classes are available in two coordinate types, `int` and `double`. The methods defined on `double` classes are a superset of those defined on `int` classes.

— All classes provide conversions to their equivalent with the other coordinate type, as well as to and from simple coordinate arrays with alternating dimensions.

— All calculations on `int` coordinates that might overflow either use an extended result type or throw `ArithmeticException` on overflow.

Figure 3.1 shows primitives with `int` coordinates, and Figure 3.2 shows primitives with `double` coordinates. The two `...Location` classes apply equally to both coordinate types, but are shown only in the first diagram to reduce clutter.

Instance methods on geometric primitives generally operate on the same type used to represent coordinates, except for algorithms that produce fractional results or might overflow `int` as mentioned above. The stand-alone algorithms described in the following sections always expect `double` coordinates and operate with `double` precision.

`PointD, PointI` — Provides spatial locations and vectors with `double` or `int` coordinates.

`LineD, LineI` — Provides directed line segments with `double` or `int` coordinates. Start and end points are stored as `PointD` or `PointI` instances, respectively.

`LineLocation` — Specifies the location of a `PointD` or `PointI` relative to a `LineD` or `LineI`, respectively.

`RectD, RectI` — Provides rectangles with `double` or `int` coordinates. Smallest and greatest coordinate pairs are stored as `PointD` or `PointI` instances, respectively.

`RectLocation` — Specifies the location of a `PointD` or `PointI` relative to a `RectD` or `RectI`, respectively. Each dimension is stored as a `LineLocation` value.

`SizeD, SizeI` — Provides spatial extensions with `double` or `int` coordinates. Checks for non-negative extensions, otherwise a subset of `PointD` or `PointI`, resp.

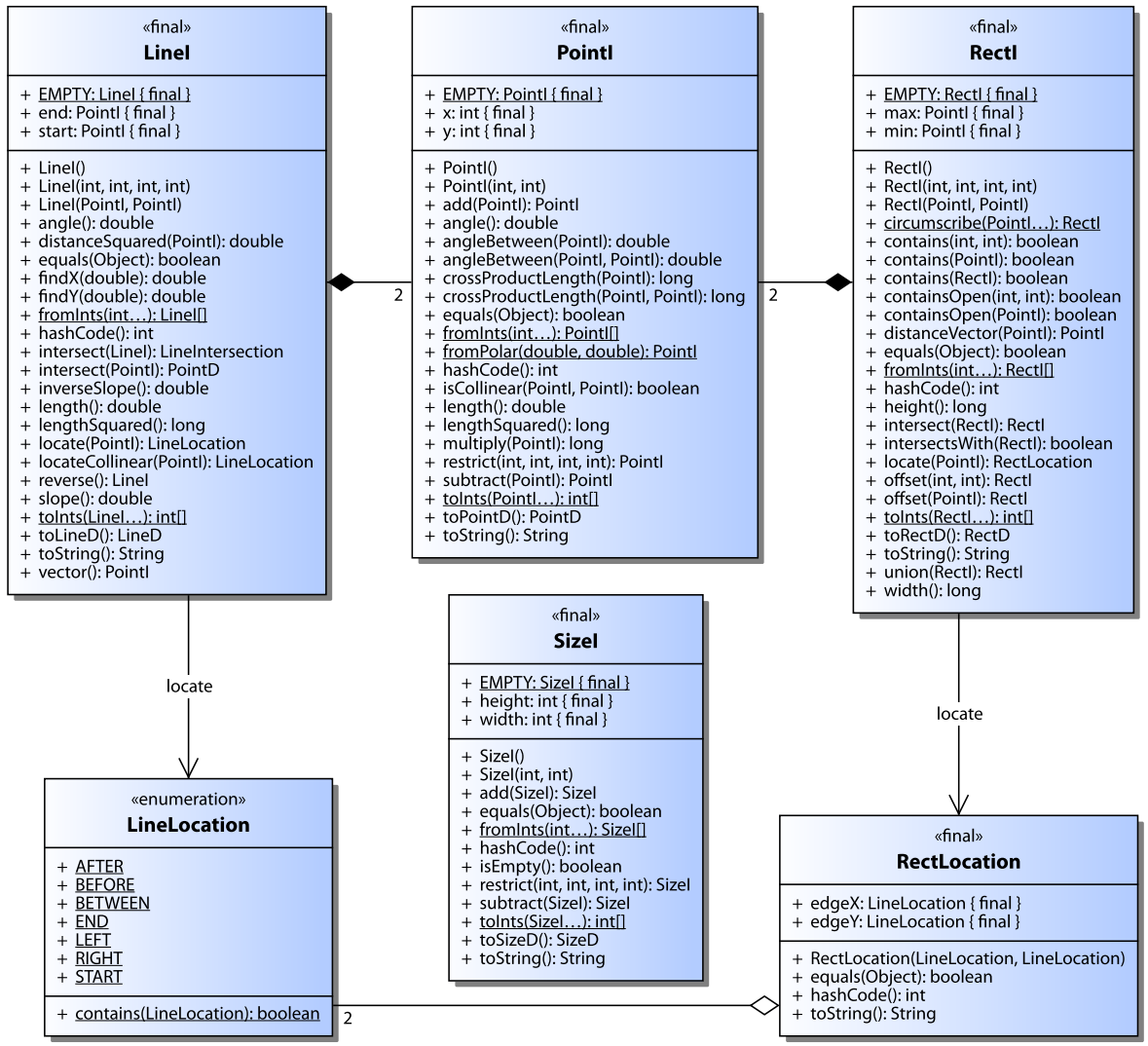

Figure 3.1: Integer-Typed Primitives

A large number of basic operations such as addition, subtraction, angle and vector calculations etc. are defined as instance methods on geometric primitives. More complex algorithms are generally available through separate classes, described in the following sections. Some noteworthy exceptions are listed below.

`LineD/I.locate/Collinear` — Finds the location of a point relative to the line segment.

`RectD/I.locate` — Finds the location of a point relative to the rectangle's edges.

`RectD.intersect(LineD)` — Performs the Liang-Barsky line clipping algorithm to intersect the rectangle with a line segment.

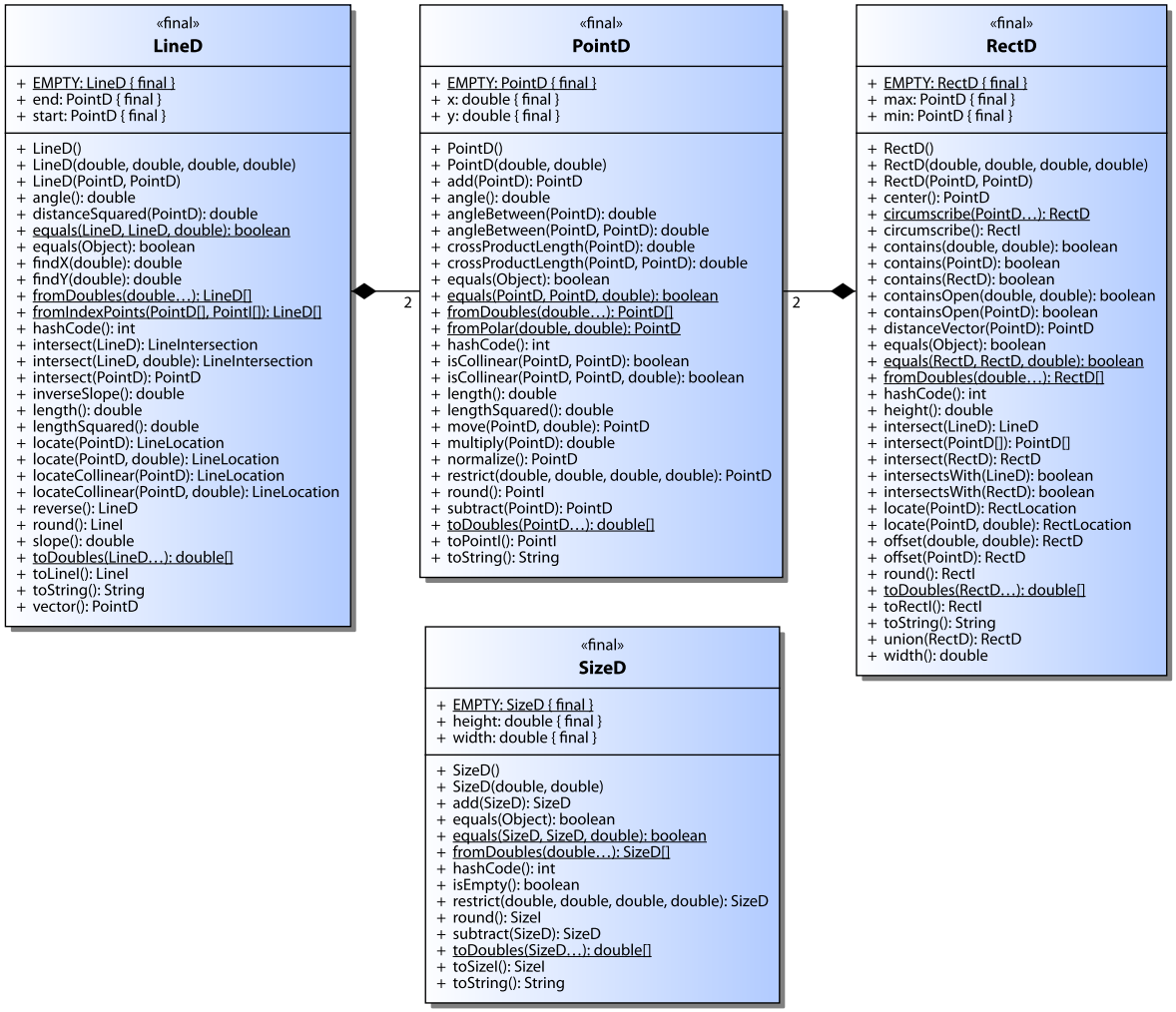

Figure 3.2: Double-Typed Primitives

`RectD.intersect(PointD[])` — Performs the Sutherland–Hodgman polygon clipping algorithm to intersect the rectangle with an arbitrary polygon.

3.2 Basic Algorithms

This section comprises basic constants and algorithms for computational geometry not covered in other sections. Figure 3.3 shows an overview.

`Angle` — Constants and methods to convert, normalize, and compare angles.

`Compass` — Specifies the eight major compass directions as angles in degrees, starting with zero degrees for north and continuing clockwise.

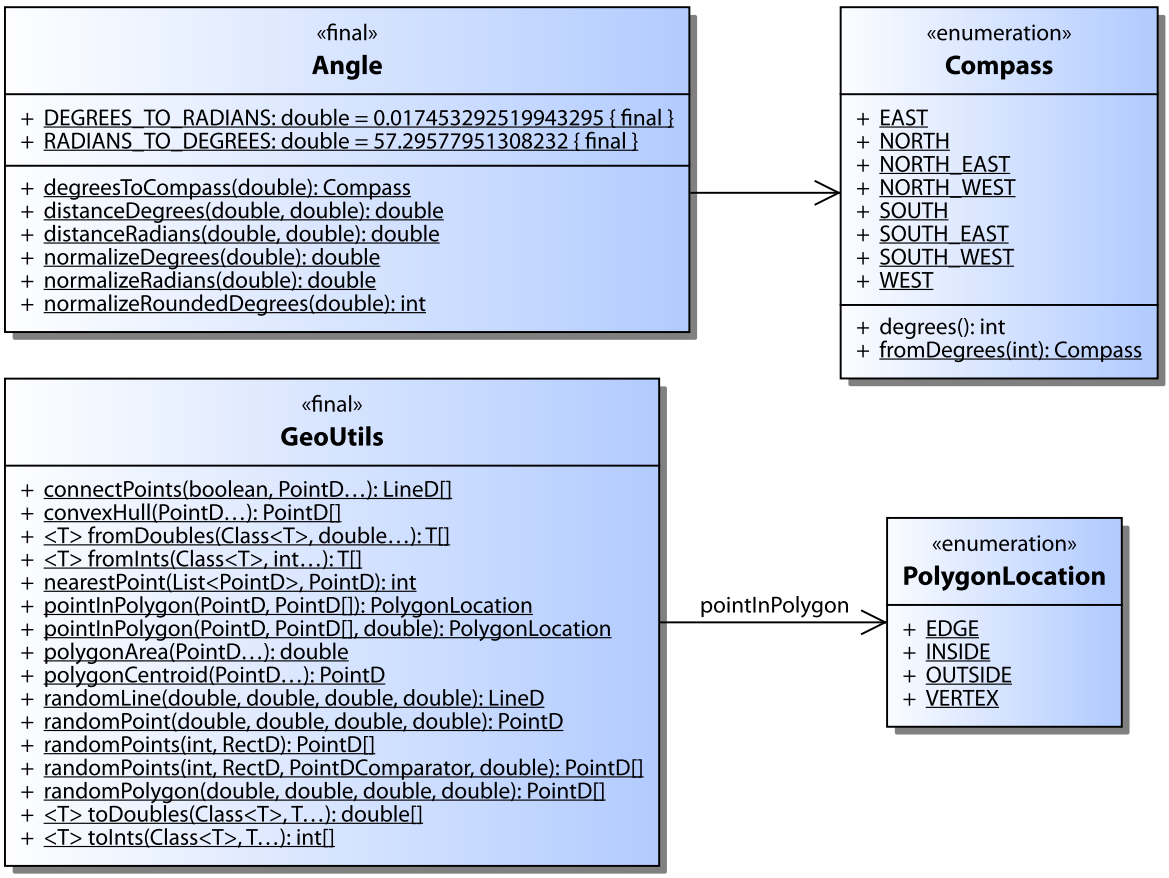

Figure 3.3: Basic Geometry Classes

`GeoUtils` — Static utility methods and general algorithms, as described below.

`PolygonLocation` — Specifies the location of a point relative to a polygon: strictly inside, strictly outside, or coinciding with an edge or a vertex.

`GeoUtils` contains a large number of disparate methods, so we list them separately.

`connectPoints` — Creates line segments that connect a given point sequence.

`convexHull` — Performs a Graham scan to compute the convex hull of a given point set.

`from…/to…` — Generic dispatchers for the array conversion methods defined on all geometric primitives.

`nearestPoint` — Linear search for the element in a given point set with the smallest Euclidean distance to a query point.

`pointInPolygon` — Performs a ray crossing algorithm to find the `PolygonLocation` of a query point relative to a given arbitrary polygon.

`polygon…` — Computes the area or centroid of a given arbitrary polygon.

`random…` — Randomly creates line segments, points, or simple closed polygons.

3.3 Line Intersection

Several algorithms intersect two or more line segments, represented either by `LineD` instances or pairs of `PointD` coordinates. Figure 3.4 shows the defining classes. `LineD` and `LineI` also define instance methods that forward to the two-segment algorithm.

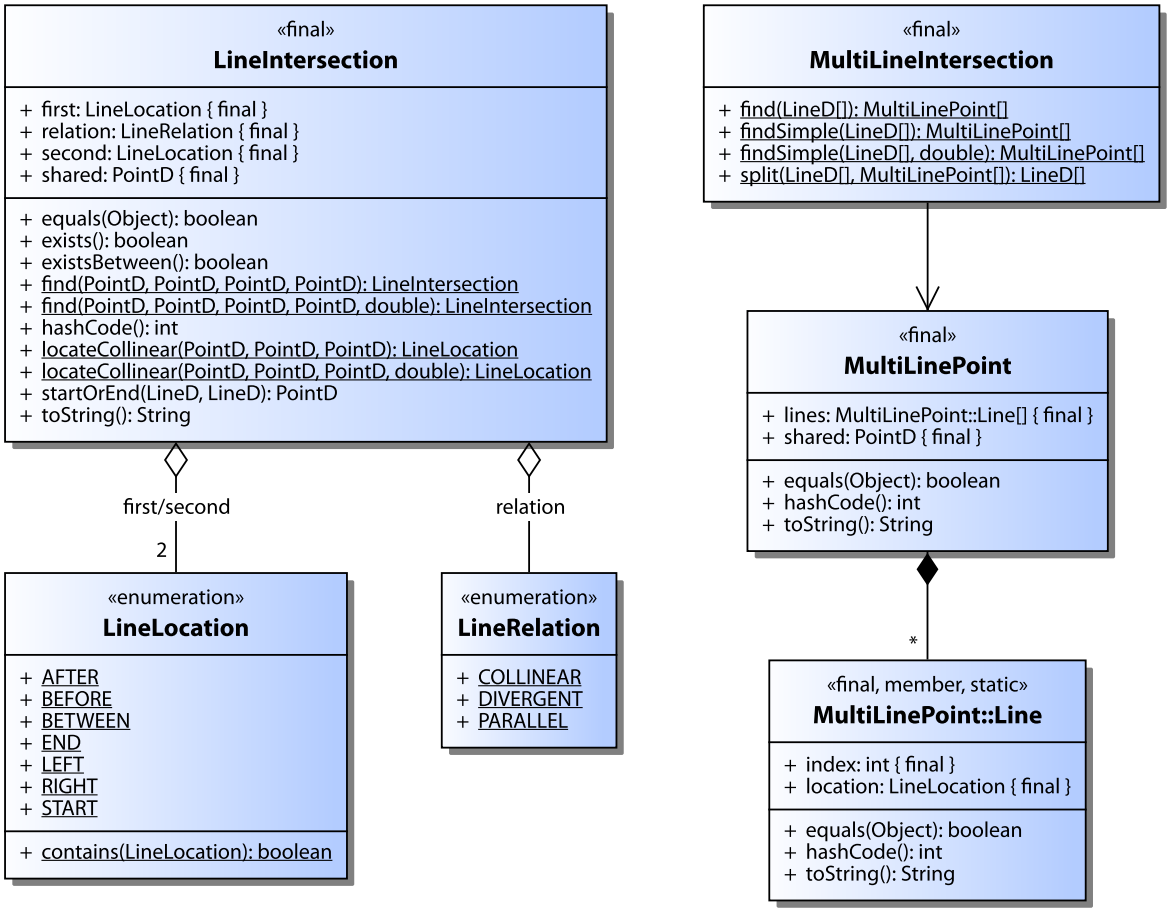

Figure 3.4: Line Intersection Classes

`LineIntersection` — Defines a robust algorithm for finding the intersection, if any, between two line segments or their infinite extensions, and also holds the result.

The algorithm examines both the cross-product lengths for each triplet of end points and the line equation parameters for both segments to determine intermediate results. If these contradict each other, the algorithm starts over with a greater comparison epsilon until both tests agree. The minimum comparison epsilon is always 1^{-10} to avoid such recursions in most cases.

`LineRelation` — Specifies the relationship between two line segments: parallel, collinear, or divergent.

`MultiLineIntersection` — Defines both a brute-force and a sweep line algorithm for finding all points of intersection between multiple line segments. The brute-force algorithm simply intersects all input lines with each other. This is always an $O(n^2)$ operation but has virtually no overhead and can accept a comparison epsilon greater than 1^{-10} to merge nearby crossings.

The Bentley-Ottmann sweep line algorithm is faster for large input sets with few intersections, but otherwise slower due to its large overhead. An improved sweep line comparer raises numerical stability to the level of the brute-force algorithm.

`MultiLinePoint` — Contains the result of either `MultiLineIntersection` algorithm.

`MultiLinePoint.Line` — Represents one of the line segments intersecting at the shared coordinates stored in a `MultiLinePoint`.

3.4 Point Comparison

Two `Comparator<PointD>` implementations compare points lexicographically, preferring either x- or y-coordinate. Comparisons can be performed exactly or with a supplied epsilon. That and related search algorithms are defined in an abstract base class, see Figure 3.5.

`PointDComparator` — Compares two points lexicographically. The comparison order depends on the concrete instance.

`PointDComparatorX` — Compares two points lexicographically, preferring x-coordinates.

`PointDComparatorY` — Compares two points lexicographically, preferring y-coordinates.

The two concrete classes define the actual comparison methods, both as instance and static versions, for clients to call as is most convenient. The more interesting algorithms are defined on the abstract base class. They work for either sorting order thanks to overridden `getPrimary/Secondary` methods.

`findNearest(List)` — Performs a nearest-point search in a lexicographically sorted list. The algorithm first performs a binary search in the preferred dimension to approximate the query point, and then expands a radius around that index until the nearest point is found. This heuristic can achieve a runtime of O(ld

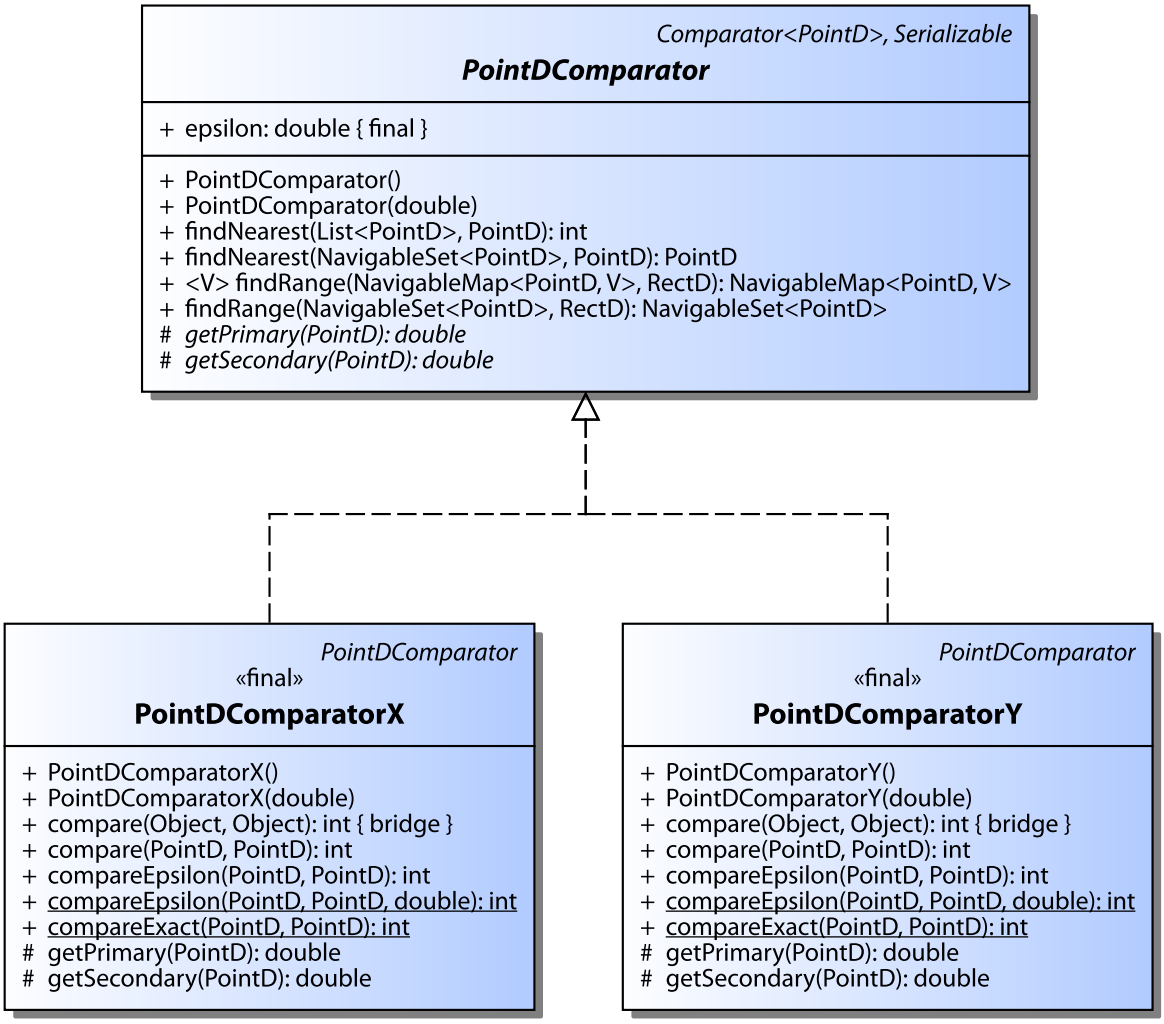

Figure 3.5: Point Comparison Classes

n) with no additional overhead, assuming that coordinates are more or less evenly distributed throughout the collection.

`findNearest(NavigableSet)` — Equivalent to the `List` overload but uses `headSet` and `tailSet` to initially approximate the query point, and then expands the search radius with bidirectional iterators. The actual performance accordingly depends on the implementation of `NavigableSet`.

`findRange` — Finds all points in a `NavigableMap/Set` that fall within a given rectangular search range. The algorithm first uses `subMap/Set` to extract all points with matching primary coordinates, and then checks each point's secondary coordinate against the search range. Runtime can approach O(ld *n*) if few points

are found but again depends on the collection's implementation.

getPrimary — Gets the primary coordinate: x for `PointDComparatorX`, else y.

getSecondary — Gets the secondary coordinate: y for `PointDComparatorX`, else x.

3.5 Regular Polygons

The classes shown in Figure 3.6 provide a flexible and efficient representation of regular polygon grids. The customizable maps of the Hexkit Strategy Game System are based on the .NET version of these types, and the *Hexkit User's Guide* describes them in greater detail. The `Tektosyne.Demo` application also provides a dialog to save and print arbitrary polygon grids.

PolygonGrid — Provides a rectangular grid of regular polygons with two-dimensional indexing. Features include efficient distance calculations, conversions between grid and world coordinates, a read-only wrapper, and pathfinding between grid locations using `Graph` algorithms (see Chapter 4).

PolygonGridMap — Maps the elements of a `PolygonGrid` to the faces of an equivalent `Subdivision` (see Chapter 5). This conversion is intended for further modification or testing, as `PolygonGrid` itself is far more efficient than `Subdivision`.

PolygonGridShift — Specifies the shifting of rows or columns in a `PolygonGrid`, i. e. whether the second row or column is shifted right or down compared to the first one.

RegularPolygon — Provides a regular polygon with three or more sides. A `RegularPolygon` with four or six sides can be used to construct a `PolygonGrid`.

PolygonOrientation — Specifies the orientation of a `RegularPolygon`: lying on an edge or standing on a vertex.

3.6 Voronoi Diagrams

The classes shown in Figure 3.7 construct two standard structures from a given set of generator sites: the Voronoi diagram, whose polygonal regions comprise all points that are nearest to each generator site; and the Delaunay triangulation, its dual graph, whose edges are the nearest-neighbor connections for all generator sites.

Voronoi — Defines a sweep line algorithm to find the Delaunay triangulation and optionally also the Voronoi diagram for a given point set, with a runtime of $O(n \log n)$. The Java implementation is based on the original C program by Steven J. Fortune.

VoronoiEdge — Represents one edge in a Voronoi diagram stored in `VoronoiResults`. This includes the diagram vertices that terminate the edge and the generator sites that are bisected by the edge.

VoronoiResults — Contains the results of the `Voronoi` algorithm. Optionally creates a planar `Subdivision` (see Chapter 5) based on the Delaunay triangulation. This subdivision may be clipped to an arbitrary rectangle, and its vertices may be mapped to polygons representing the Voronoi regions.

VoronoiMap — Maps the Voronoi regions and generator sites stored in `VoronoiResults` to the faces of an equivalent `Subdivision`. Note that pathfinding between generator sites requires the Delaunay subdivision (see above), as the pathfinding algorithms defined on `Graph` (see Chapter 4) operate only on the *edges* of a planar subdivision.

Note. `PolygonGridMap` and `VoronoiMap` actually reside in package `org.kynosarges.tektosyne.subdivision` but were included with their related classes in this chapter.

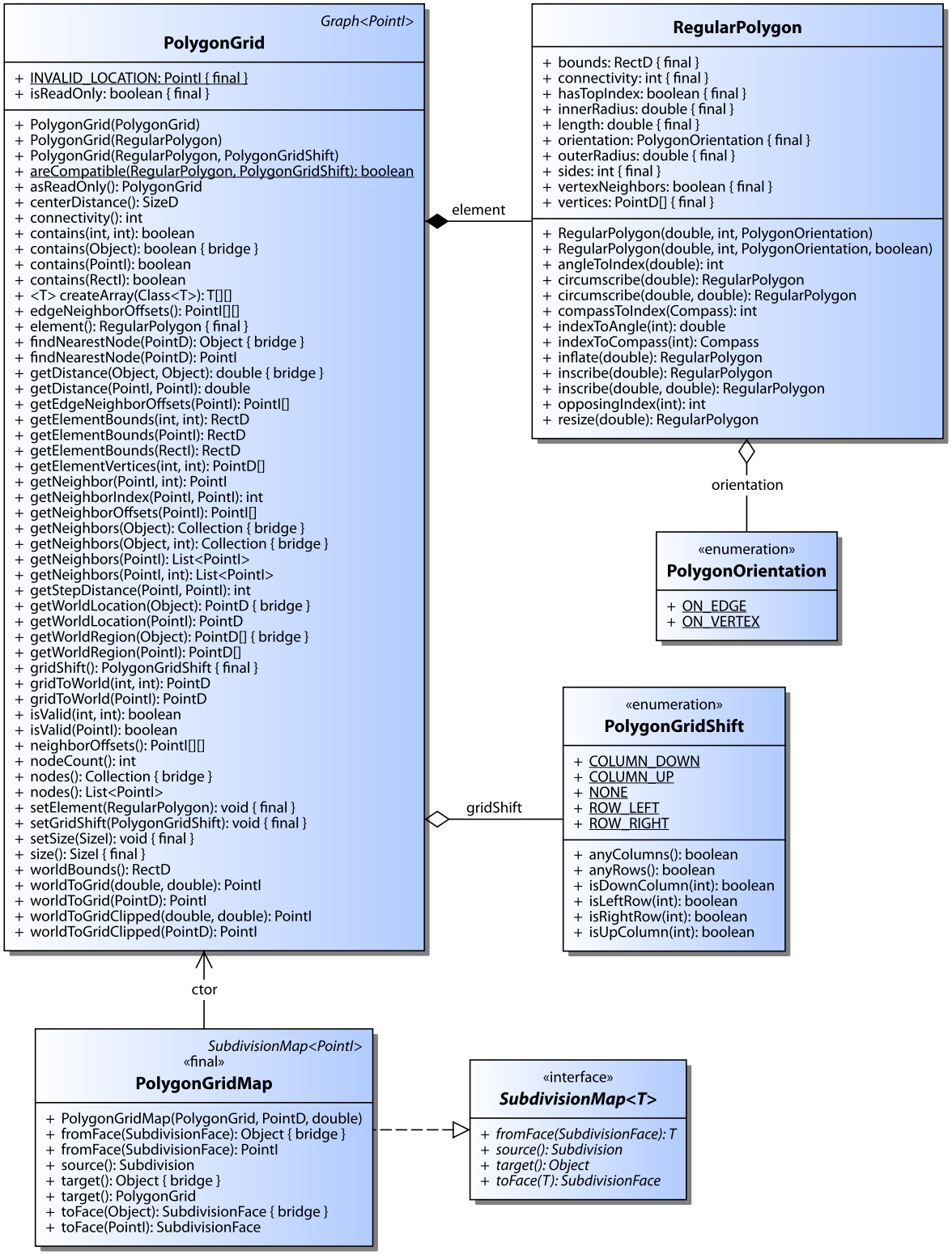

Figure 3.6: Regular Polygon Classes

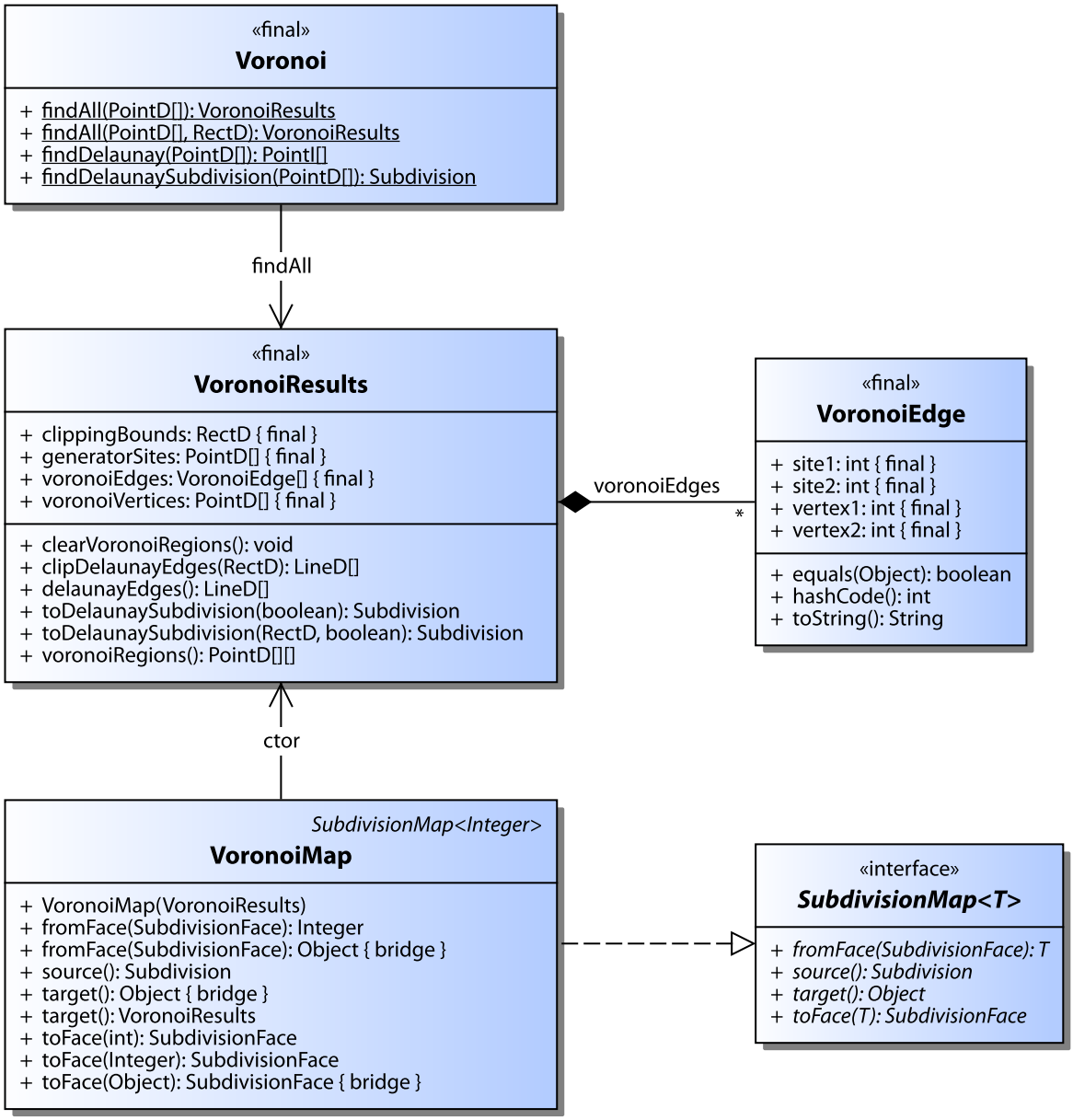

Figure 3.7: Voronoi Diagram Classes

Chapter 4

Graph Package

Package `org.kynosarges.tektosyne.graph` provides generic graph interfaces and four search algorithms that operate on them. Figure 4.1 shows all nine types in the package, as well as the two predefined Tektosyne classes that implement the central `Graph<T>` interface.

Since the mechanisms implemented here are rather complex and not based on any well-known standards, this chapter goes into greater detail than usual. To see the graph algorithms in action, try the following:

— The `Tektosyne.Demo` application contains an interactive test that runs all four algorithms on both `PolygonGrid` and `Subdivision` graphs.

— The Hexkit Strategy Game System utilizes a complex customizable implementation based on the .NET version of `PolygonGrid` graphs. The *Hexkit User's Guide* also describes the interaction of the game system and the pathfinding mechanisms.

4.1 Graphs and Agents

The two basic interfaces that connect the four generic algorithms with custom applications are `Graph<T>` and `GraphAgent<T>`. The first represents the graph itself on which searches take place, and must always be implemented. The second represents some mobile agent that traverses the graph, and is required for the `AStar<T>` and `Coverage<T>` algorithms.

4.1.1 Graph Structure

The central interface `Graph<T>` represents any graph whose `T` nodes map to polygonal regions in two-dimensional space. All graph algorithms are created with an `Graph<T>` instance on which all searches are performed.

Tektosyne contains two predefined implementations, `PolygonGrid` (see Section 3.5) and `Subdivision` (see Chapter 5). The `PolygonGrid` node type is `PointI`: each graph node is the two-

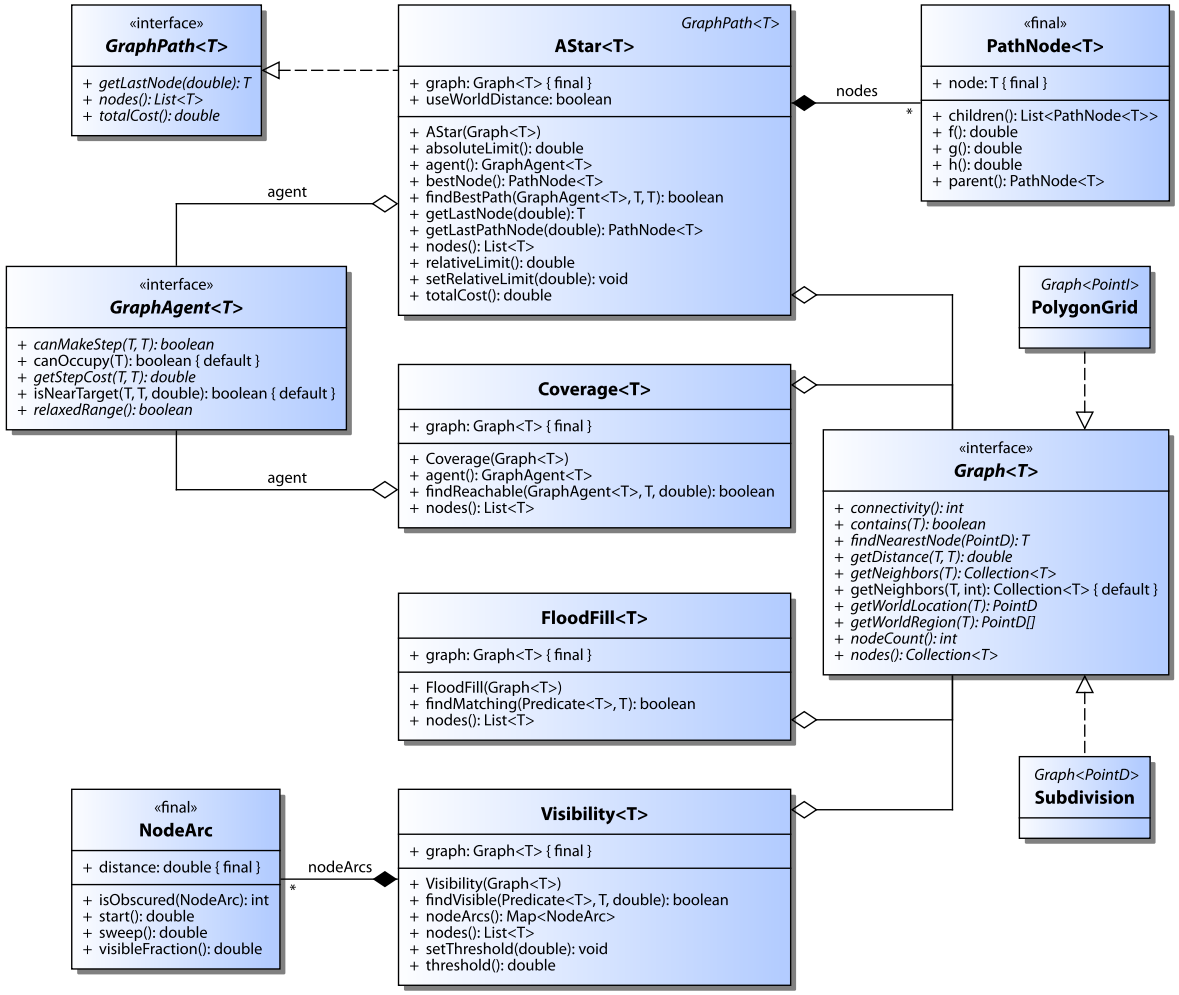

Figure 4.1: Graph Classes

dimensional index of a grid element. The `Subdivision` node type is `PointD`: each graph node is the two-dimensional location of a subdivision vertex.

The following `Graph<T>` members establish the nodes and edges of a graph.

`connectivity` — The maximum number of immediate neighbors of any graph node. A node's immediate neighbors are the nodes that share an edge with that node, and therefore can be reached within a single movement step.

`contains` — Determines whether the graph contains a specified node.

`getNeighbors` — Finds all immediate neighbors of a specified graph node, or all remoter neighbors within a given range of movement steps.

`nodeCount` — The total number of nodes in the graph.

nodes — Enumerates all nodes in the graph.

4.1.2 World Coordinates

`Graph<T>` represents a purely abstract system of node connections, but each graph node also maps to coordinates and regions in two-dimensional space. We refer to these coordinates as "world coordinates" to distinguish them from node coordinates in some graph-specific system, e. g. the two-dimensional integer indices of `PolygonGrid` elements. In the case of `Subdivision` graphs, the two systems are identical: graph coordinates equal world coordinates.

The following `Graph<T>` members establish relationships between graph nodes and world coordinates (although this is not necessarily the case for `getDistance`).

findNearestNode — Finds the graph node nearest to the specified world coordinates.

getDistance — Computes the distance between two specified graph nodes in terms of some arbitrary distance measure, which may or may not correspond to world coordinates. Generally, an implementation should use the simplest calculation that obeys two invariants. One is the step cost invariant for the associated graph agent (see below). The other requires that the sum of distances between all successive nodes within a sequence is never less than the distance between any two nodes from the same sequence.

`PolygonGrid` counts intervening nodes, i. e. the minimum number of movement steps between immediate neighbors when moving from the source node to the target node. `Subdivision` calculates the Euclidean distance in world coordinates.

getWorldLocation — Gets the world location of a specified graph node. `PolygonGrid` returns the center of the polygonal element that represents the graph node. `Subdivision` simply returns the input.

getWorldRegion — Gets the vertices, in world coordinates, of the polygonal region covered by a graph node. `PolygonGrid` returns the vertices of the polygonal element that represents the graph node. `Subdivision` requires that users manually assign regions to graph nodes, e. g. Voronoi regions if the subdivision was created from the corresponding Delaunay triangulation.

4.1.3 Moving Agents

The interface `GraphAgent<T>` represents a moving "agent," i. e. anything that can move from one graph node to another. The `AStar<T>` and `Coverage<T>` algorithms require an instance of this interface to determine valid movement steps, the cost of each step, and other data.

There is no default implementation for `GraphAgent<T>` as the behavior of moving agents is specific to each individual application. See below for tips on implementing this interface.

canMakeStep — Determines whether the agent can move from one specified graph node to another, which must be an immediate neighbor.

canOccupy — Determines whether the agent can *permanently* occupy the specified graph node, i. e. whether the agent's movement path can end on that node.

getStepCost — The cost for moving the agent from one specified graph node to another, which must be an immediate neighbor. The step cost can never be less than the `getDistance` result for the two nodes. Together with the invariant regarding distances within node sequences (see above), the step cost invariant allows the `AStar<T>` algorithm to use `getDistance` to establish lower bounds for possible path costs.

isNearTarget — Determines whether the specified node is close enough to the ultimate target node that pathfinding can successfully terminate. `canOccupy` must succeed as well.

For example, when moving units towards an attack target, reaching the target itself is unnecessary and usually even impossible. Instead, `isNearTarget` should succeed when the agent has reached a location from which it can attack the target.

relaxedRange — Indicates whether the agent's path cost limit is strict or relaxed. This option affects the `AStar<T>` and `Coverage<T>` algorithms, as described below.

4.2 A* Pathfinding Algorithm

`AStar<T>` defines the well-known A* pathfinding algorithm. The core of the Java implementation is based on the article *Basic A* Pathfinding Made Simple* by James Matthews, published in *AI Game Programming Wisdom,* Charles River Media 2002.

`AStar<T>` finds the cheapest path, in terms of the combined cost of all movement steps, from one specified graph node to another. The path is constructed as a tree of `PathNode<T>` objects which associate a graph node with the auxiliary data required by the algorithm.

Once pathfinding is complete, the final `PathNode<T>` of the cheapest path is stored in the `bestNode` property, and the path itself can be backtracked as a sequence of `parent` links. This is rather laborious, so the following properties expose the results in a more convenient way:

getLastNode — Finds the last node in the movement path whose path cost does not exceed the specified maximum cost, and for which the agent's `canOccupy` method succeeds. The last condition ensures that the result is valid as an intermediate stop in multi-turn movements.

nodes — A list of all graph nodes in the movement path, from source to target.

totalCost — The total cost of the entire movement path returned in `nodes`.

These properties are also grouped into a separate interface, `GraphPath<T>`, which `AStar<T>` implements. This interface was designed to represent graph paths without dependence on any particular pathfinding algorithm, but A* is the only algorithm available so far.

In the rest of this section we'll describe two options exposed by `AStar<T>` itself to customize pathfinding, and how `GraphAgent<T>` interacts with the pathfinding algorithms.

4.2.1 Limited Search Range

Set `AStar<T>.relativeLimit` to limit the search range during pathfinding. This may cause A* to generate suboptimal paths or even fail to find any path at all, but performance on large graphs will be greatly improved.

`relativeLimit` defaults to zero. A positive value limits all candidate paths to an elliptical area around the source and target node of the search. Any candidate nodes that would lead beyond this area are ignored. All distances are calculated using the graph's `getDistance` method.

`relativeLimit` determines the radii of the ellipse, relative to the distance of the source and target nodes. After a path search, the read-only property `absoluteLimit` holds the maximum number of movement steps that were considered for any candidate path.

4.2.2 Minimal World Distance

Set `AStar<T>.useWorldDistance` to eliminate zero-cost oscillations in the returned path. Such oscillations have no effect on the total path cost, which is guaranteed to be optimal, but might cause strangely "unnatural" unit movements.

This effect can occur on graphs such as `PolygonGrid` whose `getDistance` method does not use world coordinates but some more abstract measure (in this case, the number of movement steps) which may not assign the smallest path cost estimate to the visually most direct path. The effect is pronounced on square grids with diagonal neighbor connections: rather than moving in a direct line, an agent might "sidestep" to an adjacent row or column and then back again.

`useWorldDistance` defaults to `false`. The value `true` adds an extra comparison to decide between candidate nodes that have equal path costs. Rather than always selecting the first node that happens to be generated, A* also checks the world distance of each candidate node to the target location, and selects the node with the smallest distance.

4.2.3 Transient and Permanent Occupation

A* constructs movement paths from a sequence of individual movement steps between graph nodes that are immediate neighbors. For each step, we must ask two questions: can the moving agent make the step, and how much does it cost? The first question is answered by the two agent methods `canMakeStep` and `canOccupy`, the second by `getStepCost` (described below).

We use two methods to determine whether a movement step is possible because we want to distinguish between *transient occupation* and *permanent occupation*. `canMakeStep` performs

the fundamental test whether the agent can move between the specified nodes at all, i. e. whether the source-target step can be even a transient part of its movement path. A target node for which `canMakeStep` fails is never part of a path, unless we reach it by a different source node.

`canOccupy` represents an additional test whether the agent can end its movement path on the specified node, and thus "permanently" occupy the node for the time being. A* requires that `canOccupy` succeeds for the final path node, and also for any intermediate nodes returned by `getLastNode` since they may represent intermediate stops during multi-turn movements.

This distinction between transient and permanent occupation is common in traditional board games where pieces can jump over occupied squares but land only on free squares. War games might also relax stacking limits for tiles that units only pass through.

Occupying Intermediate Nodes

A* never calls `canOccupy` on the intermediate nodes of a path, only on the final node. This is the desired and necessary behavior. However, this can cause problems for multi-turn movements with a non-trivial `canOccupy` implementation. Because `canOccupy` has not been checked for intermediate nodes, `getLastNode` might return suboptimal nodes, or none at all, when invoked with less than the total path cost.

For this reason, you must check every `getLastNode` call for a valid result, and even if the result is valid your partial path might look rather strange. The best advice is to avoid implementations where this is a major issue. The second-best advice is to use `Coverage<T>` and heuristics to manually piece together valid movement paths through difficult environments.

4.2.4 Movement Step Costs

The total cost of a movement path found by `AStar<T>` and `Coverage<T>` is defined as the sum of the costs of all movement steps between consecutive path nodes. Step costs depend only on the moving agent and the two directly involved nodes, never on any other nodes in the same path. This assumption is fundamental since the A* algorithm constructs an optimal path from path fragments that were originally found as parts of different search paths.

The agent's integer function `getStepCost` determines the cost of one movement step from a specified graph node to one of its immediate neighbors. This cost must be positive and no less than the graph's `getDistance` result for all possible movement steps. `canMakeStep` (and also `canOccupy` for the final node of a path) is always called before `getStepCost` to ensure that the agent can enter the target node at all.

4.2.5 Relaxed Movement Range

The implementation of an agent's `relaxedRange` method determines whether the moving agent enjoys an *extended movement range.*

If `relaxedRange` is `false`, the maximum path cost supplied to A* limits the agent's range *before* a step is taken. If the cost of entering another location exceeds the remaining fraction

of the maximum cost, the agent cannot enter. Note that this might lead to situations where an agent cannot move at all because all surrounding nodes exceed the maximum path cost.

If `relaxedRange` is `true`, a movement path ends only *after* its total cost equals or exceeds the maximum path cost. As long as this has not happened, the agent can enter *any* neighboring node, regardless of the actual cost of this step. This means that the agent can always make at least one step in any direction, regardless of its cost.

Marking Nodes as Impassable

Assume you wish to prevent an agent from entering certain graph nodes, for example because they represent impassable terrain.

If `relaxedRange` is `false`, you could return very high step costs for the desired nodes in your `getStepCost` implementation. If the step costs exceed any maximum path cost supplied to the pathfinding algorithms, the agent cannot enter these nodes.

However, this trick no longer works if `relaxedRange` is `true`. In this case, your `canMakeStep` implementation must return `false` for the desired nodes to make them impassable.

4.3 Path Coverage Algorithm

`Coverage<T>` defines a path coverage algorithm whose results are compatible with `AStar<T>`. This algorithm finds all graph nodes that can be reached from a specified node within a given maximum path cost.

When running on the same `Graph<T>` and `GraphAgent<T>` instances, `Coverage<T>` produces exactly those target nodes for which A* would find a path, given the same or a lower maximum cost. `Coverage<T>` does not store the actual paths, however – you must run `AStar<T>` on any found target nodes for which you wish to obtain a movement path.

`Coverage<T>` uses the `GraphAgent<T>` interface in the same way as `AStar<T>`. Note that since all graph nodes found by `Coverage<T>` represent *end points* of possible A* movement paths, the agent's `canOccupy` implementation must succeed for all of them. Intermediate nodes of possible paths that do not allow permanent occupation will not appear in the result set.

4.4 Flood Fill Algorithm

`FloodFill<T>` defines a flood fill algorithm for arbitrary graphs that works like the eponymous function in paint programs. This algorithm finds all immediate neighbors of a specified graph node for which a given predicate succeeds, then recursively all neighbors of those neighbors and so on. The search ends when the graph is exhausted or the predicate fails for all remaining neighbors of the result nodes.

`FloodFill<T>` is essentially a simpler version of `Coverage<T>` that uses a boolean predicate instead of a full-fledged `GraphAgent<T>` instance. Therefore, its results are not necessarily related to any valid agent movements.

4.5 Visibility Algorithm

`Visibility<T>` defines a line-of-sight algorithm that operates on a graph's world coordinates. The algorithm requires a source node and a maximum world distance from that source, as well as a predicate that determines whether a specified graph node obstructs visibility.

Currently, occlusion is binary only – a given node is considered either completely opaque or completely transparent. A node's visibility is determined as follows:

1. The node is assigned a *tangential arc,* determined by drawing tangents from the location of the source node (as per `getWorldLocation`) to the extreme vertices of its polygonal world region (as per `getWorldRegion`).

2. The node is assigned a *source distance,* measured from the location of the source node to the nearest vertex of its polygonal world region.

3. The node is compared against all opaque nodes that are not completely obscured by other opaque nodes. If the node's tangential arc overlaps that of an opaque node with a smaller source distance, then the overlapping fraction is considered obscured.

4. The node is considered visible from the source exactly if a certain minimum fraction of its tangential arc remains visible after comparing it against all opaque nodes.

This fraction defaults to 1/3 but can be changed to any value between zero and one by setting `threshold`. Zero is equivalent to `Double.MIN_NORMAL`, i. e. a node is considered visible if even the slightest bit of its tangential arc remains unobscured. Conversely, a threshold of one requires that a visible node's tangential arc is not obscured anywhere.

The computed data for all visited nodes – tangential arc, visible fraction, and source distance – is available in the `nodeArcs` collection. Applications can use this information to fine-tune their own concept of node visibility.

Chapter 5

Subdivision Package

Package `org.kynosarges.tektosyne.subdivision` contains classes that represent a planar subdivision, i. e. any collection of line segments that intersect only at their end points, as a doubly-connected edge list (DCEL). This representation is memory-intensive but allows fast navigation through all elements of the subdivision.

Any planar graph with straight bounded edges can be represented as a `Subdivision`, and so `PolygonGrid` and `Voronoi` (see Chapter 3) provide conversions to this class. A dedicated interface maps the resulting `Subdivision` faces to elements of the original structure.

`Subdivision` — Provides a planar subdivision composed of straight bounded edges, vertices on the end points of edges, and faces formed by closed loops of edges. Pathfinding between vertices is supported by its `Graph<T>` implementation.

You can create a new `Subdivision` from a set of line segments or polygons, or by intersecting two existing instances. You can also add or remove individual edges, split edges in half, and move or delete individual vertices (along with their edges). These operations allows interactive editing of a `Subdivision`.

`SubdivisionEdge` — Provides one half-edge in a `Subdivision`. Half-edges are always paired with twin half-edges in the opposite direction to form one full edge of the planar subdivision, connecting two of its vertices.

`SubdivisionFace` — Provides one face in a `Subdivision`. Faces are polygons that may or may not enclose any area. Faces with a positive area may contain one or more "holes," i. e. interior faces. Every subdivision also contains one unbounded face that represents the entire two-dimensional plane and thus encloses all bounded faces as its "holes."

`SubdivisionSearch` — Provides a fast but memory-intensive search structure for a `Subdivision`. The `Subdivision` class itself provides slower brute-force searches that require no additional memory.

`SubdivisionElement` — Represents an arbitrary `Subdivision` element, i. e. one vertex, half-edge,

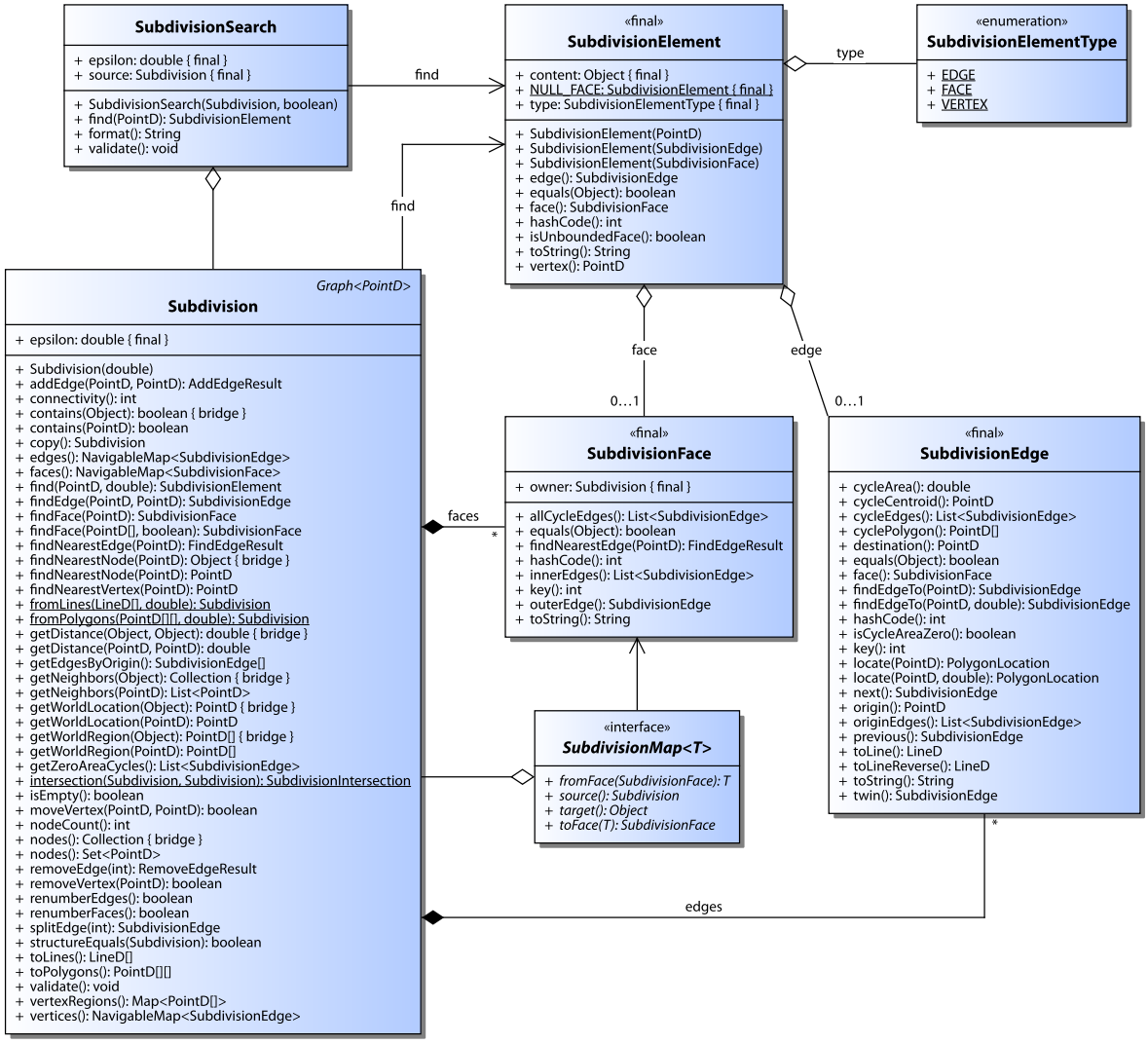

Figure 5.1: Subdivision Classes

or face. Returned by the general search algorithms `Subdivision.find` and `SubdivisionSearch.find`.

`SubdivisionElementType` — Specifies the type of a concrete `SubdivisionElement`.

`SubdivisionMap<T>` — Provides a bidirectional mapping between the faces of a `Subdivision` and the generically typed objects in some arbitrary collection.

Creating a `Subdivision` from another geometric structure automatically establishes a structural mapping. Applications might also define semantic mappings, e. g. correlating statistical information to faces that represent geographical areas.

See Figure 5.1 for an overview. Package `org.kynosarges.tektosyne.subdivision` defines four more public classes which are simple result containers for adding, finding, and removing edges, and for subdivision intersections. These classes are not very interesting and have been elided from the diagram for lack of space.

The overall implementation of `Subdivision` follows the DCEL structure outlined by Mark de Berg et al., *Computational Geometry,* Springer-Verlag 2008 (3rd ed.), but a few particular features are worth pointing out.

5.1 Edge and Face Keys

Every `SubdivisionEdge` and `SubdivisionFace` is identified by an integer key that is unique within its `Subdivision`. The `edges` and `faces` collections provide an O(ld n) access by key.

The ascending sequence of keys reflects the order in which the `Subdivision` was constructed. Keys are normally immutable but can be renumbered to plug "holes" in the sequence caused by dynamic edge deletion.

Strictly speaking, these keys are an unnecessary feature. References and/or indices would suffice to identify half-edges and faces. However, keys so enormously simplify unit testing and debugging that they are worth the extra memory.

5.2 Half-Edge Cycles

The half-edge cycle that contains a `SubdivisionEdge` constitutes a polygon which bounds the incident `SubdivisionFace` and may have a positive area. Faces may have both an outer and multiple inner boundaries, accessible through `outerEdge` and `innerEdges` respectively.

All half-edges in a cycle are linked by their `previous` and `next` pointers, but there also a number of dedicated methods to obtain information about linked half-edges. Most are defined on `SubdivisionEdge` itself.

`cycleArea` — Gets the area of the cycle polygon, which may be zero.

`cycleCentroid` — Gets the centroid of the cycle polygon, assuming its area is not zero.

`cycleEdges` — Gets a list of all half-edges in the cycle.

`cyclePolygon` — Gets all vertices of the cycle polygon.

`isCycleAreaZero` — Determines whether the area of the cycle polygon is zero, by examining `face` pointers rather than actually computing the area. This is faster and avoids rounding errors.

`originEdges` — Gets a list of all half-edges with the same `origin`. This list does not constitute a cycle and is not normally interesting, but used internally when adding or removing `Subdivision` edges or vertices.

`SubdivisionFace.allCycleEdges` — Gets a list of all half-edges in any cycle that form outer or inner boundaries of the `SubdivisionFace`.

The three methods returning lists are convenient rather than efficient, due to the overhead of creating a new `List<T>` on each call. Use explicit `do-while` loops over the linked half-edges for maximum performance.

5.3 Vertex Distances

The overridden `Graph<T>` method `getDistance` (see Chapter 4) returns the actual Euclidean distance between vertices, including the final square root, rather than the less expensive squared distance. This is necessary to avoid overestimating the total cost of compound paths within the subdivision.

Assume a straight path consisting of multiple edges so that the total Euclidean distance equals the sum of the lengths of all edges. If `getDistance` returned a squared Euclidean distance, the sum of all edge results would be smaller than the result for the two extreme vertices. This violates the invariant that the sum of the distances between all successive nodes within a sequence is never less than the distance between any two nodes from the same sequence.

5.4 Vertex Regions

The `vertexRegions` collection can associate `vertices` with user-defined polygonal regions. The overridden `Graph<T>` method `getWorldRegion` returns elements from this collection. All collection elements must be set explicitly, as the vertices of an arbitrary subdivision imply no meaningful regions.

As a typical example, you might create the `Subdivision` from a Delaunay triangulation and assign the Voronoi regions of its dual graph to the `vertexRegions` collection. This is what `VoronoiResults` does (see Section 3.6).

Chapter 6

Benchmark Results

This chapter shows a selection of benchmark results obtained with the `Tektosyne.Demo` application from the initial release of Tektosyne for Java, version 6.0.0. Many tests range over a variety of input set sizes, but we mostly just cover the largest tested set except where comparison to a smaller set was relevant.

The computer system was an Alienware Andromeda X51 R3 with one Intel Core i7 6700K CPU (4 GHz) and 16 GB RAM (dual-channel, 2133 MHz). The software environment consisted of Windows 10 (64 bit), Java SE 8u112, and .NET Framework 4.6.2.

The latter was used for comparison testing against the final release of Tektosyne for .NET, version 5.6.3. Tests and algorithms were identical except where otherwise noted. All numbers are averages of three consecutive benchmark runs.

Important! All Java tests used the 64-bit Server VM. This is unfortunately *not* the default on Windows – see Java Client VM for details. Please do make sure to run the 64-bit Server VM! Comparison testing with the 32-bit Client VM (the obsolete Windows default) showed a general slowdown by a factor of two, making Java as slow or slower than .NET.

6.1 Point Collections

These tests cover the `QuadTree` collection described in Chapter 2 as compared to non-specialized collections. For Java, that is `PointDComparator.findRange` on a standard `TreeMap<PointD, V>`. The .NET version included a custom `BraidedTree` collection which I did not bring forward as Java's `TreeMap` proved equivalent in both functionality and performance.

	Java		.NET	
	TreeMap	QuadTree	BraidedTree	QuadTree
Add	14.18	9.41	23.98	11.55
Iterate	1.22	1.58	1.60	2.19
Search	12.23	5.77	15.33	5.80
Range	56.96	11.82	61.27	22.84
Remove	12.28	6.92	16.53	7.14

Times are microsecond averages for 60,000 random points. "Search" uses single query points. "Range" performs 500 range searches over 0.56% of the total search space, $10,000^2$.

6.2 Geometric Algorithms

These tests cover some basic geometric algorithms described in Chapter 3.

	Java		.NET	
	$\varepsilon = 0$	$\varepsilon = 1^{-10}$	$\varepsilon = 0$	$\varepsilon = 1^{-10}$
Line Intersection	15.91	18.70	40.28	42.87
Point in Polygon	20.14	28.73	24.24	26.44

Times are nanosecond averages for random objects, using both exact coordinate comparisons and the indicated comparison epsilon. "Line Intersection" runs on random line pairs, "Point in Polygon" runs on random polygons with 3–60 vertices.

The Java implementation of the line pair intersection algorithm was changed compared to the .NET version, so the massively different run times are not directly comparable in this case. See the `ReadMe` file for details.

	Java	.NET
Convex Hull	6.83	11.39
Voronoi	49.57	78.29
Delaunay	42.10	68.63
Nearest Point (unsorted)	43.01	59.85
Nearest Point (sorted)	0.71	0.88

Times are microsecond averages for sets of 120 random points, except for "Nearest Point" which both use 12,000 random points. "Voronoi" runs `Voronoi.findAll` and "Delaunay" runs `Voronoi.findDelaunay`. "Unsorted" runs `GeoUtils.nearestPoint` and "sorted" runs `PointDComparator.findNearest`, in each case on an `ArrayList<PointD>`.

6.3 Multi-Line Intersection

These tests cover both algorithms of the `MultiLineIntersection` class, see Section 3.3.

	Java		.NET	
	$n = 20$	$n = 120$	$n = 20$	$n = 120$
Sweep Line (0)	8.67	40.00	14.28	69.78
Sweep Line (n)	12.67	75.00	22.38	106.16
Sweep Line ($n^2/4$)	57.00	3,046.33	84.33	5,188.67
Brute Force (0)	2.00	77.67	7.81	241.65
Brute Force (n)	4.33	96.67	12.35	258.66
Brute Force ($n^2/4$)	19.00	1,003.00	24.95	1,851.39

Times are microsecond averages for sets of n random lines. "Sweep Line" runs `find` and "Brute Force" runs `findSimple`, using exact coordinate comparisons. The numbers in parentheses indicate the number of intersections relative to the total line count. Brute force is always considerably faster except when a large number of lines produce relatively few intersections.

6.4 Subdivision Algorithms

These tests cover planar subdivision intersections and searches, see Chapter 5.

Intersection	Java	.NET
0% – 100%	143.33	255.30
10% – 90%	329.58	569.31
50% – 50%	675.83	1,017.35
90% – 10%	430.42	604.35
100% – 0%	227.92	327.13

Times are microsecond averages for calling `Subdivision.intersection` on two subdivisions with a combined 240 edges, distributed between the first and second subdivision as indicated.

Intersecting two subdivisions with roughly the same edge count is much slower than intersecting a sparse with a dense subdivision, or vice versa.

Subdivision Search	Java		.NET	
	Grid	Lines	Grid	Lines
Brute Force	12.53	25.31	13.72	29.84
Ordered Structure	5.41	0.20	7.94	0.36
Randomized Structure	0.25	0.14	0.39	0.34

Times are microsecond averages for two different subdivisions: "Grid" is a `PolygonGrid` of squares with 1860 strictly regular and ordered edges, "Lines" is a set of 1200 random line segments that do not intersect except at their end points.

"Brute Force" runs `Subdivision.find`. "Ordered Structure" runs `SubdivisionSearch.find` on a search structure that was created directly from the edge sequence of the supplied subdivision. "Randomized Structure" does the same but randomly shuffles the subdivision's edges before inserting them into the search structure.

While `SubdivisionSearch` is always much faster than a brute force search, it is very sensitive to ordered edges and only achieves peak performance when they are randomized. Interestingly, brute force prefers ordered edges but still lags far behind the search structure.

6.5 Comments on Java vs .NET

Java generally performed at least as good as .NET, and sometimes so much better that loop counts had to be multiplied for similar execution times. Discounting the line pair intersection case where the underlying implementation has changed, many tests show a speedup of 50–100% and sometimes more.

This is especially remarkable as Java currently does not support custom value types, or even numerical primitives as generic type arguments. Nevertheless Tektosyne's geometric types seem to perform better as Java classes than as .NET structs, and changing `QuadTree`'s internal node collection to a hard-coded `int` hashtable produced negligible gains.

Consequently I refrained from using any hard-coded primitive collections in Tektosyne, or other tricks to avoid classes and objects. Once value types and generics over primitives are available in Java 10 I'll retest and see how much difference they will make in practice.

6.5.1 Caveats

The Windows .NET Framework on which I tested has slipped into maintenance mode as Microsoft now focuses its efforts on cross-platform .NET Core. This is a new design which also includes a revised JIT compiler. Possibly Tektosyne for .NET would perform better there. I did not try, and in any case it would require some refactoring as the old Tektosyne library had numerous Windows .NET dependencies that are unavailable in .NET Core.

Second, the Java algorithms are equivalent but not perfectly identical to their .NET counterparts. Some bugs were fixed, and some changes were required to accommodate differences in language and standard library. However, with the notable exception of line pair intersection, I don't believe these changes could have caused the observed speedup – especially as I had spent a lot of time optimizing the .NET version but not the Java port.

Lastly, using full-size objects instead of value types or (in collections) primitives does have the inevitable disadvantage of increased memory consumption, and of additional indirections when retrieving objects from memory. For very large amounts of data, this might cause the .NET version to outperform Java (until Java 10 arrives anyway).