Available online at www.sciencedirect.com
SCIENCE DIRECT®

Computer Languages, Systems & Structures 32 (2006) 157 – 183

COMPUTER LANGUAGES, SYSTEMS & STRUCTURES
www.elsevier.com/locate/cl

A point based model of the Gregorian Calendar

Hernán Wilkinson[a, *, 1], Máximo Prieto[b], Luciano Romeo[a, 2]

[a]Tacuarí 202, 7mo Piso, C1071AAF, Buenos Aires, Argentina
[b]Lifia - Facultad de Informática, Universidad Nacional de La Plata, cc11, 1900, La Plata, Argentina

Received 3 October 2005

Abstract

Time is an important aspect of all real world entities; temporal information is crucial in many computer-based applications. The Smalltalk community does not have a good model of the time domain. Smalltalk-80 and its commercial implementations provide only the classes **Date** and **Time** to model time domain entities. Squeak augmented the model with the abstractions **Timespan, Year, Month** and **Week**. These models fall short when complex situations of the time domain have to be programmed, forcing the programmers to create their own and repetitive solutions. In this paper, we present a model of the Gregorian Calendar based on a metaphor that maps time entities into points of lines, each line with its own resolution. The model addresses a great amount of functionality and reifies almost all the Gregorian Calendar entities. It allows programmers to design and program time related issues better than current time domain implementations, and in a more natural way.
© 2005 Elsevier Ltd. All rights reserved.

Keywords: Smalltalk; Date; Time; Gregorian calendar; Time span; Time intervals; Time line filter; Relative dates; Test driven development

1. Introduction

Time entities are an important aspect of many computer applications. For example, the financial domain has a strong coupling with the time domain because the value of any financial instrument is related to a certain point in time (i.e., the value of one Euro today is not the same as it was two years ago), financial operations among traders could be settled some time after a given date (i.e., 48 h after today), instrument cash flows depend on dates relative to a certain calendar, and so on. Office information systems depend on time information to pay salaries, allow employees to leave on vacation, etc. Real time systems base their behaviour on timed events, verify the temporal evolution of the environment they control, etc.

Different types of temporal entities exist, such as

- Specific points in the time line, such as 01/01/2005 (defined as anchored data by [1]).
- Measures of time, such as 1 day (defined as unanchored temporal information by [1]).
- Temporal information about occurred events, such us “John played his guitar while Paul was outside” [2].

* Corresponding author. Tel.: +54 11 4878 1116x120.
E-mail addresses: h.wilkinson@mercapsoftware.com (H. Wilkinson), maximo.prieto@lifia.info.unlp.edu.ar (M. Prieto), l.romeo@mercapsoftware.com (L. Romeo).
[1] Mercap Development Manager.
[2] Mercap Software Architect.

1477-8424/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2005.10.001

Many time models have been proposed in the past [1,3–6] but none of those models are provided within the Smalltalk environments. Most of them are related to other technologies such as relational databases or artificial intelligence systems. Works such as [7] and [8] propose changes to the ODMG [9] object model adding temporal tracking to objects, but they do not augment the ODMG time model which lacks important time abstractions. Other programming languages such as Java [10] and .NET [11] provide basic time models that suffer from important design flaws as we show in Section 6.3, Comparison with other Research Work.

Barbic et al. in [12,13] classifies temporal systems in two categories, those that model *Time Representation* and those that model *Time Reasoning*. The former deals with the "*representation*" of time entities (time points vs. time intervals), *time ordering* (linear, circular or branching), *time boundedness* (i.e., modelling of finite or infinite times) and *time measures* (distance between time entities, arithmetic on those measures). The latter focuses on the specifications of a *time calculus* to manage temporal information and a *query language* to extract temporal information about time events.

We present in this paper an object model that focuses on the *Gregorian Calendar Time Representation*, implemented with Smalltalk, which provides abstractions for many of the time domain entities that are not model in current implementations and faces many design challenges such as

- Gregorian Calendar irregularity (Months have 28, 29, 30 or 31 days; there are leap and non-leap years)
- Time entities comparison consistency. We want to express comparisons as
  - January < July
  - January first < February twentyNinth
  - 3 months < 1year or 5 days < 1 week
- Time entity distance awareness, no matter the type of the entity
  - (January distanceTo: July) = (January, 2005 distanceTo: July, 2005)
- Relative dates according to a filter applied to the time line
  - 14 days from: (April first, 2005) counting: workingDays

The model is based on a metaphor that proposes to see time entities as points of the time line with four different resolution and it uses Measures [17] to represent the distance between two points in the time line, not just numbers as is commonly done in other models.

The scope of the model is limited to the Gregorian Calendar decreed by Pope Gregory XIII [19]. No support is given neither for the Hindu Calendar nor for the Iranian one or any other calendar. See [19] for a complete description of these calendars.

The remaining of this paper is organised as follows: Section 2 expands the problem we present in this paper. Section 3 presents the metaphor we based the model on. Section 4 discusses the model's design and behaviour. Section 5 presents the implementation. Section 6 compares the model with other time related models. Finally, Section 7 concludes the paper and gives directions for future research.

2. The problem

Smalltalk-80 [14] provides a basic time model implementation of the Gregorian Calendar. That model does not provide good solutions to all possible time related problems mainly because:

- It lacks proper abstractions of some important time domain entities (i.e., month, day)
- Time objects are not immutable (i.e., Time) therefore, they do not properly model time entities as we show in Section 4.1, Time Entities Immutability and Validity.

Smalltalk-80 provides only two classes to model time entities: **Date** and **Time**. Instances of **Number** and **Symbol** are used to represent time entities such as days, months, months of years, etc. For instance, the message #*year* implemented in **Date** returns a **Number** not an object that reifies an entity "year". The same is also true with the message #*day*, it returns a **Number** representing the day number not a "day". To obtain the month of a **Date** it is even harder because the model does not have a month class. **Date** provides two messages to accomplish that requirement, #*monthName* and #*monthIndex*. The former returns a **Symbol** (i.e., #February) and the latter a **Number** representing the position of that month in a Gregorian year (i.e., 2 for February).

The lack of abstractions makes the model difficult to use when complex time related calculations and situations need to be programmed. For instance, the Smalltalk-80 model does not easily solve the problem of getting the number of days of a month because the object that represents a month is a **Symbol** or a **Number** and neither of them answers the message *#numberOfDays*. Class protocol is provided in **Date** to answer that question with the message *#daysInMonth: aMonthName forYear: anInteger* but we argue that the class **Date** should not be responsible for this behaviour. A better solution would reify the "month of year" concept providing to this abstraction the necessary behaviour.

Comparing days of week is another problem generated by the lack of abstractions. Because days of week are modelled as instances of **Symbol**, the alphabetic order is used when they are compared. Code Sample 1 shows these examples.

```
"Note that the message #daysInMonth:forYear: is sent to the class Date"
today := Date today.
Date daysInMonth: today monthName forYear: today year.
"It returns false"
#Monday < #Friday
```

Code Sample 1. Getting the number of days of a year's month.

The Chronology package [15] released with Squeak 3.7 [16] provides abstractions proposed by the ANSI Standard [20] like the class **DateAndTime** and the class **Duration**. It also reifies concepts like **Year**, **Month** and **Week**, implemented as subclasses of **Timespan**, but:

- It lacks a good separation between anchored and unanchored time entities.
- It represents time entities as segments of the time line, not as points.
- It does not model important time entities such as month (i.e., January) and day (i.e., Monday).

Squeak's lack of abstractions, although less that Smalltalk-80, produces the same problems. The meaning of some abstractions can, at first, produce misinterpretations such as the class **Month,** which does not represent a month (i.e., January) but a month in a year (i.e., January 2005). But the main problem we found with this model is that time entities are modelled as segments in the time line; all the time classes are subclasses of **Timespan**. This modelling decision merges two different concepts, time points and time segments in one, which allows comparing entities of different granularity such as years and dates (i.e., year 2005 and January 2nd of 2005).

Representing time entities as time segments does not allow to create a total order among them, as it is explained in Goralwalla et al. in [1]. Therefore, the result of comparing time entities could be "unknown" (i.e., year 2005 is not less, equal or greater than January 2nd of 2005) and the "unknown" entity is not modelled in Squeak. Code Sample 2 shows that the year 2005 is not less, greater or equal to January 2005, a side effect of the inability of creating a total order between time entities.

```
"All these comparisons return false in Squeak"
(Year year: 2005) < (Month month: 1 year: 2005).
(Year year: 2005) > (Month month: 1 year: 2005).
(Year year: 2005) = (Month month: 1 year: 2005).
```

Code Sample 2. Inability of creating a total order of time entities side effect.

Other models like the one used by Java [10] and .Net [11] have one or a few general purpose abstractions. In Java for example, the class Calendar represents dates, months, years, etc.

The main drawback of all these models is that they do not provide abstractions for all the entities that we can observe in the time domain, which indicates, mainly, lack of understanding and therefore, they provide a poor domain language. This implies:

- It is difficult to express common situations with them.
- They are difficult to learn.
- They offer different possible interpretations.

Because software is knowledge represented in a computable model, object models should provide an abstraction for each observed entity of the problem domain. Lacking abstractions means incomplete knowledge. Incomplete knowledge leads programmers to fill the gaps between the problem domain and its model with solutions that end up producing:

- Code duplication.
- Ad-hoc implementations.
- Error prone situations.

Object models with the right abstractions are more reusable and easier to use. Due to the limitations of the existing models shown in this section we decided to create a new model of the Gregorian Calendar reifying as much time entities as we observed from reality.

3. The metaphor

We used a metaphor to understand the time domain. In this metaphor, time entities are points in a line, a line that represents the passing time. The observers of that line can zoom in and out the points it contains. When the observer zooms in she sees smaller points (i.e., dates), when the observer zooms out she sees bigger points (i.e., years). We say that the time line has different scales or that time lines of different scale can represent the passing time.

Let us see an example. A year represents a point in time but with less resolution than a date. If the year is zoomed in, new points are observed; those points are the months of that year. If one of those points is picked and zoomed in, the points representing the dates of that month are obtained. If one of these dates is selected and zoomed in, points representing the hour of that date are obtained. Let us do it with concrete entities. If the year 2005 is selected and zoomed in, months from January of 2005 to December of 2005 appear. If January of 2005 is zoomed in, dates from January 1st of 2005 to January 31st of 2005 are seen. If January 1st of 2005 is zoomed in, the entities January 1st of 2005 at 00:00:00 to January 1st of 2005 at 23:59:59 are seen. See Fig. 1 for a graphical representation.

The inverse happens when zooming out. If an hour of a day is zoomed out, a point representing its date will be obtained. If that date is zoomed out, a point representing the month where that date belongs to will be obtained. If that point is zoomed out, the year that the month belongs to will be obtained.

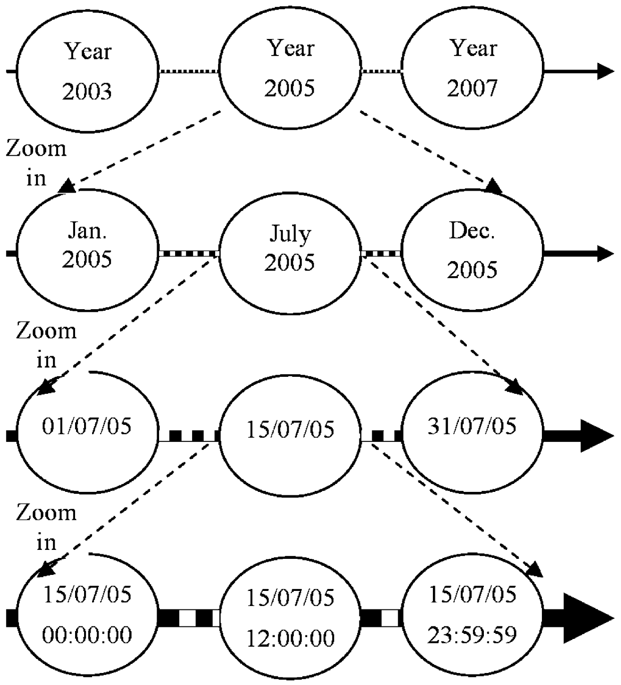

Fig. 1. Zooming in and out in the time line.

The points that can be obtained at the different scales of the time line are abstractions representing years (i.e., year 2005), months of a year (i.e., January of 2005), dates (i.e., 01/01/2004) and the time of a given date (i.e., 01/01/2005 at 00:00:00).

Not only the metaphor helped us to understand the problem and create a model based on it, but it also allowed us to easily define the expected behaviour of the model, such as

- Determine which point comes before or after another (ordering of time points along the time line).
- Go from one point in the time line to another.
- Obtain the distance between two time points.
- Represent segments of the time line of any scale.
- Switch from one scale to another.
- Represent intervals between points.
- Filter the time line with certain rules.

4. The point based model

Based on the metaphor and trying to reify as much as time entities we observed in the Gregorian Calendar, the model provides the following abstractions:

- **Years**: Modelled with the class **GregorianYear**. This class is used to represent years such as the year 2005, the year 2000, etc.
- **Months of a Year**: Modelled with the class **GregorianMonthOfYear**. This class represents entities like January of the year 2005, December of the year 2000, etc.
- **Dates**: Modelled with the class **GregorianDate**. It is used to represent entities such as 31/12/2005, which is December 31st of 2005. Note that we use the DD/MM/YYYY notation.
- **Relative Dates**: Modelled with the class **RelativeGregorianDate**. Used to represent dates that can change depending on different time events (i.e., working or none working days).
- **Time of a given Date**: Modelled with the class **GregorianDateTime**. This class represents entities such as 01/01/2005 at 10:00:00, that is, ten in the morning of January 1st of the year 2005.
- **Days of a Month**: Modelled with the class **GregorianDayOfMonth**. This class represents entities such as January 1st, December 25th, etc. Note that these are days of given months but of no particular year.
- **Months**: Modelled with the class **GregorianMonth**. Months are January, February, March, etc.
- **Days**: Modelled with the class **GregorianDay**. Days are Sunday, Monday, Tuesday, etc.
- **Time of a Day**: Modelled with the class **TimeOfDay**. It represents the time in a day such as 10 AM, 12 PM, 9:15:35 (this is quarter past nine and thirty five seconds).
- **Segments of the time line**: Modelled with the class **Timespan** (i.e., 10 days from now).
- **Time point intervals of different granularity and resolution**: Modelled with the class **MeasurementInterval** (i.e., from 01/01/2005 to 20/01/2005 every 3 days).
- **Time line filters**: Modelled with the class **TimelineFilter**. Used to mark time points according to some criteria (i.e., working day, non working day).

4.1. Time entities immutability and validity

Something we have noticed about time entities is that they are immutable; they do not change, they are immutable like the numbers. A given date such as *January 1st of 2005* should not allow its year, month or day to be changed. Therefore, the abstractions we use to model the time domain entities are immutable, they behave like “value objects” (see [21]). Immutable objects allow us to have a simpler model and not to worry about inconsistent objects, invalid modifications or invariance invalidity during a certain time.

The model also verifies, when creating an object, if the new instance will be valid. If that is true, the object is created, otherwise an exception is signalled. Therefore, the code that verifies if an object is valid is located in one place and ensures that no invalid time objects exist.

For example, the year zero is an invalid year on the Gregorian Calendar, and trying to create an object for the year zero is a semantic error, so we check that rule when trying to create an instance of **GregorianYear**. See Code Sample 3.

```
GregorianYear class≫number: aNumber
^(self isValidYearNumber: aNumber)
    ifTrue: [ create the instance ]
    ifFalse:[InvalidGregorianYearNumberException signalNumber: aNumber ].

GregorianYear class≫isValidYearNumber: aNumber
    ^aNumber ~ =0 and: [aNumber isInteger]
```

Code Sample 3. Verifying the creation of an instance of a year.

Because **GregorianYear** is immutable, no instance message is provided to set the number of the year. If **GregorianYear** were not immutable, the setter method *#number:* would have to perform the same verification as the *#number:* class method. This verification is not difficult to do with years, but what about dates? If we provide a message to change the day number, its implementation should verify that the day number is valid for the month and year the date already represents. But, what happens if it is temporarily invalid because the next collaboration modifies the month making the new day number valid? There is no way to maintain the validity of the date invariants if we provide messages to modify its day number, month or year.

A message could be provided to completely change a date such as *#yearNumber: aYearNumber monthNumber: aMonthNumber dayNumber: aDayNumber*, but that message would be the same as that one sent to the class to create a new instance as Code Sample 4 shows.

```
"Creates the date 28/2/2005"
aDate := GregorianDate yearNumber: 2005 monthNumber: 2 dayNumber: 28.
"Setting the day number to 31 should signal an exception"
aDate dayNumber: 31.
"But if the month is changed to be January the previous day number would be valid ..."
aDate monthNumber: 1.
"A message to change the year, month and day number could be provided, but
it is the same as the one the class responds to"
aDate yearNumber: 2005 monthNumber: 1 dayNumber: 31
```

Code Sample 4. Verifying the creation of an instance of a year.

4.2. Different scale time line traversal

As we said before, a year can be seen as a point in the time line at a year resolution. Because the resolution is a year, that point contains other points of higher resolution such as months of a year, dates and time in a certain date. The model provides protocol to easily move between points of different resolutions (i.e., going from a year to the dates it contains or from a date to its year). Moving to points of smaller resolution looks natural (i.e., going from a date to its year) but moving to points of higher resolution is not so commonly provided on this type of models (i.e., going from a year to its dates). Messages to go from points of one scale to another are provided on each abstraction. See Code Sample 5 for an example.

```
aYear := GregorianYear number: 2005.
"Going from years to months of year"
aYear firstMonth.            "Returns January of 2005"
aYear lastMonth.             "Returns December of 2005"
aYear months.                "Returns all the months of year 2005"
"Going from years to dates"
aYear firstDate              "Returns 01/01/2005"
aYear lastDate               "Returns 31/12/2005"
aYear dates                  "Returns the 365 dates of the year 2005"
aYear firstDay                          "Returns Saturday"
aYear lastDay                           "It is also a Saturday"
"Going from years to date times"
aYear firstDate atMidnight              "Returns 01/01/2005 00:00:00"
aYear lastDate lastTimeOfDay            "Returns 31/12/2005 23:59:59"
```

Code Sample 5. Moving from a year to other entities.

4.3. Magnitude protocol

All the time point abstractions respond to the magnitude protocol with messages such as # < , # < =, # > , # > =, #*min*: , #*max*: , #*between*: *and*: among others. Because they are points in the time line of a certain resolution, they can be compared to see which one is closer or farther from the beginning of the time line. A total order can be defined for them.

```
(GregorianYear number: 2005) < (GregorianYear number: 2010)    "Comparing years"
(December of: 2004) < (July of: 2005)                           "Comparing month of year"
GregorianDate today < GregorianDate tomorrow                    "Comparing dates"
GregorianDateTime now < GregorianDateTime now next              "Comparing datetimes"
```

Code Sample 6. Comparing points on the time line.

Not only points on the time line can be compared. Instances of **GregorianDay**, **GregorianDayOfMonth** and **GregorianMonth** can also be compared. When comparing days of the week the model assumes Sunday is the first day of the week but this can be changed to any other day such as Monday. January 1st is always the first **GregorianDayOfMonth** and January is always the first **GregorianMonth**. Code Sample 7 shows how to compare these objects.

```
Monday < Tuesday                          "Comparing days"
January < December                        "Comparing months"
January first < December twentyFifth      "Comparing days of month"
```

Code Sample 7. Comparing other time entities.

Comparing points of different resolution can end up being "unknown". For example, the year 2005 is not less, equal or greater than January 2nd of 2005. Different approaches were proposed to solve this problem. [1] and [3] propose to return "unknown" for this type of comparison. Squeak does not return unknown but it can be inferred because all the comparison messages (# < , #= and # <) return *false* when they are sent to objects under this situation. We propose a different solution where the comparison between points of different resolutions is not allowed and, if such an attempt is made an exception is signalled.

This decision is based on the metaphor used to create the model and an analogy we made with points and sets. Because points in the time line are composed by other points, they can be considered analogous to sets. For example, a year is a point that contains the months of that year. We think that comparing a year (seen as a set of its months) with a month of that year (an element of that set) is a semantic mistake because it is analogous to compare a set with elements of that set.

Propositions such as "Is the year 2004 before January 1st of 2005?" are seen as valid because only a comparison at the year resolution is necessary to answer that question, only the year 2004 and the year 2005 are compared. The problem with this type of comparison arises when comparing a year with a month of that same year such as "Is the year 2005 before March of 2005?" Because March of 2005 is part of the year 2005, it is neither before, after nor equal to the year 2005, but included in it.

4.4. Obtaining the distance between two points

Time models should provide ways to know the number of years between two years, the number of months between two months of a year, and so on. This is analogous to obtain the number of points between two points of the same time line resolution.

Messages #*distanceTo: aPoint* and #*distanceFrom: aPoint* are used to obtain the distance between two points. The same messages are used polymorphically for years, months of a year, dates, etc. The model does not provide the message

Table 1
Time units provided by default

Unit	Type	Measure example	Conversion example
month	Base Unit	10 months	(12 months convertTo: year) = 1 year
year	Derived from month	2 years	(2 years convertTo: month) = 24 months
decade	Derived from month	1 decade	(1 decade convertTo: year) = 10 years
century	Derived from month	2 centuries	(2 centuries convertTo: decade) = 20 decades
millennium	Derived from month	1 millennium	(1 millennium convertTo: century) = 10 centuries
millisecond	Base Unit	1000 milliseconds	(1000 milliseconds convertTo: second) = 1 second
second	Derived from millisecond	60 seconds	(60 seconds convertTo: minute) = 1 minute
minute	Derived from millisecond	60 minutes	(60 minutes convertTo: hour) = 1 hour
hour	Derived from millisecond	24 hours	(24 hours convertTo: day) = 1 day
day	Derived from millisecond	7 days	(7 days convertTo: week) = 1 week
week	Derived from millisecond	2 weeks	(2 weeks convertTo: day) = 14 days

#- (minus) to get the distance between two points because it does not behave like the subtraction operation. When the message #- is sent to a **Number**, it returns another **Number**, but the distance between two points in the time line is not of the same type of the points; it is a measure. Due to this observation, we decided to use a different protocol for this kind of inquires. The model also provides behaviour to obtain the distance between time entities like days, months and days of months.

```
(GregorianYear number: 2005) distanceTo: (GregorianYear: 2010)     "Returns 5 years"
(GregorianYear number: 2005) distanceTo: (GregorianYear: 2000)     "Returns -5 years"

January first, 2005 distanceTo: January tenth, 2005                "Returns 10 days"
January first, 2005 distanceFrom: January tenth, 2005              "Returns -10 days"
```

Code Sample 8. Getting the distance between two points.

4.5. Time units and time measures in the time domain

Objects returned by the distance messages are not numbers but time measures. Some models provide abstractions for such entities like [3] and [1], others just do not reify them like Smalltalk-80 and Squeak, where raw numbers are used to represent them. This model reifies them reusing another model we created, one used to represent any kind of measure. In such model, a measure is modelled as a number together with a unit.

The advantages of using measures over raw numbers are explained in [17,22,23] and they are:

- It gives numbers a meaning. Numbers without units have no meaning, they are just symbols
- It allow programmers to rely on the computer about the validations made during arithmetic operations
- It provides a better model of reality, concerning measures.

The object "*10 days*" represents in a better way the distance between days than just the number "*10*". People could argue that in reality, when they are asked how many days there are between two dates, i.e., how many days are between January 1st and January 10th, they just respond with a number, i.e., 9. That is true, we "say" a number but that number has implicit knowledge attached to it due to the context of the question that has been asked. Its meaning is not just *9*, but *9 days*.

This model provides different units to create all the possible measures of the Gregorian Calendar. These units are organised in two different categories due to the irregularity of the Gregorian Calendar. The base unit for each category is *month* and *millisecond*. Table 1 shows the units provided by default with the model; new units can be created as needed.

Note that converting measures of different scales is not always feasible due to the irregularity of the Gregorian Calendar; [1] also explains this limitation. In this model, measures can be automatically converted if they share the same base unit. A measure of years can be converted to months, decades, centuries and millenniums because they share

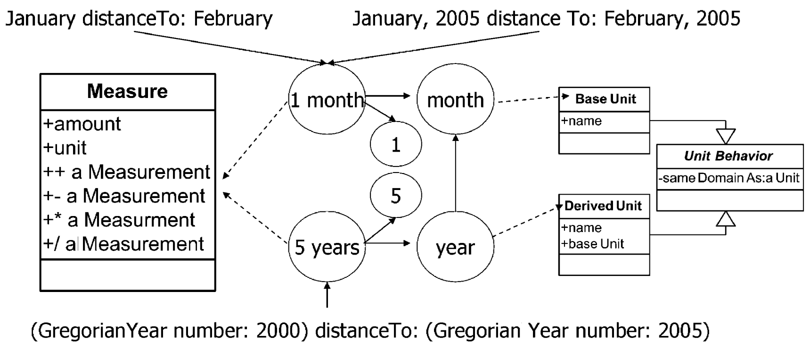

Fig. 2. Objects and classes of the measure model.

the same base unit, *month*. Automatic conversion between milliseconds, seconds, minutes, hours, days and weeks is also possible because they share the same base unit, *millisecond*.

A measure of years cannot be converted to days because the conversion could be *366 days* or *365 days* per year due to the existence of leap years in the Gregorian Calendar. The same applies to months. A month cannot be converted to days because it could represent *28*, *29*, *30* or *31 days*. This does not mean that a specific year or month of year cannot be asked for the number of days it contains. Instances of **GregorianYear** and **GregorianMonthOfYear** respond to the message *#numberOfDays*, which returns a time measure (i.e., *29 days* if the month of year is February 2004 and *28 days* if the month of year is February of 2005).

Fig. 2 shows how measures and units are related and modelled.

Because the time model uses the measure model, new time units can be created as needed. For example, the *quarter* of a year unit can be created as derived from *month* as shown in Code Sample 9.

```
month := BaseUnit nameForOne: 'month' nameForMany: 'months' "This unit is provide with the model"
quarter := DerivedUnit from: month
 nameForOne: 'quarter' nameForMany: 'quarters'
 conversionFactor: 3
```

Code Sample 9. Creating a new time unit.

It is also possible to mathematically operate with time units because the measure model provided with this one supports the basic arithmetic operations +, −, * and / among others. Because time units are reified, any kind of measure composed with time measures can be created, such as *100* Km/h (a measure of speed) or *10%/*month (an interest rate of 10% by month). Code Sample 10 shows some examples. Refer to [17] for a complete explanation of this behaviour.

```
14 days + 1 week = 1814400000 milliseconds.     "Adding measures of the same base unit"
((14 days + 1 week) convertTo: days) = 21 days. "Converting the result of an operation"
(1 year + 10 days) = (1 year + 10 days)         "Adding measures of different base unit"
10 years*10 = 100 years                         "Multiplying a measure by a number"
10 years*12 months = 10 year*year               "Multiplying measures"
10 years*12 months/24 months = 5 years          "The model automatically simplifies units"
100 kilometers/1 hour                           "Represents a speed of 100 km per hour"
0.01/1 month                                    "Represent an interest rate of 10% by month"
```

Code Sample 10. Arithmetic with time measures.

4.6. Moving through points of the same time line resolution

The model provides the *#next*, *#next: aMeasure*, *#previous* and *#previous: aMeasure* messages to move certain distance from a given point. *#next* and *#previous* messages assume that the distance to move is equal to the quantum of the time line the point receiving the message belongs to. If the point is a year, the quantum is *1 year*, if the point is a month of a year the quantum is *1 month*, if the point is a date the quantum is *1 day* and if the point is a date time the quantum is *1 millisecond*.

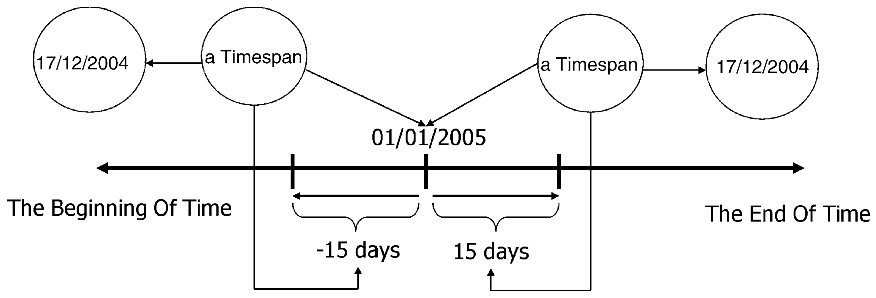

Fig. 3. Two different time line segements.

Moving certain distance from a point expects a measure of time as parameter because the distance between two points is expressed as a measure of time.

```
(GregorianYear number: 2005) next             "Returns GregorianYear number: 2006"
(GregorianYear number: 2005) next: 1 year     "Returns GregorianYear number: 2006"
(GregorianYear number: 2005) next: 12 months  "Returns GregorianYear number: 2006"
(GregorianYear number: 2005) next: 10 years   "Returns GregorianYear number: 2015"
(GregorianYear number: 2005) previous: 5 years "Returns GregorianYear number: 2000"
```

Code Sample 11. Moving on the same time line resolution.

At the moment this paper was written moving a certain distance expressed in a unit not convertible to the unit of the quantum of the point signals an exception. We found this behaviour to be too restricted when dealing with some financial observations. In Section 7.2, future work, we show some ideas to solve this problem. Code Sample 12 shows examples of how the model behaves at the time this paper was written.

```
(GregorianYear number: 2005) next: 120 days "Signals an exception because 120 days cannot
                                             be converted to years"
(January of: 2005) next: 120 days           "Signals an exception because 120 days cannot
                                             be converted to months"
```

Code Sample 12. Moving on the same time line resolution.

The model also provides protocol to move through time entities that do not belong to any time line but have an order such as days, months and days of month.

```
Monday next: 4 days              "Returns Friday"
January next: 2 months           "Returns March"
January first next: 2 days       "Returns January 3rd"
```

Code Sample 13. Moving from days, months and day of months.

4.7. Segments of the time line

The class **Timespan** represents time line segments. A segment begins on a specific point of the time line and has certain duration and direction expressed as a measure. The starting point of a time span can be a point at any of the time line resolutions. The duration and direction is given by a time measure that should be convertible to the unit of the scale the starting point belongs to. If the measure is positive, the direction is towards the end of time, if the measure is negative, the direction is towards the beginning of time. Fig. 3 shows a representation of time line segments.

Time spans are useful to represent relative time entities where the beginning of such an entity is known, but the end is not exactly known or can change. Examples of such entities are "I'll see you in 10 working days from today" or "it happened 7 months before January". Time spans are important to represent relative time entities such as relative dates which are explain further on.

Time spans can also be used with time objects that are not part of the time line but have an order such as days, months and day of months. Code Sample 14 shows examples of time spans.

```
"Creates a time span from January 1st of 2005 with 72 hours of duration"
aTimespan := Timespan from: (January first, 2005) duration: 72 hours.
aTimespan to. "Returns 4/01/2005"
"Creates a time span from year 2005 with a duration of 4 years"
aTimespan := Timespan from: (GregorianYear number: 2005) duration: 4 years
aTimespan to. "Returns year 2009"
"Creates a time span from now with a length of 3 weeks toward the beginning of time"
aTimespan := Timespan from: GregorianDateTime now duration : −3 weeks
aTimespan to. "If now is 01/01/2005 10:00:00, returns December 11th of year 2004 at 10 AM"
(Timespan from: GregorianDay today duration: 3 days) to.       "Returns Thursday if today is
                                                                Monday"
(Timespan from: GregorianMonth current duration: 6 months) to. "Returns July if the current month
                                                                is January"
```

Code Sample 14. Time spans creation and time spans of days, months and day of months.

4.8. Intervals

The model reifies the concept of intervals for time entities with an order. Those intervals behave like collections between the specified starting and ending point. Measures are used to specify the step of those intervals.

The same protocol used to create intervals of numbers is used to create intervals of time entities. For example, an interval between two years can be created sending the message *#to:anotherYear by: aDistance* to an instance of **GregorianYear**.

```
"Returns an Interval with eleven elements, the years between 2005 and 2015 inclusive".
(GregorianYear number: 2005) to: (GregorianYear number: 2015)
"Returns an Interval with six elements, the years 2005, 2007, 2009, 2011, 2013 and 2015 inclusive".
(GregorianYear number: 2005) to: (GregorianYear number: 2015) by: 2 years
"Returns an Interval with six elements, the years 2005, 2004, 2003, 2002, 2001 and 2000 inclusive".
(GregorianYear number: 2005) to: (GregorianYear number: 2000) by: −1 year
```

Code Sample 15. Interval creation.

Time intervals are polymorphic with number intervals, which at the same time behave as collections. Code Sample 16 shows some examples.

```
"Returns all the leap years between 2005 and 2100"
((GregorianYear number: 2005) to: (GregorianYear number: 2100)) select: [ :aYear | aYear isLeap ]
"Returns all Sundays between January 1st of 2005 and the last date of February 2005"
((January first, 2005) to: (February of: 2005) lastDate) select: [ :aDate | aDate isSunday ]
```

Code Sample 16. Using intervals.

The model also provides protocol to create collection of objects that are commonly used. See examples of Code Sample 17.

```
"Returns all the Tuesdays between January 1st of 2005 and June 30th of 2005"
(January first, 2005) to: (June thirtieth, 2005) everyDay: Tuesday
"Returns all dates whose day number is 10 between January 1st of 2005 and June 30th of 2005"
(January first, 2005) to: (June thirtieth, 2005) everyDayNumber: 10
"Returns all dates whose day numbers are 10 or 20 between January 1st of 2005 and June 30th of 2005"
(January first, 2005) to: (June thirtieth, 2005) everyDayNumbers: #(10 20)
```

Code Sample 17. Commonly used protocol.

The difference between time intervals and time segments is subtle. Time intervals are collections while time segments are not. Time segments cannot be iterated and they are not composed by a collection of time entities, they just have a beginning and a directed duration. Protocol to convert from a time interval to a time segment and vice versa is provided by the model.

4.9. Time line filters

The model reifies the concept of time line filter. A filter restricts the elements that belong to a time line using rules to specify which objects should be filter or not.

Deciding the days that are working or not working is a common use for these filters. For example, a filter can be created to mark all Saturdays and Sundays as non working days, another filter can be created to filter the months where the season changes, etc.

The model provides different types of rules, such as a rule for days (i.e., to include all Saturdays), a rule for a given day in a month (i.e., all the 25th of May), a rule for specific time entities and different rule decorators.

Filters behave like collections, so they can be iterated, they can be query for the inclusion of elements, etc. Code Sample 18 shows how to create a filter for non working days.

```
"Let's create a filter for all dates ..."
nonWorkingDays := TimelineFilter universe: (TheBeginningofTime to: TheEndofTime).
"Now, we want Saturdays to be on that filter"
nonWorkingDays addDayRule: Saturday.
"Now we want Sundays from January 1st of year 1000 to the end of time ..."
nonWorkingDays addDayRule: Sunday from: (January first, 1000) to: TheEndofTime.
"Now we want all July 9th since 1816 because is the Independence Day in Argentina".
nonWorkingDays addDayofMonthRule: July ninth from: (July ninth, 1816) to: TheEndofTime.

nonWorkingDays includes: (July ninth, 2005)          "Returns true"
nonWorkingDays includes: (July eighth, 2005)         "Returns false"
nonWorkingDays includes: (July sixteenth, 2005)      "Returns true, it is Saturday"
nonWorkingDays includes: (July seventeenth, 2005)    "Returns true, it is Sunday"
nonWorkingDays includes: (July eighteenth, 2005)     "Returns false, it is Monday"
```

Code Sample 18. Time line filters.

Filters can be really vast and impossible or too slow to iterate on them. The model provides streams whose responsibility is to move through an interval of the elements of the filter.

```
"Streams over the next 10 non working days, starting from today"
stream := TimelineStream from: GregorianDate today using: nonWorkingDays.
10 timesRepeat: [ stream next ]
```

Code Sample 19. Calendar streams.

Because time line filters are defined by rules, the inverse or negation of a filter is easy to obtain. A negated filter includes all the time entities that its original filter excludes and vice versa. When the message *#negated* is sent to a filter, its inverse is returned. As we shall see in the next section, negated filter are important in the financial domain.

4.10. Relative dates

In the financial domain, settlement dates are usually expressed as a distance from the trade date in a given calendar. For example, a trader can buy bonds on a Thursday, but the settlement date is set to happen within 48 h using the clearing house's calendar. That usually means that the trader's institution will receive the bonds on the next Monday, but this is true only if that Monday is a working day and it could have been true at the time the operation was done. But sometimes non-working days are created due to non-expected events (i.e., the death of some important person) and a working day is declared to be non-working.

In our example, if Monday is declared as non-working day, the new settlement date for the trade will be Tuesday. To model this new type of entity we created an abstraction called **RelativeGregorianDate** that is a date relative to a time line filter given a certain time span. See Code Sample 20 for an example. Note that the settle date is declared using the negated non-working days filter because settlements can occur only on working days.

```
"06/01/2005 is a Thursday"
aTimespan := Timespan from: (January sixth, 2005) duration: 48 hours.
aSettleDate := RelativeGregorianDate timespan: aTimespan calendar: nonWorkingDays negated.
"Returns false because 10/01/2005, a Monday, is a working day"
nonWorkingDays includes: (January tenth, 2005).
"Returns 10/01/2005"
aSettleDate absoluteDate.
"Now a new non working day is added to the filter"
nonWorkingDays addDateRuleFor: (January tenth, 2005).
"Return true. Now 10/01/2005, is a not working day"
nonWorkingDays includes: (January tenth, 2005).
"Now it returns 11/01/2005 because the filter has changed"
aSettleDate absoluteDate.
```

Code Sample 20. Relative dates.

Relative dates change according to the changes on the filter they are related to. Its instances are polymorphic with **GregorianDate**. Relative dates show the importance of reifying the time line segment.

4.11. Special time entities

The time line does not have a known end or beginning, but the mere fact that we, as human, can think on them means that they have to be reified. Two objects are provided to represent these entities. They are "*TheEndOfTime*" and "*TheBeginningOfTime*". The object "*TheEndOfTime*" is always greater than any point in time and "*TheBeginningOfTime*" is always less than any point in time.

These objects are useful to create open intervals towards infinite and minus infinite. They allow programmers to create intervals and filters on the whole time line and to create streams with no end. When using these objects, the programmer has to have special care because iterating over an interval with no end and/or beginning will never stop.

5. Model's implementation

PointInTime is the class that represents the abstract concept of a point in the time line. It is the superclass of all the concrete points of the time line such as year, date, etc., and it provides common implementation to the shared messages. Two methods have to be implemented by its subclasses, #*next*: and #*distanceTo*:. Messages such as #*previous*: and #*distanceFrom*: are implemented using them. **PointInTime** is a subclass of **IntervalAwareMagnitude,** which is an abstract class that provides common protocol and implementation to create intervals.

In Smalltalk 80, messages such as *#to:, #to:by:* and *#to:by:do:* are only implemented by **Number.** We extended the responsibility of creating intervals to all magnitudes. These intervals are instances of **MeasurementInterval**, they can be used with any **Magnitude** and they are polymorphic with **Interval**. Before a new instance of **MeasurementInterval** is created, the validity of the future interval is verified, and if it is not valid an exception is signalled. See Fig. 4.

5.1. Years

The lack of uniformity of the Gregorian Calendar has been modelled using classes to represent the special cases. For example, Gregorian years can be leap or non-leap, so there is a class representing leap years (**GregorianLeapYear**) and a class representing non-leap years (**GregorianNonLeapYear**). When the **GregorianYear** class receives the message #*number: aNumber* to create an instance of a Gregorian year, it verifies whether the number corresponds to a leap year or a non-leap year. If the number corresponds to a leap year it returns an instance of **GregorianLeapYear**, otherwise it returns an instance of **GregorianNonLeapYear**. The programmer should not care about a year's class, he just needs years to behave as expected.

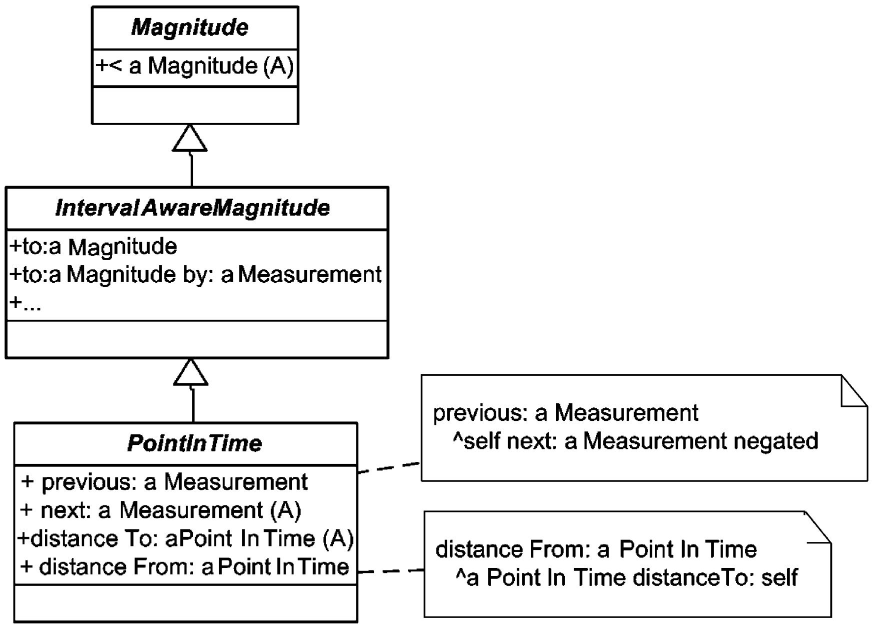
Magnitude
+< a Magnitude (A)
IntervalAwareMagnitude
+to:a Magnitude
+to:a Magnitude by: a Measurement
+...
PointInTime
+ previous: a Measurement
+ next: a Measurement (A)
+distance To: aPoint InTime (A)
+ distance From: a Point InTime
previous: a Measurement
^self next: a Measurement negated
distance From: a Point In Time
^a Point In Time distanceTo: self

Fig. 4. PointInTime abstract class.

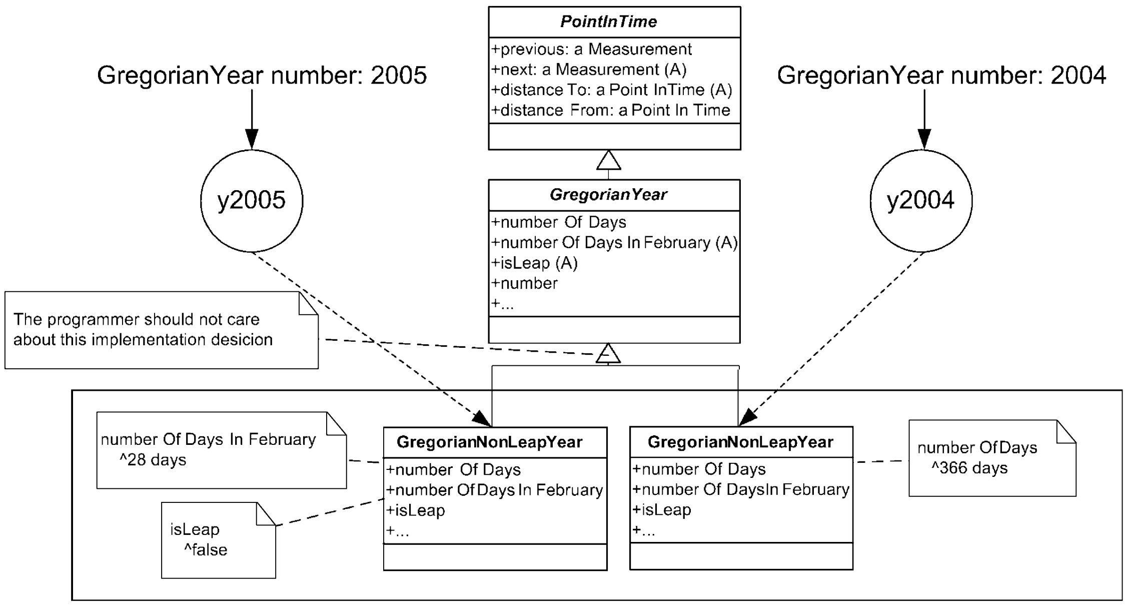
GregorianYear number: 2005
GregorianYear number: 2004
y2005
y2004
PointInTime
+previous: a Measurement
+next: a Measurement (A)
+distance To: a Point InTime (A)
+distance From: a Point In Time
GregorianYear
+number Of Days
+number Of Days In February (A)
+isLeap (A)
+number
+...
The programmer should not care about this implementation desicion
number Of Days In February
^28 days
isLeap
^false
GregorianNonLeapYear
+number Of Days
+number Of Days In February
+isLeap
+...
GregorianNonLeapYear
+number Of Days
+number Of DaysIn February
+isLeap
+...
number Of Days
^366 days

Fig. 5. GregorianYear class hierarchy diagram.

Because leap and non-leap years are reified, no conditional statement has to be used to implement messages such as *#numberOfDays*. If the year is leap, the message *#numberOfDays* returns *366 days*, if the year is not leap, the message *#numberOfDays* returns *365 days*. See Fig. 5.

5.2. Months and months of year

February is another example of the lack of uniformity of the Gregorian Calendar. Its number of days depends on the year. To solve this problem, we modelled months with an abstract class named **GregorianMonth** and specific

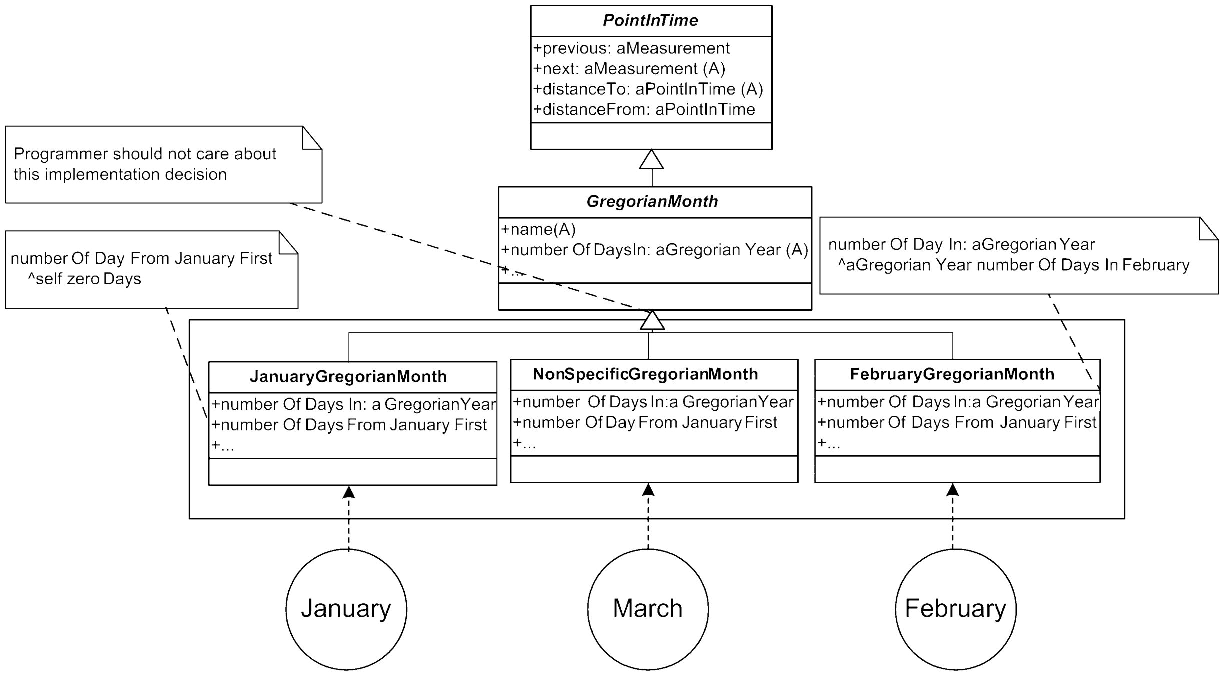
PointInTime
+previous: aMeasurement
+next: aMeasurement (A)
+distanceTo: aPointInTime (A)
+distanceFrom: aPointInTime
GregorianMonth
+name(A)
+number Of DaysIn: aGregorian Year (A)
+...
Programmer should not care about this implementation decision
number Of Day From January First ^self zero Days
number Of Day In: aGregorian Year ^aGregorian Year number Of Days In February
JanuaryGregorianMonth
+number Of Days In: a GregorianYear
+number Of Days From January First
+...
NonSpecificGregorianMonth
+number Of Days In:a GregorianYear
+number Of Day From January First
+...
FebruaryGregorianMonth
+number Of Days In:a GregorianYear
+number Of Days From January First
+...
January
March
February

Fig. 6. GregorianMonth class hierarchy diagram.

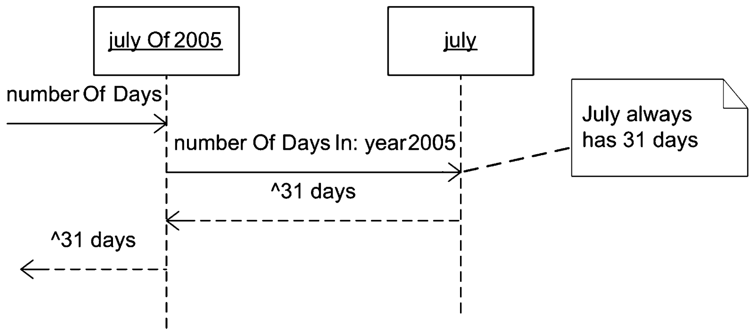

Fig. 7. Getting the number of days of a non specific Gregorian month.

implementations such as **FebruaryGregorianMonth**, **JanuaryGregorianMonth** and **NonSpecificGregorianMonth**. Months are referenced by the variables *January, February, March*, etc. and they can be obtained also sending messages to **GregorianMonth** such as *#january, #february, #march,* etc. Only one instance of each month exists. The programmer should not care about this implementation decision.

When a **FebruaryGregorianMonth** receives the message *#numberOfDaysIn: aGregorianYear*, it sends the message *#numberOfDaysInFebruary* to *aGregorianYear*. If that year is leap, it returns *29 days*, if it is non-leap, it returns *28 days*. Note that no conditional message has to be sent. When a **JanuaryGregorianMonth** receives the message *#numberOfDaysIn: aGregorianYear* it returns *31 days*. When a **NonSpecificGregorianMonth** receives that message it returns the object referenced by the instance variable *numberOfDays*. See Fig. 6.

A **GregorianMonthOfYear** references a **GregorianYear** and a **GregorianMonth**. When a **GregorianMonthOfYear** receives the message *#numberOfDays,* it only needs to send the message *#numberOfDaysIn:* to its *month* with the year referenced by its instance variable named *year* as parameter of the message. Implementing the irregularity of the Gregorian Calendar with specific abstractions for the special cases allowed us to minimize the use of the conditional message *#ifTrue:* to just one place, the creation of a year. Figs. 7–9 show how the objects interact to respond

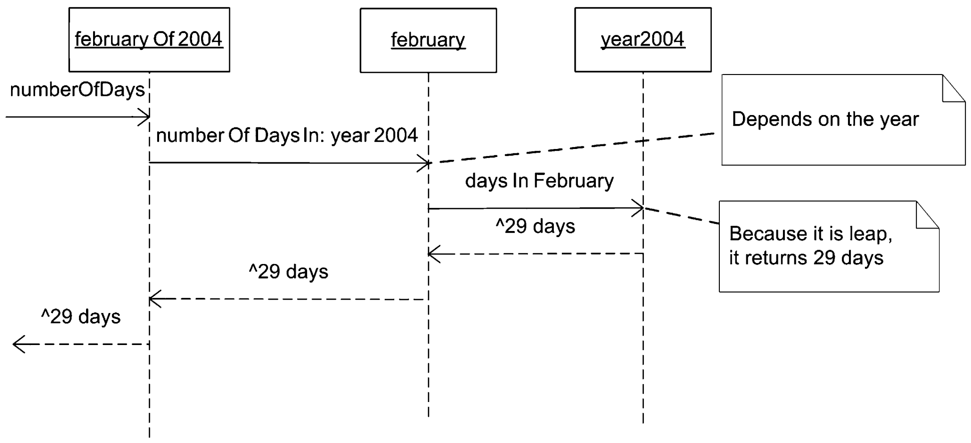

Fig. 8. Getting the February's number of days of a leap year.

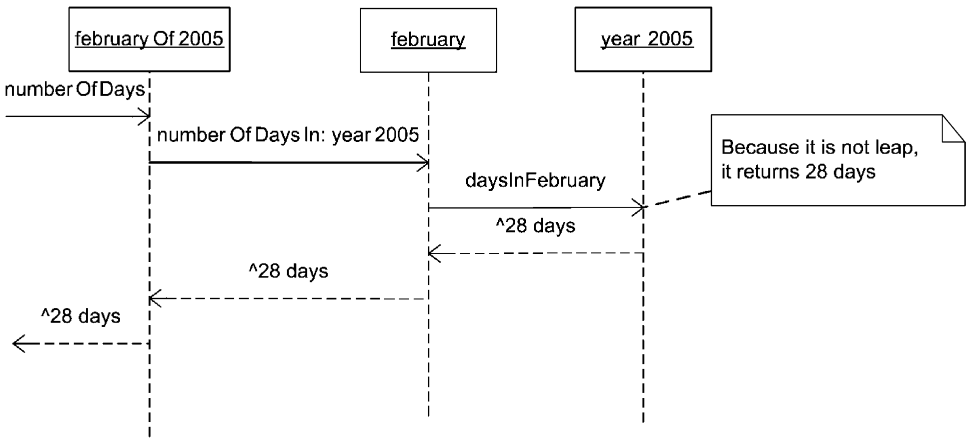

Fig. 9. Getting the February's number of days of a non leap year.

the message *#numberOfDays* when it is send to July 2005, February 2004 (a leap year) and February 2005 (a non-leap year).

5.3. Dates

Dates are modelled with the **GregorianDateBehavior** abstract class, that implements common messages for all dates, no matter if they are absolute or relative. **GregorianDate** represents absolute dates and **RelativeGregorianDate** represents relative dates in a time line filter with certain time span. The implementation of *#next:aMeasure* differs on each class. The **GregorianDate** class implements this message moving through the dates of the continuous time line, but the **RelativeGregorianDate** class uses its calendar (an instance of **TimelineFilter**) to obtain the dates it has to jump through when moving. The message *#distanceTo:aGregorianDate* is implemented in **GregorianDateBehavior** because it can be shared by its subclasses. See Fig. 10.

Fig. 11 shows an object diagram of a **RelativeGregorianDate** that represents 10 working days from today, with today equals to July 18th of 2005.

5.4. Other time entities

A **GregorianDateTime** is composed by a date (instance of **GregorianDate** or **RelativeGreogrianDate**) and a time (instance of **TimeOfDay**). Because the date can be relative, the model also supports relative date times.

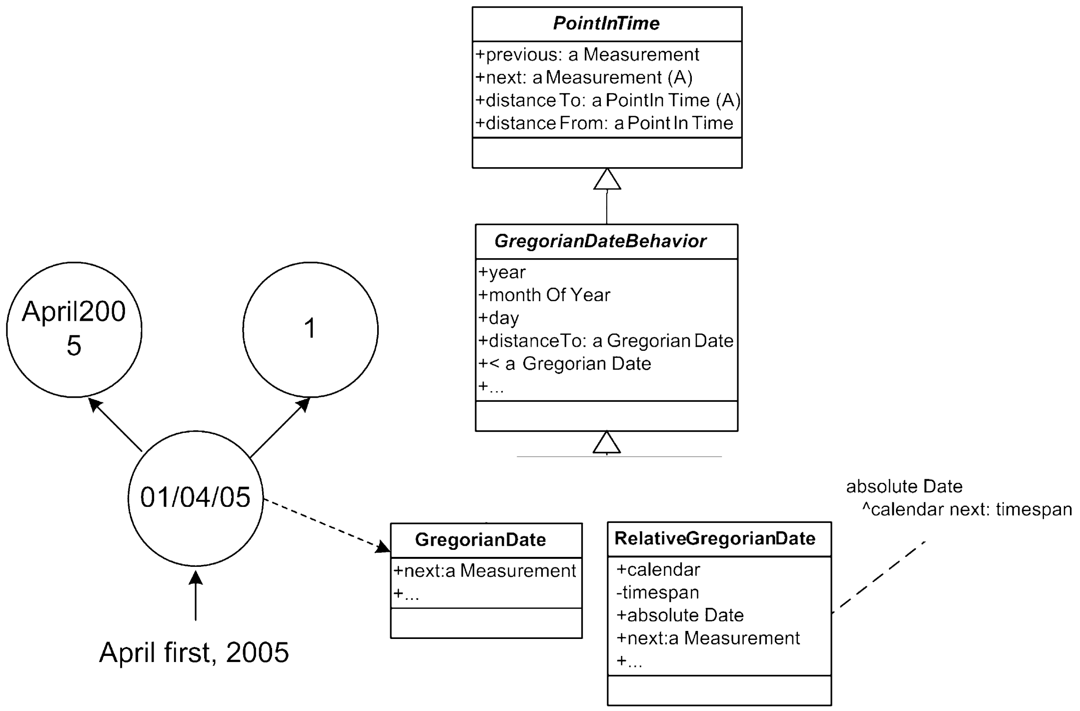

Fig. 10. GregorianDate class hierarchy diagram.

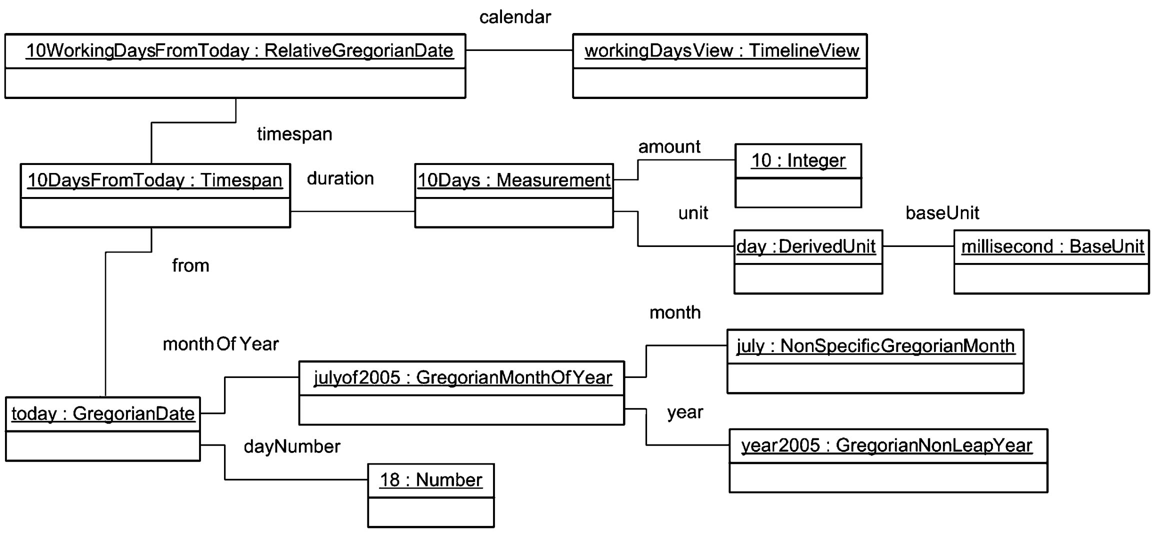
10WorkingDaysFromToday : RelativeGregorianDate
calendar
workingDaysView : TimelineView
timespan
10DaysFromToday : Timespan
duration
10Days : Measurement
amount
10 : Integer
unit
day :DerivedUnit
baseUnit
millisecond : BaseUnit
from
month
july : NonSpecificGregorianMonth
month Of Year
julyof2005 : GregorianMonthOfYear
year
year2005 : GregorianNonLeapYear
today : GregorianDate
dayNumber
18 : Number

Fig. 11. A RelativeGregorianDate object diagram.

The class **TimeOfDay** is implemented with an instance variable that represents the time passed since hour 0, that is a time measure. That time measure can be of any resolution (hour, minute, second, millisecond, nanosecond, etc.). If a better resolution that nanosecond is needed, a new time unit can be created with the new resolution to specify a more accurate time of day.

The **GregorianDay**, **GregorianDayOfMonth** and **GregorianMonth** classes are also subclasses of **PointInTime**, but their time line is more a circle than a line. For example, the message *#next* returns January 1st when it is sent to

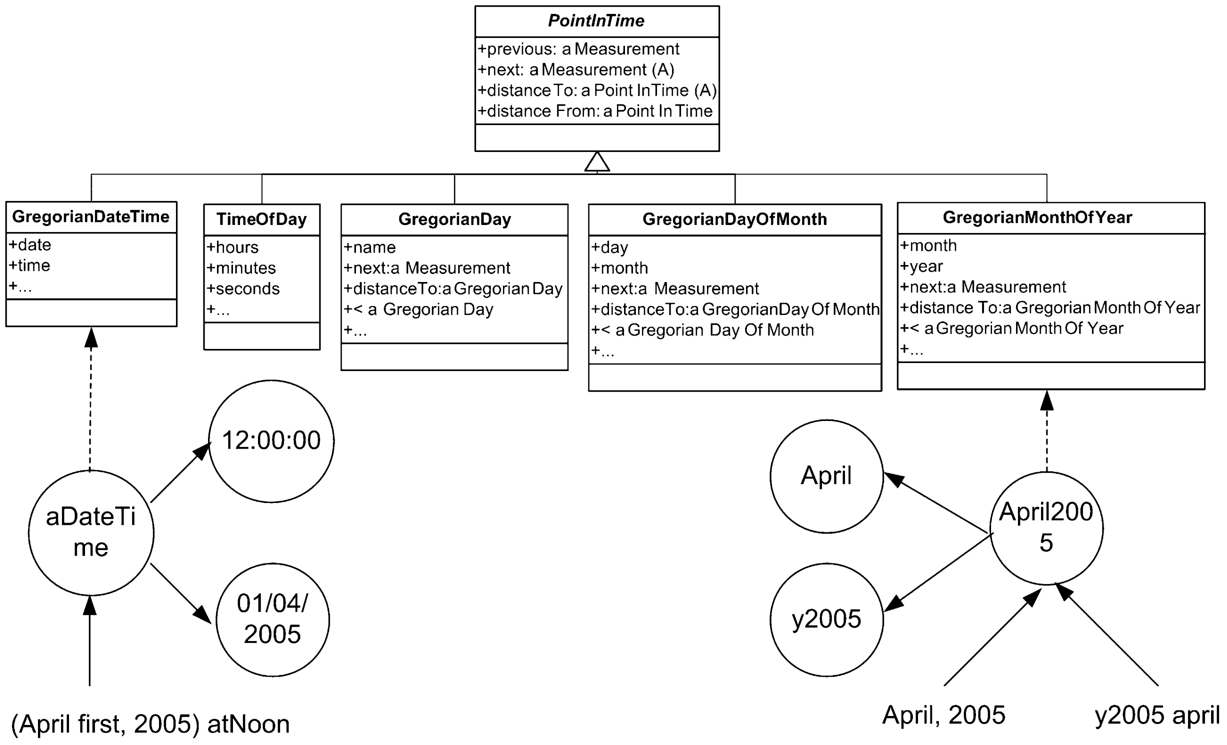

Fig. 12. Other points in time class hierarchy diagram.

December 31st, and the message #*previous* returns December 31st when it is sent to January 1st. Fig. 12 shows the class diagram for these time entities.

5.5. *Time line segments, intervals and time line filters*

The model provides new abstractions which behave like **Collection**. They are **SetDefinedByRules** and **MeasurementInterval**. The former allows the creation of sets where its elements are not added one by one. Elements belong to this set if there is a **SetRule** that returns *true* when the message #*includes*: is sent to it.

MeasurementInterval is provided by the measure model. It was necessary to create such an abstraction because the Smalltalk class **Interval** cannot be used with objects that are not **Number**. **MeasurementInterval** works with any class that defines a total order on its instances, like **Measure**, **GregorianYear**, **GregorianDate**, etc. Fig. 13 shows the class diagram for these classes.

Different subclasses of **SetRule** are provided such as **SpecificObjectSetRule** (used to define a specific object as part of the set), **TransformationSetRule** (used to decorate other SetRule with a transformation block) and **IntervalConstrainedSetRule** (used to filter other **SetRule** to the elements that are part of the interval) among others.

Time line filters are reified by three classes: **TimelineFilterBehavior,** an abstract class and superclass of **TimelineFilter** and **NegatedTimelineFilter**. A **TimelineFilter** is defined with a **SetDefinedByRules** and the message #*negated* returns an instance of **NegatedTimelineFilter**. A **NegatedTimelineFilter** has a **TimelineFilter** as source. When instances of this class receive the message #*includes*:, it forwards the message to its source and sends the message #*not* to the returned object (a **Boolean**).

Timespan is the class used to represent segments of the time line. It can be used with any **PointInTime** as the starting point (*from*). The *duration* can be any measure of time. Fig. 14 shows the class diagram of these abstractions.

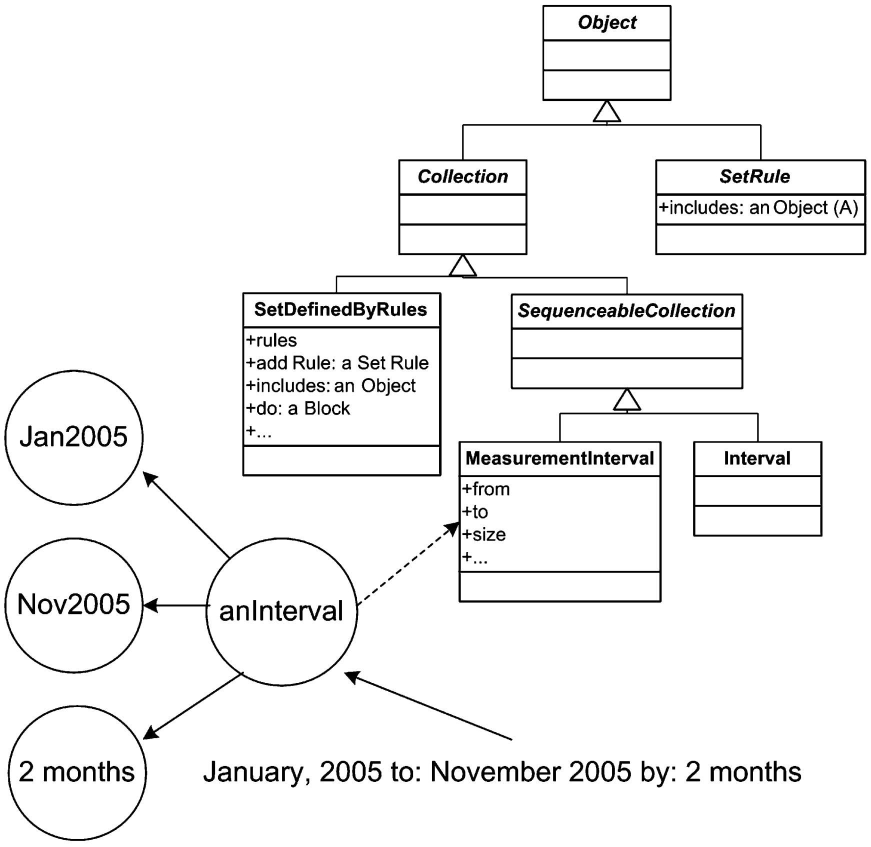

Fig. 13. Extension to the collection class hierarchy.

6. Related work

6.1. Comparison with Smalltalk-80 and squeak

Table 2 provides a brief comparison of time abstractions in Smalltalk-80, Squeak and our model. Note that the presented model reifies eleven time entities more than Smalltalk-80 and eight time entities more than Squeak. We present now some concrete examples that show the advantage of having those additional objects, thus proving, oncemore, the importance of reifying as many problem domain concepts (i.e., “model the real world”).

6.1.1. Selecting all mondays of the current year

Code Sample 21 shows how to solve this problem with Smalltalk-80, Squeak and this model.

Using the Smalltalk-80 model a collection with the correct number of days of the year 2005 is created first. Note that the number 2005 is used to refer to the year 2005 since no special object exists for it (ie.: lack of reification). This collection includes the numbers 1, 2, 3, . . . , 365 because year 2005 is not leap. A collection containing dates of year 2005 is created using the former collection. Note that the message *#newDay*: *year*: expects the number of days since January 1st plus one to create the right date, information that most of the people does not know (Does anybody know how many days are between January 1st and July 2nd?). Finally, all Mondays of year 2005 are selected comparing the date’s day name with the symbol *#Monday*.

With Squeak it is easier to obtain all year’s dates but its model still lacks an object to represent a day, therefore, a **Symbol** is needed to compare the day name which is error prone. If Monday is not correctly typed (i.e., *#mon-*

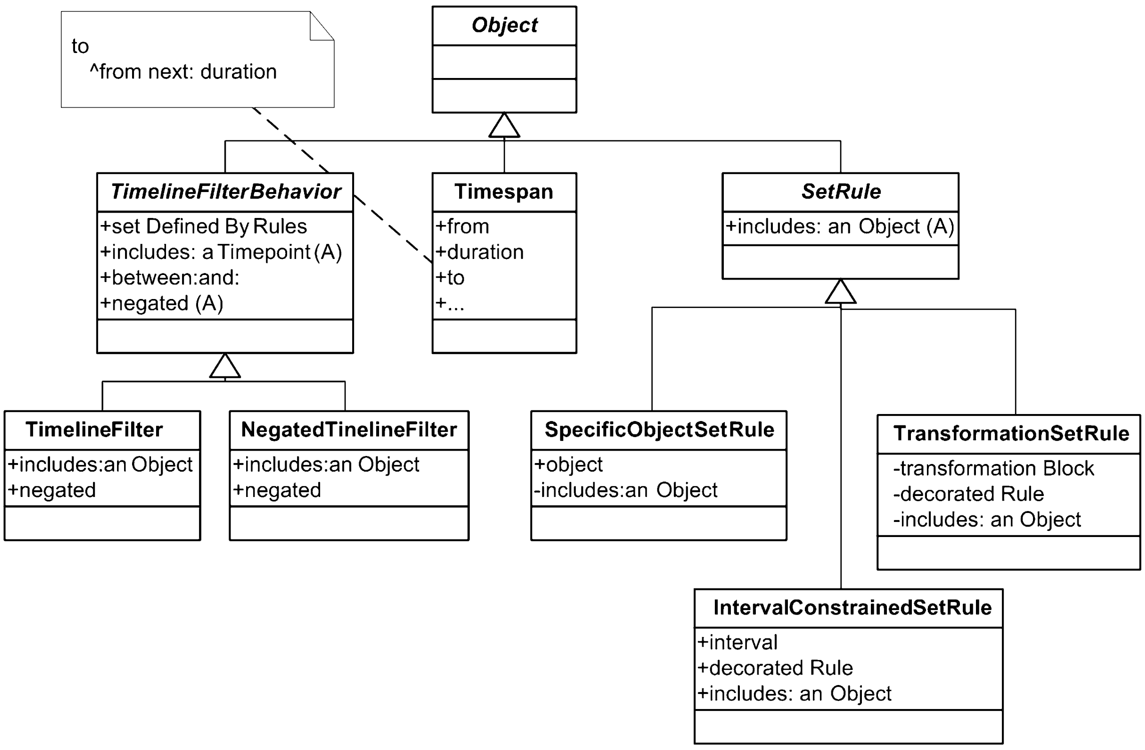

Fig. 14. Time filtering and time span class hierarchy diagram.

day instead of *#Monday*) the programmer will not get any indication of error and the program will not behave as expected.

With our model, getting the dates of a year is similar to Squeak's model, but because days are reified the message *#isMonday* is sent to the date. An error will be signalled if the message is not correctly typed or if the date protocol changes.

```
"Smalltalk-80 Solution"
yearDayCount := 1 to: (Date daysInYear: 2005).
currentYearDates := yearDayCount collect: [:aDayCount|Date newDay: aDayCount year: 2005].
currentYearDates select: [ :aDate | aDate dayName = #Monday].

"Squeak Solution"
Year current dates select: [ :aDate | aDate dayOfWeekName = #Monday].

"Our Solution"
GregorianYear current dates select: [ :aDate | aDate isMonday ].
```

Code Sample 21. Selecting all Mondays of the current year.

6.1.2. Getting the last dates of every month of a year

The solution of getting the last dates of every month of year is shown in the Code Sample 22.

We can observe with the Smalltalk-80 model, the same issues as in the previous example because all the dates of the year have to be created and because a month of a year is not reified the message *#daysLeftInMonth* is sent to a date.

With our model solution, because months of a year are reified, a collect on each month of a year is performed sending the *#lastDate* message to each of them. This solution has a better performance than the Smalltalk-80 one because the collect is done over twelve elements (the twelve months of a year) while the Smalltalk-80 does a select over 365 dates.

Table 2
Comparing smalltalk-80, Squek and the presented time model

	Smalltalk-80 and ANSI Smalltalk	Squeak's chronology package	Presented model
Year (i.e., 2005)	Modelled as a **Number**	Reified with class **Year**	Reified with class **GregorianYear**
Month of a Year (i.e., January 2005)	Not modelled	Reified with class **Month**	Reified with class **Gregorian-MonthOfYear**
Date (i.e., 01/01/2005)	Reified with class **Date**	Reified with class **Date**	Reified with class **Gregorian-Date**
Date and Time (i.e., 01/01/2005 10:00:00 AM)	Reified with class **DateAnd-Time**	Reified with class **DateAnd-Time**	Reified with class **Gregorian-DateTime**
Month (i.e., January)	Modelled as **Symbol**	Modelled as a **Symbol**	Reified with class **Gregorian-Month**
Day of Month (i.e., January First)	Not modelled	Not modelled	Reified with class **Gregorian-DayOfMonth**
Week (i.e., First week of 2005 or Second week of January 2005)	Not modelled	Not modelled	Not modelled
Day (i.e., Monday)	Modelled as **Number** and **Symbol**	Modelled as **Number** and **Symbol**	Reified with class **Gregorian-Day**
Time (i.e., at Noon, 10:00:00 AM)	Reified with class **Time**	Reified with class **Time**	Reified with class **TimeOf-Day**
Time distance (i.e., 1 year, 3 months, 10 days, etc.)	Reified with class **Duration**. Expressed only in terms of seconds	Reified with class **Duration**. A duration of 1 month is converted to 31 days	Reified as **Measure** with **Units** such as: year, month, week, day, hour, minute, second, millisecond, decade, century, millennium or any other time unit.
Time line segment (i.e., From 01/01/2005 with a length of 10 days)	Not modelled	Reified with class **Timespan**, with a start and a duration	Reified with class **Timespan**, with a start and a distance expressed as measure
Time line interval with different granularity (i.e., From 01/01/2005 to 01/02/2005, or from January 2005 to July 2005 every 2 months)	Not modelled	Not modelled	Reified with class **Measure-mentInterval** with a measure as step. Also know as time point occurrences
Relative Dates (i.e., 10 working days from 01/01/2005)	Not modelled	Not modelled	Reified with class **RelativeG-regorianDate**
Time line filter	Not modelled	Not modelled	Reified with class **Timeline-Filter**
The end of time	Not modelled	Not modelled	Reified with the object *theEndOfTime*
The beginning of time	Not modelled	Not modelled	Reified with the object *theBe-ginningOfTime*

The Squeak solution is similar to our model.

```
"Smalltalk-80 Solution"
yearDayCount := 1 to: (Date daysInYear: 2005).
currentYearDates := yearDayCount collect: [ :aDayCount|Date newDay: aDayCount year: 2005 ].
currentYearDates select: [: aDate | aDate daysLeftInMonth = 0].

"Our Solution"
GregorianYear current months collect: [ :aMonthOfYear | aMonthOfYear lastDate ].
```

Code Sample 22. Getting the last dates of every month of a year.

6.1.3. Obtaining the number of months between two months

Obtaining the number of months between two month is shown in Code Sample 23.

With the Smalltalk-80's model, since it does not deal with months of year, a mathematical expression has to be programmed to solve the problem every time we need to do so. In the code, we show there is no verification about the month number or year number, thus they could be invalid. This is a very common mistake that leads to invalid behaviour. This piece of code should be encapsulated to avoid mistakes and code duplication. Note also that the result of that expression is a number.

The Squeak model allows the programmer to deal with months (an abstraction we call **GregorianMonthOfYear**) but a **Timespan** has to be created to obtain all the months, and then the size of that segment is used to get the final result. Note that it also returns a number.

Because our model reifies the month of year concept, the *#distanceTo*: message is sent to the first one with the second one as a parameter. Note that the returned object is not the number 66 but a measure of time; in this case, measured in months: the object *66 months*.

```
"Smalltalk-80 Solution"
fromMonthNumber := 6.
fromYearNumber := 2005.
toMonthNumber := 12.
toYearNumber := 2010.
numberOfMonths := 12 − fromMonthNumber + (toYearNumber − 1 − fromYearNumber ∗ 12) +
toMonthNumber
"It returns the number 66"
"Squeak solution"
((Month month: 6 year: 2005) to: (Month month: 12 year: 2010)) months size
"It returns the number 66"
"Our solution"
(June of: 2005) distanceTo: (December of: 2010)
"It returns 66 months, not just 66"
```

Code Sample 23. Obtaining the number of months between two months.

6.2. Comparison with chronology squeak's package

The main difference between our model and Squeak's one is how time entities are understood. In our model, time entities are points in the time line and measures are used to represent time distances. In Squeak, time entities are segments in the time line modelled with the class **Timespan**. For example, **Month** is a subclass of **Timespan**, so it behaves like a time segment. Therefore, the object created with the expression "*Month month: 13 year: 2010*" (note that a month number thirteen is invalid in the Gregorian Calendar) is the same as "*Month month: 1 year: 2011*", because they are the same segment.

Because **Timespan** is the superclass of all time entities in Squeak, it has confusing protocol such as *#lastDate* or *#firstDate* and strange behaviour when comparing time entities. Messages such as *#lastDate* and *#firstDate* make sense when they are sent to a year or a month of a year, but they loose meaning when the receiver is a date or a date time.

Squeak allows comparing points of different resolution because time entities are modelled as segments, as it is shown in Code Sample 24.

```
"Returns true if today is not the first day of the current year"
Year current < Date today
"Returns true if today is not the first day of the current moth"
Month current < Date today
```

Code Sample 24. Comparison in Squeak.

We believe this is confusing and inconsistent with the analogy of time entities with segments. It is confusing because it does not make sense to ask "*Is the current year before today?*" How can a year be compared with a date? Only if a year is seen as a segment from the beginning of time to the first day of that year this question can be answered. But that is not the “common” meaning of year. A year is not the first day of that year.

Likewise, Squeak does not model a year as a segment starting at year 1 with a duration of the passed years; it models a year as a segment that starts at hour 00:00:00 of January 1st of that year, with a duration of 365 or 366 days. Therefore, we thought the # < message meant “does the segment receiving the # < message **include** the one given as parameter?”, but that is not the behaviour of # <. It does not mean “does the segment receiving the # < message **intersects** the one passed as parameter?” either. We could not find a consistent meaning for the # < message when sent to these objects.

Code Sample 25 shows the peculiar behaviour of the # < message. The behaviour when comparing a year with the first month or date of that year is the most puzzling one. They are not less, nor greater or equal between them. Strange behaviour is observed also when comparing instances of **Timespan** with its subclasses. To avoid this type of confusion our model does not allow points of different resolution to be compared as we showed and demonstrated before.

```
"All these comparisons return false"
(Year year: 2005) < (Month month: 1 year: 2005).
(Year year: 2005) > (Month month: 1 year: 2005).
(Year year: 2005) = (Month month: 1 year: 2005).
"This collaboration shows 'inclusion' behaviour"
(Year year: 2005) < ((Month month: 2 year: 2005) to: (Month month: 3 year: 2005)).
"This collaboration shows 'inclusion' behaviour"
(Year year: 2005) < ((Month month: 2 year: 2005) to: (Month month: 3 year: 2006)).
((Month month: 12 year: 2004) to: (Month month: 2 year: 2005)) < (Year year: 2005).
```

Code Sample 25. Puzzling behaviour of # < message in Squeak.

6.3. *Comparison with other research work*

Barbic and Pernici [3], in their work related to office information systems, propose a similar model to ours. The concept of time they present is based on a discrete temporal axis (our notion of time line) where time points can be mapped to integers, but it differs from our work because they provide a quantum of minute to every point, while in our work time entities are part of different time lines, each one with its own quantum. They also mention that “time is infinite in the past and in the future”, but it is not clear how they represent that characteristic in the final design, while our model provides two objects to reify those entities, *TheEndOfTime* and *TheBeginningOfTime*. Their model supports absolute and relative time entities, but they do not provide abstractions to filter the time line, so relative time entities fall in what we model as time spans. They propose to return “UNKNOWN” when comparing time points of different granularity which differ from our solution where such comparisons are not allowed, but they do not allow having specifications that cannot be converted to a same level of granularity. Week days, days of month and months are not modelled as first class entities.

Goralwalla et al. [1], in their work related to database management systems, propose a model that introduces the idea of temporal granularity, which is “a special kind of unanchored temporal primitive that can be used as a unit of time”. Such entities are related to the time units we propose in our model. The anchored time entities are what we represent as point in the time line, and unanchored entities are what we model as time measures. Due to the Gregorian Calendar irregularity, they propose to use indeterminate spans to represent conversions between measures of not related units. For example, *1 month* would be converted to *[28 days − 31 days]* therefore, comparing *1 month* with *30 days* would return UNKNOWN, the same solution adopted by Barbic et al. [3].

To represent anchored times, Goralwalla uses the concept of time granule defined by Bettini et al. [24], where an anchored entity is “an interval on the global timeline” which differs from our work where anchored entities are modelled as points in time. Because they use intervals they have to differentiate three types of interval, *Beginning Instant*, *Determinate Interval* and *Indeterminate Instant*. Examples of *Beginning Instant* are 1995_{beg}, January 1995_{beg} and January 1st of the year 1995_{beg}, that return true when compared for equality. Our approach is simpler because there is no need to implement different types of interval and we treat years, months of years and dates as completely different time entities, therefore, no confusion about the year 1995 and the month of year January of 1995 is allowed.

When moving from a point a measure of time with higher resolution that the quantum of the point, for example 3 days from January 1995, they assume they are moving from the first point of the same granularity contained in the former point, which means 3 days from January 1st of 1995 for the presented example. We see this as an arbitrary solution, therefore, we do not allow this type of expressions.

The ODMG time model [9] is similar to the Smalltalk-80 one; therefore, it lacks important abstractions and has the same problems mentioned in this paper as Smalltalk-80, such as representing years and months with numbers, modelling time distances with numbers and so on. Bertino et al. [7] proposes an extension to the ODMG model providing temporal information to objects and a new abstraction, time interval. Although this model helps to keep historic information about objects, abstractions to represent time spans, relative time entities, time granularity among others are left as future work. Huang et al. [8] also extends the ODMG model but adding a new dimension to the temporal one: the spatial dimension. Although this new dimension is being taken into account to keep historical information about objects, the time model has the same limitations as Bertino's et al. [7].

Goralwalla et al. [27] presents in a newer paper an object-oriented framework that provides a unified infrastructure to support temporal entities. The framework divides the time domain in four dimensions: Temporal Structure, Temporal Representation, Temporal Order and Temporal History. The Temporal Structure classifies temporal entities as anchored and unanchored. Anchored time entities are classified as instants and intervals. Both, anchored and unanchored, can be discrete or continuous and determinate or indeterminate. Our model supports this classification but some of the entities we provide do not behave as they proposed in [1] like it is previously explained. The Temporal Representation dimension is automatically provided by the classes composing our model. The Temporal Order is also provided in our model by means of protocol that each time objects respond to which, at the same time, are polymorphic no matter the type of the object. The Temporal History can be achieved using the collection objects and the time objects provided in our model.

The Java time model is completely different from our's. The Java Language [10] provides a single abstraction named **Calendar** to handle all types of time entities. **Calendar** is an abstract class that has concrete subclasses such as **GregorianCalendar**. This class is a combination of fields that are set with the message *set (int field, int value)* and get with the message *get(int field)*, being *field* a number that represents the time entity to be changed. For example: *set (Calendar.MONTH, Calendar.JANUARY)* changes the month of that calendar instance to be January.

An instance of **Calendar** with just the field *Calendar.YEAR* represents a year, an instance of **Calendar** with the fields *Calendar.YEAR* and *Calendar.MONTH* represents a month of a year and so on. Because **Calendar** represents all types of time entities, no specific protocol is provided to objects that represent specific time entities such as years, months, days, etc. For example, there is no message such as *#dates* to obtain all dates of a specific year. On the contrary, it provides generic protocol like *#isLeap* that can be answered by any instance of **GregorianCalendar**, such as dates. This ambiguity makes the model confusing, difficult to learn and use. For instance, the distance between two "calendars" is represented by the number of milliseconds that separate those "calendars". Therefore, if a year is compared with a date, the number of milliseconds since January 1st of that year to the hour 00:00:00 of the compared date is returned.

We believe this model suffers from the same design issues as any generic model. Real-world concepts should have a one-to-one mapping with the classes provided by object models. The Java model does not follow this rule, it joins the concept of year, month of year, date and date time in one concept they call **Calendar**; therefore, it provides a one-to-many mapping between real world concepts and model concepts, creating a different language that the one used in the problem domain. Note that when comparing time entities the programmer has to use the word "calendars" which is completely unreal; people do not compare calendars, people compare years, months, dates, etc.

This model also lacks abstraction to represent time measures, time intervals, time spans, days of months, relative dates and time line filters which are important entities of the time domain. This lack of abstractions forces the programmer to create their own implementation of such entities.

The .NET model [11] is similar to the Java one. It provides an abstract class named **Calendar,** which is subclassed by **GregorianCalendar**. In contrast to the Java model, the messages such as *AddDays*, *AddHours* sent to a **Calendar** to move through the time line do not return instances of **Calendar** but instances of **DateTime**. A **DateTime** is a "Structure" that measures the number of 100-nanoseconds since a particular origin defined by the calendar they belong to, for example January 1st of year 1 for the Gregorian Calendar. An abstraction called **TimeSpan** is provided to represent time measures expressed in 100 of nanoseconds. No different time units are provided. The .NET model has the same modelling problems as the Java one. It lacks important abstractions and it has the same design flaws.

7. Conclusion

This paper presents an object model that focus on the representation of the Gregorian Calendar time entities. It is based on a simple metaphor where time entities are represented by points in the time line. Those points have different resolutions and they include points of higher resolution.

The model provides a total order between time points which allows programmers to determine which point comes before or after another, go from one point to another and obtain the distance between two points.

Because time entities are analogous to points within a line, the model permits the representation of segments of the time line and it provides abstractions to create intervals between points.

A distinguishing feature of this model is that it uses a generic Measure Model to reify Time Measures. This modelling decision allows programmers to share the concept of measures of time with any other type of measure and it permits to operate arithmetically with them.

Time line filters created to include or exclude time line points is another important feature. Relative points in time can be created based on these filters. The model also provides abstractions for time entities such as a day, a day of a month and months.

The main benefit obtained with this model is that complex observations of the time domain can be easily programmed. Although this characteristic is difficult to be formally proved, it can be inferred because of the provided abstractions and protocol. This model is being used as “the” time model at Mercap Inc. in all its new applications. It proved us to be very powerful and easy to use.

7.1. Lessons learned

7.1.1. Create only valid objects

Objects should only exist if they are valid. For example, a Gregorian year can only exist if its number is not zero. This rule, combined with the rule of immutable objects, gives programmers the security that the objects they work with are always valid.

This rule also implies that invalid or incomplete objects should be represented with specific abstractions. Builders [25] are an example of this type of objects. When a builder is created it is “incomplete” because it cannot build the desired object until all the information of that object is provided. The builder will be modified until it reaches a state where it is able to create the specified object.

7.1.2. Immutable objects

The implementation of time entities as immutable objects simplified the model’s design and implementation. Not only they provide the benefits mentioned Baümer et al. in [21], but they also avoid non-contemplated consistency problems that could appear during an object’s life cycle.

Immutable objects that are valid from the time they are created ensure the programmer that she’s not dealing with invalid ones, because an object is not instantiated if its preconditions are not met.

7.1.3. Development technique

We cannot conclude this paper without mentioning the advantages we obtained due to the use of the “Test Driven Development” technique, presented by Beck in [18] and [26]. Each observation we made of the time domain was programmed as a test that we took as the starting point to implement and improve the model.

It is also necessary to highlight the advantages that a dynamically typed and late binding programming language offers when using this technique. It is because of the dynamically typed characteristic of Smalltalk that we could make our model evolve smoothly. The late binding characteristic allowed us to “program on demand” completely within the debugger, defining classes, methods and instance variables as required by the tests, a characteristic still very restricted in languages such as Java or C#.

7.2. Future work

We need to research the addition of time zone entities in our model. The time zone adds some complexity because we would like date times such as January 1st of 2005 at 10:00:00 in Buenos Aires, Argentina to be equal to January 1st of 2005 at 11:00:00 in Montevideo, Uruguay.

The **Timespan** protocol is limited at this time. We need to expand it with protocol related to line segments.

New abstractions need to be created like **Hour**, **Minute**, etc. We have not created them yet because measures are used to represent these entities. One advantage of having a class to represent hours is that an hour less than 0 or greater than 23 could not be created.

We have not reified time lines. We think that modelling time lines would simplify the implementation of moving along them or along lines of different resolution.

At this moment the model implements relative dates as the only relative points, but there is no reason to have such a limitation. We will expand the model to support any point in time to be relative.

Mercap Inc, is studing the open source licences to open this model to the Smalltalk community.

Acknowledgements

We would like to thank Mercap Inc.'s Software Development Team, for their comments and use of this Gregorian Calendar model. We want to thank also our shepherds: Alan Knigth and Stéphane Ducasse. Alan gave us a lot of feedback and Stéphane has supported this paper as much as he supports Smalltalk and he has dedicated more time on it than we have expected. We are really grateful to him.

References

[1] Goralwalla I, Leontief Y, Özsu M, Szafron D. Temporal granularity: completing the puzzle. Boston: Kluwer Academic Publishers.
[2] Allen F. Maintaining knowledge about temporal intervals. Communications of the ACM 1983;26(11).
[3] Barbic F, Pernici B.Time modeling in office information systems. ACM SIGMOD international conference on management of data, Proceedings; 1985.
[4] Wang XS, Jajodia S, Subrahmanian V. Temporal modules: an approach toward temporal databases. ACM SIGMOD international. Conference on management of data, Proceedings; 1993.
[5] Montanari A, Maim E, Ciapessoni E, Ratto E. Dealing with time granularity in event calculus. International conference on fifth generation computer systems, Proceedings, Tokyo; 1992.
[6] Corsetti E, Montanari A, Ratto E. Dealing with different time granularities in formal specifications of real-time systems. The Journal of Real-Time Systems; 1991.
[7] Bertino E, Ferrari E, Guerrini F, Merlo I. Extending the ODMG object model with time, ECOOP'98. Berlin, Heidelberg: Springer; 1998.
[8] Huang B, Claramunt C. STOQL: An ODMG-based spatio-temporal object model and query language. Symposium on geospatial theory, preocessing and applications, Ottawa 2002.
[9] Cattel R. The object database standard: ODMG93. Los Altos, CA: Morgan-Kaufmann; 1996.
[10] ⟨http://www.javasoft.com⟩
[11] ⟨http://www.microsoft.com⟩
[12] Maiocchi R, Pernici B, Temporal data management systems: a comparative view. IEEE Transactions on Knowledge and Data Engineering 1991.
[13] Maiocchi R, Pernici B, Barbic F. Automatic deduction of temporal information. ACM Transactions on Database Systems 1992.
[14] Goldberg A, Robson D. Smalltalk-80: the language and its implementation. Reading, MA: Addison-Wesley; 1983.
[15] Pinkney B. Squeak chronology package, ⟨http://minnow.cc.gatech.edu/squeak/1871⟩
[16] ⟨http://www.squeak.org⟩
[17] Wilkinson H, Prieto M, Romeo L. Arithmetic with measurements on dynamically-typed object-oriented languages, Practitioner report, Companion booklet of the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA 2005.
[18] Beck K. Test driven development: by example. Reading, MA: Addison-Wesley; 2002.
[19] Reingold E, Dershowitz N. Calendrical calculations: the millennium edition. Reading, MA: Cambridge University Press; 2001.
[20] ANSI Smalltalk ⟨http://www.smalltalk.org/versions/ANSIStandardSmalltalk.html⟩
[21] Baümer D, Riehle D, Siberski W, Lilienthal C, Mergert D, Sylla K, Züllighoven H. Values in objects systems. UBILAB Technical Report, 1998-10-10, Zurich, Switzerland.
[22] Allen E, Chase D, Luchangco V, Maessen J, Steele G. Object-oriented units of measurement. Proceedings of the 19th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications, OOPSLA 2004.

[23] Kennedy, Andrew J. Programming languages and dimensions. PhD Thesis, University of Cambridge. Published as Technical Report No. 391, University of Cambridge Computer Laboratory, April 1996.

[24] Bettini C, Dyreson CE, Evans WS, Snodgrass RR, Wang XS. A glossary of time granularity concepts, in temporal databases—research and practice, 1998.

[25] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: elements of reusable object-oriented software. Reading, MA: Addison-Wesley; 1995.

[26] Beck K. Extreme programming explained: embrace change. Reading, MA: Addison-Wesley; 1999.

[27] Goralwalla I, Tamer Özsu M, Szafron D. A framework for temporal data models: exploiting object-oriented technology, Proceedings of TOOLS'97, IEEE, New York.