# Структура приложения

Разработка приложения проводилась на языке Java, ограничиваясь возможностями шестой версии. Целевым Андроид API было выбрано API версии 19 (Android KITKAT). Архитектура приложения следует принципам объектно ориентированного программирования. Основные классы приложения имеют сложную структуру наследования.

## Параметрически заданные кривые

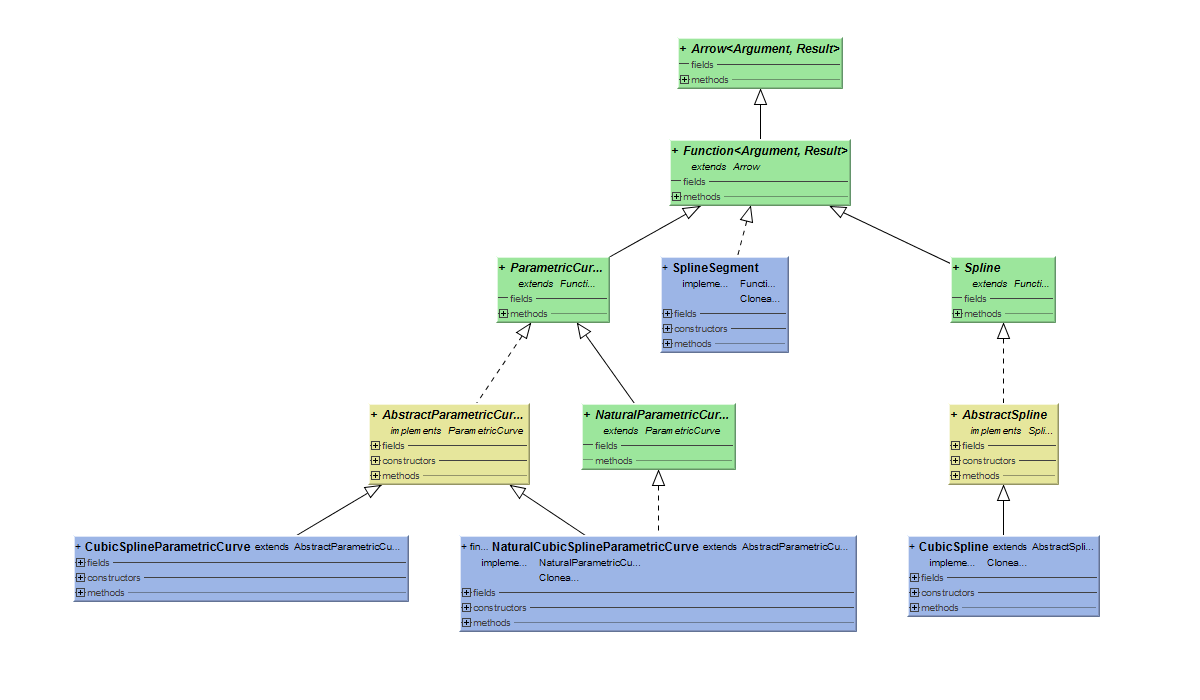


Рисунок 1 - UML диаграма классов параметрически заданных кривых

### Arrow<Argument, Result>

Поскольку в языке Java 6 нет встроенной поддержки лямбда-выражений, приложение получило свою реализацию интерфейсов функций. Во главе иерархии находится параметризированный интерфейс **Arrow**, служащий для представления объекта, умеющего по параметру функции типа **Argument** возвращать результат типа **Result**. Следует отметить, что не правильно считать **Arrow** функцией одной переменной, либо функцией, возвращающей одно значение, поскольку не смотря на то, что функция имеет один параметр и возвращает один результат, **Argument** и **Result** могут представлять собой обёртки, содержащие несколько значений.

Данный интерфейс в силу своей гибкости и простоты используется во многих методах классов в качестве типа входного параметра.

### Function<Argument, Result>

Данный интерфейс является в приложении представлением математической функции. Объект класса, реализующего данный интерфейс, в дополнение к значению имеет и производные.

### SplineSegment implements Function<Double, Double>

Класс **SplineSegment** служит для представления звена сплайновой кривой. Полями этого класса являются степень полинома, массив коэффициентов, и границы, на которых данный сегмент следует применять. Реализация интерфейса **Function** требует реализации методов вычисления значения и производной.

### Иерархия Spline <- AbstractSpline <- CubicSpline

Данная иерархия классов реализует принципы построения сплайновых кривых. Основной интерфейс сплайна объявлен в интерфейсе **Spline**. В **AbstractSpline** реализован механизм проксирования вызовов функций получения значений и производных в соответствующие сегменты сплайна. Класс **CubicSpline** отвечает за контроль порядка сплайна и недопускает добавление сегментов степени выше третьей.

### ParametricCurve extends Function<Double, Point2D>

Класс **ParametricCurve** является базовым интерфейсом в иерархии классов параметрически заданных кривых. Он наследуется от интерфнйса **Function<Double, Point2D>**, что означает, что классы, реализующие данный интерфейс, должны иметь значение и производные в вещественных точках. Класс **Point2D** является представлением точки на двумерной плоскости и хранит две координаты.

### NaturalParametricCurve extents ParametricCurve

Данный интерфейс не вносит дополнительной функциональности. Его наличие обусловлено необходимостью отделения естественно параметризованных кривых в отдельный класс объектов, так как они обладают некоторыми отличительными особенностями. Данный тип гарантирует, что в качестве агрумента будет передаваться естественно параметризованная кривая.

### AbstractParametricCurve implements ParametricCurve

На данном уровне иерархии наследования реализуются методы, унаследованные **ParametricCurve** от **Function**. При этом предполагается, что за значение координаты в точке будет отвечать функция одной переменной (объект, реализующий интерфейс **Function<Double, Double>**).

### CubicSplineParametricCurve и NaturalSplineParametricCurve

Данные классы конкретизируют функции, использующиеся для получения значений координат, поскольку многие из использованных подходов при получении объекта подписи опираются именно на сплайновую природу параметрически заданной функции.

## Сравнение подписей

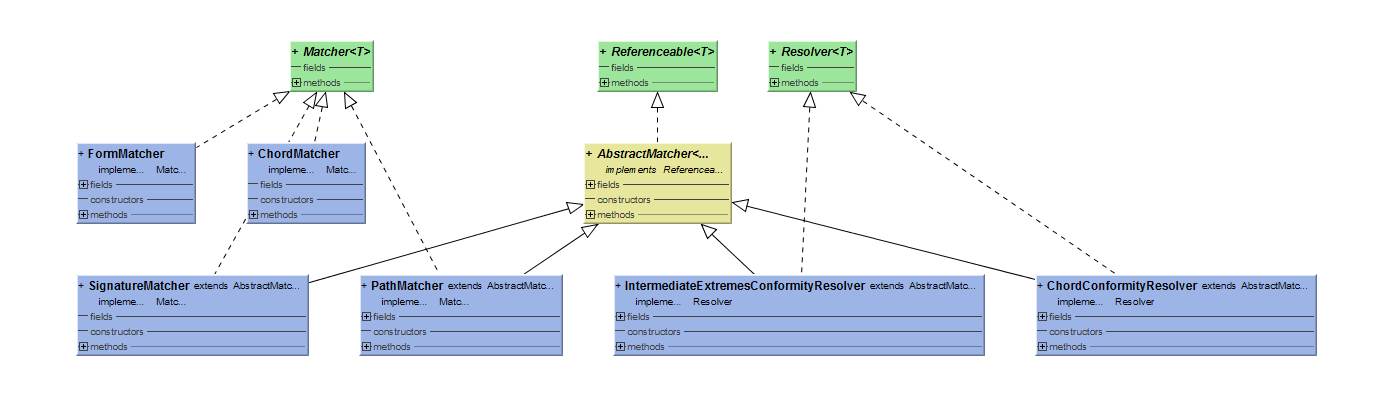


Рисунок 2 - Диаграма классов иерархии Matcher и Resolver

Алгоритм сравнения подписей разбит на несколько шагов. За каждый шаг отвечает отдельный класс, имеющий свои параметры сопоставления. Результатом сопоставления является класс, содержащий в себе результаты сопоставления, а также параметры, при которых данный результат был получен.

### SignatureMatcher

Данный класс является обёрткой для всех остальных сопоставлений. В нём происходит анализ результатов и построение финального ответа.

Первый шаг сопоставления – сопоставление траекторий. Класс **SignatureMatcher** инициализирует экземпляр класса **PathMatcher** и передаёт ему объект подписи для анализа структуры траектории. Результатом сопоставления **PathMatcher** являются оптимальная метрика и карта сопоставления экстремальных точек подписей друг другу. В случае, когда структура сравниваемых подписей совпадает, **SignatureMatcher** инициализирует экземпляр класса **FormMatcher** - следующий шаг сопоставления, осуществляющий анализ формы траектории. Класс **FormMatcher** возвращает информацию о среднем и максимальном расхождении углов касательных векторов, а также индекс хорды, на которой расхождение максимально. Все эти данные и возвращаются **SignatureMatcher** в качестве ответа.

### PathMatcher

На класс **PathMatcher** возложена задача сопоставления структуры траектории и построение карты соответствия экстремальных точек сопоставляемых подписей. Для первоначального сопоставления структуры происходит построение допустимого набора хорд. Задача сопоставления хорд делегируется классу **ChordConformityResolver.** Из полученной в результате сопоставления карты соответствия хорд формируется карта соответствия вертикальных экстремумов. После этого для соответствующих пар экстремумов, уже находящихся в карте, происходит отбор горизонтальных экстремумов, заключённых между ними. Для наборов этих экстремумов также запускается сопоставление, делегируемое классу **IntermediateConformityResolver.** Полученные соответствия добавляются в карту к вертикальным экстремумам, а процесс сопоствления повторяется для экстремумов по кривизне.

В результате формируется карта соответствия экстремумов сопоставляемых подписей, а также метрика, с которой было получено сопоставление вертикальных экстремумов.

### ChordConformityResolver и IntermediateConformityResolver

Данный класс отвечает за нахождение оптимального соответствия между двумя наборами хорд. Для начала процесса оптимизации необходимо построить таблицу штрафов. Этим занимается параметризированный Builder-класс **MetricsTableBuilder**. Сгенерированная с его помощью таблица штрафов (объект класса **MetricsTable**) позволяет произвести оптимизацию и получить список ячеек, через которые проходит оптимальный путь. Затем по списку индексов строится карта соответствий хорд.

Аналогично работает **IntermediateConformityResolver**, но с другими метриками.

### FormMatcher

Все рассмотренные выше классы отвечали за сопоставление структуры траектории. Класс **FormMatcher** отвечает за следуюзий шаг сопоставления - анализ формы траектории.

Последовательно получаем соответствующиепары соседних точек и рассчитываем угол между их касательными векторами. Считаем максимальное расхождение значений углов и среднее. Эти величины и возвращаются в **SignatureMatcher**.