Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik
Studiengang Media Systems
Mobile Systeme
SoSe 2014
Dozent: Prof. Dr. Andreas Plaß

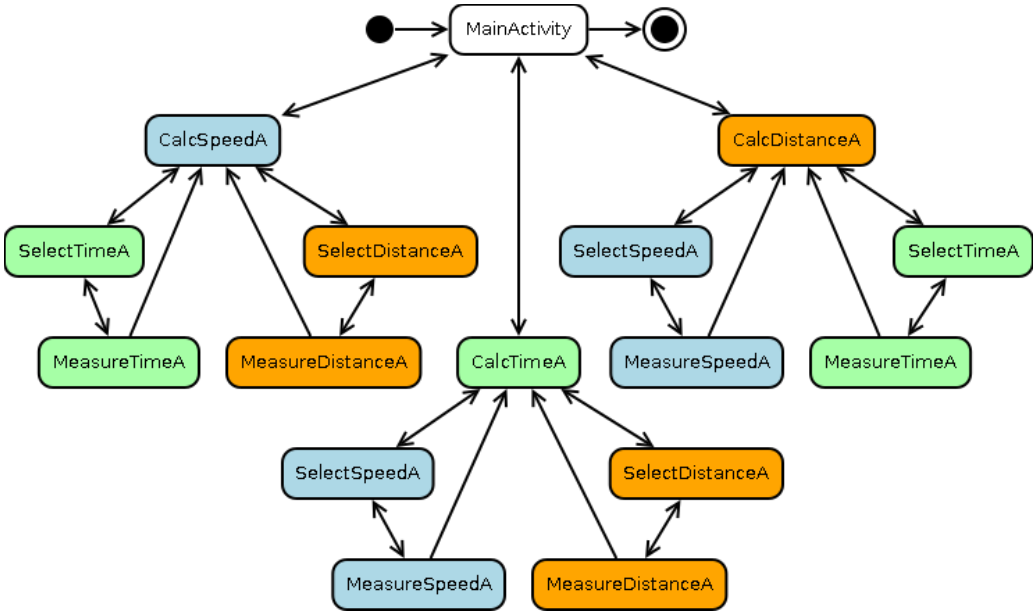
Dokumentation zu „Sind wir schon da?“

Urheber: Robin Christopher Ladiges
Matrikelnummer: 1978865
Abgabedatum: 2014-06-29

Inhaltsverzeichnis

1. Einleitung

„Sind wir schon da?“ ist eine Android App, für die Android API 4.0 oder höher, die es erlaubt zwischen den physikalischen Größen Geschwindigkeit, Entfernung und Zeit hin und her zu rechnen. Dabei werden für jede Größe mehrere Einheiten angeboten, aus denen der Benutzer wählen kann. Neben der Funktion, die Werte zu den jeweiligen Einheiten selbst einzugeben, besteht auch die Möglichkeit einige vorgegebene Größen auszuwählen oder mit dem mobilen Gerät zu messen.



Die App startet in der MainActivity, von wo aus hat der Benutzer die Möglichkeit, in eine der drei Activities CalcSpeedActivity, CalcTimeActivity oder CalcDistanceActivity zu wechseln (Activity ist im Diagramm als A abgekürzt).

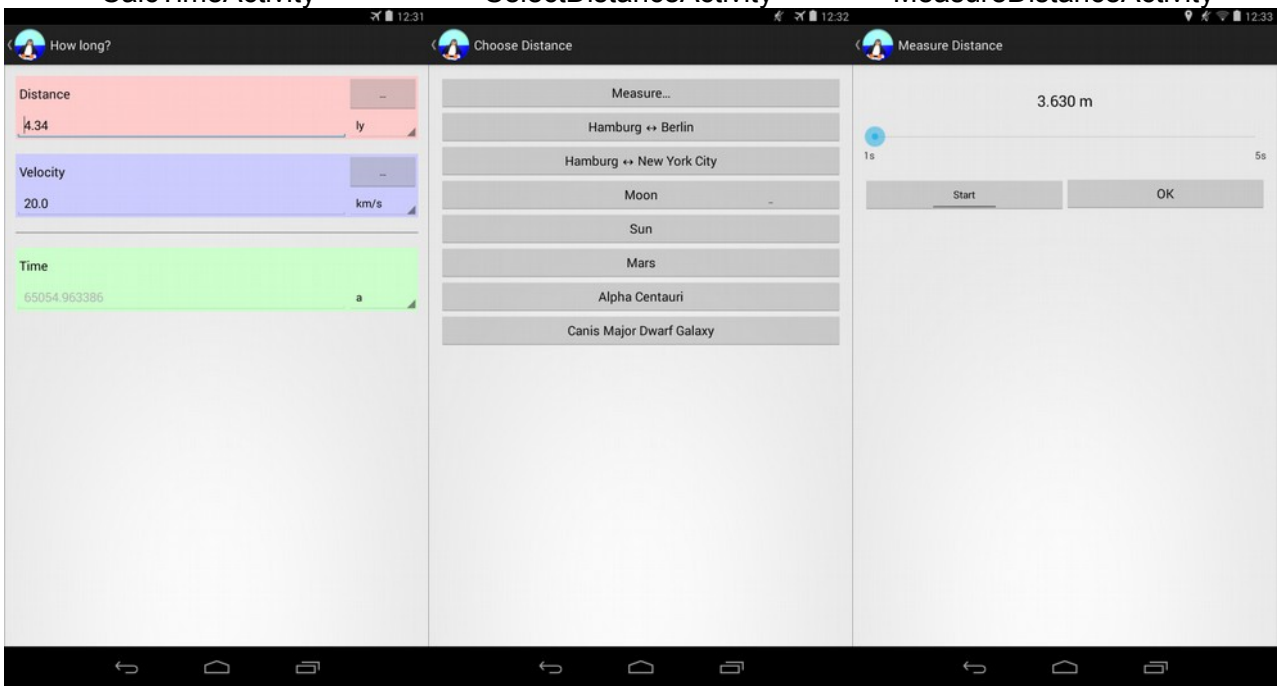
In den Calc-Activities kann der Benutzer selbst Werte eintragen und Einheiten auswählen um dadurch die gewünschte Berechnung durchzuführen. Über den Button mit der Bezeichnung „...“ gelangt er in eine Select-Activity, in der er vorgegebene Wert-Einheit-Kombinationen aus einer Liste auswählen kann, welche dann in die Calc-Activity übernommen werden.

Drückt der Benutzer in den Select-Activities auf den Button „Messe...“ (oder Englisch: „Measure...“), gelangt er in eine Measure-Activities, in der er selbst mit seinem mobilen Gerät diese

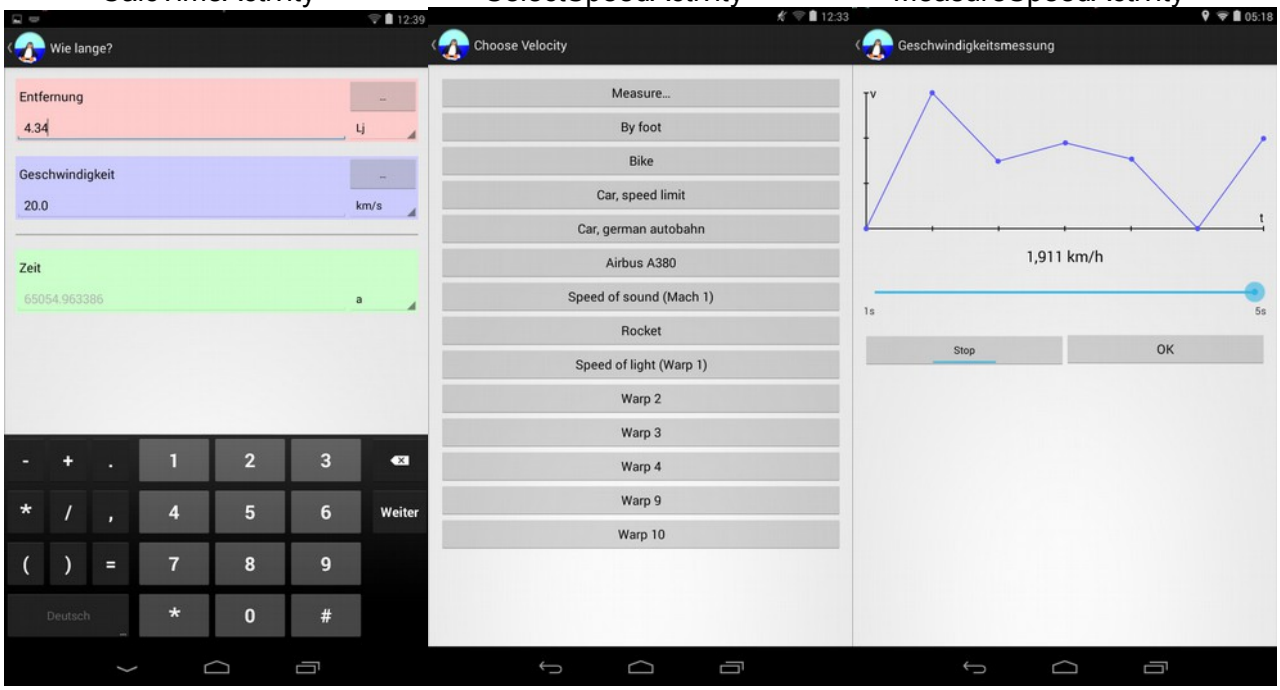
physikalische Größe messen kann. Der gemessene Wert wird über den Button „OK“ in die entsprechende Calc-Activity übernommen.

2. Screenshots

CalcTimeActivity SelectDistanceActivity MeasureDistanceActivity



CalcTimeActivity SelectSpeedActivity MeasureSpeedActivity



3. Entwicklungsumgebung

Dieses Projekt unterscheidet sich von üblichen Android-Projekten dadurch, dass es nicht in der Programmiersprache Java sondern in Scala entwickelt wurde. Dadurch sind einige zusätzliche An-

passungen der Eclipse-Version von ADT notwendig, um den Quelltext kompilieren zu können.

3.1 Scala

Warum Scala und nicht Java? Scala-Quelltext ist in vielen Situationen kürzer und dadurch einfacher nachzuvollziehen als Java-Quelltext. Dies liegt daran, dass in Java viele Informationen doppelt und dreifach angegeben werden müssen. Zum Beispiel muss in Java beim Anlegen von Variablen für Objektreferenzen immer der Klassenname mit angegeben werden, obwohl dieser oft noch in der selben Zeile wiederholt wird, um das Objekt mit new zu erzeugen. Der Scala-Kompiler erlaubt es in vielen Situationen Typbezeichnungen wegzulassen, an denen dieser den Typ aus dem Kontext selbst erschließen kann. Auch wenn Scala-Quelltext dadurch oft wie der einer dynamisch typisierten Programmiersprache aussieht, steckt dahinter ein statisches Typsystem.

Ebenso wie neuerdings seit März mit Java 8, gibt es in Scala schon seit Jahren Lambda-Expressions, die es ermöglichen anonyme Funktionen in einer kurzen Schreibweise zu deklarieren, ohne sie manuell in einem Objekt zu kapseln (z.B. in Java üblicherweise als ein Runnable, für anonyme Funktionen, die nur Seiteneffekte erzeugen). In Scala werden alle Methoden (auch jene in der Java-API) automatisch als first-class Objekte behandelt und können deshalb als Parameter verwendet werden (was ebenfalls erst seit Java 8 mittels Methoden-Referenzen in Java machbar ist).

Methodennamen sind in Scala freier wählbar, so können die Zeichen +, -, * und / als Methodennamen verwendet werden (aber auch beliebige andere Unicode-Zeichen wie z.B. ≤, ≠, ÷, ∀, ∃, ¬, ∧, ∨, ⊕, ⊗, ∩, ∪, ∈, etc. sofern diese nicht bereits von Scala selbst reserviert sind), was hier insbesondere bei den physikalischen Größen Anwendung findet. Zusammen mit der Möglichkeit, teilweise den Punkt und die Klammern bei Methodenaufrufen wegzulassen (Infix-Notation), führt dies zu sehr schön lesbaren Quelltext.

Ein großer Vorteil bietet sich auch durch das Schlüsselwort implicit, welches ein umfangreiches und komplexes Thema ist (erkennbar durch von der IDE unterstrichenen Quelltext). Kurz zusammengefasst, lassen sich damit z.B. eigene Wrapperklassen, wegzulassende Parameterübergaben oder Typumwandlungen definieren, die der Kompiler automatisch dort anwendet wo sie benötigt werden.

Beispiel 1 aus der CalcDistanceActivity, um zwei SIValues in den SIValueInput-Objekten a und b miteinander zu multiplizieren und in die gewünschte Einheit des SIValueOutputs c umzuwandeln:

```
// Java:
public void changed() {
  c.setValue(
    a.getValue()
      .multiply( b.getValue() )
      .toUnit( c.getValue().getUnit() )
  );
}
// Scala: (eigene Operatoren, implizite Typumwandlungen und Infix-Notation)
def changed = c := (a * b) toUnit c
```

Beispiel 2 aus der MeasureTimeActivity, um beim Drücken des ToggleButtons den Timer ein- bzw. auszuschalten:

```
// Java:
butStartStop.setOnCheckedChangeListener(
  new OnCheckedChangeListener(){
    @Override
    public void onCheckedChanged(CompoundButton b, boolean state){
```

```
      if(state) timer.start();
      else timer.stop();
    }
  }
);
// Scala: (implizite Wrapperklasse, Lambda-Expression, weglassen des Typs von state)
butStartStop onChange { state =>
  if(state) timer.start
  else timer.stop
}
```

3.2 Eclipse-Addons

Um mit Eclipse in Scala zu programmieren, gibt es das Eclipse-Addon ScalaIDE for Eclipse, welches unter http://scala-ide.org/download/current.html angeboten wird.

Der darin enthaltene Scala-Kompiler erzeugt Java-Bytecode, der wie jedes andere Java-Programm auch auf der JVM läuft. Es werden jedoch für viele Funktionalitäten von Scala Klassen benötigt, die nicht in der Standard Java-API vorhanden sind, weshalb oft die Scala Library (als JAR-Datei) zu dem JVM-Classpath hinzugefügt werden muss.

Dies ist für Android-Projekte, die in APK-Dateien ausgeliefert werden, nicht ganz so einfach, weshalb es dafür das Eclipse-Addon AndroidScalaProguard gibt, welches dafür sorgt, dass eine minimal große JAR-Datei (also nur die Klassen enthaltend, die auch benötigt werden) in das APK-Archiv eingebunden wird. Eine Anleitung zur Einrichtung gibt es unter http://scala-ide.org/docs/tutorials/androiddevelopment/.

Für dieses Projekt wurde eine ScalaIDE für Scala 2.10 verwendet, weil es bei mir mit der seit 21. April verfügbaren Version 2.11 zu Kompatibilitätsproblemen mit AndroidProguardScala kam.

4. Pakete

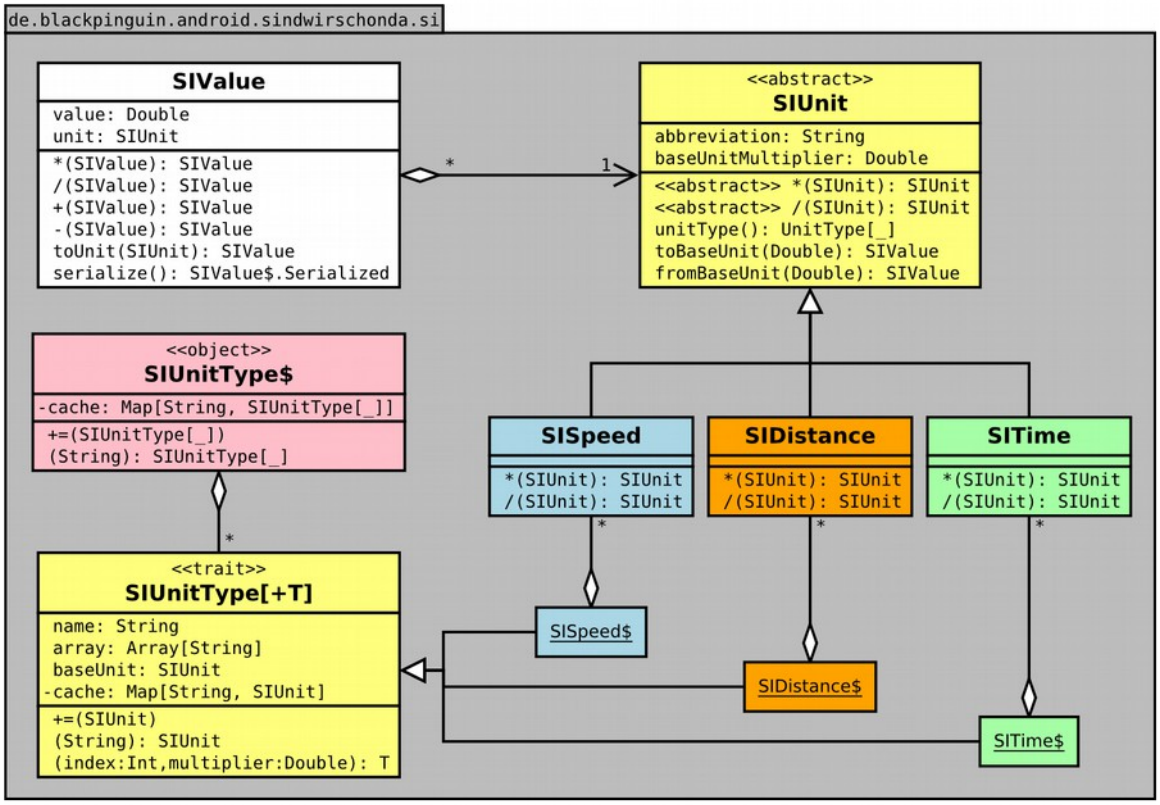
Dieses Projekt ist in vier Pakete untergliedert:
Das Paket **si** kümmert sich um die Abstraktion von physikalischen Größen. In dem Paket **activities** liegen die 10 Activities dieses Projektes zusammen mit deren abstrakten Oberklassen, die das gemeinsame Verhalten der Activities generalisieren. Eigene Views, die in den Layouts der Activities referenziert werden, befinden sich im Paket **views**. Für die Zeit-, Geschwindigkeits- und Entfernungsmessung liegen im Paket **util** wiederverwendbare Klassen, um die Messung abstrakt von der Anwendung mittels Callback-Funktionen zu kapseln.

Beim Betrachten der Klassendiagramme der Pakete ist es wichtig, auf die Einrückung zu achten, um die Methodensichtbarkeit (+, -, #) von Methodennamen (+, -, +=) unterscheiden zu können.

Die Methoden ohne Name, die direkt mit den Klammern beginnen, sind eine Kurzschreibweise von Scala. Hierbei handelt es sich eigentlich um Methoden mit dem Namen apply, die häufig für Konstruktormethoden in Begleitobjekten oder als Getter verwendet werden.

```
// Scala:
var arr = Array(1, 2, 3) //Array ist das Begleitobjekt Array$ zu der Klasse Array[T]
var elem = arr(0)        //arr ist eine Instanz der Klasse Array[Int].
// Java:
Array<Integer> arr = Array$.apply<Integer>(1, 2, 3);
  // Entspricht: int[] arr = new int[]{1, 2, 3};
Integer elem = arr.apply(0);
  // Entspricht: int elem = arr[0];
```

4.1 si



Der Kern dieses Paketes ist die Klasse SIValue. Ein SIValue-Objekt stellt eine physikalische Größe dar, die sich aus einem Zahlenwert (value) und einer Maßeinheit (unit) zusammensetzt. Da SIUnit-Objekte nicht serialisierbar sind, bietet die SIValue-Klasse die Methode serialize an, die ein serialisierbares Objekt erstellt (aus Zahlenwert, Einheitenabkürzung und Einheitentypenname), welches benötigt wird um SIValue-Objekte zwischen Activities auszutauschen.

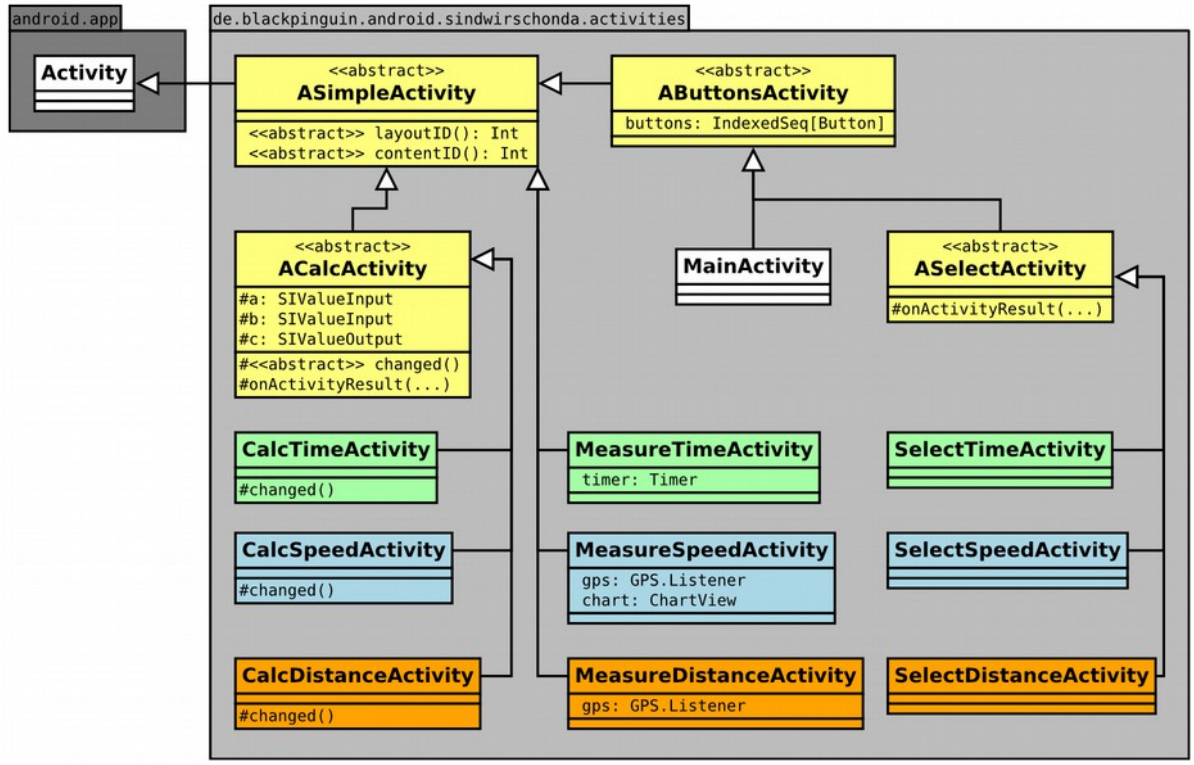
Physikalische Einheiten (SIUnit), wie Meter oder Sekunden, gehören immer einem bestimmten Typ, wie Länge (SIDistance) oder Zeit (SITime), an, die in der Multiplikation oder Division andere Einheiten ergeben. Verschiedene Einheiten des selben Types wie z.B. Lichtjahre und Meilen haben eine Abkürzung (abbreviation) und lassen sich mittels bekannten Umrechnungsmultiplikator (baseUnitMultiplier) in eine Basiseinheit umrechnen (toBaseUnit) und darüber in beliebige andere Einheiten umwandeln (fromBaseUnit).

Um das richtige Einheitenobjekt über eine bekannte Abkürzung zu finden, gibt es die Begleitobjekte (Schlüsselwort: object) SISpeed$, SIDistance$ und SITime$, die eine gemeinsame „Oberklasse“ SIUnitType haben, um deren gemeinsames Verhalten zu generalisieren. Diese haben jeweils getrennt voneinander eine eigene Map, zur Abbildung von Abkürzungen auf Einheiten, was wichtig ist, weil z.B. die Abkürzung „m“ sowohl für Meter, als auch Minuten stehen kann.

Ebenso können die gewünschten Begleitobjekte mittels deren Namen (z.B. „Geschwindigkeit“) über das Begleitobjekt SIUnitType$ gefunden werden (notwendig für die Deserialisierung).

Die Strings für die Einheitentypennamen und die Einheitenabkürzungen werden aus Androids String- bzw. String-Array-Ressourcen entnommen und beinhalten dadurch sprachliche Unterschiede. So wird z.B. die astronomische Einheit in Deutschland mit AE abgekürzt und nicht mit dem internationalen AU für Astronomical Unit.

4.2 activities



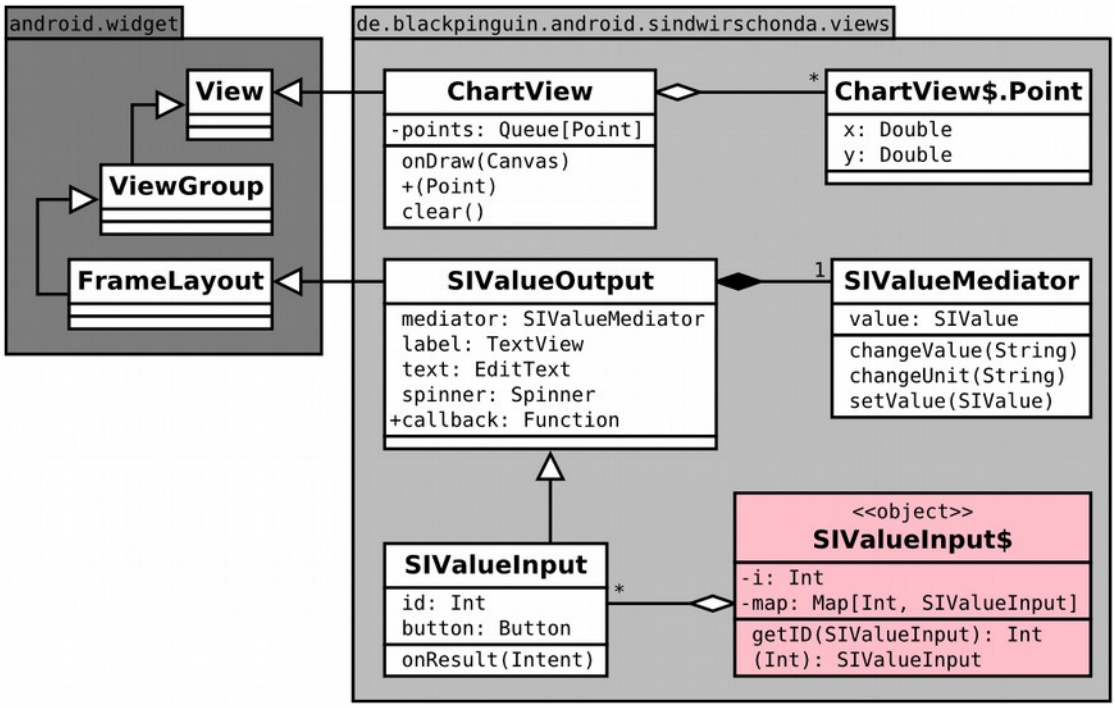
Alle Activities in diesem Paket leiten sich von der abstrakten Oberklasse ASimpleActivity ab, welche den Unterklassen u.a. die Arbeit abnimmt, das Layout in der onCreate Methode zu setzen, wozu diese wiederum lediglich einzeilig die Methode getLayoutID implementieren müssen (z.B.: `def getLayoutID = R.layout.activity_calc_distance`).

Die MainActivity und die Select-Activities haben gemeinsam, das auf ihnen nur Buttons aufgelistet werden, weshalb das Verhalten, die Buttons zu erstellen und ihnen Aktionen (Listener) zuzuweisen, in der abstrakten Klasse AButtonsActivity generalisiert wird. Der zweite Parameter (nach dem Pfeil) der += Methode, welche Buttons hinzufügt, kann entweder eine Activity-Klasse sein, wodurch bei einem Klick dann die Activity geöffnet wird, oder eine beliebige Funktion mit Seiteneffekt sein, die bei einem Klick ausgeführt wird.

Für die Calc-Activities generalisiert die abstrakte ACalcActivity-Klasse gemeinsames Verhalten. Die Layouts für die Calc-Activities definieren für jede Activity zwei Eingabeviews und ein Ausgabeview (siehe Abschnitt 4.3), die über die Variablen a, b und c referenziert werden können. Die konkreten Unterklassen von ACalcActivity realisieren die changed-Methode, die dann aufgerufen wird, wenn sich bei einem der Eingabeviews der Wert ändert, welche die Rechenvorschrift vorgibt mit welcher der Wert des Ausgabeviews berechnet wird (siehe Beispiel 1 von Abschnitt 3.1).

Die onActivityResult-Methoden in ACalcActivity und ASelectActivity können als Rückgabewert einer geöffneten Unteractivity ein Intent bekommen, das ein serialisiertes SIValue-Objekt enthält, welches entweder vom Benutzer ausgewählt oder gemessen wurde. Die Unterklassen von ASelectActivity reichen das Intent als Rückgabewert an die ACalcActivity weiter, von der sie aufgerufen wurden. Die ACalcActivity-Unterklassen reichen das Intent weiter an die onResult-Methode des SIValueInput-Views, dessen „...“-Button gedrückt wurde (siehe Abschnitt 4.3).

4.3 views



Die Klasse ChartView stellt grafisch ein zweidimensionales Koordinatensystem dar, welches in der MesasureSpeedActivity verwendet wird, um die Geschwindigkeit über die Zeit darzustellen. Dabei werden maximal die letzten 7 Punkte angezeigt, und die Anzeige passt sich dynamisch an den größten und kleinsten X- und Y-Werten an (die Achsenstriche stellen also keine konstante, sondern eine, zu den anderen Punkten, relative Geschwindigkeit und Zeit dar).

Um in den Calc-Activities mit den physikalischen Größen zu interagieren, existieren in diesem Paket die beiden Views SIValueOutput und SIValueInput. Für das Berechnungsergebnis ist das SIValueOutput-View und für die beiden Berechnungseingaben das SIValueInput-View zuständig.

Über die callback-Funktion erfährt die Calc-Activity, dass sich eine der beiden Eingabevariablen geändert hat, und die Ausgabe deshalb neu berechnet werden muss. Der Typ dieser Instanzvariablen ist nicht Function wie im Diagramm dargestellt, sondern Function1[Unit, Unit] (Generische Parameter stehen in eckigen, nicht in spitzen Klammern), bzw. in Scala Funktions-Syntax: Unit => Unit. Der Typ Unit sollte hierbei nicht missverstanden werden, es handelt sich nicht um eine physikalische Einheit (wie SIUnit), sondern eine von Scala vorgegebene Klasse, die isomorph zu dem Java-Schlüsselwort void verstanden werden kann. Es handelt sich also um eine Funktion ohne Parameter und ohne Rückgabewert, die nur mit Seiteneffekten arbeitet, so wie Objekte die das Java-Interface Runnable implementieren.

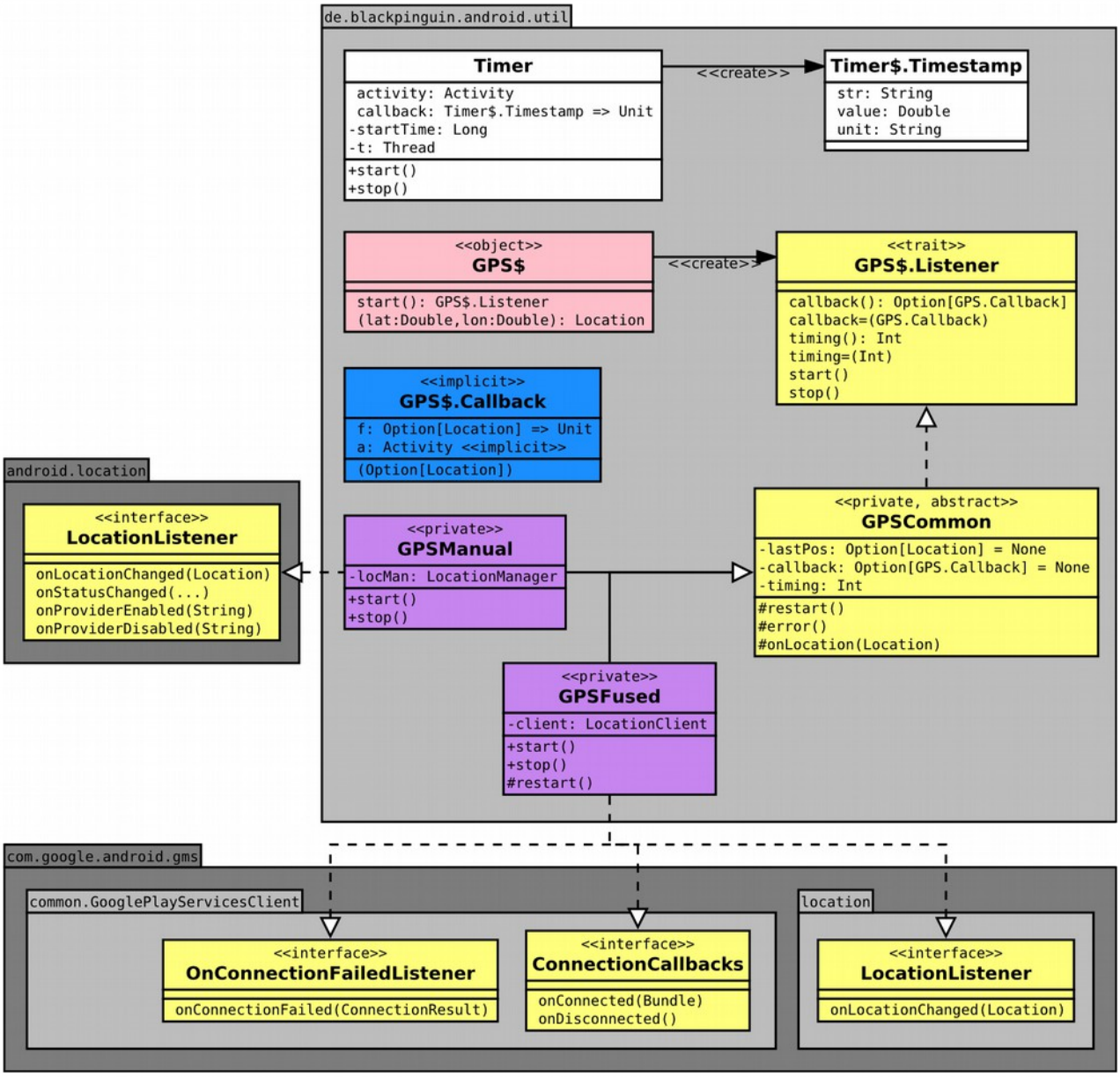
SIValueOutput-Objekte enthalten unter anderem die interaktiven Android-Widgets EditText und Spinner. Das EditText zeigt den Zahlenwert des SIValues an, und erlaubt diesen in der SIValueInput-Klasse zu bearbeiten. Für die Einheit des SIValue-Objektes ist analog dazu das Spinner-Widget zuständig, welches beim Ändern der Einheit den Zahlenwert entsprechend von der alten in die neue Einheit umrechnet.

Da die SIValueOutput-Klasse in sich ziemlich viel Funktionalität vereinigt und mit vielen Objekten interagiert, wird das SIValue-Objekt und die Interaktion damit (Wert ändern, Einheit ändern, beides

ändern, EditText ändern, Spinner ändern, etc.) über die SIValueMediator-Klasse gekapselt (siehe Mediator-Pattern, GoF:273).

Weil jede Calc-Activity Referenzen für zwei SIValueInput-Objekte besitzt, wird ein Mechanismus benötigt, um erkennen zu können, für welches der beiden Objekte ein Ergebnis über die onActivityResult-Methode eingegangen ist (siehe Abschnitt 4.2). Dazu teilt das SIValueInput-Begleitobjekt SIValueInput$ diesen bei ihrer Erstellung eine eindeutige ID mit, die der aufzurufenden Select-Activity übergeben wird. Über diese ganzzahlige ID, welche der onActivityResult-Methode übergeben wird, kann das richtige SIValueInput-Objekt in der Map des Begleitobjektes gefunden werden (mit der apply-Methode).

4.4 util



Die Klasse Timer realisiert über einen zusätzlichen eigenen Thread die Funktionalität einer Stoppuhr, die kontinuierlich während sie läuft über die callback-Funktion ihren aktuellen Anzeigewert der

Activity mitteilt. Eine Referenz auf die Activity wird dabei benötigt, um die callback-Methode nicht innerhalb des eigenen Threads auszuführen, sondern im Kontext der Activity, weil andere Threads nicht über die ausreichenden Berechtigungen verfügen, um die Widgets der Activity verändern zu dürfen.

Das Objekt GPS$ bietet zum einen eine einfache Möglichkeit, Location-Objekte im Quelltext zu erzeugen (z.B.: `val hh = GPS(53.550556, 9.993333)`), und zum anderem besitzt sie die Factory-Methode start (siehe Factory Method Pattern, GoF:107), welche ein Objekt des traits GPS$.Listener erstellt. (Scala hat keine Interfaces, dafür aber Traits, die zusätzlich auch Instanzvariablen und Methodenimplementierungen enthalten können. Dadurch sind sie ziemlich ähnlich zu abstrakten Klassen, ermöglichen aber anders als diese Mehrfachvererbung.)

GPS$.Listener wird von der abstrakten Klasse GPSCommon und dessen Subklassen GPSManual und GPSFused realisiert. GPSCommon generalisiert das Verhalten, das für beide Unterklassen identisch ist.

Mit GPSManual habe ich zunächst selbst den manuellen Zugriff über den Android LocationManager auf das GPS Modul programmiert. Später bin ich mit der Klasse GPSFused auf den neueren Location Client des Google Play Services gewechselt, welcher die verschiedenen Sensoren des mobilen Gerätes (GPS, WiFi, Akkumulator, etc.) kombiniert, um genauere Location-Objekte zu erzeugen. Welche der beiden Realisierungen des GPS$.Listener „Interfaces“ verwendet wird, ist im GPS$ Objekt transparent für die Anwendung festgelegt (transparent im Sinne von Information Hiding). Durch diese Architektur kann einfach die eine gegen die andere Realisierung ausgetauscht werden, da sich beide zur Anwendung hin identisch verhalten.

Mit der timing Variable, welche vom Benutzer in der Activity über ein SeekBar-Widget gesteuert werden kann, wird der minimale Zeitabstand zwischen zwei Location-Objekten unverbindlich eingestellt.

Sowohl die MeasureDistanceActivity, als auch die MeasureSpeedActivity, bekommen von der selben GPS$.Listener Klasse ihre Location-Objekte über die callback-Funktion. Sie unterscheiden sich lediglich in der Auswertung und in der Darstellung der Location-Objekte. Die MeasureDistanceActivity berechnet die Distanz aus der aktuellen und der vorherigen Location und summiert diese über den gesamten Messzeitraum auf. MeasureSpeedActivity berechnet ebenso die Distanz, dividiert diese jedoch durch die vergangene Zeit zwischen beiden Messpunkten, um die Geschwindigkeit zu berechnen. Statt die Geschwindigkeiten lediglich zu summieren, werden diese noch durch die Anzahl der Messpunkte dividiert, um die Durchschnittsgeschwindigkeit zu berechnen (ein Wechsel des Timings während einer Messung verfälscht das Messergebnis zugunsten der Messzeiträume mit einer höherer Messfrequenz).