# Einführung

In dieser Aufgabe geht es um die Entwicklung eines Terminplaners. Denken Sie an das Szenario, einen Zahnarzt anzurufen, um einen Termin zu vereinbaren. Sie fragen nach einem Termin, ggf. aus zeitlichen Gründen, die Zahnarzthelferin prüft, ob ein Termin vereinbart werden kann.

Ziel der Aufgabe ist es, sich mit der Datenstruktur der Linked List vertraut zu machen, aber auch Ihre Fähigkeiten im Bereich Test Driven Development weiterzuentwickeln.

|  |
| --- |
| Inhalt  [Einführung 1](#_Toc114673804)  [Zeiteinteilung 1](#_Toc114673805)  [Umgang mit der Zeit in Java 2](#_Toc114673806)  [Aufgabe 1: Untersuchen Sie die API 2](#_Toc114673807)  [Bereitstellen Ihrer Implementierung 4](#_Toc114673808)  [Testgetriebene Entwicklung 4](#_Toc114673809)  [Testdaten des Lehrers 4](#_Toc114673810)  [Aufgabe 2: Beginnen Sie mit der Implementierung des Dienstes 6](#_Toc114673811)  [Timeline-Modell 7](#_Toc114673812)  [Hinweise für den Implementierer 7](#_Toc114673813)  [Termine zusammenlegen, gemeinsame Freizeit finden. 8](#_Toc114673814)  [Ihre Aufgaben pro Woche 9](#_Toc114673815) |

## Zeiteinteilung

Natürlich können Sie beim Verwalten eines Kalenders die Strategie wählen, nur Termine zu verfolgen. Wenn an einem Tag keine Termine vorhanden sind, ist Ihre Terminliste leer. Wenn Sie einen Termin hinzufügen, befindet sich ein Termin in der Liste usw. Der empfohlene Ansatz ist jedoch die Verwendung einer Zuordnungsstrategie auf einer Zeitachse. Wenn es keine Termine gibt, gibt es ein Element in Ihrer Liste: ein verfügbares Zeitfenster (vom Beginn des Tages bis zum Ende des Tages). Wenn ein Termin mitten am Tag hinzugefügt wird, haben Sie am Ende drei Einträge in Ihrer Liste:

* Ein freier TimeSlot vom Tagesbeginn bis zum Beginn des Termins
* Ein belegter TimeSlot, der Termin
* Ein freier TimeSlot von Terminende bis zum Ende des Tages

Das Hinzufügen eines Termins ist jetzt grundsätzlich in zwei Schritte unterteilt:

* Suche nach einem freien Zeitfenster, in das der Termin passt
* Den freien TimeSlot in Stücke schneiden. Was sind die Szenarien?
  + Der Termin hat dieselbe Dauer wie der TimeSlot; der TimeSlot wird einfach belegt. Kein Schnitt erforderlich.
  + Der Termin steht am Anfang des TimeSlots; Der TimeSlot wird in zwei Teile geteilt, wobei es sich um den Termin handelt, gefolgt von einem freien TimeSlot.
  + Der Termin liegt mitten in einem TimeSlot; der TimeSlot ist in drei Teile geteilt, ein freies Timeslot, der Termin, gefolgt von einem freien TimeSlot.
  + Der Termin liegt am Ende des TimeSlots; der TimeSlot wird in zwei Teile geteilt, wobei es sich um einen freien TimeSlot handelt, gefolgt von dem Termin.

Sie werden merken, die Allokationsstrategie macht das Leben leichter! Schauen Sie sich die Abbildung unten an. Sie werden erkennen, dass hier das 4. Szenario (wie oben beschrieben) visualisiert wird.

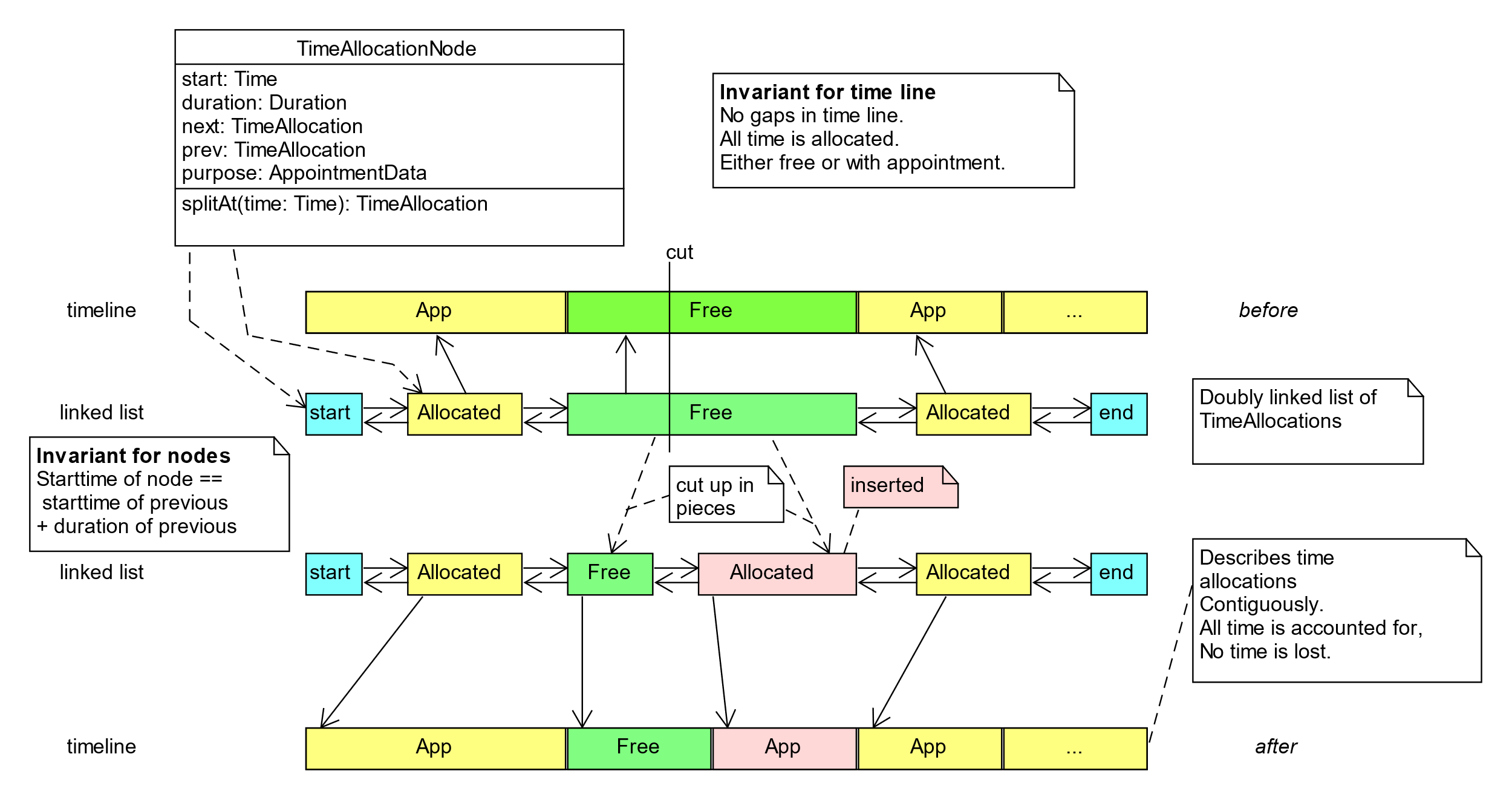


Abbildung : Zeiteinteilung

## Umgang mit der Zeit in Java

In der vorherigen Aufgabe haben wir selbst einige zeitbezogene Klassen entwickelt. Das ist natürlich schlechte Praxis. Nutzen Sie besser vorhandene und bereits getestete Komponenten! Für Uhrzeit und Datum verwenden wir die java.time-API. Dies impliziert, dass Zeitpunkte als Instanzen von java.time.Instant und Zeitdauer in java.time.Duration ausgedrückt werden. LocalTime und LocalDate (in einer bestimmten Zeitzone, z. B. Amsterdamer Zeit) werden auf Endbenutzerebene verwendet, um Zeit- und Datumswerte darzustellen.

Die Anforderungen besagen, dass der Terminplanerdienst über Zeitzonengrenzen hinweg funktionieren soll. Dies ist mit der java.time API durchaus möglich. Zögern Sie nicht, diese API erneut zu studieren.

Um die API robust zu machen, wenn sie auf die Vereinbarung von Terminen auf der ganzen Welt angewendet wird, wird die Zeit basierend auf java.time.Instant-Objekten verwaltet, die Zeitpunkte in UTC darstellen. TimeSlot-Objekte sind dann einfach Zeitspannen zwischen zwei Momenten auf der Zeitachse.

Um dem menschlichen Benutzer eine benutzerfreundliche Oberfläche zu bieten, können die geplanten Zeiten und Daten als LocalTime- und LocalDate-Werte dargestellt werden, indem die LocalDay-Klasse verwendet wird, die in der API bereitgestellt wird. Dies ist eine von uns bereitgestellte Hilfsklasse, die für Zeitumrechnungen verwendet werden kann.

## Aufgabe 1: Untersuchen Sie die API

Wir werden gegen Schnittstellen testen. Die Schnittstellen beschreiben die Anforderungen, die Sie umsetzen müssen. Sie implementieren die Schnittstellen.

Klicken Sie in NetBeans mit der rechten Maustaste auf das Projekt „appointmentplanner-api“ und wählen Sie „JavaDoc generieren“. Dadurch wird die gesamte Java-Dokumentation in ein gut lesbares HTML-Format konvertiert. Den Link zur Dokumentation finden Sie in der Ausgabe oder einfach im Verzeichnis "/target/site/apidocs". NEHMEN SIE SICH ZEIT, UM DIE API ZUERST ZU LESEN! Ein wesentlicher Erfolgsfaktor ist, nicht sofort mit dem Programmieren zu beginnen! (Natürlich können Sie Javadoc auch aus Code lesen, aber das ist möglicherweise weniger bequem).

Das folgende Klassendiagramm gibt einen Überblick über die vollständige API. Details wurden weggelassen, sind aber in der API beschrieben.

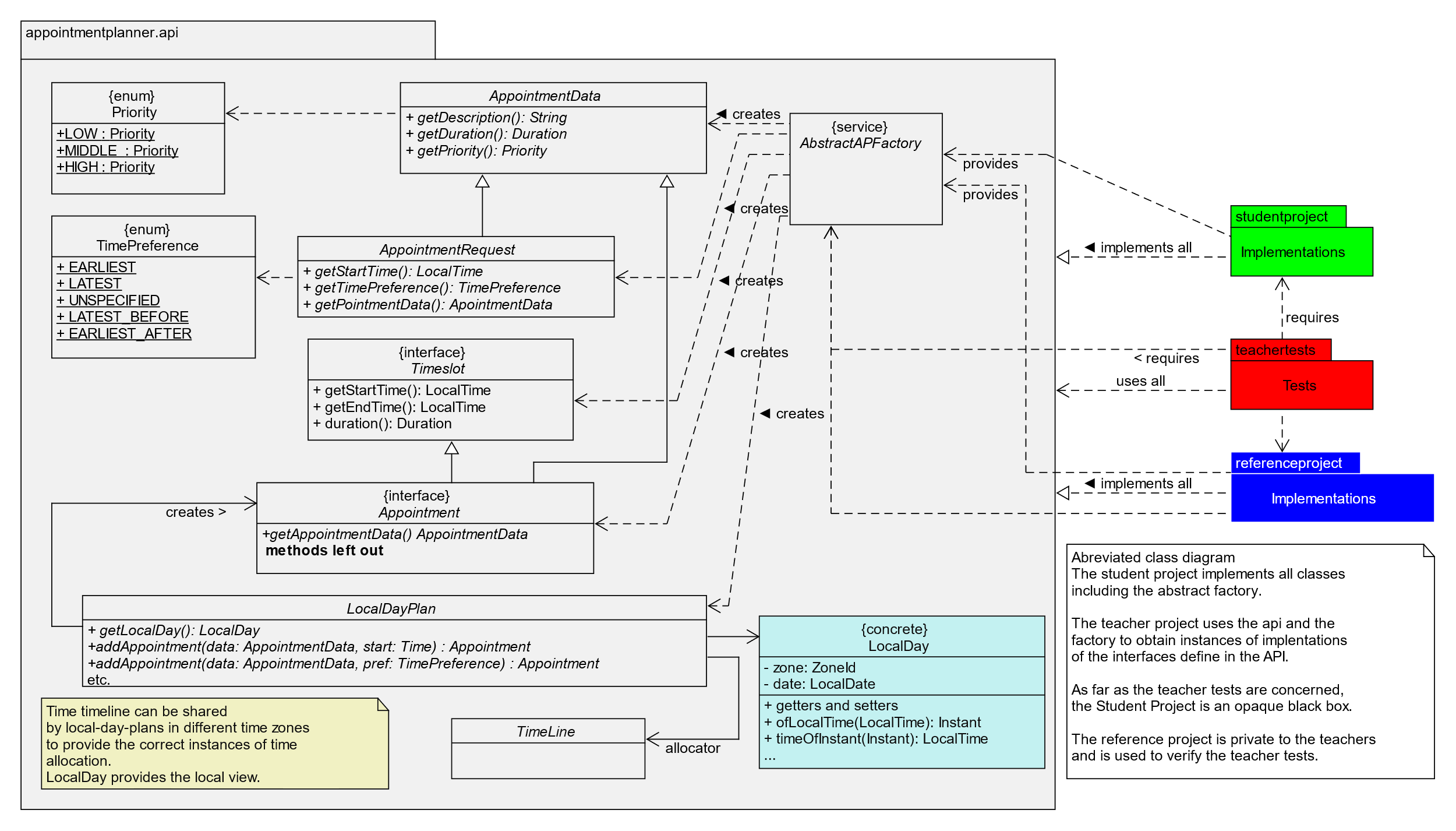


Abbildung : Abgekürztes API-Klassendiagramm

Einige Anmerkungen zu den Typen, die Sie im Klassendiagramm sehen können:

* Der LocalDayPlan kann als externe Schnittstelle für Benutzer des Planungsdienstes angesehen werden, eine Art Fassade (Sie könnten das Fassadenmuster in DARC diskutieren). Die Interaktion mit dem „Benutzer“ (der typischerweise eine grafische Benutzeroberfläche wäre) basiert hauptsächlich auf AppointmentData, LocalTime und TimePreference. Basierend auf diesen erstellt das System intern einen AppointmentRequest und möglicherweise einen Termin; die Erstellung von Objekten dieser Art erfolgt niemals durch den "Benutzer", sie werden jedoch über die LocalDayPlan-Schnittstelle an den Benutzer zurückgegeben.
* Da LocalDayPlan als externe Schnittstelle dient, ist dies der perfekte Ausgangspunkt, um das System basierend auf den Benutzeranforderungen testgetrieben zu entwickeln. Deshalb werden wir das hauptsächlich in der Übung tun.
* Die LocalDayPlan-Implementierung ist hauptsächlich ein "Passthrough". Das eigentliche Arbeitstier ist die TimeLine. Die meisten Methoden in LocalDayPlan sind Standardmethoden. Der Unterschied zwischen diesen beiden (die Schnittstelle ist ziemlich ähnlich) besteht darin, dass eine TimeLine die Zeit in UTC verwaltet. Der LocalDayPlan befindet sich in der Zeitzone des Benutzers und kümmert sich um die Konvertierung von LocalTime-Anforderungen in UTC.
* LocalDay ist ein bestimmter Tag in einer bestimmten Zeitzone. Keine Arbeit für Sie in diesem Kurs.
* Typischerweise möchte ein Benutzer einen Termin vereinbaren, indem er AppointmentData und eine Angabe der bevorzugten Zeit bereitstellt:
  + Wenn nur eine bevorzugte LocalTime übergeben wird, bedeutet dies, dass der Benutzer zu dieser bestimmten Zeit einen Termin haben möchte. Wenn kein passendes Zeitfenster verfügbar ist, wird kein Termin erstellt.
  + Wenn nur eine TimePreference übergeben wird, wie EARLIEST, interessiert sich der Benutzer nicht für eine bestimmte Zeit und überlässt es dem Terminplaner, eine Zeit zu finden, die zu der Zeitpräferenz passt.
  + Wenn LocalTime UND TimePreference angegeben werden, versucht der Terminplaner, den Termin zur angegebenen Zeit zu planen, aber wenn dies nicht möglich ist, verwendet er den TimePreference-Ansatz als Fallback-Szenario.
* Die Signaturen der Factory-Methoden in AbstractAPFactory geben einen klaren Hinweis darauf, wie Konstruktoren der verschiedenen Typen aussehen werden. Zumindest geben sie einen Hinweis, die Umsetzungsdetails bleiben natürlich Ihnen überlassen.
* Der Terminplaner kann nicht für tageübergreifende Termine verwendet werden.

## Bereitstellen Ihrer Implementierung

Da wir gegen Schnittstellen testen, kennen wir die Namen Ihrer Klassen, die die Schnittstellen implementieren, nicht. Das ist auch nicht nötig, denn wir verwenden wieder eine Factory, genau wie in der SimpleTime-Aufgabe.

Das Projekt „appointmentplanner-impl“, in dem Sie Ihre Lösungen schreiben, enthält bereits die Klasse APFactory, die die Schnittstelle AbstractAPFactory implementiert. Das module-info.java in diesem Projekt ist vordefiniert (siehe Standardpaket) und sorgt dafür, dass der APFactory-Dienst automatisch geladen werden kann (liefert…​mit…​). In den Testpaketen stellen wir Ihnen einen ServiceFinder zur Verfügung, der eine statische Methode zum Abrufen eines AbstractAPFactory-Objekts enthält. Wir werden den ServiceFinder in unseren Lehrertests verwenden; Sie können es auch in Ihren eigenen Tests verwenden.

## Testgetriebene Entwicklung

|  |  |
| --- | --- |
|  | **Testen** und **Feedback**  Sie sollten probegefahren arbeiten. Schreiben Sie nur den Code, der in Ihren Tests benötigt wird. Auf diese Weise halten Sie Ihre **Codeabdeckung bei nahezu 100 %.**  Ihr Code wird bald nach einem Push getestet und die Ergebnisse werden in Codegrade veröffentlicht, wenn die Abdeckung IHRER Tests auf IHRER Implementierung über 95 % liegt. |

## Testdaten des Lehrers

In unseren Black-Box-Tests, die wir auf das Studentenprojekt anwenden, verwenden wir die folgenden Testdaten, die als Diagramm dargestellt werden.

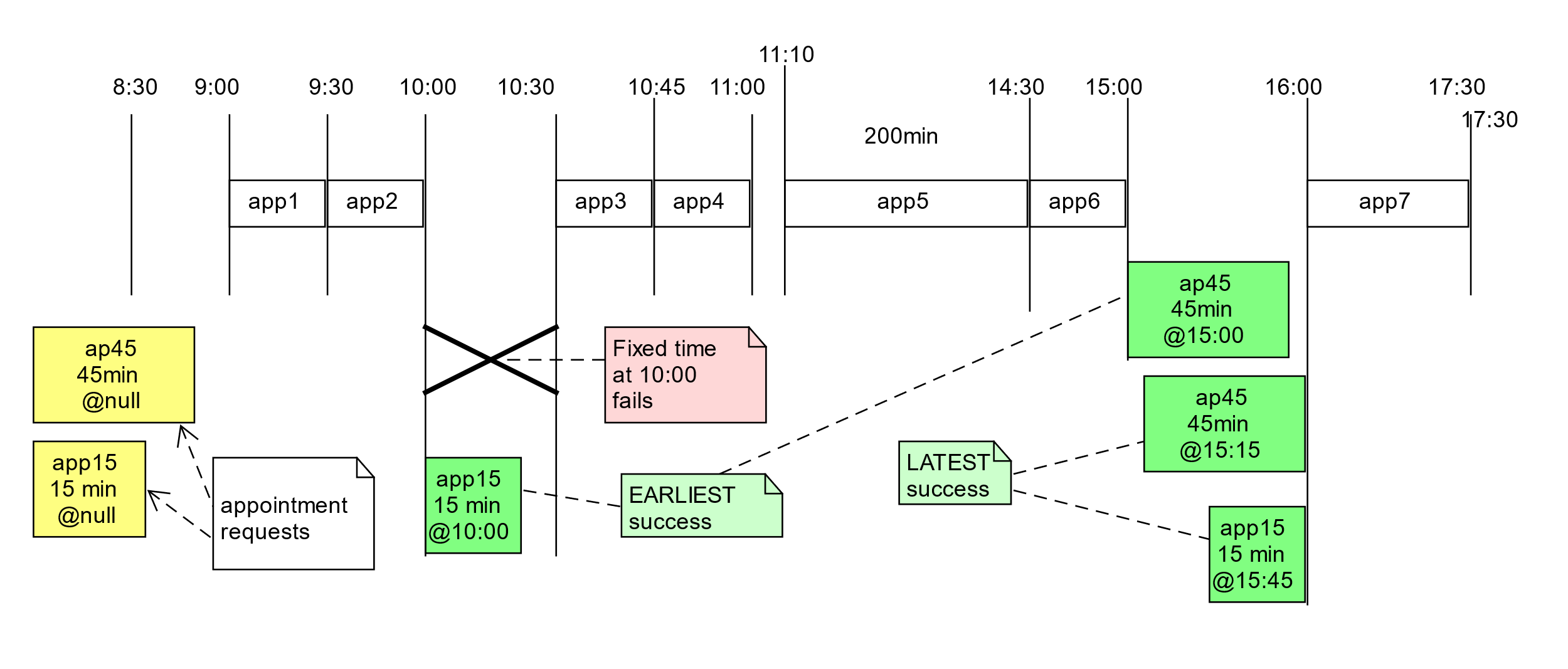


Abbildung : Lehrertestdaten

Als Beispiel: Wenn Sie wissen müssen, was z. B. app6 in einer Fehlermeldung bedeutet, können Sie dem Diagramm entnehmen, dass es sich um einen zwischen 14:30 (Ortszeit) und 15:00 zu planenden Termin handelt, Länge 30 Minuten.

Wir werden dieses Testset in vielen Tests verwenden.

* Beispielsweise wird der Tag, mit dem wir testen, mit app1..7 aus dem Diagramm gefüllt, sodass implizit getestet wird, dass Termine mit festen Zeiten hinzugefügt werden können.
* Dann können einige der Termine entfernt werden, was das Entfernen von Terminen testet. Danach sollte es möglich sein, einen Termin mit längerer Dauer hinzuzufügen.
* In einigen anderen Tests kann der „Tag“ kürzer angelegt werden, um sicherzustellen, dass kein Termin passt und um zu testen, wie die Implementierung darauf reagiert.

Tabelle : Testdatendatei, die in den Lehrertests verwendet wird

|  |
| --- |
| package appointmentplanner;  import appointmentplanner.api.AbstractAPFactory; import appointmentplanner.api.LocalDay; import appointmentplanner.api.LocalDayPlan; import appointmentplanner.api.Priority; import appointmentplanner.api.TimePreference; import java.time.Duration; import java.time.LocalTime;  /\*\*  \*  \* @author Pieter van den Hombergh {@code p.vandenhombergh@fontys.nl}  \*/ interface TestData {   static final AbstractAPFactory fac = GetFactory.getFactory();  static final LocalDay TODAY = new LocalDay();  static final LocalTime T08\_30 = LocalTime.of( 8, 30 );  static final LocalTime T09\_00 = LocalTime.of( 9, 0 );  static final LocalTime T09\_30 = LocalTime.of( 9, 30 );  static final LocalTime T10\_00 = LocalTime.of( 10, 0 );  static final LocalTime T10\_30 = LocalTime.of( 10, 30 );  static final LocalTime T10\_45 = LocalTime.of( 10, 45 );  static final LocalTime T11\_10 = LocalTime.of( 11, 10 );  static final LocalTime T14\_30 = LocalTime.of( 14, 30 );  static final LocalTime T15\_00 = LocalTime.of( 15, 0 );  static final LocalTime T15\_15 = LocalTime.of( 15, 15 );  static final LocalTime T15\_45 = LocalTime.of( 15, 45 );  static final LocalTime T16\_00 = LocalTime.of( 16, 00 );  static final LocalTime T17\_30 = LocalTime.of( 17, 30 );   static final Duration D15 = Duration.ofMinutes( 15 );  static final Duration D30 = Duration.ofMinutes( 30 );  static final Duration D80 = Duration.ofMinutes( 80 );  static final Duration D90 = Duration.ofMinutes( 90 );  static final Duration D200 = Duration.ofMinutes( 200 );   static final APAppointmentData DATA1 = new APAppointmentData( "app1 30 min @9:00", D30, Priority.LOW );  static final APAppointmentData DATA2 = new APAppointmentData( "app2 30 min @9:30", D30, Priority.LOW );  static final APAppointmentData DATA3 = new APAppointmentData( "app3 15 min @10:30", D15, Priority.MEDIUM );  static final APAppointmentData DATA4 = new APAppointmentData( "app4 15 min @10:45", D15, Priority.HIGH );  static final APAppointmentData DATA5 = new APAppointmentData( "app5 200 min @11:10", D200, Priority.HIGH );  static final APAppointmentData DATA6 = new APAppointmentData( "app6 30 min @14:30", D30, Priority.LOW );  static final APAppointmentData DATA7 = new APAppointmentData( "app7 90 min @16:00", D90, Priority.LOW );   static final APAppointmentRequest AR1 = new APAppointmentRequest( DATA1, T09\_00, TimePreference.UNSPECIFIED );  static final APAppointmentRequest AR2 = new APAppointmentRequest( DATA2, T09\_30 );  static final APAppointmentRequest AR3 = new APAppointmentRequest( DATA3, T10\_30 );  static final APAppointmentRequest AR4 = new APAppointmentRequest( DATA4, T10\_45 );  static final APAppointmentRequest AR5 = new APAppointmentRequest( DATA5, T11\_10 );  static final APAppointmentRequest AR6 = new APAppointmentRequest( DATA6, T14\_30 );  static final APAppointmentRequest AR7 = (APAppointmentRequest) fac.createAppointmentRequest( DATA7, T16\_00, TimePreference.EARLIEST );   static LocalDayPlan standardDay() {  LocalDayPlan td = emptyWorkingDay();  addApps( td, AR1, AR2, AR3, AR4, AR5, AR6, AR7 );  return td;  }   static LocalDayPlan emptyWorkingDay() {  return fac.createLocalDayPlan( TODAY, LocalTime.of( 8, 30 ), LocalTime.of( 17, 30 ) );  }   static LocalDayPlan addApps( LocalDayPlan dp, APAppointmentRequest... app ) {   for ( APAppointmentRequest ar : app ) {  ar.apply( fac, dp );  }  return dp;  } } |

## Aufgabe 2: Beginnen Sie mit der Implementierung des Dienstes

Du solltest einfach anfangen. Es gibt einige Datenklassen, die als Interfaces AppointmentRequest und AppointmentData spezifiziert sind und einfach zu implementieren sein sollten. Die implementierende Klasse könnte einfach denselben Namen haben, solange Sie sie in ein anderes Paket einfügen. AppointmentDataImpl, myimpl.AppointmentData oder JohnsAppointmentData wären also alle in Ordnung.

Die TimeLine ist der knifflige Teil.

## Timeline-Modell

Da wir sicherstellen wollen, dass die Übung machbar ist, haben wir eine Implementierung erstellt, die als Ideenquelle dienen kann. Wir teilen die Ideen, nicht die Umsetzung.

In der Implementierung verwaltet die TimeLine intern eine doppelt verknüpfte Liste von Knoten für spezielle Zwecke.

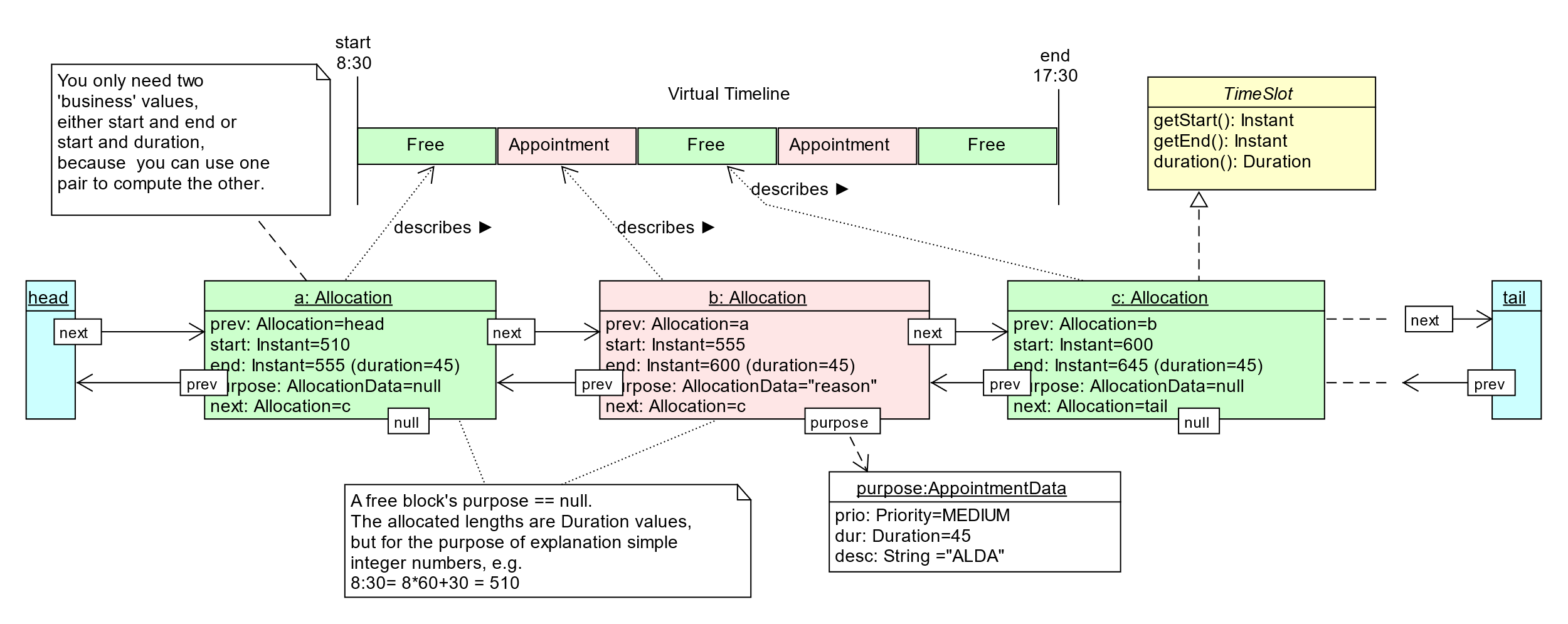


Abbildung : Zeitlinienmodell

Das Zeitachsenmodell zeigt eine doppelt verknüpfte Liste von Knoten für spezielle Zwecke des Typs „AllocationNode“, die eine Vorstellung von Zeitpunkten und Entfernungen (Dauer) zwischen diesen Punkten und einem „Zweck“ haben. Die **Invariante** der Timeline-Implementierungsklasse besteht darin, dass es niemals benachbarte freie Slots gibt. Um diese Invariante wahr zu halten, muss ein Slot, wenn er freigegeben wird, mit einem beliebigen freien benachbarten Slot zusammengeführt werden. Im Bild: Wenn die Zuordnung **b** freigegeben würde, würde sie mit dem linken und rechten freien Block zu einem freien Block zusammengeführt, der sich vom Anfang von **a** bis zum Ende von **c** erstreckt.

## Hinweise für den Implementierer

Wenn Sie moderne Programmiertechniken wie Lambda-Ausdrücke und Streams verwenden, wird die Implementierung eleganter und hat insgesamt weniger Code. Die Verwendung von Streams macht die Auswahl von Zeitfenstern oder Terminen durch entsprechende Filterung besonders einfach. Das Vorhandensein von Streams für jede Richtung (von früh nach spät und von spät nach früh) hilft auch, die Implementierung einiger API-Methoden zu vereinfachen.

Auch wenn Sie Ihre eigene doppelt verkettete Liste haben, ist es möglich, Streams zu verwenden. Die einzige Voraussetzung ist, dass Sie Ihren eigenen Iterator schreiben. Wenn Sie schon dabei sind, erstellen (und natürlich testen) Sie einen Iterator, der auch am anderen Ende beginnt. Wenn Sie einen Iterator haben, ist das Erstellen eines Streams einfach, verwenden Sie einfach das folgende Rezept.

Tabelle : Streaming mit Ihrem selbst erstellten Iterator

|  |
| --- |
| Stream<AllocationNode> stream() { Spliterator<AllocationNode> spliterator = Spliterators.spliteratorUnknownSize( iterator(), ORDERED );  return StreamSupport.stream( spliterator, false ); } |

Die iterator-Methode gibt Ihren Iterator zurück, der dann zum Erstellen eines Streams verwendet wird. Im Beispiel streamt der Stream Zuordnungsknoten. Von dort aus können Sie eine map(…) verwenden, um beispielsweise die Termininformationen abzurufen oder andere Objekte im Handumdrehen zu erstellen.

Sie können auch einfach einen Reverse-Stream erstellen, indem Sie einen reverseIterator() verwenden, den Sie implementieren können. In allen Fällen kann der resultierende Stream als normaler (Java 8) Stream zum Filtern, Sortieren, Zuordnen und Reduzieren verwendet werden. Nützliche Reduce-Operationen sind min, max, collect etc. In vielen Fällen können die benötigten API-Methoden dann mit ein oder zwei nicht allzu komplexen Anweisungen implementiert werden.

## Termine zusammenlegen, gemeinsame Freizeit finden.

Die nützlichste Art, Termine zu vereinbaren, besteht darin, mindestens zwei Parteien einzubeziehen. Beispiele: Sie und Ihre Klasse oder Sie beim Zahnarzt. Das Problem ist, gemeinsame Freizeit zu finden.

Die folgende Abbildung zeigt ein Szenario mit 4 TimeLines. Freie TimeSlots (Lücken) sind Cyan gefärbt (rot im Dunkelmodus). Die "passenden freien Slots" sind Slots, die in allen TimeLines verfügbar sind. Die drei übereinstimmenden Slots im folgenden Szenario sind grün (lila im dunklen Modus).

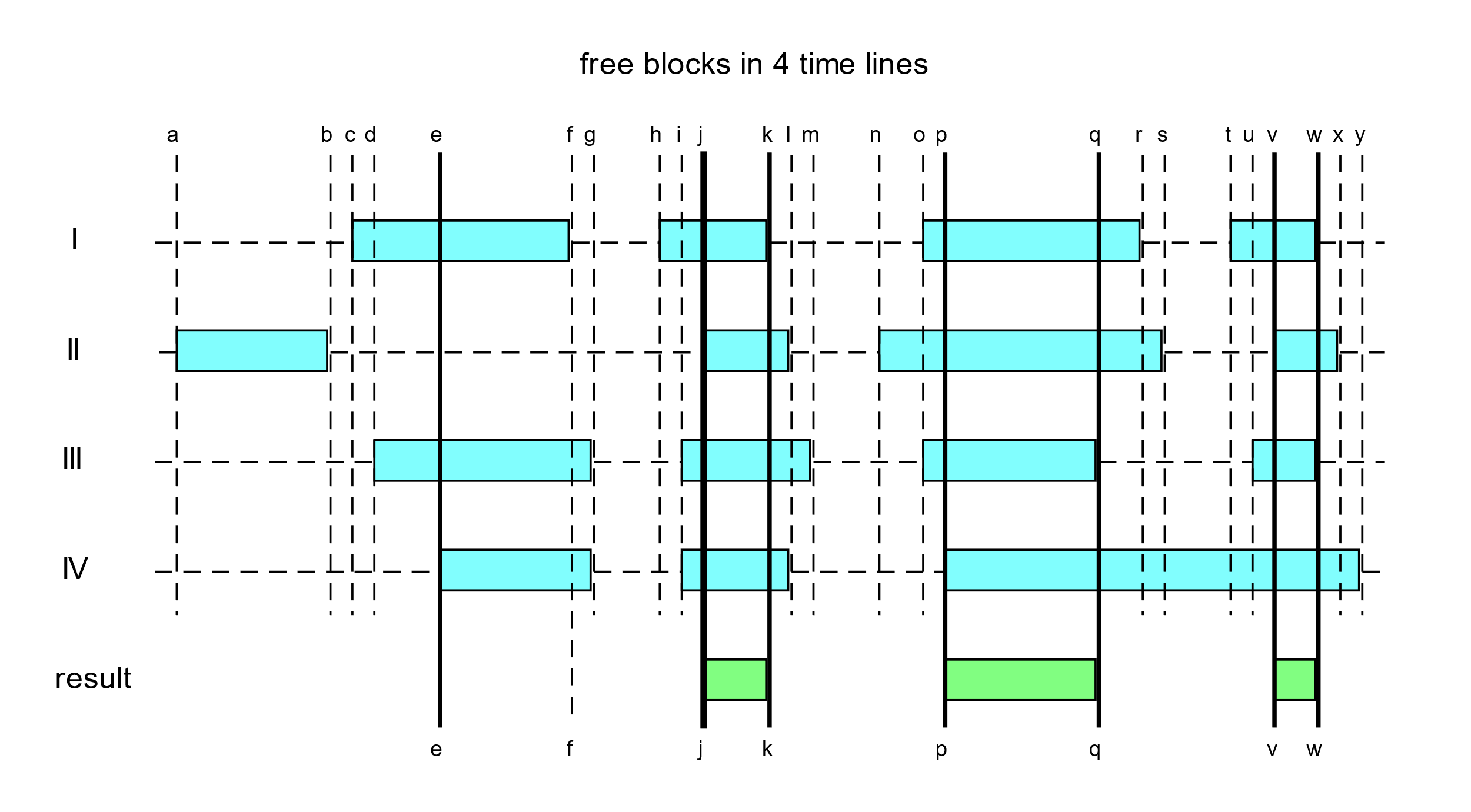


Abbildung : Vier Zeitlinien mit ihren freien Zeitfenstern Input Cyan, Output Grün

**Wie findet man passende freie Slots?**

* Die freien Slots haben zwei Kanten, die Startkante und die Endkante.
* Eine vertikale gestrichelte Linie grenzt einen interessanten Zeitpunkt (einen Moment) in der Zeit ab, wie beispielsweise die Zeit einer oder mehrerer Kanten.
* Damit eine gestrichelte Linie für das Problem des Auffindens gemeinsamer freier Slots von Interesse ist, muss diese Linie einen freien Zeitslot in ALLEN Zeitlinien berühren oder schneiden.
* Im Beispiel schneiden a, b und c nicht; d schneidet nur I und e schneidet I und III.
* Damit eine Startkante von Interesse ist, muss sie die maximale Startkante aller Timelines sein.
* Die Anfangskante von I ist c, von II ist a, von III ist d und von IV ist e. Daher ist die maximale Anfangskante e.
* Für eine Endkante muss es die minimale Endkante des ersten freien Slots in allen TimeLines sein.
* Die Endkante von I ist f, von II ist b, von III ist g und von IV ist auch g. Daher ist die minimale Endkante b.
* Da die minimale Endkante vor der maximalen Startkante liegt (b < e), gibt es keinen passenden Slot für die ersten freien Lücken auf jeder TimeLine.
* Überspringen Sie für jede TimeLine alle Lücken, die vor oder gleichzeitig mit der aktuellen maximalen Startkante enden.
* In diesem Fall muss die erste Lücke in TimeLine II übersprungen werden.
* Beginnen Sie nun erneut mit der Suche nach der maximalen Startkante…​ und wiederholen Sie den Vorgang.
* Die neue maximale Startkante ist j und die minimale Endkante ist k. Da k größer als j ist, haben wir unseren ersten "passenden freien Slot" gefunden.
* Wiederholen Sie den Vorgang, bis eine der TimeLines keine verfügbaren TimeSlots mehr hat.

Es sind verschiedene Implementierungen möglich. Denken Sie zuerst darüber nach, "was" getan werden soll, und schreiben Sie Pseudo-Code, denken Sie dann über das "Wie" nach und beginnen Sie mit dem Codieren.

## Ihre Aufgaben pro Woche

* Woche 2
  + Lesen Sie die API sorgfältig durch! Nehmen Sie sich mindestens eine Stunde Zeit, um es zu studieren und den gegebenen Code zu studieren (auf Ihren Händen sitzen, nicht Code schreiben!).
  + Testgetriebene Entwicklung der Implementierung von AppointmentData und AppointmentRequest.
  + Testgetriebene Entwicklung der Implementierung der Methoden in LocalDayPlan (einfache Getter):
    - getDay()
    - earliest()
    - tooLate()
    - getTimeline()
    - getStartTime()
    - getEndTime()
* Woche 3
  + Testgetriebene Entwicklung der Implementierung der Methoden in LocalDayPlan:
    - toString()
    - getGapsFitting(Duration dur )
    - canAddAppointmentOfDuration(Duration dur )
* Woche 4
  + Testgetriebene Entwicklung der Implementierung der Methoden in LocalDayPlan:
    - getNrOfAppointments()
    - contains(Appointment appointment)
    - addAppointment(AppointmentData appointmentData, LocalTime startTime )
* Woche 5
  + Testgetriebene Entwicklung der Implementierung der Methoden in LocalDayPlan:
    - getGapsFittingReversed(Dauer dur)
    - getGapsFittingLargestFirst(Duration dur )
    - getGapsFittingSmallestFirst(Duration dur )
    - addAppointment(AppointmentData terminData, LocalTime start, TimePreference fallback )
    - addAppointment(AppointmentData appointmentData, TimePreference pref )
* Woche 6
  + Testgetriebene Entwicklung der Implementierung der Methoden in LocalDayPlan:
    - getAppointments()
    - findAppointments(Predicate<Appointment> filter )
    - removeAppointment(Appointment appointment)
    - removeAppointments(Predicate<Appointment> filter)
* Woche 7
  + Testgetriebene Entwicklung der Implementierung der Methoden in LocalDayPlan:
    - getMatchingFreeSlotsOfDuration()
    - Andere Methoden, die oben möglicherweise nicht erwähnt wurden.