Designbeschreibung der dynamischen Geometriesoftware GeoFuxx 1.1

Andy Stock

20. August 2004

Inhaltsverzeichnis

1 Einführung

Die erste Version der dynamischen Geometriesoftware GeoFuxx entstand im Rahmen des von Prof. Dr. Gräbe betreuten Softwaretechnik-Praktikums an der Universität Leipzig im Sommersemester 2003. Sie wurde von der Projektgruppe geo-08 (A. Aderhold, J. Bachmann, T. Chiacos, Y. Metzner, M. Todorov und A. Vollmer) entwickelt.

Diese Urversion von GeoFuxx zeichnete sich hauptsächlich durch ihre ansprechende Benutzeroberfläche aus. Ich habe selbige übernommen und um beispielsweise eine Vorschaufunktion erweitert. Ebenso habe ich die zeichenflächenbezogene Ereignisverarbeitung sowie das zugrunde liegende geometrische Modell durch eine eigene, strukturiertere und flexiblere Lösung ersetzt. In der aktuellen Version sind nun auch Schnittobjekte verfügbar.

Dieses Dokument soll eine Übersicht über die Struktur und Funktionsweise eben jener neuen Programmteile geben.

2 Aufbau der Anwendung

2.1 Schichtenmodell

Ein wesentliches Merkmal der Architektur von GeoFuxx liegt in der strikten Trennung von graphischer Oberfläche (einschließlich Ereignisverarbeitung) und geometrischem Modell. Kommunikation zwischen diesen findet ausschließlich über die Konstruktionen statt. Die Klasse `GeoConstruction` kapselt das gesamte Modell. Ihre Methoden werden in den Ereignisverarbeitungsroutinen aufgerufen. Sie nimmt Änderungen - das Hinzufügen oder Bewegen von Objekten - vor und liefert zur graphischen Darstellung notwendige Informationen.

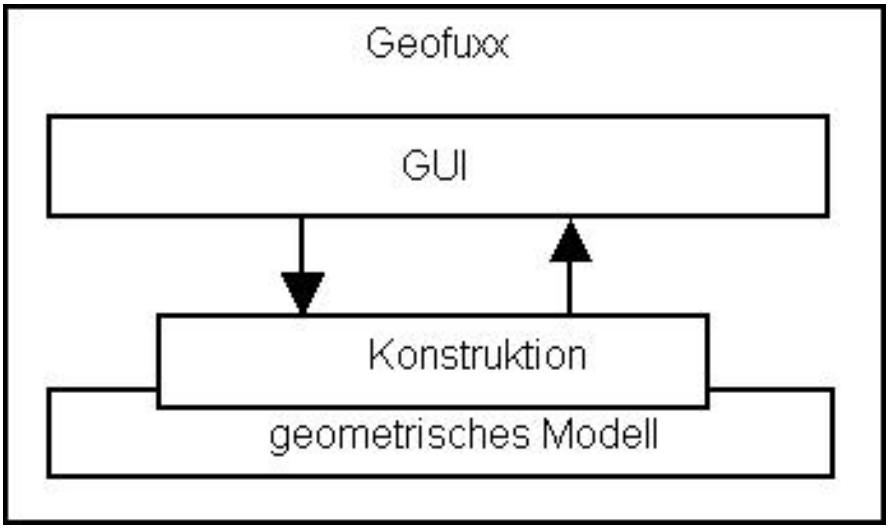

Abbildung 1: Logische Schichten von GeoFuxx

2.2 Paketstruktur

Das Paket `geofuchs.view` beinhaltet die Klassen für die graphische Benutzeroberfläche, also u. a. das Haupt- und die Kindfenster sowie die Zeichenflächen.

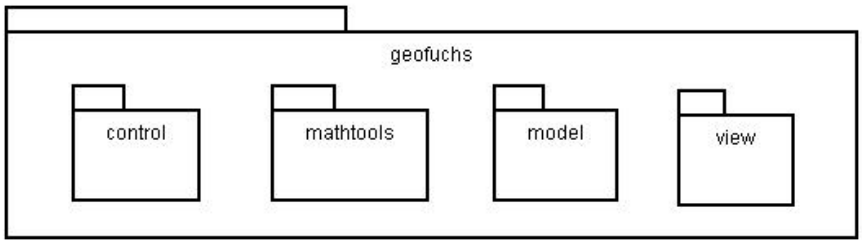

Abbildung 2: Paketstruktur von GeoFuxx

Im Paket `geofuchs.view` finden sich die Werkzeug-Klassen sowie zur Kommunikation zwischen den Anwendungsschichten bestimmte Zeichenketten.

Das geometrische Modell, vor allem die Klassen für die Konstruktionen, Objekte und Konfigurationen, finden sich im Paket `geofuchs.model`, während `geofuchs.mathtools` Klassen mit mathematischen Hilfsfunktionen beinhaltet.

3 Das geometrische Modell

3.1 Objekte und Konfigurationen

Das geometrische Modell unterscheidet zwischen zwei grundlegenden Konzepten. Das erste sind die *geometrischen Objekte*. Dieser Begriff wird in seiner intuitiven Bedeutung verwendet. So sind beispielsweise Punkte, Geraden, Kreise, Kegelschnitte und Kugeln geometrische Objekte. Auch deren programminternen Besprechungen werde ich mit diesem Begriff bezeichnen. An diese werden weitere Bedingungen gestellt, die im Abschnitt 'Schnittstellen für Objekte' beschrieben werden.

> **Kommentar:** (hgg) Das einzige primitive gO sollte ein „Punkt“ sein und alle anderen gO sind davon abgeleitet. Das halten die meisten DGS so. Insbesondere kann man nur Punkte dynamisieren sowie nur von Punkten die Spur aufzeichnen.

Das zweite wichtige Konzept des geometrischen Modells sind die Konfigurationen. Eine *Konfiguration* ist eine Relation, die geometrische Objekte zueinander in Beziehung setzt. Ein Beispiel für eine Konfiguration ist die dreistellige Relation 'Mittelpunkt'.

> **Kommentar:** (hgg) In meinem Geometrie-Skript (gs) ist eine Konfiguration eine konstruktiv erzeugbare Folge von gO's, in welcher die Abhängigkeiten einen DAG bilden. Ein Mittelpunkt ist in diesem Sinne also keine Relation, sondern eine Funktion. Das zu unterscheiden ist wesentlich, da der relationale Aspekt in Beweisschematas vom Gleichungstyp vorkommt, deren Beschreibung auf (konstruktiven) Konfigurationen beruht, in denen zusätzliche Abhängigkeiten zwischen den zunächst freien Parametern bestehen.
>
> Außerdem ist hier terminologisch die Unterscheidung zwischen der Konfiguration als Vorschrift (gs: allgemeine Konfiguration) und einer mit `calculate` berechneten konkreten Ausprägung der Konfiguration (gs: spezielle Konfiguration) wahrscheinlich sinnvoll.

Der Begriff des Freiheitsgrades wird bei der Definition der semantischen Gleichheit von Konfigurationen benötigt: Der *Freiheitsgrad* eines zweidimensionalen geometrischen Objektes ist die Anzahl der unabhängigen Parameter, von denen seine Koordinaten abhängen. Somit ist der Freiheitsgrad von freien Punkten gleich zwei, von Gleitern gleich eins und für alle anderen bisher implementierten Objekte bzw. Konfigurationen gleich null.

> **Kommentar:** (hgg) Die Definition des Begriffs Freiheitsgrad habe ich präzisiert. Danach ist Freiheitsgrad ein Begriff einer Konfiguration und nicht eines einzelnen gO.

3.2 Schnittstellen für Objekte

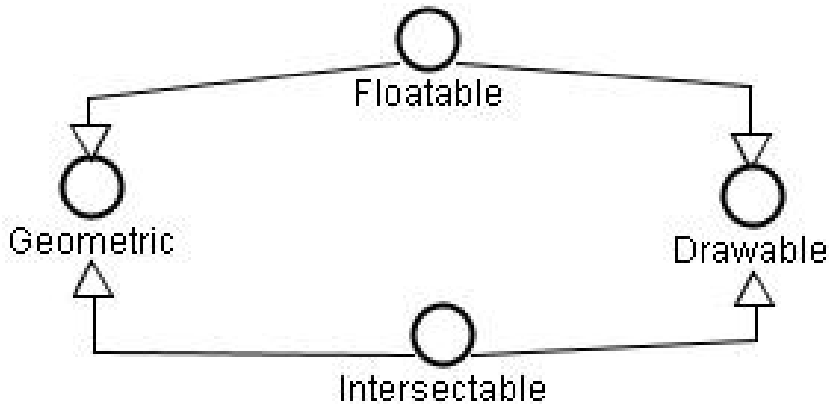

Abbildung 3: Schnittstellen für Objekte

Alle geometrischen Objekte müssen die Schnittstelle `model.Geometric` implementieren. Jedes solche Objekt hat einen Namen, welcher zu seiner Identifikation dient und dementsprechend eindeutig sein muss. Die Methode `getName()` liefert diesen. Mittels `calculate()` wird die Position des Objektes aufgrund seiner Konfiguration gesetzt. Sofern eine Konfiguration vorhanden ist, wird dieser Aufruf stets an selbige weitergeleitet. Der Methode `boolean isAt(Point2D.Double, double)` wird ein Punkt sowie ein Toleranzparameter übergeben. Sie gibt genau dann `true` zurück, wenn der Abstand des Objektes zum spezifizierten Punkt kleiner oder gleich der Quadratwurzel der gegebenen Toleranz ist. Letztlich ist noch `boolean equals(Geometric)` zu erwähnen. Diese Methode bezieht sich auf die semantische Gleichheit der Objekte: Zwei geometrische Objekte sind genau dann semantisch gleich, wenn beide in einer Konfiguration stehen und diese Konfigurationen semantisch gleich sind. Der Begriff der semantischen Gleichheit für Konfigurationen wird im Abschnitt 'Klassen für Konfigurationen' geklärt.

> **Kommentar:** (hgg) Hier wäre jeweils eine tabellarische Kurzübersicht über die semantische Bedeutung der Methoden der Schnittstelle sinnvoll, der eine Beschreibung der dabei verwendeten Begriffe vorangeht, etwa so

`boolean isAt(Point2D.Double p, double tolerance)`	Kurzbeschreibung
`String getName()`	
`String getType()`	...

Alle Objekte, die auf der grafischen Oberfläche gezeichnet werden sollen, müssen die Schnittstelle `model.Drawable` implementieren. Zentraler Punkt davon ist die Methode `draw` zum Zeichnen des Objektes. Zeichenbare Objekte haben einen Zeichenbarkeits- und einen Highlight-Zustand, auf welche durch entsprechende Get- und Set-Methoden zugegriffen werden kann. Ersterer gibt an, ob das Objekt überhaupt gezeichnet werden kann. So ist z. B. eine Gerade, die durch zwei Punkte mit den gleichen Koordinaten definiert ist, nicht zeichenbar. Der Highlight-Zustand bewirkt gegebenfalls eine graphische Hervorhebung durch die `draw`-Methode.

> **Kommentar:** Warum nicht `setHighlighted(boolean isHighlighted)`? Wäre hier nicht ein allgemeineres Auszeichnungskonzept mit mehr als zwei Zuständen sinnvoll?

Alle Objekte, auf welchen Gleiter definiert werden können, implementieren die Schnittstelle `.model.Floatable`, die sowohl `Drawable` als auch `Geometric` erweitert. Die einzige Methode `projectFloater` erhält zweidimensionale Koordinaten (hgg: Point2D-Objekt?). Sie gibt die Koordinaten des entsprechenden auf das geometrische Objekt projizierten Punktes zurück. Damit können Gleiter auf geometrischen, zeichenbaren Objekten unabhängig von deren Typ durch eine einzige Konfiguration realisiert werden.

Objekte, die mit anderen Objekten geschnitten werden sollen, müssen die Schnittstelle `model.Intersectable` implementieren. Auch diese erweitert `Drawable` und `Geometric` und fordert zusätzlich eine Methode `getIntersectionSet`, welche ein anderes `Intersectable`-Objekt entgegennimmt und ein Feld mit den Koordinaten der Schnittpunkte zurückgibt.

3.3 Klassen für Objekte

In der aktuellen Version unterstützt GeoFuxx ausschließlich Objekte, die sowohl geometrisch als auch zeichenbar sind. Gemeinsame Superklasse aller solchen Objekte ist die abstrakte Klasse `model.DrawableGeoObject`. Sie implementiert lediglich Grundfunktionalität wie z. B. die Zugriffsmethode für den Namen.

Davon abgeleitet sind die Klassen `DrawableGeoPoint`, `DrawableGeoLine` und `DrawableGeoCircle`. Alle diese Klassen verfügen über eine Referenz auf eine Konfiguration. Diese darf bisher nur bei `DrawableGeoPoint` gleich `null` sein. In diesem Fall ist der Punkt unabhängig von anderen Objekten und kann vom Nutzer frei bewegt werden.

Instanzen von `DrawableGeoPoint` haben eine Referenz auf ein Objekt vom Typ `Point2DTransformable`. Dabei handelt es sich um die Koordinaten des Punktes (Genaueres findet sich im Abschnitt 'Änderbarkeit der zugrunde liegenden Mathematik'). Auf Grundlage dieser wird der Punkt gezeichnet, mit der Methode `move` können die Koordinaten gesetzt werden.

> **Kommentar:** (hgg) Hier ist die Unterscheidung zwischen Weltkoordinaten und Zeichenblattkoordinaten zu treffen. Es muss noch ein Glossar her!

Die Klasse `DrawableGeoLine` repräsentiert eine Gerade. Sie beinhaltet zwei zweidimensionale Vektoren, auf deren Grundlage die `draw`- und `isAt`-Methoden arbeiten. Die Referenzen

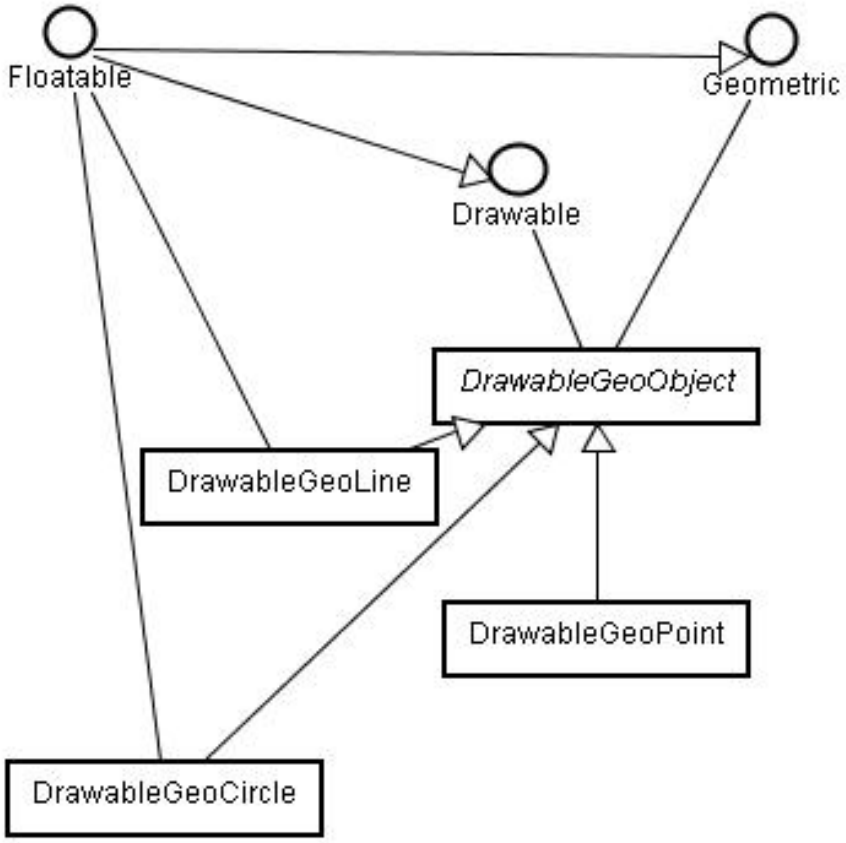

Abbildung 4: Klassen für Objekte

auf die definierenden geometrischen Objekte selbst finden sich dagegen in der obligatorischen Konfiguration, welche sich darum kümmert, die genannten Zeichenkoordinaten aktuell zu halten. Diese können mit der Methode `setPoints` gesetzt werden.

Kommentar: (hgg) Warum diese Unterscheidung?

Zu erwähnen sind außerdem die Hilfsmethoden `getPointOnParallelThrough` und `getPointOnPlumbThrough`, welche die Koordinaten eines Punktes als Parameter erhalten und die Koordinaten eines anderen Punktes auf der Parallele bzw. Senkrechten der Gerade durch ersteren Punkt zurückgeben.

Kommentar: (hgg) orthoLine statt Plumb. Über diese Punkte müsste noch einmal nachgedacht werden. Parallele zu g(AB) durch C: 4. Punkt des Parallelogramms ABCD. Lot aus einem Punkt auf g(AB): Zweiter Punkt ist der Schnittpunkt mit g. Senkrechte in C auf g(AB): ?? Wie konstruiert man die mit Zirkel und Lineal?

`DrawableGeoCircle` enthält die Koordinaten des Mittelpunktes und einen Radius. Ebenso wie bei Geraden werden diese durch die notwendigerweise vorhande Konfiguration mittels der Methode `setPosition` gesetzt.

Kommentar: Wie im GeoProver besser Mittelpunkt und ein Punkt auf der Peripherie.

3.4 Klassen für Konfigurationen

Die Konfigurationen stellen den Zusammenhang zwischen den Objekten her. Dazu hat jede Konfiguration Referenzen auf die definierenden sowie die davon abhängigen Objekte.

Jede Konfiguration muss die Schnittstelle `GeoConfiguration` implementieren. Dort werden vier Methoden vereinbart: `createObject` erzeugt ein geometrisches Objekt und ordnet ihm die Konfiguration zu. So erzeugt beispielsweise die `createObject`-Methode einer Mittelpunkt-Konfiguration eine Instanz von `DrawableGeoPoint`. Die Methode `calculate` dagegen setzt die Parameter des dazugehörigen abhängigen Objektes aufgrund der Positionen der definierenden Objekte, während `getDescription` eine textuelle Beschreibung liefert. Die Gleichheit von Konfigurationen kann mittels `equals` überprüft werden. Dabei gilt folgende Bedingung:

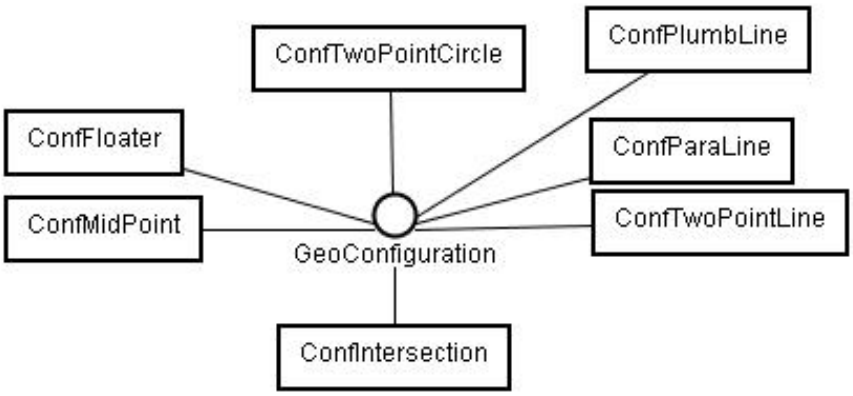

Abbildung 5: Klassen für Konfigurationen.

Zwei Konfigurationen sind genau dann gleich, wenn sie Instanzen der gleichen Klasse und durch die gleichen Objekte definiert sind und ihr Freiheitsgrad null ist.

3.5 Änderbarkeit der zugrunde liegenden Mathematik

Die momentan implementierten mathematischen Berechnungen sind lediglich als Testimplementierung zu verstehen und sollen langfristig ersetzt werden. Um dies zu vereinfachen, werden die Koordinaten von geometrischen Punkten momentan durch die Klasse `Coords2D` dargestellt. Diese implementiert die Schnittstelle `Point2DTransformable`, welche Methoden zur Transformation in einfache zweidimensionale Koordinaten fordert. Im gesamten Programm werden Koordinaten als Instanzen dieser Schnittstelle behandelt. Damit können andere – z. B. homogene – Koordinaten einfacher eingeführt werden. Implementiert die dazugehörige Klasse die genannte Schnittstelle, ist das Programm weiterhin voll funktionsfähig und die dahinter stehende Mathematik (welche ausgehend von den `calculate`-Methoden der Konfigurationen aufgefunden werden kann) lässt sich schrittweise ändern und testen.

3.6 Die Konstruktion

Die Konstruktion ist eine Menge von geometrischen Objekten (und den dazugehörigen Konfigurationen). Die Klasse `GeoConstruction` verwaltet die Objekte in einer privaten Liste. Sie bietet Methoden wie beispielsweise `addTwoPointLine` an, die zum Hinzufügen entsprechender

Objekte dienen. Die definierenden Objekte müssen zuvor mittels `selectObject` anhand ihres Namens ausgewählt werden. Darüber hinaus beinhaltet diese Klasse weitere Funktionalität zur Interaktion mit der graphischen Oberfläche. Diese wird im folgenden Kapitel beschrieben.

4 Interaktion mit dem Nutzer

4.1 Auswahl eines Tools

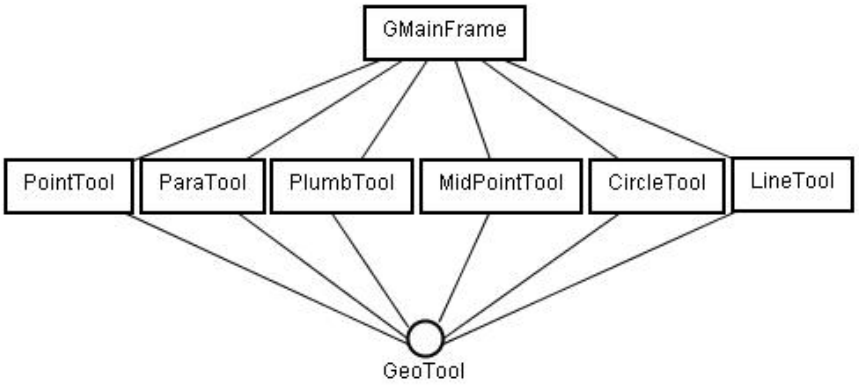

Abbildung 6: GeoFuxx-Tools

Für die Verarbeitung der Werkzeugauswahl ist das Hauptfenster `GMainFrame` zuständig. Dazu hat `GMainFrame` eine Referenz auf eine Instanz einer Klasse, die die Schnittstelle `GeoTool` implementiert. Diese fordert nur eine einzige Methode `getType`, die eine Zeichenkette zur Bezeichnung des Werkzeugtyps zurückgibt. Die entsprechenden (auch andernorts genutzten) Zeichenketten finden sich als statische Konstanten in der Klasse `GeoStrings`.

Bei der Auswahl eines Tools mittels Button oder Menü wird eine neue Instanz der entsprechenden Tool-Klasse erzeugt und die Werkzeug-Referenz des Hauptfenster darauf gesetzt. Dabei ruft der Konstruktor zunächst die statische Methode `toolChanged` der Konstruktion auf. Sie erhält eine das Tool bezeichnende Zeichenkette als Parameter. Anhand dieser kann später beim Erzeugen der Vorschau die Art des zu erschaffenden geometrischen Objektes festgestellt werden. Dann ruft der Konstruktor die ebenfalls statischen Methoden `expectObject` und `setExpectedObjectNr` auf. Ersterer werden eine den Objekttyp bezeichnende Zeichenkette und die maximal zulässige Anzahl von Objekten dieses Typs, zweiterer die Anzahl erwarteter Objekte übergeben. Zum hinzufügen einer Parallele werden beispielsweise je ein Punkt und eine Gerade erwartet.

```
public ParaTool()
  {
    GeoConstruction.toolChanged(GeoStrings.PARALINE);
    GeoConstruction.expectObject(GeoStrings.POINT,1);
    GeoConstruction.expectObject(GeoStrings.LINE,1);
    GeoConstruction.setExpectedObjectNr(2);
  }
```

Auf diese Art und Weise lassen sich natürlich nicht alle möglichen Eingabeerwartungen ausdrücken, da eine Disjunktion fehlt.

4.2 Mauseingaben

Die Zeichenfläche wird durch die Klasse `GCoordinateArea` realisiert. Diese hat eine Referenz auf die dazugehörige Konstruktion.

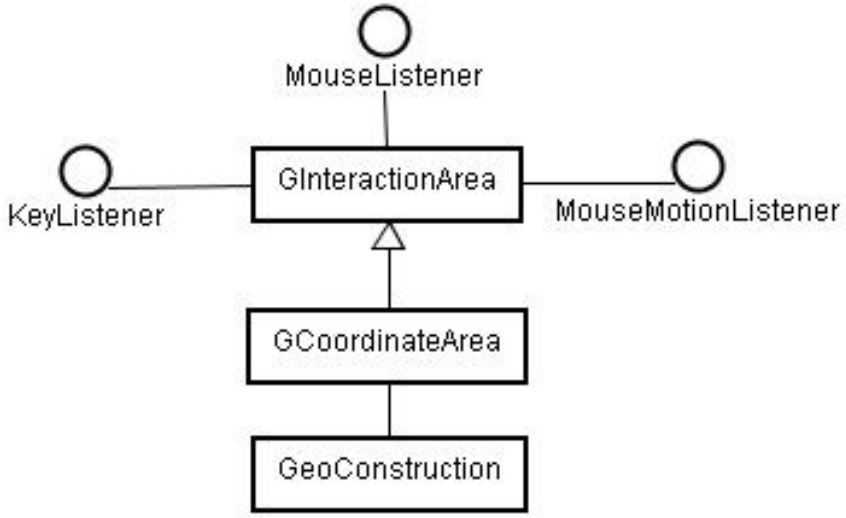

Abbildung 7: Die Zeichenfläche `GCoordinateArea`

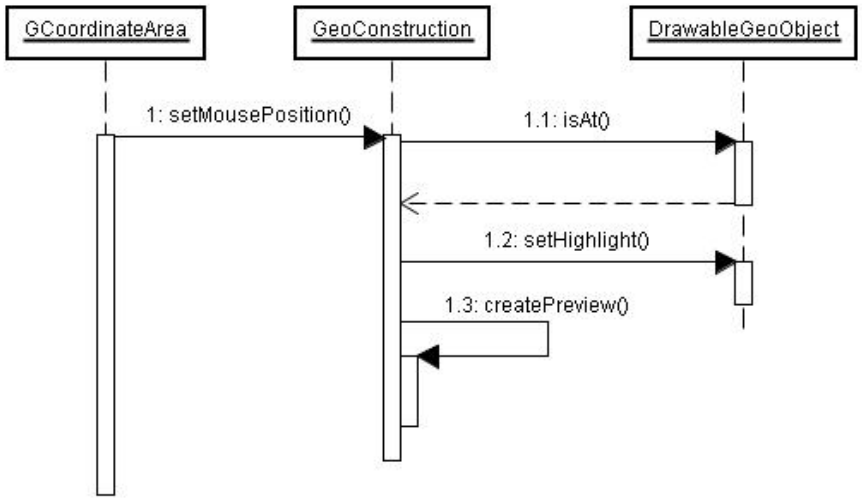

Abbildung 8: Mausbewegung

Die Mauslistener-Methode `mouseMoved` ruft die Methode `setMousePosition` auf und übergibt die in das Modellkoordinatensystem transformierte Mausposition. Die Konstruktion

prüft, welche Objekte sich dort befinden und welche davon einem erwarteten Typ entsprechen. Deren Highlight-Zustand wird gesetzt und damit eine Hervorhebung beim Zeichnen bewirkt. Findet sich an der angegeben Position nur ein passendes Objekt, wird sein Index in der Objektliste für die Vorschau gespeichert und versucht, ein Vorschau-Objekt zu erzeugen. Schließlich gibt `setMousePosition` eine Beschreibung der gefundenen passenden Objekte zurück, welche im Statusbalken des Hauptfensters angezeigt wird. Nach jedem Mausereignis wird neu gezeichnet. Die Methode `getDrawableGeoObjects` der Konstruktion liefert die zeichenbare Teilmenge der zugehörigen geometrischen Objekte als Liste. Dieser wird auch, sofern vorhanden, das Vorschau-Objekt hinzugefügt.

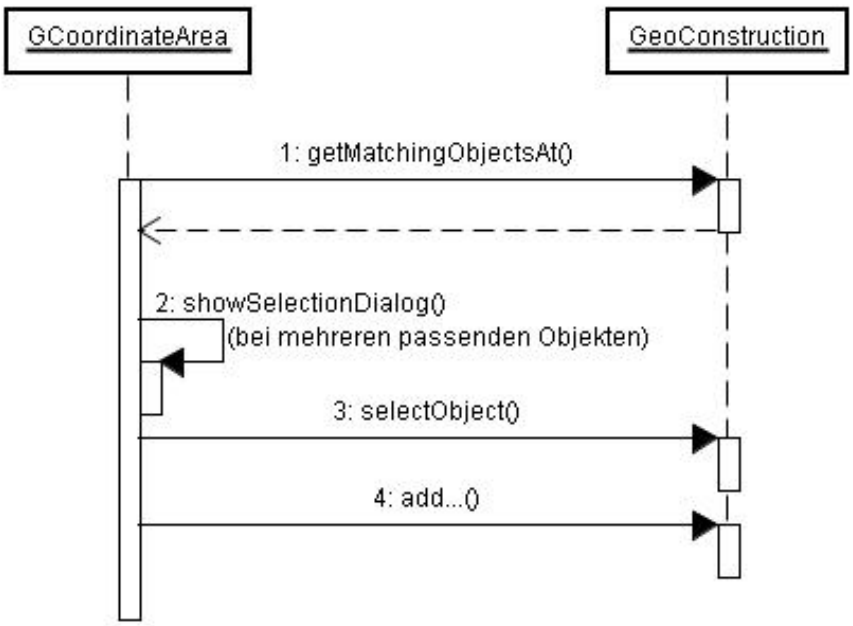

Abbildung 9: Mausclick links

Mausklicks werden durch die Methode `mouseClicked` abgehört. Im Falle eines Klicks mit der linken Taste wird zunächst der Typ des aktuellen Tools geprüft. Handelt es sich dabei um 'Punkt hinzufügen', so wird wird die Methode `isAnyObjectAt` der Konstruktion aufgerufen. Diese gibt zurück, ob sich an der angegebenen Position ein geometrisches Objekt befindet. Falls nicht, wird die Methode `addPoint` der Konstruktion aufgerufen, um einen neuen freien Punkt zu erschaffen. Befinden sich an der angegebenen Position jedoch ein bzw. zwei Objekte, wird stattdessen `addFloater` bzw. `addIntersection` genutzt.

Ist das aktuelle Werkzeug ein anderes, so stellt die Konstruktion über die Methode `getMatchingObjectsAt` eine Liste der Namen aller zu den Eingabeerwartungen passenden Objekte an der Mausposition zur Verfügung. Enthält die Liste genau ein Objekt, so wird dieses durch die Methode `selectObject` der Konstruktion ausgewählt. Enthält die Liste aber mehrere Objekte, wird zunächst ein Auswahldialog angezeigt. Letztendlich wird in Abhängigkeit vom aktuellen Werkzeug noch eine der zum Hinzufügen von Objekten bestimmten Methoden der Konstruktion (wie beispielsweise `addParaLine`) aufgerufen. Wurde bereits eine hinreichende Anzahl geometrischer Objekte ausgewählt, wird erst eine entsprechende Konfigurati-

on und dann über deren `createObject`-Methode ein neues geometrisches Objekt erzeugt und der Objektliste hinzugefügt. Andernfalls bleibt der Aufruf wirkungslos.

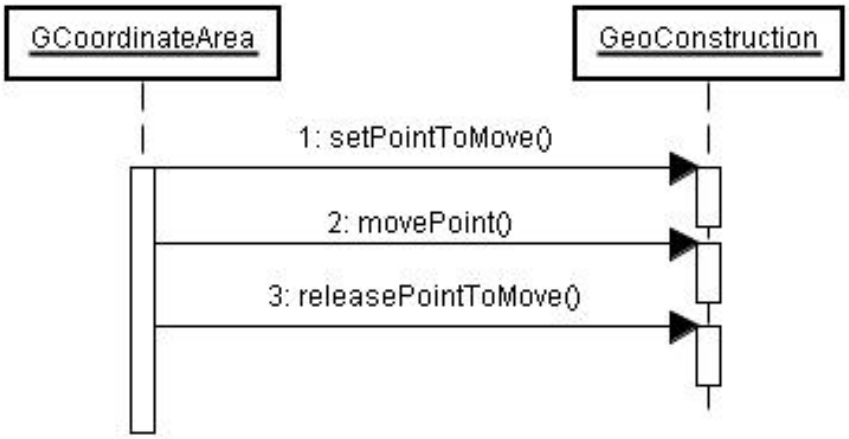

Abbildung 10: Mausclick rechts

Durch die Methoden `setPointToMove` und `releasePointToMove` der Konstruktion wird beim Drücken bzw. Loslassen der rechten Maustaste ein Punkt der Konstruktion zum Bewegen ausgewählt (natürlich nur, wenn sich an der entsprechenden Position ein solcher befindet) bzw. wieder 'losgelassen'. Beim Ziehen der Maus wird dann die aktuelle Mausposition an `movePoint` übergeben. Damit werden die Koordinaten des entsprechenden Punktes geändert und die komplette Konstruktion neu berechnet.

Index