FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Implementing Complex Calendar Systems in Java

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Frieder Ulm
Matrikelnummer 0527031

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Silvia Miksch
Mitwirkung: Dipl.-Inf. Dr.techn. Tim Lammarsch

Wien, 12.10.2013

(Unterschrift Frieder Ulm) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Implementing Complex Calendar Systems in Java

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Frieder Ulm
Registration Number 0527031

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Silvia Miksch
Assistance: Dipl.-Inf. Dr.techn. Tim Lammarsch

Vienna, 12.10.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Frieder Ulm
Horneckgasse 4/4 1170 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Frieder Ulm)

Acknowledgements

First and foremost, I would like to thank my parents for always supporting me during my studies.

I would like to thank my advisor Dr. Tim Lammarsch for providing me guidance and feedback during the time that I worked on this thesis.

Special thanks goes to my friends and colleagues Matthias Steinböck, Hubert Hirsch, and Eugen Dahm, without whom I certainly would not be where I am today.

Abstract

A calendar is a system designed to organize time periods into coherent groupings in order to make time instants and intervals more easily manageable and understandable for human social, religious, commercial and administrative use.

Calendars are a ubiquitous aspect of modern life, and are especially important in computer science. Although common elements exist across all calendric systems, structural differences between these systems are abundant. Due to the complex nature of calendric systems, and the abundant number of calendric systems of historical nature and still in use around the globe, modeling and conjointly use of multiple systems is non-trivial.

This thesis provides an introduction to calendric systems, how they can be classified, their rough structural organization, and how they can be modeled in general terms. A number of calendric systems is presented, highlighting both their similarities and differences. Furthermore, implementations of calendric systems in Java are evaluated based on their merits and downsides. Amongst the implementations presented is the *TimeBench* framework, which is being improved within the scope of this work. The changes done result in structural improvements of the TimeBench calendar modeling module, add XML extensibility to the calendar modeling process, introduce code guidelines, and improve overall code quality.

Kurzfassung

Ein Kalender ist ein System dass der Organisation von Zeitperioden in kohärente Gruppierungen dient. Dies macht Zeitintervalle und Zeitpunkte besser verständlich und verwendbar für soziale, religiöse, kommerzielle und administrative Zwecke.

Kalender sind ein allgegenwärtiger Aspekt des modernen Lebens, und sind insbesondere im Bereich der Informatik wichtig. Obwohl gemeinsame Elemente in allen Kalendersystemen existieren gibt es zahlreiche strukturelle Differenzen. Aufgrund des komplexen Aufbaus von Kalendersystemen und der hohen Anzahl von sowohl historischen als auch noch verwendeten Systemen ist die Modellierung und gemeinsame Verwendung von mehreren Systemen alles andere als trivial.

In dieser Arbeit wird eine Einleitung zu Kalendersystemen bereitgestellt, in der elaboriert wird wie diese klassifiziert werden können, wie ihre groben Strukturen aussehen, und wie sie in generellen Bausteinen modelliert werden können. Weiters wird eine Anzahl von weitverbreiteten Kalendersystemen vorgestellt, wobei sowohl die Ähnlichkeiten als auch die Unterschiede hervorgehoben werden. Unter den vorgestellten Frameworks befindet sich das *TimeBench* Framework, welches im Rahmen dieser Arbeit erweitert wird. Diese Erweiterungen verbessern die Struktur des Kalendermodellierungsmoduls, fügen XML-Erweiterbarkeit hinzu, führen Code-Richtlinien ein, und verbessern die insgesamte Code-Qualität.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Motivation & Problem Definition

In the modern world of computer science, time has an inherently important role in everyday life. Traditionally, time stamps are used to indicate the temporal relations between events. However, the way time intervals and instants are represented has a non-negligible effect on the correctness of the data and the interpretation thereof.

Generally, the organizational structure of time revolves around calendar systems. Traditional calendar systems are based on periodically occurring astronomical phenomena such as lunar phases or the seasonal cycle as the earth revolves around the sun. Today, many different calendric systems exist and are in concurrent use around the world. In order to programmatically use various numbers of calendric systems interchangeably, a common approach to model calendric systems in software is needed.

Modeling calendar systems as well as performing time calculations and conversions on different calendar systems are a non-trivial task. Time-oriented data can be used to identify trends and reveal significant characteristics of a set of data [27]. For example, it is possible to make data discrepancies such as curve peaks easily recognizable through utilization of visualization techniques. In order to construct informative visualizations, it is necessary to deal with time-oriented data with a measured and methodical approach. Using calendric systems as abstractions for time to make the time domain more easily handleable and understandable is an obvious choice.

The *TimeBench* framework has been developed by the Information Engineering Group (IEG) of the Institute of Software Technology & Interactive Systems of the Vienna University of Technology [27]. The TimeBench framework provides a data model as well as a software library for visual analytics of time-oriented data. In this thesis, we will elaborate the data model of the framework.

A focal point of the implementation is the solution to the problem of extensibility of supported calendric systems. At this time, many programming libraries only provide a set of calendric system implementations out of the box; adding new calendric systems is a task that requires lengthy implementation of calendar specific logic. As a possible solution to this problem, we

will implement a component of TimeBench that allows a more streamlined process of adding new calendars through XML and Java files through a well-defined set of fields and functions. As a result, users will be provided with the ability to define new calendars under the granularity concept to use with the TimeBench framework.

1.2 Research Questions

In this thesis, we will elaborate the following research questions:

1. What is the current state of the art of calendric system frameworks?
2. How can the existing state-of-the-art solutions be improved conceptually?
3. What software design patterns can be used and invented to aid the implementation of the conceptual improvements of calendric systems?

1.3 Method

In order to answer the research questions stated above, the following approach will be taken:

- We will research the domain of time measurement and modeling. This extends to how time is measured today, and how time measuring has evolved to come to where it is now. Furthermore, we will find common aggregations of time, and introduce a scientific approach to model time.
- To illustrate the structural differences between calendric systems, we will research which calendric systems exist, and how they may be categorized, and what commonalities and differences exist.
- We will perform a research into which implementations of date-time modeling exist in Java. We will perform an analysis of these libraries and frameworks by summarization of documentation, as well as code inspection. A comparison between existing libraries will be provided.
- Finally, we will perform implementational improvements upon the existing TimeBench framework to improve its extensibility.

1.4 Structure

This thesis is structured as follows:

- In **Chapter 2: Calendar Mechanics**, we will give an overview of the history and mechanics of calendric systems, as well as an approach to model calendars.
- In **Chapter 3: Calendar Types & Systems**, we will introduce a selection of popular calendars, and highlight their similarities and differences.

- In **Chapter 4: Calendars in Java Development** we will compare a number of state-of-the-art implementations of calendric systems in information technology, and analyze their ease of use, implementational structure, and how easy it is to extend supported calendric systems, if at all possible.
- In **Chapter 5: TimeBench**, we will present the calendar module implementation of the TimeBench framework, provide documentation, and present design details of the TimeBench calendar module.

CHAPTER 2

Calendar Mechanics

As calendric systems are a tool to measure, compare and organize time intervals, we will briefly explain how time is measured, and how its measuring has changed over centuries.

2.1 How time is measured?

The first set of calendric systems were constructed by observation of the heavens, and approximation of the recurring periods of appearances of celestial objects. Over time, measurement of time has become more precise.

Apparent Solar Day

Initially, the length of a day was based purely on observation. A now so-called *apparent solar day* was defined as the interval between the sun reaching a *local meridian* twice. A local meridian in astronomy is a great circle which includes the celestial poles of the earth, and the zenith. In easy terms, an apparent solar day is the interval it takes for the sun, starting from the midday zenith, to pass through the zenith again.

Astronomers soon realized that the duration of the day differed from day to day, and from year to year. Due to the fact that earth's orbit around the sun is elliptical, and due to the axial tilt towards the sun of earth's rotational axis, the apparent solar days' length varies over the seasons of a year. Therefore, the notion of the *mean solar day* was introduced to compensate for this fluctuation.

In the early 20^{th} century, with increasing precision of measuring tools, scientists came to the conclusion that earth's rotational speed was not constant. On short time scales, irregular motions in the core of earth lead to inconsistencies in rotational speed. Also, in the long run, earth's spin is slowing down due to tidal friction. These revelations led to a change of definition of the second.

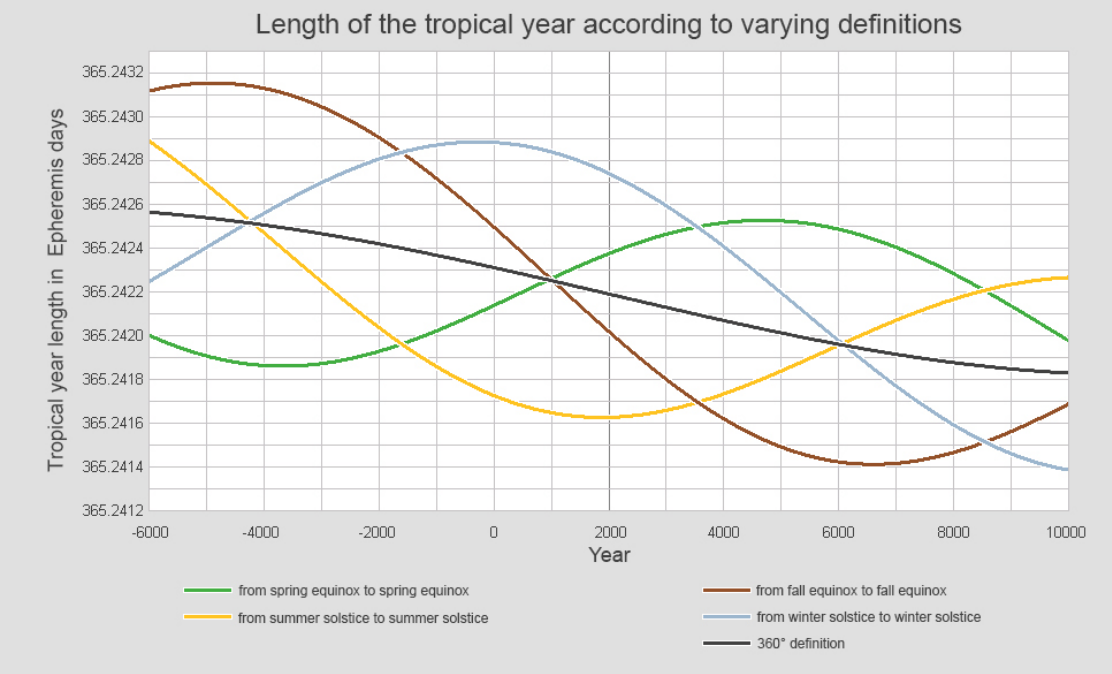


Figure 2.1: Change of tropical year length over time [28]

Mean Solar Day

Until 1954, the second was defined in relation to the mean solar day, which is the duration of a day averaged over a certain time frame. it was defined as follows:

$$\text{seconds} := 60\frac{sec}{min} * 60\frac{min}{hr} * 24\frac{hr}{day} = \frac{86,400sec}{day} \tag{2.1}$$

As such, a second was $\frac{1}{86,400}$ of a mean solar day. As illustrated in Figure 2.1, . Since the mean solar day is not constant, for reasons mentioned above, *ephemeris time* was introduced in 1954.

Epheremis Time

Ephemeris time is based on the tropical year (the time it takes the earth to complete a 360°-rotation around the sun) of January 0, 1900. The second became an SI standard unit and was defined as $\frac{1}{31,556,925.9747}$ of the tropical year for January 0, 1900. When atom clocks became available, more precise time measurements became possible. After three years of observation, Markowitz et. al determined that the ephemeris second corresponds to 9,192,631,770 ±20 cycles of transitions between two energy levels of the caesium-133 atom [17]. In a revision to this standard, in 1967/68, the General Conference on Weights and Measures replaced the definition of the SI second with the following:

> *The second is the duration of 9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium*

133 atom. This definition refers to a caesium atom at rest at a temperature of 0 K. [20]

Finally, we have arrived at the definition of a second how it is still in force today. Since the definition of a second has changed by a relatively large margin over the centuries, adjustments in calendric systems were needed to adjust for these changes. When calendars are modeled in information systems, taking these adjustments into account becomes highly necessary to perform accurate conversions between dates of different calendars.

Greenwich Mean Time

In treaties of 1883 and 1884, the international community agreed to establish time zones. These time zones would span to $7\frac{1}{2}^\circ$ on either side of a series of meridians spaced in 15° intervals. The prime meridian at 0° was chosen to pass through Greenwich, London. Thus, the time zone that spans to equal parts east and west of this meridian was called the Greenwich Mean Time (sometimes also called Zulu Time).

International Atomic Time

The principal international time standard is *International Atomic Time* (TAI, french: *Temps atomique international*), which was officially formalized in 1963. TAI is measured by over 200 caesium clocks all over the world, and is compared through GPS and two-way satellite time and frequency transfer. Each clock is adjusted for relativistic and environmental effects, and the aggregated data is averaged to obtain a stable standard time. [21]

Universal Coordinated Time

Universal Coordinated Time is the current de-facto international time standard that emerged out of a serious of reforms of the Greenwich Mean Time. The most important time standard it is based off of is UT1, which is computed by observing very distant star constellations, laser ranging of the moon, and GPS satellite orbits. Universal Coordinated was introduced in 1963 and was adjusted a few times to accommodate the addition of leap seconds. This time standard is defined to always be within 0.9 seconds of UT1, and is measured with the help of International Atomic Time. Since its inception, a total of 25 leap seconds were added to keep it in line.

2.2 Calendar Basics: Common Concepts

The basic task of each calendric system is to allow its users to label and put order to instants of time, and by extension, to allow each instant in time to be completely qualified and to be identified disjointly. This means that for each moment in time, there is a separate name, different from each other moment. Not only allows this distinguishing between two moments, but as these moments are lined up in a timeline of moments by the calendric system, one can determine which moment occurred before or after the other, and in some calendric systems, *how much* time lies between them. To ease human understanding, and to follow observed astronomical phænomena, these time periods are often aggregated into larger time periods.

The Day

All historical calendars are based on the daily cycle [26]: The fundamental unit for nearly every calendar, even today, is the day: the period it takes the earth to complete one revolution around its axis. While the Bahá'í calendar begins the day at sunset, the Hindu calendar starts and ends it at dawn; other calendric systems begin and end the day at midnight. The reasons for beginning a day at dusk or dawn are obvious: The moment when the sun sets or rises is easy to observe. However, even during one solar year, dusk and dawn fluctuate to a significant degree over the seasons, depending on the position of observation relative to the equator. Using these events as a measuring point quickly became impractical, and the midnight-to-midnight definition of a day quickly prevailed (though a noon-to-noon definition of a day offers many of the same advantages).

Some calendars begin the day at midnight, others begin at dusk or dawn. However, there is a fundamental difference of which astronomical phænomena were followed by the creators of the first calendric systems. Today, the definition of a day is no longer strictly based on the solar day, but on the definition of the SI second, discussed above [22]:

$$\text{day} := 86400\text{sec} \tag{2.2}$$

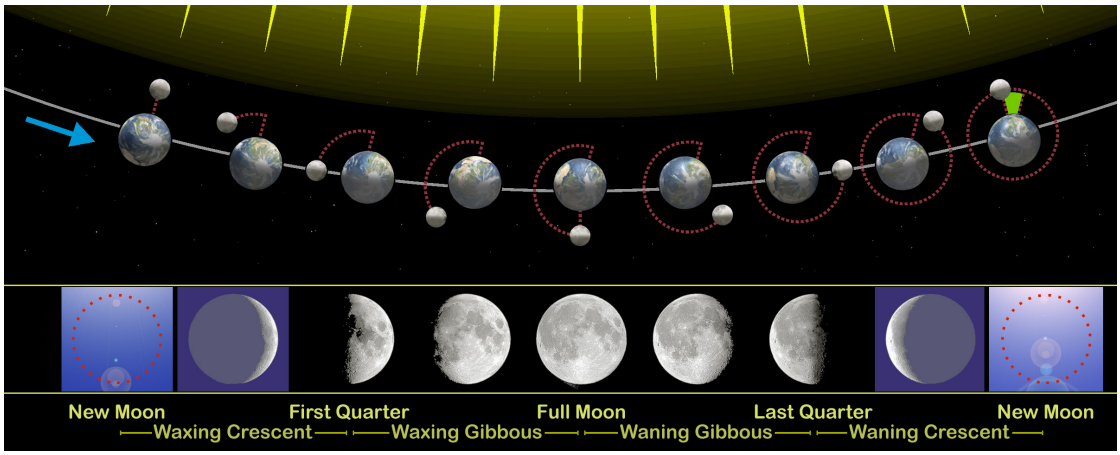


Figure 2.2: Lunar Phase [1]

Additionally, most traditional calendars feature the scheme of grouping days into longer units of time, the week, month, and year. These groupings give users of the calendar a better grasp of long time intervals and help pinpointing precise moments in time that are quite far apart.

The Week

In many cultures, a week has seven days. These days were often named after the seven "wandering stars" visible to the naked eye, the celestial objects that today are known as the Sun, the Moon, Mercury, Venus, Mars, Jupiter, and Saturn [11]. In other cultures, a variety of week mod-

els featuring different numbers of days per week were used, such as 4-day weeks in the Congo, 5-day weeks in Africa, Russia and Bali, or 5-day weeks in Japan.

The Month

Understanding of the moon's cycles was an early development in astronomy. Visibility of the moon from earth depends on its position relative to earth and the sun. In ancient times, astronomers observed the movement of the moon whenever it was visible and were able to determine the recurring interval of the moon's movements. The period of one rotation of the moon around the earth, illustrated in Figure 2.2, is called a lunar phase, lunation, or *synodic month*. However, due to the fact that the movement of the moon around around the earth is not uniform, much along the lines of the movement of the earth around the sun, the synodic month is not of perpetual equal length. While even in ancient times this was known and measurable, today we can define the mean synodic month to a much more detailed degree: The long-term average of the synodic month equals 29.530589 mean solar days [11].

In the fifth century BC, the Greek astronomer Meton of Athens observed a repeating cycle of constellations in the solar systems that became henceforth known as the *Metonic cycle*. His observation was that the duration of 235 synodic months would very closely match the duration of 19 solar years (give and take a few hours, depending on the definition of the year and month). The Metonic cycle became an important foundation for many traditional calendars, because it allows the formal synchronization of the lunar and solar cycles in one calendric system.

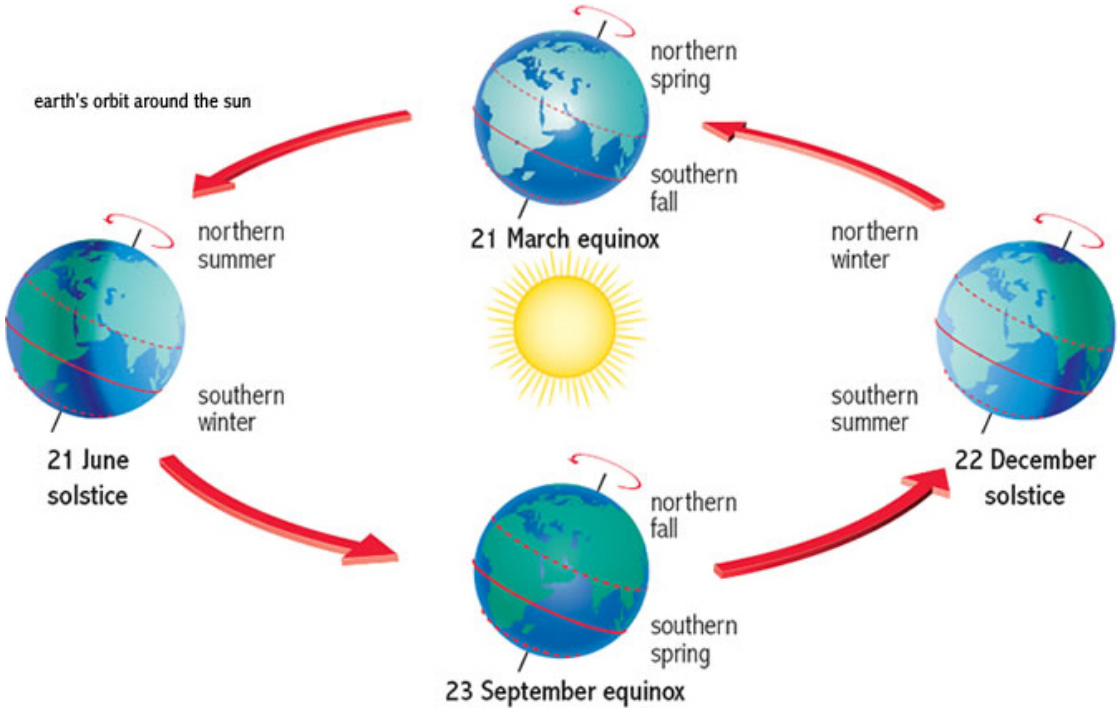


Figure 2.3: Solar Cycle [12]

The Year

Another commonly used calendar concept is the yearly cycle. However, its implementation is highly dependent on the type of calendric system. That said, the underlying idea of the year is to align a recurring cycle to the observable seasons: winter, spring, summer, fall. Recognition of obvious meteorological and biological signs of the seasons are ancient beyond memory or record. The seasons are a result of the earth's rotation on an elliptical path around the sun, as well as the earth's tilted axis relative to the plane of revolution around the sun. Due to this tilt, the northern hemisphere is exposed to solar radiation for a longer duration per revolution during summer as during winter, and the reverse for the southern hemisphere. This leads to the fluctuations of the daily cycle discussed above.

In Figure 2.3, a full revolution of the e around the sun is illustrated. There are four important events every year which were used in ancient times to measure the length of a year: the fall and spring *equinox*, and the summer and winter *solstice*.

The equinox, which is derived from the Latin words *aequus* (equal) and *nox* (night), is the time when the plane of earth's equator passes through the center of the sun. During this event, the axial tilt of the earth is inclined neither away nor towards the sun. More generally put, the day of the equinox is one of the two days of the year when the length of the day (or night) is the same at every point on the earth's surface. Furthermore, during this day, the periods of nighttime and daylight are roughly equal.

Similar to the word equinox, the word solstice is derived from Latin words. It is a conjunction of the words *sol* (sun) and *sistere* (to stand still). Each year, the summer and winter solstices are marked as the day of the year when the sun reverses its seasonal movement over the horizon: during the summer solstice, at midday, the sun comes to its highest point on the horizon, whereas during the winter solstice, it reaches the lowest point.

Recognizing and understanding the occurrences of solstices and equinoxes was monumental to the construction of calendric systems in ancient times. Before people learnt of the equinoxes and solstices, there was no clear way to start a year, and to determine its duration. After the modern definition, a year is the period it took the earth to revolve around the sun at January 0, 1900 - the tropical year of that date. This definition is closely related to that of epheremis time.

Finally, the definition of a year arises. Years, months, and weeks are a necessary aggregation of time intervals to ease human interaction and coordination of timed events. However, months and years are also based on seasonal and astronomical phænomena. Furthermore, none of these time intervals has, in the astronomical sense, a constant time duration, but is subject to change due do environmental factors. This is a problem that highlights a central problem in the design of each calendric system: The need for calendar units to remain stable and constant in terms of time intervals, while also having to (as much as possible) accurately fit seasonal and astronomical events.

To assuage this issue, many calendars and timekeeping systems feature leap seconds, months, and years.

Intercalation

There are three mechanisms by which time calculation is adjusted to keep time measurement in line with the sidereal year and/or the metonic cycle. These mechanisms are collectively referred to as *intercalation* [26].

1. Leap days: Adding an extra day to a year is one of the most commonly used mechanisms to adjust the year to seasonal occurrences. Usually, an extra day is added to the end of a certain month. For example, see the Gregorian calendar.

2. Leap weeks: Some calendar definitions specify a year to have a set number of weeks. If a certain day of the week is specified to begin the week, and the days cycle in a way to put more than the specified number of week starting days into a given year, an extra week is added to said year.

3. Leap (or embolismic) months: Often used to adjust a calendar for the metonic cycle or occurrences of lunations, such as in the Hebrew calendar.

2.3 Modeling calendars

When a model of time is created, several aspects have to be considered when the decision is made how the model should fit the real physical time dimension. Depending on the needs of the application, there are different solutions, some of which are better suited than others. Aigner et al. [2] gave a definition of important general design aspects of time modeling that we will summarize in the upcoming section. In the upcoming section we will summarize a set of important general design aspects of time modeling.

Design Aspects

Scale

In an **ordinal** time scale, only relative events are given. The temporal ordering of events is set by putting these events into relation with one another with relations such as “before”, “after”, “concurrently with”. An important aspect of modeling time scales is *temporal distance*, which is a measurement of *how much* time has elapsed between two events. Since points in time on an ordinal time scale are related to one another strictly by qualitative relationships, and no quantitative temporal information is available, is not possible to measure temporal distances on an ordinate time scale.

In **discrete** time domains, temporal distances can be considered. Every discrete time domain has a smallest possible measurement unit, which can be used to measure the time between two events to the accuracy of that measurement unit. Every measurable disjoint point in time can be assigned an integer, and while there can be infinitely many measurable points, the number of measurable points of time between two points is always finite.

Analogously to real numbers in mathematics, in a **continuous** time domain, between any two points in time, another point in time can be portrayed in the model.

Scope

When deciding on a model of the time domain, the second important design aspect is the choice of scope. The scope decides between the default interpretation of a time value. For example, the time value October 21, 2013 in a **point-based** time domain could refer to October 21, 2013 at 00:00:00, or noon, or whichever interpretation is chosen.

In an **interval-based** time domain, the same time value might relate to the interval [October 21, 2013, 00:00:00; October 21, 2013, 23:59:59].

Arrangement

A model of the time domain can contain both linear and cyclic arrangements. The natural perception of time is that points in time follow one another in a **linear** manner, lined up from the past over the present into the future.

However, many calendar systems incorporate **cyclic** organization of time intervals to reflect cyclic seasons or astronomical observations (further explained in 2.2).

Viewpoint

Time domains with an **ordered** perspective allow events to be modeled that happen one after another. More specifically, an ordered viewpoint may be partially or totally ordered. Where a total order allows only one event to occur at any one time, in a partially ordered domain, multiple occurrences may happen simultaneously.

A more complex viewpoint is the **branching** viewpoint, which allows modeling of multiple (partially) simultaneous timelines. This viewpoint is useful when planning future actions and consequences thereof, or when evaluating past cause and effect relations.

Finally, the **multiple** perspective time domain allows the same event to occur at different points in time. An example from computer science that demonstrates multiple perspectives would be in database engineering: In a distributed database, the time of which a certain change is propagated to different database mirrors can (and most often is) different from the time the change was originally committed, and may even arrive at different points at different database mirrors. This may lead to inconsistent database states and is an enormously important subject in database engineering.

Time Granularity

The concept of time *granularity* was formally introduced by Bettini et al. [6]. The concepts of time granularity may be used as a meta-model to model the structure of calendars and the relationships between different calendric systems.

Time Domain

A time domain is a collection of instants of time. Furthermore, it is possibly to determine whether each instant occurred before, after, or at the same time as any other instant:

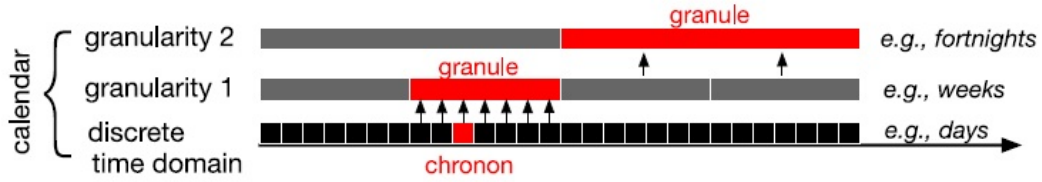


Figure 2.4: Example of a discrete time domain with multiple granularities [2]

> A time domain is a pair $(T, \leq)$ where T is a non-empty set of *time instants* and $\leq$ is a total order on T. [5]

Furthermore, a time domain may be bounded if it contains an instant that represents either the start of time, or the end of time, or both.

Chronon

In discrete time domains, the chronon denotes the smallest possible underlying unit of time that is modeled. In other words, the time domain is a set of consecutive, non-decomposable time intervals called chronons [16]. Chronons may be either be grouped into larger intervals, or into equally long intervals. In both cases, these groupings are called granules.

Granularity

Granularities are abstractions of time to make the concept of time more easy to grasp for human use. One can think of granularities as a system of time units that, mapped from one to another, together form a calendar.

> A granularity is a mapping G from the integers (the *index set*) to subsets of the time domain such that:
>
> 1. if $i < j$ and $G(i)$ and $G(j)$ are non-empty, then each element of $G(i)$ is less than all elements of $G(j)$, and
> 2. if $i < k < j$ and $G(i)$ and $G(j)$ are non-empty, then $G(k)$ is non-empty. [5]

In Figure 2.4, a sample calendar with two granularities is illustrated. It shows a granularity lattice with two mappings :

- **Chronon $\rightarrow$ Granularity 1**: A granule of granularity 1 consists of a time interval equal to seven chronons.
- **Granularity 1 $\rightarrow$ Granularity 2**: A granule of granularity 2 consists of a time interval equal to two granules of granularity 1.

For ease of reference, the granularities are related to real world examples: The chronon may have interval length of a *day*, and granularity 1 is equal to the granularity *week*, which encapsulates seven consecutive days. Granularity 2 is of interval length of two consecutive weeks called *fortnight*.

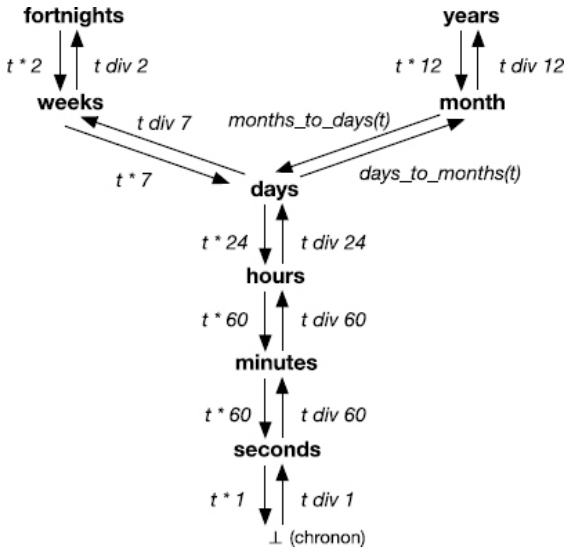


Figure 2.5: Annotated granularity lattice of the Gregorian Calendar (without leap seconds) [2]

Granule

Closely connected to granularities is the granule.

> Each non-empty subset $G(i)$ is called a *granule* of the granularity G. [5]

Granules may be single instants, such as an arbitrary date. Granules can also be a set of contiguous or non-contiguous instants, forming one or more time intervals. An example for a contiguous granule would be the week of October 20 to October 26, 2013. On the other hand, an example for a non-contiguous granule would be the academic weeks of October 2013, which contain the days Monday through Friday of every regular week.

Bottom Granularity

In information technology, calendar models usually have a smallest granularity, the *bottom granularity* [2] [5]:

> Given a granularity order relationship *g-rel* and a set of granularities having the same time domain, a granularity G in the set is a bottom granularity with respect to *g-rel*, if G *g-rel* H for each granularity H in the set. [5]

A bottom granularity is the smallest temporal entity in a granularity lattice. In Figure 2.5, the bottom granularity is the chronon (as it often is), which, in this case, is equivalent to the second.

Granularity Lattice

A granularity lattice is a system of mappings between granularities. As a whole, a granularity lattice defines a calendric system. Mappings between granularities may be *regular* or *irregular*.

> A set of granularities, having the same time domain, forms a granularity lattice with respect to a granularity order relationship *g-rel* if for each pair of granularities in the set there exists a least upper bound and a greatest lower bound with respect to the relationship *g-rel*. [5]

Regular mappings exist for mapping relationships between granularities that are always of the same quantity. An example for a regular mapping between the granularity *second* to the granularity *minute* is illustrated in Figure 2.5: A minute is always comprised of 60 seconds.

On the other hand, irregular mappings refer to mappings between granularities that are context-sensitive and need to be determined upon creation of granules. The only example of this, again illustrated in Figure 2.5, is the granularity mapping between *days* and *month*. As opposed to all other mappings appearing in this granularity lattice, the exact relation between the granularities is not set. In this case, *days_to_months()* and its inverse, *months_to_days()* denote functions that determine the exact mapping values.

Temporal Primitives & Determinacy

The Consensus Glossary of Temporal Database Concepts [16] defines a number of *temporal primitives* that are essential for performing temporal calculations. The following definitions will be used in this paper:

- *"An instant is a time point on an underlying time axis."* [16]. On discrete time scales, instants may be mapped to the natural numbers, meaning that for every instant, an immediate predecessor and successor instant exists. This analogy can be expanded to a contiguous time scale by mapping instants to real numbers. When the concept of the instant is used in conjunction with the time granularity concept, the instant is an element of a granule. Thus, a granule may represent more than one instant.

- *"A time interval is the time between two instants."* [16] In a model that utilizes time granularity, a time interval may be modeled by a series of contiguous granules set into context. For example, the interval between January 1, 2014 and February 1, 2014 is the series of day granules for January 1, January 2, and so on.

- *"A span is a directed duration of time."* [16] This means that a span may either be a positive or negative. For example, the span *two months, three days, 2 hours* is a positive span that may be used with temporal primitive operators to perform calendric calculations. It is important to note that these time spans are not anchored on the time scale.

There are a number of defined relations [3] [2] [30] that may be used to set these temporal primitives into relation to one another. The defined relations are listed below and further illustrated in Figure 2.6.

Utilizing spans, temporal calculations are made possible by defining a temporal calculus. The semantics shown in Table 2.1 and Figure 2.6 may be augmented. The following exemplary set of operations [30], defined akin to the standard mathematical operations addition, subtraction, multiplication, and division, form the basis for a sample temporal calculus, displayed in Table

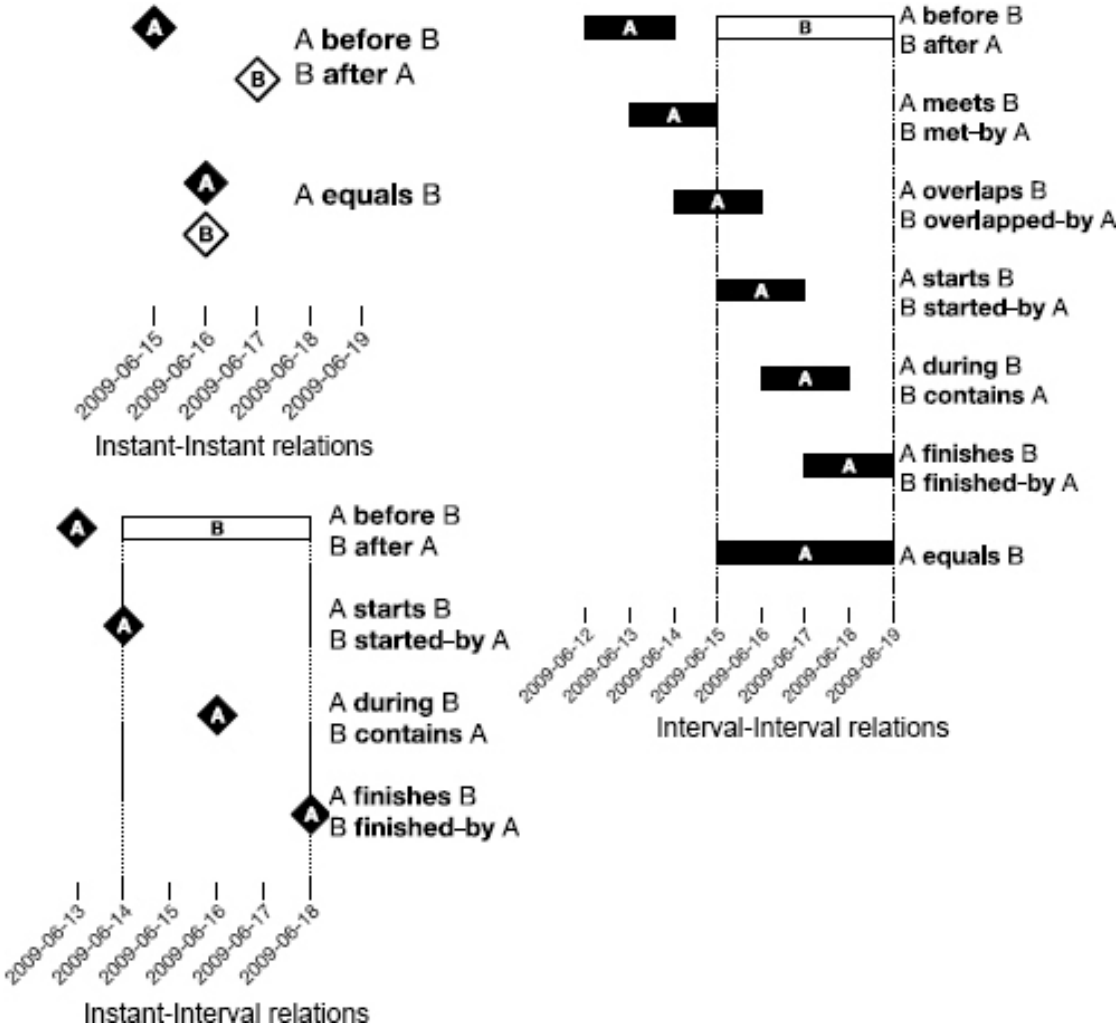


Figure 2.6: Temporal primitive relations illustrated [2]

2.2. When implementing these operators, special care has to be taken to document the route taken during implementation. Consider the following case: We add one month to the instant February 4th, 2012, and then we subtract a month from the resulting instant. Depending on the definition of *one month*, the result of the calculation may be a variety of different dates.

Indeterminacy is an inherent quality of temporal data. Most of the time, when working with granularities, incomplete qualification of data can lead to different interpretations of granule instances. Consider the statement: "*Activity A started on June 14, 2009 and ended on June 17, 2009* – this statement can be modeled by the beginning instant June 14, 2009 and the end instant June 17, 2009, both at the granularity of days. If we look at this interval from a granularity of hours, the interval might begin and end at any point in time between 0 a.m. and 12 p.m. of the specified day." [2]

Operand 1	Relation	Operand 2
Instant	Before	Instant
Instant	After	Instant
Instant	Equals	Instant
Interval	Before	Interval
Interval	After	Interval
Interval	Meets	Interval
Interval	Met by	Interval
Interval	Overlaps	Interval
Interval	Overlapped by	Interval
Interval	Starts	Interval
Interval	Started by	Interval
Interval	During	Interval
Interval	Contains	Interval
Interval	Finishes	Interval
Interval	Finished by	Interval
Interval	Equals	Interval
Instant	Before	Interval
Interval	After	Instant
Instant	Starts	Interval
Interval	Started by	Instant
Instant	During	Interval
Interval	contains	Instant
Instant	Finishes	Interval
Interval	Finished by	Instant

Table 2.1: Temporal Primitive Relations

Operand 1	Operator	Operand 2	Yield
	-	span	span
span	+	span	span
span	-	span	span
instant	+	span	instant
instant	-	span	instant
span	+	instant	instant
instant	-	instant	span
span	*	numeric	span
numeric	*	span	span
span	/	numeric	span
span	/	span	numeric
span	+	interval	interval
interval	+	span	interval
interval	-	span	interval

Table 2.2: Sample temporal calculus definition

CHAPTER 3

Calendar Types & Systems

In this chapter, a number of calendric systems will be presented in detail. The similarities and differences will be highlighted in order to make clear the challenges to an implementation that aims to create a meta-model for calendric systems.

There are some underlying characteristics that apply to all calendars:

- Based on calculcation or observation:
  - *Rule-based* or *arithmetic* calendars are based on a set of rules that mathematically define when a certain event will happen within the calendar. However, they often need to be adjusted with special rules to stick to recurring astronomical cycles.
  - *Astronomical* calendars are primarily based off of ongoing observations. Since many astronomical occurrences do not happen in regular time intervals, this calendar type makes it hard to pinpoint exact times for recurring events, such as Easter.
- The calendar may be complete or incomplete:
  - Complete calendars fully map every point in time, often in a cyclic manner, down to a certain degree of precision, or granularity
  - Incomplete calendars may contain gaps, or inconsistent application of calendar granularities: For example, work-day calendars leave gaps for the weekend days, and some ancient calendars lump the days of winter together in one big “month”, with no way of describing a certain day.
- Most calendars have an epoch - a reference date to which all points in time described with the calendar are relative to.
- Many calendars feature some kind of intercalation.

Additionally, it need be mentioned that depending on the application, calendars may either be used proleptically or non-proleptically. This can easiest be explained by example of the Gregorian and Julian calendars. The Julian calendar was used between the years 46 BC and 1582, while the Gregorian calendar has been in use since 1582 until today (varying by country). If a calendar is applied proleptically, its rules are simply applied backwards (or forwards), even though in reality it may never have been used that way. Alternately, nonproleptic use of calendars requires use of different calendar systems in certain eras to acquire historically accurate dates.

3.1 Lunar Calendars

Lunar calendars are primarily based on the lunar cycle. Often, this kind of calendar is used for religious purposes, but in ancient times, they were also used to help with hunting and fishing. It has been observed that animal movement and the tides (obviously) are influenced by the lunar phase. Hunters and gatherers that know this, or act according to a lunar calendar, will have a bigger chance to yield better catch.

Islamic calendar

The Islamic Calendar is used primarily for religious purposes, and is a astronomical, purely lunar calendar. It regulates feasts and fasts of the Islamic religion, as well as the time for pilgrimage to Mecca.

The epoch of the Islamic calendar refers to the year in which the Prophet Muhammad emigrated from Mecca to Medina. This emigration is known as *Hijra*, and numbering of years began relative to the year of Hijra.

The Islamic year is structured as follows:

Month of year	**Name**	**Number of days**
1	Muharram	30
2	Safar	29
3	Rabî' I	30
4	Rabî' II	29
5	Jumada I	30
6	Jumada II	29
7	Rajab	30
8	Sha'bān	29
9	Ramadân	30
10	Shawwâl	29
11	Dhu'l-qu'da	30
12	Dhu'l-hejji	29/30

As can be seen, each month has 29 or 30 days, with the first month having 29, and each successive month alternating. The only exception to this rule is when an embolismic month is added: During yeap lears, the 12th month has 30 days, instead of the usual 29. This means that normal years have a length of 354 days, while leap years have 354 days.

Embolismic months occur under a strict schedule: In a 30 year cycle, the following years are leap years: 2, 5, 7, 10, 13, 16, 18, 21, 26, 29

It should be noted that in a 30 year cycle, there are 360 months, or 10631 days. The mean length of a month of the Islamic calendar is:

$$\text{mean Islamic month length} := 10631 \text{ days}/360 \text{ months} = 29.53055\bar{5}\frac{\text{days}}{\text{month}} \tag{3.1}$$

This is very close to the mean synodic period of 29.53509 days.

Islamic weekdays begin at sunset. Islamic calendar divides months and years into weeks of seven days. The weekly days of rest depend on the country where the Islamic calendar is used. Sometimes the weekend only consists of al-Jumu'ah (Friday), the day of gathering. Other countries add either al'Khamīs (Thursday) or al-Sabt (Saturday) to the weekend. Finally, some countries have adopted the Western Saturday-Sunday weekend.

Day of week	**Arabic name**	**English equivalent**
1	(Yaum) al-Ahad	Sunday
2	(Yaum) al-Ithnayn	Monday
3	(Yaum) ath-Thalaathaa'	Tuesday
4	(Yaum) al-Arba'aa'	Wednesday
5	(Yaum) al-Khamīs	Thursday
6	(Yaum) al-Jumu'ah	Friday
7	(Yaum) as-Sabt	Saturday

Roman Calendar

The earliest instance of the Roman calendar is believed to initially be a solar calendar. It is described to have been invented by Romulus, the founder of Rome, in 753 BC. It is arranged into 10 months with either 30 or 31 days, the distribution of days illustrated in the following table.

The names of the months follow a fairly simply naming pattern. They are either numbered, or named after certain Roman deities.

The months Martius, Aprilis, Maius, Iunius are believed to be named after a the names of gods: The first month of the year, Martius, is named after the Roman god of war, Mars. The etymology of the second month, Aprilis, is uncertain: One possibility is that it is named after the Latin verb *aperire*, "to open", hinting at the beginning of spring, referring to the blossoming of flowers. The other possibility is that it is named after "Aphrilis", referring to the Greek goddess Aphrodite (or its Roman counterpart, Venus). Maius is named after Maia, the goddess of growth. Iunius is named after Juno, the goddess of women. The names of the months Quintilis, Sextilis, September, October, November, and December are simply numbered months: The Latin words *quintus*, *sextus*, *septem*, *octo*, *novem*, and *decem* translate into fifth, sixth, seven, eighth, nine, and tenth, respectively.

Quick arithmetic will lead to the conclusion that this year consists of only 304 days. The March equinox occurs during the first month, Martius. The organization of the remaining (roughly) two lunations between the end of December and the beginning of Martius is uncertain and is believed to have simply been lumped together into a long winter month.

This calendar was reformed by Numa Pompilius around 713 BC. The number of days per month were slightly altered; also, the previously unallocated winter days were arranged into two additional months: Ianuarius, referring to the god of beginnings, Janus, and Februarius, which alludes to the purification ritual (Latin *februum*) held in that month.

Month of year (Romulus/Pompilius)	**Calendar of Romulus**	**Calendar of Numa**	**Number of days** (Romulus/Pomilius)
-/1	-	Ianuarius	-/29
-/2	-	Februarius	-/28
1/3	Martius	Martius	31/31
2/4	Aprilis	Aprilis	30/29
3/5	Maius	Maius	31/31
4/6	Iunius	Iunius	30/29
5/7	Quintilis	Quintilis	31/31
6/8	Sextilis	Sextilis	30/29
7/9	September	September	30/29
8/10	October	October	31/31
9/11	November	November	30/29
10/12	December	December	30/29

There is no known method of year numbering of the earliest implementation of the Roman calendar. After 509 BC, years were labeled by the elected consuls in office. Later on, different systems were used. The most notable numbering system is to number years by counting back to the foundation of the city of Rome, or *ab urbe condita* (AUC). The problem with this scheme was that there is no precise record of the date of the foundation of Rome. At some point, whether right or wrong, it was agreed that the city was founded in the year 753 BC.

This reformation led to a transformation of the calendar from being lunar to solar: The number of days in a year after the reformation of Numa is 355. To keep the calendar roughly in synchronization with the solar year, after further reforms, an intercalary month was added every odd year. The length of this intercalary month, called *mensis intercalaris*, or Mercedonius, is 22 or 23 years. The average year length can be calculated over a cycle of four years:

$$\text{average Roman year length} := \frac{355 + 377 + 355 + 378 \text{ days}}{4 \text{ years}} = 366.25 \frac{\text{days}}{\text{year}} \tag{3.2}$$

This duration is remarkably close to that of the solar year.

The Romans used a weekly cycle of eight days, which were simply marked A to H.

3.2 Solar Calendars

Contrary to lunar calendars, solar calendars are based off of the Sun's movement behavior relative to the observer on the surface of the Earth. One of the primary advantages of organizing calendars this way is the synchronization of the calendar with the seasons of the year.

Julian Calendar

The Julian calendar evolved over a number of centuries, and is the final reform to the Roman calendar. Its final version, which will be covered in this section, was introduced by Julius Caesar in the year 46 BC. Following the advice of the Egyptian astronomer Sosigenes, Caesar decided to reform the Roman calendar. The goal of the reform was to keep the months of the Roman calendar, while incorporating the fixed length of a year from the Egyptian calendar. Observations of the tropical year by Greek astronomers had been made a long time ago and offered much better precision knowledge of the tropical year. The new calendar was intended to approximate the solar year as closely as possible. To reach these goals, the following alterations were made to the calendar:

- 10 days were added to the regular year of the Roman calendar. Two days were added to the months Ianuarius, Sextilis, and December. One day was added to the months Aprilis, Iunius, September, and November.
- The mensis intercalaris was abolished. Instead, an intercalary day was to be added to the month Februarius every four years. This brought the average length of the year down to 365.25 days.
- A special intercalary period of 67 days was added once only in 46 BC to bring the year back into synchronization with the solar year. Thus, 46 BC contained 445 days.

As is commonly know, Julius Caesar was assassinated in 44 BC. To honor his legacy, the Roman senate voted to rename the month Quintilis to Iulius. After his assassination, it appears that the order to include an intercalary day every four years was not applied properly. Instead, the intercalary day was added every three years. This was not noticed until 9 BC, when Emperor Augustus decreed that the next three intercalary days be omitted to bring the calendar back in line. Also, the month Sextilis was renamed after Augustus. After these final adjustments, the Julian calendar worked as intended for a long number of years.

Gregorian Calendar

Today, the Gregorian calendar is the internationally most commonly used civil calendar [29].

It is named after Pope Gregory XIII, during whose reign it was introduced. Essentially, the Gregorian calendar is (yet) another reform to the Julian calendar. The original motivation for this reform was of religious nature. The Easter holiday is not fixed in relation to the civil calendar, and was defined as the first Sunday after the full moon following the March equinox in 325 BC. The Julian calendar approximates the solar year's duration: the average length of a Julian year is 365.25 days, whereas the mean tropical year is 365.24219 days long. This means that the Julian year is approximately 11 minutes longer than the tropical year, which leads to a drift of roughly three days every four centuries. Since the Julian calendar had been in use for the better part of 16 centuries, this drift had become noticeable, and in turn also led to the drift of the date of Easter.

The solution to this problem had been discussed for centuries before an actual implementation followed. To increase the precision of the match between the Gregorian and solar year, it

was decided to remove three leap days out of every 400 year cycle. As such, the rule to determine leap years was expanded by the third line of the following rule:

$$\begin{gathered}\text{leap year if:}\\ ((year \bmod 4) \equiv 0) \wedge \\ ((year \bmod 100) \notin \{100, 200, 300\})\end{gathered}$$

Due to this adjustment, the precision of the calendric year was improved from 365.25 days to 365.2524 days. While still not perfect, this calendar only drifts by one day in every approximately 3,300 years.

To eradicate the already established drift from the apparent date, Pope Gregory decreed that 10 days were to be omitted in year 1582 AD. Thus, in year 1582, October 4 was followed by October 15 in the countries that adopted the Gregorian calendar reform that year. Other countries elected to adopt this calendar at a later time, or to omit the 10 days over a longer time span instead.

Solar Hijri Calendar

The Solar Hijri calendar, also called the modern Iranian calendar, is used as the primary civil calendar in Iran and Afghanistan. It was adopted in Persia in 1925 and is a solar calendar.

The epoch of the calendar is similar to the Islamic calendar, it uses the Hejra of the Prophet Muhammad as origin. The new year begins at or near the spring equinox: If the equinox occurs before noon, it begins on the same day; otherwise, it is delayed until midnight the next day. Days begin at midnight.

The common Solar Hijri calendar year has 365 days. It is an observation-based calendar, and is based on the calculation of the occurrence of the spring equinox in Tehran. This calculation determines when leap years are required to be inserted into the year cycle. The leap year rule is quite complex, it is based on recurring cycles which are divided on their approximate length. The largest cycle is 2820 years in length. A total of 683 leap years are distributed over this interval by dividing it into the following two subcycles:

$$2820 \text{ years} = 21 * 128 \text{ years} + 132 \text{ years} \tag{3.3}$$

These two subcycles of 128 and 132 years length are again divided into subsubcycles:

$$\begin{aligned}128 \text{ years} &= 29 \text{ years} + 3 * 33 \text{ years}\\ 132 \text{ years} &= 29 \text{ years} + 2 * 33 \text{ years} + 37 \text{ years}\end{aligned} \tag{3.4}$$

Finally, for each subsubcycle, the following leap year rule applies:

$$\text{leap year if: year} > 1 \wedge \text{ year} \bmod 4 \equiv 1 \tag{3.5}$$

In easier terms, this means that a 29-year subsubcycle has 7 leap years, a 33-year subsubcycle has 8 leap years, and a 37-year subsubcycle has 9 leap years.

It has been shown how leap years are determined for the modern Persian calendar; the following table displays the layout of the months, and how leap years are different from normal years: During leap years, an additional day is added to the month Esfand to extend its length to 30 days, making the year 366 days long.

Month of year	**Iranian-English name**	**Number of days**
1	Fardarvīn	31
2	Ordībehesht	31
3	Khordād	31
4	Tīr	31
5	Mordād	31
6	Shahrīvar	31
7	Mehr	30
8	Ābān	30
9	Āzar	30
10	Dey	30
11	Bahman	30
12	Esfand	29/30

Finally, the Solar Hijri calendar uses a seven-day-week, as outlined in the following table:

Day of week	**Iranian-English name**	**English equivalent**
1	Shanbēh	Saturday
2	Yek-shanbēh	Sunday
3	Do-shanbēh	Monday
4	Se-shanbēh	Tuesday
5	Chār-shanbēh	Wednesday
6	Panj-shanbēh	Thursday
7	Jom'ēh	Friday

3.3 Lunisolar Calendars

Lunisolar calenders are a kind of hybrid between lunar and solar calendars. Generally, lunar calendars have 12 lunar months that reflect the lunations. Contrary to purely lunar calendars, lunisolar calendars use some sort of cycle to insert embolismic months to converge the year with the tropical or sidereal year. The astronomical foundation for this process is the metonic cycle.

Hebrew calendar

The Hebrew or Jewish calendar is different from the previously introduced calendars in certain ways. While it is mostly being used to determine the date of religious events, its daily structure diverges from the otherwise common convention of dividing a day into hours, minutes, and seconds.

The Hebrew calendar divides a day into 24 hours, like most other calendars do. However, hours are subdivided into 1080 units called *chalaks*. 10 chalakim are equal to the length of three

seconds. Also, the day begins at the 18th hour of a day, a full six hours before midnight. It uses a seven-day week, laid out and named as follows:

Day of week	**Hebrew name**	**English equivalent**
1	Yom Rishon	Sunday
2	Yom Sheni	Monday
3	Yom Shlishi	Tuesday
4	Yom Revi'i	Wednesday
5	Yom Chamishi	Thursday
6	Yom Shishi	Friday
7	Yom Shabbat	Saturday

The modern Jewish calendar uses the Metonic cycle. A theoretical moon is employed to this purpose, which is not the same as the real moon. This theoretical moon's position is calculated, and each lunation is initiated by a so-called *molod*, which is the event of new moon. The calculation of the theoretical moon is done in such a way that the interval between molods is constant: the interval between molods is 29 days, 12 hours, and 793 chalakim, which translates to 29.530594 days. This is very close to the average synodic month. Over a period of 16000 years, the molods will stay in synchronization with the average moon to within a day.

The common year features 12 months, but leap years have an additional intercalary 13th month. In the utilized 19-year Metonic cycle the calendar follows, there are 7 leap years in the following pattern: 3, 6, 8, 11, 14, 17, and 19. Thus, a 19-year Metonic cycle contains 235 months, and the time interval between the first and last molod is 6939 days, 16 hours, and 595 chalakim, which translates to an average year length of 365.24682 days. This is slightly longer than a tropical year, and therefore the Jewish year will get ahead of the seasons by one day every roughly 216 years.

The Jewish calendar starts counting time at the epoch named *molod tohu*. This instant can be expressed in the Julian calendar year as 4 hours 204 chalakim into Monday, 7 October 3761 BC.

The rules to determine the length of the year is quite complicated: Every year, common or embolismic, may be deficient, regular, or abundant. A regular common year has 354 days, and a regular embolismic year has 384 days. A deficient or abundant common year has 353 days or 355, respectively. A deficient or abundant embolismic year has 383 or 385 days, respectively.

The year starts with the month Nisan, and is celebrated with the Jewish holiday Rosh Hashanah. However, this day is sometimes postponed, and to compensate, a day is either added to Hesvan, or removed from Kinslev. Furthermore, in leap years, an extra month is inserted after Shevat. In common years, the twelfth month is named Adar. This month is renamed to Adar II, and an embolismic month of 30 days named Adar I is inserted in its place.

Month of year	**Hebrew name**	Common year number of days			Leap year number of days		
		d	r	a	D	R	A
7	Tishri	30	30	30	30	30	30
8	Hesvan	29	29	30	29	29	30
9	Kislev	29	30	30	29	30	30
10	Tevet	29	29	29	29	29	29
11	Shevat	30	30	30	30	30	30
(12)	Adar I	29	29	29	30	30	30
12(13)	Adar II				29	29	29
1	Nisan	30	30	30	30	30	30
2	Iyyar	29	29	29	29	29	29
3	Sivan	30	30	30	30	30	30
4	Tammuz	29	29	29	29	29	29
5	Av	30	30	30	30	30	30
6	Elul	29	29	29	29	29	29

d/D = deficient year, r/R = regular year, a/A = abundant year

Chinese calendar

While the official calendar used in China is the Gregorian calendar, many people still use the modern Chinese calendar, also called Han calendar, to determine the dates of festive events. It is also still commonly used in some regions of China.

In modern China, the day is subdivided into hours, minutes, and seconds. In some regions, however, some variations of the old systems are still being used. The day commonly begins at midnight. Two major systems to divide the day exist:

- The Shí-Kè system:
  - The day is divided into 12 *shí*, each with an equal duration of 120 minutes. Each shí is named and starts at an odd hour, for example, the *zîshí* goes from 23:00 to 01:00.
  - The day is also divided into 100 *kè*, each of which is equivalent to 14.4 minutes (14 minutes, 24 seconds). To indicate a specific time, kè are counted forwards or backwards from the closest of the four midpoints of the shí, which are named by prepending *zhèng* to the shi name *zhèngmǎoshí* (06:00:00), *zhèngwǔshí* (12:00), *zhèngyǒushí*(18:00), and *zhèngyínshí* (00:00). To make this system more compatible with the more commonly used Western hour-minute-second system, efforts have been made to change the duration of a ké to 15 minutes.
- The Gēng-Diǎn system:
  - The *gēng* divides the day into 10 evenly spaced parts, each equivalent to 144 minutes. They are grouped into the day and night gēng.

– The *diǎn* are spaced evenly between the gēng; there are always five diǎ between two gēng. As such, a diǎn is equivalent to 24 minutes. Points in time of the day are expressed by stating the closest previous gēng and the count of diǎn elapsed since then.

A commonly recurring theme of the Chinese calendar is the use of the Heavenly Stems and the Earthly Branches timekeeping systems. These systems are used to name days and years, and provide a set of names for this purpose. While the Earthly Branches represent the Chinese zodiacs, the Heavenly stems refer to the Wu Xing, the teaching of the five elements, in their Yin and Yang forms.

Heavenly Stems		
Heavenly stem	**Mandarin chinese**	**English**
1	jiǎ	Yang Wood
2	yǐ	Yin Wood
3	bǐng	Yang Fire
4	dǐng	Yin Fire
5	wù	Yang Earth
6	jǐ	Yin Earth
7	gēng	Yang Metal
8	xīn	Yin Metal
9	rén	Yang Water
10	guǐ	Yin Water

Earthly Branches		
Earthly branch	**Mandarin chinese**	**English**
1	zǐ	Rat
2	chǒu	Ox
3	yín	Tiger
4	mǎo	Rabbit
5	chén	Dragon
6	sì	Snake
7	wǔ	Horse
8	wèi	Goat
9	shēn	Monkey
10	yǒu	Rooster
11	xū	Dog
12	hài	Pig

These two cycles are often combined to form the *Stem-Branch* cycle, also called the Chinese sexagenary cycle. This cycle is used to record days and years. The names “generated” by this cycle are established by combining the first heavenly stem with the first earthly branch, the second heavenly stem with the second earthly branch, and so on. Once the stems or branches are exhausted, the use of the names simply repeats from the start of the list. Using this system,

60 unique combinations of stems and branches are available.

To organize weeks, there are a number of systems being used:

- The Luminaries week, 7 days: The week starts with Sunday. Sunday and Monday, are named for the Sun and Moon. The remaining five days are named for their respective elements of the Wu Xing, in the following order: Fire, Water, Wood, Metal, Earth.
- The Heavenly Stems week, 10 days: The days of the week are simply named after the Heavenly Stems listed above, with *-rì appended*, such as *jiǎrì*.
- The Earthly Branches week, 12 days: Analogously to the Heavenly Stems week, the days are named after the Earthly Branches.
- The Mansions week, 28 days: The days each have a separate name, which will be omitted for brevity's sake. The 28-day week works well in conjunction with lunations.
- The Stem-Branches week, 60 days: This system follows the Stem-Branch cycle.

The primary method of measuring months is lunar. The lunar month starts on the day of a new moon, and ends on the day before the next one. This means that the duration of the month alternates between 29 and 30 days. Leap months are inserted into the Chinese lunar year by counting the the new moons between the start of the 11th month of the year and the start of the 11th month of the next year. If there are 13 new moons, an embolismic month is inserted. Effectively, this means that the Chinese common lunar year has 353, 354, or 355 days. In leap years, the year has 383, 384, or 385 days. The months are named as follows:

Month of year	1	2	3	4	5	6	7
Chinese name	Zhēngyuè	Èryuè	Sānyuè	Sìyuè	Wǔyuè	Lìuyuè	Qīyuè
Month of year	8	9	10	11	12	13 (leap)	
Chinese name	Bāyuè	Jǐuyuè	Shìyuè	Shìyīyuè	Làyuè	Rùnyuè	

The Chinese calendar has a number of ways to label and count years. Historically, the years since the accession of the last emperor were used to count the years. There is no official system to continuously count the years, but there are a number of (disputed) epochs that are used to reverse-calculate the origin of Chinese time keeping. One of these epochs is the first year of reign of the Yellow Emperor, 2698 BC. Instead of recording years by their number, the Chinese calendar utilizes the Stem-Branch cycle.

3.4 Other calendars

Unix Time

The Unix or POSIX time system is a quite simple calendar that is used in many operating systems as the underlying time scale. Essentially, the Unix time stamp is a single signed integer number.

The Unix time epoch is 00:00:00 UTC on 1 January 1970. Time instants after the epoch are delineated as positive numbers, and time instants before the epoch are negative numbers. The temporal distance between increments is one second.

One important fact about Unix time is that it is not a linear time scale. When it is necessary to insert or remove leap seconds, Unix time either skips or repeats one second. Since, until now, it was only necessary to skip seconds, Unix time has never repeated any. When leap seconds are inserted, there exists an ambiguity, because one time stamp represents two instants of time.

ISO 8601 calendar

The ISO-8601 calendar is an international standard that adopts the Gregorian calendar rules of mapping dates in conjunction with the Coordinated Universal Time system for timekeeping.

The dates expressed in ISO-8601 standard must be consecutive. This means that this calendar is applied proleptically for dates before the Gregorian calendar epoch.

One of the major objectives of the ISO-8601 standard is standardized formatting of date-time related data. It contains standard notations and formatting patterns for both date and time fields. This standard includes formatting options for designating numerical time zone offsets, or just stating time at UTC.

Furthermore, standard formatting patterns for delineating periods (a conjunctive interval of years, months, days, and so on) and intervals (temporal distances between to points in time) are defined.

CHAPTER 4

Calendars in Java Development

In this chapter, we will introduce a number of time-keeping solutions in information technology, with regard to API support for software development.

4.1 Java Date-Time API

The Java Date-Time API is part of the standard libraries that come with the Java Development Kit. At the time of writing of this thesis, the standard API in use with Java is the Date and Time API included with the Java version 1.0 release, `Date`. This functionality was further expanded in 1998 with the Java 1.1 release to include the `Calendar` class.

Java 7

The current Java Date API is split into the previously mentioned two main classes:

`Date`

The `Date` class encapsulates a timestamp and represents specific instant in time. This instant is composed of several different fields, as outlined in the Java API [8]:

- A year y is represented by the integer $y - 1900$.
- A month is represented by an integer from 0 to 11; 0 is January, 1 is February, and so forth; thus 11 is December.
- A date (day of month) is represented by an integer from 1 to 31 in the usual manner.
- An hour is represented by an integer from 0 to 23. Thus, the hour from midnight to 1 a.m. is hour 0, and the hour from noon to 1 p.m. is hour 12.
- A minute is represented by an integer from 0 to 59 in the usual manner.

- A second is represented by an integer from 0 to 61; the values 60 and 61 occur only for leap seconds and even then only in Java implementations that actually track leap seconds correctly. Because of the manner in which leap seconds are currently introduced, it is extremely unlikely that two leap seconds will occur in the same minute, but this specification follows the date and time conventions for ISO C.

These fields are intended to represent a timestamp in UTC notation. However, this is dependent on the host environment of the Java Virtual Machine, and may not always be exactly the case. The input of these fields in the constructor and accessor methods is handled *leniently*, meaning that arguments are handled even if their values fall outside of the normally accepted ranges. For example, a month-day input combination of April 34th will result in May 4th.

The Java 7 Date and Time API also supports time zones. With the `TimeZone` implements offsets for time zones, as well as regionally dependent implementations of daylight savings.

When this API was released, it quickly became apparent that it contains numerous flaws:

- The `Date` object is mutable. This means that the underlying field structure is alterable through use of setters. Although these accessor functions have been deprecated, alteration is still supported through the `Calendar` class. This becomes highly problematic in multithreading environments, since this implementation is not thread-safe and is susceptible to race conditions.

- Non-consistent numbering of fields: As can be seen above, months are numbered starting with 0 through 11, while days are numbered at 1 through 31. Furthermore, the year is offset by 1900.

- (Initial) lack of internationalization support: When released, the `Date` class only supported English input and output with certain formats. Later on, this functionality was extended with the `DateFormat` class, which allows formatting `Date` objects to `String`, and parsing `String` inputs to `Date` objects. To adjust language formatting and regional preferences, the `DateFormat` class is initialized with an instance of `Locale`, which contains geographical formatting and language information. The original `parse` and `format` functions of the `Date` class remain, but are also deprecated.

- (Initial) lack of alternative calendric systems: At release time, the `Date` object only supported UTC notation, which generally identifies days using the Gregorian calendar, and sometimes the Julian. With the release of Java 1.1, the `Calendar` class added functionality to support other calendric systems.

- Interoperability is not given, since the representation of the `Date` object relies on the host system of the Java Virtual Machine. For example, while nearly all modern systems have inbuilt rules to take the possibility of added leap seconds into account, some legacy or niche systems may remain that do not.

- `Date` objects offer precision up to milliseconds. Greater precision is not possible.

Calendar

The Calendar class is an abstract class that was added to perform conversion calculations between time instants that are based upon an epoch, and a set of calendar fields, depending on the implemented calendric system.

Essentially, the Calendar class duplicates much of the functionality of the Date class. Similar to the Date class, the calendar class contains a time field that represents the time elapsed from the Epoch, January 1, 1970 00:00:00.000 GMT (making it very similar to a UNIX timestamp). This time field can be converted into a Date object using the class' getTime() method.

While the Date object is always handled leniently, the Calendar class may be toggled to be either lenient, or not. When in lenient mode, it handles input as the Date class does. Otherwise, an ArrayIndexOutOfBoundsException is thrown when a specified field is out of range.

The date and time of the Calendar object can be set and adjusted through a number of functions. Upon instantiation of the Calendar object, it may either be initialized with the default time zone and locale, depending on the host system, or by passing a TimeZone and/or Locale object to it. Based upon the rules set in the Locale object, the first week of the year or month is determined dynamically. Whenever a method that alters the date by week of month or week of year, this calculation is performed to reach a correct result under the rules set by the Locale and the calendric system.

There are three methods to alter a date and time in a Calendar object. Some of the methods are overloaded and allow a different set of arguments to be passed, but the general idea behind the functionality of methods with the same name is similar.

- The set() method allows the user to set a certain calendar field to a certain value. Each calendar field has a numeric identifier, and the Calendar class provides a number of static final member variables that refer to identifiers for commonly used calendric fields.

- As the name implies, the add() method adds a specified amount of time of a certain calendar field granularity to the current calendar date and time.

- The roll() function works similar to add, but it does not change the time units of larger granularity. For example, if one invokes roll(Calendar.MONTH, 11) on a Calendar object with the date set at March 31, 1998, it will result in the date being set to February 28th, 1998.

In order to cut down on unnecessary recalculations, the timestamp and calendar fields are not immediately recalculated upon alteration of any one field. Instead, invocation of any of the following methods will lead to adjustments of these fields: get(), getTime(), getTimeInMillis(), add(), or roll().

The Calendar class has the same design fault as the Date class: It is mutable and therefore not thread-safe. The intention behind the division of functionality is that the Date class is intended to be used synonymously with an immutable timestamp, and the Calendar class is supposed to produce these timestamps after adjustments and calculations under calendric rules

have been done. However, the class structure of either class, as mentioned before, still allows mutation, thus it is never safe to use these classes in a multithreading environment.

Furthermore, the Java API comes with only one implementation of the abstract `Calendar` class: the `GregorianCalendar`. To use other calendric systems, a user either has to obtain third party libraries that extend the `Calendar` class, or implement the calendric system himself by extending `Calendar`.

Java 8

The Java Version 8 slated for release in early 2014 includes an overhauled Date-Time API. This API was developed under the name ThreeTen (referring to the Java Specification Request 310) by the Core Libraries Group [10]. The reference implementation of this request has been integrated into the JDK 8.

The Java 8 Date-Time API was created with the following features and design principles in mind [24]:

- The Java 8 Date-Time API allows formatting of date and time in- and output, languages and local preferences, through both readily-shipped and customizable formatting styles. The Unicode Common Locale Data Repository (CLDR) [15] is utilized for this purpose.
- Time-zones are supported dynamically and reliably with the Time Zone Database (TZDB) [4]
- The Java 8 Date-Time API comes with a number of predefined calendric systems.
- Thread safety: Most classes of the Java 8 Date-Time API are immutable after instantiation. To alter the value of an object, a new object has to be created. This behaviour is inherently thread-safe.
- Fluent code: Most methods do not permit `null` parameter values, nor do they return `null` values. Method names are self-explanatory and can easily be chained together to create an easily human-readable expression.

Standard Calendar

The Java 8 Date-Time API defines its own time scale, the *Java Time Scale* to deal with certain adjustments that are made to time on a somewhat regular basis. This time scale is divided up into segments. Whenever the internationally agreed-upon time scale is modified, a new segment is inserted into the Java Time Scale. Each segment must fulfill the following constraints [9]:

- the Java Time-Scale shall closely match the underlying international civil time scale
- the Java Time-Scale shall exactly match the international civil time scale at noon each day
- the Java Time-Scale shall have a precisely-defined relationship to the international civil time scale.

Currently, there are two defined and implemented segments in the Java Time Scale.

The first segment starts with the beginning of time, and ends with 1972-11-03T00:00. For this open-ended time interval the internationally accepted time scale is UT1 applied proleptically, which is implemented for the Java Time Scale.

The second segment starts with 1972-11-04T12:00 (the exact point in time where UT1 = UTC), with an open-ended boundary. During this time, the internationally accepted leap second rules are applied. Contrary to the previous implementation in Java, the new API does not allow a day to be made up of more or less than 86400 seconds. To deal with leap seconds, whenever a leap second needs to be inserted on a day, it's duration is divided up and spread out over the last 1000 seconds of that day. Thus, it is possible to insert leap seconds into the time scale while keeping each day at exactly 86400 seconds. The downside of this mechanism is that a second is not guaranteed to always be of the same length.

The Java 8 Date-Time API consists of a number of classes that allow instantiation of temporal values in various granularities. Unless specified otherwise, all classes utilize the ISO-8601 calendar.

- The `Instant` class is intended to represent an instantaneous point in time. It contains a timestamp in the form of two fields. For both fields, a larger value represents a later moment in time.
  - A `long` variable stores the number of seconds elapsed since the Epoch.
  - An `int` that stores the nanosecond-of-second, between 0 and 999,999,999.
- The `Month` enum contains constants for the months of January through December for ready, strongly typed use. Each `Month` object contains the minimum and maximum number of days possible in that month, e.g. Month.FEBRUARY.maxLength() returns 29.
- The `DayOfWeek` enum works analogously to the `Month` enum.
- A `LocalDate` object is a date, with no time information. Furthermore, it does not include any time zone information. As such, objects of this class store temporal values with year, month, and day granularity.
- `YearMonth`, `MonthDay`, `Year` objects work very similar to `LocalDate`. However, each class only saves information of the granularity that it contains in it's name.
- `LocalTime` only contains the time of the day information, but no information about the day, month, or year. The maximum precision is nanoseconds, therefore the value range is between 00:00 and 23:59:59.999999999
- **`LocalDateTime`** is essentially a combination of `LocalDate` and `LocalTime` and one of the core classes of the API. It contains both information about the time of day, as well as the day, month, and year.

Time Zone Support

The Date-Time API offers innate time zone support by either allowing conversion of already created instances with no time zone information into zoned objects, or by direct instantiation. To do this, two differing mechanics exist.

A `ZonedDateTime` object represents a date and time with a corresponding time zone with a time zone offset from Greenwich or UTC.

`OffsetDateTime` and `OffsetTime` objects represent a date and time, and only time, respectively, with a corresponding time zone offset from Greenwich/UTC, but no time zone ID. This means that these classes only include an absolute offset from UTC or GMT, but do not contain information regarding the locality of the time stamp.

`ZoneOffset` objects encapsulate time offsets of the interval $[-18:00; +18:00]$ from UTC, with second precision. To illustrate the use of this class: Utilizing the `ZoneOffset` class, it is possible to define an arbitrary time offset, for example entering a custom time offset that represents the local time of minute and second precision calculated with latitude and longitude of a location.

`ZoneId` objects represent time zones as they are internationally defined by political bodies of respective countries. It is possible to obtain a `ZoneId` by entering a zone ID in standard format, such as `ZoneId.of("America/New_York")`. A number of groups supply time zone information for use with this class, such as IANA (Time Zone Database) [4], Microsoft (Microsoft Time Zone Index) [7], or IATA (IATA Time Zone Codes). The default provider is the IANA TZDB, but different group data can be used by providing the group prefix.

Formatting & Localization

The following classes mentioned above implement the `parse(CharSequence, DateTimeFormatter)` and `format(DateTimeFormatter)` methods:

- `LocalTime`
- `LocalDate`
- `LocalDateTime`
- `MonthDay`
- `Year`
- `YearMonth`
- `OffsetDateTime`
- `OffsetTime`
- `ZonedDateTime`

These methods control how input is parsed and output is formatted. The `DateTimeFormatter` class provides a broad set of pattern variables that allow the user to specify the exact format with which data is to be parsed upon input. Conversely, the same mechanic can be used to edit the output format of temporal data. The `DateTimeFormatter` class can further be customized by invoking `withLocale(Locale)`, and `withChronology(Chronology)`. The `Locale` class is the same class used in the old Java 7 Date-Time API, it contains geographical, political, or cultural information regarding language, punctuation, spacing, orientation and formatting, amongst other things. Initializing a `DateTimeFormatter` instance with a certain `Locale` will lead to input and output being interpreted and produced according to the rules set forth by the Locale object.

Invoking `withChronology` will override the default chronology used with the formatter. Chronologies are further discussed in the Non-ISO Calendar Support section.

Temporal Modeling & Calculation

The `java.time.temporal` package contains a number of collections, classes, interfaces and enums that are used at a low level to perform temporal calculations and conversions. These classes define the underlying structure of classes used in conjunction with the Date-Time API.

Figure 4.1 shows a reduced class diagram, depicting class structure of the Java 8 Date-Time API. In this section, we will elaborate the low-level classes and interfaces that are used to define a calendar system with the API:

- `TemporalAccessor`: This interface defines the read-access methods. It is the baseline interface that is implemented by classes that represent concrete temporal data.

- The `Temporal` interface defines functions that allow a user "write access" object. For example, it allows addition or subtraction of units of time. It needs to be noted that in order for the requirement of thread-safety, the specified methods need to be implemented in a way that the original object is not altered, but a new, adjust instance is returned instead. This is outlined in the API, but ultimately, responsibility lies with the implementing developer. All concrete classes that implement this method provided by the API follow this requirement.

- The arithmetic methods defined in the `Temporal` class require arguments of type `TemporalAmount`. There are two concrete implementations of this interface provided the API:

  - `Period` represents a temporal distance with granularities day, month, and year of the ISO-8601 calendar.
  - `Duration` represents a temporal distance with granularities seconds and nanoseconds.

  The `Period` class represents values for a number of days, months, and years. Objects of type `Period` do not always represent the same temporal distances but are dependent

on context information. This is due to the fact that the year and month do not have constant temporal distances in the Gregorian Calendar. Conversely, instances of `Duration` always cover the same temporal distance.

- The interfaces `Temporal` and `TemporalAccessor` define their member variables as `TemporalField` objects. These objects define a range within which a time interval with type `TemporalUnit` may lie. The interface implementation of `TemporalUnit` defines how long a certain unit may be. Once again, this temporal distance is estimated, due to calendric constraints mentioned above. The enumerations `ChronoField` and `ChronoUnit` provide a rich set of preset constants that reflect the granularities used with the Gregorian Calendar.

- There are three interfaces that set ground rules for abstract date objects and date-time objects: `ChronoLocalDate`, `ChronoZonedDateTime`, and `ChronoLocalDateTime`: These interfaces are to be used when designing classes for use with different calendric systems. The classes `ZonedDateTime`, `LocalDate`, and `LocalDateTime` also implement these interfaces.

Non-ISO Calendar Support

The readily implemented calendric systems that are shipped with the API are the following:

- ISO-8601 (Gregorian)
- Hijrah (Islamic)
- Imperial Japanese
- Minguo
- Thai Buddhist

Adding a new calendric system requires an implementation of the interfaces `Chronology`, `ChronoLocalDate`, and `Era`. Java 8 names a calendric system a chronology.

Most of the calendar logic representing the rules unique to the calendric system will be implemented within the `ChronoLocalDate` implementation, while the subclass of `Chronology` is a factory for instantiation of objects. For each of the pre-implemented calendric systems, an implementation of calendar specific classes is provided for use.

Legacy Date-Time Compatibility

The legacy Date-Time API that comes with versions of Java 7 and prior has been extended to allow interoperability with the new API. The `Calendar` class was extended to be convertible into `Instant`, the `GregorianCalendar` class may be converted into as well as be instanced from `ZonedDateTime`. The `Date` class is also two-way convertible to and from `Instant`.

4.2 Joda-Time

The Joda-Time API was implemented due to lack of a well defined Date-Time API for Java. Since many users found the default implementation discussed above lacking and prone to bugs, the Joda-Time API started development in 2003. [25]

One of the core design differences of Joda-Time is that it does not strictly enforce immutability. To be more specific: the most used classes in Joda-Time are all immutable. Invoking methods to change certain calendric fields on these immutable objects has the same behavior as the Java 8 Date-Time API objects: a new object with adjusted field(s) is returned. This behavior is illustrated in Algorithm 4.1. It has to be noted that in this algorithm, each line of code creates a new `DateTime` instance.

However, it is possible to convert a number of temporal objects into a mutable version of themselves. The advantage of this implementation is that it is possible to change multiple member fields of one object without having to instantiate a new object for every alteration. However, in most of the time, this functionality should not be needed. Finally, each of the mutable classes contains a method to convert itself into an immutable object again (which does, however, instantiate another new object). The use of this mechanic is visualized in Algorith 4.2. While this algorithm performs the exact same task as Algorithm 4.1. The number of object instances, however, is decidedly lower: Only line 1 and line 5 of this algorithm create an object instance. Utilizing this method, only two objects are necessary, regardless of the number of fields that are changed. Compared to the method used in Algorithm 4.1, this can lead to a substantial performance difference.

```
1 DateTime currentDateTime = DateTime.now();
2 DateTime changedMonthDateTime = currentTime.withMonthOfYear(11);
3 DateTime changedYearDateTime = changedMonthDateTime.withYear(2013);
4 DateTime finalDateTime = changedYearDateTime.withHourOfDay(17);
```

Algorithm 4.1: Field adjustments of immutable object

```
1 MutableDateTime mutableCurrentDateTime = MutableDateTime.now();
2 mutableCurrentDateTime.setMonthOfYear(11);
3 mutableCurrentDateTime.setYear(2013);
4 mutableCurrentDateTime.setHourOfDay(17);
5 DateTime finalDateTime = mutableCurrentDateTime.toDateTime();
```

Algorithm 4.2: Field adjustments of immutable object through conversion to mutable object

Concepts

Joda-Time features a number of core concepts that are most important for the end user to use:

- `DateTime`, `LocalDate`, `LocalDateTime`: These three classes are the high-level API objects that are most commonly used by the end-user of the API. The `DateTime`

represents a date and time, as the name says, and uses the ISO-8601 calendric system per default, and no time zone, unless specified differently. The `LocalDate` and `LocalDateTime` work analogously, but remove the possibility of specifying a time zone.

- `Chronology`: Joda-Time comes with a set of pre-implemented chronologies that represent calendric systems. Contrary to the Java 8 API, regardless of which `Chronology` is utilized, the user still uses the same type of object to represent various points in time, e.g. the `DateTime` class. To change the default calendric system in use to a custom one, a variety of methods and constructors is available to set a chronology.

- `DateTimeZone` objects represent time zones. The API is based on data of the Time Zone Database. `DateTimeZone` objects are applied to chronologies to adjust time calculation logic. It is also possible to change the offset of a time zone with the `adjustOffset` method to a custom offset.

Temporal Modeling

- The `Instant` in Joda-Time works much the same as the Java 8 version - it represents a timestamp from the January 1, 1970, 00:00 era. One notable difference is that it offers precision only up to the millisecond.

- An `Interval` in Joda-Time represents the temporal distance between two instants, as per the definition of the `Instant` class. The `Interval` is half-open: The start instant is inclusive, the end instant is exclusive. Furthermore, both instants have to use the same `Chronology` and have to be in the same time zone.

- `Duration`: This class is defined the same as the Java 8 duration, but, again, only with millisecond precision.

- A `Period` represents a temporal distance with a set of fields with the following granularities: years, months, weeks, days, hours, minutes, seconds, and milliseconds. Once again, these objects are dependent on the date they are relative to, and can be put into context by relating them to an `Instant`. The set of fields that are usable may be restricted by the user by providing an instance of `PeriodType`.

Formatting & Localization

Joda-Time comes with its own formatter that allows a wide degree of customization. The `DateTimeFormatter` works very similar to its Java 7 and 8 counterparts. Beyond the standard formatters provided by the framework, it is possible to design custom patterns with the CLDR date/time pattern notation. Furthermore, it is possible to create a formatter that is capable of parsing date and time information that is not easily represented by a pattern. Through methods of the `DateTimeFormatterBuilder` class, it is possible to incrementally build a set of fields with constraints that is capable of parsing said timestamps. Regarding localization, Joda-Time is fully compatible with JDK `Locale` objects.

Supported chronologies

The following calendric systems are supported by default:

- ISO 8601 Calendar
- Buddhist Calendar
- Coptic Calendar
- Ethiopic Calendar
- Gregorian Calendar
- GregorianJulian Calendar
- Islamic Calendar
- Julian Calendar

Custom Chronologies

The Joda-Time abstract class to implement custom calendar systems is the `Chronology`. Extending this class allows for flexible implementation of custom chronologies, and all provided chronologies extend it as well. However, the abstract `Chronology` class only supports a set of predefined, named granularities: Millisecond, second, minute, hour, halfday, day, week, weekyear, month, year, century, and era. The semantics of these granularities rely on the calendar-specific implementation of the `DateTimeField` and `DurationField` abstract classes. By implementing them in conjunction with a custom implementation of `Chronology`, a new calendric system may be defined. It is, however, not possible to add new granularities, nor is it possible to change the accessor methods of the provided set of granularities.

4.3 τZAMAN

The τZAMAN framework was developed to allow users easy implementation and conversion of temporal data represented in different calendric systems and languages. The primary purpose of this framework is to allow definition and integration of custom calendric systems. This is done by creating and relating their granularities with one another by setting granularity mappings, and then allowing conversions between granules of granularities of different calendric systems, if possible.

However, contrary to previous date-time APIs discussed previously, the τZAMAN framework does not have a pre-built definite time scale that is related to the actual time. The framework does not contain a clock, nor does it *measure time* [30]. Instead, it is responsible for representing, converting and serializing and deserializing of measured time related data. Any calendric system implemented remains an abstract model, and time values can only be entered ad-hoc.

The framework makes some design decisions for the time domain: it is to be discrete, and bounded. However, there is no set minimum granularity. All granularities to be used with a

calendric system are fully customizable by the user, including the the chronon. It is possible to implement commonly used calendric systems, such as the Gregorian or Buddhist calendars, or to implement a completely novel system that does not exist in the real world (say, a custom calendar system for a fantasy world such as Lord of the Rings).

τZAMAN is implemented as a local single-part system, as well as a client-server architecture. When started up locally, the host machine acts as both client and server. When launched in a distributed scenario, calendric systems may be fetched from the remote server to be used in conjunction with the locally available systems.

An integral part of the framework is its ability to load and store calendric systems at runtime. Each calendric system consists of a pair of XML mappings and a Java class.

The XML mapping defines the structure of the calendar. All granularities of a calendric system are outlined in the XML file, as well as the mappings between them. Algorithm 4.3 displays a sample taken from the Gregorian Calendar implementation taken from the τZAMAN website. It can be seen that each calendar defines its own chronon by referring to a specific granularity, in this case the *second*. Furthermore, an example for a regular and irregular mapping each is visible. The implementation of the method `castMinuteToSecond`, implemented in the Java file paired to the XML file is visible in Algorithm 4.4.

```
1  <calendarSpecification underlyingGranularity="second"
   implUrl="./ADGregorianCalendar.class">
2      <granularity name="second">
3          <irregularMapping from="minute" relationship="coarserToFiner">
4              <method name="castMinuteToSecond"/>
5          </irregularMapping>
6      </granularity>
7      <granularity name="minute">
8          <irregularMapping from="second">
9              <method name="castSecondToMinute"/>
10         </irregularMapping>
11     </granularity>
12     <granularity name="hour">
13         <regularMapping from="minute" groupSize="60"/>
14     </granularity>
15 </calendarSpecification>
```

Algorithm 4.3: Sample truncated XML calendar specification, taken from Gregorian Calendar

Modeling calendars

The underlying model of calendars used in the τZAMAN framework closely adheres to the principles laid out by Bettini et al. [6]. As such, it declares a collection of classes that are of central importance to the framework:

```
1 public static long castMinuteToSecond(long minute){
2     long seconds;
3     seconds = (NUM_SECS_MINUTE * (minute - 1)) + 1;
4     return seconds += getNumLeapSecsFromMin(minute);
5 }
```

Algorithm 4.4: Sample reference implementation of `castMinuteToSecond`, taken from Gregorian Calendar

- A `TimeValue` object is a numbered object on the time line. The time line is essentially a numbered list of objects.
- The `Granularity` class represents the granularity concept.
- A `Granule` instance refers to a concrete point in time with the temporal distance of a given granularity. There are three different kinds of granules implemented in the framework (but only one has its own class):
  - A determinate granule represents a specific `TimeValue` instance.
  - An indeterminate granule represents a randomly chosen `TimeValue` between a lower and upper bound, also represented by `TimeValue`. The probability with which each discrete point in time between those bounds is chosen is customizable by implementing different probability functions.
  - A `NowRelativeGranule` represents a `TimeValue` instance that is relative to the current time, e.g: *now* + 5 days.
- The `Mapping` class is abstract and defines the underlying concept of a granularity mapping: It defines a `from` and `to` granularity, contains information whether the mapping is from coarser to finer, congruent, or coarser to finer.
  - A concrete implementation of `Mapping`, `RegularMapping` is used for modeling regular mappings between granularities, such as day_to_week. The multiplicity of this example mapping is always 7, and this value is recorded in the `RegularMapping` instance.
  - The class `IrregularMapping` is used to model irregular mappings, such as days_to_year. Since the actual count of days in a year is dependent on context information, the actual count has to be determined on a case-by-case basis. The method name of the method that performs this calculation is recorded within the `IrregularMapping` and invoked with reflection at runtime.
- `Calendar`: This class contains a set of granularities, as well as the mappings between them. Furthermore, it contains the logic needed to perform irregular mapping resolutions.
- The `CalendricSystem` class is a container for a set of `Calendar` objects, the granularities of which are mapped to one another with `Mapping`. This combined set of mappings is stored in a structure called textttGranularityLattice.

Using the τZAMAN framework has some serious drawbacks, and at this stage it can be said that this framework is of value for its interest at best. During inspection of the documentation, website, and code, some problems quickly became apparent:

- The framework does not come with a set of implemented calendars. There are three demonstrational calendar implementations available for download on the website: the Gregorian Calendar, the University of Arizona Academic Calendar, and the University of Arizona (Limited) Calendric System. The latter two are both based on the Gregorian Calendar.

- Work on the project seems to have been discontinued in 2006. There are numerous locations in the source code that are not implemented and/or documented.

- The code is not written robustly. There are numerous `final static int` constants used to set the type of an object. It is easy to mix these up when instanciating an object, creating undesired behavior. This is likely a result of the state the Java programming language was in when work on τZAMAN began in 2003. At this time, the J2SE v1.4 was the most recent release, which did not yet include the `enum`.

- The source code is littered with comments discussing the necessity of fields, methods, and classes. A refactorization of the source code is necessary to get rid of unneeded or otherwise implemented functions. Furthermore, there is a section "Open Issues" on the website [13]. It contains a rudimentary implementation of a bug ticket tracking system, which indicates the lack of utilization of a professional development process, and the tools to support it.

- Parsing and deserialization of XML documents is done manually instead of using a readily available, standardized method, such as JAXB. This is also likely due to the (relatively) early state the Java programming language was still in.

Regardless of these implementational downfalls, the τZAMAN framework still holds value; analyzing its structure and concepts offer an alternative to contemporary date-time APIs.

4.4 Date4J

Date4J was created to streamline a Date-Time API for use in conjunction with relational databases. Its goals are to be simpler than Joda-Time, have a better design than the Java 7 Date-Time API, offer more precision than either of them, be performant on mobile devices, and allow customization of the way temporal calculation is performed.

To keep things simple, the author of Date4J decided to focus on one calendric system, the Gregorian Calendar. This calendar is applied proleptically, and is valid in the year range [1-9999]. It ignores the transition from the Julian to the Gregorian calendar, leap seconds, summer time, and time zones in general. This choice was made because, according to the author of Date4J, most databases behave this way [23].

The core class of Date4J is the `DateTime` class. It may represent a date, a time, or a date-time timestamp. The underlying concept of the class separates behaviour into two categories: Basic, and computational operations.

A `DateTime` object in "basic mode" simply acts as a container for a `String`, which does not have to conform to any kind of format. This functionality is provided to allow to keep the exact formatting of database result sets. Only the `String` constructor, `toString()` and `getRawDateString()` may be used in this mode.

When a `DateTime` object is constructed explicitly, or when computational methods are invoked, the object enters the "computational mode". At this time, a `String`, if present, is parsed, and calendric fields are filled in. Any getter, comparison operation, calculation, or formatting operation will trigger this mode.

There are a number of ways to acquire a `DateTime` object:

- A constructor to enter a date with each of the following fields entered explicitly and separately as an `Integer`: Year, month, day, hour, minute, second, nanosecond.
- A constructor that accepts a `String`. *Any* `String` may be entered upon construction of the object, but as soon as a computiational method is called, the input will be parsed. The accepted formats are non-customizable and are documented in the API. It is possible to check an input `String` with the static method `isParseable(String)`.
- Static methods to construct instances for dates, instants presented by milliseconds or nanoseconds, for a time, as well as methods to create an instance for the current day, or the current day and time.

Date4J utilizes the JDK's `Locale` class to localize output strings. `DateTime` objects may be formatted for output without a `Locale`, which will result in strictly numerical `String` output, with the format supplied as an argument. By supplying a `Locale`, the output will be formatted according to the supplied format, but non-numerical fields will be filled in according to the `Locale` language.

4.5 Summary

Each of the mentioned APIs has its advantages, and disadvantages. The following list summarizes these for easy comparison of the different APIs and frameworks against one another.

- Java 7 API:
  - Advantages:
    - Widely used
    - Java language standard
    - Extensible, allows implementation of other calendric systems
  - Disadvantages:
    - Mutable objects, not thread-safe

        * Only calendar implementation: Gregorian Calendar
        * Non-consistent implementation of offsets
        * Maximum precision: millisecond

- Java 8 API:
  - Advantages:
    * Java language standard
    * Immutable objects, thread-safe
    * Maximum precision: nanosecond
    * Extensible, allows implementation of other calendric systems
    * Comes with a set of pre-implemented calendric systems
  - Disadvantages:
    * Does not allow conversion to mutable object, this may cause possible performance issues
    * Complex API

- Joda-Time
  - Advantages:
    * Widely used
    * Immutable objects, thread-safe
    * Allows conversion to mutable objects for performance reason
    * Comes with a set of pre-implemented calendric systems
  - Disadvantages:
    * Maximum precision: millisecond
    * Complex API

- τZAMAN:
  - Advantages:
    * Maximum precision: freely choosable
    * Calendars freely implementable
    * Run-time calendar deployment
    * **Very** complex API
  - Disadvantages:
    * Does not come with pre-implemented calendric systems
    * Development ceased in 2006, the framework is unfinished
    * Implemented on very outdated Java standard
    * Error-prone code

- Date4J:
  - Advantages:
    * Maximum precision: nanosecond
    * Very simple to use: only one class relevant to the user
    * Immutable objects, thread-safe
    * Made to work seamlessly with most relational database implementations
  - Disadvantages:
    * Only calendar implementation: Gregorian Calendar
    * Not made for extensibility, no other calendar implementations possible
    * Does not implement modern timekeeping adjustments such as the leap second
    * Does not implement time zones
    * Only allows years between 1 and 9999 (most databases have the same constraint)
    * Does not allow conversion to mutable object, this may cause possible performance issues

At this time, the Java 7 and Joda-Time Date-Time APIs are the most widely used date-time frameworks in use with the Java programming language. This is likely to change within the next few years, as the Java 8 update addresses many of the issues that were pointed out with the Java 7 API.

The Date4J API is useful for simple and concise tasks, mostly in use with databases. Here it performs well, but is not very useful for tasks that require the time to be synchronized with the actual time as closely as possible.

Finally, the τZAMAN framework offers some unique insights into how time can be modeled in an information technology environment. The implementation is very complex, and has been discontinued as well as outdated, so its practical usefulness is very limited.

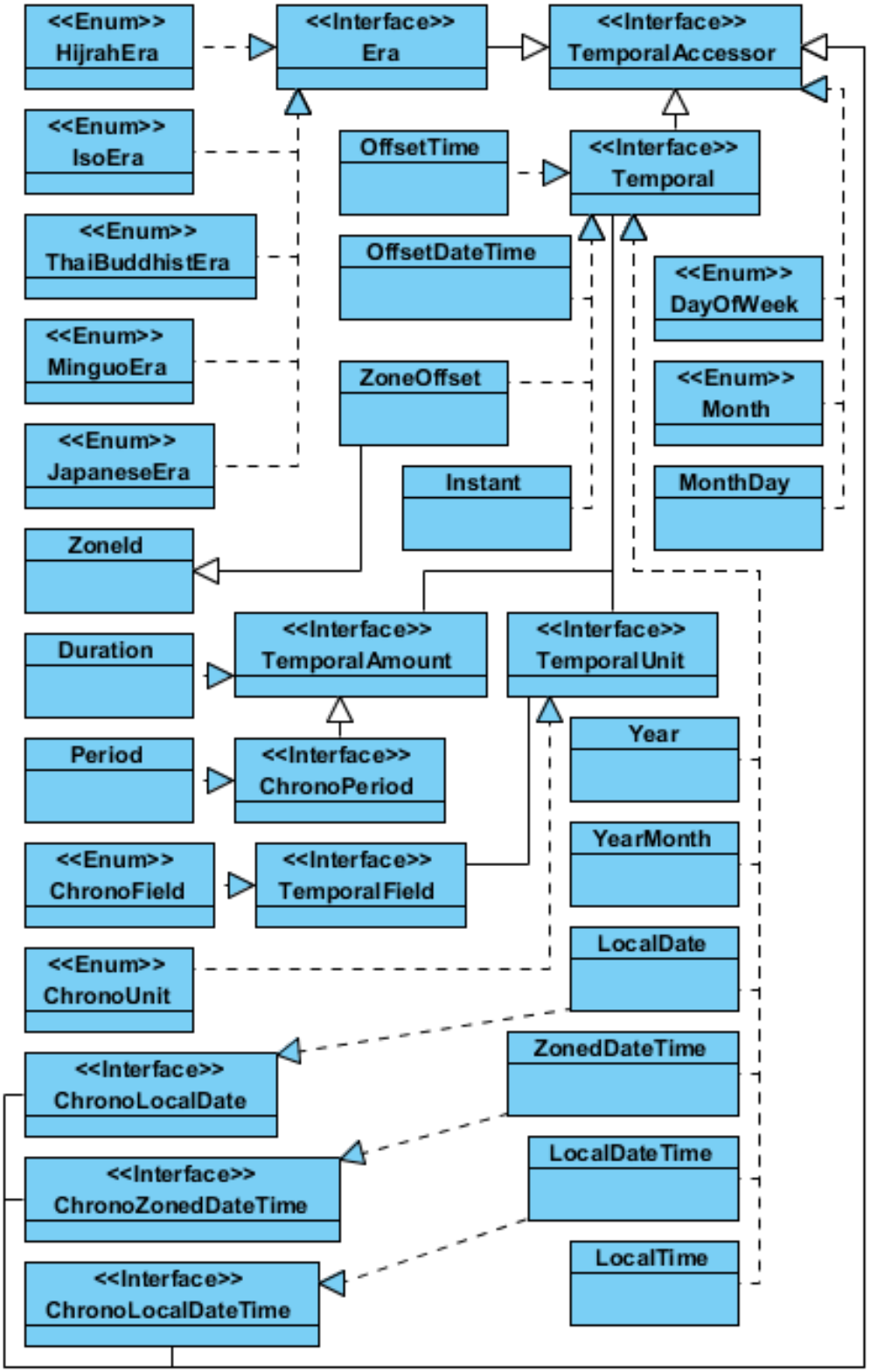


Figure 4.1: Java 8 Date-Time API Class Diagram

CHAPTER 5

TimeBench

TimeBench is a software development project that is conducted by the Information Engineering Group (IEG), part of the Vienna University of Technology, Faculty of Computer Science, Institute of Software Technology & Interactive Systems, as well as the St. Pölten University of Applied Sciences, Austria. The project is being developed by the Centre of Visual Analytics Science and Technology (CVAST) team, the main mission of which is *"to design and develop innovative methods for data interpretation to capture the daily flood of information in interactive visualizations and analyses."* [19] Visulization techniques of temporal data are a focal point of the research being done by the CVAST team, and TimeBench is intended to provide a general framework for a wide array of data visualization methods.

5.1 Structure

TimeBench is designed in a modular structure, with several packages that are integrated to form a functional data visualization framework. It consists of the packages listed below. Refer to Chapter 6 of the TimeBench submission [27] for more detailed descriptions of the listed packages.

Data Structures

The `data` package provides a model of low-level temporal primitives. More specifically, the following data objects are implemented:

- `Instant`: Defined according to the concept discussed previously.
- `Interval`: Three versions of intervals are implemented: The definite interval discussed previously, and two indefinite intervals: An interval with an imprecise start, and an interval with an imprecise end.

- `Span`: Once again, three versions of spans are implemented: The definite span discussed previously, and versions of minimum and maximum possible spans for an imprecisely defined span.

To this end, the library provides a set of generic data structures called `TemporalDataset`, a set of data items containing non-temporal attributes called `TemporalObject`, and a timing function that maps each temporal object to exactly one temporal element. Furthermore, functionality to import and export data sets is contained in the `data` package. These data sets are stored in a relational data tables.

Calendar Operations

The `calendar` package contains the modeling logic to implement calendric systems in TimeBench for use with the framework. For these modeling purposes, classes that implement the granularity concept have been created. It is also responsible to perform calendric calculations and conversions between granularities, granules, of possibly different calendric systems. Finally, logic to deserialize calendric system granularities from XML are implemented. The partial implementation of these requirements is a central aspect of this thesis; implementation details are further discussed in the following documentation section.

Transformations On Data Tables

The data structures implemented in TimeBench may be analyzed automatically through implementation of data transformations over said data structures. The toolkit library prefuse provides a large set of implementations of different information visualization techniques. In addition to the provided ability to modify visual mappings, TimeBench extends the functionality of prefuse to allow data transformations. Refer to the prefuse submission [14] for further details.

Visual Mapping, Rendering, and Interaction

The `TemporalDataSet` is an extension of a prefuse `ParentChildGraph`. This means that prefuse's visualization objects directly inherit the data contained within the `TemporalDataSet` and allow the prebuilt renderers to convert data sets into visualizations of themselves seamlessly.

5.2 Current implementation

The `calendar` package of TimeBench is of central importance to this paper. Its classes are intended to address the complexities of modeling calendars discussed previously. To this end, the granularity concept for modeling calendars is realized.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
M	M	M	M	M	V	V	V	V	V	V	V	V	C	C	C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	C	C	C	T	T	T	T	T	G	G	G	G	G	G	G

Table 5.1: Identifier bit field designation

Common concepts

For in/out-operation performance reasons, granularities are addressed by a single integer field throughout TimeBench. Each granularity is defined within a certain context, and these context identifiers need to be stored within and retrievable from the granularity identifier.

By default, the `int` implemented by the standard Java Virtual Machine is a 32-bit signed integer with the value range -2^{31} to $2^{31} - 1$. TimeBench subdivides this 32-bit field as follows:

- Calendar Manager Identifier (**M**): 5 bits, valid value range: 0 - 31 (0b00000 - 0b11111)
- Calendar Manager Version Identifier(**V**): 8 bits, valid value range: 0 - 255 (0b00000000 - 0b11111111)
- Calendar Identifier (**C**): 7 bits, valid value range: 0 - 127 (0b0000000 - 0b1111111)
- Granularity Type Identifier (**T**): 5 bits, valid value range: 0 - 31 (0b00000 - 0b11111)
- Granularity Identifier (**G**): 7 bits, valid value range: 0 - 127 (0b0000000 - 0b1111111)

Combining these bit fields into a single 32 bit `int` yields the the identifier layout shown in Table 5.1. Due to this structure, it is possible to address calendar managers and calendars with this same identifier by using only the necessary significant bits. Therefore, within certain classes, it is often easier to use the identifier up to the necessary point of precision. For example, to uniquely address a `Calendar`, only 20 bits consisting of the calendar manager identifier (M), the calendar manager version identifier (V), and the calendar identifier (C) are required. Conversely, when registering and accessing granularities within a calendar, only the last 12 bits consisting of the granularity type identifier (T) and granularity identifier (G) are necessary.

`CalendarManager`

The central class of the TimeBench calendar package, which handles implementation of calendric systems, is the `CalendarManager` interface. As the TimeBench CalendarManager revolves around the granularity concept introduced earlier, this class provides a multitude of ways to obtain and modify granularity objects and granules. A class extending the `CalendarManager` class represents an implementation of a calender system. Each instance of `CalendarManager` implements calendar-specific logic.

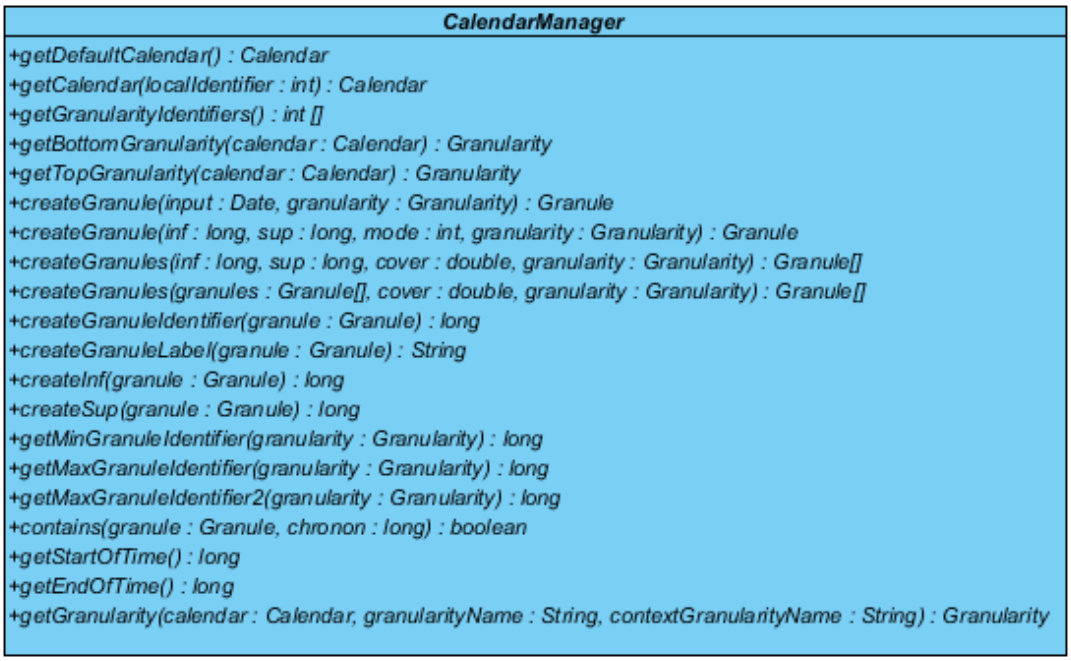


Figure 5.1: CalendarManager Interface Specification

At the current stage, this is limited to calculating the `inf` and `sup` of a `Granule` of a given `Granularity` with a specific context `Granularity`. Also, the current implementation requires an implementation to provide users with readable labels for a `Granule`. A full interface specification is shown in Figure 5.1.

Since the `CalendarManager` is not only identified by a calendar manager identifier, but also a calendar manager version identifier, it is the intention of the design that multiple implementations of the same calendar system may exist in different `CalendarManager` instances. These instances can (but need not) use class inheritance to realize this goal.

Extending implementations of `CalendarManager` with additional sets of granularities is done by alteration of Java code. The respective `CalendarManager` implementation has to be changed to allow instantiation of `Granularity` in some way or another. The same can be said analogously of `Calendar`.

Instances of `CalendarManager` implementations are handled by the `CalendarFactory` class. This class defines access points for developers to retrieve `CalendarManager`, `Calendar`, and `Granularity` objects. Since `CalendarManager` implementations are versioned, all of the object retrieval methods mentioned are overloaded and allow the user to enforce strict versioning. If not specified otherwise, the `CalendarFactory` will use a default implementation version of `CalendarManager` to fetch the requested objects.

For more details, refer to the JavaDoc of `CalendarManager` in the `calendar` package.

`Calendar`

In its current implementation status, objects of the `Calendar` class serve as a simple mapper of functionality: The `Calendar` object is injected into instances of `Granularity` and it itself contains a reference to a `CalendarManager` object. It is intended to be serializable and contains a `@XmlJavaTypeAdapter` annotation that references a nested class responsible for XML marshalling. However, this class is not implemented.

`Granularity`

The `Granularity` class is a direct implementation of the granularity concept. Instances of `Granularity` are intended to represent different granularities, such as *day*, *week*, *month*, etc.

Each `Granularity` object retains a reference to a mapping object of type `Calendar` that connects it to its `CalendarManager`. Most of the public methods exposed by the `Granularity` implementation simply relay method invocations to the `Calendar` object. The `Calendar` in turn relays these invocations further up the chain to the `CalendarManager`, since these implementations are the only point where actual calculation can be performed.

CHAPTER 6

TimeBench Calendar

Within the scope of this thesis, a number of changes were performed to improve the `calendar` package of the TimeBench library. The work was done with the following goals in mind:

- Create a common interface to create granules, retrieve granularities, and access calendar information.
- Allow developers to easily access and convert granularity identifiers.
- Provide a common access to retrieve implemented calendars.
- Extend existing calendar implementations through addition of Java calendars and XML representations of calendar structure.
- Provide a sample implementation of `CalendarManager`.

6.1 Realization

The implementational changes to realize the goals listed above are concentrated around the following classes.

`Granularity`

The major change to the `Granularity` class revolves around JAXB annotations that enable it to be serialized and deserialized from XML. The XML example shown in Algorithm 6.1 denotes a possible XML specification of a granularity.

One important aspect of the refactorizations done is that `Granularity` objects contained within `Calendar` objects are now treated as a kind of `final` blueprint. They should not be altered after deserialization (besides initialization of helper fields), and are context-free. This changed was done to ensure that the base structure of a calendar always remains the same and is

not altered. To set a granularity into context of another, the method `setIntoContext(Granularity)` returns a cloned object of the original, with every aspect identical, but a context granularity set. Thus, the new workflow to obtain a contextualized `Granularity` is as follows:

1. Obtain a `Granularity` through a `Calendar`, `CalendarManager`, or through the `CalendarRegistry` (*granularity*).

2. Similarly obtain a context `Granularity` (*contextGranularity*).

3. Invoke `granularity.setIntoContext(contextGranularity)`. The returned `Granularity` object is the desired result.

The fields and their annotations are depicted in Algorithm 6.2. A number of fields were added to support performant data access after deserialization:

- The `GranularityIdentifier` class is a simple JAXB annotated class that serves as a wrapper for the granularity identifier and granularity type identifier fields.

- Within calendric systems, it only makes sense to put a granularity into context of a coarser granularity. For example, a granule of day granularity in can be put into context of a granule of granularity year, but the reverse is not true. This behavior is modeled through the implementation of a list of permitted context granularity identifiers.

- A granularity label of type `String` was added to improve meaningfulness of `Granularity` objects (for instance, a `Granularity` object may now be labeled as "week").

- A calendar needs to have a top and a bottom granularity. To this end, the boolean fields `isTopGranularity` and `isBottomGranularity` were added to identify these granularities.

```
1 <granularity isBottomGranularity=”true”
2     <identifier identifier=”1” typeIdentifier=”1”/>
3     <granularityLabel>Millisecond</granularityLabel>
4     <!– permitted context granularity identifier 1 –>
5     <permittedContextIdentifier identifier=”2” typeIdentifier=”1”/>
6     ...
7     <!– permitted context granularity identifier n –>
8     <permittedContextIdentifier identifier=”10” typeIdentifier=”1”/>
9 </granularity>
```

Algorithm 6.1: `Granularity` XML Specification

The `GranularityIdentifier` class simply serves as a wrapper for the granularity identifier and granularity type identifier mentioned above.

```
1  @XmlElement(required = true)
2  private GranularityIdentifier identifier;
3  @XmlElement(required = true)
4  private String granularityLabel;
5  @XmlAttribute
6  private Boolean isTopGranularity = false;
7  @XmlAttribute
8  private Boolean isBottomGranularity = false;
9  @XmlElement(required = true, name = "permittedContextIdentifier")
10 private List<GranularityIdentifier> permittedContextIdentifiers = new ArrayList<>();
```

Algorithm 6.2: `Granularity` Java Field Specification

`Calendar`

The `Calendar` class has gone through a number of changes. Before, it simply served as a mapping object between `Granularity` and `CalendarManager` instances. Now, it is also an XML annotated container class that contains a full set of `Granularity` objects. The XML structure of a `Calendar` is depicted in Algorithm 6.3. Similarly to the `Granularity` class, the `Calendar` was extended by a number of fields shown in Algorithm 6.4:

- A `List` of `Granularity` objects was added to implement container functionality. This list of objects is serializable from XML.
- The `Calendar` needs to refer to a specific `CalendarManager`. To this end, the `Calendar` contains fields that hold the calendar manager version identifier and calendar manager identifier. With these fields, an instance of `CalendarManager` can be obtained.
- Finally, the `Calendar` needs to have its own identifier.

```
1 <calendar localCalendarIdentifier="1" localCalendarManagerIdentifier="2"
  localCalendarManagerVersionIdentifier="1">
2     <granularity> ... </granularity> <!– granularity 1–>
3     ...
4     <granularity> ... </granularity> <!– granularity n–>
5 </calendar>
```

Algorithm 6.3: `Calendar` XML Specification

`CalendarManager`

The `CalendarManager`, originally an interface, was changed to an abstract class. Since it is also a container, a number of operations related to data retrieval of the contained information

```
1 @XmlAttribute
2 private int localCalendarIdentifier;
3 @XmlElement(name = ”granularity”)
4 private List<Granularity> granularities;
5 @XmlAttribute(required = true)
6 private int localCalendarManagerIdentifier;
7 @XmlAttribute
8 private Integer localCalendarManagerVersionIdentifier = null;
```

Algorithm 6.4: `Calendar` Java Specification

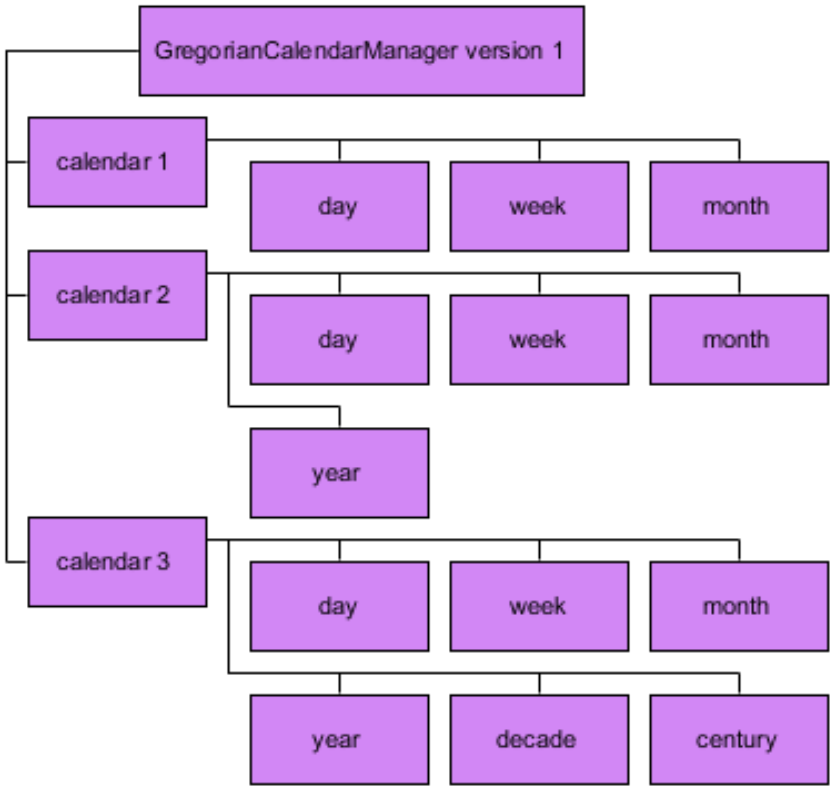


Figure 6.1: Sample CalendarManager instance structure

is always the same, and thus is implemented in the abstract parent class. Data structures to support the storage of the contained `Calendar` objects are also defined in the parent class. The full containment structure put together from the class definitions of `CalendarManager`, `Calendar`, and `Granularity` is shown in Figure 6.1.

`CalendarRegistry`

The `CalendarRegistry` class, implemented with the Singleton and Registry pattern, provides a single point of access for retrieval of CalendarManagers. It represents the central access

Manager Identifier	Manager Version Identifier	CalendarManager instance
1	1	GregorianCalendarManager v1
	2	GregorianCalendarManager v2
2	1	ChineseCalendarManager v1
	2	ChineseCalendarManager v2
	3	ChineseCalendarManager v3

Table 6.1: Calendar Registry Data Structure

point for `CalendarManager` instances. Upon first access, it self-initializes its `TreeMap` registry with instances of `CalendarManager`. When adding new implementations of `CalendarManager`, object instantiation and registration should be added within the `initialize` method. In the future, inversion of control could be implemented here by use of dependency injection to redistribute instance management responsibilities to an instance management container.

The instances of `CalendarManager` are stored in a double nested `TreeMap` data structure. Since `TreeMap` objects sort their entries by their key values, the stored objects are sorted ascending. This data structure allows quick access to specific instances without the need to iterate through lists, simply by querying the `TreeMap` with the calendar manager identifier and calendar manager version identifier. If no version is enforced, the lowest version entry is assumed to be the default implementation. An example data structure is shown in Table 6.1.

Utility

`IdentifierConverter` Given the somewhat complex nature of the identifiers defined in the common concepts section, the `IdentifierConverter` class was implemented using the Singleton pattern.

To ease usability of identifiers, code conventions were introduced for fields, methods, and arguments within the `calendar` package to differentiate between the scope of identifiers:

- An identifier labeled as **global** will always use the full 32-bit int field and initialize the significant bits in their proper position according to their value.
- Identifiers without scope are labeled as **local**. This means that they may only use the first number of bits of the `int` field, according to the number of bits and subsequent value ranges described above.

The `IdentifierConverter` serves as an abstraction layer for the conversion between local and global identifiers. It provides methods that perform bitshifting and bitwise logical operations to extract local identifiers from a global identifier, as well as assembling a global identifier out of a set of local identifiers.

We will illustrate the previously mentioned convention with the following arbitrary example:

1 CalendarManager **local**Identifier = 3 := 0b00011
2 CalendarManagerVersion **local**Identifier = 37 := 0b00100101
3 Calendar **local**Identifier = 25 := 0b0011001
4 GranularityType **local**Identifier = 15 := 0b01111
5 Granularity **local**Identifier = 120 := 0b1111000

6 CalendarManagerVersion **global**Identifier = 422051840 :=
7 0b00011 00100101 0000000 00000 0000000
8 Calendar **global**Identifier = 422154240 :=
9 0b00011 00100101 0011001 00000 0000000
10 Granularity **global**Identifier = 422156280 :=
11 0b00011 00100101 0011001 01111 1111000

`GranularityAssociation<T extends Enum<T>` At some point or another, the `Granularity` objects deserialized from XML and stored in `Calendar` will likely have to be linked to the logical granularities that each calendric system has. These logic granularities are dependent on the implementation of the calendric system, and are equivalent to the granularities it supports. The class `GranularityAssociation` provides a generic way to create a two-way link between these granularities. It implements a two-way `Hashtable` for easy and performant lookups and reverse-lookups. These `Hashtable` instances are parameterized to `<T extends Enum<T>, Granularity>` (and its reverse for reverse lookup) to force the user to implement an enumeration data structure that represents a calendric system's granularities. This class is intended to encourage developers to use enumerations to define their calendric system granularities over `static final` fields or simply using primitive numbers, since enumerations offer a number of advantages:

- Enumerations increase code readability by eliminating magic number literals from source code. [18]
- Enumerations come with type safety: while it is possible to mix up which `static final int` field serves which purpose, enumerations can be bound tightly to their intended purpose [18].
- Enumerations come with value safety: If an `int` argument is desired, it is possible to pass any integer, even if they might not be assigned a valid context. Enumerations have a finite and usually easily overseen number of valid values, and may not take any other value than those defined, besides `null` [18].
- Enumerations are easily extensible and maintainable while still keeping an overview of the possible values.
- Enumerations are treated as objects in Java, and thus may encapsulate statically defined values to use in conjunction with the enum type object. This makes it possible to embed a rich set of context information within enum objects with a streamlined method access.

- Java provides an easy way to loop through all values of an enumeration through use of the `values()` function.

Adding new granularities to a `CalendarManager`

To add new granularities to an existing `CalendarManager`, the `loadCalendar(File)` method of the `CalendarRegistry` may be used. The `File` object should reference an XML file structured as show in Algorithm 6.3 and Algorithm 6.1.

When a `Calendar` object is deserialized, a couple of things have to be verified and initialized:

- Each `Granularity` has to be unique within its associated `Calendar`. This is verified by ensuring that only one `Granularity` object with a specific `GranularityIdentifier` may be contained within a `Calendar` at any one time.
- The `Calendar` must contain **exactly** one `Granularity` object with the `isBottomGranularity` flag set, and a separate `Granularity` with the `isTopGranularity` flag set.
- The `Calendar` needs to register itself with its referenced `CalendarManager`. To do this, a few things need to be ensured:
  - The `CalendarManager` with the specified version and manager identifiers must exist. The `CalendarRegistry` is queried to retrieve this `CalendarManager`, if possible.
  - The `Calendar` object instance must be unique within the retrieved `CalendarManager`. Calendars are uniquely identified within their `CalendarManager` by their `localCalendarIdentifier`.
- The `Calendar` sets the fully qualified global granularity identifiers for its instances of `Granularity`
- Finally, the deserialized data is reorganized into data structures that support easier iteration and access.

6.2 Use Case Examples

The implementational changes described in this chapter are intended to ease the realization of the following use case examples.

Adding or removing granularities

Addition or removal of granularities may be become necessary when modeling a calendar that should or should not be capable of expressing certain granularities. For example, imagine a Gregorian Calendar with the following granularities:

- Second
- Minute
- Hour
- Day
- Month
- Week
- Year

For some uses, this may be enough. For others, it could be required to add more precision by adding millisecond or nanosecond granularities. Another example of an additional, non-continuous granularity would be adding one that models weekends or work days. Restricting or extending a calendar can be done by utilizing the following methods:

- Granularities may be added by simply editing the existing XML file for a calendar. This editing changes the calendar, but does not result in an additional `Calendar` instance being added to its `CalendarManager`. Furthermore, all granularity identifiers (except for the edited ones) may remain unchanged.
- The existing XML file may be duplicated, and the new version augmented with the desired granularities. Using this alternative, both the old and new `Calendar` objects will be available for use with their `CalendarManager`. However, these `Calendar` objects (and their respective `Granularity` instances) will have to have disjoint global identifiers.

Adding a calendric system

Calendric systems are added to TimeBench by extending `CalendarManager` and implementing the calendar-specific logic. An example for this would be the addition of a `CalendarManager` that implements the Chinese calendar. Again, there are two ways to do this:

- If the desired calendric system is already implemented, but needs to have slightly different behavior, the implementing `CalendarManager` could simply be extended, and the desired methods overridden.
- If the calendric system has not yet been implemented, a new class that extends `CalendarManager` needs to be added.

In either case, the new `CalendarManager` classes need to be registered with the `CalendarRegistry`. This is done by adding them to the `CalendarRegistry` private method `initialize`.

6.3 Testing

The behavior described above is tested in three test case classes through unit testing.

IdentifierConverterTest

The `IdentifierConverter` is strictly focused on performing bit shifting operations to extract the specified fields of an identifier. Therefore, the unit tests implemented test this behavior to ensure that the returned values match the expected values. Furthermore, it also contains a test that guarantees the correct assembling of a global identifier out of separate identifier fields.

CalendarRegistryTest

The `CalendarRegistry` is the primary access point for developers to use the `calendar` package. The requirements to deserialized `Granularity` and `Calendar` objects mentioned above are checked implicitly upon loading an XML file via the `loadCalendar` method. A number of unit tests are implemented that load faulty as well as correct XML files and check whether the appropriate failure messages are thrown.

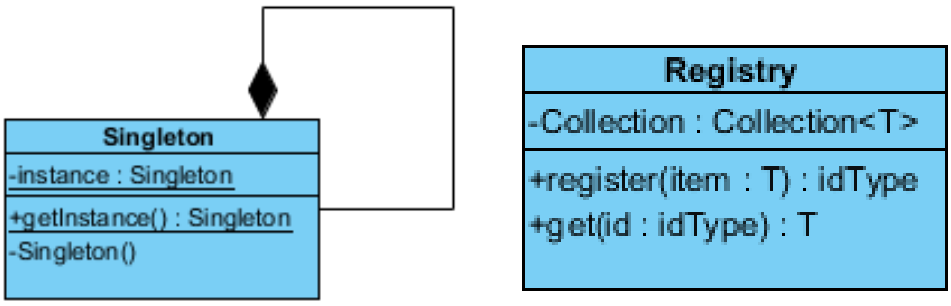
GregorianCalendarManagerTest

The `GregorianCalendarManager` is a reference implementation of `CalendarManager`. The unit test for `GregorianCalendarManager` focuses on ensuring that values for generated granules match the expected results.

6.4 Future improvements

The work done on TimeBench within the scope of this thesis has focused primarily on realizing XML deserialization of calendars and granularities, as well as refactoring existing classes to make them more easily maintainable, readable, and extensible. Suggested future work on this project may include:

- Implement more calendric systems to provide a more rich set of default implementations for users.
- Possibly extend `CalendarRegistry` to allow users to serialize in-memory calendars to XML.
- Implement dynamic run-time extension of calendars and calendric systems through Java reflection. To this end, a set of annotations could be developed to set guidelines for implementing classes.
- Formalize granularity lattice mappings in XML and code. Use annotations to specify conversion operations of irregular mappings.



(a) Singleton pattern class diagram (b) Registry pattern class diagram

Figure 6.2: Singleton and Registry pattern class diagrams

- Implement conversion logic that allows translation of dates between different calendric systems. At the least, this would require these calendric systems to utilize overlapping granularities.

6.5 Design Patterns

Registry Pattern

The Registry pattern is utilized in the `CalendarRegistry` and `CalendarManager` classes. Each of these class provides a public interface that allows other classes to register specific types of objects with the Registry class. Upon registration, an identifier is returned. To retrieve an object from the Registry class, the identifier has to be provided. A class diagram of the Registry pattern is provided in Figure 6.2b.

A number of classes of the `calendar` package have to register and manage their composite objects themselves, and thus serve as a Registry.

Singleton Pattern

The Singleton pattern is used in a number of classes of the `calendar` package, such as the `CalendarRegistry`. The Singleton pattern restricts instantiation of one class to one instance, which itself is contained as a static class member within the Singleton class. Access to this static instance is provided through a static getter method that returns the single instance. A class diagram of the Singleton pattern is provided in Figure 6.2a.

The use of the Singleton pattern offers itself as a useful tool to the `calendar` package because some classes should be instantiated exactly once. The `CalendarRegistry` should represent a global state that contains all registered `CalendarManager` instances.

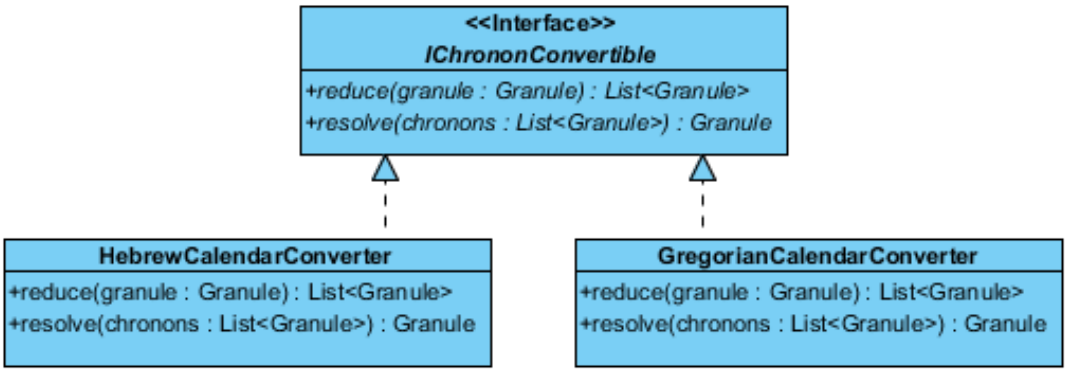


Figure 6.3: Strategy pattern implementation class diagram

Strategy Pattern

The use of the Strategy pattern is recommended for future development of the `calendar` package. The Strategy pattern decouples the interface of an algorithm from the specific implementation of the algorithm.

A number of calendric systems may be interconnected by providing mappings between granularities of the granularity lattices. Ideally, a system of granularity lattices will find a common granularity that it may reduce temporal data to in order to perform conversion of temporal data from one calendric system to another. For example, if a Gregorian calendar and Hebrew calendar were to share the same time domain and chronon, a granule of a certain granularity of the Gregorian calendar could be reduced to a set of chronons. This set of chronons could then be passed to the Hebrew calendar, and in turn be converted to granules.

Every `Calendar` object would contain an object implementing the Strategy pattern interface. A sample class diagram of the Strategy pattern implementation is show in Figure 6.3.

Data Transfer Object & Factory Pattern

At this time, the `Granularity` and `Calendar` classes mix both behavior and marshalling structure. It is recommended to implement the Data Transfer Object pattern to realize a better separation of concerns: One class should be used to represent XML structure - in this case, the Data Transfer Object with XML annotations. The other class should represent the behavorial object that is used throughout the program and contains business logic. Conversion between these two classes can either be centralized in two methods, or refactored into a Factory object.

CHAPTER 7

Summary & Conclusion

7.1 Summary

Chapter 1 gives a brief introduction into the problem domain of modeling calendars in information technology. It highlights the pervasiveness of time-oriented data in application scenarios. To tackle the problem of common calendar modeling, the calendar package, as part of the TimeBench framework, is introduced.

Chapter 2 is subdivided intro three sections: In section 2.1, a basic explanation of how time is measured is provided. This includes how common time elements are related to astronomical phaenomena, and how the standard definition of time intervals has evolved. In section 2.2, we introduce a number of basic concepts that are based on astronomical observations and are common to a large number of calendric systems. In section 2.3, a number of design aspects of temporal modeling is introduced, with special focus on the time granularity concept.

In Chapter 3, an overview of calendric systems in use today is given. A number of exemplary calendars are explained in detail, for different characterizations of calendric systems. For lunar calendars, which evolve around the movement of the Moon around the Earth, the Islamic and Roman calendars are described. The section describing solar calendars provides details about the Gregorian, Solar Hijri, and Julian calendars. Calendar systems that are intended to work in conjunction with both solar and lunar celestial movement are called lunisolar calendars; the Hebrew and Chinese calendar (amongst others) fall under this category. Finally, in the section named other calendars, the ISO 8601 date and time exchange standard is covered. Furthermore, the UNIX timestamp, which is used widely in information technology, is explained.

Chapter 4 is a State of the Art overview of current calendar implementations in Java. A number of libraries that provide the user with various ability to model calendric systems, as well as utilize them programmatically are analyzed:

- The Java 7 Date-Time API is shipped with the standard Java Development Kit and provides a reference implementation of the Gregorian calendar system.

- The Java 8 Date-Time API is an upcoming release that overhauls a number of issues of the Java 7 Date-Time API according to a Java Specification Request.
- Joda-Time is a project that was developed to address the same issues of the Java 7 Date-Time API. It is used widely and offers an implementation for eight different calendric systems.
- τZAMAN was developed by the University of Arizona and directly implements the granularity concepts mentioned above.
- Date4J is a lean API that is feature-centric on providing support of date-time related data for use with relational database management systems.

In Chapter 5, a documentation of the TimeBench framework is provided. The first section gives an introduction to what TimeBench is, what the overall intent behind the framework is, and who is developing it. A brief glimpse into the structure of the framework is given in the second section. Then the focus shifts to the `calendar` package, which is a core package of TimeBench that deals with modeling calendric systems. In this section, the implementation goals and package design are presented. Finally, the last section focuses on possible future improvements that can be done the `calendar` package.

7.2 Conclusion & Future Work

Time-oriented data plays a significant role in software & information engineering. Specifically with regards to visual analytics, a common approach is required to properly handle time-oriented data. Throughout the development of mankind, calendars have been used in various forms and methods to abstract the flow of time for human use. In order to use calendric systems in a constructive data visualization environment, it is necessary to engineer a common data model. To this end, the granularity model is being employed by the TimeBench framework. The `calendar` package of said framework is intended to allow modeling of calendric systems as easily extensible as possible, while keeping simple data structures in mind.

To the research questions stated in the introduction, the following conclusions are made:

1. Existing date-time frameworks for time modeling in Java were researched. Each of the frameworks analyzed defines its own calendar modeling characteristics. Of these frameworks, only the τZAMAN framework directly implements the granularity model outlined earlier. With the exception of τZAMAN, extension of implemented calendric systems is either only possible through direct implementation in Java, or not at all. τZAMAN allows deserialization of granularities, but the library as a whole is outdated, and development is discontinued.

2. The granularity model is a scientifically developed approach to dynamic calendar modeling. Therefore, it suggests itself as a basis for a library that allows extension of calendric systems. The TimeBench framework implements both the temporal calculus consisting of temporal primitives and temporal relations, as well as the granularity concepts. With

these implementations, a structured approach at calendar modeling and calendric system conversions are possible.

3. A number of software design patterns were used and are suggested for use in future development efforts. These patterns can be found in the corresponding TimeBench calendar subsection.

A set of improvement suggestions for future work are listed in section 6.4. To improve code quality and general maintainability of the code, a number of refactorizations might become advisable as well:

- Decouple calendric system behavior from structure: The calculation logic to create granules out of various parameters could be moved to one or more separate classes altogether. Realistically, implementations of calendar logic right now may contain easily upwards of 1,000 lines of code.
- Similarly, logic to allow XML serialization/deserialization could be moved to separate classes. To this end, the value object pattern could be implemented. Adding (a set of) classes to perform conversion work between these value objects and the behavioral objects would streamline the usage of these classes and further improve the separation of concerns.
- To achieve further modularity of calculation operations, the strategy pattern could be employed to encapsulate operations within their own separate object.

Since the TimeBench framework aims to implement conversion between different calendric systems at some point, it is especially advisable to strictly separate behavior from structure in the future. To do this, a strongly typed interface structure is necessary to allow operations to remain interchangeable with one another.

Bibliography

[1] Orion 8. http://commons.wikimedia.org/wiki/File:Moon_phases_00.jpg, 2010. Accessed: 2013-10-01, licensed under Creative Commons Attribution-Share Alike 3.0 Unported license.

[2] Wolfgang Aigner, Silvia Miksch, Heidrun Schumann, and Christian Tominski. *Visualization of time-oriented data*. Springer, 2011.

[3] James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[4] Internet Assigned Numbers Authority. Time Zone Database. http://www.iana.org/time-zones/repository/tz-link.html, Accessed: 2013-11-21.

[5] Claudio Bettini, Curtis E Dyreson, William S Evans, Richard T Snodgrass, and X Sean Wang. A glossary of time granularity concepts. In *Temporal databases: research and practice*, pages 406–413. Springer, 1998.

[6] Claudio Bettini, Sushil Jajodia, and Sean Wang. *Time Granularities in Databases, Data Mining, and Temporal Reasoning*. Springer, 2000.

[7] Microsoft Corporation. Microsoft Time Zone Index. http://msdn.microsoft.com/en-us/library/ms912391(v=winembedded.11).aspx, Accessed: 2013-11-26.

[8] Oracle Corporation. Java 7 API Reference. http://docs.oracle.com/javase/7/docs/api/, Accessed: 2013-11-15.

[9] Oracle Corporation. Java 8 API Reference. http://download.java.net/jdk8/docs/api/, Accessed: 2013-11-25.

[10] Oracle Corporation. OpenJDK ThreeTen Project Website. http://openjdk.java.net/projects/threeten/, Accessed: 2013-12-04.

[11] Nachum Dershowitz and Edward M Reingold. *Calendrical Calculations*. Cambridge University Press, 2008.

[12] Oxford Advanced American Dictionary. http://oaadonline.oxfordlearnersdictionaries.com/media/oaad8/fullsize/e/ear/earth/earth_seasons.jpg. Accessed: 2013-10-01.

[13] Curtis E. Dyreson. τZAMAN Project website. http://www.cs.arizona.edu/projects/tau/tauZaman/index.htm, Accessed: 2013-12-03.

[14] Jeffrey Heer, Stuart K Card, and James A Landay. Prefuse: a toolkit for interactive information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 421–430. ACM, 2005.

[15] Unicode Inc. Common locale data repository. http://cldr.unicode.org/, Accessed: 2013-11-21.

[16] Christian S Jensen, Curtis E Dyreson, Michael Böhlen, James Clifford, Ramez Elmasri, Shashi K Gadia, Fabio Grandi, Pat Hayes, Sushil Jajodia, Wolfgang Käfer, et al. The consensus glossary of temporal database concepts—february 1998 version. In *Temporal Databases: Research and Practice*, pages 367–405. Springer, 1998.

[17] William Markowitz, R Glenn Hall, L Essen, and JVL Parry. Frequency of cesium in terms of ephemeris time. *Physical Review Letters*, 1958.

[18] Steve McConnell. *Code complete*. O'Reilly Media, Inc., 2004.

[19] Vienna University of Technology Institute of Software Technology and Interactive Systems. CVAST project page. http://www.cvast.tuwien.ac.at/cvast, Accessed: 2014-01-20.

[20] BIPM Bureau International Des Poids Et Mesures (International Bureau of Weights and Measures). http://www.bipm.org/en/si/si_brochure/chapter2/2-1/second.html, 1967/68/97. Accessed: 2013-10-07.

[21] BIPM Bureau International Des Poids Et Mesures (International Bureau of Weights and Measures). http://www.bipm.org/en/scientific/tai/tai.html, 1977. Accessed: 2013-11-12.

[22] BIPM Bureau International Des Poids Et Mesures (International Bureau of Weights and Measures). The international system of units (si), 2006. http://www.bipm.org/utils/common/pdf/si_brochure_8_en.pdf, Accessed: 2013-11-07.

[23] John O'Hanley. Date4J Project Website. http://www.date4j.net/, Accessed: 2013-12-04.

[24] Oracle. Java Date-Time API Trail. http://docs.oracle.com/javase/tutorial/datetime/TOC.html, Accessed: 2013-11-21.

[25] Joda Project. Joda-Time - Java Date and Time API. http://www.joda.org/joda-time/, Accessed: 2013-12-02.

[26] Edward Graham Richards. *Mapping Time: The Calendar and its History*. Oxford University Press, 1999.

[27] Alexander Rind, Tim Lammarsch, Wolfgang Aigner, Bilal Alsallakh, and Silvia Miksch. Timebench: A data model and software library for visual analytics of time-oriented data. *Visualization and Computer Graphics, IEEE Transactions on*, 19(12):2247–2256, 2013.

[28] Sch. `http://commons.wikimedia.org/wiki/File:CompareTropicalYears.png`, 2006. Accessed: 2013-10-02, language of graph description changed from German to English, licensed under Creative Commons Attribution-Share Alike 3.0 Unported license.

[29] P Kenneth Seidelmann. *Explanatory Supplement to the Astronomical Almanac: A Revision to the Explanatory Supplement to the Astronomical Ephemeris and the American Ephemeris and Nautical Almanac*. University Science Books, 2005.

[30] Bedirhan Urgun, Curtis E Dyreson, Richard T Snodgrass, Jessica K Miller, Nick Kline, Michael D Soo, and Christian S Jensen. Integrating multiple calendars using τ zaman. *Software: Practice and Experience*, 37(3):267–308, 2007.